Experimental Study of Minimum Cut Algorithms

Chandra S. Chekuri*
Computer Science Department
Stanford University
Stanford, CA 94305
chekuri@theory.stanford.edu

Andrew V. Goldberg
NEC Research Institute
Princeton, NJ 08540
avg@research.nj.nec.com

David R. Karger†
Laboratory for Computer Science
MIT
Cambridge, MA 02139
karger@theory.lcs.mit.edu

Matthew S. Levine‡
Laboratory for Computer Science
MIT
Cambridge, MA 02139
mslevine@theory.lcs.mit.edu

Cliff Stein§
Department of Computer Science
Dartmouth College
Hanover, NH, 03755
cliff@cs.dartmouth.edu.

October 1996

Abstract

Recently, several new algorithms have been developed for the minimum cut problem. These algorithms are very different from the earlier ones and from each other and substantially improve worst-case time bounds for the problem. We conduct experimental evaluation the relative performance of these algorithms. In the process, we develop heuristics and data structures that substantially improve practical performance of the algorithms. We also develop problem families for testing minimum cut algorithms. Our work leads to a better understanding of practical performance of the minimum cut algorithms and produces very efficient codes for the problem.

*Supported by NSF Award CCR-9357849, with matching funds from IBM, Mitsubishi, Schlumberger Foundation, Shell Foundation, and Xerox Corporation.

†Research partly supported by ARPA contract N00014-95-1-1246.

‡Research partly supported by a grant from the World Wide Web Consortium. Some of this work was done while visiting the second author at NEC.

§Research partly supported by NSF Award CCR-9308701, a Walter Burke Research Initiation Award and a Dartmouth College Research Initiation Award. Some of this work was done while visiting the second author at NEC, and while visiting Stanford University.

Contents

1 Introduction

The minimum cut problem is the problem of partitioning the vertices of an n-node, m-edge weighted undirected graph into two sets so that the total weight of the set of edges with endpoints in different sets is minimized. This problem has many applications, including network reliability theory [24, 37], information retrieval [4], compilers for parallel languages [5], and as a subroutine in cutting-plane algorithms for the Traveling Salesman problem (TSP) [3].

The problem of finding a minimum capacity cut between two specified vertices, s and t, is called the minimum s-t cut problem, and is closely related to the minimum cut problem. The classical Gomory-Hu algorithm [20] solves the minimum cut problem using $n-1$ minimum s-t cut computations. The fastest current algorithms for the s-t cut problem [1, 6, 7, 18, 28] use flow techniques, in particular the push-relabel method [18], and run in $\omega(nm)$ time. For the minimum cut problem, Hao and Orlin [22, 23] have given an algorithm (HO), based on the push-relabel method, that shows how to perform all $n-1$ minimum s-t cuts in time asymptotically equal to that needed to perform one s-t minimum cut computation. This algorithm runs in $O(nm\log(n^2/m))$ time.

Several new algorithms discovered recently are theoretically more efficient – time bounds for these algorithms are competitive with or better than the best time bounds for the minimum s-t cut problem. The algorithm of Nagamochi and Ibaraki [32] (NI) runs in $O(n(m+n\log n))$ time. The algorithm of Karger and Stein [26] (KS) runs in $O(n^2\log^3 n)$ expected time. Two closely related algorithms of Karger [25] (K) run in $O(m\log^3 n)$ and $O(n^2\log n)$ expected time. These algorithms are based on new techniques which do not use flows. These algorithms do not extend to directed graphs, while the flow based algorithms, including HO, do.

Theoretical considerations suggest that some of these new algorithms should be practical, and an experimental study of Nagamochi et al. [33] confirms this for NI. The question of practical performance of the other new algorithms, however, has not been addressed.

In this paper we study the practical performance of recent minimum cut algorithms. The main part of our study is the design of data structures and heuristics for efficient implementations of these algorithms. This is an iterative process – the implementation performance is measured and the data is used to improve the implementations. Our goal is to obtain efficient implementations of the new algorithms and to compare them against each other and against a good previous code. In order to perform meaningful comparisons, we develop problem generators and test families for evaluating and comparing performance of the minimum cut codes. We also run and analyze our algorithms on data that arises during the TSP algorithm of Applegate and Cook [3]. Our problem families are carefully selected and proved very useful for comparing and tuning minimum cut codes.

Our codes use heuristics that, on some problems, significantly reduce the number of operations performed by the underlying algorithms. On other problems, the heuristics cost time but do not save enough work to pay for themselves. In order to avoid a situation in which the time spent on heuristics dominates the running time, we adopt the following rule: Either a heuristic takes near-linear time, or its work can be amortized by the work of the underlying algorithm. We try to take maximum advantage of the heuristics while making sure that the heuristics do not significantly increase the running time when they fail.

The most efficient implementation of the Gomory-Hu algorithm that we are aware of is due to Padberg and Rinaldi [35]. In order to reduce the number of maximum flow computations needed, which is $n-1$ for the Gomory-Hu algorithm, they developed a set of heuristics which contract certain edges during the computation and, if successful, reduce the number of maximum flow computations. These contractions, which we call the *PR heuristics*, apply in the context of other algorithms and often lead to a big improvement in performance.

Nagamochi et al. [33] describe an efficient implementation of NI that uses the PR heuristics. The data of [33] suggests that the `hybrid` code of Nagamochi et al. is more efficient than the Padberg-Rinaldi code. The former code seems to be the fastest minimum cut code described in a paper published prior to our work.

A major contribution of this paper are new heuristics for improving practical performance of the algorithms. We propose a common preprocessing, based on PR heuristic, which takes $O(m \log n)$ time and usually is nearly as helpful as the $\Omega(nm)$ version of Padberg and Rinaldi. We introduce algorithm-specific ways of using PR heuristics during execution of main subroutines of the algorithms. We also develop several new heuristics that significantly improve performance of our implementations.

Implementations using graph contraction are usually difficult to code (see *e.g.* [21]) and may be inefficient. Our fastest implementations of all the algorithms we study use contraction. Although indeed difficult to code, these implementations are efficient because of the graph data structures we use.

We now briefly detail the highlights of the implementations of our four codes. Greater details will be given in subsequent sections.

Our implementation of `ni` builds on `hybrid` [33]. However, we use different strategy for applying the PR heuristics and develop a new technique for graph contraction. As a result, our code is always competitive with `hybrid`, and sometimes outperforms it by a wide margin.

Implementations of the push-relabel method for the maximum flow problem have been well-studied, *e.g.* [2, 10, 13, 14, 34]. A maximum flow code of Cherkassky and Goldberg [10] was the starting point of our implementation, `ho`. The implementation uses the heuristics global update and gap relabeling heuristics that are used in the maximum flow code. In addition, we use the graph contraction data structures mentioned above, the PR heuristics, and several new heuristics. Efficient use of these heuristics often requires a good understanding of the underlying algorithm and data structures.

The Karger-Stein and Karger's algorithms are randomized. For these algorithms, we needed to develop somewhat different strategies for random edge selection than those that appeared in the original papers [25, 26]. These codes also benefit from the PR heuristics as well as new heuristics.

We make significant progress in understanding practical performance of minimum cut algorithms and in establishing testing standards for future codes. Our study shows that no single algorithm dominates the others. Overall, `ho` is the best code. It is the fastest on many problem instances, sometimes by a significant margin and never looses by a very significant margin. The second best code is `ni`.

Our implementations of `ks` and `k` do not perform as well as the best of `ho` and `ni`. We observe, however, that for these randomized algorithms the number of trials before finding the minimum cut can be much lower than the theoretical bound. In fact, our implementation of `k` uses a much lower sampling probability than the theory suggests while finding the correct solution to every problem instance. We use a theoretical bound for `ks`, and our data suggests that this bound cannot be significantly reduced without missing a minimum cut on some problem instances.

The `ks` and `k` algorithms have some advantages. With high probability, they find all minimum cuts, which may be useful in some applications, especially the TSP application. On some problem classes, they may compute correct answers while performing fewer trials. Thus, on these classes, they can run faster.

The push-relabel method has been extensively studied, and our implementations of HO take advantage what has been learned in the maximum flow context. Our implementation of NI takes advantage of performance-improving ideas of [33]. Our implementations of KS and K, however, were developed from scratch. Our study shows how important data structures and heuristics are for the minimum cut algorithm performance. New ideas may improve performance. This is more likely for KS and K, which have been studied less.

This paper is organized as follows. Section 2 gives definitions and background, including a review of data structures for graph contraction. Section 3 describes the PR heuristic, including our implementation of it common to all our codes. Sections 4, 5, 6, 7 reviews the four algorithms we study and describe our implementations; each section is devoted to one of the algorithms. Section 8 describes problem families used in our study and Section 9 describes the experimental setup. Experimental results are discussed in Section 10. We give our conclusions in Section 11. The last section contains data tables and plots.

2 Background

In this section, we define and describe some of the basic techniques and concepts that are used in our algorithms. In Section 2.1 we give the basic definitions of graphs and cuts. In Section 2.2 we discuss edge contraction and in Section 2.3 we discuss two different data structures for representing graphs with contractions. In Section 2.4 we give the basic network flow definitions. In Section 2.5 we discuss the first efficient minimum cut algorithm, the Gomroy-Hu algorithm, and in Section 2.6 we discuss the difference between discovering a cut, verifying a cut and saving a cut.

2.1 Graphs and Cuts

Let $G = (V, E, c)$ be an undirected graph with vertex set V, edge set E and non-negative real edge capacities $c : E \rightarrow \mathbf{R}^+$. Let $n = |V|$ and $m = |E|$. We will denote an undirected edge with endpoints v and w by $\{v, w\}$, but use $c(v, w)$ as shorthand for $c(\{v, w\})$. A *cut* is a partition of the vertices into two nonempty sets A and $\overline{A}$. The *capacity* of a cut $c(A, \overline{A})$ is defined by

$$c(A, \overline{A}) = \sum_{u \in A, v \in \overline{A}, \{u,v\} \in E} c(u, v). \tag{1}$$

We will sometimes unambiguously refer to a cut just by naming one side, and use the shorthand $c(A) = c(A, \overline{A})$. Also, if $A = \{v\}$, we may use $c(v)$ instead of $c(A)$. The edges included in the sum in (1) will be referred to as edges *in the cut* or edges that *cross the cut.* The *minimum cut* is the cut A that minimizes $c(A)$.

We will use $\lambda(G)$ to denote the value of the minimum cut of G, and $\lambda_{v,w}(G)$ to denote the value of the minimum v-w cut, that is, the minimum cut of G in which v and w are required to be on opposite sides of the cut.

Although terms *node* and *vertex* are often used interchangeably, we make a distinction. We use the term node for the base set of a graph potentially created by a contraction operation, and the term vertex otherwise. Nodes correspond to sets of vertices of the input graph.

2.2 Edge Contraction

Given a graph G and edge $\{v, w\} \in E$, we define $G/\{v, w\}$, the contraction of edge $\{v, w\}$, by deleting w and replacing each edge of the form $\{w, x\}$ by an edge $\{v, x\}$. If this process creates parallel edges, we merge them and add the capacities. We also delete any self-loops.

Almost all of our algorithms use edge contraction, either as a fundamental step (`ks` and `ni`), or as part of the `PR` heuristics. The basic idea is that if we can identify an edge that is not in a minimum cut, then we can safely contract that edge (in `ks` we may actually contract an edge that is in the minimum cut). We give one basic lemma to that effect here, we will give others later in the paper.

Lemma 2.1 *Given a network G and an edge $\{v, w\}$,*

$$\lambda(G/\{v, w\}) = \min\{\lambda(G), \lambda_{v,w}(G)\}$$

Proof. Fix a minimum cut A. If v and w are on the same side of the minimum cut, then $\lambda(G/\{v, w\}) = \lambda(G)$ since the minimum cut has not been contracted and no new cuts were created. If v and w are on opposite sides of the minimum cut A, then $\lambda(G/\{v, w\}) = \lambda_{v,w}(G)$ by definition. ■

2.3 Data Structures for Graph Contraction

In this section we describe the graph data structure used to deal with edge contractions efficiently.

We represent an undirected graph as a symmetric directed graph using the adjacency list representation. Each vertex has a doubly linked list of edges adjacent to it and pointers to the

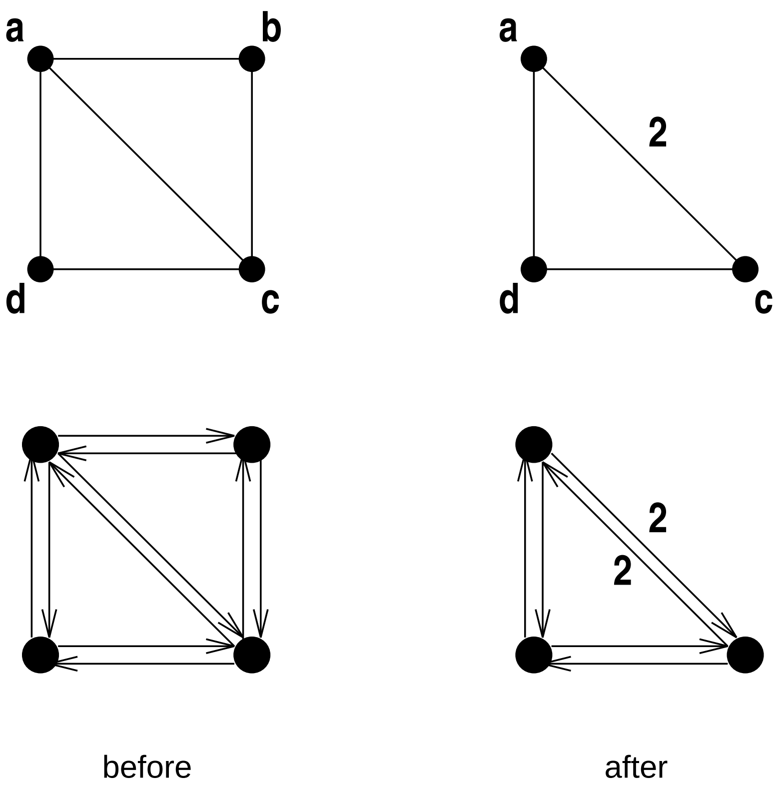

Figure 1: Compact contraction example: Contracting edge $\{a, b\}$. Graphs in the top row are represented as shown in the bottom row. If an arc capacity is equal to one, the capacity is not shown.

beginning and the end of the list. An edge $\{u, v\}$ is represented by two arcs, (u, v) and (v, u). These arcs have pointers to each other. An arc (u, v) appears on the adjacency list of u and has a pointer to v.

Suppose we contract an edge $\{u, v\}$. One way to implement the contraction is to do *compact contraction* as follows.

1. For each (v, w) on the adjacency list of v, replace the reverse arc (w, v) by (w, u).

2. Append the arc list of v to the arc list of u.

3. Delete v from $V(G)$.

4. Delete self-loops adjacent to u.

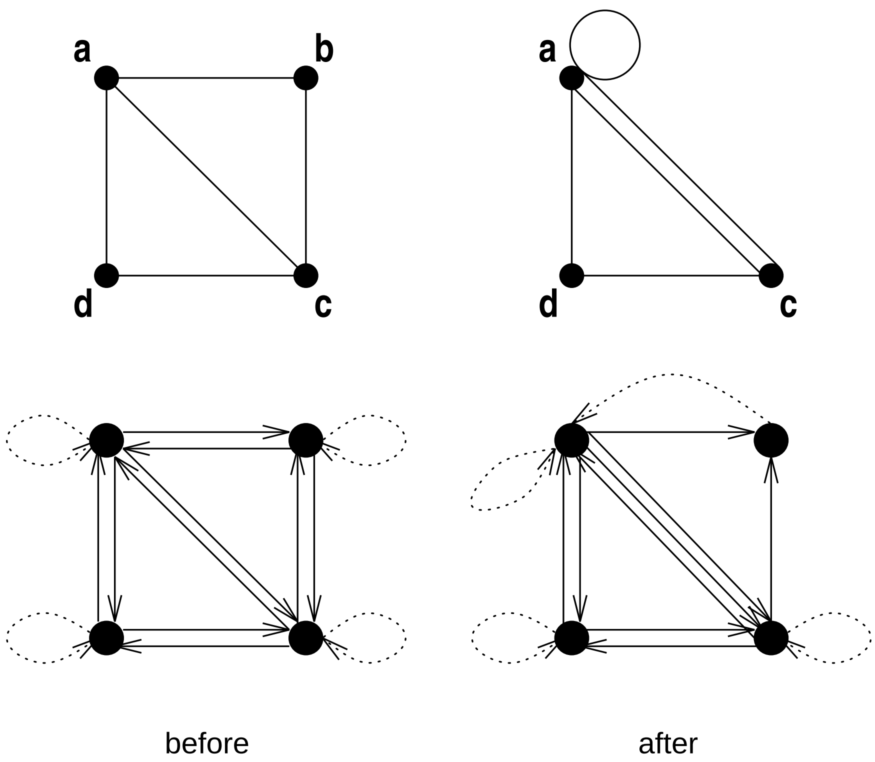

Figure 2: Set-union contraction example: Contracting edge $\{a, b\}$. Graphs in the top row are represented as shown in the bottom row. If an arc capacity is equal to one, the capacity is not shown.

5. Merge parallel edges adjacent to u.

Figure 1 gives an example of this implementation of edge contraction. A careful implementation of compact contraction of $\{u, v\}$ takes time proportional to the sum of degrees of u and v before the contraction.

An alternative way to contract an edge $\{u, v\}$ is to use *set-union contraction*. In this implementation, each node in the current graph corresponds to a set of vertices of the input graph. The sets are represented using the disjoint-set union data structure implemented by disjoint-set forests with path compression; see *e.g.* [11]. Each set has a distinct representative. Each vertex has a pointer towards (but not necessarily directly to) the representative of its set. Representatives point to themselves. We assume that the reader is familiar with the set-union data structure.

Suppose u and v are currently different nodes. An edge $\{u, v\}$ is contracted as follows.

1. Form a union of the vertex sets corresponding to u and v.

2. Append the adjacency list of v to that of u.

A set-union contraction operation takes constant time.

The advantage of the set-union contraction is in efficiency of the contraction operation. Its disadvantages come from the parallel arcs and self-loops which remain in the graph and from the increased cost of finding the head node of an arc.

Our algorithms will make use of both of these data structures at different times. We will discuss this in subsequent sections.

2.4 Network Flows

Although we study an undirected version of the minimum cut problem in this paper, flows are more natural in directed graphs. We transform an undirected graph into a directed graph in a standard way: Replace each edge $\{v, w\}$ of capacity $c(v, w) = x$ by two arcs (v, w) and (w, v) of capacity x each. For any pair of vertices s, t, a cut is a minimum s-t cut in the original graph if and only if it is a minimum s-t cut in the transformed graph.

Let $G = (V, E, c)$ be a directed graph with two distinguished vertices s (source) and t (sink). A *flow* is a function $f : E^d \to \mathbf{R}$ satisfying

$$f(v, w) \leq c(v, w)\,, \quad \forall (v, w) \in E^d \tag{2}$$

$$f(v, w) = -f(w, v)\,, \quad \forall (v, w) \in E^d \tag{3}$$

$$\sum_{v \in V} f(v, w) = 0\,, \quad \forall w \in V - \{s, t\}. \tag{4}$$

The *value* of a flow is the net flow into the sink, i.e.,

$$|f| = \sum_{v \in V} f(v, t).$$

The *maximum flow problem* is to determine a flow f for which $|f|$ is maximum. The well-known maxflow-mincut theorem [16, 15] states that the value of the maximum s-t flow is equal to the value of the minimum s-t cut, *i.e.,* $|f| = \lambda_{s,t}(G)$. An s-t maximum flow algorithm can thus be used to find an s-t minimum cut, and minimizing over all $\binom{n}{2}$ possible choices of s and t yields a minimum cut.

2.5 Gomory-Hu Algorithm

In 1961, Gomory and Hu [20] showed that $\lambda_{s,t}(G)$ for all $\binom{n}{2}$ pairs of s and t could actually be computed using only $n-1$ maximum flow computations. This immediately yields an algorithm for computing minimum cuts using only $O(n)$ maximum flow computations. Note that Gomory and Hu are solving a problem that is more general that the minimum cut problem, and we can see more directly that $O(n)$ maximum flow computations suffice to compute a minimum cut. The underlying idea, which is stated in Lemma 2.1, is important for understanding other minimum cut algorithms. Consider a pair of distinct vertices s and t and a minimum s-t cut A. Then either $\lambda_{s,t}(G) = \lambda(G)$ or s and t are on the same side of every minimum cut.

A simple version of the Gomory-Hu algorithm is based on this idea and works as follows. Pick a source s arbitrarily. Then, until s contains all the vertices we repeat the following three steps. First, we pick a sink t distinct from s and compute a minimum s-t cut. Second, if the value of the cut is smaller than smallest previous cut, we save the cut and its value as the smallest so far. Third, we contract s and t into s. When the loop terminates, we return the smallest s-t cut found.

The above algorithm computes a minimum cut in $n-1$ minimum s-t cut computations.

2.6 Discovery Times

Discovery time is the time from the beginning of the computation to finding the minimum cut. Note that the discovery time may be much less then the total running time of an algorithm. Discovery times are useful for several reasons.

One reason concerns saving candidate cuts. Like the Gomory-Hu algorithm, all algorithms we study produce a sequence of cuts with smaller and smaller values. Although with all our algorithms, on all our data, we observed that the number of candidate cuts tends to be small, in principle it is possible to have $n-1$ candidate cuts. In this case, it is possible that the work in saving the cuts will dominate the running time.

One way to get around this issue is to save the cut values only and compute the minimum cut value λ^*. If a minimum cut is desired, we repeat the computation and output the first cut of value λ^* found. The discovery time of the first computation is the total time taken by the second computation (ignoring the time to output the cut). All our codes do not save cuts and measure the discovery time as well as the total time.

Another use of discovery time is to estimate real-life performance of randomized algorithms.

Theory gives an upper bound on the number of "trials" required to achieve a desired success probability. These bounds, however, can be very pessimistic. If discovery time of a randomized algorithm is always a tiny fraction of the total time, one may be justified in making a smaller number of "trials."

3 Padberg-Rinaldi Heuristics

In 1990, Padberg and Rinaldi [35] introduced heuristic improvements to the Gomory-Hu algorithm. Recall that the Gomory-Hu algorithm performs a series of maximum flow computations; each maximum flow computation identifies one edge which can be contracted. The improvements of Padberg and Rinaldi are based on additional tests that allow contracting certain nodes without performing any maximum flow computations. In the worst case, the algorithm still performs $n-1$ maximum flow computations. On some problem instances, however, the heuristics allow many nodes to be contracted and reduce the running time significantly.

We found the heuristics useful in speeding up all of our codes. The codes used the heuristics in two different places. First, all codes apply the tests at the beginning of the algorithm. We call this PR-preprocessing. All our codes use the same PR preprocessing routine, which consists of repeatedly executing passes. Each code, however, uses a different rule to decide how many passes to run. Second, some of the codes apply the PR heuristics internal to the algorithm. Due to the interaction with the algorithms, internal PR heuristics are different for different codes.

The Padberg-Rinaldi heuristics are based on *PR tests*. In their original paper, Padberg and Rinaldi give a very general class of tests. Some of these tests are, in fact, quite time-consuming and would dominate the time taken by any of our codes. However, four of the tests are inexpensive enough to be incorporated into our code and we describe them below.

First we describe two very simple tests:

Lemma 3.1 ([35]) *Let $\hat{\lambda}$ be an upper bound on $\lambda(G)$. If $v, w \in V$ satisfy either one of the following conditions:*

`(PR1)` $c(v, w) \geq \hat{\lambda}$,

`(PR2)` $c(v) \leq 2c(v, w)$ *or* $c(w) \leq 2c(v, w)$,

then one of the following conditions must hold:

1. $\{v\}$ *is a minimum cut,*

2. $\{w\}$ *is a minimum cut,*

3. *the edge* $\{v, w\}$ *forms the unique minimum cut,*

4. $\exists$ *a minimum cut* $(A, \overline{A})$ *with* $v \in A$ *and* $w \in A$.

The PR1 test says that if we have an upper bound on the minimum cut $\hat{\lambda}$ and we find an edge $\{v, w\}$ with $c(v, w) \geq \hat{\lambda}$, then either edge $\{v, w\}$ is a one-edge minimum cut, or it is safe to contract $\{v, w\}$, as any cut containing edge $\{v, w\}$ must have total capacity at least $\hat{\lambda}$.

To understand the PR2 test, consider a cut $(A, \overline{A})$ with $v \in A$ and $w \in \overline{A}$, and assume that $c(w) \leq 2c(v, w)$ (the other case is symmetric). Also assume that v and w are not singleton cuts. Let N denote the set of neighbors of w. Then $c(w) \leq 2c(v, w)$ implies that

$$\sum_{x \in N - v} c(w, x) \leq c(v, w). \tag{5}$$

Currently the edges incident to w that are in the cut $(A, \overline{A})$ include $\{v, w\}$ and hence have total capacity at least $c(v, w)$. If we move w from $\overline{A}$ to A, then $\{v, w\}$ will not be in the cut and so the edges incidents to w that are in the cut will have total capacity at most $\sum_{x \in N-v} c(w, x)$. By (5) this is at most $c(v, w)$ and thus $c(A \cup w) \leq c(A)$. Therefore, we know that the graph must have a minimum cut in which v and w are on the same side of the cut, and it is safe to contract v and w.

In general, when a PR2 test passes, it might be the case that v or w is actually the minimum cut. Thus we always check $c(v)$ and $c(w)$ before contracting $\{v, w\}$. If either is less than $\hat{\lambda}$, then we update $\hat{\lambda}$ appropriately.

In all our algorithms, we explicitly maintain node capacities $c(v)$, edge capacities $c(v, w)$, and $\hat{\lambda}$, an upper bound on the minimum cut, which corresponds to the smallest capacity cut we have encountered so far. Thus tests PR1 and PR2 can be performed in $O(1)$ time per edge. We always apply the two tests together.

A second set of tests are more expensive but still useful in our implementations.

Lemma 3.2 ([35]) *Let* $\hat{\lambda}$ *be an upper bound on* $\lambda(G)$. *If* $v, w \in V$ *satisfy either one of the following conditions:*

(PR3) $\exists u$ *such that* $c(v) \leq 2\{c(v, w) + c(v, u)\}$ *and* $c(w) \leq 2\{c(v, w) + c(w, u)\}$, (PR4) $c(v,w)+\sum_u \min(c(v,u),c(w,u)) \geq \hat{\lambda}$

then one of the following conditions must hold:

1. *$\{v\}$ is a minimum cut,*

2. *$\{w\}$ is a minimum cut,*

3. *there exists a minimum cut $(A,\overline{A})$ with $v \in A$ and $w \in A$.*

For a fixed pair v, w, PR3 and PR4 can be checked in $O(n)$ time. We always apply the two tests together, and compute the sum for PR4 while doing PR3.

Condition PR3 is similar to PR2. Condition PR4 says that if the flow between v and w along one- and two-edge paths is at least $\hat{\lambda}$ and we know a cut achieving $\hat{\lambda}$, v and w can be contracted.

3.1 Previous PR Heuristic Implementations

Padberg and Rinaldi suggested two ways of applying PR tests in the context of the Gomory-Hu algorithm. The first way, which we call *exhaustive PR heuristic*, is described in [35]. The second way, which we call *source PR heuristic*, is described in [33] and is similar to another heuristic of Padberg and Rinaldi.

The exhaustive PR heuristic applies all four tests PR1 – PR4 until no edge can be contracted using these tests. The exhaustive heuristic takes $O(nm)$ time [35]. The heuristic was developed in the context of the Gomory-Hu algorithm, which had a worst case running time of $\omega(n^2m)$. Since, we study more efficient algorithms, this heuristic is too expensive for our purposes.

The source PR heuristic works as follows. Let x be the node created by the last contraction of the algorithm. (In the Gomory-Hu algorithm, x is created by contracting the source and the sink.) The heuristic examines edges adjacent to x starting from the beginning of the edge list, and applies tests PR1 – PR4 to the first "large capacity" edge $\{x,y\}$ found. If a test succeeds, x and y are contracted using compact contraction, and edges of the newly formed node are scanned again from the beginning of the edge list. If the test fails, the heuristic terminates. In [33], an edge $\{x,y\}$ if is considered to have large capacity $c(x,y) \geq 0.7c(x)/n'$, where n' is the current number of nodes.

The above source PR heuristic is from [33]. The `hybrid` code we obtained from the authors, however, implements a somewhat different version of this heuristic.

3.2 PR Passes

We introduce the idea of *PR passes*. All our codes that use the PR heuristics apply these passes as preprocessing; `ni` also applies the passes internally. Each pass takes linear time. *Pass-1-2* combines tests `PR1` and `PR2` and *pass-3-4* combines tests `PR3` and `PR4`.

Intuitively, pass-1-2 applies the two tests to every edge and every node of the graph. This is not precise, however, because edge contractions change the graph. Our implementation has the invariant that if an edge $\{u, v\}$ was not contracted, and did not become a self-loop (and deleted) during the pass, then the PR tests failed on $\{u, v\}$ at some point during a pass. We also ensure that the work done by the pass is either amortized against other work done by the algorithm, or the pass decreases the number of nodes in the graph by a constant fraction.

To facilitate implementation, we maintain the capacity for every node in the current graph. When node capacities are computed initially or updated after a contraction, we check if the corresponding cut is smaller than the best cut seen so far. We apply the PR heuristic to the compact graph data structure, and use compact contraction on edges that pass the PR tests. The implementation is nontrivial because we want to assure linear running time of a PR pass.

We assume that nodes of the current graph are linearly ordered (for example, by their ids). During a pass, we scan the current nodes in order. When scanning a node u, we go through its adjacency list. Let (u, v) be the current arc being examined. If $u > v$, we proceed to the next arc on the list. Otherwise we apply the PR tests to $\{u, v\}$. The tests take constant time because we maintain the node capacities. If one of the tests succeeds, we contract $\{u, v\}$ using compact contraction and proceed to the next arc on the list. (Note that in this case the adjacency list of v is appended to the list of u.) One can implement the scan so that the total time to process a node is proportional to the number of arcs adjacent to the nodes contracted during the scan.

Consider a node u before a pass. If we contract an edge adjacent to u, we delete u from the graph and never scan it. If we scan u, the nodes scanned after that are greater than u, so u will not be contracted from the end of the scan of u to the end of the pass. Thus a PR pass takes linear time. Suppose an edge $\{u, v\}$ was not contracted, and did not become a self-loop and deleted during the pass. Assume, without loss of generality, that $u < v$. Then the edge $\{u, v\}$ was examined during the scan of u or the scan of a node that u was contracted into, and the edge failed the PR tests at that time.

Remark. It is possible that an edge $\{u, v\}$ passes a PR test after a pass. For example, after u

has been scanned, v may be contracted and the capacity of the resulting node may be smaller than the capacity of v during the previous PR tests of $\{u, v\}$.

Next we describe pass-3-4. Intuitively, such a pass applies `PR3` and `PR4` to as many nodes as possible while scanning each node at most twice. To maintain this invariant, we sometimes apply a weaker version of `PR4` by taking the sum over a subset of neighbors of v. Pass-3-4 work as follows.

First all nodes are marked as unscanned. Then we pick an unscanned node v and apply tests 3 and 4 to its adjacent edges as discussed below. In the process, we mark v and neighbors of v (including new neighbors created by contractions) as scanned. The pass terminates when all nodes are scanned.

The choice of the next unscanned node is important. We experimented with a number of strategies: highest degree, highest capacity, highest average capacity, least recently scanned, etc. We use the following strategy: assign an *age* to every node. The age set to zero initially and is reset to zero every time the node is picked during a PR pass. The age of v is increased by one for every node contracted into v. The age is also increased by two every time a node is scanned as a neighbor of a picked node in pass-3-4 (and the tests are not applied to the node during this pass). This strategy tends to pick the least recently scanned nodes and the nodes that have been successful in contracting with other nodes.

After we pick an unscanned node v, we scan it as follows. First we mark all neighbors w of v as scanned. Then for every neighbor w of v, if w is unscanned, we mark it scanned, examine its neighbors x, and apply `PR3` to $\{v, w\}$ if x is a neighbor of v. While scanning the neighbors of w, we also compute the sum needed for the `PR4` test. When the scanning of w's neighbors is complete, we apply the test. If a test succeeds, we contract $\{v, w\}$ and continue.

Note that v is scanned twice, once to mark its neighbors and once to apply the tests. Neighbors of v are scanned once. The scanned nodes are not rescanned during the current pass. Thus pass-3-4 takes linear time.

3.3 Common Preprocessing

On some instances, in particular the TSP instances, the problem size can be substantially reduced using PR tests on the input problem. For this reason, we use the following preprocessing algorithm. We start by finding the minimum single-vertex cut in order to get an upper bound on the minimum cut. Then we repeatedly apply pass-1-2 as long as a pass decreases the number of nodes by at least a factor of $1-\gamma$. After that we repeatedly apply pass-3-4 as long as a pass decreases the number of nodes by at least a factor of $1-\gamma$. Finally, if either pass-1-2 was executed more than once or pass-3-4 was executed more than once or if pass-1-2 followed by pass-3-4 reduces the number of nodes by at least a factor of $1-\beta$, we repeat the pass. The bail-out parameters β and γ are algorithm-specific, and we always choose $\beta \geq \gamma$. For all our codes, $\gamma = 0.5$. The value of β is higher for algorithms that use PR tests internally and lower for other algorithms. Note that the preprocessing runs in $O(m \log n)$ time.

4 Nagamochi-Ibaraki Algorithm

4.1 Review of the Algorithm

We begin by briefly reviewing the Nagamochi-Ibaraki algorithm. The algorithm proceeds in phases and maintains a value $\hat{\lambda}$, which is the capacity of the capacity cut seen so far. In each phase, the algorithm identifies one edge $\{v, w\}$ such that $\lambda_{v,w}(G) \geq \hat{\lambda}$. Once such an edge is found, then by Lemma 2.1, we know that the edge is safe to contract. Repeating this $n-2$ times, the minimum cut is found. Assuming a routine called `ContractSafe` that contracts the safe edge and returns the contracted graph along with a new minimum cut estimate, we get the following high-level pseudocode:

<u>**Algorithm** `ni`</u>
input: graph G
output: minimum cut value λ

$\quad\hat{\lambda} = \min\{c(v) : v \in V\}$
$\quad$**while** $|V| \geq 2$
$\quad\quad(G, \hat{\lambda}) \leftarrow \texttt{ContractSafe}(G, \hat{\lambda})$
$\quad$**return** $\hat{\lambda}$

The original Nagamochi-Ibaraki algorithm [32] implements `ContractSafe` via a graph search. Initially, all edges are unscanned and all nodes are unvisited. The algorithm maintains variables $r(v)$ for each node v and $q(e)$ for each edge e, where $r(v)$ is the sum of the capacities of the edges between v and nodes already visited, and $q(e)$, for $e = \{v, w\}$, is the value of $r(w)$ when e is scanned, starting at v. The search always chooses the unvisited node v with maximum $r(v)$ and scans all its outgoing edges. Let x, y be the next-to-last and last node visited during the search, respectively. As is shown in [32], the trivial cut $\{y\}$ is a minimum x-y cut, so one can update $\hat{\lambda}$ updated to be the minimum of $\hat{\lambda}$ and $c(y)$, and contract x and y. Thus each phase contracts at least one edge, and there are at most $n-3$ phases to contract down to two nodes.

The implementation of [33] incorporates two additional heuristics which we use as well. The first may directly contract additional edges, while the second may decrease the estimate on $\hat{\lambda}$. For complete justification of these heuristics, see [33].

The first heuristic is to contract any edge $e = \{v, w\}$ with $q(e) \geq \hat{\lambda}$, as $q(e)$ is actually a lower bound on $\lambda_{v,w}(G)$. This observation greatly speeds up the algorithm, as every additional edge contracted reduces the number of phases by one. Further, since we compute the q values anyway, the heuristic is essentially free. The code for this heuristic appears in the loop starting at line (**) below.

The second heuristic is based on the observation that whenever we are scanning a node v, we have already visited a connected set of nodes V'. We can keep track of the cut defined by each V' and if $c(V')$ happens to be less than $\hat{\lambda}$, we can update $\hat{\lambda}$ accordingly. The code for this heuristic, often called the α-heuristic, appears at line (*) below.

The pseudocode appears below:

Procedure `ContractSafe`

input: graph G, upper bound on cut $\hat{\lambda}$

output: contracted graph G, new upper bound $\hat{\lambda}$

  For all $v \in V$

    $r(v) \leftarrow 0$

    mark v unvisited

  For all $e \in E$

    mark w unscanned

  while there is an unscanned node

    v is the unscanned node with largest $r(v)$

(*) $\alpha \leftarrow \alpha + c(v) - 2r(v)$

    $\hat{\lambda} \leftarrow \min\{\alpha, \hat{\lambda}\}$

    for each unscanned $e = \{v, w\}$

      $r(w) \leftarrow r(w) + c(v, w)$

$q(e) \leftarrow r(y)$

Mark e scanned

Mark v visited

(**) **for all** $e = \{v, w\}$ with $q(e) \geq \hat{\lambda}$

$G \leftarrow G/(v, w)$ with new node v'

$\hat{\lambda} = \min\{c(v'), \hat{\lambda}\}$

return $\hat{\lambda}$

4.2 Data Structures

We implemented `ni` using both compact contraction and set-union contraction. Preliminary experiments showed that the two data structures are incomparable in practice: Each was significantly faster on some problems and significantly slower on others.

We use amortization techniques to combine the two data structures in a way that, by design, assures that the combination never takes much more time than the best of the two. The structure of the Nagamochi-Ibaraki algorithm makes this possible. Recall that the algorithm operates in phases. A sequence of edge contractions takes place after each phase. Each phase takes linear time in the current graph size, plus some priority queue operations.

We combine the two data structures as follows. At the beginning of every phase, we compact the graph by deleting self-loops, merging parallel arcs, and making every arc head point directly to the set representative. We use the compact representation during a phase. During the edge contraction sequence after the phase, we use the set-union data structure. This combination is efficient because the more efficient data structure is used in an appropriate part of the computation. Graph compaction is the additional overhead of the combination scheme, but the compaction takes linear time – usually less time than the preceding phase.

4.3 Incorporating the Padberg-Rinaldi Heuristic

Nagamochi et al. [33] show that the Nagamochi-Ibaraki algorithm can benefit from the PR heuristics. They incorporate a variant of the source version of the heuristic (see Section 3): At the end of every phase, they take a node created by the last contraction of the phase and apply the heuristic to it. This is the strategy used in their code `hybrid`.

Although in some cases this strategy works well, it has several disadvantages. First, there is no preprocessing stage. Second, the tests are applied only to one node after every phase. Third, the tests may be expensive because we may end up doing $\Omega(n^2)$ work.

We incorporate PR heuristics into `ni` as follows. We use PR preprocessing on the input graph and, after every k-th phase, we apply pass-1-2 and pass-3-4. (We set $k = 2$ for our tests.) We use node ages for pass-3-4 selection. We set node ages to zero at the beginning of every phase, so before pass-3-4, age of a node is the number of nodes contracted into it during the phase and the preceding pass-1-2. (Several different node selection rules we tried were less effective on some problems.)

The total cost of our PR heuristic is well-behaved. Preprocessing takes $O(m \log n)$ time. Since the cost of each PR pass is linear, it is amortized by the cost of the preceding phase.

As we shall see, the PR heuristic is crucial to `ni`'s efficiency.

5 Hao-Orlin Algorithm

5.1 Push-Relabel Method

First we review the push-relabel method [19] for finding minimum s-t cuts (as well as maximum flows) in directed graphs. We omit many details of the push-relabel methods, as they are not directly related to our implementation of the Hao-Orlin algorithm, `ho`. For a detailed description of the push-relabel method, see [19], and for details of an efficient implementation of the push-relabel algorithm for the maximum flow problem, see [10]. Our code `ho` is in many respects similar to that maximum flow code.

We begin with some additional definitions. Our algorithm maintains a *preflow*, which is a relaxed version of a flow. A preflow satisfies conditions (2) and (3), and the following relaxation of condition (4):

$$\sum_{v \in V} f(v, w) \geq 0 \ , \forall w \in V - \{s, t\}. \tag{6}$$

Given a preflow f, we define the *excess* at vertex v with respect to preflow f by $e_f(w) = \sum_{v \in V} f(v, w)$. We define the *residual capacity* $c_f(v, w)$ of an edge (v, w) to be $c_f(v, w) = u(v, w) - f(v, w)$. The *residual network* $G_f = (V, E_f)$ is the network induced by the edges that have positive residual capacity.

A *distance labeling* is a function $d : V \rightarrow \mathbf{N}$ that satisfies $d(v) \leq d(w) + 1$ for every $(v, w) \in E_f$ and $d(s) - d(t) \leq n$. Intuitively, a distance labeling gives a "locally consistent" estimate on the distance to the sink. The second condition, $d(s) - d(t) \leq n$, implies that if

$d(v) \geq d(t) + n$, then t is not reachable from v in G_f. In particular, the sink is not reachable from the source. An arc $(v, w) \in E_f$ is *admissible* if $d(v) > d(w)$. We say that a vertex v is *active* if $e_f(v) > 0$ and $d(v) < d(t) + n$.

Given a preflow f and a distance labeling d, we define *push* and *relabel* operations, which update f and d, respectively, as follows. The push operation applies to an admissible arc (v, w) such that v is active, increases flow on (v, w) by as much as possible: $\min(c_f(v, w), e_f(v))$. The relabel operation applies to an active vertex v with no outgoing admissible arcs, and sets $d(v)$ to the highest value allowed by the distance labeling constraints: one plus the smallest distance label of a vertex reachable from v via a residual arc.

Lemma 5.1 [19] *The relabel operation increases $d(v)$.*

We assume that a relabel operation always uses the *gap relabeling heuristic* [9, 13]. Just before relabeling v, the heuristic checks if any other vertex has a label of $d(v)$. If the answer is yes, then v is relabeled. Otherwise, the heuristic deletes all vertices with distance labels $d(v)$ or greater from the graph, as the sink is not reachable from these vertices. This heuristic often speeds up push-relabel algorithms for the maximum flow problem [2, 10, 13, 34] and is essential for the analysis of the Hao-Orlin algorithm.

We use a standard implementation of gap relabeling that maintains an array of buckets, $B[0 \ldots 2n-1]$, with bucket $B[i]$ containing a doubly linked list of all vertices with distance labels equal to i. To check if v is the only vertex with distance label $d(v)$, we check if it is the only vertex in the bucket $B[d(v)]$. To find all vertices with distance label $d(v)$ or greater, we examine the buckets starting from $B[d(v)]$.

The generic push-relabel algorithm for finding a minimum s-t cut starts by setting all distance labels to zero. Then the algorithm sets $d(s) = 2n - 1$ [1] and saturates all arcs out of s. This gives the initial preflow and distance labeling. The the algorithm applies push and relabel operations in an arbitrary order. When no operation applies, the algorithm terminates.

Theorem 5.2 [19] *When the generic algorithm terminates, the set of vertices that can reach t in G_f defines a minimum cut, and $e_f(t)$ is the minimum s-t cut value.*

Theorem 5.3 [19] *The number of relabel operations in the generic algorithm is $O(n^2)$. The number of push operations is $O(n^2 m)$.*

[1] For an s-t cut computation, we can set $d(s) = n$; the higher value is needed for the Hao-Orlin algorithm.

The *discharge* operation combines the push and relabel operations at a low level. The discharge operation applies to an active vertex v. The operation applies push operations to arcs out of v and relabel operations to v until v is no longer active.

At the high level, push-relabel algorithms differ by the strategy for selecting the next active vertex to discharge. We used the *highest label (HL)* strategy: discharge an active vertex with the highest distance label. This strategy, in combination with appropriate heuristics, seems to give the best results in practice [10]. This strategy also reduces the number of push operations.

Theorem 5.4 [8] *The push-relabel algorithm with the highest label selection performs $O(n^2\sqrt{m})$ push operations and runs in $O(n^2\sqrt{m})$ time.*

Sophisticated data structures, such as dynamic trees, improve running times of the push-relabel algorithm in the worst case [19]. However, these data structures do not seem to help in practice, at least in the maximum flow context. We did not use sophisticated data structures in our implementation, except for the data structures used for graph contraction.

5.2 Hao-Orlin Algorithm

Recall that the Gomory-Hu minimum cut algorithm solves the minimum cut problem in $n-1$ minimum s-t cut computations. The Hao-Orlin algorithm is an implementation of the Gomory-Hu algorithm that is based on the push-relabel method; it uses the preflow and the distance labeling from the previous s-t cut computation to obtain an initial preflow and distance labeling for the current one. This allows us to amortize work of the $(n-1)$ s-t cut computations, and to obtain worst-case time bounds that are the same as those to perform one maximum flow computation. We give a brief description of this algorithm below. See [23] for details.

A key concept of the Hao-Orlin algorithm is that of a *frozen layer* of vertices. A frozen layer is a set of vertices; different layers are distinct. A vertex is *frozen* if it belongs to a frozen layer and *alive* otherwise. We denote the set of frozen vertices by F and the set of alive vertices by A. Initially all vertices are alive. When gap relabeling discovers a set of vertices disconnected from the sink, these vertices form a new frozen layer. This layer is deleted from the graph and put on a stack of layers. When a layer of vertices is frozen, the values of the vertex distance labels are the same as they were just before the relabeling operation during which the layer was discovered. At some point of the algorithm, the top layer will be popped from the stack and the vertices of this layer will become alive.

The Hao-Orlin algorithm starts as follows. The first source and sink pair is selected arbitrarily.[2] We set the distance label of the source to $2n-1$ and saturate all arcs out of the source. Distance labels of all other vertices are set to zero. Then we start the first s-t cut computation.

After an s-t cut computation terminates, we examine the cut it finds and remember the cut if its capacity is smaller than that of the best cut we have seen so far. Then we start the next computation as follows.

First we contract the source and the sink; the contracted vertex s becomes the source of the next cut computation. Next we select the sink t of the new computation. If $A - \{s\} = \emptyset$, we unfreeze the top frozen layer if F is not empty and halt if F is empty. Assuming that F is not empty, we pick the vertex in $A - \{s\}$ with the smallest distance label as the new sink. Finally we set the distance label of the new source to $2n-1$ and saturate all arcs going from the source to alive vertices. Then the next cut computation begins.

One can show that at every s-t cut computation except for the first one, the distance label of the new sink does not exceed the distance label of the old sink by more than one. Thus when we start a new s-t cut computation, the distance labels are valid, which implies the correctness of the algorithm. The running time of the algorithm is related to the total number of relabelings done. The total number of relabels over all s-t cut computations can be bound by $O(n^2)$, the same as the bound for one s-t cut computation done in isolation. Results of [8, 23] then imply the following theorem.

Theorem 5.5 [8] *The Hao-Orlin algorithm with the highest label selection runs in $O(n^2\sqrt{m})$ time.*

5.3 Implementation Details

In this section we describe implementation details of the `ho` code. We describe general implementation issues and heuristics used. The internal PR heuristic for `ho` is described in 5.4.1.

Graph representation. With no PR heuristics, the algorithm can be implemented without graph contraction. Instead of contracting the source and the sink at the end of an s-t cut computation, we declare the sink to be another source. The PR heuristics, however, are very

[2] On some problem families, the algorithm is sensitive to this choice. We use a heuristic to make a better choice of the first source-sink pair. See Section 5.3.

effective on some problem classes. To support these heuristics, we use graph data structures similar to those used in `ni`, with additional fields to support variables needed by the push-relabel method, such as flow values on edges and distance labels and excess at nodes. When using PR heuristics, we contract the source and the sink at the end of every s-t cut computation.

Highest label selection. We use the array of buckets $B[0 \dots 2n-1]$ to implement highest label selection in addition to gap relabeling. A bucket $B[i]$ contains a list of all alive vertices v with $d(v) = i$ and a list of all alive and active vertices with $d(v) = i$. We also maintain an index b into the array such that for $b < j \leq 2n-1$, $B[j]$ does not contain any active vertices. To select the next active vertex to scan, we remove a vertex from the active list of $B[b]$ if the list is not empty; otherwise we decrement b until the list is nonempty. If no active vertex is alive, the current s-t cut computation terminates.

Global updates. As in the maximum flow context, *global updates, i.e.,* computing exact distances to the sink, are useful for many problem classes. In the minimum cut context, there are several natural modifications.

- The sink's distance label is nonzero, so the computation is done using backwards breadth-first search with the sink distance set to the sink's distance label.
- The computation is done only on the nonfrozen graph.
- Vertices not reachable by the breadth-first search computation are frozen without changing their distance labels. (The sink is not reachable from these vertices.)

We use an amortization strategy to determine when to perform a global update: A global update is performed as soon as the number of relabeling from the beginning of the computation or from the last global update exceeds β times the number of nonfrozen vertices. In our implementation, $\beta = 2$.

Global updates do not always improve the running time; on some problem families, such as NOI families, the running time becomes worse. However, the running times never become much worse in our tests, and sometimes are much better than without global updates.

Source and sink selection. On some problem classes, the algorithm is sensitive to the way the first source-sink pair is chosen. We chose a vertex with the largest capacity as the first source and a neighbor of this vertex as the first sink.

Single node layers. Suppose a frozen layer consists of a single node v and the node was frozen and let x be the time immediately after freezing x. Then when v is unfrozen at time y, there are two regular nodes: v and s, and v becomes the sink. All vertices contained in regular nodes at time x are contained in s at time y. Instead of computing the s-v minimum cut at time y, we can compute it at time x: it is equal to the excess at v is all arcs from active nodes into v are saturated.

Instead of freezing v, we compute the cut value and update the currently best minimum cut if needed. Then we saturate all arcs out of v and contract v into s.

This earlier contraction of v increases the chances of success for the internal PR heuristic.

5.4 Heuristics

For `ho` we used two different types of heuristics. First, as in all the other algorithms, we used the PR heuristics. Second, we introduced a new heuristic, which we call *excess contractions*.

5.4.1 Padberg-Rinaldi Heuristics

As in all our algorithms, we apply PR preprocessing at the beginning. Our *internal* PR heuristic is based on the following fact that is easy to prove using [23]: If s is the source and the edge $\{s, w\}$ passes one of the tests `PR1`, ..., `PR4`, then we can saturate all arcs out of w, contract the edge, and continue.

We use amortization to decide when to apply `PR1` and `PR2` to the source. When the algorithm performs enough work to amortize the tests previous, we apply the tests again. We use similar strategy to apply the `PR3` and `PR4` tests.

The internal tests in `ho` are not as helpful as those in `ni`. Either the way `ho` works does not create as may opportunities for the tests as in the case of `ni`, or a better internal PR heuristic exists, but we did not find it.

Remark. It is possible to use PR tests to contract edges not adjacent to the source. For example, if v and w are regular nodes distinct from s and t and $\{v, w\}$ passes a PR tests, we can contract $\{v, w\}$, provided that we perform a global update immediately afterwards to restore the guarantee that the distance label is valid. In our experience, such non-source PR tests usually slowed down the code, and we decided against using these tests. It would be interesting to find a more effective implementation of these internal tests.

5.5 Excess contractions.

We introduce a simple heuristic that often allows us to contract a vertex in the middle of a flow computation. The general results on the push-relabel method [19] imply that the excess at a vertex v is a lower bound on the capacity of the minimum s-v cut. Thus, if at some point during an s-t cut computation the excess at v becomes greater or equal to the capacity of the minimum cut we have seen so far, we contract v into the source and saturate arcs going out of v. Note that v can be either regular or frozen. The correctness proof for this heuristic is straight-forward. We call this heuristic *excess detection.*

A special case of the excess detection heuristic occurs when v is the sink. In this case we stop the current s-t cut computation and, if not all vertices are contracted into the source, we start the next s-t cut computation.

Excess detection is inexpensive and on some problems, it reduces the number of s-t cut computations significantly. One needs to be careful when implementing excess detection because of the hidden recursion: Suppose excess at v becomes large and v is contracted into the source. When v's outgoing arcs are saturated, excess at some of v's neighbors may become large, and these neighbours are contracted as well. This greatly complicates the code as we now need code to compute a maximum flow on a graph that is changing during the course of the flow computation.

6 Karger-Stein Algorithm

6.1 Review of algorithm

We begin by reviewing, at a high level, the recursive contraction algorithm of Karger and Stein [26]. Similar to the Nagamochi-Ibaraki algorithm, this algorithm repeatedly contracts edges. In contrast to NI, this algorithm will sometimes contract an edge that is in the minimum cut. In particular, the algorithm repeatedly chooses edges at random to contract. The key insight is that in an m-edge graph, the probability that a randomly chosen edge is in the minimum cut is at most $2/m$, and thus a randomly chosen edge is actually not very likely to be in the cut. Thus it is possible to contract a large number of edges and still have a reasonably good chance of not having contracted the minimum cut. In particular, if one contracted the edges of the graph one at a time, the min-cut would survive with probability $\Omega(n^{-2})$. Through the use of recursion, we are able to run a series of edge contractions for which the minimum cut survives with probability $\Omega(1/\log n)$. Repeating the algorithm $O(\log^2 n)$ times yields an algorithm which finds the minimum cut with high probability.

We give a high level description of algorithm KS:

<u>Algorithm KS(G)</u>

input A graph G of size n.

if G has 2 nodes

then return the weight of the cut in G

else repeat <u>**twice**</u>

Let W be an upper bound on the minimum cut.

mark each edge (u, v) with probability $1 - 2^{-w(u,v)/W}$ (*)

Let G' be the result of contracting all marked edges in G

recursively call KS(G)

return the smaller of the two resulting values.

This differs slightly from the recursive contraction algorithm described in [26]. In that algorithm, rather than implementing the line marked (*), we repeatedly select and contract one edge at a time until the number of graph nodes is reduced to $n/\sqrt{2}$. It is shown there that such a sequence of contractions preserves the minimum cut with probability at least 1/2. Relying on this fact it is shown that the overall algorithm returns the minimum cut with probability $\Omega(1/\log n)$.

Our new algorithm's contraction phase (*) also preserves the minimum cut with probability at least 1/2. For if the edges of the minimum cut have weights $w_1, \ldots, w_k$ such that $\sum w_i = c$, then the probability that no minimum cut edge is contracted is

$$\begin{aligned} \prod 2^{-w_i/W} &= 2^{-c/W} \\ &\geq 1/2. \end{aligned}$$

We chose to modify the algorithm as described above for the following reason. The original contraction algorithm made the pessimistic assumption that the total edge weight when n nodes remained was $nc/2$. Under this assumption, contraction to less than $n/\sqrt{2}$ nodes might not preserve the minimum cut with probability at least $1/2$. Consider, however, the case of two cliques joined by a single edge. In this case, the original algorithm is being overly conservative in contracting to only $n/\sqrt{2}$ nodes. It could in fact contract to a far smaller graph while still preserving the minimum cut with reasonable probability. Suppose, on the other hand, that we use our new algorithm with an upper bound W that is close to c. In this case, we get far more contraction than the original algorithm does. This reduces the recursion depth of the algorithm, which reduces the number of leaves in the recursion tree. Since (at least for sparse graphs) the running time of the algorithm is dominated by the leaves of the recursion, we get improved performance.

We have introduced a new parameter W which must be handled carefully. Initially, we have no upper bound on the minimum cut value. If W is extremely large relative to c then the probability of edge contraction is extremely small, which could make the recursion depth and running time arbitrarily large. However, we have a convenient upper bound on c in the form of the minimum degree of the input graph. It is easy to prove that if we use this upper bound, with high probability the size of the graph is reduced by a constant fraction at each iteration. This suffices to prove a time bound which is polynomial, though not quite as good as the original algorithm's. We conjecture that the new algorithm's performance is in fact equal to that of the old one's, but this remains to be proved. In practice, experience has shown that we usually stumble across a minimum cut almost immediately. Once this happens, W is set to c for the remainder of the execution and we get good performance.

In order to implement the algorithm most efficiently we need to address several issues. These include the graph data structure, random edge selection, and the use of PR tests. We discuss these issues below.

6.2 Depth Analysis

Our new implementation requires an analysis somewhat different from that of [26]. The analysis of the Recursive Contraction Algorithm involved two parts. The first was a recurrence for the time taken to run a single iteration of the Recursive Contraction Algorithm. Our new implementation no longer satisfies this recurrence, since we do not guarantee a specific reduction in the number of nodes in a recursive call. However, we have observed that this implementation is faster in practice. The second important part of [26] was an analysis of the success probability of a single iteration, which determined the number of iterations needed to give a high probability of success. In order to guarantee a certain overall success probability for our new implementation, we must determine the success probability of a single iteration of our new algorithm.

The success probability of the original RCA was determined by a recurrence that applies to our new algorithm as well. This recurrence relied only on the fact that each of the two recursive calls involved contractions with a $1/2$ chance of preserving the minimum cut. Since our new algorithm has this property as well, the same recurrence applies. We slightly reformulate the main lemma proven there.

Lemma 6.1 *The probability that the minimum cut survives at some recursion node at depth d in the (new) RCA is at least $\frac{4}{7/2+d+\ln(d+2)}$.*

In the original RCA, the depth of the recursion tree was known to be $2\log n$ and the $\Omega(1/\log n)$ probability of finding the minimum cut followed immediately. We remark that our base case is somewhat different and simpler, since we do not need to terminate the algorithm when 7 nodes remain. On the other hand, there is an important detail that is missing. In the original algorithm, it was known that at depth $2\log n$ there were exactly 2 nodes remaining, so that if the minimum cut survived, it was also discovered. In our new implementation, the number of nodes in subproblems is a random variable, so we cannot guarantee contraction to 2 nodes at a given depth. We will therefore have to do some additional work to prove that when the minimum cut survives, it is actually found. We begin by assuming that the upper bound W has been set to c. At the end of this section, we justify our assumption.

Our analysis is based on a network reliability analysis from [24]. That paper considers a graph in which each edge fails with probability p, and determines the probability that the graph remains connected. This is related to our objective as follows. Our goal is to show that at a certain recursion depth, the recursion has terminated because our graph has been contracted to a single node. That is, we want the set of contracted edges to span (connect) all of G. Inverting this objective, we can consider deleting the set of edges that were not contracted, and require that deleting these edges not disconnect the graph.

Now consider a particular recursion node at depth d. The graph at this node is the outcome of a series of independent "contraction phases" in which each edge is contracted with probability

$1 - 2^{-1/c}$ (by our assumption that $W = c$). That is, the probability of not being contracted is $2^{-1/c}$. It follows that at depth d, the probability that any edge is not contracted is $2^{-d/c}$. We now invert our perspective as in the previous paragraph. We ask whether deleting the uncontracted edges leaves us with a single component. In other words: we consider deleting every edge of G with probability $2^{-d/c}$, and ask whether the remaining (contracted) edges connected G.

The following is proven in [30] (see also [24]), using the fact that among all graph with minimum cut c, the graph most likely to become disconnected under random edge failures is a cycle:

Lemma 6.2 *Let G have n edges and minimum cut c. Then the probability that G is not connected after edge failures is at most n^2p^c.*

We might hope to apply this lemma as follows.

Corollary 6.3 *At recursion depth $k \log n$, for $k > 2$, the probability that G has not been contracted to a single node is at most n^{2-k}.*

Proof. At depth $d \log n$, the (cumulative) probability of non-contraction is $p = 2^{-(k \log n)/c} = n^{-k/c}$. Plugging into Lemma 6.2, we find that $\delta = k - 2$. Now apply Lemma 6.2. ∎

Unfortunately, this is not sufficient to prove what we want. At depth $3 \log n$ in the recursion tree, there are n^3 recursion nodes. Although each one has a $1/n$ chance of not being a leaf, there is a reasonable chance that not all are leaves. We must therefore perform a more careful analysis.

We evaluate the probability of success as the product of two quantities: the probability that the minimum cut survives contraction to the given depth and the probability that the minimum cut is found by our algorithm given that it survives. Conditioning on the survival of the minimum cut makes our analysis somewhat complicated.

Given the conditioning event, there is some node at depth $k \log n$ in which the minimum cut has survived. We would like to claim that this node has been contracted to two nodes. Unfortunately, conditioning on the survival of the minimum cut means that no minimum cut edge has been contracted, a condition that breaks our reliability model.

To deal with this problem, we rely on the fact that our new algorithm examines the degrees of the nodes in its inputs. It therefore suffices to show that at least one side of the minimum cut is contracted to a single node, since this single node will be examined by the algorithm. We will in fact argue that both sides will be contracted to a single node. Another way to say this is that the edges failures break G into exactly two connected components.

Lemma 6.4 *Conditioned on the fact that a minimum cut has failed, the cycle is the most likely graph to partition into more than two pieces under random edge failures.*

Proof. A straightforward modification of [30]. ■

Corollary 6.5 *Conditioned on the fact that a minimum cut has failed, the probability a graph partitions into* 3 *or more pieces is at most* $np^{c/2}$

The following lemma is an immediate corollary.

Lemma 6.6 *Conditioned on that fact that the minimum cut survives at some node of depth* $k \log n$*, the RCA finds the minimum cut with probability at least*

$$f(k,n) = 1 - n^{1-k/2}.$$

Proof. Consider the depth $k \log n$ recursion node at which the minimum cut survives. At this depth, the probability of edge "failure" is $n^{-k/c}$. From the previous Lemma, $f(k,n)$ is the probability that the contracted edges at this recursion node reduce the graph to two nodes, implying we find the minimum cut. ■

Lemma 6.7 *For any* k*, at depth* $k \log n$*, the new RCA finds the minimum cut with probability at least*

$$\frac{4}{7/2 + k \log n + \ln(2 + k \log n)} f(k,n).$$

Proof. From Lemma 6.4 we find that the probability that the minimum cut survives in some recursion node at depth $d = k \log n$ is at least $\frac{4}{7/2+d+\ln(2+d)}$. We now condition on the event having taken place and apply Lemma 6.6 to find the probability of success $f(k,n)$ given this event. The overall probability is the product of these two quantities. ■

Corollary 6.8 *The probability that a single iteration finds the minimum cut is at least*

$$\frac{4}{7/2 + 2\log n + 2\log\log n + \ln(2 + 2\log n + 2\log\log n)}(1 - \frac{1}{\log n})$$

Proof. Set $k = 2 + \frac{2\log\log n}{\log n}$ in Lemma 6.7 ■

From this analysis of the success probability of a single iteration, it is easy to compute the number of iterations needed to achieve a specified success probability.

6.3 Implementation Details

We use a simple data structure for storing the graphs. We maintain an array of edges, where each edge stores its endpoints and weight. We also maintain an array of nodes, where each node maintains the sum of the weight of its incident edges, its name, and several auxiliary fields. Note that each node does not maintain a list of its incident edges, as this was not necessary in our implementation. An advantage of this representation is that edge lists are stored without a linked-list data structure overlay; this improves performance by making the data structure smaller and increasing memory reference locality as we traverse the edge list.

We found that it is much wiser to randomly choose and contract sets of edges, as described above, rather than choosing and contracting the edges one by one, as is done in [26]. We needed to use an exponential distribution to sample each edge with probability exponential in its weight. Though we were initially concerned by the resulting large number of calls to exponential/logarithm functions, we found that in practice the generation of these random numbers was not a significant part of the running time.

It should be noted that our algorithm as stated makes no use of adjacency lists. There is no need to maintain a list of edges incident on each node; rather, each edge is processed independently. However, we still found it useful to sort edges according to their endpoints. One reason is that contracting nodes tends to create parallel edges. By merging these parallel edges, we reduce the number of edges the algorithm must consider for contraction in later iterations.

As edges are contracted, we use a set union data structure to maintain the nodes of the graph. Besides letting us determine the structure of the minimum cut we find, this also lets us sort the edges. After a pass through the edges, we use the set-union structure to construct the contracted graph. We assign node numbers to the new nodes, update the edges to reflect the new node numbers, and bucket-sort the edges to their corresponding nodes. This gives us a sorted list from which it is easy to remove duplicates and on which it is easy to perform PR tests.

6.4 PR tests

We incorporated PR tests into the code in two places. As in the other algorithms, we incorporated PR tests in the beginning of the algorithm, immediately after reading in the input graph. For this algorithm, these initial PR tests are particularly important, as they occur before beginning the randomized portion of the algorithm. Since the number of independent runs of the algorithm needed increases with n, we can decrease the number of independent runs needed. Thus we continue to iterate PR tests over all edges as long as the number of nodes decreases by at least 5%.

6.4.1 PR1 and PR2 Tests

PR1 and PR2 tests in `ks` were quite easy to implement. These tests compare the weight of an edge to either a global upper bound on the minimum cut or to the weights (total incident edge weight) of the edge's endpoints. As described above, in each iteration of the algorithm, each edge is marked contractible with some probability. Since a successful PR test also results in determining that an edge is contractible, each time we sample an edge, we can apply PR tests to that edge. Thus we actually mark an edge as contractible if it passes either of the PR tests or if we choose it randomly. This clearly adds only a small overhead. In order to implement the PR2 tests, we need to compute edge weights. This simply involves an additional traversal of the edge array prior to the contraction traversal.

There are additional complications associated with using PR tests in the KS algorithm. These complications arise for two reasons. The first reason is that the algorithm does not store candidate minimum cuts as it progresses. It waits until the graph is contracted down to 2 nodes, and then inspects the minimum cut. However, Lemma 3.1 states that when the PR tests apply to edge (v, w), either it is safe to contract edge (v, w) or v or w or (v, w) is the trivial minimum cut. Therefore, we must modify the algorithm to check for trivial cuts. In particular, when PR1 is satisfied with equality, it is possible that edge (v, w) is the minimum cut. Thus we must save the value of edge (v, w) as a possible cut value. Also, when executing PR2, we must save the trivial minimum cuts. Note that if we only care about the value of the cut, this modification is not necessary.

The second, and more serious, complication arises because we are contracting many edges simultaneously. It is possible to mark and contract a set of edges and thereby "miss" the minimum cut. This particular problem only arises from PR2 tests that are met with equality.

To avoid this problem, we modify the algorithm so that in any iteration, for each node, at most one incident edge is contracted because of a PR2 test that is met with equality.

6.4.2 PR3 and PR4 Tests

As with the other codes, PR3 and PR4 tests significantly improved the performance of our algorithm. However, our experience has shown that these tests had almost no effect after the preprocessing phase of the implementation. A possible explanation for this effect is the following (we focus on the PR4 case; a similar arguments applies for the PR3 test). The PR4 test applies when the sum of capacities of length-one and length-two paths exceeds c, where the capacity of a length-two path is the smaller of its edge's capacities. Consider a randomized contraction phase and its impact on such a PR4 sructure. The total weight of edges in the PR4 structure is $2c$, implying that the probability some edge in the structure is contracted exceeds 3/4. Especially over multiple levels of recursion, this accumulates much faster than the 1/2 chance that a minimum cut edge will be contracted. Once we contract an edges in PR4 structure, the PR4 test will no longer apply. In other words, in an intuitive sense, randomized contraction is taking care of the PR3 and PR4 tests before we have time to apply them explicitly.

Consequently, we decided not to use PR3 and PR4 tests at all after preprocessing. Since we could not think of an efficient implementation of these tests without using adjacency lists, we use the the data structures and code of `ni` to do the PR preprocessing, and then switch to the simpler and more compact data structure described above after the preprocessing.

7 Karger's Algorithm

Karger's algorithm [25] is based on the following two observations:

- Any undirected graph with minimum cut c has a packing of c spanning trees that uses each edge at most twice [17].
- Given any packing of c spanning trees that uses each edge at most twice, at least some of the trees only cross the minimum cut twice.

The second fact follows from a simple averaging argument: there are at most $2c$ "half edges" crossing the minimum cut which must be shared among c trees. So some trees only get 2 edges. This motivates the following definition:

Definition 7.1 *Let T be a spanning tree of G. We say that a cut in G k-*respects *T if it cuts at most k edges of T. We say that T k-*constrains *the cut in G.*

Karger's algorithm works as follows. First it finds a packing of spanning trees; then it examines the trees in the packing and for each tree finds the smallest cut that 1- or 2-respects the tree. The following is shown in [25]

Lemma 7.2 *The smallest cut that 1-respects a given tree can be found in $\Theta(m+n)$ time. The smallest cut that 2-respects a given tree can be found in $\Theta(n^2)$ time.*

The only operations used in the algorithm are least common ancestor computations and array sums. Thus, the constants hidden by the Θ-notation are quite small. On the other hand, the 2-respects construction involves explicitly computing a set of $\binom{n}{2}$ values (one for each pair of tree edges). Therefore, unlike the other algorithms we examine, this one is *guaranteed* to take $\Omega(n^2)$ time regardless of the input.

A related algorithm of [25] has a better time bound of $O(m \log^3 n)$. This algorithm, however, is more complicated and the constant factors of its implementations are likely to be large. Because of this and our experience with the simpler algorithm, we think that the theoretically faster algorithm is unlikely to be competitive with the best codes on the problems in our study.

7.1 Packing spanning trees

It remains only to find a tree that constrains the minimum cut. We do so by finding a tree packing and considering the trees in it; some of them appropriately constrain the minimum cut. To find the packing of spanning trees, we use Gabow's Algorithm [17]. On an unweighted graph, this algorithm finds a packing of c trees in $O(mc \log m/n)$ time. This is fine for small values of c but expensive for large values. To ensure a small value of c, we use random sampling:

Lemma 7.3 ([27]) *Given any graph G, in linear time we can construct a* skeleton *graph H on the same vertices with the following properties:*

- *H has $m' = O(\epsilon^{-2} n \log n)$ edges,*
- *the minimum cut of H is $c' = O(\epsilon^{-2} \log n)$,*
- *the minimum cut in G corresponds (under the same vertex partition) to a $(1+\epsilon)$-minimum cut of H.*

Corollary 7.4 *Let $\epsilon = 1/6$. If we pack c' trees in H, then either*

- *some tree* 1*-constrains the minimum cut of* G*, or*
- 1/5 *of the trees* 2*-constrain the minimum cut of* G

This corollary suggests a natural theoretical implementation of the algorithm. Construct the skeleton with $\epsilon = 1/6$ and pack the trees in the skeleton using Gabow's Algorithm. For each tree, find the minimum cut that 1-respects it. Then, for a small random subset of the trees, find the minimum cut that 2-respects each tree in the sample. Since checking for 1-respected cuts is much easier than checking for 2-respected cuts, it makes sense to avoid performing the 2-respects test for every tree. Since, if the 1-respects test fails, many trees must 2-constrain the minimum cut, testing a random sample of them is sufficient.

7.2 Implementation Details

Karger's algorithm, `k`, leaves open many implementation options. We discuss the major ones below.

Checking trees for 2-respecting cuts. Karger gives two different ways to check for 2-respecting cuts: a relatively simple dynamic programming method that takes $O(n^2)$ time per tree, and more elaborate method that uses dynamic trees, but takes only $O(m \log^2 n)$ per tree. We believe that the constant factor hidden in the O of the latter is large, so we only implemented the former.

Packing spanning trees. Karger suggests two entirely different ways to pack spanning trees. We chose to use Gabow's algorithm [17]. It would be interesting to try the alternative ([36]) too.

We tried checking to see if the tree packing contained multiple copies of the same tree, but this did not appear to help unless our sampling probability was too high, so we disabled it. It would be interesting try to use scaling in the implementation of Gabow's algorithm. This could potentially run faster and return fewer distinct trees on some graphs.

Estimating the Minimum Cut In order to perform the sampling step correctly, the algorithm needs to know the value of the minimum cut. In [27] Karger gives two ways to resolve this problem. The first is to run Matula's ([31]) linear time 3-approximation algorithm; using that value divided by 3 gives a probability that doesn't affect correctness and at worst multiplies the number of edges in the sample by 3. The other option starts by getting a crude approximation and then samples the edges, finds a tree packing, and doubles the probability and repeats if the number of trees in the packing is smaller than expected. Since doubling the probability doubles the number of trees, finding the final tree packing dominates the time of finding all the others. If the crude approximation was within a factor of n, then the time spent sampling is at most $O(m \log n)$.

We chose to implement the latter method because of its simplicity. We used the minimum trivial cut as the crude starting approximation. This is perfectly good for an unweighted graph, but it can do very badly on a weighted graph. So after finding a tree packing we compute the value of the cut in the graph defined by the minimum cut in the sampled graph. This will get us a much better estimate when the starting point is very bad.

Sampling from Weighted Graphs A major concern is weighted graphs. The theoretical version says to pick each edge independently with probability $6\epsilon^2 \log n/c$. We can treat a graph with integer weights as an unweighted graph with multiple edges, but we don't want to flip a coin w times for an edge of weight w, and this still doesn't cover real weights. So for integer weight edges, what we want is to pick a number from 0 to w according to the binomial distribution. Notice that if we were to multiply all the edge weights by some large factor the minimum cut would go up by that factor, so the probability would go down and the mean of the distribution would stay the same. Therefore we can approximate with the Poisson distribution, which is very close to the binomial for large numbers and small mean. It is less difficult to pick a number according to the Poisson distribution, and since it is a continuous distribution it solves the problem of real edge weights.

Picking ϵ Another problem is picking the ϵ used to compute the sampling probability. In order to have high probability of some tree 2-constraining, the expected number of trees has to be $150 \log n$. With $216 \log n$ trees, if none 1-constrain at least $1/5$ of them 2-constrain, so we could check fewer than $150 \log n$ trees for 2-constraining at the cost of having found more trees in the first place. Unfortunately, in either case, we found that the time spent finding that many trees was so expensive as to make the running time several orders of magnitude worse than the other algorithms. We discovered, however, that on our test examples, finding only $6 \log n$ trees and checking only 10 of them for 2-constraining gave the right answer all the time. On the theory side, there is reason to believe that the analysis is not tight, but since we haven't been able to tighten it this implementation must be considered heuristic. There is no proof that it will be correct with high probability in all cases. Note that $6 \log n$ trees causes all cuts to be within $\epsilon = 1$ of their expected values with probability $1 - O(1/n)$ according to the theoretical analysis.

Graph Representation The graph itself is represented as in `ho` and `ni`, but we also had to represent the sampled graph and the tree packing, so there are some further issues.

Gabow's algorithm is intended for unweighted graphs, so each weighted graph edge can be in many trees. It is impractical to take the sampled graph and make it unweighted by making copies of edges. Instead we represent the sampled graph as the original and make and delete unweighted copies as needed during the algorithm. Therefore the tree edges are necessarily a different edge structure. They are stored in doubly linked edge lists, and each tree edge maintains a pointer to the corresponding graph edge.

Adding the PR Heuristics We use the same PR preprocessing as `ho` and `ni`. Since the algorithm doesn't do any other contractions, the only way the PR heuristics could help internally is if we discover a new upper bound on the minimum cut before we get a good tree packing. We implicitly get a new upper bound when we are doubling probabilities, but it is not safe to run PR tests based on an expected upper bound. Nevertheless, as mentioned above, we check the cut defined by the minimum cut in the sample to update our upper bound if possible, so when an update occurs that reduces the lower bound by more than 5% we run the PR tests again.

We also tried checking 1-respecting cuts after each tree packing step, in the hope that we might get a better estimate and be able to run PR tests again. It turned out that on our test data our initial estimates of the minimum cut were never too bad, so there wasn't much to be gained by trying to tighten the estimate.

Gabow's Algorithm on Small Cut Graphs Observe that when we sample the graph, we create a graph with minimum cut $O(\log n)$ on which to run Gabow's algorithm. If the minimum cut of the original graph was $O(\log n)$, sampling is unnecessary. In this case we can just use Gabow's algorithm directly to determine the minimum cut.

8 Problem Families

Generator	Class name	Brief description
NOIGEN	NOI1–NOI6	Random graphs with "heavy" components
REGGEN	REG1–REG4	Unions of random cycles
RANDOMGEN	RAND1–RAND2	Regular random graphs
BIKEWHEELGEN	BIKEWHE	Bicycle wheel graphs
	TSP	TSP problem instances
	PRETSP	Preprocessed TSP instances
PRGEN	PR1–PR8	Two components with a mincut between them

Table 1: *Summary of problem families.*

After experimenting with many problem generators and families, we restricted our attention to the most interesting ones. In our final experiment, we used five different problem generators, and for each one we generated several different problem families, for a total of 21 different families. We describe each family individually below. We also did experiments on minimum cut problems that arise in the solution of large TSP problems via cutting plane algorithms [3]. One of our generators, NOIGEN, is an implementation of the generator of [33], and another one, PRGEN, is an implementation of the generator from [35]. The remaining generators are original. One of the contributions of our work is to introduce this useful collection of generators than can be used by others to test minimum cut algorithms.

A brief summary of our problem families appears in Table 1. A detained description of generators and families follows. For the remainder of this section, when we say something is chosen at random from a given range, we mean an integer chosen uniformly at random in the given range.

8.1 NOIGEN generator

Our NOIGEN generator is an implementation of a generator of Nagamochi et al. [33]. The generator produces a graph consisting of several heavily connected components. With properly chosen parameters, vertices of each component are on the same side of any minimum cut.

The generator takes the following parameters:

- n - the number of vertices in the graph,
- d - the density of edges as a percentage, *i.e.*, $m = n(n-1)(d/200)$,

- k - the number of tight components,
- P - the scale between intercomponent and intracomponent edges.

The generator works as follows. We first compute m from n and d. Next the vertices are randomly colored with k different colors. We then add m edges to graph, at random. If the two endpoints of an edge have the different colors, the edge weight is chosen at random from the range $[1, 100]$, while if the two endpoints have the same color, the edge weight is chosen at random from the range $[1, 100 \cdot P]$. The desired effect is that, for large enough P, the graph will have k components and the minimum cut will consist solely of edges with endpoints in different components.

Following [33], we generated 6 different problem families. Our families are similar to those of [33]. However, we use larger problem sizes because our algorithms and hardware are faster. Based on experimental feedback, we place additional data points with "interesting" places.

Family	n	d	k	P
NOI1	300,400,500,600 700,800,900,1000	50	1	300,400,500,600, 700,800,900,1000
NOI2	300,400,500,600 700,800,900,1000	50	2	300,400,500,600, 700,800,900,1000
NOI3	1000	5,10,25,50,75,100	1	1000
NOI4	1000	5,10,25,50,75,100	2	1000
NOI5	1000	50	1,2,3,5,7,10,20,30,33,35 40,50,100,200,300,400,500	1000
NOI6	1000	50	2	5000,2000,1000,500 250,100,50,10,1

Families NOI1 and NOI2 study the affect of varying the number of vertices. Families NOI3 and NOI4 study the affect of varying the density of the graph. Family NOI5 studies the affect of varying the number of components, and family NOI6 studies the affect of varying the ratio between the weights of the intercomponent and intracomponent edges.

8.2 REGGEN generator

The REGGEN generator produces unions of random cycles. It takes the following parameters:

- n - the number of vertices,
- d - the number of random cycles.

All edges have weight 1, and the cycles are not necessarily edge-disjoint. We generated two families of graphs:

Family	n	d
REG1	1001	1,3,10,33,100,333,1000
REG2	50,100,200,400,800	50
REG3	256,512,1024,2048,4096,8192,16384	2
REG4	128,256,512,1024,2048	8,16,32,64,128

REG1 tests the affect of increasing the number of random cycles, while REG2 tests the affect of increasing the number of vertices. REG3 looks at the union of two random cycles, a problem that is hard for all of our codes, and REG4 looks at the case when the number of vertices and cycles increase simultaneously.

8.3 RANDOMGEN generator

The RANDOMGEN generator generates a random regular graph, that is, a graph in which each vertex has the same degree. It takes the following parameters:

- n - the number of vertices
- d - the degree

It generates a graph with n vertices and $nd/2$ edges in which each vertex has degree d. It does this by generating a list of length nd which contains each vertex d times and then randomly picking pairs from this list (without replacement) to get edges. The graph formed may have self-loops.

We generated the following families:

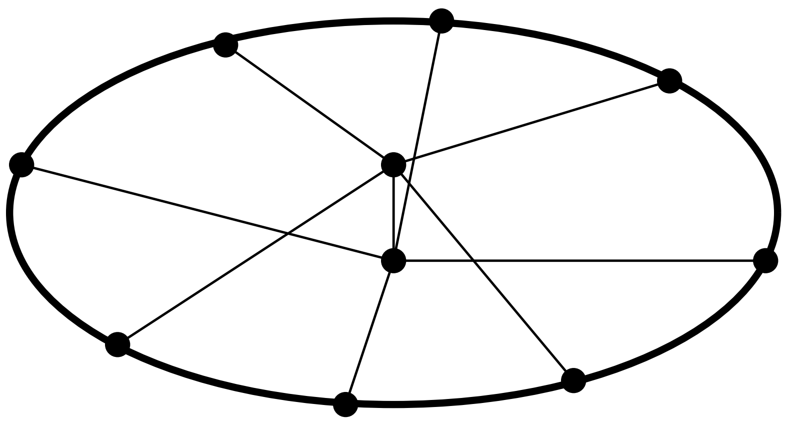

Figure 3: A 10-vertex bicycle wheel graph.

Family	n	d
RAND1	1000	16,32,64,125,250,500,999
RAND2	500,708,1000,1414,2000,2828,4000,5658,8000	50

In RAND1, we hold the number of vertices constant and test the affect of varying the degree. In RAND2, we hold the degree constant, and test the affect of varying the number of vertices. The number of vertices in each instance is roughly $\sqrt{2}$ times the number of vertices in the previous instance.

8.4 BIKEWHEELGEN generator

The BIKEWHEELGEN generator produces a graph that looks like a bicycle wheel. It takes a parameter n and generates an n-vertex, $(2n-3)$-edge graph consisting of a "rim" cycle on vertices $1, \ldots n-2$ plus two additional vertices, $n-1$ and n. There are $n-1$ "spoke" edges of the form $\{1, n-1\}, \{2, n\}, \{3, n-1\} \ldots, \{n-2, n\}$. There is also an "axis" edge $\{n-1, n\}$. If you draw vertices $n-1$ and n in the middle of the cycle, the graph looks like a bicycle wheel (see Figure 3). The capacity of each rim edge is $n-1$, the capacity of each spoke edge is 2, and the capacity of the axis edge is also 2. These instances have n trivial minimum cuts of value $2(n-2)$. However, because of their uniform structure, these problems are hard for all codes in our study.

The values used are:

Family	n
BIKEWHE	$2^{10}, 2^{11}, 2^{12}, 2^{13}$

8.5 TSP and PRETSP instances

The TSP instances are subproblems generated by the TSP code of Applegate and Cook [3]. This is the state-of-the-art code for solving TSP problems exactly, and is based on the technique of *cutting planes*. The set of feasible traveling salesman tours in a given graph induces a convex polytope in a high-dimensional vector space. Cutting plane algorithms find the optimum tour by repeatedly solving a linear relaxation of an integer programming formulation of TSP and adding linear inequalities that cut off undesirable parts of the polytope until the optimum solution to the relaxed problem is integral. The inequalities that have been very useful are *subtour elimination constraints*, first introduced by Dantzig, Fulkerson and Johnson [12]. The problem of identifying a subtour elimination constraint can be rephrased as the problem of finding a minimum cut in a graph with real-valued edge weights. Thus, cutting plane algorithms for the traveling salesman problem must solve a large number of minimum cut problems (see [29] for a survey of the area). We obtained some of the minimum cut instances that were solved by Applegate and Cook. Table 2 gives a summary of these instances, including their "names" which correspond to the original TSP problems.

The Padberg-Rinaldi tests are extremely effective of the TSP instances, and so is our PR preprocessing. To study preprocessing effects, we introduce the PRETSP family. A problem of this family is obtained by taking a TSP problem and applying PR preprocessing with $\beta = 0$ (*i.e.*,, the PR passes are applied until no PR tests succeeds. PRETSP problem size is a lower bound on the size of the corresponding TSP problems after PR preprocessing, and this bound is usually close to the real size.

Problem number	Problem name	n	m	n'	m'
1	tsp.att532.x.1	532	787	20	38
2	tsp.d1291.x.1	1291	1942	88	185
3	tsp.fl1400.x.1	1400	2231	148	300
4	tsp.pr76.x.2	76	90	2	1
5	tsp.r15934.x.1	5934	7287	150	292
6	tsp.r15934.x.2	5934	7627	261	517
7	tsp.rl1323.x.1	1323	2169	113	221
8	tsp.rl1323.x.2	1323	2195	106	208
9	tsp.vm1084.x.1	1084	1252	19	36
10	tsp.vm1748.x.1	1748	2336	77	131
11	d15112.xo.19057	15112	19057	605	1162
12	pla33810.xo.38600	33810	38600	2	1
13	pla33810.xo.39367	33810	39367	2	1
14	pla33810.xo.39456	33810	39456	2	1
15	pla85900.xo.102596	85900	102596	2	1
16	pla85900.xo.102934	85900	102934	2	1
17	pla85900.xo.102988	85900	102988	52	90
18	usa13509.xo.15631	13509	15631	325	561
19	usa13509.xo.17048	13509	17048	477	920
20	usa13509.xo.17079	13509	17079	449	861
21	usa13509.xo.17111	13509	17111	454	886
22	usa13509.xo.17130	13509	17130	403	786
23	usa13509.xo.17156	13509	17156	532	1029
24	usa13509.xo.17156a	13509	17156	501	950
25	usa13509.xo.17183	13509	17183	492	946
26	usa13509.xo.17193	13509	17193	549	1072
27	usa13509.xo.17210	13509	17210	465	926
28	usa13509.xo.17303	13509	17303	542	1064
29	usa13509.xo.17358	13509	17358	573	1104
30	usa13509.xo.17375	13509	17375	476	945
31	usa13509.xo.17386	13509	17386	607	1150
32	usa13509.xo.17390	13509	17390	557	1091
33	usa13509.xo.17494	13509	17494	505	971

Table 2: Summary of TSP and PRETSP instances. n and n' is the number of nodes in the TSP and PRETSP instances, respectively. m and m' are the corresponding numbers of edges.

Remark. Given a TSP problem, the resulting PRETSP problem may depend on the ordering of the TSP problem nodes and edges and on implementation details of the PR passes.

Table 2 gives summary of the PRETSP instances. One can see how effective the PR tests are. The PRETSP problems are drastically smaller than the corresponding TSP problems; in several cases, the problem is completely solved by the tests.

The absence of PR preprocessing in an algorithm is a big disadvantage on TSP problems, and PRETSP problems compensate for this.

The TSP, and especially the PRETSP problem size is smaller than we would have liked and hence running times of the best algorithms are small. Unfortunately we were unable to obtain bigger TSP instances. The Traveling Salesman Problem is much harder than the minimum cut problem, and the size of problems that can be solved by the current cutting plane algorithms is limited.

8.6 PRGEN generator

Our PRGEN generator is an implementation of the generator used by Padberg and Rinaldi [35]. This generator produces two different types of graphs. The first type is a random graph with an expected density d. The second type is a random graph that consists of two components connected by "heavy" edges, with "light" edges going between the components, thus the minimum cut is very likely to separate the two components. The generator takes three parameters

- n - the number of vertices,
- d - the density (as a percentage),
- c - the type of graph to generate (1 or 2).

If $c = 1$, for each pair of vertices, with probability d, we include an edge with weight uniformly distributed in $[1, 100]$. If $c = 2$, we split the graph into two components, one containing vertices 1 through $n/2$ and the other containting vertices $n/2 + 1$ through n. Again, for each pair of vertices, we include an edge with probability d. If the two vertices are in the same component, the edge weight is chosen uniformly from $[1, 100n]$, but if the vertices are in different components, the edge weight is chosen uniformly from $[1, 100]$.

We used eight families, PR1–PR8, with parameter values given in Table 3. These values are precisely those used by Padberg and Rinaldi in their paper [35]. We use these problem

Family	n	d	c
PR1	100,200,300,400	2	1
PR2	100,200,300,400	10	1
PR3	100,200,300,400	50	1
PR4	100,200,300,400	100	1
PR5	100,200,300,400,500,1000,1500,2000	2	2
PR6	100,200,300,400	10	2
PR7	100,200,300,400	50	2
PR8	100,200,300,400	100	2

Table 3: PR families.

families to compare heuristics in `ho` with those in the Padber-Rinaldi code. See Section 10.3. We note that NOIGEN is a generalization of PRGEN, but we implement PRGEN to ensure that the graphs we generate are as similar as possible to those generated by Padberg and Rinaldi.

9 Experimental Setup

Our experiments were conducted on a DEC 2100 server with two 175-MHz Alpha processors, 320M of shared memory and 1M of cache per processor. Our codes, however, only made use of one processor. Our codes are written in C and compiled with the UNIX **gcc** compiler using the **O4** optimization option.

All our problem generators use random numbers, hence for each setting of the parameters for each generator, we actually generated 5 graphs. Our numbers for running time, etc. are all averages taken over those 5 inputs. We also report standard deviations sometimes.

Our timer resolution is 1/60 seconds. The reader should keep this in mind, especially when considering times that are close to this value.

As mentioned in Section 2.6, our algorithms do not actually output the minimum cut, or save the minimum cut in any special data structure. However, at the time the minimum cut is encountered they do have the minimum cut stored in some internal data structure from which it could easily be extracted.

For all codes, we keep track of the following quantities:

- total running time

- discovery time - the time at which the algorithm first encountered the minimum cut.
- edge scans - the number of times an edge was scanned. This is a basic unit of work that is common to all of the codes.

For those codes that perform PR tests, we also keep track of:

- preprocessing time - the time spent in the initial PR tests.
- initial PR - the number of initial PR tests that were succesful.
- internal PR - the number of internal PR tests that were succesful.

For each individual code, we keep track of problem specific quantities also.

For the `ni` family, we keep track of the following additional quantity:

- phases - the number of phases of the algorithm.

For the `ho` family, we keed track of the following additional quantities:

- st cuts - the number of s-t cut (max flow) computations performed.
- avg. size - the average number of vertices in an s-t cut problem
- 1 node layers - the number of times a frozen layer has exactly one node.
- excess contr - the number of excess contractions performed.

10 Experimental Results

In this section we present our experimental results. We study the following codes:

- NI implementations.
  - `hybrid`, the Nagamochi *et al.* [33] implementation of the Nagamochi-Ibaraki algorithm with a PR heuristic,
  - `ni`, our implementation of the Nagamochi-Ibaraki algorithm with all heuristics turned on.
  - `ni-nopr`, same as `ni` but with the PR heuristic turned off.

- HO implementations.
  - `ho`, our implementation of the Hao-Orlin algorithm with all heuristics turned on.
  - `ho-noprxs`, same as `ho` but with all heuristics turned off.
  - `ho-noxs`, same as `ho` but with excess detection turned off.
  - `ho-nopr`, same as `ho` but with the PR heuristic turned off.
- KS implementations.
  - `ks`, our implementation of the Karger-Stein algorithm with all heuristics turned on.
  - `ks-nopr`, same as `ks` but with PR heuristic turned off.
- K implementations
  - `k`, our heuristic implementation of the $O(n^2 \log n)$ version of Kargr's algorithm with PR heuristic.

For the last code, we use the version that picks $6 \log n$ trees, which is well below the theoretical bound. (See Section 7.) The theoretically justified version, however, was very slow, and `k` always produced correct answers in our tests.

First we study relative performance of the best previous code, `hybrid`, and our best implementations of the four algorithms, `ni`, `ho`, `ks`, and `k`. Next we study individual algorithms. For each algorithm, we compare our best code with implementations of the same algorithm with fewer or different heuristics. We use running times and operation counts to explain algorithm performance and to study performance effects of heuristics and data structures.

To get a better perspective on effectiveness of `ho`'s internal PR heuristic, we use the data from [35] to indirectly compare `ho` and `ho-noxs` with the Padberg-Rinaldi code `pr` [35]. See the end of Section 10.3 for details.

10.1 Relative Performance

The most robust code in our study is `ho`, but it does not dominate all other algorithms. The second best code is `ni`.

On NOI families, `ni` is the fastest code, with `ho` and `hybrid` slower by about a factor of two. The other two codes, `ks` and `k`, are slower, usually by at least an order of magnitude.

The latter code is usually, but not always, faster than the former. See Tables 4 through 10 and Figures 4 through 9 for a more detailed comparison.

On RAND families, `ni` is the best overall, with `ho` close second. The other codes perform significantly worse. Usually `hybrid` is faster than `k`, but on dense problems the latter code is faster. The slowest code, `ks`, loses to the fastest codes by over an order of magnitude. See Tables 11 and 12 and Figures 10 and 11 for more information.

The REG1 family data is interesting. (See Table 13 and Figure 12). PR preprocessing solves the smallest problem quickly, and all our codes are fast on this problem (a unit weight cycle). The problem, however, is nontrivial for `hybrid` which does no PR preprocessing. Performance of `ni` is nonmonotone in graph density. On dense problems, internal PR tests become very effective, and `ni` is the fastest, although `ho` is very close. On sparse problems, `ni` performs poorly. On these problems, `ho` is the fastest, in some cases by over an order of magnitude. Performance of `hybrid` is the worst overall. The second-worst performance is exhibited by `ks`. On sparse problems, `k` is the second best; on dense problems, it is the third best.

On REG2 and REG4 problems, `ho` is clearly the fastest, with other algorithms asymptotically slower. On REG4 `ni` shows its worst case $O(nm) = O(n^3)$ behavior. See Tables 12 and 16 and Figures 11 and 15 for more information.

Recall that REG3 problems are unions of two random cycles. On these problems, `k` works like Gabow's algorithm and its running time is almost linear. For `ks` the constant factors are large, but the asymptotic performance is the second best. Even on the largest problem, however, `ks` is significantly slower than `ho`. Other codes, `ho`, `ni`, and `hybrid`, exhibit growth that is at least quadratic in the problem's size. However, `ho` has very small constants and is the fastest on all problems except for the largest, where it is slower than `k`. The worst codes are `ni` and `hybrid`. The data appears in Table 15 and Figure 14.

The BIKEWHE problems (see Table 17 and Figure 16) are hard for all codes, which exhibit a roughly quadratic time growth. The constants are significantly smaller for `ho` and the worst for `k`. The latter code also did not run on larger problems because of excessive memory requirements.

On TSP problems, `ho` and `ni` are the fastest, as is shown in Tables 18, 19 and 20 and Figures 17, 18 and 19. PR preprocessing is very effective on these problems and takes significant portion of time for these codes. On PRETSP problems, however, `ho` is usually faster than `ni`; see Tables 21, 22, and 23 and Figures 20 and 21. The other codes are significantly slower on this family.

The more effective internal PR tests of `ni` are the reason why this algorithm is as fast as `ho` on the TSP family and slower on the PRETSP family. Because the preprocessing terminates while a relatively large number of PR tests may apply, the internal tests are likely to succeed.

10.2 Nagamochi-Ibaraki Algorithm

In this section we study implementations of the Nagamochi-Ibaraki algorithm. The corresponding data appears in Tables 52 through 71 and Figures 40 through 57. In general, the number of phases is the key to performance of this algorithm. PR heuristics are important because they reduce the number of phases. Data structures are important because time to execute a phase depends on them. When a small number of contractions take place in a phase (and the number of phases is large), our data structure are somewhat slower that `hybrid`'s because we need to examine and compact the whole graph after a phase. This work, however, is amortized by the phase and cannot cost too much. When the number of contractions during a phase is large, using compact contraction can take $\Theta(n^2)$ time, and our data structures guarantees $\Theta(m)$ time. Our data structure can be significantly faster on sparse graphs.

First we compare `hybrid`, the previous implementation of the Nagamochi-Ibaraki algorithm that was a starting point of our implementation, and our implementation `ni`. The two codes differ in data structures and PR heuristics. Recall that our code uses a combination of set-union and compact contraction, PR preprocessing, a version of source PR heuristic, and PR passes between phases. In contrast, `hybrid` uses compact contraction and a variant of the source PR heuristic (see Section 3). Note also that the two codes are written in different languages and in different programming styles.

Our code `ni` is more robust than `hybrid`. The former code is never significantly slower but sometimes it is significantly faster.

On NOI families, the two codes perform similarly, with `ni` faster by roughly a factor of two. These problems are easy for the Nagamochi-Ibaraki algorithm and the number of phases is small. Although `ni` usually has fewer phases than `hybrid` because of the PR heuristic, the time difference is not drastic. See Tables 52 through 58 and Figures 40 through 45 for more data.

On RAND families, `ni` is significantly faster, especially on dense RAND1 problems, where the difference is over an order of magnitude. The number of phases of `ni` is significantly smaller, as is shown in Table 59 and Figure 46.

Data for the REG1 family is shows several big distinctions between the two codes. The first problem is a unit-weight cycle. This problem is hard for `hybrid`, but easy for `ni`, where PR preprocessing solves the problem. On problems of moderate density, the two codes perform similarly. On dense problems, however, `ni` is faster by over an order of magnitude because of the use of the more effective PR heuristic. See Table 61 and Figure 48.

On small REG2 problems, PR tests of `ni` are much more effective and the code is faster. (See Table 62 and Figure 49.) For larger problems, the number of phases of `ni` is not much less then the that of `hybrid`, and the latter algorithm is somewhat faster.

On REG3 and REG4 problems the two codes behave similarly, as is shown in Tables 63 and 64 and Figures 50 and 51.

On BIKEWHE problems, the number of phases is large for both codes, although `ni` has fewer phases and performs somewhat better, as is shown in Table 65 and Figure 52.

Finally we consider TSP and PRETSP families. On TSP problems, `ni` is much faster – sometimes by over two orders of magnitude. PR preprocessing on TSP instances is very effective and responsible for a big fraction of the speedup. The PRETSP data shows that internal PR tests in `ni` are also effective. Small sizes of the PRETSP problems make it hard to determine how effective the internal tests are. See the data in Tables 66 through 71 and Figures 53 through 57.

Next we compare `ni` and `ni-nopr` and study the value of the `ni`'s PR heuristic.

On NOI families, the two codes perform similarly. The number of phases is small for `ni-nopr`, and even though the PR tests reduce the number of phases, the work involved in the tests is about the same as the work saved.

On RAND problems, PR tests are very helpful, and `ni` is faster than `ni-nopr`, especially on dense graphs. Preprocessing PR tests fail, but the internal tests are highly successful.

On REG1 problems, PR tests help for the first problem, where the preprocessing tests solve the problem, and on dense problems, where the internal tests drastically reduce the number of phases. On problems of moderate density, the tests are not as helpful, and `ni` is a little slower than `ni-nopr`.

On small REG2 problems, the PR tests are effective, but on large problems they are much less effective; `ni` is faster on the former and a little slower on the latter.

On REG3 and REG4 problems, the tests are not very effective, and `ni` is a little slower. On BIKEWHE problems, PR tests reduce the number of phases, but only by a factor of about three, and ni is a little faster.

On TSP problems, the PR heuristic is extremely effective and ni is much faster – sometimes by over two orders of magnitude. PR preprocessing on TSP instances is very effective and responsible for a big fraction of the speedup. The PRETSP data shows that internal PR tests in ni are also effective.

10.3 Hao-Orlin Algorithm

In this section we study performance of ho heuristics. The corresponding data appears in Tables 24 through 43 and Figures 22 through 39 First consider excess detection. This heuristic usually helps, but not drastically – the running times of ho and ho-nopr are usually better than the corresponding times for the variants with no excess detection, but usually the times are within a factor of two. The data suggests that excess contractions tend to increase the number of internal PR contractions. However, many more single-node layers occur in the absence of excess contractions, and the number of s-t cut computations does not increase too much. In addition, the average problem size decreases. A similar phenomena occurs with ho-nopr vs. ho-noprxs: one-node layers replace most of excess contractions.

Next we study the PR heuristics by comparing ho and ho-nopr. On NOI (except NOI5), RAND, REG, and BIKEWHE families the algorithms perform in a similar way. For large k in NOI5 problems, where PR preprocessing solves the problem, ho is somewhat faster.

Note that the REG3 data exhibits a large time variation.

On TSP problems, ho is significantly faster, often by over an order of magnitude. On PRETSP, the algorithm is a little slower, mostly because of the time spent on preprocessing, which is pointless on these problems.

Although the PR heuristic makes HO more robust, this is mostly due to its preprocessing phase. The value of our internal PR heuristic is less clear. Sometimes it helps, and sometimes hurts, although never by more then a factor of two in our tests. One of the reasons for this phenomena is the fact that the source PR tests are relatively expensive and because of amortization they are rarely applied.

A more effective strategy for internal PR tests may improve ho's performance. To better understand these tests, we compare ho and ho-noxs with the Padberg-Rinaldi code pr, which implements the Gomory-Hu algorithm and performs exhaustive PR tests after every maximum flow computation. We use the data from [35] and compare the number of heuristic contraction operations only.

We would like to make a few remarks on this experiment. First, the comparison is indirect. We compare the data from [35] with the results we obtained by implementing the generator described in that paper and using the same parameter values as in the paper. Our instances are probably not the same as [35], but the variances are low, making the operation count comparison valid. Second, comparing the running times in this context is meaningless. Our times are hundreds of times smaller. However, we think that the `pr` code would be much faster if its maximum flow subroutine (which is an implementation of the Sleator-Tarjan algorithm [38]) is replaced by a good implementation if the push-relabel method, such as that of [10]. Also, our hardware is faster than that used by Padberg and Rinaldi.

Tables 44—51 give data for the PR1–PR8 families. Recall that the number of s-t cut computations for `ho` and `ho-noxs` is $n-1$ minus the sum of the numbers contractions due to PR, excess detection, and single node layer heuristics.

The number of PR contractions in `ho` is less than or equal to that of `pr`; the only exception is the first problem in the PR1 family, where the difference is small (98 vs. 96). The PR heuristic of `pr` often seems to be much more effective than that of `ho`. However, `ho` has other heuristics, excess detection and single node layer heuristic, which are very effective. The combination of `ho` heuristics is often more effective than the `pr` heuristics.

To measure the effect of excess contraction, we compare `ho` and `pr` to `ho-noxs`. The latter usually performs significantly more s-t cut computations than the former. Also, most of the time, `ho-noxs` performs more s-t cut computations that `pr`, but not always and the difference is not as big.

Experiments with PR1—PR8 families give insight into the effectiveness of `ho`'s internal PR tests, but the question of whether the tests can be improved remains open. The data for `pr` suggests that the internal tests can be more effective and increase the number of PR contractions. However, this may come at the expense of excess and single node layer contractions.

Remark. If we use a push-relabel maximum flow subroutine in `pr`, we can use excess detection and improve performance.

10.4 Karger-Stein Algorithm

For the `ks` family, we developed two codes, `ks-nopr`, which does not use the PR heuristics, and `ks` which uses the heuristics. We found that in spite of the randomization in the algorithm, ks-nopr actually had a very predictable running time of $O(n^2)$. This is not surprising, as the algorithm, in order to output the minimum cut with high enough probability, must examine $O(n^2)$ cuts. For dense graphs this would be a competitive running time, but for sparse problems this is usually orders of magnitude slower than ho or ni. Thus we do not report the running times for ks-nopr.

We now focus on the performance of ks. We note that the tables show that ks performs many more PR tests than any of the algorithms, in fact it usually performs more PR tests than the number of nodes in the graph. This is not a paradox, it happens only because the algorithm does repeated trials and examines many cuts. As was mentioned in Section 6, we do not see much correlation between the number of internal PR tests and the running time of the algorithms. However, the initial PR tests are a tremendous help, as a reduction from n_0 nodes to n_1 nodes takes at most $n_0 \log n_0$ time, but reduces the remaining running time from $O(n_0^2)$ to $O(n_1^2)$. One can see this advantage throughout Tables 4 through 23 and Figures 4 through 21.

One particularly interesting item to look at is discovery times. In many cases ks, despite being significantly slower than ho or ni, has discovery times that are similar to those of the faster codes. However, there are cases where the discovery time of ks is larger than the run time of ni, and also is a large fraction of the total run time. For example, in the second to last problem of the NOI2 family, ks has a discovery time that is more than a fifth of its total time and almost 4 times worse than ni's total time. Thus on some problems it would be impossible to reduce the number of trials enough to beat the fastest codes without missing the minimum cut.

10.5 Karger's Algorithm

It is clear that our implementation k of Karger's algorithm would not be a wise choice for most applications. Recall, however, that Karger's algorithm consists of two main parts — finding a tree packing and finding the minimum cut that 2-respects a tree — and both could be implemented in dramatically different ways, so it is valuable to try to establish whether there is hope for a better implementation.

k's primary misfortune is that it always runs in its worst-case $\Theta(n^2 \log n)$ time bound. Even worse, it uses $\Omega(n^2)$ space, so it can't handle very large problems. The only exception to these misfortunes is the special case of unweighted graphs with small minimum cut where `k` turns into Gabow's $O(cm \log n)$ time algorithm. This special case accounts for many of the cases where performance is reasonable, most notably the REG3 family, but also RAND2 and the first few problems of RAND1, REG1, and REG4. The switchover can be seen in RAND1 and REG1, where times increase sharply as m increases, and then the rate of increase slows dramatically when the runtime becomes dependent primarily on the fixed n. Note that this is really a special case; simply multiplying all edge weights by n would be sufficient to destroy `k`'s good performance on REG3 and in general force it to always use $\Theta(n^2 \log n)$ time.

Naturally, the other place where `k`'s performance is reasonable is on dense graphs, because there $\Omega(n^2)$ time is required. Unfortunately, except when `ni` shows its worst-case $O(n^3)$ performance in REG4, `k` continues to lose to both `ho` and `ni` because of inferior constants.

So one avenue of improvement is to implement the more complicated $O(m \log^2 n)$ algorithm to find the minimum cut that 2-respects a tree. At least for large sparse graphs, this should significantly improve `k`'s running time. In particular, the BIKEWHE family has $O(n)$ edges but causes $O(n^2)$ behavior in `ho` and `ni`. Even with bad constants $O(n \log^3 n)$ should quickly overcome $O(n^2)$ as n grows. Even on moderate density graphs it may help just because it doesn't use $\Omega(n^2)$ space.

It may also be profitable to replace Gabow's algorithm with the packing algorithm of [36]. There are two reasons for this. First, as implemented Gabow's algorithm doesn't have good constants. `ho` beats Gabow's algorithm until the largest of the REG3 problems, even though they are a great case for the latter. Also, on many NOI family problems, `ni` takes less time total that the tree packing step in `k`. Most of the time the tree packing is not the bottleneck, but as much as half the time is spent there on some TSP problems, and the alternative packing algorithm also has the potential to give a tree that the minimum cut 1-respects. If it turns out that this happens often in practice, `k` would perform much better.

11 Concluding Remarks

Our study produced several efficient codes which improve previous state of the art. Some of the algorithms in our study have not been implemented before. We also introduce new or improved heuristics that substantially improve performance.

Our results confirm practical importance of the Padberg-Rinaldi heuristics. We introduce PR passes and show that they are an effective way to apply PR tests in the preprocessing stage of all the algorithms. The passes are also an important component of our internal PR heuristic for `ni`.

The combination of improved data structures, PR heuristics, and the previous work of Nagamochi *et al.* lead to an efficient implementation of the Nagamochi-Ibaraki algorithm. This implementation, `ni`, is more robust than the previous implementation `hybrid`.

The implementation `ho` of the Hao-Orlin algorithm is the most robust in our tests. This is due in part to previous work on implementations of push-relabel algorithms for the maximum flow problem. Even a simple implementation of the algorithm, `ho-noprxs`, can perform quite well with the addition of PR preprocessing. This makes `ho-noprxs` hard to improve upon. Our excess detection heuristic succeeds in improving upon `ho-noprxs`; the improvement is noticeable and consistent but not dramatic. The effectiveness of our internal PR heuristic is less clear. We leave open the question of weather it is possible to have a better internal PR heuristic for the Hao-Orlin algorithm.

Performance of `k` shows that this algorithm needs to be investigated further. The version we used is a heuristic because the number of trees used is far below the theoretical bound. Since `k` found the optimal solution in all our tests, it may be possible to improve the bound. The implementation may also benefit from sophisticated data structures and a different tree-packing subroutine.

Our implementation of the Karger-Stein algorithm does not perform as well as the best codes, but its performance is not bad when compared to the previous codes. This implementation may be useful in some contexts, for example if one would like to find all minimum cuts.

`ni` outperforms `ho` if the number of `ni` phases is very small, *i.e.*, when a phase contracts many nodes. This suggests a hybrid algorithm that, after the preprocessing, applies `ni` phases (with internal PR tests) while the number of nodes decreases by a constant factor after each phase, and then switches to `ho`. We tested this hybrid algorithm, and it proved to be more robust than `ni` and `ho`. In fact, this is the most robust implementation we currently have. This and other hybrid algorithms deserve further study.

12 Tables and Plots

12.1 Tables comparing different codes

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	300	22425	0.31	0.07	—	—	—	—	—	—	3	0.51	—	—
ho	300	22425	0.32	0.05	0.04	0.33	0.05	0.25	0	0.00	165	0.11	384908	0.08
ni	300	22425	0.18	0.02	0.03	0.57	0.04	0.26	0	0.00	240	0.04	202990	0.03
k	300	22425	3.03	0.04	0.03	0.38	0.05	0.17	0	0.00	0	0.00	1049549	0.07
ks	300	22425	5.12	0.06	0.03	0.32	0.04	0.22	0	0.00	2961	0.04	1051651	0.04
hybrid	400	39900	0.60	0.02	—	—	—	—	—	—	3	0.76	—	—
ho	400	39900	0.65	0.03	0.07	0.16	0.10	0.08	0	0.00	250	0.03	676746	0.01
ni	400	39900	0.39	0.03	0.05	0.42	0.10	0.08	0	0.00	326	0.04	362615	0.04
k	400	39900	5.78	0.03	0.06	0.29	0.09	0.11	0	0.00	0	0.00	1887817	0.07
ks	400	39900	10.28	0.03	0.05	0.34	0.10	0.08	0	0.00	3971	0.04	1921294	0.03
hybrid	500	62375	1.02	0.05	—	—	—	—	—	—	4	0.36	—	—
ho	500	62375	1.05	0.01	0.05	0.62	0.17	0.05	0	0.00	325	0.06	1028294	0.01
ni	500	62375	0.66	0.04	0.06	0.65	0.15	0.05	0	0.00	407	0.05	553674	0.02
k	500	62375	9.66	0.03	0.06	0.64	0.14	0.08	0	0.00	0	0.00	3017636	0.09
ks	500	62375	18.15	0.06	0.05	0.65	0.16	0.06	0	0.00	5252	0.07	3129604	0.05
hybrid	600	89850	1.61	0.05	—	—	—	—	—	—	5	0.28	—	—
ho	600	89850	1.60	0.01	0.15	0.45	0.25	0.05	0	0.00	406	0.03	1525886	0.02
ni	600	89850	0.97	0.01	0.14	0.42	0.23	0.02	0	0.00	507	0.03	802661	0.02
k	600	89850	14.51	0.10	0.14	0.46	0.21	0.04	0	0.00	0	0.00	4561222	0.23
ks	600	89850	30.53	0.03	0.13	0.48	0.23	0.02	0	0.00	6871	0.05	4868725	0.03
hybrid	700	122325	2.54	0.06	—	—	—	—	—	—	5	0.25	—	—
ho	700	122325	2.27	0.01	0.23	0.30	0.35	0.02	0	0.00	506	0.02	2101482	0.01
ni	700	122325	1.41	0.02	0.24	0.28	0.31	0.03	0	0.00	620	0.02	1132793	0.03
k	700	122325	19.78	0.02	0.23	0.24	0.29	0.03	0	0.00	0	0.00	5726697	0.07
ks	700	122325	45.04	0.02	0.20	0.25	0.32	0.02	0	0.00	8665	0.03	6915953	0.02
hybrid	800	159800	3.26	0.07	—	—	—	—	—	—	7	0.31	—	—
ho	800	159800	3.02	0.01	0.22	0.73	0.46	0.02	0	0.00	570	0.05	2729953	0.01
ni	800	159800	1.83	0.04	0.21	0.72	0.42	0.01	0	0.00	686	0.04	1448069	0.03
k	800	159800	26.47	0.03	0.21	0.72	0.39	0.02	0	0.00	0	0.00	8006861	0.09
ks	800	159800	59.34	0.04	0.19	0.75	0.41	0.02	0	0.00	9476	0.06	8831240	0.04
hybrid	900	202275	4.54	0.05	—	—	—	—	—	—	7	0.09	—	—
ho	900	202275	3.91	0.01	0.28	0.71	0.58	0.01	0	0.00	664	0.02	3483609	0.01
ni	900	202275	2.36	0.02	0.28	0.65	0.53	0.01	0	0.00	802	0.02	1847438	0.01
k	900	202275	34.42	0.11	0.28	0.66	0.50	0.02	0	0.00	0	0.00	10396242	0.25
ks	900	202275	79.20	0.02	0.25	0.68	0.53	0.02	0	0.00	11206	0.04	11478872	0.02
hybrid	1000	249750	5.94	0.03	—	—	—	—	—	—	5	0.37	—	—
ho	1000	249750	4.93	0.01	0.30	0.60	0.73	0.01	0	0.00	759	0.01	4284685	0.01
ni	1000	249750	3.02	0.01	0.29	0.61	0.66	0.01	0	0.00	902	0.01	2355983	0.01
k	1000	249750	42.30	0.02	0.30	0.57	0.62	0.01	0	0.00	0	0.00	12173350	0.06
ks	1000	249750	104.34	0.01	0.26	0.60	0.66	0.02	0	0.00	13227	0.01	14845424	0.01

Table 4: NOI1 family

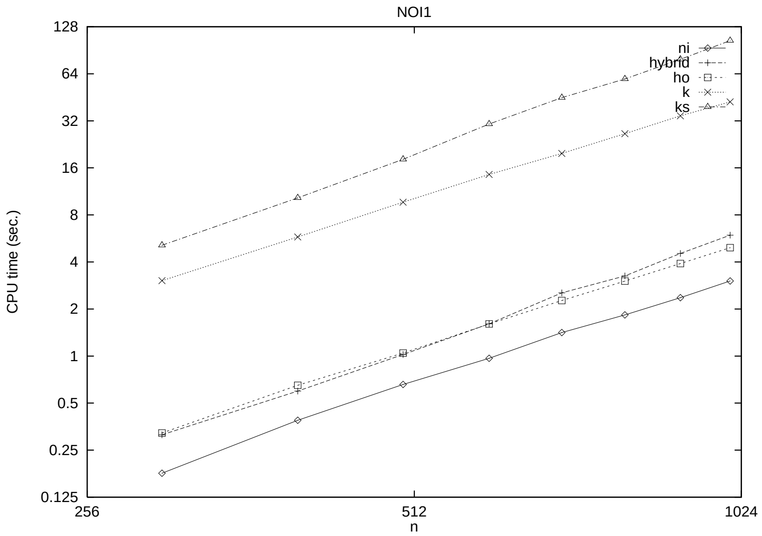

Figure 4: Different algorithms on NOI1 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	300	22425	0.27	0.03	—	—	—	—	—	—	13	1.20	—	—
ho	300	22425	0.26	0.09	0.21	0.11	0.05	0.17	0	0.00	72	0.23	316490	0.12
ni	300	22425	0.17	0.00	0.10	0.00	0.05	0.00	0	0.00	166	0.08	184785	0.01
k	300	22425	3.78	0.33	1.11	0.43	0.05	0.16	0	0.00	0	0.00	1154448	0.27
ks	300	22425	2.76	0.06	0.44	0.42	0.04	0.24	0	0.00	2499	0.04	606378	0.03
hybrid	400	39900	0.53	0.02	—	—	—	—	—	—	22	0.85	—	—
ho	400	39900	0.56	0.13	0.48	0.16	0.10	0.08	0	0.00	87	0.24	563229	0.17
ni	400	39900	0.38	0.05	0.23	0.09	0.10	0.16	0	0.00	230	0.06	329887	0.01
k	400	39900	6.51	0.07	1.91	0.24	0.08	0.10	0	0.00	0	0.00	1965190	0.11
ks	400	39900	5.59	0.05	0.91	0.58	0.10	0.09	0	0.00	3324	0.02	1089172	0.04
hybrid	500	62375	0.92	0.03	—	—	—	—	—	—	35	0.84	—	—
ho	500	62375	1.06	0.12	0.72	0.07	0.16	0.08	0	0.00	125	0.27	1015152	0.14
ni	500	62375	0.63	0.04	0.36	0.05	0.15	0.05	0	0.00	302	0.05	521470	0.01
k	500	62375	10.65	0.07	2.78	0.32	0.14	0.07	0	0.00	0	0.00	3100515	0.09
ks	500	62375	9.98	0.05	1.68	0.60	0.15	0.05	0	0.00	4286	0.03	1760832	0.05
hybrid	600	89850	1.37	0.02	—	—	—	—	—	—	28	0.64	—	—
ho	600	89850	1.62	0.05	1.20	0.16	0.25	0.00	0	0.00	163	0.28	1518734	0.05
ni	600	89850	0.92	0.02	0.53	0.02	0.22	0.02	0	0.00	357	0.03	745424	0.00
k	600	89850	16.01	0.08	3.33	0.33	0.21	0.05	0	0.00	0	0.00	4588094	0.10
ks	600	89850	15.70	0.03	2.21	0.46	0.23	0.02	0	0.00	5283	0.02	2578238	0.03
hybrid	700	122325	2.01	0.02	—	—	—	—	—	—	71	0.21	—	—
ho	700	122325	2.21	0.04	1.70	0.13	0.35	0.00	0	0.00	227	0.21	2011692	0.05
ni	700	122325	1.31	0.02	0.75	0.03	0.31	0.04	0	0.00	436	0.02	1021707	0.01
k	700	122325	22.81	0.13	4.55	0.32	0.29	0.04	0	0.00	0	0.00	6413031	0.20
ks	700	122325	22.65	0.06	3.92	0.67	0.32	0.02	0	0.00	6151	0.02	3595053	0.05
hybrid	800	159800	2.73	0.01	—	—	—	—	—	—	90	0.13	—	—
ho	800	159800	3.41	0.07	2.64	0.19	0.46	0.03	0	0.00	183	0.33	3068408	0.08
ni	800	159800	1.73	0.01	0.97	0.02	0.42	0.03	0	0.00	506	0.01	1339026	0.00
k	800	159800	28.28	0.06	5.86	0.29	0.38	0.02	0	0.00	0	0.00	8001240	0.09
ks	800	159800	30.37	0.05	5.45	0.63	0.42	0.02	0	0.00	6982	0.02	4701328	0.05
hybrid	900	202275	3.59	0.02	—	—	—	—	—	—	115	0.11	—	—
ho	900	202275	3.91	0.06	2.87	0.11	0.58	0.02	0	0.00	311	0.21	3399594	0.08
ni	900	202275	2.24	0.02	1.24	0.02	0.54	0.04	0	0.00	582	0.03	1702206	0.01
k	900	202275	37.98	0.12	7.57	0.36	0.50	0.01	0	0.00	0	0.00	10639215	0.20
ks	900	202275	39.71	0.05	8.44	0.43	0.53	0.03	0	0.00	8035	0.02	6081654	0.04
hybrid	1000	249750	4.58	0.02	—	—	—	—	—	—	132	0.07	—	—
ho	1000	249750	5.66	0.08	3.81	0.17	0.73	0.01	0	0.00	238	0.31	4855050	0.09
ni	1000	249750	2.77	0.01	1.54	0.01	0.66	0.03	0	0.00	652	0.02	2102361	0.00
k	1000	249750	49.50	0.22	9.47	0.40	0.63	0.01	0	0.00	0	0.00	14227165	0.32
ks	1000	249750	50.02	0.05	7.75	0.67	0.66	0.02	0	0.00	9226	0.02	7575644	0.05

Table 5: NOI2 family

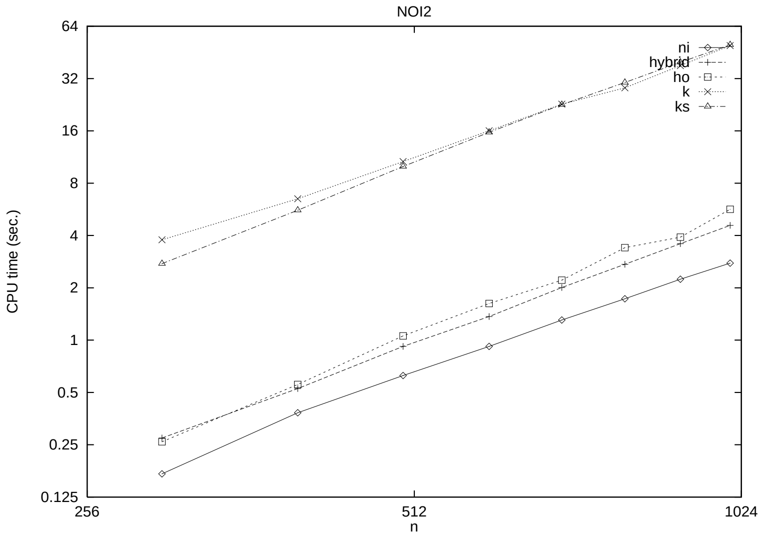

Figure 5: Different algorithms on NOI2 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	1000	24975	0.85	0.03	—	—	—	—	—	—	0	2.00	—	—
ho	1000	24975	0.43	0.05	0.02	0.73	0.08	0.05	0	0.00	309	0.05	505140	0.05
ni	1000	24975	0.24	0.04	0.02	0.85	0.07	0.00	0	0.00	490	0.08	249472	0.01
k	1000	24975	33.42	0.02	0.02	0.66	0.07	0.06	0	0.00	0	0.00	1933383	0.09
ks	1000	24975	4.82	0.08	0.02	0.86	0.07	0.05	0	0.00	10003	0.06	1039062	0.07
hybrid	1000	49950	1.36	0.02	—	—	—	—	—	—	1	0.31	—	—
ho	1000	49950	1.02	0.12	0.07	0.07	0.16	0.06	0	0.00	386	0.50	1113648	0.13
ni	1000	49950	0.55	0.03	0.08	0.12	0.15	0.05	0	0.00	667	0.06	504398	0.01
k	1000	49950	34.96	0.02	0.07	0.07	0.14	0.07	0	0.00	0	0.00	3524085	0.09
ks	1000	49950	16.80	0.12	0.07	0.06	0.14	0.06	0	0.00	9812	0.05	2997651	0.10
hybrid	1000	124875	3.02	0.03	—	—	—	—	—	—	4	0.37	—	—
ho	1000	124875	3.24	0.02	0.10	0.67	0.38	0.02	0	0.00	590	0.03	3261326	0.01
ni	1000	124875	1.50	0.02	0.12	0.59	0.35	0.00	0	0.00	821	0.02	1238518	0.02
k	1000	124875	38.15	0.02	0.11	0.63	0.33	0.02	0	0.00	0	0.00	7487131	0.05
ks	1000	124875	56.57	0.02	0.09	0.66	0.35	0.02	0	0.00	11361	0.02	8492396	0.01
hybrid	1000	249750	5.96	0.03	—	—	—	—	—	—	5	0.37	—	—
ho	1000	249750	4.95	0.02	0.31	0.54	0.73	0.01	0	0.00	759	0.01	4284685	0.01
ni	1000	249750	3.04	0.01	0.29	0.61	0.66	0.03	0	0.00	902	0.01	2355983	0.01
k	1000	249750	42.33	0.02	0.30	0.56	0.62	0.01	0	0.00	0	0.00	12173350	0.06
ks	1000	249750	104.91	0.01	0.27	0.60	0.66	0.02	0	0.00	13227	0.01	14845424	0.01
hybrid	1000	374625	8.74	0.03	—	—	—	—	—	—	6	0.43	—	—
ho	1000	374625	7.93	0.04	0.73	0.30	1.02	0.01	0	0.00	794	0.01	6706222	0.06
ni	1000	374625	4.36	0.01	0.70	0.28	0.94	0.02	0	0.00	931	0.01	3128006	0.00
k	1000	374625	46.44	0.02	0.74	0.28	0.88	0.01	0	0.00	0	0.00	16330860	0.05
ks	1000	374625	138.11	0.01	0.63	0.30	0.93	0.01	0	0.00	14135	0.01	19416647	0.01
hybrid	1000	499500	11.46	0.06	—	—	—	—	—	—	10	0.19	—	—
ho	1000	499500	10.45	0.01	0.43	0.65	1.29	0.02	0	0.00	818	0.01	8340469	0.01
ni	1000	499500	5.69	0.02	0.41	0.66	1.17	0.01	0	0.00	945	0.01	3917042	0.02
k	1000	499500	49.06	0.02	0.45	0.66	1.09	0.01	0	0.00	0	0.00	19064714	0.06
ks	1000	499500	165.94	0.01	0.37	0.67	1.16	0.01	0	0.00	14568	0.01	22795939	0.01

Table 6: NOI3 family

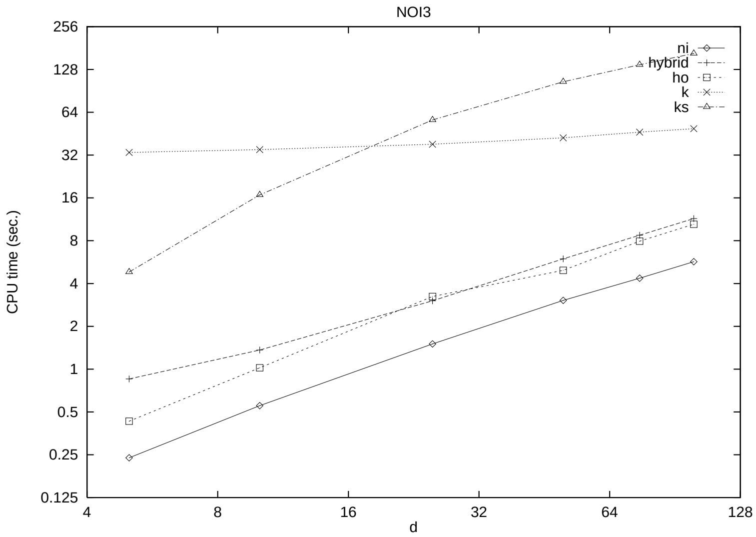

Figure 6: Different algorithms on NOI3 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	1000	24975	0.83	0.03	—	—	—	—	—	—	0	2.00	—	—
ho	1000	24975	0.38	0.03	0.03	0.92	0.08	0.05	0	0.00	63	0.25	419805	0.05
ni	1000	24975	0.23	0.03	0.03	0.86	0.07	0.06	0	0.00	220	0.23	237406	0.01
k	1000	24975	36.23	0.04	0.03	0.55	0.07	0.06	0	0.00	0	0.00	2087650	0.11
ks	1000	24975	3.76	0.05	0.03	0.62	0.07	0.06	0	0.00	5319	0.23	809563	0.03
hybrid	1000	49950	1.28	0.01	—	—	—	—	—	—	0	0.50	—	—
ho	1000	49950	1.00	0.04	0.52	0.72	0.16	0.06	0	0.00	141	0.08	1010936	0.02
ni	1000	49950	0.52	0.01	0.19	0.63	0.13	0.06	0	0.00	429	0.09	476370	0.01
k	1000	49950	39.32	0.04	6.64	0.94	0.14	0.07	0	0.00	0	0.00	3982094	0.11
ks	1000	49950	9.96	0.05	0.91	1.03	0.14	0.07	0	0.00	8815	0.09	1793567	0.04
hybrid	1000	124875	2.70	0.04	—	—	—	—	—	—	4	0.40	—	—
ho	1000	124875	2.77	0.12	1.91	0.14	0.38	0.01	0	0.00	217	0.58	2589176	0.14
ni	1000	124875	1.42	0.02	0.77	0.03	0.35	0.00	0	0.00	589	0.06	1150995	0.01
k	1000	124875	41.66	0.07	8.07	0.39	0.33	0.02	0	0.00	0	0.00	7479035	0.11
ks	1000	124875	27.17	0.04	4.01	0.50	0.35	0.02	0	0.00	9247	0.02	4308804	0.03
hybrid	1000	249750	4.58	0.02	—	—	—	—	—	—	132	0.07	—	—
ho	1000	249750	5.75	0.07	3.88	0.17	0.74	0.02	0	0.00	238	0.31	4855050	0.09
ni	1000	249750	2.80	0.02	1.55	0.01	0.68	0.03	0	0.00	652	0.02	2102361	0.00
k	1000	249750	49.41	0.22	9.46	0.40	0.62	0.01	0	0.00	0	0.00	14227165	0.32
ks	1000	249750	50.07	0.05	7.77	0.67	0.66	0.01	0	0.00	9226	0.02	7575644	0.05
hybrid	1000	374625	6.17	0.01	—	—	—	—	—	—	0	1.46	—	—
ho	1000	374625	8.18	0.05	5.47	0.27	1.04	0.02	0	0.00	370	0.10	6626486	0.07
ni	1000	374625	4.09	0.01	2.31	0.01	0.94	0.02	0	0.00	687	0.00	2889799	0.00
k	1000	374625	52.83	0.17	10.98	0.44	0.87	0.01	0	0.00	0	0.00	18183977	0.27
ks	1000	374625	67.73	0.05	11.66	0.58	0.93	0.01	0	0.00	9305	0.02	10092219	0.05
hybrid	1000	499500	7.19	0.01	—	—	—	—	—	—	0	0.00	—	—
ho	1000	499500	10.53	0.07	7.45	0.24	1.29	0.02	0	0.00	282	0.34	8181626	0.08
ni	1000	499500	5.26	0.01	3.07	0.01	1.17	0.01	0	0.00	691	0.01	3519347	0.00
k	1000	499500	51.88	0.24	13.20	0.30	1.10	0.01	0	0.00	60	2.71	19610160	0.28
ks	1000	499500	81.76	0.08	13.70	0.91	1.17	0.01	0	0.00	9244	0.03	11924189	0.07

Table 7: NOI4 family

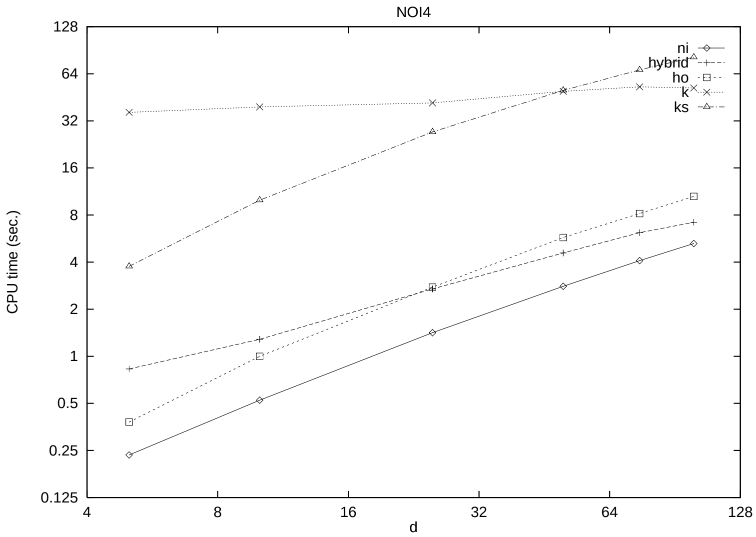

Figure 7: Different algorithms on NOI4 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	1000	249750	5.95	0.03	—	—	—	—	—	—	5	0.37	—	—
ho	1000	249750	4.88	0.01	0.29	0.56	0.72	0.01	0	0.00	759	0.01	4284685	0.01
ni	1000	249750	3.02	0.01	0.28	0.62	0.66	0.01	0	0.00	902	0.01	2355983	0.01
k	1000	249750	42.30	0.02	0.30	0.57	0.62	0.01	0	0.00	0	0.00	12173350	0.06
ks	1000	249750	104.82	0.01	0.26	0.60	0.66	0.02	0	0.00	13227	0.01	14845424	0.01
hybrid	1000	249750	4.55	0.02	—	—	—	—	—	—	132	0.07	—	—
ho	1000	249750	5.60	0.08	3.77	0.17	0.72	0.01	0	0.00	238	0.31	4855050	0.09
ni	1000	249750	2.76	0.01	1.53	0.00	0.66	0.01	0	0.00	652	0.02	2102361	0.00
k	1000	249750	49.42	0.22	9.45	0.40	0.62	0.01	0	0.00	0	0.00	14227165	0.32
ks	1000	249750	50.10	0.05	7.78	0.67	0.66	0.02	0	0.00	9226	0.02	7575644	0.05
hybrid	1000	249750	4.45	0.03	—	—	—	—	—	—	18	1.92	—	—
ho	1000	249750	5.60	0.15	3.38	0.51	0.73	0.01	0	0.00	198	0.57	4817520	0.18
ni	1000	249750	3.10	0.02	2.08	0.54	0.67	0.02	0	0.00	430	0.12	2344959	0.02
k	1000	249750	47.13	0.05	9.81	0.63	0.62	0.01	0	0.00	0	0.00	14280450	0.10
ks	1000	249750	64.32	0.07	6.52	0.85	0.66	0.01	0	0.00	11388	0.04	9756063	0.07
hybrid	1000	249750	4.41	0.02	—	—	—	—	—	—	0	0.00	—	—
ho	1000	249750	6.93	0.03	0.47	0.35	0.73	0.02	0	0.00	134	0.24	6065805	0.04
ni	1000	249750	3.12	0.02	0.45	0.38	0.67	0.03	0	0.00	245	0.07	2356937	0.02
k	1000	249750	48.68	0.04	0.45	0.34	0.63	0.01	0	0.00	0	0.00	14923220	0.06
ks	1000	249750	59.54	0.04	0.41	0.36	0.66	0.02	0	0.00	11920	0.02	9116181	0.04
hybrid	1000	249750	4.37	0.02	—	—	—	—	—	—	9	1.26	—	—
ho	1000	249750	7.78	0.06	0.33	0.43	0.73	0.02	0	0.00	106	0.39	6902801	0.06
ni	1000	249750	3.02	0.04	0.32	0.45	0.65	0.01	0	0.00	325	0.50	2276407	0.06
k	1000	249750	48.50	0.06	0.33	0.43	0.62	0.01	0	0.00	0	0.00	14749085	0.10
ks	1000	249750	55.93	0.05	0.29	0.47	0.67	0.01	0	0.00	11281	0.06	8458768	0.06
hybrid	1000	249750	4.18	0.01	—	—	—	—	—	—	6	1.54	—	—
ho	1000	249750	7.09	0.18	0.41	0.41	0.83	0.09	47	0.58	85	0.43	6134482	0.21
ni	1000	249750	2.58	0.02	0.39	0.44	0.75	0.08	47	0.58	355	0.15	1875425	0.03
k	1000	249750	44.43	0.11	0.40	0.40	1.08	0.48	58	0.62	0	0.00	14440045	0.09
ks	1000	249750	41.77	0.11	0.36	0.43	1.13	0.48	58	0.62	7547	0.19	6601313	0.07
hybrid	1000	249750	4.18	0.03	—	—	—	—	—	—	3	0.65	—	—
ho	1000	249750	5.78	0.50	0.32	0.79	0.92	0.05	235	1.62	28	0.81	4658492	0.61
ni	1000	249750	2.36	0.18	0.30	0.77	0.80	0.04	235	1.62	119	0.59	1588310	0.27
k	1000	249750	35.99	0.49	0.31	0.77	0.81	0.10	235	1.62	0	0.00	12226921	0.49
ks	1000	249750	29.15	0.49	0.28	0.79	0.81	0.05	235	1.62	4576	0.45	4746895	0.47
hybrid	1000	249750	4.20	0.01	—	—	—	—	—	—	1	0.85	—	—
ho	1000	249750	4.77	0.35	0.45	0.48	0.93	0.05	276	1.31	16	0.80	3574562	0.44
ni	1000	249750	2.29	0.17	0.42	0.47	0.82	0.05	276	1.31	66	0.52	1505651	0.26
k	1000	249750	28.83	0.52	0.43	0.48	1.71	0.27	335	1.01	0	0.00	10267490	0.49
ks	1000	249750	21.93	0.51	0.38	0.48	1.78	0.30	335	1.01	2671	0.04	4602031	0.48
hybrid	1000	249750	4.24	0.02	—	—	—	—	—	—	0	0.94	—	—
ho	1000	249750	2.75	0.71	0.25	0.51	1.05	0.10	810	0.46	12	2.00	1760322	1.07
ni	1000	249750	1.81	0.23	0.25	0.51	0.93	0.06	810	0.46	27	2.00	1012838	0.38
k	1000	249750	9.49	1.65	0.25	0.49	1.12	0.15	816	0.44	0	0.00	3404606	1.53
ks	1000	249750	8.45	1.64	0.22	0.51	1.06	0.21	816	0.44	4146	0.17	1520518	1.50
hybrid	1000	249750	4.21	0.01	—	—	—	—	—	—	0	1.22	—	—
ho	1000	249750	1.84	0.12	0.42	0.47	1.17	0.20	998	0.00	0	0.00	890688	0.25
ni	1000	249750	1.66	0.12	0.40	0.47	1.02	0.20	998	0.00	0	0.00	890687	0.25
k	1000	249750	1.76	0.11	0.41	0.46	1.12	0.17	998	0.00	0	0.00	890687	0.25
ks	1000	249750	1.61	0.13	0.37	0.48	1.02	0.20	998	0.00	4483	0.00	444391	0.51
hybrid	1000	249750	4.22	0.01	—	—	—	—	—	—	0	0.00	—	—
ho	1000	249750	1.69	0.01	0.25	0.34	1.01	0.02	998	0.00	0	0.00	748225	0.01
ni	1000	249750	1.55	0.01	0.24	0.34	0.89	0.02	998	0.00	0	0.00	748224	0.01
k	1000	249750	1.63	0.01	0.25	0.33	1.00	0.02	998	0.00	0	0.00	748224	0.01
ks	1000	249750	1.48	0.01	0.22	0.35	0.90	0.01	998	0.00	4483	0.00	301929	0.02

Table 8: NOI5 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	1000	249750	4.24	0.01	—	—	—	—	—	—	0	0.00	—	—
ho	1000	249750	1.68	0.01	0.49	0.21	1.02	0.00	998	0.00	0	0.00	758572	0.00
ni	1000	249750	1.55	0.01	0.48	0.21	0.91	0.01	998	0.00	0	0.00	758571	0.00
k	1000	249750	1.64	0.01	0.48	0.20	1.01	0.02	998	0.00	0	0.00	758571	0.00
ks	1000	249750	1.49	0.01	0.44	0.22	0.91	0.01	998	0.00	0	0.00	312275	0.01
hybrid	1000	249750	4.47	0.01	—	—	—	—	—	—	6	0.48	—	—
ho	1000	249750	1.77	0.00	0.31	0.43	1.11	0.01	998	0.00	0	0.00	872233	0.00
ni	1000	249750	1.63	0.01	0.30	0.47	0.99	0.02	998	0.00	0	0.00	872232	0.00
k	1000	249750	1.72	0.01	0.31	0.44	1.08	0.02	998	0.00	0	0.00	872232	0.00
ks	1000	249750	1.57	0.01	0.27	0.45	0.98	0.01	998	0.00	0	0.00	425936	0.01
hybrid	1000	249750	4.73	0.02	—	—	—	—	—	—	74	0.82	—	—
ho	1000	249750	1.97	0.01	0.34	0.49	1.32	0.01	998	0.00	0	0.00	1131721	0.01
ni	1000	249750	1.83	0.02	0.33	0.47	1.18	0.02	998	0.00	0	0.00	1131720	0.01
k	1000	249750	1.89	0.01	0.34	0.48	1.26	0.02	998	0.00	0	0.00	1131720	0.01
ks	1000	249750	1.76	0.01	0.30	0.49	1.17	0.01	998	0.00	0	0.00	685425	0.02
hybrid	1000	249750	4.93	0.01	—	—	—	—	—	—	0	2.00	—	—
ho	1000	249750	1.97	0.01	0.39	0.16	1.31	0.01	998	0.00	0	0.00	1128305	0.01
ni	1000	249750	1.80	0.00	0.39	0.14	1.15	0.00	998	0.00	0	0.00	1128304	0.01
k	1000	249750	1.90	0.01	0.40	0.14	1.26	0.01	998	0.00	0	0.00	1128304	0.01
ks	1000	249750	1.75	0.01	0.36	0.15	1.16	0.01	998	0.00	0	0.00	682009	0.01
hybrid	1000	249750	5.00	0.01	—	—	—	—	—	—	5	0.74	—	—
ho	1000	249750	2.01	0.01	0.48	0.16	1.34	0.01	998	0.00	0	0.00	1183977	0.01
ni	1000	249750	1.82	0.00	0.45	0.17	1.18	0.01	998	0.00	0	0.00	1183976	0.01
k	1000	249750	1.94	0.01	0.47	0.14	1.30	0.01	998	0.00	0	0.00	1183976	0.01
ks	1000	249750	1.78	0.01	0.41	0.16	1.19	0.01	998	0.00	0	0.00	737680	0.01
hybrid	1000	249750	5.14	0.03	—	—	—	—	—	—	10	0.70	—	—
ho	1000	249750	2.03	0.01	0.34	0.60	1.37	0.01	998	0.00	0	0.00	1215232	0.00
ni	1000	249750	1.84	0.01	0.34	0.56	1.20	0.00	998	0.00	0	0.00	1215231	0.00
k	1000	249750	1.95	0.01	0.34	0.57	1.31	0.01	998	0.00	0	0.00	1215231	0.00
ks	1000	249750	1.80	0.01	0.31	0.59	1.21	0.01	998	0.00	0	0.00	768936	0.01

Table 9: NOI5 family (continued)

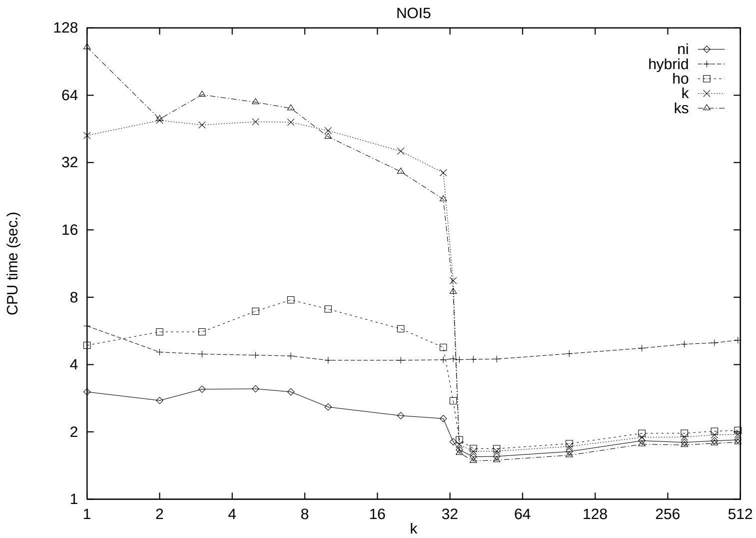

Figure 8: Different algorithms on NOI5 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	1000	249750	4.02	0.01	—	—	—	—	—	—	393	0.04	—	—
ho	1000	249750	4.90	0.09	4.04	0.14	0.72	0.01	0	0.00	25	0.05	4340237	0.09
ni	1000	249750	2.57	0.01	1.52	0.00	0.66	0.01	0	0.00	423	0.03	1954265	0.00
k	1000	249750	11.98	0.53	4.09	0.60	0.63	0.01	0	0.00	694	0.36	4604713	0.39
ks	1000	249750	32.94	0.02	4.91	0.14	0.66	0.02	0	0.00	254	0.47	6232298	0.01
hybrid	1000	249750	4.48	0.01	—	—	—	—	—	—	11	1.19	—	—
ho	1000	249750	4.74	0.07	3.69	0.11	0.72	0.01	0	0.00	117	0.23	4193474	0.08
ni	1000	249750	2.65	0.01	1.53	0.01	0.67	0.03	0	0.00	512	0.03	1999281	0.01
k	1000	249750	20.35	0.73	7.37	0.49	0.62	0.01	0	0.00	576	0.63	6173240	0.59
ks	1000	249750	36.84	0.05	6.18	0.38	0.67	0.01	0	0.00	2294	0.10	6497669	0.03
hybrid	1000	249750	4.57	0.01	—	—	—	—	—	—	132	0.07	—	—
ho	1000	249750	5.61	0.09	3.79	0.17	0.72	0.01	0	0.00	238	0.31	4855050	0.09
ni	1000	249750	2.77	0.02	1.53	0.01	0.68	0.02	0	0.00	652	0.02	2102361	0.00
k	1000	249750	49.41	0.22	9.46	0.40	0.62	0.01	0	0.00	0	0.00	14227165	0.32
ks	1000	249750	49.96	0.05	7.74	0.66	0.66	0.01	0	0.00	9226	0.02	7575644	0.05
hybrid	1000	249750	5.19	0.05	—	—	—	—	—	—	59	1.13	—	—
ho	1000	249750	6.64	0.13	0.35	0.55	0.73	0.02	0	0.00	265	0.23	6058229	0.17
ni	1000	249750	3.03	0.02	0.33	0.53	0.65	0.00	0	0.00	838	0.04	2326294	0.03
k	1000	249750	44.76	0.03	0.34	0.51	0.62	0.01	0	0.00	0	0.00	12979289	0.08
ks	1000	249750	97.33	0.03	0.31	0.51	0.66	0.02	0	0.00	11532	0.05	13775438	0.03
hybrid	1000	249750	5.23	0.06	—	—	—	—	—	—	76	0.82	—	—
ho	1000	249750	5.71	0.07	0.33	0.45	0.73	0.00	0	0.00	479	0.08	4980880	0.08
ni	1000	249750	3.02	0.02	0.31	0.46	0.66	0.01	0	0.00	848	0.04	2312736	0.02
k	1000	249750	44.15	0.02	0.33	0.44	0.62	0.01	0	0.00	0	0.00	12663329	0.06
ks	1000	249750	97.33	0.03	0.29	0.45	0.66	0.01	0	0.00	11729	0.05	13887649	0.03
hybrid	1000	249750	4.98	0.03	—	—	—	—	—	—	3	0.67	—	—
ho	1000	249750	5.17	0.09	0.35	0.60	0.73	0.01	0	0.00	628	0.06	4387209	0.09
ni	1000	249750	2.94	0.01	0.34	0.60	0.66	0.01	0	0.00	808	0.03	2246478	0.01
k	1000	249750	44.60	0.02	0.34	0.59	0.62	0.01	0	0.00	0	0.00	13390353	0.05
ks	1000	249750	93.03	0.03	0.31	0.61	0.66	0.02	0	0.00	11008	0.05	13387199	0.03
hybrid	1000	249750	5.12	0.03	—	—	—	—	—	—	3	0.47	—	—
ho	1000	249750	4.93	0.03	0.31	0.61	0.74	0.01	0	0.00	655	0.02	4155937	0.04
ni	1000	249750	2.94	0.01	0.29	0.59	0.66	0.03	0	0.00	832	0.02	2226590	0.01
k	1000	249750	43.66	0.03	0.30	0.57	0.62	0.01	0	0.00	0	0.00	12789773	0.09
ks	1000	249750	95.45	0.02	0.27	0.59	0.66	0.02	0	0.00	11343	0.04	13677285	0.02
hybrid	1000	249750	5.91	0.07	—	—	—	—	—	—	5	0.43	—	—
ho	1000	249750	4.92	0.02	0.44	0.37	0.72	0.02	0	0.00	677	0.03	4304370	0.02
ni	1000	249750	2.99	0.02	0.42	0.40	0.66	0.01	0	0.00	909	0.02	2325945	0.02
k	1000	249750	43.24	0.02	0.44	0.37	0.62	0.01	0	0.00	0	0.00	12916102	0.05
ks	1000	249750	98.38	0.02	0.39	0.38	0.66	0.02	0	0.00	11867	0.04	14057911	0.02
hybrid	1000	249750	5.85	0.05	—	—	—	—	—	—	7	0.42	—	—
ho	1000	249750	4.91	0.02	0.37	0.29	0.72	0.02	0	0.00	754	0.02	4287080	0.01
ni	1000	249750	2.96	0.02	0.36	0.26	0.67	0.03	0	0.00	892	0.02	2279187	0.02
k	1000	249750	42.45	0.03	0.37	0.26	0.62	0.01	0	0.00	0	0.00	12378874	0.08
ks	1000	249750	103.89	0.01	0.33	0.26	0.66	0.02	0	0.00	13089	0.03	14783121	0.01

Table 10: NOI6 family

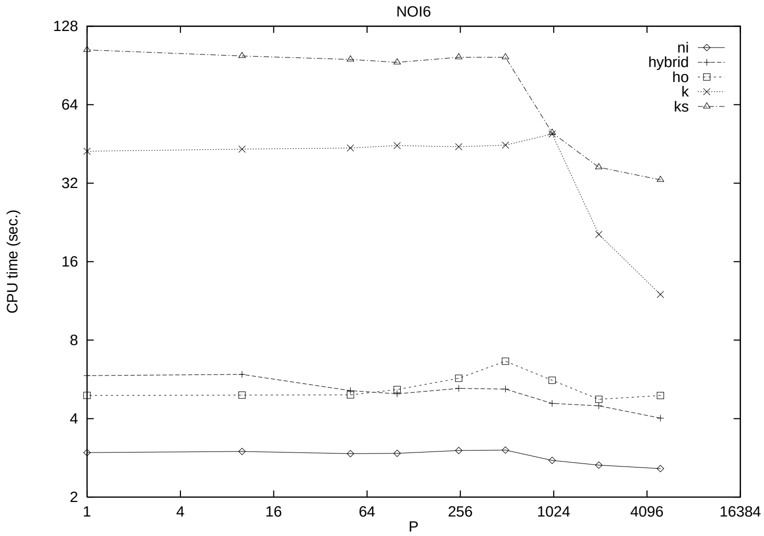

Figure 9: Different algorithms on NOI6 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	1000	8000	0.62	0.04	—	—	—	—	—	—	8	0.10	—	—
ho	1000	8000	0.18	0.06	0.00	0.00	0.01	1.22	0	0.00	226	0.77	351789	0.12
ni	1000	8000	0.13	0.00	0.01	1.22	0.01	0.82	0	0.00	867	0.01	223903	0.01
k	1000	8000	0.90	0.07	0.00	3.39	0.01	0.75	0	0.00	0	0.00	50277	0.00
ks	1000	8000	4.32	0.01	0.00	4.90	0.02	0.44	0	0.00	14769	0.01	924373	0.01
hybrid	1000	16000	1.82	0.35	—	—	—	—	—	—	15	0.10	—	—
ho	1000	16000	0.45	0.12	0.00	2.00	0.03	0.24	0	0.00	583	0.22	714757	0.13
ni	1000	16000	0.36	0.44	0.00	2.00	0.03	0.00	0	0.00	921	0.04	514597	0.47
k	1000	16000	3.16	0.13	0.01	1.22	0.05	0.14	0	0.00	0	0.00	97862	0.00
ks	1000	16000	8.87	0.02	0.01	1.60	0.03	0.07	0	0.00	15683	0.05	1797610	0.02
hybrid	1000	32000	6.51	0.48	—	—	—	—	—	—	29	0.13	—	—
ho	1000	32000	1.04	0.11	0.01	1.22	0.10	0.08	0	0.00	734	0.22	1293980	0.12
ni	1000	32000	1.22	0.49	0.01	1.22	0.09	0.11	0	0.00	962	0.01	1357742	0.52
k	1000	32000	12.95	0.12	0.01	1.37	0.09	0.11	0	0.00	0	0.00	190884	0.00
ks	1000	32000	18.81	0.01	0.01	1.36	0.09	0.11	0	0.00	16480	0.03	3422863	0.02
hybrid	1000	62500	10.26	0.37	—	—	—	—	—	—	42	0.17	—	—
ho	1000	62500	2.10	0.15	0.05	0.67	0.20	0.04	0	0.00	863	0.03	2413652	0.16
ni	1000	62500	1.63	0.17	0.06	0.56	0.18	0.02	0	0.00	973	0.01	1605205	0.19
k	1000	62500	34.21	0.07	0.05	0.52	0.17	0.02	0	0.00	0	0.00	3192230	0.18
ks	1000	62500	37.92	0.01	0.05	0.60	0.18	0.03	0	0.00	16737	0.02	6109943	0.01
hybrid	1000	125000	40.45	0.02	—	—	—	—	—	—	73	0.07	—	—
ho	1000	125000	3.98	0.27	0.06	0.52	0.38	0.01	0	0.00	898	0.06	4166085	0.30
ni	1000	125000	3.58	0.09	0.07	0.66	0.35	0.04	0	0.00	987	0.00	3317215	0.09
k	1000	125000	37.26	0.10	0.06	0.52	0.34	0.03	0	0.00	0	0.00	6653209	0.32
ks	1000	125000	72.32	0.01	0.05	0.52	0.35	0.02	0	0.00	17251	0.00	10730693	0.01
hybrid	1000	250000	72.80	0.01	—	—	—	—	—	—	87	0.04	—	—
ho	1000	250000	7.82	0.25	0.24	0.50	0.73	0.02	0	0.00	767	0.19	7375673	0.30
ni	1000	250000	5.35	0.19	0.24	0.46	0.69	0.02	0	0.00	990	0.00	4412996	0.22
k	1000	250000	42.13	0.11	0.25	0.46	0.62	0.01	0	0.00	0	0.00	11799446	0.30
ks	1000	250000	121.55	0.01	0.22	0.48	0.66	0.02	0	0.00	17306	0.01	17369038	0.01
hybrid	1000	499500	149.77	0.26	—	—	—	—	—	—	95	0.13	—	—
ho	1000	499500	11.41	0.14	0.40	0.92	1.28	0.01	0	0.00	972	0.01	9394579	0.19
ni	1000	499500	8.61	0.24	0.39	0.92	1.23	0.02	0	0.00	992	0.00	6151137	0.27
k	1000	499500	49.17	0.14	0.42	0.93	1.09	0.01	0	0.00	0	0.00	18850629	0.32
ks	1000	499500	183.88	0.01	0.35	0.93	1.17	0.01	0	0.00	17422	0.01	25494079	0.01

Table 11: RAND1 family

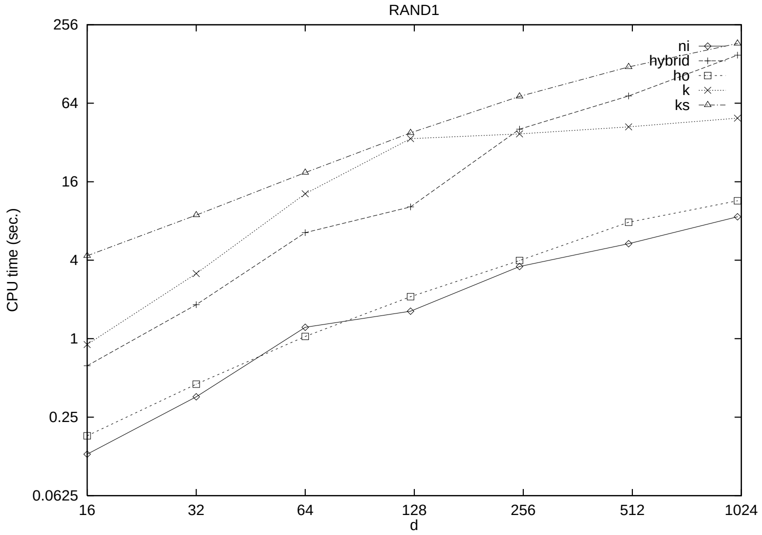

Figure 10: Different algorithms on RAND1 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	500	12500	1.79	0.32	—	—	—	—	—	—	16	0.08	—	—
ho	500	12500	0.24	0.11	0.00	2.00	0.02	0.18	0	0.00	361	0.10	519648	0.13
ni	500	12500	0.33	0.42	0.00	0.00	0.02	0.00	0	0.00	466	0.02	711803	0.43
k	500	12500	3.32	0.08	0.00	2.29	0.03	0.13	0	0.00	0	0.00	73782	0.00
ks	500	12500	5.56	0.01	0.00	2.71	0.02	0.38	0	0.00	7638	0.03	1185523	0.02
hybrid	708	17700	2.59	0.46	—	—	—	—	—	—	17	0.06	—	—
ho	708	17700	0.51	0.18	0.01	1.26	0.04	0.23	0	0.00	481	0.21	794567	0.18
ni	708	17700	0.48	0.52	0.02	0.50	0.03	0.00	0	0.00	674	0.01	664953	0.53
k	708	17700	4.28	0.14	0.01	1.13	0.05	0.19	0	0.00	0	0.00	105706	0.00
ks	708	17700	8.90	0.02	0.01	1.46	0.03	0.25	0	0.00	11074	0.03	1801538	0.02
hybrid	1000	25000	4.95	0.23	—	—	—	—	—	—	25	0.08	—	—
ho	1000	25000	0.80	0.17	0.01	1.22	0.07	0.00	0	0.00	695	0.16	1086410	0.18
ni	1000	25000	1.04	0.29	0.01	2.00	0.07	0.06	0	0.00	954	0.01	1218246	0.31
k	1000	25000	7.80	0.07	0.01	1.81	0.07	0.06	0	0.00	0	0.00	150505	0.00
ks	1000	25000	14.61	0.01	0.01	1.38	0.07	0.05	0	0.00	16551	0.02	2762359	0.01
hybrid	1414	35350	5.90	0.43	—	—	—	—	—	—	27	0.14	—	—
ho	1414	35350	1.40	0.15	0.01	0.86	0.11	0.09	0	0.00	989	0.11	1718330	0.15
ni	1414	35350	1.13	0.38	0.01	0.86	0.11	0.09	0	0.00	1353	0.01	1164292	0.41
k	1414	35350	11.00	0.06	0.02	0.66	0.10	0.08	0	0.00	0	0.00	214170	0.00
ks	1414	35350	23.62	0.01	0.02	0.58	0.10	0.07	0	0.00	23780	0.03	4108212	0.02
hybrid	2000	50000	11.44	0.40	—	—	—	—	—	—	33	0.19	—	—
ho	2000	50000	2.23	0.11	0.03	0.83	0.17	0.03	0	0.00	1366	0.13	2572105	0.11
ni	2000	50000	2.16	0.42	0.03	1.13	0.16	0.06	0	0.00	1900	0.01	2140653	0.45
k	2000	50000	18.03	0.10	0.02	1.29	0.16	0.06	0	0.00	0	0.00	304643	0.00
ks	2000	50000	37.22	0.01	0.02	1.31	0.16	0.06	0	0.00	34042	0.03	5985431	0.02
hybrid	2828	70700	23.94	0.02	—	—	—	—	—	—	44	0.06	—	—
ho	2828	70700	3.25	0.10	0.01	0.86	0.25	0.00	0	0.00	2090	0.11	3534336	0.11
ni	2828	70700	3.75	0.11	0.00	2.00	0.23	0.00	0	0.00	2707	0.01	3529980	0.12
k	2828	70700	28.62	0.07	0.01	1.36	0.23	0.00	0	0.00	0	0.00	431752	0.00
ks	2828	70700	60.33	0.01	0.00	2.00	0.23	0.01	0	0.00	50877	0.01	8985335	0.00
hybrid	4000	100000	33.40	0.30	—	—	—	—	—	—	49	0.27	—	—
ho	4000	100000	5.73	0.12	0.01	1.22	0.38	0.03	0	0.00	2743	0.13	5582265	0.12
ni	4000	100000	5.66	0.44	0.01	1.22	0.34	0.03	0	0.00	3798	0.02	5063980	0.47
k	4000	100000	42.27	0.08	0.01	0.89	0.34	0.03	0	0.00	0	0.00	611757	0.00
ks	4000	100000	93.58	0.01	0.01	1.13	0.34	0.03	0	0.00	73382	0.03	13232213	0.01
hybrid	5656	141400	52.90	0.22	—	—	—	—	—	—	58	0.18	—	—
ho	5656	141400	7.45	0.14	0.05	0.84	0.58	0.01	0	0.00	4405	0.08	6921169	0.14
ni	5656	141400	8.04	0.17	0.05	0.89	0.51	0.02	0	0.00	5419	0.01	6404503	0.18
k	5656	141400	66.55	0.08	0.04	0.88	0.52	0.01	0	0.00	0	0.00	866665	0.00
ks	5656	141400	142.90	0.01	0.04	0.92	0.52	0.01	0	0.00	108172	0.01	19593647	0.01
hybrid	8000	200000	95.02	0.01	—	—	—	—	—	—	70	0.03	—	—
ho	8000	200000	13.43	0.14	0.03	0.19	0.84	0.01	0	0.00	6289	0.06	11742261	0.14
ni	8000	200000	13.29	0.04	0.02	0.20	0.77	0.01	0	0.00	7699	0.00	9785313	0.04
k	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
ks	8000	200000	218.69	0.00	0.02	0.49	0.77	0.01	0	0.00	158969	0.00	28870838	0.00

Table 12: RAND2 family

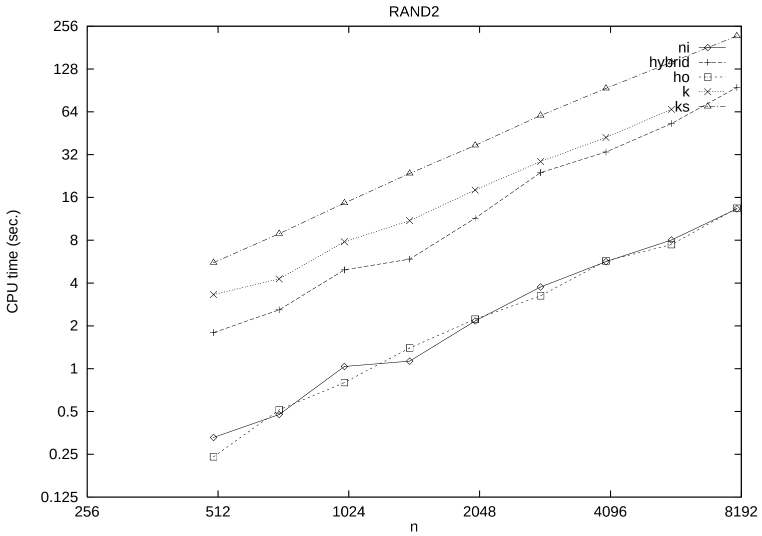

Figure 11: Different algorithms on RAND2 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	1001	1001	1.54	0.02	—	—	—	—	—	—	300	0.00	—	—
ho	1001	1001	0.00	0.00	0.00	0.00	0.00	0.00	999	0.00	0	0.00	5002	0.00
ni	1001	1001	0.01	1.22	0.00	0.00	0.00	0.00	999	0.00	0	0.00	5001	0.00
k	1001	1001	0.00	0.00	0.00	0.00	0.00	0.00	999	0.00	0	0.00	5001	0.00
ks	1001	1001	0.01	1.60	0.00	0.00	0.00	2.71	999	0.00	0	0.00	2999	0.00
hybrid	1001	3003	3.28	0.02	—	—	—	—	—	—	3	0.54	—	—
ho	1001	3003	0.14	0.13	0.00	0.00	0.02	0.50	0	0.00	635	0.05	205983	0.15
ni	1001	3003	3.04	0.02	0.00	0.00	0.00	2.00	0	0.00	411	0.16	4456979	0.02
k	1001	3003	0.35	0.10	0.00	0.00	0.01	1.13	0	0.00	0	0.00	19627	0.00
ks	1001	3003	2.55	0.03	0.00	0.00	0.00	2.00	0	0.00	17955	0.02	582574	0.02
hybrid	1001	10010	12.91	0.02	—	—	—	—	—	—	77	0.06	—	—
ho	1001	10010	0.38	0.14	0.00	0.00	0.02	0.00	0	0.00	617	0.12	809440	0.14
ni	1001	10010	10.02	0.02	0.00	0.00	0.02	0.50	0	0.00	594	0.12	19899570	0.02
k	1001	10010	1.93	0.10	0.00	0.00	0.02	0.15	0	0.00	0	0.00	62370	0.00
ks	1001	10010	5.94	0.03	0.00	0.00	0.02	0.38	0	0.00	18154	0.01	1296561	0.01
hybrid	1001	33033	37.21	0.02	—	—	—	—	—	—	156	0.02	—	—
ho	1001	33033	1.65	0.09	0.00	2.00	0.10	0.00	0	0.00	706	0.24	2192055	0.10
ni	1001	33033	51.66	0.01	0.00	0.00	0.08	0.10	0	0.00	681	0.01	61711219	0.01
k	1001	33033	16.23	0.07	0.00	4.90	0.09	0.11	0	0.00	0	0.00	196988	0.00
ks	1001	33033	20.51	0.01	0.00	0.00	0.09	0.11	0	0.00	17635	0.01	3697457	0.01
hybrid	1001	100100	111.02	0.01	—	—	—	—	—	—	195	0.01	—	—
ho	1001	100100	4.29	0.12	0.00	2.00	0.31	0.03	0	0.00	629	0.37	4790970	0.11
ni	1001	100100	151.94	0.00	0.00	0.00	0.28	0.00	0	0.00	710	0.00	166466428	0.00
k	1001	100100	46.46	0.68	0.00	4.90	0.27	0.02	0	0.00	0	0.00	4320291	0.32
ks	1001	100100	61.81	0.01	0.00	4.90	0.28	0.01	0	0.00	17761	0.01	9307883	0.01
hybrid	1001	333333	289.28	0.01	—	—	—	—	—	—	196	0.01	—	—
ho	1001	333333	12.26	0.07	0.00	0.00	0.95	0.02	0	0.00	883	0.22	11752797	0.07
ni	1001	333333	12.65	0.07	0.00	0.00	0.85	0.01	0	0.00	993	0.00	11099620	0.07
k	1001	333333	47.28	0.18	0.00	2.71	0.80	0.01	0	0.00	0	0.00	16383614	0.41
ks	1001	333333	148.00	0.01	0.00	4.90	0.85	0.01	0	0.00	17667	0.01	20992191	0.01
hybrid	1001	1001000	442.91	0.01	—	—	—	—	—	—	404	0.06	—	—
ho	1001	1001000	21.25	0.25	0.01	1.22	2.00	0.01	0	0.00	989	0.01	15623660	0.31
ni	1001	1001000	18.14	0.02	0.01	1.22	1.79	0.01	0	0.00	996	0.00	12777462	0.02
k	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
ks	1001	1001000	238.84	0.01	0.01	1.22	1.79	0.01	0	0.00	17538	0.01	32812382	0.01

Table 13: REG1 family

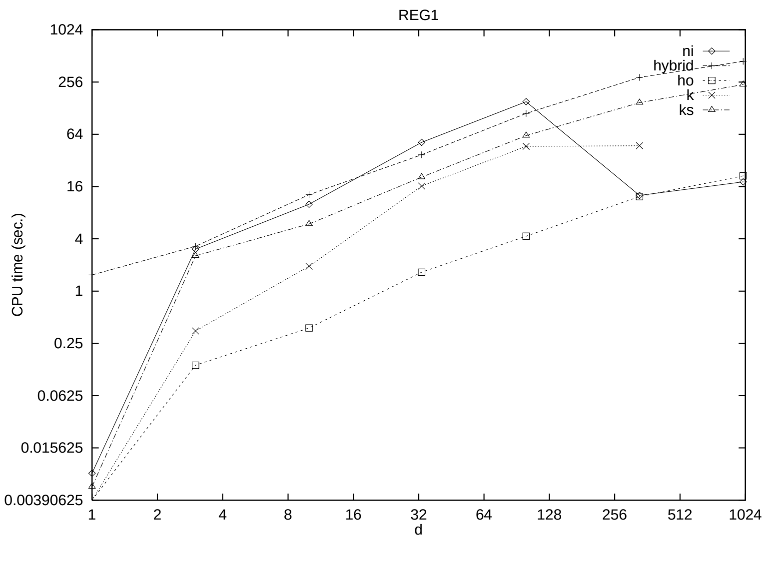

Figure 12: Different algorithms on REG1 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	50	2500	0.04	0.19	—	—	—	—	—	—	10	0.07	—	—
ho	50	2500	0.02	0.00	0.00	0.00	0.00	2.00	0	0.00	42	0.02	30955	0.05
ni	50	2500	0.02	0.00	0.00	0.00	0.00	2.00	0	0.00	46	0.00	20664	0.01
k	50	2500	0.13	1.05	0.00	0.00	0.00	2.29	0	0.00	0	0.00	38170	0.29
ks	50	2500	0.26	0.07	0.00	0.00	0.00	0.00	0	0.00	561	0.05	61049	0.04
hybrid	100	5000	0.25	0.03	—	—	—	—	—	—	14	0.07	—	—
ho	100	5000	0.05	0.36	0.00	0.00	0.01	1.22	0	0.00	72	0.21	112675	0.21
ni	100	5000	0.05	0.17	0.00	0.00	0.00	2.00	0	0.00	95	0.00	89037	0.03
k	100	5000	0.52	0.81	0.00	0.00	0.00	2.71	0	0.00	0	0.00	106738	0.44
ks	100	5000	0.92	0.04	0.00	0.00	0.01	1.60	0	0.00	1318	0.03	213474	0.04
hybrid	200	10000	1.34	0.05	—	—	—	—	—	—	32	0.03	—	—
ho	200	10000	0.17	0.11	0.00	0.00	0.02	0.50	0	0.00	154	0.19	354763	0.10
ni	200	10000	0.17	0.15	0.00	0.00	0.00	2.00	0	0.00	191	0.01	376816	0.17
k	200	10000	1.23	0.35	0.00	0.00	0.01	0.62	0	0.00	0	0.00	353150	0.18
ks	200	10000	2.80	0.03	0.00	4.90	0.01	1.06	0	0.00	2936	0.02	627613	0.02
hybrid	400	20000	6.64	0.03	—	—	—	—	—	—	66	0.01	—	—
ho	400	20000	0.56	0.11	0.00	0.00	0.05	0.15	0	0.00	291	0.17	872105	0.13
ni	400	20000	9.06	0.02	0.00	0.00	0.04	0.26	0	0.00	277	0.01	13649236	0.02
k	400	20000	4.93	0.16	0.00	3.39	0.04	0.23	0	0.00	0	0.00	865319	0.18
ks	400	20000	7.80	0.02	0.00	0.00	0.04	0.24	0	0.00	6205	0.02	1597809	0.02
hybrid	800	40000	33.25	0.03	—	—	—	—	—	—	139	0.02	—	—
ho	800	40000	2.00	0.17	0.00	0.00	0.12	0.03	0	0.00	646	0.08	2556624	0.18
ni	800	40000	48.19	0.01	0.00	0.00	0.10	0.00	0	0.00	557	0.00	56706079	0.01
k	800	40000	26.04	0.07	0.00	4.90	0.11	0.09	0	0.00	0	0.00	231376	0.00
ks	800	40000	22.34	0.01	0.00	0.00	0.11	0.09	0	0.00	13606	0.01	3931687	0.01

Table 14: REG2 family

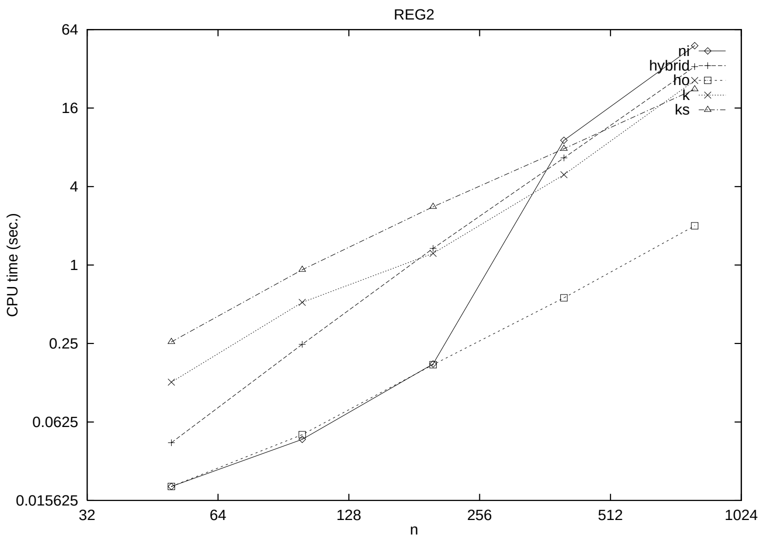

Figure 13: Different algorithms on REG2 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	256	512	0.13	0.11	—	—	—	—	—	—	1	0.73	—	—
ho	256	512	0.02	0.50	0.00	0.00	0.00	0.00	4	0.44	139	0.08	20739	0.16
ni	256	512	0.10	0.13	0.00	0.00	0.00	0.00	4	0.44	125	0.22	142274	0.07
k	256	512	0.04	0.24	0.00	0.00	0.00	0.00	4	0.44	0	0.00	3407	0.00
ks	256	512	0.38	0.06	0.00	0.00	0.00	0.00	4	0.44	4476	0.02	97943	0.04
hybrid	512	1024	0.45	0.04	—	—	—	—	—	—	2	0.42	—	—
ho	512	1024	0.04	0.26	0.00	0.00	0.00	0.00	4	0.27	318	0.13	51902	0.11
ni	512	1024	0.39	0.07	0.00	2.00	0.00	0.00	4	0.27	231	0.12	529756	0.08
k	512	1024	0.08	0.16	0.00	0.00	0.00	2.71	4	0.27	0	0.00	6831	0.00
ks	512	1024	0.91	0.04	0.00	0.00	0.00	2.29	4	0.27	9855	0.02	220931	0.02
hybrid	1024	2048	1.80	0.03	—	—	—	—	—	—	4	0.42	—	—
ho	1024	2048	0.09	0.23	0.00	0.00	0.01	0.82	6	0.32	651	0.03	124559	0.11
ni	1024	2048	1.46	0.05	0.00	0.00	0.00	0.00	6	0.32	358	0.22	1954614	0.05
k	1024	2048	0.22	0.13	0.00	0.00	0.00	4.90	6	0.32	0	0.00	13666	0.00
ks	1024	2048	2.18	0.02	0.00	0.00	0.01	0.69	6	0.32	21221	0.02	486707	0.01
hybrid	2048	4096	8.62	0.04	—	—	—	—	—	—	2	0.50	—	—
ho	2048	4096	0.31	0.29	0.00	0.00	0.01	0.82	6	0.30	1291	0.02	401943	0.30
ni	2048	4096	5.95	0.03	0.00	0.00	0.01	1.22	6	0.30	626	0.35	7326869	0.03
k	2048	4096	0.57	0.07	0.00	0.00	0.01	0.62	6	0.30	0	0.00	27313	0.00
ks	2048	4096	4.84	0.02	0.00	0.00	0.01	1.33	6	0.30	45040	0.01	1032079	0.00
hybrid	4096	8192	39.82	0.02	—	—	—	—	—	—	2	0.42	—	—
ho	4096	8192	1.04	0.16	0.00	0.00	0.03	0.14	5	0.36	2674	0.04	1177472	0.15
ni	4096	8192	26.06	0.02	0.00	0.00	0.03	0.00	5	0.36	972	0.32	28309175	0.03
k	4096	8192	1.62	0.05	0.00	0.00	0.03	0.00	5	0.36	0	0.00	54598	0.00
ks	4096	8192	11.42	0.00	0.00	0.00	0.02	0.20	5	0.36	98027	0.00	2242652	0.00
hybrid	8192	16384	177.96	0.01	—	—	—	—	—	—	1	0.31	—	—
ho	8192	16384	4.14	0.39	0.00	0.00	0.10	0.08	5	0.22	5400	0.02	3589940	0.40
ni	8192	16384	167.89	0.03	0.00	2.00	0.08	0.00	5	0.22	2275	0.25	105627355	0.02
k	8192	16384	4.03	0.04	0.00	4.90	0.08	0.10	5	0.22	0	0.00	109226	0.00
ks	8192	16384	29.19	0.00	0.00	0.00	0.09	0.10	5	0.22	210328	0.00	4806367	0.00
hybrid	16384	32768	843.45	0.00	—	—	—	—	—	—	1	1.10	—	—
ho	16384	32768	18.53	0.37	0.00	0.00	0.24	0.04	5	0.61	10651	0.07	13903247	0.38
ni	16384	32768	780.60	0.01	0.00	2.00	0.21	0.05	5	0.61	4665	0.28	392336486	0.01
k	16384	32768	9.51	0.02	0.00	4.90	0.20	0.03	5	0.61	0	0.00	218432	0.00
ks	16384	32768	73.87	0.00	0.00	0.00	0.21	0.06	5	0.61	434794	0.00	9938542	0.00

Table 15: REG3 family

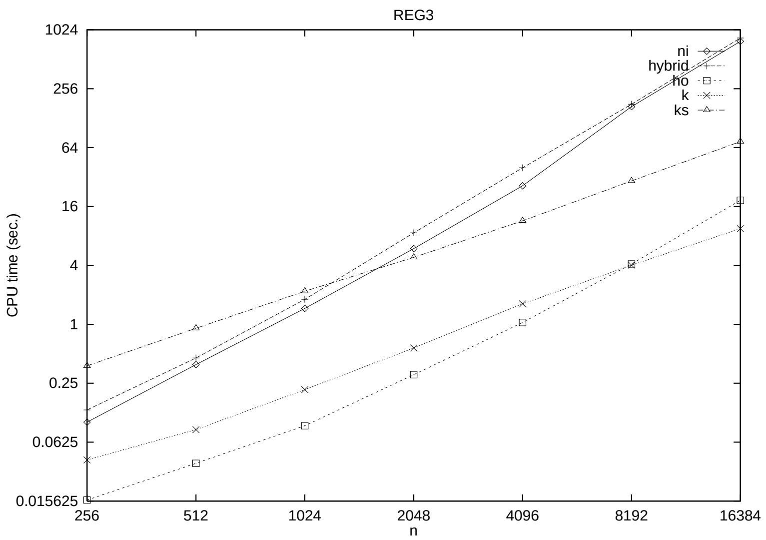

Figure 14: Different algorithms on REG3 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	128	1024	0.14	0.06	—	—	—	—	—	—	4	0.18	—	—
ho	128	1024	0.02	0.18	0.00	0.00	0.01	1.22	0	0.00	73	0.21	43691	0.04
ni	128	1024	0.12	0.08	0.00	0.00	0.00	2.00	0	0.00	79	0.02	268717	0.04
k	128	1024	0.07	0.11	0.00	0.00	0.00	2.71	0	0.00	0	0.00	6184	0.01
ks	128	1024	0.40	0.04	0.00	4.90	0.00	3.39	0	0.00	1868	0.03	99598	0.03
hybrid	256	4096	0.93	0.01	—	—	—	—	—	—	26	0.04	—	—
ho	256	4096	0.09	0.11	0.00	0.00	0.01	1.22	0	0.00	190	0.09	196194	0.08
ni	256	4096	0.87	0.01	0.00	0.00	0.01	1.22	0	0.00	168	0.01	2049661	0.01
k	256	4096	0.56	0.11	0.00	0.00	0.01	1.33	0	0.00	0	0.00	24191	0.00
ks	256	4096	1.58	0.03	0.00	0.00	0.00	2.29	0	0.00	3791	0.03	375841	0.03
hybrid	512	16384	7.73	0.02	—	—	—	—	—	—	76	0.02	—	—
ho	512	16384	0.48	0.06	0.00	0.00	0.03	0.00	0	0.00	366	0.23	837741	0.06
ni	512	16384	8.85	0.00	0.00	0.00	0.03	0.14	0	0.00	352	0.00	15173694	0.00
k	512	16384	5.14	0.09	0.00	4.90	0.04	0.26	0	0.00	0	0.00	95421	0.00
ks	512	16384	7.53	0.02	0.00	0.00	0.03	0.13	0	0.00	8353	0.02	1586117	0.02
hybrid	1024	65536	74.20	0.02	—	—	—	—	—	—	187	0.01	—	—
ho	1024	65536	2.94	0.14	0.00	2.00	0.21	0.05	0	0.00	920	0.05	3586320	0.16
ni	1024	65536	103.96	0.01	0.00	0.00	0.19	0.05	0	0.00	719	0.00	116991994	0.01
k	1024	65536	41.54	0.13	0.00	0.00	0.18	0.00	0	0.00	0	0.00	3111697	0.26
ks	1024	65536	41.50	0.01	0.00	4.90	0.19	0.05	0	0.00	18049	0.01	6654966	0.01
hybrid	2048	262144	725.92	0.01	—	—	—	—	—	—	418	0.01	—	—
ho	2048	262144	15.28	0.12	0.00	0.00	0.88	0.01	0	0.00	1834	0.10	16028877	0.14
ni	2048	262144	857.65	0.01	0.00	2.00	0.81	0.01	0	0.00	1466	0.00	896240114	0.01
k	2048	262144	205.99	0.05	0.00	3.39	0.81	0.01	0	0.00	0	0.00	15713419	0.30
ks	2048	262144	196.05	0.01	0.00	0.00	0.80	0.01	0	0.00	37532	0.00	27407217	0.00

Table 16: REG4 family

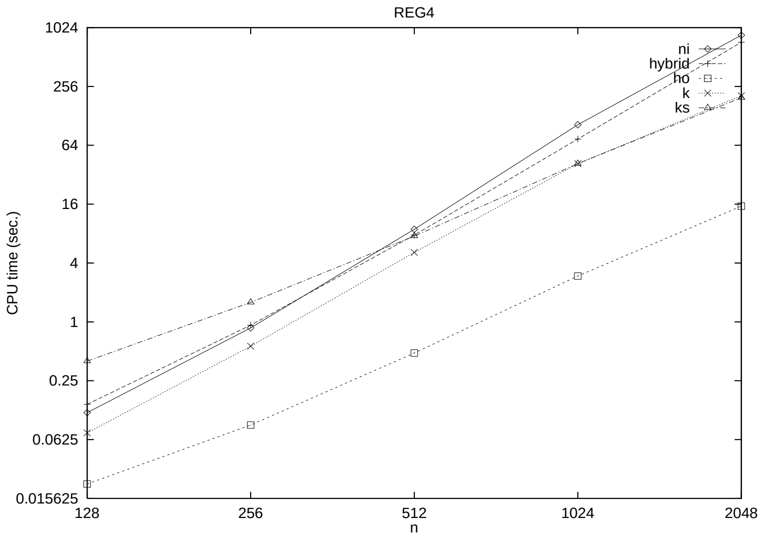

Figure 15: Different algorithms on REG4 family

	nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
hybrid	1024	2044	5.18	0.00	—	—	—	—	—	—	17	0.00	—	—
ho	1024	2044	0.26	0.05	0.00	0.00	0.00	0.00	0	0.00	9	0.00	578086	0.00
ni	1024	2044	1.73	0.01	0.00	0.00	0.00	0.00	0	0.00	764	0.00	2879946	0.00
k	1024	2044	43.49	0.15	0.00	4.90	0.00	3.39	0	0.00	0	0.00	329227	0.21
ks	1024	2044	14.10	0.05	0.00	0.00	0.00	3.39	0	0.00	122613	0.04	3942581	0.04
hybrid	2048	4092	23.08	0.00	—	—	—	—	—	—	107	0.00	—	—
ho	2048	4092	1.58	0.01	0.00	0.00	0.01	0.82	0	0.00	10	0.00	3141891	0.00
ni	2048	4092	7.11	0.00	0.00	0.00	0.01	0.82	0	0.00	1532	0.00	11480028	0.00
k	2048	4092	208.07	0.14	0.00	0.00	0.01	0.82	0	0.00	0	0.00	725144	0.23
ks	2048	4092	47.35	0.03	0.00	0.00	0.01	1.04	0	0.00	383795	0.03	13137386	0.03
hybrid	4096	8188	105.98	0.01	—	—	—	—	—	—	322	0.00	—	—
ho	4096	8188	6.85	0.01	0.00	0.00	0.03	0.00	0	0.00	11	0.00	11277803	0.00
ni	4096	8188	28.95	0.00	0.00	2.00	0.02	0.20	0	0.00	3067	0.00	45877881	0.00
k	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
ks	4096	8188	159.54	0.03	0.00	0.00	0.02	0.17	0	0.00	1206226	0.03	43201113	0.03
hybrid	8192	16380	510.13	0.00	—	—	—	—	—	—	682	0.00	—	—
ho	8192	16380	28.80	0.00	0.00	0.00	0.09	0.11	0	0.00	12	0.00	39571898	0.00
ni	8192	16380	201.12	0.00	0.00	0.00	0.07	0.00	0	0.00	6140	0.00	183242209	0.00
k	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL	FAIL
ks	8192	16380	588.84	0.02	0.00	0.00	0.08	0.08	0	0.00	4190341	0.02	155749909	0.01

Table 17: BIKEWHE family

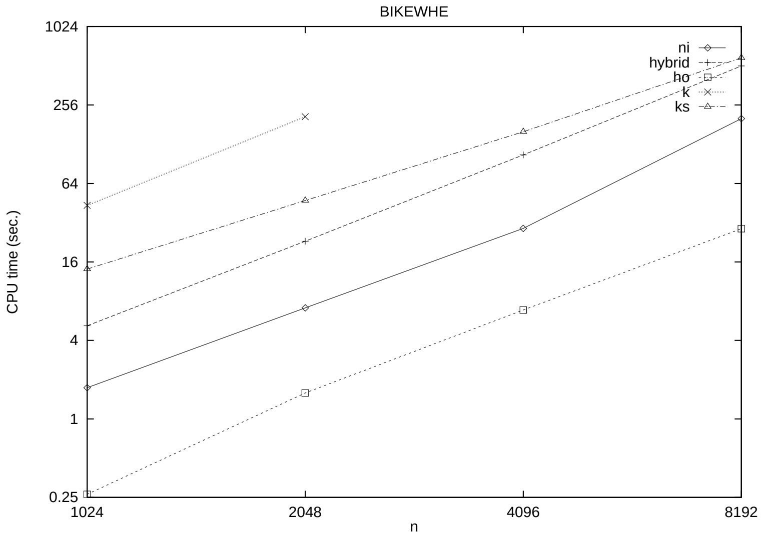

Figure 16: Different algorithms on BIKEWHE family

problem		nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
				avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
1	hybrid	532	787	0.60	0.00	—	—	—	—	—	—	136	0.00	—	—
1	ho	532	787	0.02	0.00	0.00	0.00	0.02	0.00	487	0.00	14	0.00	10052	0.00
1	ni	532	787	0.00	0.00	0.00	0.00	0.00	0.00	487	0.00	35	0.00	10042	0.00
1	k	532	787	0.02	0.00	0.00	0.00	0.00	0.00	512	0.00	0	0.00	11678	0.00
1	ks	532	787	0.02	0.20	0.00	0.00	0.00	2.00	512	0.00	208	0.04	12021	0.02
2	hybrid	1291	1942	3.57	0.00	—	—	—	—	—	—	326	0.00	—	—
2	ho	1291	1942	0.03	0.00	0.00	0.00	0.00	0.00	976	0.00	58	0.00	64710	0.00
2	ni	1291	1942	0.05	0.00	0.02	0.00	0.00	0.00	976	0.00	257	0.00	61047	0.00
2	k	1291	1942	1.27	0.00	0.00	0.00	0.02	0.00	1170	0.00	0	0.00	66011	0.00
2	ks	1291	1942	0.28	0.07	0.00	2.00	0.02	0.00	1170	0.00	2644	0.02	94575	0.02
3	hybrid	1400	2231	2.37	0.00	—	—	—	—	—	—	240	0.00	—	—
3	ho	1400	2231	0.07	0.00	0.05	0.00	0.02	0.00	1091	0.00	47	0.00	84281	0.00
3	ni	1400	2231	0.05	0.00	0.03	0.00	0.02	0.00	1091	0.00	238	0.00	64451	0.00
3	k	1400	2231	3.25	0.00	2.58	0.00	0.03	0.00	1213	0.00	0	0.00	97096	0.00
3	ks	1400	2231	0.35	0.04	0.09	0.70	0.02	0.20	1213	0.00	3578	0.02	117326	0.02
5	hybrid	5934	7287	20.93	0.00	—	—	—	—	—	—	787	0.00	—	—
5	ho	5934	7287	0.08	0.00	0.08	0.00	0.03	0.00	5651	0.00	35	0.00	93504	0.00
5	ni	5934	7287	0.07	0.00	0.05	0.00	0.03	0.00	5651	0.00	169	0.00	88043	0.00
5	k	5934	7287	1.18	0.00	0.07	0.00	0.07	0.00	5762	0.00	0	0.00	102417	0.00
5	ks	5934	7287	0.34	0.05	0.05	0.17	0.05	0.25	5762	0.00	2726	0.05	130473	0.02
6	hybrid	5934	7627	84.68	0.00	—	—	—	—	—	—	1090	0.00	—	—
6	ho	5934	7627	0.15	0.00	0.15	0.00	0.07	0.00	5444	0.00	51	0.00	157298	0.00
6	ni	5934	7627	0.12	0.00	0.08	0.00	0.05	0.00	5444	0.00	360	0.00	139309	0.00
6	k	5934	7627	2.52	0.00	1.22	0.00	0.08	0.00	5655	0.00	0	0.00	121313	0.00
6	ks	5934	7627	0.86	0.04	0.15	0.39	0.05	0.00	5655	0.00	6841	0.01	270582	0.01
7	hybrid	1323	2169	4.37	0.00	—	—	—	—	—	—	338	0.00	—	—
7	ho	1323	2169	0.03	0.00	0.02	0.00	0.00	0.00	1115	0.00	43	0.00	51818	0.00
7	ni	1323	2169	0.05	0.00	0.02	0.00	0.02	0.00	1115	0.00	162	0.00	70018	0.00
7	k	1323	2169	0.52	0.00	0.02	0.00	0.02	0.00	1205	0.00	0	0.00	46842	0.00
7	ks	1323	2169	0.32	0.05	0.00	0.00	0.02	0.00	1205	0.00	2800	0.04	101393	0.03
8	hybrid	1323	2195	4.70	0.00	—	—	—	—	—	—	368	0.00	—	—
8	ho	1323	2195	0.05	0.00	0.03	0.00	0.02	0.00	1167	0.00	38	0.00	48740	0.00
8	ni	1323	2195	0.03	0.00	0.03	0.00	0.00	0.00	1167	0.00	113	0.00	62461	0.00
8	k	1323	2195	0.42	0.00	0.20	0.00	0.02	0.00	1214	0.00	0	0.00	43756	0.00
8	ks	1323	2195	0.31	0.03	0.04	0.42	0.02	0.00	1214	0.00	2670	0.03	98943	0.02
9	hybrid	1084	1252	1.72	0.00	—	—	—	—	—	—	336	0.00	—	—
9	ho	1084	1252	0.02	0.00	0.00	0.00	0.00	0.00	1052	0.00	7	0.00	11580	0.00
9	ni	1084	1252	0.02	0.00	0.00	0.00	0.00	0.00	1052	0.00	23	0.00	11403	0.00
9	k	1084	1252	0.07	0.00	0.00	0.00	0.00	0.00	1061	0.00	0	0.00	15128	0.00
9	ks	1084	1252	0.03	0.00	0.00	0.00	0.00	2.00	1061	0.00	261	0.04	13324	0.02
10	hybrid	1748	2336	5.87	0.00	—	—	—	—	—	—	535	0.00	—	—
10	ho	1748	2336	0.03	0.00	0.02	0.00	0.02	0.00	1611	0.00	33	0.00	39664	0.00
10	ni	1748	2336	0.02	0.00	0.02	0.00	0.02	0.00	1611	0.00	109	0.00	37891	0.00
10	k	1748	2336	0.20	0.00	0.00	0.00	0.02	0.00	1671	0.00	0	0.00	37268	0.00
10	ks	1748	2336	0.14	0.14	0.00	2.00	0.02	0.00	1671	0.00	1353	0.02	54031	0.01
11	hybrid	15112	19057	392.22	0.00	—	—	—	—	—	—	1948	0.00	—	—
11	ho	15112	19057	0.48	0.00	0.20	0.00	0.20	0.00	13912	0.00	188	0.00	443071	0.00
11	ni	15112	19057	0.52	0.00	0.18	0.00	0.17	0.00	13912	0.00	793	0.00	423025	0.00
11	k	15112	19057	15.42	0.00	0.22	0.00	0.27	0.00	14473	0.00	0	0.00	329892	0.00
11	ks	15112	19057	1.71	0.01	0.17	0.00	0.22	0.00	14473	0.00	14407	0.01	558845	0.01

Table 18: TSP misc family

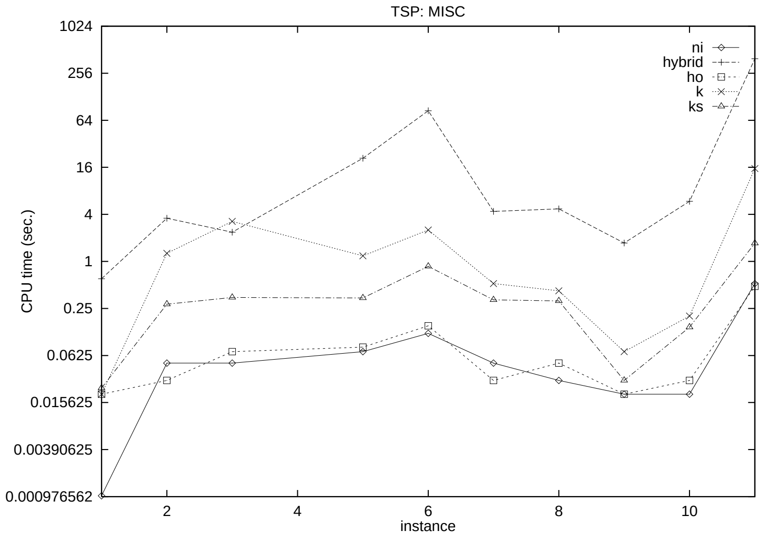

Figure 17: Different algorithms on TSP misc family

problem		nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	preprocess time avg	preprocess time dev %	initial PR avg	initial PR dev %	internal PR avg	internal PR dev %	edge scans avg	edge scans dev %
12	hybrid	33810	38600	3.27	0.00	—	—	—	—	—	—	0	0.00	—	—
12	ho	33810	38600	0.48	0.00	0.47	0.00	0.35	0.00	33808	0.00	0	0.00	271169	0.00
12	ni	33810	38600	0.43	0.00	0.43	0.00	0.28	0.00	33808	0.00	0	0.00	271168	0.00
12	k	33810	38600	0.47	0.00	0.47	0.00	0.35	0.00	33808	0.00	0	0.00	271168	0.00
12	ks	33810	38600	0.40	0.02	0.39	0.02	0.28	0.03	33808	0.00	0	0.00	193968	0.00
13	hybrid	33810	39367	11.37	0.00	—	—	—	—	—	—	320	0.00	—	—
13	ho	33810	39367	0.52	0.00	0.50	0.00	0.38	0.00	33808	0.00	0	0.00	303275	0.00
13	ni	33810	39367	0.47	0.00	0.47	0.00	0.32	0.00	33808	0.00	0	0.00	303274	0.00
13	k	33810	39367	0.47	0.00	0.45	0.00	0.37	0.00	33808	0.00	0	0.00	303274	0.00
13	ks	33810	39367	0.43	0.01	0.43	0.01	0.32	0.02	33808	0.00	0	0.00	224540	0.00
14	hybrid	33810	39456	3.98	0.00	—	—	—	—	—	—	0	0.00	—	—
14	ho	33810	39456	0.52	0.00	0.47	0.00	0.37	0.00	33808	0.00	0	0.00	295708	0.00
14	ni	33810	39456	0.45	0.00	0.43	0.00	0.30	0.00	33808	0.00	0	0.00	295707	0.00
14	k	33810	39456	0.52	0.00	0.48	0.00	0.40	0.00	33808	0.00	0	0.00	295707	0.00
14	ks	33810	39456	0.42	0.01	0.38	0.02	0.32	0.00	33808	0.00	0	0.00	216795	0.00
15	hybrid	85900	102596	16.62	0.00	—	—	—	—	—	—	5	0.00	—	—
15	ho	85900	102596	1.60	0.00	1.40	0.00	1.20	0.00	85898	0.00	0	0.00	810397	0.00
15	ni	85900	102596	1.37	0.00	1.25	0.00	1.02	0.00	85898	0.00	0	0.00	810396	0.00
15	k	85900	102596	1.57	0.00	1.42	0.00	1.28	0.00	85898	0.00	0	0.00	810396	0.00
15	ks	85900	102596	1.29	0.01	1.17	0.01	1.02	0.02	85898	0.00	0	0.00	605204	0.00
16	hybrid	85900	102934	21.18	0.00	—	—	—	—	—	—	22	0.00	—	—
16	ho	85900	102934	1.63	0.00	1.60	0.00	1.27	0.00	85898	0.00	0	0.00	855359	0.00
16	ni	85900	102934	1.43	0.00	1.42	0.00	1.08	0.00	85898	0.00	0	0.00	855358	0.00
16	k	85900	102934	1.50	0.00	1.50	0.00	1.20	0.00	85898	0.00	0	0.00	855358	0.00
16	ks	85900	102934	1.33	0.01	1.33	0.01	1.06	0.01	85898	0.00	0	0.00	649490	0.00
17	hybrid	85900	102988	21.30	0.00	—	—	—	—	—	—	11	0.00	—	—
17	ho	85900	102988	1.67	0.00	0.52	0.00	1.28	0.00	85830	0.00	3	0.00	869204	0.00
17	ni	85900	102988	1.43	0.00	0.50	0.00	1.08	0.00	85830	0.00	25	0.00	868777	0.00
17	k	85900	102988	2.05	0.00	0.52	0.00	1.35	0.00	85839	0.00	0	0.00	883540	0.00
17	ks	85900	102988	1.43	0.01	0.42	0.00	1.09	0.01	85839	0.00	790	0.03	677591	0.00

Table 19: TSP PLA family

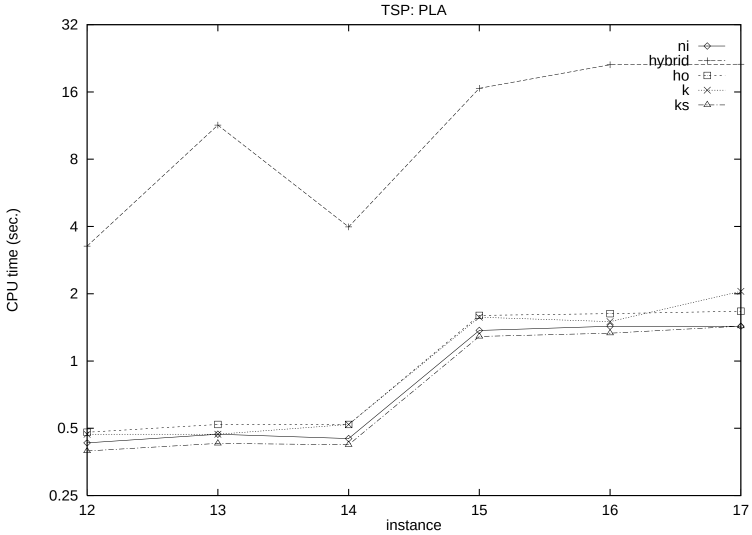

Figure 18: Different algorithms on TSP PLA family

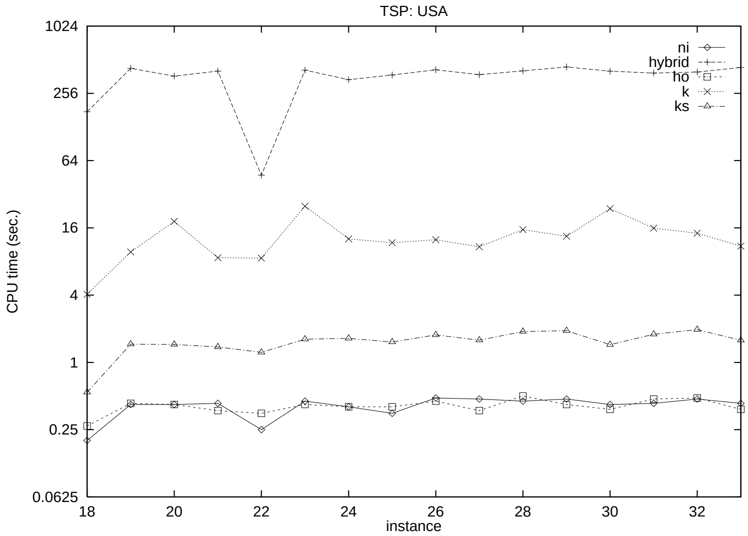

Figure 19: Different algorithms on TSP USA family

problem		nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
				avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
18	hybrid	13509	15631	175.25	0.00	—	—	—	—	—	—	1222	0.00	—	—
18	ho	13509	15631	0.27	0.00	0.25	0.00	0.13	0.00	12976	0.00	89	0.00	197589	0.00
18	ni	13509	15631	0.20	0.00	0.18	0.00	0.12	0.00	12976	0.00	281	0.00	164535	0.00
18	k	13509	15631	4.07	0.00	0.22	0.00	0.18	0.00	13185	0.00	0	0.00	209073	0.00
18	ks	13509	15631	0.54	0.03	0.17	0.05	0.14	0.07	13185	0.00	4409	0.02	214358	0.01
19	hybrid	13509	17048	428.88	0.00	—	—	—	—	—	—	2591	0.00	—	—
19	ho	13509	17048	0.43	0.00	0.42	0.00	0.18	0.00	12492	0.00	194	0.00	376734	0.00
19	ni	13509	17048	0.42	0.00	0.30	0.00	0.13	0.00	12492	0.00	678	0.00	351284	0.00
19	k	13509	17048	9.75	0.00	3.03	0.00	0.22	0.00	12985	0.00	0	0.00	273244	0.00
19	ks	13509	17048	1.46	0.04	0.33	0.12	0.18	0.06	12985	0.00	11828	0.01	468471	0.01
20	hybrid	13509	17079	364.72	0.00	—	—	—	—	—	—	2099	0.00	—	—
20	ho	13509	17079	0.42	0.00	0.28	0.00	0.17	0.00	12456	0.00	219	0.00	376394	0.00
20	ni	13509	17079	0.42	0.00	0.33	0.00	0.13	0.00	12456	0.00	732	0.00	354701	0.00
20	k	13509	17079	18.25	0.00	11.28	0.00	0.22	0.00	13003	0.00	0	0.00	343949	0.00
20	ks	13509	17079	1.45	0.01	0.51	0.47	0.18	0.00	13003	0.00	12195	0.01	481750	0.00
21	hybrid	13509	17111	404.72	0.00	—	—	—	—	—	—	2140	0.00	—	—
21	ho	13509	17111	0.37	0.00	0.02	0.00	0.17	0.00	12707	0.00	148	0.00	304868	0.00
21	ni	13509	17111	0.43	0.00	0.02	0.00	0.13	0.00	12707	0.00	526	0.00	360997	0.00
21	k	13509	17111	8.65	0.00	0.02	0.00	0.23	0.00	13021	0.00	0	0.00	273842	0.00
21	ks	13509	17111	1.37	0.03	0.01	1.22	0.18	0.04	13021	0.00	11330	0.01	464202	0.02
22	hybrid	13509	17130	47.42	0.00	—	—	—	—	—	—	662	0.00	—	—
22	ho	13509	17130	0.35	0.00	0.32	0.00	0.20	0.00	12725	0.00	97	0.00	276235	0.00
22	ni	13509	17130	0.25	0.00	0.22	0.00	0.13	0.00	12725	0.00	406	0.00	213941	0.00
22	k	13509	17130	8.58	0.00	0.27	0.00	0.23	0.00	13065	0.00	0	0.00	290612	0.00
22	ks	13509	17130	1.23	0.01	0.23	0.02	0.18	0.00	13065	0.00	10103	0.02	421984	0.01
23	hybrid	13509	17156	413.37	0.00	—	—	—	—	—	—	2410	0.00	—	—
23	ho	13509	17156	0.42	0.00	0.35	0.00	0.20	0.00	12649	0.00	194	0.00	359133	0.00
23	ni	13509	17156	0.45	0.00	0.45	0.00	0.15	0.00	12649	0.00	522	0.00	382971	0.00
23	k	13509	17156	25.03	0.00	16.32	0.00	0.23	0.00	12948	0.00	0	0.00	391233	0.00
23	ks	13509	17156	1.61	0.02	0.29	0.10	0.18	0.00	12948	0.00	13462	0.02	518571	0.01
24	hybrid	13509	17156	339.32	0.00	—	—	—	—	—	—	2151	0.00	—	—
24	ho	13509	17156	0.40	0.00	0.38	0.00	0.17	0.00	12684	0.00	180	0.00	330043	0.00
24	ni	13509	17156	0.40	0.00	0.25	0.00	0.15	0.00	12684	0.00	529	0.00	345126	0.00
24	k	13509	17156	12.75	0.00	8.13	0.00	0.23	0.00	12925	0.00	0	0.00	287698	0.00
24	ks	13509	17156	1.64	0.04	0.33	0.36	0.18	0.02	12925	0.00	13680	0.02	520969	0.02
25	hybrid	13509	17183	374.07	0.00	—	—	—	—	—	—	2122	0.00	—	—
25	ho	13509	17183	0.40	0.00	0.17	0.00	0.17	0.00	12419	0.00	141	0.00	344153	0.00
25	ni	13509	17183	0.35	0.00	0.15	0.00	0.13	0.00	12419	0.00	716	0.00	296352	0.00
25	k	13509	17183	11.78	0.00	0.15	0.00	0.22	0.00	12966	0.00	0	0.00	304583	0.00
25	ks	13509	17183	1.52	0.03	0.15	0.00	0.18	0.00	12966	0.00	12706	0.01	496048	0.01
26	hybrid	13509	17193	415.75	0.00	—	—	—	—	—	—	2319	0.00	—	—
26	ho	13509	17193	0.45	0.00	0.43	0.00	0.17	0.00	12427	0.00	242	0.00	394799	0.00
26	ni	13509	17193	0.48	0.00	0.42	0.00	0.15	0.00	12427	0.00	745	0.00	398840	0.00
26	k	13509	17193	12.52	0.00	4.77	0.00	0.20	0.00	12918	0.00	0	0.00	288607	0.00
26	ks	13509	17193	1.76	0.03	0.40	0.19	0.19	0.08	12918	0.00	14523	0.01	556273	0.01
27	hybrid	13509	17210	376.00	0.00	—	—	—	—	—	—	2300	0.00	—	—
27	ho	13509	17210	0.37	0.00	0.18	0.00	0.18	0.00	12645	0.00	175	0.00	327103	0.00
27	ni	13509	17210	0.47	0.00	0.18	0.00	0.15	0.00	12645	0.00	547	0.00	408682	0.00
27	k	13509	17210	10.83	0.00	0.20	0.00	0.22	0.00	12978	0.00	0	0.00	292039	0.00
27	ks	13509	17210	1.58	0.02	0.17	0.02	0.18	0.04	12978	0.00	12794	0.02	508462	0.01
28	hybrid	13509	17303	407.12	0.00	—	—	—	—	—	—	2115	0.00	—	—
28	ho	13509	17303	0.50	0.00	0.12	0.00	0.17	0.00	12402	0.00	259	0.00	458091	0.00
28	ni	13509	17303	0.45	0.00	0.10	0.00	0.15	0.00	12402	0.00	745	0.00	381080	0.00
28	k	13509	17303	15.43	0.00	0.10	0.00	0.22	0.00	12880	0.00	0	0.00	298952	0.00
28	ks	13509	17303	1.89	0.01	0.10	0.08	0.18	0.04	12880	0.00	16000	0.02	591107	0.01
29	hybrid	13509	17358	440.32	0.00	—	—	—	—	—	—	1990	0.00	—	—
29	ho	13509	17358	0.42	0.00	0.07	0.00	0.20	0.00	12582	0.00	215	0.00	355828	0.00
29	ni	13509	17358	0.47	0.00	0.07	0.00	0.15	0.00	12582	0.00	610	0.00	401551	0.00
29	k	13509	17358	13.45	0.00	0.08	0.00	0.23	0.00	12896	0.00	0	0.00	296496	0.00
29	ks	13509	17358	1.92	0.02	0.07	0.00	0.19	0.05	12896	0.00	16411	0.01	611674	0.01
30	hybrid	13509	17375	403.95	0.00	—	—	—	—	—	—	1875	0.00	—	—
30	ho	13509	17375	0.38	0.00	0.38	0.00	0.20	0.00	12714	0.00	165	0.00	330343	0.00
30	ni	13509	17375	0.42	0.00	0.25	0.00	0.15	0.00	12714	0.00	485	0.00	364594	0.00
30	k	13509	17375	23.80	0.00	0.28	0.00	0.23	0.00	12996	0.00	0	0.00	387113	0.00
30	ks	13509	17375	1.44	0.03	0.23	0.02	0.18	0.04	12996	0.00	11612	0.01	476135	0.01
31	hybrid	13509	17386	388.87	0.00	—	—	—	—	—	—	2086	0.00	—	—
31	ho	13509	17386	0.47	0.00	0.38	0.00	0.18	0.00	12343	0.00	193	0.00	403020	0.00
31	ni	13509	17386	0.43	0.00	0.30	0.00	0.15	0.00	12343	0.00	765	0.00	363960	0.00
31	k	13509	17386	15.90	0.00	0.27	0.00	0.27	0.00	12850	0.00	0	0.00	307160	0.00
31	ks	13509	17386	1.79	0.01	0.21	0.05	0.20	0.00	12850	0.00	15186	0.01	564545	0.01
32	hybrid	13509	17390	396.90	0.00	—	—	—	—	—	—	2435	0.00	—	—
32	ho	13509	17390	0.48	0.00	0.03	0.00	0.18	0.00	12385	0.00	157	0.00	419990	0.00
32	ni	13509	17390	0.47	0.00	0.03	0.00	0.15	0.00	12385	0.00	751	0.00	400896	0.00
32	k	13509	17390	14.33	0.00	0.03	0.00	0.22	0.00	12874	0.00	0	0.00	298705	0.00
32	ks	13509	17390	1.97	0.02	0.03	0.00	0.19	0.05	12874	0.00	16615	0.02	614098	0.02
33	hybrid	13509	17494	435.72	0.00	—	—	—	—	—	—	1981	0.00	—	—
33	ho	13509	17494	0.38	0.00	0.12	0.00	0.20	0.00	12626	0.00	162	0.00	321681	0.00
33	ni	13509	17494	0.43	0.00	0.10	0.00	0.17	0.00	12626	0.00	566	0.00	366605	0.00
33	k	13509	17494	10.97	0.00	0.12	0.00	0.25	0.00	12966	0.00	0	0.00	291153	0.00
33	ks	13509	17494	1.58	0.01	0.08	0.05	0.20	0.00	12966	0.00	12809	0.01	508943	0.01

Table 20: TSP USAfamily

problem		nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
				avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
1	hybrid	20	38	0.00	0.00	—	—	—	—	—	—	3	0.00	—	—
1	ho	20	38	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	6	0.00	1090	0.00
1	ni	20	38	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	16	0.00	647	0.00
1	k	20	38	0.02	0.18	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2658	0.22
1	ks	20	38	0.02	0.18	0.00	0.00	0.00	0.00	0	0.00	214	0.10	4539	0.07
2	hybrid	88	185	0.02	0.00	—	—	—	—	—	—	11	0.00	—	—
2	ho	88	185	0.02	0.00	0.00	0.00	0.00	0.00	0	0.00	23	0.00	12296	0.00
2	ni	88	185	0.02	0.00	0.00	0.00	0.00	0.00	0	0.00	63	0.00	16960	0.00
2	k	88	185	0.52	0.45	0.00	0.00	0.00	0.00	0	0.00	0	0.00	20455	0.30
2	ks	88	185	0.19	0.09	0.00	0.00	0.00	0.00	0	0.00	1837	0.06	47603	0.06
3	hybrid	148	300	0.03	0.00	—	—	—	—	—	—	10	0.00	—	—
3	ho	148	300	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	49	0.00	16998	0.00
3	ni	148	300	0.02	0.00	0.00	0.00	0.00	0.00	0	0.00	94	0.00	21620	0.00
3	k	148	300	0.68	0.07	0.32	0.10	0.01	1.22	0	0.00	0	0.00	23017	0.05
3	ks	148	300	0.26	0.04	0.02	0.18	0.00	0.00	0	0.00	2569	0.03	62200	0.02
4	hybrid	2	1	0.00	0.00	—	—	—	—	—	—	0	0.00	—	—
4	ho	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	3	0.00
4	ni	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00
4	k	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00
4	ks	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	2464	0.00	0	0.00
5	hybrid	150	292	0.05	0.00	—	—	—	—	—	—	16	0.00	—	—
5	ho	150	292	0.02	0.00	0.02	0.00	0.00	0.00	0	0.00	27	0.00	14948	0.00
5	ni	150	292	0.02	0.00	0.00	0.00	0.00	0.00	0	0.00	95	0.00	23816	0.00
5	k	150	292	1.36	0.42	0.97	0.59	0.01	1.22	0	0.00	0	0.00	35195	0.29
5	ks	150	292	0.39	0.03	0.02	0.20	0.00	2.00	0	0.00	3513	0.02	95174	0.02
6	hybrid	261	517	0.13	0.00	—	—	—	—	—	—	19	0.00	—	—
6	ho	261	517	0.02	0.00	0.02	0.00	0.00	0.00	0	0.00	44	0.00	40893	0.00
6	ni	261	517	0.03	0.00	0.00	0.00	0.00	0.00	0	0.00	162	0.00	53429	0.00
6	k	261	517	3.72	0.53	2.45	0.83	0.00	2.00	0	0.00	0	0.00	58087	0.32
6	ks	261	517	0.73	0.04	0.06	0.66	0.00	0.00	0	0.00	6243	0.02	188759	0.03
7	hybrid	113	221	0.05	0.00	—	—	—	—	—	—	3	0.00	—	—
7	ho	113	221	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	26	0.00	14372	0.00
7	ni	113	221	0.03	0.00	0.00	0.00	0.00	0.00	0	0.00	71	0.00	34228	0.00
7	k	113	221	0.37	0.03	0.00	0.00	0.00	0.00	0	0.00	0	0.00	16451	0.03
7	ks	113	221	0.29	0.03	0.00	0.00	0.00	0.00	0	0.00	2644	0.04	73742	0.04
8	hybrid	106	208	0.03	0.00	—	—	—	—	—	—	3	0.00	—	—
8	ho	106	208	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	27	0.00	12416	0.00
8	ni	106	208	0.02	0.00	0.00	0.00	0.00	0.00	0	0.00	68	0.00	29148	0.00
8	k	106	208	0.63	0.39	0.43	0.54	0.00	0.00	0	0.00	0	0.00	24010	0.30
8	ks	106	208	0.29	0.06	0.06	0.53	0.00	0.00	0	0.00	2598	0.04	72097	0.04
9	hybrid	19	36	0.00	0.00	—	—	—	—	—	—	2	0.00	—	—
9	ho	19	36	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	5	0.00	848	0.00
9	ni	19	36	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	15	0.00	612	0.00
9	k	19	36	0.03	0.42	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2832	0.25
9	ks	19	36	0.02	0.00	0.00	0.00	0.00	0.00	0	0.00	196	0.04	4058	0.09
10	hybrid	77	131	0.02	0.00	—	—	—	—	—	—	4	0.00	—	—
10	ho	77	131	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	23	0.00	5748	0.00
10	ni	77	131	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	54	0.00	9331	0.00
10	k	77	131	0.20	0.14	0.00	0.00	0.00	0.00	0	0.00	0	0.00	10825	0.05
10	ks	77	131	0.13	0.06	0.00	0.00	0.00	0.00	0	0.00	1353	0.03	31341	0.04
11	hybrid	605	1162	0.32	0.00	—	—	—	—	—	—	15	0.00	—	—
11	ho	605	1162	0.07	0.00	0.00	0.00	0.00	0.00	0	0.00	126	0.00	106232	0.00
11	ni	605	1162	0.18	0.00	0.00	0.00	0.00	0.00	0	0.00	332	0.00	204810	0.00
11	k	605	1162	13.58	0.04	0.00	0.00	0.00	0.00	0	0.00	0	0.00	113109	0.05
11	ks	605	1162	1.37	0.02	0.00	0.00	0.00	0.00	0	0.00	13605	0.01	360873	0.01

Table 21: PRETSP misc family

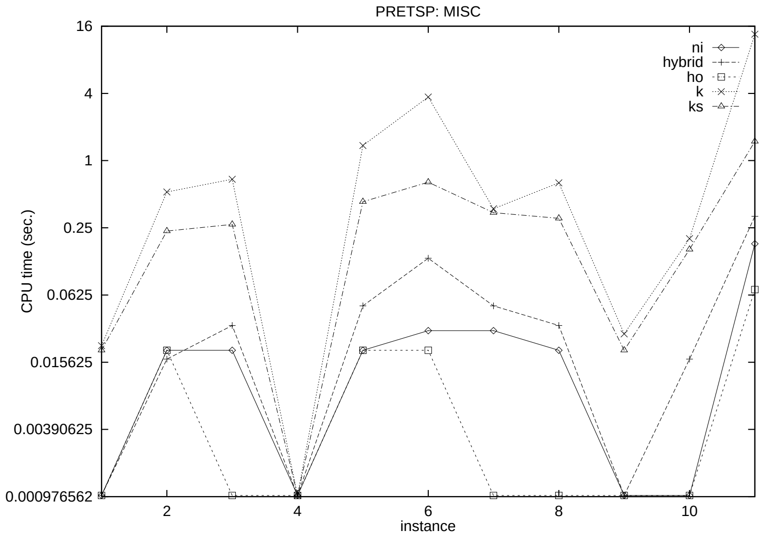

Figure 20: Different algorithms on PRETSP misc family

problem		nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
				avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
12	hybrid	2	1	0.00	0.00	—	—	—	—	—	—	0	0.00	—	—
12	ho	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	3	0.00
12	ni	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00
12	k	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00
12	ks	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	0	0.00
13	hybrid	2	1	0.02	0.00	—	—	—	—	—	—	0	0.00	—	—
13	ho	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	3	0.00
13	ni	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00
13	k	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00
13	ks	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	0	0.00
14	hybrid	2	1	0.00	0.00	—	—	—	—	—	—	0	0.00	—	—
14	ho	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	3	0.00
14	ni	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00
14	k	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00
14	ks	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	0	0.00
15	hybrid	2	1	0.00	0.00	—	—	—	—	—	—	0	0.00	—	—
15	ho	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	3	0.00
15	ni	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00
15	k	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00
15	ks	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	0	0.00
16	hybrid	2	1	0.00	0.00	—	—	—	—	—	—	0	0.00	—	—
16	ho	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	3	0.00
16	ni	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00
16	k	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00
16	ks	2	1	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	0	0.00	0	0.00
17	hybrid	52	90	0.02	0.00	—	—	—	—	—	—	0	0.00	—	—
17	ho	52	90	0.00	0.00	0.00	0.00	0.00	0.00	0	0.00	10	0.00	2048	0.00
17	ni	52	90	0.02	0.00	0.00	0.00	0.00	0.00	0	0.00	31	0.00	2329	0.00
17	k	52	90	0.13	0.67	0.00	0.00	0.00	0.00	0	0.00	0	0.00	8354	0.31
17	ks	52	90	0.06	0.17	0.00	0.00	0.00	0.00	0	0.00	694	0.06	12514	0.05

Table 22: PRETSP PLA family

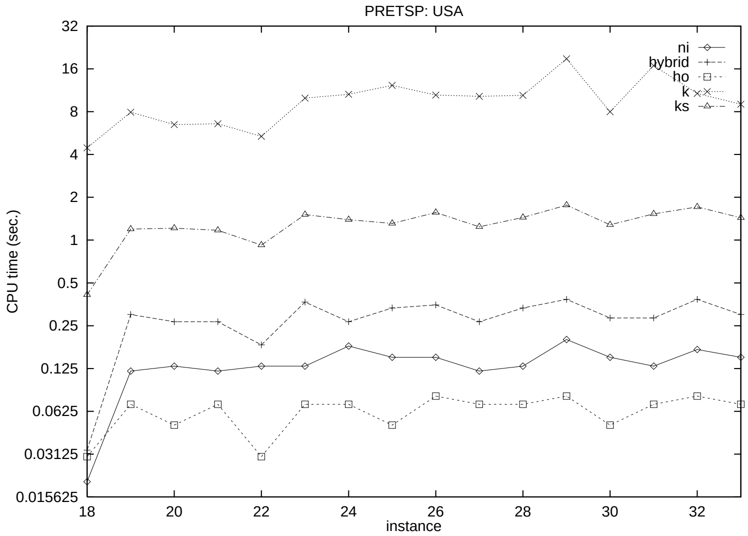

Figure 21: Different algorithms on PRETSP USA family

problem		nodes	arcs	total time		discovery time		preprocess time		initial PR		internal PR		edge scans	
				avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %
18	hybrid	325	561	0.03	0.00	—	—	—	—	—	—	8	0.00	—	—
18	ho	325	561	0.03	0.00	0.02	0.00	0.00	0.00	0	0.00	80	0.00	34755	0.00
18	ni	325	561	0.02	0.00	0.00	0.00	0.00	0.00	0	0.00	135	0.00	28537	0.00
18	k	325	561	4.46	0.47	1.19	0.25	0.00	0.00	0	0.00	0	0.00	70036	0.33
18	ks	325	561	0.37	0.03	0.08	0.61	0.00	2.00	0	0.00	4478	0.02	100814	0.03
19	hybrid	477	920	0.30	0.00	—	—	—	—	—	—	26	0.00	—	—
19	ho	477	920	0.07	0.00	0.03	0.00	0.02	0.00	0	0.00	68	0.00	85447	0.00
19	ni	477	920	0.12	0.00	0.02	0.00	0.00	0.00	0	0.00	260	0.00	147999	0.00
19	k	477	920	7.93	0.02	2.42	0.10	0.00	2.00	0	0.00	0	0.00	87513	0.05
19	ks	477	920	1.07	0.03	0.07	0.82	0.01	0.82	0	0.00	10668	0.02	286697	0.02
20	hybrid	449	861	0.27	0.00	—	—	—	—	—	—	19	0.00	—	—
20	ho	449	861	0.05	0.00	0.05	0.00	0.00	0.00	0	0.00	118	0.00	83948	0.00
20	ni	449	861	0.13	0.00	0.07	0.00	0.00	0.00	0	0.00	226	0.00	149716	0.00
20	k	449	861	6.49	0.03	1.86	0.15	0.00	0.00	0	0.00	0	0.00	76354	0.03
20	ks	449	861	1.06	0.02	0.10	1.03	0.00	2.00	0	0.00	10816	0.02	290509	0.02
21	hybrid	454	886	0.27	0.00	—	—	—	—	—	—	11	0.00	—	—
21	ho	454	886	0.07	0.00	0.00	0.00	0.00	0.00	0	0.00	80	0.00	82086	0.00
21	ni	454	886	0.12	0.00	0.00	0.00	0.00	0.00	0	0.00	241	0.00	139677	0.00
21	k	454	886	6.58	0.02	0.00	0.00	0.00	0.00	0	0.00	0	0.00	77929	0.04
21	ks	454	886	1.09	0.03	0.00	0.00	0.01	1.22	0	0.00	10670	0.03	291432	0.03
22	hybrid	403	786	0.18	0.00	—	—	—	—	—	—	12	0.00	—	—
22	ho	403	786	0.03	0.00	0.03	0.00	0.00	0.00	0	0.00	72	0.00	58094	0.00
22	ni	403	786	0.13	0.00	0.13	0.00	0.00	0.00	0	0.00	224	0.00	126807	0.00
22	k	403	786	5.36	0.03	1.86	0.10	0.00	0.00	0	0.00	0	0.00	69655	0.05
22	ks	403	786	0.89	0.01	0.07	0.80	0.00	0.00	0	0.00	9115	0.02	241598	0.02
23	hybrid	532	1029	0.37	0.00	—	—	—	—	—	—	28	0.00	—	—
23	ho	532	1029	0.07	0.00	0.05	0.00	0.02	0.00	0	0.00	128	0.00	108142	0.00
23	ni	532	1029	0.13	0.00	0.00	0.00	0.00	0.00	0	0.00	291	0.00	174077	0.00
23	k	532	1029	9.96	0.05	3.37	0.18	0.00	2.00	0	0.00	0	0.00	93386	0.05
23	ks	532	1029	1.35	0.03	0.18	0.53	0.00	0.00	0	0.00	12995	0.02	350271	0.02
24	hybrid	501	950	0.27	0.00	—	—	—	—	—	—	16	0.00	—	—
24	ho	501	950	0.07	0.00	0.00	0.00	0.02	0.00	0	0.00	97	0.00	78473	0.00
24	ni	501	950	0.18	0.00	0.00	0.00	0.00	0.00	0	0.00	273	0.00	218389	0.00
24	k	501	950	10.59	0.36	0.00	0.00	0.00	2.00	0	0.00	0	0.00	105218	0.31
24	ks	501	950	1.26	0.04	0.00	0.00	0.00	0.00	0	0.00	12222	0.02	330751	0.02
25	hybrid	492	946	0.33	0.00	—	—	—	—	—	—	19	0.00	—	—
25	ho	492	946	0.05	0.00	0.00	0.00	0.00	0.00	0	0.00	88	0.00	75279	0.00
25	ni	492	946	0.15	0.00	0.00	0.00	0.00	0.00	0	0.00	256	0.00	149765	0.00
25	k	492	946	12.25	0.40	0.00	0.00	0.00	0.00	0	0.00	0	0.00	119819	0.34
25	ks	492	946	1.17	0.02	0.00	2.00	0.00	2.00	0	0.00	11504	0.01	310030	0.01
26	hybrid	549	1072	0.35	0.00	—	—	—	—	—	—	12	0.00	—	—
26	ho	549	1072	0.08	0.00	0.07	0.00	0.00	0.00	0	0.00	76	0.00	100108	0.00
26	ni	549	1072	0.15	0.00	0.12	0.00	0.00	0.00	0	0.00	303	0.00	178225	0.00
26	k	549	1072	10.48	0.02	2.94	0.10	0.00	0.00	0	0.00	0	0.00	96879	0.04
26	ks	549	1072	1.41	0.04	0.21	0.83	0.01	1.22	0	0.00	13492	0.02	369945	0.03
27	hybrid	465	926	0.27	0.00	—	—	—	—	—	—	17	0.00	—	—
27	ho	465	926	0.07	0.00	0.00	0.00	0.00	0.00	0	0.00	106	0.00	107500	0.00
27	ni	465	926	0.12	0.00	0.00	0.00	0.00	0.00	0	0.00	254	0.00	147139	0.00
27	k	465	926	10.26	0.34	0.00	0.00	0.00	2.00	0	0.00	0	0.00	114973	0.30
27	ks	465	926	1.10	0.04	0.00	0.00	0.01	0.82	0	0.00	10962	0.02	298687	0.03
28	hybrid	542	1064	0.33	0.00	—	—	—	—	—	—	28	0.00	—	—
28	ho	542	1064	0.07	0.00	0.00	0.00	0.00	0.00	0	0.00	135	0.00	103811	0.00
28	ni	542	1064	0.13	0.00	0.00	0.00	0.02	0.00	0	0.00	293	0.00	145787	0.00
28	k	542	1064	10.42	0.01	0.00	2.00	0.00	2.00	0	0.00	0	0.00	99154	0.02
28	ks	542	1064	1.34	0.02	0.00	0.00	0.01	1.22	0	0.00	13065	0.01	357056	0.02
29	hybrid	573	1104	0.38	0.00	—	—	—	—	—	—	21	0.00	—	—
29	ho	573	1104	0.08	0.00	0.00	0.00	0.00	0.00	0	0.00	121	0.00	104487	0.00
29	ni	573	1104	0.20	0.00	0.00	0.00	0.00	0.00	0	0.00	313	0.00	198529	0.00
29	k	573	1104	18.83	0.34	0.00	0.00	0.00	0.00	0	0.00	0	0.00	159432	0.31
29	ks	573	1104	1.53	0.02	0.00	0.00	0.00	0.00	0	0.00	14605	0.02	402105	0.02
30	hybrid	476	945	0.28	0.00	—	—	—	—	—	—	17	0.00	—	—
30	ho	476	945	0.05	0.00	0.00	0.00	0.00	0.00	0	0.00	89	0.00	87625	0.00
30	ni	476	945	0.15	0.00	0.00	0.00	0.00	0.00	0	0.00	247	0.00	177708	0.00
30	k	476	945	7.98	0.03	0.00	0.00	0.01	1.22	0	0.00	0	0.00	87580	0.05
30	ks	476	945	1.12	0.02	0.00	2.00	0.00	2.00	0	0.00	10744	0.01	296343	0.02
31	hybrid	607	1150	0.28	0.00	—	—	—	—	—	—	14	0.00	—	—
31	ho	607	1150	0.07	0.00	0.00	0.00	0.02	0.00	0	0.00	86	0.00	86629	0.00
31	ni	607	1150	0.13	0.00	0.00	0.00	0.00	0.00	0	0.00	301	0.00	160409	0.00
31	k	607	1150	16.88	0.40	0.00	2.00	0.00	0.00	0	0.00	0	0.00	134437	0.33
31	ks	607	1150	1.44	0.02	0.00	0.00	0.01	1.22	0	0.00	13895	0.01	373567	0.01
32	hybrid	557	1091	0.38	0.00	—	—	—	—	—	—	14	0.00	—	—
32	ho	557	1091	0.08	0.00	0.00	0.00	0.00	0.00	0	0.00	138	0.00	114243	0.00
32	ni	557	1091	0.17	0.00	0.00	0.00	0.00	0.00	0	0.00	307	0.00	191347	0.00
32	k	557	1091	10.74	0.02	0.00	0.00	0.01	1.22	0	0.00	0	0.00	95862	0.01
32	ks	557	1091	1.52	0.02	0.01	0.82	0.01	1.22	0	0.00	14294	0.01	396642	0.02
33	hybrid	505	971	0.30	0.00	—	—	—	—	—	—	10	0.00	—	—
33	ho	505	971	0.07	0.00	0.00	0.00	0.00	0.00	0	0.00	103	0.00	90697	0.00
33	ni	505	971	0.15	0.00	0.00	0.00	0.00	0.00	0	0.00	270	0.00	169536	0.00
33	k	505	971	9.03	0.02	0.00	0.00	0.00	0.00	0	0.00	0	0.00	90956	0.04
33	ks	505	971	1.22	0.02	0.00	0.00	0.00	0.00	0	0.00	12031	0.02	329176	0.02

Table 23: PRETSP USAfamily

12.2 Tables comparing ho codes

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	300	22425	0.32	0.05	0	0.00	165	0.11	384908	0.08	3	298	0	129
ho_nopr	300	22425	0.29	0.08	0	0.00	0	0.00	368767	0.10	8	276	16	274
ho_noxs	300	22425	0.46	0.07	0	0.00	127	0.26	720732	0.04	38	236	132	0
ho_noprxs	300	22425	0.40	0.04	0	0.00	0	0.00	705176	0.05	49	186	249	0
ho	400	39900	0.65	0.03	0	0.00	250	0.03	676746	0.01	3	398	0	143
ho_nopr	400	39900	0.65	0.07	0	0.00	0	0.00	778865	0.09	12	374	37	349
ho_noxs	400	39900	1.09	0.10	0	0.00	162	0.37	1477102	0.11	40	307	195	0
ho_noprxs	400	39900	1.00	0.10	0	0.00	0	0.00	1474180	0.11	53	234	345	0
ho	500	62375	1.05	0.01	0	0.00	325	0.06	1028294	0.01	3	465	0	169
ho_nopr	500	62375	1.12	0.08	0	0.00	0	0.00	1270908	0.09	13	456	41	444
ho_noxs	500	62375	1.84	0.03	0	0.00	213	0.24	2278172	0.03	54	378	230	0
ho_noprxs	500	62375	1.65	0.05	0	0.00	0	0.00	2249321	0.05	70	298	429	0
ho	600	89850	1.60	0.01	0	0.00	406	0.03	1525886	0.02	3	598	0	187
ho_nopr	600	89850	1.78	0.06	0	0.00	0	0.00	1953035	0.07	13	558	69	516
ho_noxs	600	89850	2.98	0.06	0	0.00	266	0.27	3603150	0.06	56	418	275	0
ho_noprxs	600	89850	2.77	0.08	0	0.00	0	0.00	3649288	0.07	77	318	521	0
ho	700	122325	2.27	0.01	0	0.00	506	0.02	2101482	0.01	3	698	0	188
ho_nopr	700	122325	2.84	0.07	0	0.00	0	0.00	3117236	0.07	13	644	71	615
ho_noxs	700	122325	4.04	0.04	0	0.00	285	0.23	4762589	0.04	78	499	334	0
ho_noprxs	700	122325	3.75	0.05	0	0.00	0	0.00	4785148	0.06	95	412	603	0
ho	800	159800	3.02	0.01	0	0.00	570	0.05	2729953	0.01	3	798	0	224
ho_nopr	800	159800	3.61	0.03	0	0.00	0	0.00	3875115	0.04	15	729	99	683
ho_noxs	800	159800	5.35	0.03	0	0.00	347	0.31	6206945	0.03	82	582	367	0
ho_noprxs	800	159800	4.89	0.03	0	0.00	0	0.00	6221303	0.05	106	464	693	0
ho	900	202275	3.91	0.01	0	0.00	664	0.02	3483609	0.01	3	898	0	230
ho_nopr	900	202275	4.90	0.04	0	0.00	0	0.00	5390937	0.04	18	830	187	692
ho_noxs	900	202275	7.13	0.03	0	0.00	461	0.07	8296609	0.03	68	665	368	0
ho_noprxs	900	202275	6.64	0.04	0	0.00	0	0.00	8383682	0.04	96	495	803	0
ho	1000	249750	4.93	0.01	0	0.00	759	0.01	4284685	0.01	3	958	0	235
ho_nopr	1000	249750	6.55	0.08	0	0.00	0	0.00	7106372	0.10	16	933	243	738
ho_noxs	1000	249750	9.33	0.11	0	0.00	283	0.47	10554256	0.12	82	688	632	0
ho_noprxs	1000	249750	8.69	0.14	0	0.00	0	0.00	10550654	0.14	100	565	898	0

Table 24: NOI1 family, HO codes

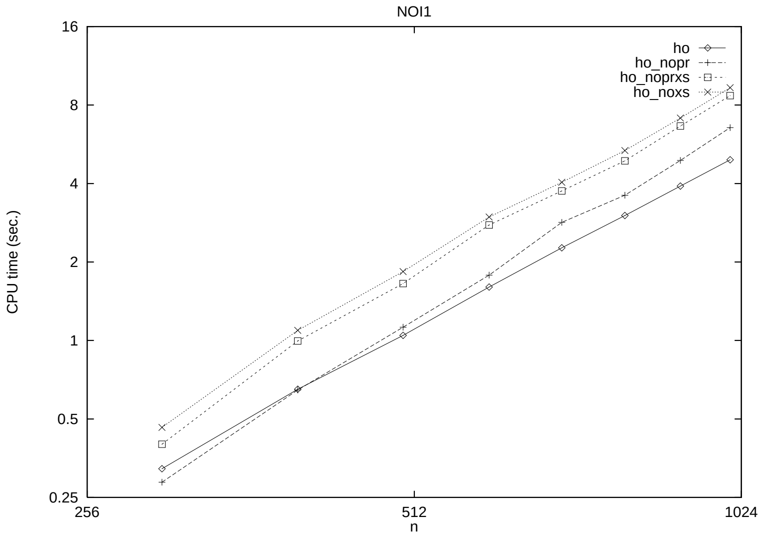

Figure 22: HO variants on NOI1 family

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	300	22425	0.26	0.09	0	0.00	72	0.23	316490	0.12	5	212	0	221
ho_nopr	300	22425	0.23	0.05	0	0.00	0	0.00	304551	0.06	10	178	0	288
ho_noxs	300	22425	0.42	0.10	0	0.00	148	0.17	640685	0.14	62	135	87	0
ho_noprxs	300	22425	0.35	0.12	0	0.00	0	0.00	672897	0.12	83	106	215	0
ho	400	39900	0.56	0.13	0	0.00	87	0.24	563229	0.17	5	289	3	302
ho_nopr	400	39900	0.49	0.06	0	0.00	0	0.00	552917	0.07	8	243	6	384
ho_noxs	400	39900	0.87	0.02	0	0.00	228	0.11	1110182	0.03	88	171	81	0
ho_noprxs	400	39900	0.78	0.03	0	0.00	0	0.00	1189634	0.07	114	138	284	0
ho	500	62375	1.06	0.12	0	0.00	125	0.27	1015152	0.14	5	349	0	367
ho_nopr	500	62375	0.90	0.05	0	0.00	0	0.00	951293	0.05	11	271	0	487
ho_noxs	500	62375	1.39	0.05	0	0.00	243	0.18	1703991	0.06	105	212	148	0
ho_noprxs	500	62375	1.26	0.05	0	0.00	0	0.00	1791169	0.07	138	170	360	0
ho	600	89850	1.62	0.05	0	0.00	163	0.28	1518734	0.05	3	536	0	430
ho_nopr	600	89850	1.40	0.02	0	0.00	0	0.00	1435509	0.02	15	306	0	583
ho_noxs	600	89850	2.14	0.06	0	0.00	343	0.05	2448590	0.07	137	257	117	0
ho_noprxs	600	89850	1.93	0.05	0	0.00	0	0.00	2549478	0.05	188	200	410	0
ho	700	122325	2.21	0.04	0	0.00	227	0.21	2011692	0.05	1	699	0	469
ho_nopr	700	122325	2.02	0.01	0	0.00	0	0.00	1996601	0.02	16	357	0	682
ho_noxs	700	122325	3.09	0.04	0	0.00	370	0.12	3419830	0.07	159	298	168	0
ho_noprxs	700	122325	2.90	0.04	0	0.00	0	0.00	3636673	0.05	219	232	479	0
ho	800	159800	3.41	0.07	0	0.00	183	0.33	3068408	0.08	5	696	3	606
ho_nopr	800	159800	2.84	0.05	0	0.00	0	0.00	2886664	0.06	15	449	3	780
ho_noxs	800	159800	4.40	0.04	0	0.00	439	0.09	4838982	0.06	173	329	185	0
ho_noprxs	800	159800	4.14	0.05	0	0.00	0	0.00	5186013	0.06	225	268	573	0
ho	900	202275	3.91	0.06	0	0.00	311	0.21	3399594	0.08	1	877	0	584
ho_nopr	900	202275	3.60	0.02	0	0.00	0	0.00	3504869	0.01	17	459	0	881
ho_noxs	900	202275	5.49	0.01	0	0.00	499	0.08	5855968	0.02	204	370	194	0
ho_noprxs	900	202275	5.12	0.02	0	0.00	0	0.00	6242312	0.02	270	294	628	0
ho	1000	249750	5.66	0.08	0	0.00	238	0.31	4855050	0.09	7	734	0	752
ho_nopr	1000	249750	4.68	0.03	0	0.00	0	0.00	4646956	0.03	17	530	0	982
ho_noxs	1000	249750	7.08	0.03	0	0.00	533	0.18	7395961	0.04	233	410	231	0
ho_noprxs	1000	249750	6.51	0.04	0	0.00	0	0.00	7770716	0.04	315	322	684	0

Table 25: NOI2 family, HO codes

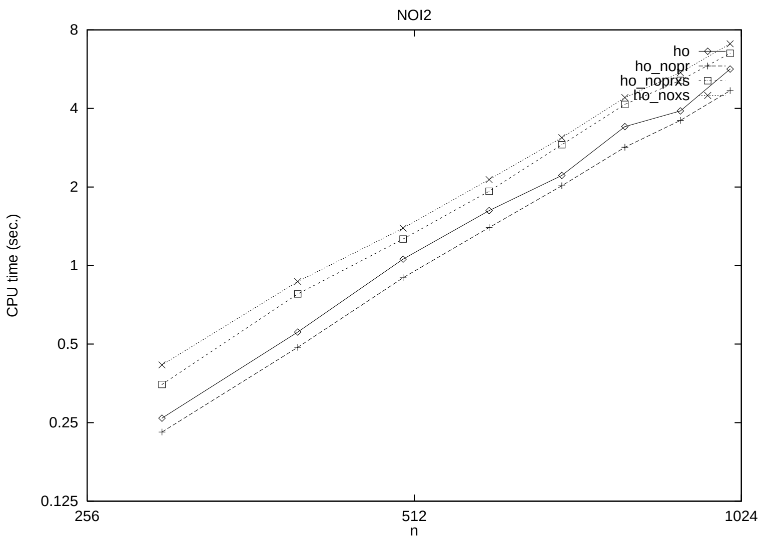

Figure 23: HO variants on NOI2 family

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	1000	24975	0.43	0.05	0	0.00	309	0.05	505140	0.05	5	977	0	683
ho_nopr	1000	24975	0.34	0.05	0	0.00	0	0.00	413508	0.04	11	968	0	987
ho_noxs	1000	24975	0.98	0.02	0	0.00	532	0.21	1463032	0.03	57	836	408	0
ho_noprxs	1000	24975	0.89	0.03	0	0.00	0	0.00	1478084	0.03	73	692	925	0
ho	1000	49950	1.02	0.12	0	0.00	386	0.50	1113648	0.13	6	986	2	602
ho_nopr	1000	49950	0.89	0.06	0	0.00	0	0.00	1013579	0.05	13	963	6	979
ho_noxs	1000	49950	2.03	0.09	0	0.00	386	0.26	2725982	0.11	67	860	544	0
ho_noprxs	1000	49950	1.81	0.13	0	0.00	0	0.00	2703915	0.13	78	759	920	0
ho	1000	124875	3.24	0.02	0	0.00	590	0.03	3261326	0.01	8	988	0	398
ho_nopr	1000	124875	2.92	0.04	0	0.00	0	0.00	3264192	0.06	15	934	31	952
ho_noxs	1000	124875	5.09	0.07	0	0.00	450	0.27	6064858	0.08	72	741	474	0
ho_noprxs	1000	124875	4.74	0.08	0	0.00	0	0.00	6133165	0.08	91	605	907	0
ho	1000	249750	4.95	0.02	0	0.00	759	0.01	4284685	0.01	3	958	0	235
ho_nopr	1000	249750	6.54	0.08	0	0.00	0	0.00	7106372	0.10	16	933	243	738
ho_noxs	1000	249750	9.39	0.11	0	0.00	283	0.47	10554256	0.12	82	688	632	0
ho_noprxs	1000	249750	8.63	0.14	0	0.00	0	0.00	10550654	0.14	100	565	898	0
ho	1000	374625	7.93	0.04	0	0.00	794	0.01	6706222	0.06	4	996	0	199
ho_nopr	1000	374625	9.59	0.10	0	0.00	0	0.00	10077034	0.09	18	906	215	765
ho_noxs	1000	374625	12.76	0.08	0	0.00	381	0.44	13650787	0.09	97	628	518	0
ho_noprxs	1000	374625	11.83	0.09	0	0.00	0	0.00	13696322	0.09	124	505	874	0
ho	1000	499500	10.45	0.01	0	0.00	818	0.01	8340469	0.01	5	996	0	174
ho_nopr	1000	499500	13.23	0.05	0	0.00	0	0.00	13704734	0.06	19	896	313	665
ho_noxs	1000	499500	16.02	0.02	0	0.00	320	0.34	16864169	0.03	86	574	592	0
ho_noprxs	1000	499500	14.67	0.03	0	0.00	0	0.00	16812484	0.04	107	472	891	0

Table 26: NOI3 family, HO codes

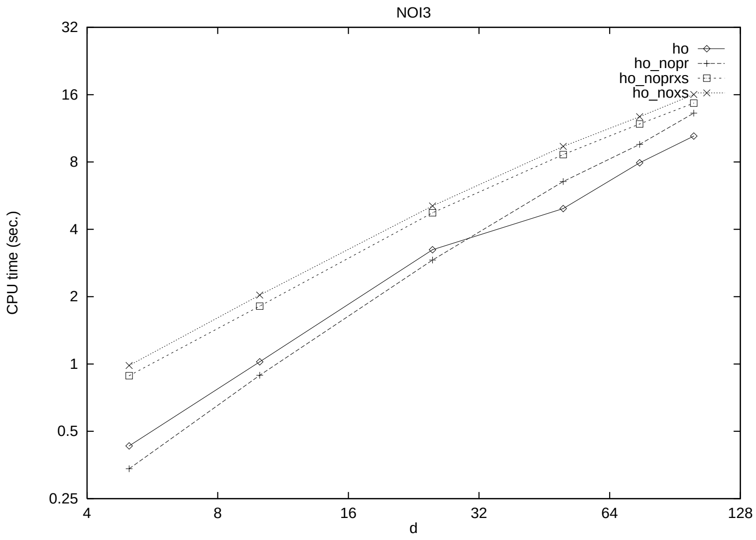

Figure 24: HO variants on NOI3 family

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	1000	24975	0.38	0.03	0	0.00	63	0.25	419805	0.05	3	733	0	930
ho_nopr	1000	24975	0.27	0.02	0	0.00	0	0.00	302558	0.09	7	631	1	990
ho_noxs	1000	24975	0.82	0.05	0	0.00	630	0.19	1173910	0.10	80	443	287	0
ho_noprxs	1000	24975	0.84	0.04	0	0.00	0	0.00	1555858	0.07	119	314	879	0
ho	1000	49950	1.00	0.04	0	0.00	141	0.08	1010936	0.02	3	734	0	853
ho_nopr	1000	49950	0.78	0.02	0	0.00	0	0.00	847685	0.03	10	549	0	988
ho_noxs	1000	49950	1.70	0.04	0	0.00	595	0.09	2254368	0.04	100	435	302	0
ho_noprxs	1000	49950	1.61	0.05	0	0.00	0	0.00	2587487	0.06	118	378	881	0
ho	1000	124875	2.77	0.12	0	0.00	217	0.58	2589176	0.14	9	669	0	771
ho_nopr	1000	124875	2.34	0.03	0	0.00	0	0.00	2415011	0.04	16	534	0	982
ho_noxs	1000	124875	3.96	0.06	0	0.00	495	0.04	4534065	0.07	159	448	343	0
ho_noprxs	1000	124875	3.62	0.05	0	0.00	0	0.00	4883287	0.04	193	378	805	0
ho	1000	249750	5.75	0.07	0	0.00	238	0.31	4855050	0.09	7	734	0	752
ho_nopr	1000	249750	4.69	0.03	0	0.00	0	0.00	4646956	0.03	17	530	0	982
ho_noxs	1000	249750	7.17	0.04	0	0.00	533	0.18	7395961	0.04	233	410	231	0
ho_noprxs	1000	249750	6.46	0.05	0	0.00	0	0.00	7770716	0.04	315	322	684	0
ho	1000	374625	8.18	0.05	0	0.00	370	0.10	6626486	0.07	6	724	0	622
ho_nopr	1000	374625	6.58	0.05	0	0.00	0	0.00	6444146	0.07	17	563	0	981
ho_noxs	1000	374625	9.46	0.07	0	0.00	577	0.01	9535335	0.10	273	400	146	0
ho_noprxs	1000	374625	8.58	0.06	0	0.00	0	0.00	9724673	0.06	361	324	637	0
ho	1000	499500	10.53	0.07	0	0.00	282	0.34	8181626	0.08	7	717	0	708
ho_nopr	1000	499500	8.16	0.06	0	0.00	0	0.00	7741986	0.10	16	577	11	970
ho_noxs	1000	499500	11.92	0.07	0	0.00	567	0.01	11556174	0.09	292	409	137	0
ho_noprxs	1000	499500	10.55	0.06	0	0.00	0	0.00	11486081	0.06	408	314	590	0

Table 27: NOI4 family, HO codes

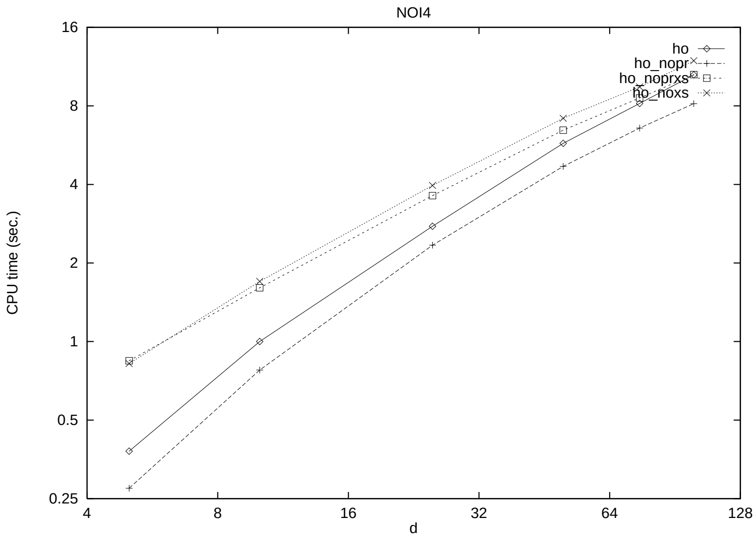

Figure 25: HO variants on NOI4 family

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	1000	249750	4.88	0.01	0	0.00	759	0.01	4284685	0.01	3	958	0	235
ho_nopr	1000	249750	6.54	0.08	0	0.00	0	0.00	7106372	0.10	16	933	243	738
ho_noxs	1000	249750	9.34	0.11	0	0.00	283	0.47	10554256	0.12	82	688	632	0
ho_noprxs	1000	249750	8.94	0.14	0	0.00	0	0.00	10550654	0.14	100	565	898	0
ho	1000	249750	5.60	0.08	0	0.00	238	0.31	4855050	0.09	7	734	0	752
ho_nopr	1000	249750	4.68	0.02	0	0.00	0	0.00	4646956	0.03	17	530	0	982
ho_noxs	1000	249750	7.03	0.03	0	0.00	533	0.18	7395961	0.04	233	410	231	0
ho_noprxs	1000	249750	6.59	0.04	0	0.00	0	0.00	7770716	0.04	315	322	684	0
ho	1000	249750	5.60	0.15	0	0.00	198	0.57	4817520	0.18	21	372	4	774
ho_nopr	1000	249750	4.99	0.07	0	0.00	0	0.00	5343890	0.06	25	391	7	966
ho_noxs	1000	249750	7.86	0.04	0	0.00	474	0.13	8673406	0.04	170	287	353	0
ho_noprxs	1000	249750	7.89	0.04	0	0.00	0	0.00	9687748	0.04	220	230	778	0
ho	1000	249750	6.93	0.03	0	0.00	134	0.24	6065805	0.04	25	262	1	837
ho_nopr	1000	249750	6.10	0.04	0	0.00	0	0.00	6680523	0.05	33	275	11	954
ho_noxs	1000	249750	8.23	0.04	0	0.00	511	0.06	9239626	0.04	179	193	307	0
ho_noprxs	1000	249750	8.37	0.04	0	0.00	0	0.00	10757072	0.03	249	147	749	0
ho	1000	249750	7.78	0.06	0	0.00	106	0.39	6902801	0.06	26	255	3	862
ho_nopr	1000	249750	6.87	0.08	0	0.00	0	0.00	7537622	0.09	32	262	7	959
ho_noxs	1000	249750	9.22	0.03	0	0.00	508	0.07	10363995	0.03	210	170	279	0
ho_noprxs	1000	249750	9.47	0.02	0	0.00	0	0.00	12305603	0.03	279	134	719	0
ho	1000	249750	7.09	0.18	47	0.58	85	0.43	6134482	0.21	8	429	1	856
ho_nopr	1000	249750	5.66	0.29	0	0.00	0	0.00	6012856	0.34	19	294	0	978
ho_noxs	1000	249750	8.43	0.10	47	0.58	627	0.04	9156729	0.11	198	112	125	0
ho_noprxs	1000	249750	8.86	0.12	0	0.00	0	0.00	11482142	0.11	346	95	652	0
ho	1000	249750	5.78	0.50	235	1.62	28	0.81	4658492	0.61	4	577	0	729
ho_nopr	1000	249750	5.28	0.33	0	0.00	0	0.00	5157881	0.41	6	469	1	991
ho_noxs	1000	249750	7.86	0.47	235	1.62	642	0.50	8273974	0.53	106	105	14	0
ho_noprxs	1000	249750	9.32	0.20	0	0.00	0	0.00	11922427	0.20	428	101	570	0
ho	1000	249750	4.77	0.35	276	1.31	16	0.80	3574562	0.44	2	581	0	701
ho_nopr	1000	249750	5.48	0.15	0	0.00	0	0.00	5620753	0.21	6	587	0	992
ho_noxs	1000	249750	7.98	0.45	276	1.31	630	0.50	8281563	0.51	79	93	12	0
ho_noprxs	1000	249750	10.66	0.21	0	0.00	0	0.00	13513747	0.19	490	78	508	0
ho	1000	249750	2.75	0.71	810	0.46	12	2.00	1760322	1.07	1	125	0	174
ho_nopr	1000	249750	5.82	0.48	0	0.00	0	0.00	5924706	0.59	5	555	1	991
ho_noxs	1000	249750	3.79	1.09	810	0.46	153	2.00	3320990	1.50	29	16	5	0
ho_noprxs	1000	249750	11.25	0.07	0	0.00	0	0.00	14237605	0.07	508	55	490	0
ho	1000	249750	1.84	0.12	998	0.00	0	0.00	890688	0.25	1	2	0	0
ho_nopr	1000	249750	4.58	0.35	0	0.00	0	0.00	4207207	0.47	4	617	1	993
ho_noxs	1000	249750	1.82	0.11	998	0.00	0	0.00	890688	0.25	1	2	0	0
ho_noprxs	1000	249750	9.60	0.25	0	0.00	0	0.00	12353761	0.24	513	97	485	0
ho	1000	249750	1.69	0.01	998	0.00	0	0.00	748225	0.01	1	2	0	0
ho_nopr	1000	249750	3.02	0.30	0	0.00	0	0.00	2343088	0.48	3	708	0	995
ho_noxs	1000	249750	1.68	0.01	998	0.00	0	0.00	748225	0.01	1	2	0	0
ho_noprxs	1000	249750	7.98	0.27	0	0.00	0	0.00	10124108	0.26	535	153	463	0

Table 28: NOI5 family, HO codes

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	1000	249750	1.68	0.01	998	0.00	0	0.00	758572	0.00	1	2	0	0
ho_nopr	1000	249750	2.11	0.05	0	0.00	0	0.00	1226610	0.11	2	901	0	995
ho_noxs	1000	249750	1.70	0.01	998	0.00	0	0.00	758572	0.00	1	2	0	0
ho_noprxs	1000	249750	6.76	0.14	0	0.00	0	0.00	8561336	0.15	558	171	440	0
ho	1000	249750	1.77	0.00	998	0.00	0	0.00	872233	0.00	1	2	0	0
ho_nopr	1000	249750	2.24	0.03	0	0.00	0	0.00	1404123	0.06	2	968	0	995
ho_noxs	1000	249750	1.77	0.01	998	0.00	0	0.00	872233	0.00	1	2	0	0
ho_noprxs	1000	249750	11.59	0.58	0	0.00	0	0.00	14703598	0.56	615	89	383	0
ho	1000	249750	1.97	0.01	998	0.00	0	0.00	1131721	0.01	1	2	0	0
ho_nopr	1000	249750	2.67	0.05	0	0.00	0	0.00	1888840	0.06	3	920	11	984
ho_noxs	1000	249750	1.94	0.01	998	0.00	0	0.00	1131721	0.01	1	2	0	0
ho_noprxs	1000	249750	11.37	0.07	0	0.00	0	0.00	14053932	0.07	606	85	392	0
ho	1000	249750	1.97	0.01	998	0.00	0	0.00	1128305	0.01	1	2	0	0
ho_nopr	1000	249750	2.99	0.03	0	0.00	0	0.00	2341188	0.05	4	965	38	956
ho_noxs	1000	249750	1.98	0.01	998	0.00	0	0.00	1128305	0.01	1	2	0	0
ho_noprxs	1000	249750	14.40	0.17	0	0.00	0	0.00	17673766	0.18	546	83	452	0
ho	1000	249750	2.01	0.01	998	0.00	0	0.00	1183977	0.01	1	2	0	0
ho_nopr	1000	249750	3.79	0.03	0	0.00	0	0.00	3297812	0.05	6	965	50	942
ho_noxs	1000	249750	2.00	0.00	998	0.00	0	0.00	1183977	0.01	1	2	0	0
ho_noprxs	1000	249750	11.27	0.25	0	0.00	0	0.00	13978560	0.24	495	113	503	0
ho	1000	249750	2.03	0.01	998	0.00	0	0.00	1215232	0.00	1	2	0	0
ho_nopr	1000	249750	4.34	0.15	0	0.00	0	0.00	3939107	0.18	6	909	34	957
ho_noxs	1000	249750	2.01	0.01	998	0.00	0	0.00	1215232	0.00	1	2	0	0
ho_noprxs	1000	249750	12.70	0.13	0	0.00	0	0.00	15745458	0.13	442	115	556	0

Table 29: NOI5 family, HO codes (continued)

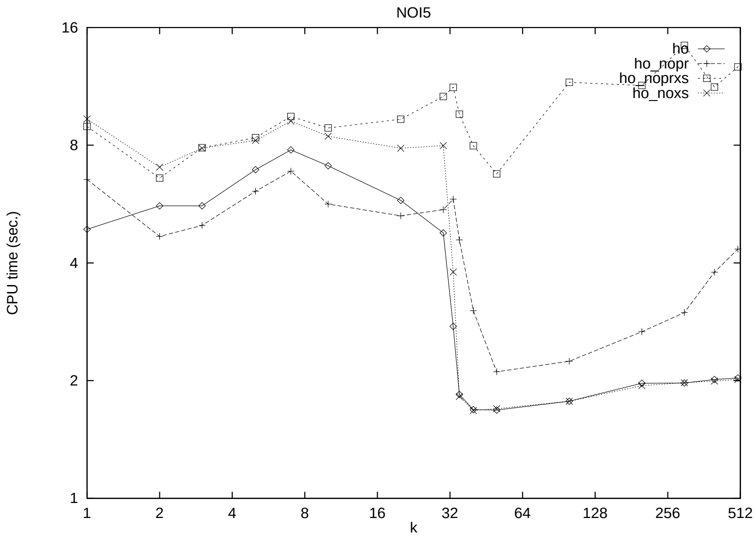

Figure 26: HO variants on NOI5 family

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	1000	249750	4.90	0.09	0	0.00	25	0.05	4340237	0.09	1	999	0	970
ho_nopr	1000	249750	3.50	0.07	0	0.00	0	0.00	3116479	0.10	5	682	0	994
ho_noxs	1000	249750	4.85	0.02	0	0.00	908	0.06	4573824	0.04	87	480	2	0
ho_noprxs	1000	249750	4.71	0.05	0	0.00	0	0.00	5214581	0.07	423	320	575	0
ho	1000	249750	4.74	0.07	0	0.00	117	0.23	4193474	0.08	2	999	0	879
ho_nopr	1000	249750	3.86	0.03	0	0.00	0	0.00	3536045	0.07	11	566	0	987
ho_noxs	1000	249750	5.68	0.02	0	0.00	820	0.01	5503175	0.05	176	459	1	0
ho_noprxs	1000	249750	5.49	0.03	0	0.00	0	0.00	6316071	0.04	352	344	646	0
ho	1000	249750	5.61	0.09	0	0.00	238	0.31	4855050	0.09	7	734	0	752
ho_nopr	1000	249750	4.69	0.02	0	0.00	0	0.00	4646956	0.03	17	530	0	982
ho_noxs	1000	249750	7.06	0.03	0	0.00	533	0.18	7395961	0.04	233	410	231	0
ho_noprxs	1000	249750	6.46	0.04	0	0.00	0	0.00	7770716	0.04	315	322	684	0
ho	1000	249750	6.64	0.13	0	0.00	265	0.23	6058229	0.17	12	665	57	663
ho_nopr	1000	249750	5.87	0.12	0	0.00	0	0.00	6497492	0.17	20	575	88	890
ho_noxs	1000	249750	8.36	0.05	0	0.00	444	0.08	9791101	0.06	108	405	445	0
ho_noprxs	1000	249750	8.56	0.06	0	0.00	0	0.00	11117566	0.07	155	297	844	0
ho	1000	249750	5.71	0.07	0	0.00	479	0.08	4980880	0.08	4	760	40	474
ho_nopr	1000	249750	5.71	0.08	0	0.00	0	0.00	6078152	0.09	17	616	78	903
ho_noxs	1000	249750	8.55	0.05	0	0.00	507	0.06	9765780	0.07	95	430	395	0
ho_noprxs	1000	249750	8.60	0.05	0	0.00	0	0.00	10852182	0.06	123	338	876	0
ho	1000	249750	5.17	0.09	0	0.00	628	0.06	4387209	0.09	4	928	13	351
ho_nopr	1000	249750	5.36	0.05	0	0.00	0	0.00	5571145	0.06	18	684	68	912
ho_noxs	1000	249750	8.43	0.05	0	0.00	452	0.08	9616467	0.05	102	493	443	0
ho_noprxs	1000	249750	8.23	0.08	0	0.00	0	0.00	10367525	0.08	136	380	862	0
ho	1000	249750	4.93	0.03	0	0.00	655	0.02	4155937	0.04	4	997	0	338
ho_nopr	1000	249750	5.14	0.05	0	0.00	0	0.00	5270417	0.06	15	839	61	922
ho_noxs	1000	249750	8.70	0.05	0	0.00	370	0.21	10161364	0.06	88	549	539	0
ho_noprxs	1000	249750	8.48	0.06	0	0.00	0	0.00	10749085	0.06	116	420	883	0
ho	1000	249750	4.92	0.02	0	0.00	677	0.03	4304370	0.02	4	998	0	316
ho_nopr	1000	249750	5.87	0.05	0	0.00	0	0.00	6186159	0.07	13	924	118	867
ho_noxs	1000	249750	9.22	0.03	0	0.00	414	0.27	10489253	0.05	83	699	501	0
ho_noprxs	1000	249750	8.65	0.03	0	0.00	0	0.00	10721057	0.04	111	531	887	0
ho	1000	249750	4.91	0.02	0	0.00	754	0.02	4287080	0.01	3	948	0	239
ho_nopr	1000	249750	6.29	0.04	0	0.00	0	0.00	6838809	0.04	19	919	256	724
ho_noxs	1000	249750	9.35	0.08	0	0.00	379	0.40	10607271	0.08	90	700	528	0
ho_noprxs	1000	249750	8.67	0.10	0	0.00	0	0.00	10653776	0.10	108	596	890	0

Table 30: NOI6 family, HO codes

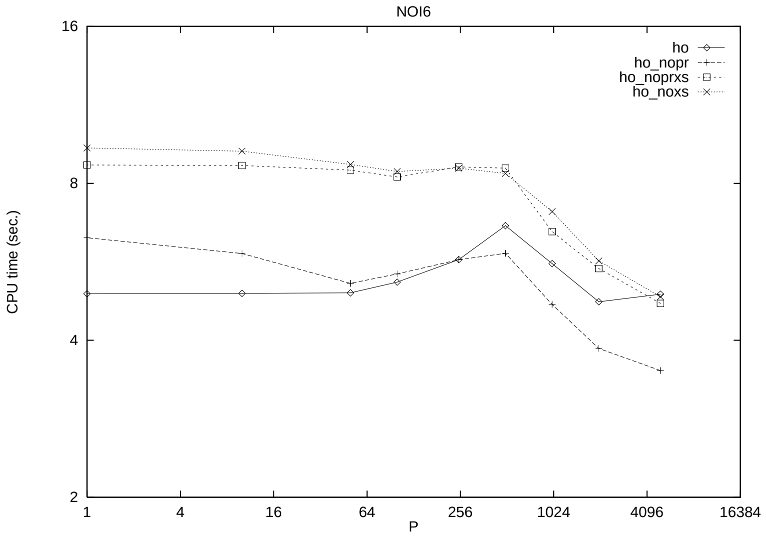

Figure 27: HO variants on NOI6 family

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	1000	8000	0.18	0.06	0	0.00	226	0.77	351789	0.12	10	969	88	673
ho_nopr	1000	8000	0.15	0.08	0	0.00	0	0.00	305199	0.06	12	902	250	736
ho_noxs	1000	8000	0.29	0.05	0	0.00	561	0.07	609150	0.07	38	569	397	0
ho_noprxs	1000	8000	0.26	0.09	0	0.00	0	0.00	588382	0.08	79	293	920	0
ho	1000	16000	0.45	0.12	0	0.00	583	0.22	714757	0.13	10	986	29	374
ho_nopr	1000	16000	0.42	0.14	0	0.00	0	0.00	753260	0.11	12	917	397	588
ho_noxs	1000	16000	0.59	0.02	0	0.00	555	0.10	1122567	0.03	33	624	409	0
ho_noprxs	1000	16000	0.52	0.06	0	0.00	0	0.00	1094614	0.04	58	381	941	0
ho	1000	32000	1.04	0.11	0	0.00	734	0.22	1293980	0.12	9	979	57	197
ho_nopr	1000	32000	1.20	0.10	0	0.00	0	0.00	1798196	0.12	13	918	597	388
ho_noxs	1000	32000	1.44	0.12	0	0.00	408	0.22	2196233	0.13	28	634	561	0
ho_noprxs	1000	32000	1.33	0.12	0	0.00	0	0.00	2185559	0.13	36	525	962	0
ho	1000	62500	2.10	0.15	0	0.00	863	0.03	2413652	0.16	8	991	6	119
ho_nopr	1000	62500	2.51	0.07	0	0.00	0	0.00	3434358	0.07	13	908	691	294
ho_noxs	1000	62500	2.93	0.08	0	0.00	467	0.23	4051979	0.07	24	700	506	0
ho_noprxs	1000	62500	2.80	0.08	0	0.00	0	0.00	4126624	0.08	29	606	969	0
ho	1000	125000	3.98	0.27	0	0.00	898	0.06	4166085	0.30	7	944	45	47
ho_nopr	1000	125000	6.14	0.06	0	0.00	0	0.00	7806199	0.05	12	873	813	173
ho_noxs	1000	125000	6.13	0.06	0	0.00	298	0.40	7767430	0.07	23	661	676	0
ho_noprxs	1000	125000	6.05	0.08	0	0.00	0	0.00	7870539	0.08	28	559	970	0
ho	1000	250000	7.82	0.25	0	0.00	767	0.19	7375673	0.30	5	922	184	41
ho_nopr	1000	250000	11.92	0.12	0	0.00	0	0.00	14154803	0.11	10	872	846	141
ho_noxs	1000	250000	12.53	0.04	0	0.00	399	0.36	14803088	0.03	23	562	574	0
ho_noprxs	1000	250000	12.12	0.05	0	0.00	0	0.00	15038257	0.04	30	446	968	0
ho	1000	499500	11.41	0.14	0	0.00	972	0.01	9394579	0.19	5	996	0	19
ho_nopr	1000	499500	20.56	0.08	0	0.00	0	0.00	23210488	0.08	12	875	896	89
ho_noxs	1000	499500	20.65	0.10	0	0.00	340	0.38	22454806	0.10	26	480	630	0
ho_noprxs	1000	499500	19.70	0.11	0	0.00	0	0.00	22559161	0.11	31	423	968	0

Table 31: RAND1 family, HO codes

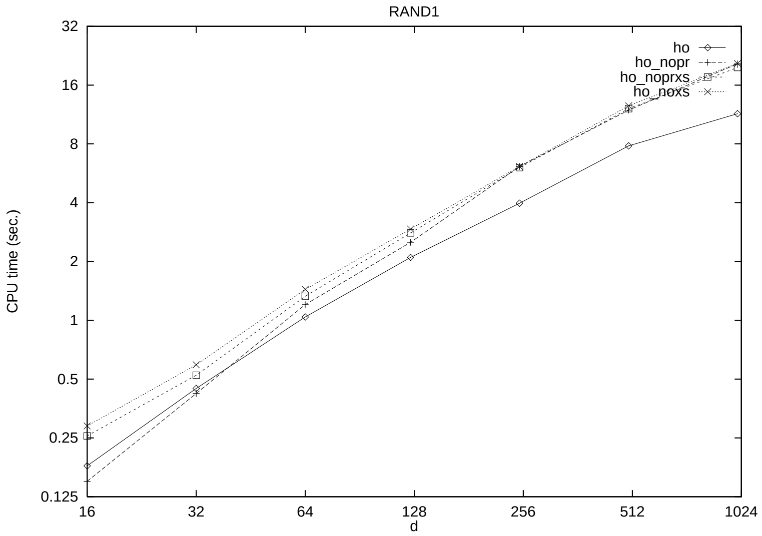

Figure 28: HO variants on RAND1 family

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	500	12500	0.24	0.11	0	0.00	361	0.10	519648	0.13	7	492	12	116
ho_nopr	500	12500	0.24	0.14	0	0.00	0	0.00	592535	0.13	10	426	290	198
ho_noxs	500	12500	0.30	0.14	0	0.00	253	0.24	736372	0.12	20	312	223	0
ho_noprxs	500	12500	0.28	0.18	0	0.00	0	0.00	754750	0.14	28	237	470	0
ho	708	17700	0.51	0.18	0	0.00	481	0.21	794567	0.18	8	685	35	180
ho_nopr	708	17700	0.47	0.14	0	0.00	0	0.00	867981	0.16	12	612	358	336
ho_noxs	708	17700	0.59	0.05	0	0.00	348	0.21	1094099	0.04	29	436	328	0
ho_noprxs	708	17700	0.52	0.02	0	0.00	0	0.00	1094242	0.03	39	343	667	0
ho	1000	25000	0.80	0.17	0	0.00	695	0.16	1086410	0.18	9	962	106	187
ho_nopr	1000	25000	0.91	0.08	0	0.00	0	0.00	1417207	0.08	12	885	590	395
ho_noxs	1000	25000	1.04	0.09	0	0.00	528	0.13	1659524	0.11	26	585	442	0
ho_noprxs	1000	25000	0.96	0.10	0	0.00	0	0.00	1664085	0.10	41	418	957	0
ho	1414	35350	1.40	0.15	0	0.00	989	0.11	1718330	0.15	10	1398	35	376
ho_nopr	1414	35350	1.37	0.18	0	0.00	0	0.00	1974699	0.16	14	1213	731	666
ho_noxs	1414	35350	1.72	0.05	0	0.00	655	0.07	2568381	0.06	33	887	722	0
ho_noprxs	1414	35350	1.55	0.05	0	0.00	0	0.00	2539180	0.06	45	691	1367	0
ho	2000	50000	2.23	0.11	0	0.00	1366	0.13	2572105	0.11	14	1958	32	583
ho_nopr	2000	50000	2.22	0.16	0	0.00	0	0.00	3025812	0.15	17	1825	1090	890
ho_noxs	2000	50000	2.92	0.10	0	0.00	1108	0.19	4018708	0.13	41	1243	848	0
ho_noprxs	2000	50000	2.66	0.11	0	0.00	0	0.00	4107044	0.12	61	901	1938	0
ho	2828	70700	3.25	0.10	0	0.00	2090	0.11	3534336	0.11	14	2791	133	588
ho_nopr	2828	70700	3.39	0.09	0	0.00	0	0.00	4362321	0.08	17	2633	1782	1027
ho_noxs	2828	70700	4.68	0.06	0	0.00	1288	0.28	5963298	0.06	41	1585	1496	0
ho_noprxs	2828	70700	4.17	0.05	0	0.00	0	0.00	5963508	0.05	69	1040	2757	0
ho	4000	100000	5.73	0.12	0	0.00	2743	0.13	5582265	0.12	13	3910	203	1038
ho_nopr	4000	100000	5.75	0.09	0	0.00	0	0.00	6462203	0.10	16	3612	2254	1728
ho_noxs	4000	100000	7.89	0.07	0	0.00	1833	0.22	9137102	0.07	46	2238	2118	0
ho_noprxs	4000	100000	6.98	0.08	0	0.00	0	0.00	9156698	0.08	69	1612	3929	0
ho	5656	141400	7.45	0.14	0	0.00	4405	0.08	6921169	0.14	16	5509	212	1019
ho_nopr	5656	141400	8.88	0.10	0	0.00	0	0.00	9643072	0.09	20	5203	3296	2338
ho_noxs	5656	141400	12.52	0.13	0	0.00	2410	0.30	13778083	0.13	55	3305	3188	0
ho_noprxs	5656	141400	11.25	0.12	0	0.00	0	0.00	13674790	0.13	77	2597	5577	0
ho	8000	200000	13.43	0.14	0	0.00	6289	0.06	11742261	0.14	14	7842	84	1609
ho_nopr	8000	200000	15.10	0.07	0	0.00	0	0.00	15105243	0.06	18	7237	4568	3412
ho_noxs	8000	200000	19.48	0.08	0	0.00	4018	0.18	19919669	0.09	53	4104	3925	0
ho_noprxs	8000	200000	17.93	0.07	0	0.00	0	0.00	19980712	0.08	97	2694	7902	0

Table 32: RAND2 family, HO codes

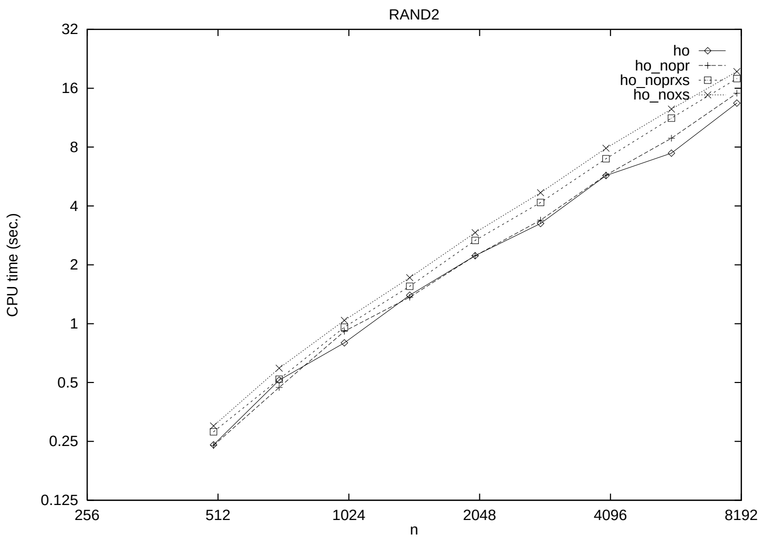

Figure 29: HO variants on RAND2 family

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	1001	1001	0.00	0.00	999	0.00	0	0.00	5002	0.00	1	2	0	0
ho_nopr	1001	1001	0.01	1.22	0	0.00	0	0.00	7992	0.00	1	1001	0	999
ho_noxs	1001	1001	0.01	1.22	999	0.00	0	0.00	5002	0.00	1	2	0	0
ho_noprxs	1001	1001	0.01	1.22	0	0.00	0	0.00	6993	0.00	1000	501	0	0
ho	1001	3003	0.14	0.13	0	0.00	635	0.05	205983	0.15	10	918	16	337
ho_nopr	1001	3003	0.17	0.31	0	0.00	0	0.00	285103	0.30	21	675	327	650
ho_noxs	1001	3003	0.21	0.13	0	0.00	740	0.07	354791	0.11	34	533	224	0
ho_noprxs	1001	3003	0.24	0.07	0	0.00	0	0.00	403142	0.08	141	165	858	0
ho	1001	10010	0.38	0.14	0	0.00	617	0.12	809440	0.14	12	840	155	213
ho_nopr	1001	10010	0.35	0.18	0	0.00	0	0.00	866459	0.21	17	710	679	303
ho_noxs	1001	10010	0.39	0.03	0	0.00	516	0.16	819082	0.05	30	491	451	0
ho_noprxs	1001	10010	0.31	0.03	0	0.00	0	0.00	800136	0.04	59	291	941	0
ho	1001	33033	1.65	0.09	0	0.00	706	0.24	2192055	0.10	10	842	214	68
ho_nopr	1001	33033	1.70	0.10	0	0.00	0	0.00	2688754	0.09	12	829	894	93
ho_noxs	1001	33033	1.58	0.11	0	0.00	382	0.29	2481260	0.10	24	501	592	0
ho_noprxs	1001	33033	1.50	0.10	0	0.00	0	0.00	2473762	0.11	33	401	966	0
ho	1001	100100	4.29	0.12	0	0.00	629	0.37	4790970	0.11	7	844	345	17
ho_nopr	1001	100100	5.37	0.09	0	0.00	0	0.00	7332057	0.09	12	801	952	35
ho_noxs	1001	100100	5.00	0.03	0	0.00	358	0.31	6916701	0.04	19	547	620	0
ho_noprxs	1001	100100	4.94	0.04	0	0.00	0	0.00	7013081	0.04	23	474	976	0
ho	1001	333333	12.26	0.07	0	0.00	883	0.22	11752797	0.07	5	947	104	5
ho_nopr	1001	333333	16.06	0.08	0	0.00	0	0.00	19683521	0.07	11	842	971	17
ho_noxs	1001	333333	17.31	0.03	0	0.00	312	0.35	20462693	0.02	17	561	669	0
ho_noprxs	1001	333333	16.88	0.02	0	0.00	0	0.00	20683326	0.02	20	493	979	0
ho	1001	1001000	21.25	0.25	0	0.00	989	0.01	15623660	0.31	4	866	3	2
ho_nopr	1001	1001000	35.15	0.08	0	0.00	0	0.00	37474290	0.08	11	830	976	11
ho_noxs	1001	1001000	37.18	0.05	0	0.00	197	0.41	37276534	0.07	18	523	782	0
ho_noprxs	1001	1001000	35.19	0.06	0	0.00	0	0.00	37398700	0.07	21	460	978	0

Table 33: REG1 family, HO codes

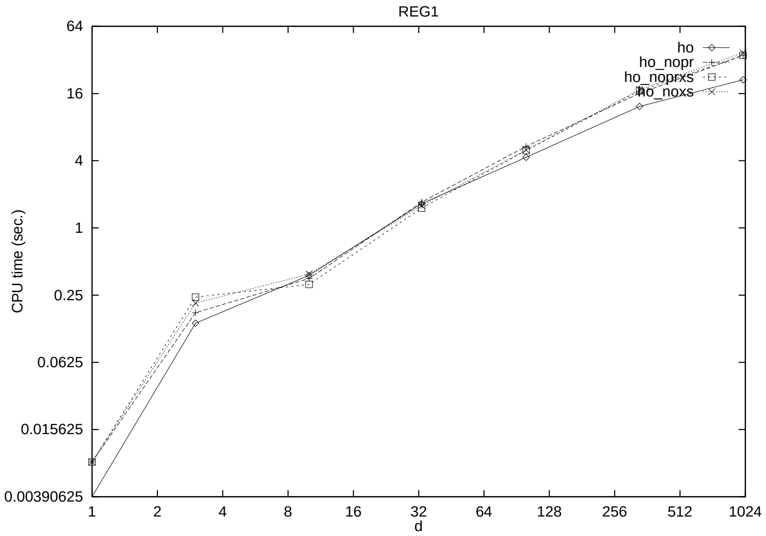

Figure 30: HO variants on REG1 family

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	50	2500	0.02	0.00	0	0.00	42	0.02	30955	0.05	3	48	1	2
ho_nopr	50	2500	0.02	0.55	0	0.00	0	0.00	39304	0.08	6	38	36	6
ho_noxs	50	2500	0.02	0.20	0	0.00	22	0.35	41225	0.06	8	29	18	0
ho_noprxs	50	2500	0.02	0.00	0	0.00	0	0.00	39480	0.08	11	22	37	0
ho	100	5000	0.05	0.36	0	0.00	72	0.21	112675	0.21	4	84	16	4
ho_nopr	100	5000	0.06	0.17	0	0.00	0	0.00	141232	0.06	7	82	80	10
ho_noxs	100	5000	0.07	0.12	0	0.00	49	0.22	154192	0.11	9	62	38	0
ho_noprxs	100	5000	0.06	0.16	0	0.00	0	0.00	153330	0.12	14	46	84	0
ho	200	10000	0.17	0.11	0	0.00	154	0.19	354763	0.10	5	191	32	6
ho_nopr	200	10000	0.15	0.10	0	0.00	0	0.00	423872	0.09	9	160	175	15
ho_noxs	200	10000	0.17	0.06	0	0.00	100	0.20	439598	0.07	13	120	85	0
ho_noprxs	200	10000	0.17	0.06	0	0.00	0	0.00	443174	0.07	18	90	180	0
ho	400	20000	0.56	0.11	0	0.00	291	0.17	872105	0.13	7	352	83	16
ho_nopr	400	20000	0.60	0.09	0	0.00	0	0.00	1153398	0.07	10	327	354	34
ho_noxs	400	20000	0.69	0.08	0	0.00	155	0.36	1251946	0.07	16	227	227	0
ho_noprxs	400	20000	0.66	0.08	0	0.00	0	0.00	1269251	0.09	20	183	378	0
ho	800	40000	2.00	0.17	0	0.00	646	0.08	2556624	0.18	7	735	115	29
ho_nopr	800	40000	2.03	0.06	0	0.00	0	0.00	3074250	0.08	11	651	716	70
ho_noxs	800	40000	1.96	0.05	0	0.00	332	0.33	2991965	0.06	20	458	445	0
ho_noprxs	800	40000	1.88	0.05	0	0.00	0	0.00	3028397	0.05	26	374	772	0

Table 34: REG2 family, HO codes

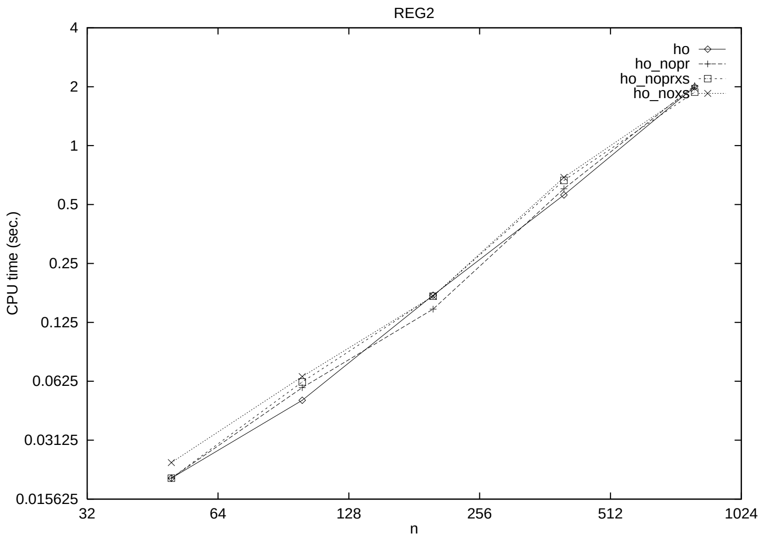

Figure 31: HO variants on REG2 family

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	256	512	0.02	0.50	4	0.44	139	0.08	20739	0.16	5	238	11	93
ho_nopr	256	512	0.02	0.00	0	0.00	0	0.00	23226	0.15	12	168	42	200
ho_noxs	256	512	0.02	0.20	4	0.44	196	0.02	43600	0.09	21	129	32	0
ho_noprxs	256	512	0.03	0.14	0	0.00	0	0.00	45644	0.09	85	46	169	0
ho	512	1024	0.04	0.26	4	0.27	318	0.13	51902	0.11	7	470	12	166
ho_nopr	512	1024	0.03	0.31	0	0.00	0	0.00	55854	0.15	14	305	94	402
ho_noxs	512	1024	0.07	0.16	4	0.27	407	0.02	103282	0.11	27	277	71	0
ho_noprxs	512	1024	0.09	0.11	0	0.00	0	0.00	135290	0.06	158	64	352	0
ho	1024	2048	0.09	0.23	6	0.32	651	0.03	124559	0.11	9	949	17	338
ho_nopr	1024	2048	0.15	0.15	0	0.00	0	0.00	199834	0.18	23	655	169	831
ho_noxs	1024	2048	0.23	0.24	6	0.32	824	0.03	312465	0.24	44	474	147	0
ho_noprxs	1024	2048	0.22	0.18	0	0.00	0	0.00	315307	0.16	273	88	749	0
ho	2048	4096	0.31	0.29	6	0.30	1291	0.02	401943	0.30	13	1861	42	692
ho_nopr	2048	4096	0.46	0.37	0	0.00	0	0.00	591134	0.37	30	1290	374	1642
ho_noxs	2048	4096	0.52	0.23	6	0.30	1702	0.01	695879	0.23	51	908	286	0
ho_noprxs	2048	4096	0.61	0.12	0	0.00	0	0.00	855309	0.14	511	124	1535	0
ho	4096	8192	1.04	0.16	5	0.36	2674	0.04	1177472	0.15	19	3621	35	1359
ho_nopr	4096	8192	1.25	0.48	0	0.00	0	0.00	1517555	0.48	35	2798	706	3353
ho_noxs	4096	8192	1.81	0.15	5	0.36	3410	0.01	2101631	0.15	85	1547	593	0
ho_noprxs	4096	8192	1.49	0.10	0	0.00	0	0.00	1944874	0.10	927	150	3167	0
ho	8192	16384	4.14	0.39	5	0.22	5400	0.02	3589940	0.40	28	7412	118	2638
ho_nopr	8192	16384	5.44	0.55	0	0.00	0	0.00	5109417	0.56	51	6095	1447	6692
ho_noxs	8192	16384	5.40	0.20	5	0.22	6857	0.01	4794858	0.21	111	2713	1215	0
ho_noprxs	8192	16384	4.90	0.15	0	0.00	0	0.00	4746152	0.14	1899	180	6291	0
ho	16384	32768	18.53	0.37	5	0.61	10651	0.07	13903247	0.38	55	14339	230	5439
ho_nopr	16384	32768	10.64	0.15	0	0.00	0	0.00	8406198	0.15	43	12190	2537	13803
ho_noxs	16384	32768	21.92	0.18	5	0.61	13724	0.00	16689261	0.18	208	5095	2444	0
ho_noprxs	16384	32768	12.68	0.14	0	0.00	0	0.00	10534611	0.15	3329	236	13054	0

Table 35: REG3 family, HO codes

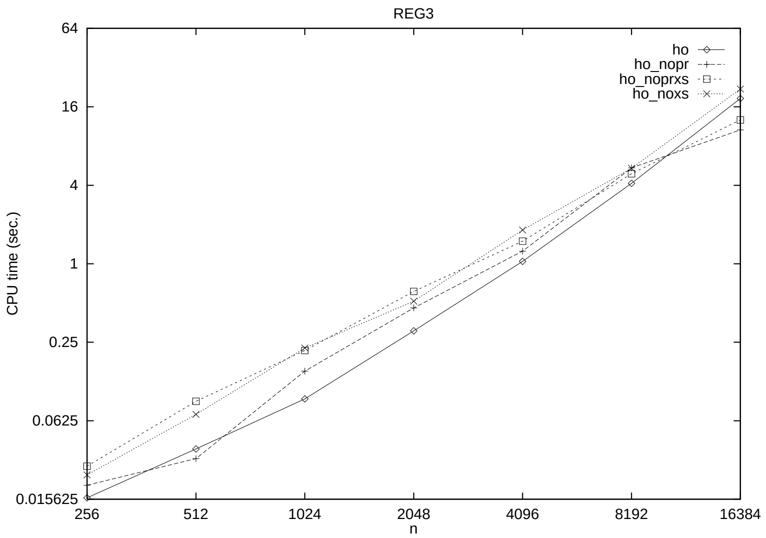

Figure 32: HO variants on REG3 family

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	128	1024	0.02	0.18	0	0.00	73	0.21	43691	0.04	7	106	15	29
ho_nopr	128	1024	0.02	0.00	0	0.00	0	0.00	42064	0.11	11	86	67	47
ho_noxs	128	1024	0.02	0.00	0	0.00	69	0.16	47055	0.08	13	72	44	0
ho_noprxs	128	1024	0.02	0.18	0	0.00	0	0.00	45797	0.06	21	50	105	0
ho	256	4096	0.09	0.11	0	0.00	190	0.09	196194	0.08	6	226	28	29
ho_nopr	256	4096	0.08	0.05	0	0.00	0	0.00	217538	0.04	12	171	190	53
ho_noxs	256	4096	0.08	0.10	0	0.00	136	0.17	226665	0.10	16	150	101	0
ho_noprxs	256	4096	0.08	0.15	0	0.00	0	0.00	223750	0.12	26	99	228	0
ho	512	16384	0.48	0.06	0	0.00	366	0.23	837741	0.06	7	467	105	31
ho_nopr	512	16384	0.51	0.05	0	0.00	0	0.00	1073405	0.05	10	415	445	55
ho_noxs	512	16384	0.53	0.17	0	0.00	210	0.26	1118930	0.19	19	270	279	0
ho_noprxs	512	16384	0.51	0.18	0	0.00	0	0.00	1114198	0.19	26	214	484	0
ho	1024	65536	2.94	0.14	0	0.00	920	0.05	3586320	0.16	9	857	61	31
ho_nopr	1024	65536	3.33	0.07	0	0.00	0	0.00	4840474	0.07	13	796	945	63
ho_noxs	1024	65536	3.39	0.09	0	0.00	394	0.25	4802006	0.09	21	553	606	0
ho_noprxs	1024	65536	3.28	0.10	0	0.00	0	0.00	4829193	0.10	26	474	996	0
ho	2048	262144	15.28	0.12	0	0.00	1834	0.10	16028877	0.14	9	1736	174	28
ho_nopr	2048	262144	18.24	0.13	0	0.00	0	0.00	23116059	0.13	14	1654	1972	61
ho_noxs	2048	262144	17.82	0.13	0	0.00	731	0.43	21885382	0.12	22	1089	1292	0
ho_noprxs	2048	262144	16.77	0.13	0	0.00	0	0.00	22214770	0.12	26	942	2020	0

Table 36: REG4 family, HO codes

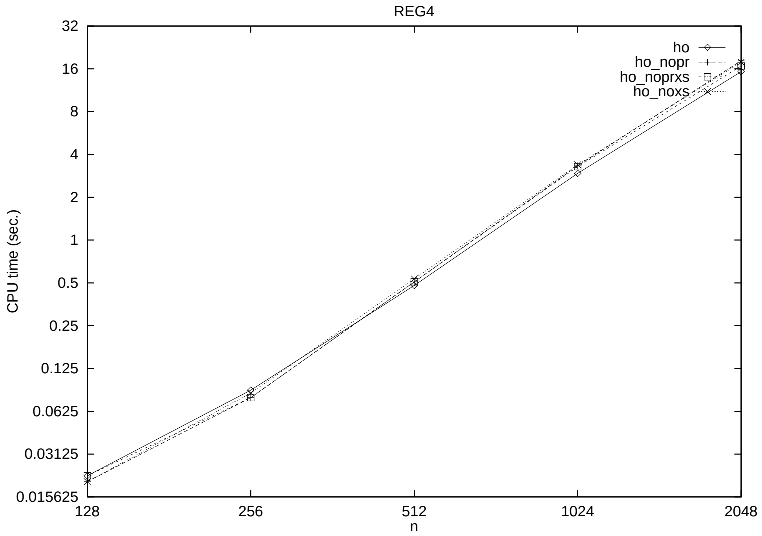

Figure 33: HO variants on REG4 family

	nodes	arcs	total time		initial PR		internal PR		edge scans		st cuts	avg. size	1 node	excess cont
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	avg	avg	avg
ho	1024	2044	0.26	0.05	0	0.00	9	0.00	578086	0.00	9	912	988	16
ho_nopr	1024	2044	0.28	0.07	0	0.00	0	0.00	692716	0.00	10	922	997	16
ho_noxs	1024	2044	0.22	0.09	0	0.00	510	0.00	473723	0.00	10	972	502	0
ho_noprxs	1024	2044	0.27	0.04	0	0.00	0	0.00	693736	0.00	11	931	1012	0
ho	2048	4092	1.58	0.01	0	0.00	10	0.00	3141891	0.00	10	1845	2008	18
ho_nopr	2048	4092	1.66	0.01	0	0.00	0	0.00	3601485	0.00	11	1862	2018	18
ho_noxs	2048	4092	1.38	0.02	0	0.00	961	0.00	2589304	0.00	20	1532	1065	0
ho_noprxs	2048	4092	1.55	0.01	0	0.00	0	0.00	3406832	0.00	22	1437	2025	0
ho	4096	8188	6.85	0.01	0	0.00	11	0.00	11277803	0.00	11	3725	4052	20
ho_nopr	4096	8188	5.83	0.01	0	0.00	0	0.00	13063147	0.00	12	3755	4063	20
ho_noxs	4096	8188	6.02	0.01	0	0.00	1921	0.00	9204733	0.00	23	3019	2150	0
ho_noprxs	4096	8188	5.67	0.01	0	0.00	0	0.00	12579995	0.00	25	2854	4070	0
ho	8192	16380	28.80	0.00	0	0.00	12	0.00	39571898	0.00	12	7511	8144	22
ho_nopr	8192	16380	26.26	0.00	0	0.00	0	0.00	47916126	0.00	13	7562	8156	22
ho_noxs	8192	16380	27.19	0.00	0	0.00	3841	0.00	34117188	0.00	26	5969	4323	0
ho_noprxs	8192	16380	24.37	0.00	0	0.00	0	0.00	42208186	0.00	30	5429	8161	0

Table 37: BIKEWHE family, HO codes

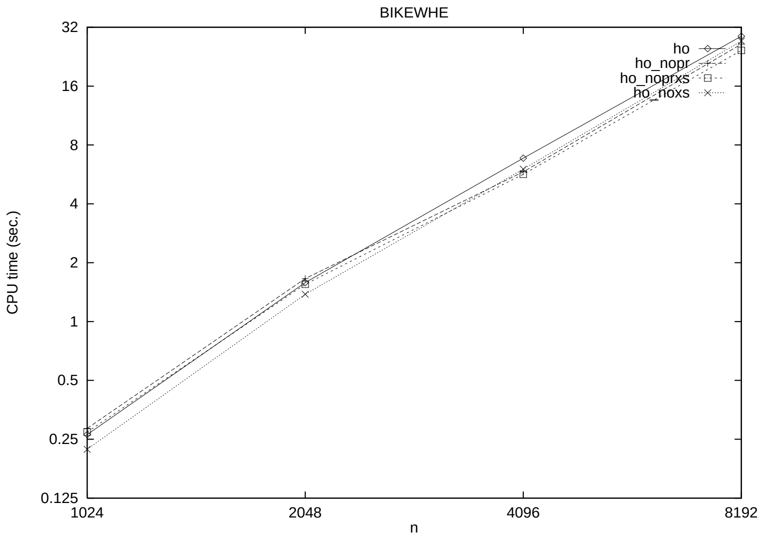

Figure 34: HO variants on BIKEWHE family

problem		nodes	arcs	total time	initial PR	internal PR	edge scans	st cuts	avg. size	1 node	excess cont
1	ho	532	787	0.02	487	14	10052	9	16	7	13
1	ho_nopr	532	787	0.05	0	0	61827	65	54	15	451
1	ho_noxs	532	787	0.02	487	8	10155	23	9	12	0
1	ho_noprxs	532	787	0.05	0	0	71544	480	15	51	0
2	ho	1291	1942	0.03	976	58	64710	83	25	49	124
2	ho_nopr	1291	1942	0.22	0	0	294605	215	63	90	985
2	ho_noxs	1291	1942	0.03	976	55	61664	165	14	93	0
2	ho_noprxs	1291	1942	0.22	0	0	304794	1143	19	147	0
3	ho	1400	2231	0.07	1091	47	84281	88	50	51	121
3	ho_nopr	1400	2231	0.17	0	0	239127	291	123	110	998
3	ho_noxs	1400	2231	0.07	1091	55	89283	179	33	73	0
3	ho_noprxs	1400	2231	0.28	0	0	321558	1170	149	229	0
4	ho	76	90	0.00	74	0	692	1	2	0	0
4	ho_nopr	76	90	0.02	0	0	2341	7	36	0	68
4	ho_noxs	76	90	0.02	74	0	692	1	2	0	0
4	ho_noprxs	76	90	0.02	0	0	3375	73	25	2	0
5	ho	5934	7287	0.08	5651	35	93504	40	43	54	152
5	ho_nopr	5934	7287	1.90	0	0	2133047	235	581	93	5605
5	ho_noxs	5934	7287	0.12	5651	85	109171	130	22	66	0
5	ho_noprxs	5934	7287	2.00	0	0	2320676	5563	74	370	0
6	ho	5934	7627	0.15	5444	51	157298	75	55	112	251
6	ho_nopr	5934	7627	1.73	0	0	2079106	228	259	158	5547
6	ho_noxs	5934	7627	0.15	5444	103	166965	227	28	159	0
6	ho_noprxs	5934	7627	1.55	0	0	1901007	5551	34	382	0
7	ho	1323	2169	0.03	1115	43	51818	40	38	48	75
7	ho_nopr	1323	2169	0.18	0	0	266752	175	75	72	1075
7	ho_noxs	1323	2169	0.03	1115	51	53321	96	22	59	0
7	ho_noprxs	1323	2169	0.23	0	0	343691	1188	22	134	0
8	ho	1323	2195	0.05	1167	38	48740	37	39	44	35
8	ho_nopr	1323	2195	0.28	0	0	389352	148	81	93	1081
8	ho_noxs	1323	2195	0.03	1167	28	50332	72	23	54	0
8	ho_noprxs	1323	2195	0.25	0	0	392530	1166	19	156	0
9	ho	1084	1252	0.02	1052	7	11580	7	15	5	11
9	ho_nopr	1084	1252	0.10	0	0	136202	90	163	8	985
9	ho_noxs	1084	1252	0.00	1052	12	11549	15	12	3	0
9	ho_noprxs	1084	1252	0.22	0	0	208920	1037	122	46	0
10	ho	1748	2336	0.03	1611	33	39664	34	32	24	44
10	ho_nopr	1748	2336	0.28	0	0	330121	213	126	55	1479
10	ho_noxs	1748	2336	0.03	1611	39	39138	69	17	27	0
10	ho_noprxs	1748	2336	0.22	0	0	293364	1623	39	124	0
11	ho	15112	19057	0.48	13912	188	443071	284	43	250	477
11	ho_nopr	15112	19057	6.07	0	0	5855678	1910	107	335	12866
11	ho_noxs	15112	19057	0.53	13912	151	447929	693	19	354	0
11	ho_noprxs	15112	19057	5.95	0	0	6079117	14081	26	1030	0

Table 38: TSP misc family, HO codes

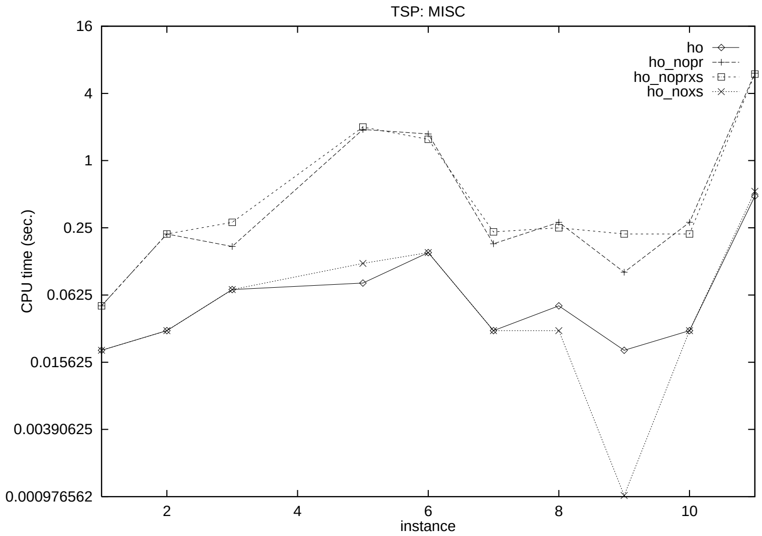

Figure 35: HO variants on TSP misc family

problem		nodes	arcs	total time	initial PR	internal PR	edge scans	st cuts	avg. size	1 node	excess cont
12	ho	33810	38600	0.48	33808	0	271169	1	2	0	0
12	ho_nopr	33810	38600	10.47	0	0	8664468	151	2121	12	33646
12	ho_noxs	33810	38600	0.47	33808	0	271169	1	2	0	0
12	ho_noprxs	33810	38600	15.75	0	0	14274390	32201	165	1608	0
13	ho	33810	39367	0.52	33808	0	303275	1	2	0	0
13	ho_nopr	33810	39367	11.15	0	0	8031804	283	4226	51	33475
13	ho_noxs	33810	39367	0.52	33808	0	303275	1	2	0	0
13	ho_noprxs	33810	39367	23.57	0	0	19183112	31796	707	2013	0
14	ho	33810	39456	0.52	33808	0	295708	1	2	0	0
14	ho_nopr	33810	39456	8.40	0	0	6711437	368	1257	53	33388
14	ho_noxs	33810	39456	0.50	33808	0	295708	1	2	0	0
14	ho_noprxs	33810	39456	18.33	0	0	15990331	31845	305	1964	0
15	ho	85900	102596	1.60	85898	0	810397	1	2	0	0
15	ho_nopr	85900	102596	9.97	0	0	6894899	202	3915	21	85676
15	ho_noxs	85900	102596	1.58	85898	0	810397	1	2	0	0
15	ho_noprxs	85900	102596	71.53	0	0	63755623	79725	248	6174	0
16	ho	85900	102934	1.63	85898	0	855359	1	2	0	0
16	ho_nopr	85900	102934	24.82	0	0	18005276	477	6293	134	85288
16	ho_noxs	85900	102934	1.65	85898	0	855359	1	2	0	0
16	ho_noprxs	85900	102934	70.40	0	0	55543971	79900	241	5999	0
17	ho	85900	102988	1.67	85830	3	869204	4	47	0	61
17	ho_nopr	85900	102988	45.42	0	0	34455038	674	5491	160	85065
17	ho_noxs	85900	102988	1.68	85830	39	870252	19	19	10	0
17	ho_noprxs	85900	102988	70.35	0	0	59571910	79931	209	5968	0

Table 39: TSP PLA family, HO codes

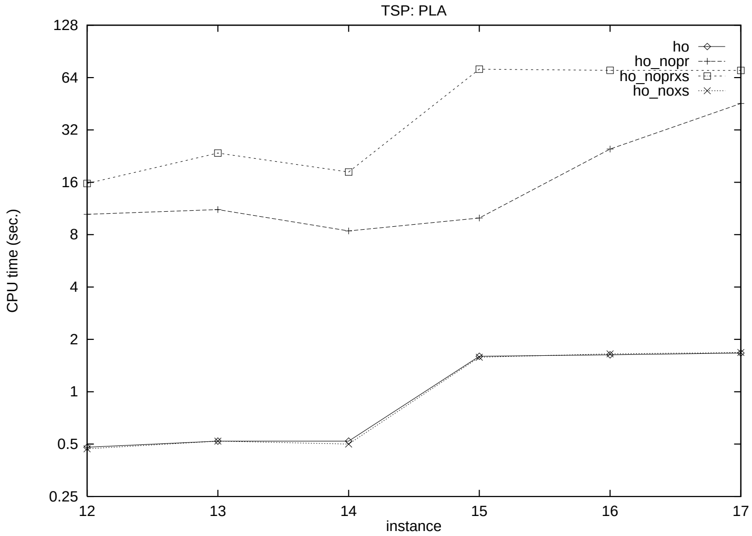

Figure 36: HO variants on TSP PLA family

problem		nodes	arcs	total time	initial PR	internal PR	edge scans	st cuts	avg. size	1 node	excess cont
18	ho	13509	15631	0.27	12976	89	197589	77	45	64	302
18	ho_nopr	13509	15631	4.95	0	0	4594728	803	261	127	12578
18	ho_noxs	13509	15631	0.32	12976	122	239010	279	33	130	0
18	ho_noprxs	13509	15631	4.93	0	0	4866661	12733	41	775	0
19	ho	13509	17048	0.43	12492	194	376734	287	43	187	348
19	ho_nopr	13509	17048	6.07	0	0	5854604	1580	113	292	11636
19	ho_noxs	13509	17048	0.48	12492	190	418972	557	26	268	0
19	ho_noprxs	13509	17048	5.97	0	0	6051171	12591	34	917	0
20	ho	13509	17079	0.42	12456	219	376394	250	44	227	356
20	ho_nopr	13509	17079	6.53	0	0	6301508	1662	118	284	11562
20	ho_noxs	13509	17079	0.42	12456	191	367502	555	21	305	0
20	ho_noprxs	13509	17079	6.10	0	0	6188473	12615	35	893	0
21	ho	13509	17111	0.37	12707	148	304868	212	36	186	255
21	ho_nopr	13509	17111	5.95	0	0	5663962	1724	135	305	11479
21	ho_noxs	13509	17111	0.38	12707	163	320219	406	18	231	0
21	ho_noprxs	13509	17111	5.77	0	0	5919777	12607	29	901	0
22	ho	13509	17130	0.35	12725	97	276235	79	64	94	513
22	ho_nopr	13509	17130	5.53	0	0	5374962	777	195	172	12559
22	ho_noxs	13509	17130	0.38	12725	266	328839	329	27	187	0
22	ho_noprxs	13509	17130	5.50	0	0	5593513	12558	32	950	0
23	ho	13509	17156	0.42	12649	194	359133	200	40	191	273
23	ho_nopr	13509	17156	5.32	0	0	5072396	1735	141	310	11463
23	ho_noxs	13509	17156	0.42	12649	178	357124	426	25	254	0
23	ho_noprxs	13509	17156	5.20	0	0	5297156	12605	67	903	0
24	ho	13509	17156	0.40	12684	180	330043	181	48	185	278
24	ho_nopr	13509	17156	5.05	0	0	4869890	1728	103	303	11477
24	ho_noxs	13509	17156	0.42	12684	196	334790	389	39	238	0
24	ho_noprxs	13509	17156	5.12	0	0	5228046	12624	34	884	0
25	ho	13509	17183	0.40	12419	141	344153	185	43	141	621
25	ho_nopr	13509	17183	5.60	0	0	5472719	995	192	211	12302
25	ho_noxs	13509	17183	0.50	12419	243	463694	536	30	309	0
25	ho_noprxs	13509	17183	5.65	0	0	5827803	12565	37	943	0
26	ho	13509	17193	0.45	12427	242	394799	280	39	220	339
26	ho_nopr	13509	17193	5.87	0	0	5739492	1703	109	321	11484
26	ho_noxs	13509	17193	0.48	12427	176	440442	596	23	308	0
26	ho_noprxs	13509	17193	5.78	0	0	5992762	12595	33	913	0
27	ho	13509	17210	0.37	12645	175	327103	183	33	188	316
27	ho_nopr	13509	17210	5.27	0	0	5244261	1479	124	285	11744
27	ho_noxs	13509	17210	0.38	12645	175	332757	445	21	242	0
27	ho_noprxs	13509	17210	5.22	0	0	5449933	12577	35	931	0
28	ho	13509	17303	0.50	12402	259	458091	306	50	212	328
28	ho_nopr	13509	17303	5.40	0	0	5248136	1791	101	325	11392
28	ho_noxs	13509	17303	0.53	12402	229	471048	571	35	305	0
28	ho_noprxs	13509	17303	5.03	0	0	5124583	12611	31	897	0
29	ho	13509	17358	0.42	12582	215	355828	243	37	183	284
29	ho_nopr	13509	17358	5.77	0	0	5625720	1856	113	372	11280
29	ho_noxs	13509	17358	0.43	12582	188	367622	477	20	260	0
29	ho_noprxs	13509	17358	5.57	0	0	5812456	12512	30	996	0
30	ho	13509	17375	0.38	12714	165	330343	172	52	154	302
30	ho_nopr	13509	17375	5.82	0	0	5732691	1600	142	293	11615
30	ho_noxs	13509	17375	0.42	12714	108	353589	432	24	253	0
30	ho_noprxs	13509	17375	5.93	0	0	6107892	12593	35	915	0
31	ho	13509	17386	0.47	12343	193	403020	248	39	228	495
31	ho_nopr	13509	17386	5.10	0	0	4904421	1512	164	329	11667
31	ho_noxs	13509	17386	0.47	12343	204	419406	618	19	342	0
31	ho_noprxs	13509	17386	4.87	0	0	4848184	12526	35	982	0
32	ho	13509	17390	0.48	12385	157	419990	326	36	287	353
32	ho_nopr	13509	17390	6.25	0	0	5986243	1821	147	362	11325
32	ho_noxs	13509	17390	0.52	12385	171	471095	615	20	336	0
32	ho_noprxs	13509	17390	6.35	0	0	6268169	12600	66	908	0
33	ho	13509	17494	0.38	12626	162	321681	193	34	197	329
33	ho_nopr	13509	17494	6.70	0	0	6574591	1631	106	296	11581
33	ho_noxs	13509	17494	0.40	12626	129	330898	465	17	288	0
33	ho_noprxs	13509	17494	4.77	0	0	4915289	12534	31	974	0

Table 40: TSP USA family, HO codes

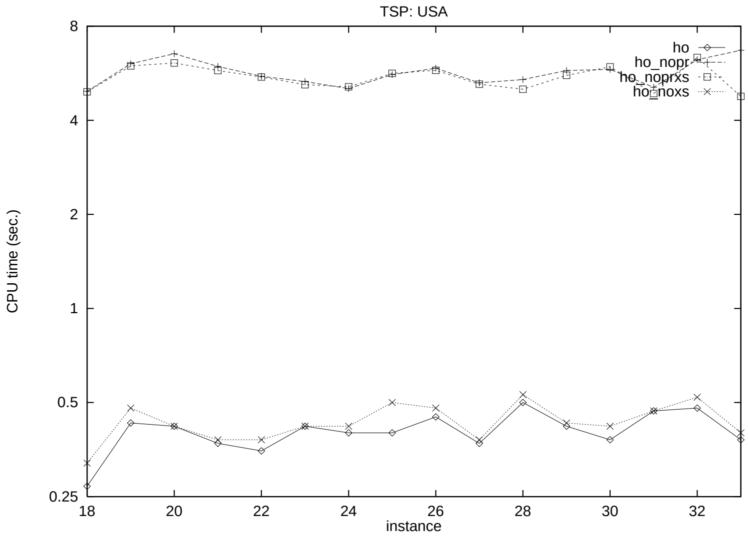

Figure 37: HO variants on TSP USA family

problem		nodes	arcs	total time	initial PR	internal PR	edge scans	st cuts	avg. size	1 node	excess cont
1	ho	20	38	0.00	0	6	1090	4	11	3	5
1	ho_nopr	20	38	0.00	0	0	765	6	10	8	5
1	ho_noxs	20	38	0.00	0	3	1097	10	6	5	0
1	ho_noprxs	20	38	0.00	0	0	761	11	6	8	0
2	ho	88	185	0.02	0	23	12296	26	23	19	18
2	ho_nopr	88	185	0.02	0	0	7681	34	19	27	26
2	ho_noxs	88	185	0.00	0	32	11846	35	20	19	0
2	ho_noprxs	88	185	0.00	0	0	9935	53	14	34	0
3	ho	148	300	0.00	0	49	16998	43	33	15	40
3	ho_nopr	148	300	0.02	0	0	21502	50	42	42	55
3	ho_noxs	148	300	0.02	0	57	24904	66	36	23	0
3	ho_noprxs	148	300	0.02	0	0	21982	96	24	51	0
4	ho	2	1	0.00	0	0	3	1	2	0	0
4	ho_nopr	2	1	0.00	0	0	2	1	2	0	0
4	ho_noxs	2	1	0.00	0	0	3	1	2	0	0
4	ho_noprxs	2	1	0.00	0	0	2	1	2	0	0
5	ho	150	292	0.02	0	27	14948	33	21	40	48
5	ho_nopr	150	292	0.00	0	0	11238	36	20	54	59
5	ho_noxs	150	292	0.00	0	17	16197	70	12	61	0
5	ho_noprxs	150	292	0.02	0	0	14212	79	14	70	0
6	ho	261	517	0.02	0	44	40893	55	44	45	115
6	ho_nopr	261	517	0.03	0	0	32584	57	43	72	131
6	ho_noxs	261	517	0.03	0	45	43979	125	25	89	0
6	ho_noprxs	261	517	0.02	0	0	35888	147	22	113	0
7	ho	113	221	0.00	0	26	14372	26	28	39	20
7	ho_nopr	113	221	0.00	0	0	11273	28	27	54	30
7	ho_noxs	113	221	0.02	0	28	13838	41	18	42	0
7	ho_noprxs	113	221	0.00	0	0	10094	53	14	59	0
8	ho	106	208	0.00	0	27	12416	24	29	30	23
8	ho_nopr	106	208	0.00	0	0	9532	28	26	48	29
8	ho_noxs	106	208	0.02	0	21	11562	44	16	39	0
8	ho_noprxs	106	208	0.00	0	0	10727	49	14	56	0
9	ho	19	36	0.00	0	5	848	6	8	5	2
9	ho_nopr	19	36	0.00	0	0	529	6	9	7	5
9	ho_noxs	19	36	0.00	0	7	821	6	8	4	0
9	ho_noprxs	19	36	0.00	0	0	525	11	6	7	0
10	ho	77	131	0.00	0	23	5748	20	19	11	22
10	ho_nopr	77	131	0.00	0	0	4340	26	17	18	32
10	ho_noxs	77	131	0.00	0	18	5981	30	14	27	0
10	ho_noprxs	77	131	0.00	0	0	4440	41	11	35	0
11	ho	605	1162	0.07	0	126	106232	117	34	135	225
11	ho_nopr	605	1162	0.07	0	0	85882	145	29	197	262
11	ho_noxs	605	1162	0.07	0	102	114132	294	18	207	0
11	ho_noprxs	605	1162	0.05	0	0	91399	337	16	267	0

Table 41: PRETSP misc family, HO codes

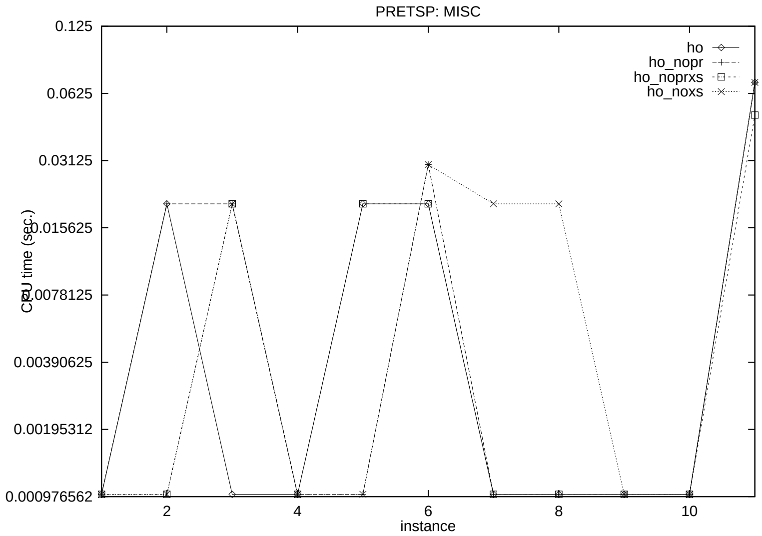

Figure 38: HO variants on PRETSP misc family

problem		nodes	arcs	total time	initial PR	internal PR	edge scans	st cuts	avg. size	1 node	excess cont
12	ho	2	1	0.00	0	0	3	1	2	0	0
12	ho_nopr	2	1	0.00	0	0	2	1	2	0	0
12	ho_noxs	2	1	0.00	0	0	3	1	2	0	0
12	ho_noprxs	2	1	0.00	0	0	2	1	2	0	0
13	ho	2	1	0.00	0	0	3	1	2	0	0
13	ho_nopr	2	1	0.00	0	0	2	1	2	0	0
13	ho_noxs	2	1	0.00	0	0	3	1	2	0	0
13	ho_noprxs	2	1	0.00	0	0	2	1	2	0	0
14	ho	2	1	0.00	0	0	3	1	2	0	0
14	ho_nopr	2	1	0.00	0	0	2	1	2	0	0
14	ho_noxs	2	1	0.00	0	0	3	1	2	0	0
14	ho_noprxs	2	1	0.00	0	0	2	1	2	0	0
15	ho	2	1	0.00	0	0	3	1	2	0	0
15	ho_nopr	2	1	0.00	0	0	2	1	2	0	0
15	ho_noxs	2	1	0.00	0	0	3	1	2	0	0
15	ho_noprxs	2	1	0.00	0	0	2	1	2	0	0
16	ho	2	1	0.00	0	0	3	1	2	0	0
16	ho_nopr	2	1	0.00	0	0	2	1	2	0	0
16	ho_noxs	2	1	0.00	0	0	3	1	2	0	0
16	ho_noprxs	2	1	0.00	0	0	2	1	2	0	0
17	ho	52	90	0.00	0	10	2048	8	20	10	22
17	ho_nopr	52	90	0.00	0	0	1199	12	14	13	26
17	ho_noxs	52	90	0.00	0	6	2265	30	7	14	0
17	ho_noprxs	52	90	0.00	0	0	1366	32	7	19	0

Table 42: PRETSP PLA family, HO codes

problem		nodes	arcs	total time	initial PR	internal PR	edge scans	st cuts	avg. size	1 node	excess cont
18	ho	325	561	0.03	0	80	34755	50	40	41	153
18	ho_nopr	325	561	0.02	0	0	26405	55	34	62	207
18	ho_noxs	325	561	0.03	0	96	46544	138	24	89	0
18	ho_noprxs	325	561	0.03	0	0	36956	198	21	126	0
19	ho	477	920	0.07	0	68	85447	111	35	122	175
19	ho_nopr	477	920	0.05	0	0	70925	120	33	142	214
19	ho_noxs	477	920	0.07	0	67	89417	241	19	167	0
19	ho_noprxs	477	920	0.05	0	0	90435	289	22	187	0
20	ho	449	861	0.05	0	118	83948	89	38	104	136
20	ho_nopr	449	861	0.05	0	0	64193	114	28	157	177
20	ho_noxs	449	861	0.07	0	87	80366	207	17	153	0
20	ho_noprxs	449	861	0.03	0	0	67161	254	15	194	0
21	ho	454	886	0.07	0	80	82086	118	33	108	147
21	ho_nopr	454	886	0.05	0	0	61760	122	33	152	179
21	ho_noxs	454	886	0.07	0	74	98450	233	25	145	0
21	ho_noprxs	454	886	0.05	0	0	64621	268	19	185	0
22	ho	403	786	0.03	0	72	58094	93	26	119	117
22	ho_nopr	403	786	0.03	0	0	45860	104	24	151	147
22	ho_noxs	403	786	0.05	0	91	64603	180	19	130	0
22	ho_noprxs	403	786	0.03	0	0	53718	236	16	166	0
23	ho	532	1029	0.07	0	128	108142	121	28	116	165
23	ho_nopr	532	1029	0.07	0	0	90033	143	25	185	203
23	ho_noxs	532	1029	0.07	0	125	108047	245	16	160	0
23	ho_noprxs	532	1029	0.07	0	0	90688	297	14	234	0
24	ho	501	950	0.07	0	97	78473	109	29	130	164
24	ho_nopr	501	950	0.03	0	0	62335	120	27	182	198
24	ho_noxs	501	950	0.07	0	70	86930	240	14	189	0
24	ho_noprxs	501	950	0.05	0	0	70035	283	13	217	0
25	ho	492	946	0.05	0	88	75279	100	36	129	174
25	ho_nopr	492	946	0.05	0	0	60305	112	33	175	204
25	ho_noxs	492	946	0.05	0	64	87652	264	15	162	0
25	ho_noprxs	492	946	0.03	0	0	70677	302	14	189	0
26	ho	549	1072	0.08	0	76	100108	147	31	138	187
26	ho_nopr	549	1072	0.07	0	0	80765	159	29	176	213
26	ho_noxs	549	1072	0.07	0	57	107232	270	23	220	0
26	ho_noprxs	549	1072	0.05	0	0	86945	292	21	256	0
27	ho	465	926	0.07	0	106	107500	106	52	86	165
27	ho_nopr	465	926	0.07	0	0	89154	129	47	140	195
27	ho_noxs	465	926	0.07	0	95	115089	232	31	136	0
27	ho_noprxs	465	926	0.07	0	0	99005	265	27	199	0
28	ho	542	1064	0.07	0	135	103811	120	42	117	169
28	ho_nopr	542	1064	0.08	0	0	97967	146	36	189	206
28	ho_noxs	542	1064	0.08	0	83	129100	296	26	161	0
28	ho_noprxs	542	1064	0.07	0	0	102606	317	25	224	0
29	ho	573	1104	0.08	0	121	104487	144	29	144	163
29	ho_nopr	573	1104	0.05	0	0	84573	157	27	205	210
29	ho_noxs	573	1104	0.07	0	86	107038	296	15	189	0
29	ho_noprxs	573	1104	0.07	0	0	85141	339	14	233	0
30	ho	476	945	0.05	0	89	87625	85	41	100	200
30	ho_nopr	476	945	0.05	0	0	73169	100	36	134	241
30	ho_noxs	476	945	0.07	0	106	110167	230	25	138	0
30	ho_noprxs	476	945	0.07	0	0	94897	267	21	208	0
31	ho	607	1150	0.07	0	86	86629	121	27	121	278
31	ho_nopr	607	1150	0.05	0	0	68290	135	25	157	314
31	ho_noxs	607	1150	0.08	0	83	118918	299	16	223	0
31	ho_noprxs	607	1150	0.07	0	0	96714	336	14	270	0
32	ho	557	1091	0.08	0	138	114243	129	42	129	159
32	ho_nopr	557	1091	0.05	0	0	95563	149	35	198	209
32	ho_noxs	557	1091	0.07	0	113	118286	265	20	177	0
32	ho_noprxs	557	1091	0.07	0	0	100091	314	19	242	0
33	ho	505	971	0.07	0	103	90697	119	32	131	151
33	ho_nopr	505	971	0.05	0	0	69790	126	29	180	198
33	ho_noxs	505	971	0.05	0	120	95463	222	18	162	0
33	ho_noprxs	505	971	0.05	0	0	78169	272	16	232	0

Table 43: PRETSP USA family, HO codes

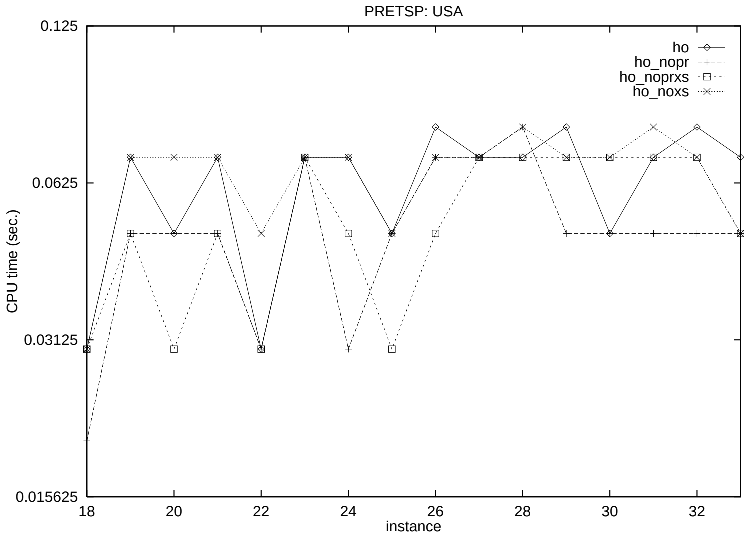

Figure 39: HO variants on PRETSP USA family

	nodes	arcs	st cuts		pr tests		excess conts.		1 node layers	
			avg	dev	avg	dev	avg	dev	avg	dev
pr	100	—	3	1.026320	96	0.513160	—	—	—	—
ho	100	191	1	0.000000	98	0.000000	0	0.000000	0	0.000000
ho_noxs	100	191	1	0.000000	98	0.000000	0	0.000000	0	0.000000
pr	200	—	1	0.888889	198	0.444445	—	—	—	—
ho	200	583	1	0.000000	198	0.000000	0	0.000000	0	0.000000
ho_noxs	200	583	1	0.000000	198	0.000000	0	0.000000	0	0.000000
pr	300	—	1	0.333333	298	0.166666	—	—	—	—
ho	300	1182	1	0.000000	298	0.000000	0	0.000000	0	0.000000
ho_noxs	300	1182	1	0.000000	298	0.000000	0	0.000000	0	0.000000
pr	400	—	1	0.000000	399	0.000000	—	—	—	—
ho	400	1968	1	0.310000	166	1.295000	229	0.820000	0	0.000000
ho_noxs	400	1968	12	0.900000	368	1.010000	0	0.000000	16	1.060000

Table 44: PR1 family

	nodes	arcs	st cuts		pr tests		excess conts.		1 node layers	
			avg	dev	avg	dev	avg	dev	avg	dev
pr	100	—	3	0.342105	96	0.171052	—	—	—	—
ho	100	584	1	0.330000	60	1.405000	36	1.230000	0	2.000000
ho_noxs	100	584	5	1.030000	89	1.030000	0	0.000000	3	2.000000
pr	200	—	12	0.238095	187	0.119048	—	—	—	—
ho	200	2171	2	0.320000	39	0.095000	156	0.050000	0	0.000000
ho_noxs	200	2171	24	0.120000	130	0.070000	0	0.000000	43	0.390000
pr	300	—	23	0.408696	277	0.204348	—	—	—	—
ho	300	4727	2	0.180000	75	0.105000	220	0.070000	0	0.000000
ho_noxs	300	4727	29	0.230000	185	0.065000	0	0.000000	84	0.280000
pr	400	—	42	0.275701	357	0.137850	—	—	—	—
ho	400	8307	3	0.240000	131	0.125000	263	0.120000	1	1.320000
ho_noxs	400	8307	39	0.220000	188	0.135000	0	0.000000	169	0.340000

Table 45: PR2 family

	nodes	arcs	st cuts		pr tests		excess conts.		1 node layers	
			avg	dev	avg	dev	avg	dev	avg	dev
pr	100	—	1	0.357143	98	0.178571	—	—	—	—
ho	100	2519	1	0.330000	54	0.040000	43	0.110000	0	0.000000
ho_noxs	100	2519	17	0.130000	45	0.120000	0	0.000000	35	0.320000
pr	200	—	1	0.312500	198	0.156250	—	—	—	—
ho	200	10053	4	0.330000	111	0.045000	81	0.110000	0	0.000000
ho_noxs	200	10053	22	0.180000	84	0.170000	0	0.000000	91	0.280000
pr	300	—	2	0.181818	297	0.090909	—	—	—	—
ho	300	22564	5	0.180000	163	0.110000	130	0.270000	0	0.000000
ho_noxs	300	22564	34	0.080000	122	0.170000	0	0.000000	141	0.310000
pr	400	—	2	0.333333	397	0.166666	—	—	—	—
ho	400	40047	5	0.400000	225	0.165000	166	0.440000	1	2.000000
ho_noxs	400	40047	41	0.220000	132	0.235000	0	0.000000	223	0.260000

Table 46: PR3 family

	nodes	arcs	st cuts		pr tests		excess conts.		1 node layers	
			avg	dev	avg	dev	avg	dev	avg	dev
pr	100	—	1	0.000000	99	0.000000	—	—	—	—
ho	100	4899	1	0.000000	75	0.020000	21	0.130000	0	0.000000
ho_noxs	100	4899	16	0.110000	47	0.085000	0	0.000000	34	0.220000
pr	200	—	1	0.000000	199	0.000000	—	—	—	—
ho	200	19693	1	0.000000	164	0.020000	32	0.220000	0	0.000000
ho_noxs	200	19693	31	0.130000	102	0.025000	0	0.000000	64	0.080000
pr	300	—	1	0.000000	299	0.000000	—	—	—	—
ho	300	44396	1	0.000000	254	0.010000	42	0.120000	0	0.000000
ho_noxs	300	44396	42	0.200000	120	0.155000	0	0.000000	135	0.280000
pr	400	—	1	0.000000	399	0.000000	—	—	—	—
ho	400	79002	1	0.000000	345	0.005000	51	0.070000	0	0.000000
ho_noxs	400	79002	49	0.160000	159	0.185000	0	0.000000	189	0.300000

Table 47: PR4 family

	nodes	arcs	st cuts		pr tests		excess conts.		1 node layers	
			avg	dev	avg	dev	avg	dev	avg	dev
pr	100	—	6	0.419355	93	0.209677	—	—	—	—
ho	100	191	1	0.000000	98	0.000000	0	0.000000	0	0.000000
ho_noxs	100	191	1	0.000000	98	0.000000	0	0.000000	0	0.000000
pr	200	—	1	0.500000	198	0.250000	—	—	—	—
ho	200	583	1	0.000000	198	0.000000	0	0.000000	0	0.000000
ho_noxs	200	583	1	0.000000	198	0.000000	0	0.000000	0	0.000000
pr	300	—	1	0.333333	298	0.166666	—	—	—	—
ho	300	1182	1	0.000000	298	0.000000	0	0.000000	0	0.000000
ho_noxs	300	1182	1	0.000000	298	0.000000	0	0.000000	0	0.000000
pr	400	—	1	0.000000	399	0.000000	—	—	—	—
ho	400	1968	1	0.000000	359	1.105000	38	2.000000	0	0.000000
ho_noxs	400	1968	1	0.890000	396	1.105000	0	0.000000	0	0.000000
pr	500	—	1	0.000000	499	0.000000	—	—	—	—
ho	500	2958	1	0.000000	449	0.110000	48	2.000000	0	0.000000
ho_noxs	500	2958	2	1.090000	496	1.110000	0	0.000000	0	0.000000
pr	1000	—	6	0.187500	993	0.093750	—	—	—	—
ho	1000	10923	2	0.710000	408	0.850000	586	0.630000	0	0.000000
ho_noxs	1000	10923	29	0.860000	849	0.760000	0	0.000000	119	1.450000
pr	1500	—	100	1.072420	1399	0.536210	—	—	—	—
ho	1500	23937	4	0.380000	32	1.015000	1460	0.020000	0	1.220000
ho_noxs	1500	23937	88	0.290000	1063	0.755000	0	0.000000	345	0.820000
pr	2000	—	215	0.759053	1784	0.379526	—	—	—	—
ho	2000	42000	9	0.160000	58	0.610000	1930	0.040000	0	0.820000
ho_noxs	2000	42000	86	0.080000	1534	0.030000	0	0.000000	377	0.240000

Table 48: PR5 family

	nodes	arcs	st cuts		pr tests		excess conts.		1 node layers	
			avg	dev	avg	dev	avg	dev	avg	dev
pr	100	—	2	0.571429	97	0.285714	—	—	—	—
ho	100	584	1	0.000000	79	0.145000	18	1.220000	0	0.000000
ho_noxs	100	584	1	0.350000	96	0.755000	0	0.000000	0	0.000000
pr	200	—	2	0.416667	197	0.208334	—	—	—	—
ho	200	2171	1	0.570000	83	0.835000	113	0.290000	0	0.000000
ho_noxs	200	2171	9	1.450000	173	0.260000	0	0.000000	14	2.000000
pr	300	—	12	1.023810	287	0.511905	—	—	—	—
ho	300	4727	4	0.650000	112	0.920000	181	0.570000	0	1.330000
ho_noxs	300	4727	30	0.670000	213	0.855000	0	0.000000	54	1.040000
pr	400	—	34	0.488235	366	0.244117	—	—	—	—
ho	400	8307	3	0.820000	59	1.270000	334	0.230000	0	0.940000
ho_noxs	400	8307	42	0.260000	269	1.135000	0	0.000000	85	0.710000

Table 49: PR6 family

	nodes	arcs	st cuts		pr tests		excess conts.		1 node layers	
			avg	dev	avg	dev	avg	dev	avg	dev
pr	100	—	1	0.000000	99	0.000000	—	—	—	—
ho	100	2519	2	1.090000	33	0.710000	61	0.260000	0	1.220000
ho_noxs	100	2519	20	0.330000	64	0.635000	0	0.000000	13	0.550000
pr	200	—	1	0.000000	199	0.000000	—	—	—	—
ho	200	10053	4	0.610000	53	0.100000	140	0.090000	0	0.000000
ho_noxs	200	10053	47	0.150000	115	0.045000	0	0.000000	35	0.320000
pr	300	—	1	0.571429	298	0.285714	—	—	—	—
ho	300	22564	7	0.630000	92	0.050000	199	0.030000	0	0.000000
ho_noxs	300	22564	71	0.100000	182	0.020000	0	0.000000	44	0.170000
pr	400	—	1	0.000000	399	0.000000	—	—	—	—
ho	400	40047	4	0.840000	81	0.265000	308	0.130000	4	1.880000
ho_noxs	400	40047	103	0.040000	225	0.045000	0	0.000000	69	0.300000

Table 50: PR7 family

	nodes	arcs	st cuts		pr tests		excess conts.		1 node layers	
			avg	dev	avg	dev	avg	dev	avg	dev
pr	100	—	1	0.000000	99	0.000000	—	—	—	—
ho	100	4899	3	0.640000	35	0.780000	59	0.060000	0	0.000000
ho_noxs	100	4899	28	0.310000	59	0.670000	0	0.000000	9	0.530000
pr	200	—	1	0.000000	199	0.000000	—	—	—	—
ho	200	19693	2	1.000000	90	1.110000	105	0.220000	0	0.000000
ho_noxs	200	19693	58	0.300000	115	1.120000	0	0.000000	23	0.240000
pr	300	—	1	0.000000	299	0.000000	—	—	—	—
ho	300	44396	2	1.230000	122	0.020000	173	0.010000	0	0.000000
ho_noxs	300	44396	100	0.090000	162	0.015000	0	0.000000	35	0.210000
pr	400	—	1	0.000000	399	0.000000	—	—	—	—
ho	400	79002	3	1.330000	170	0.010000	225	0.010000	0	0.000000
ho_noxs	400	79002	129	0.090000	208	0.045000	0	0.000000	60	0.350000

Table 51: PR8 family

12.3 Tables comparing ni codes

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	300	22425	0.31	0.07	—	—	0	0.00	3	0.51	—	—	4
ni	300	22425	0.18	0.02	0.04	0.26	0	0.00	240	0.04	202990	0.03	1
ni_nopr	300	22425	0.17	0.09	0.00	0.00	0	0.00	0	0.00	185886	0.08	4
hybrid	400	39900	0.60	0.02	—	—	0	0.00	3	0.76	—	—	4
ni	400	39900	0.39	0.03	0.10	0.08	0	0.00	326	0.04	362615	0.04	1
ni_nopr	400	39900	0.39	0.05	0.00	0.00	0	0.00	0	0.00	339659	0.08	4
hybrid	500	62375	1.02	0.05	—	—	0	0.00	4	0.36	—	—	5
ni	500	62375	0.66	0.04	0.15	0.05	0	0.00	407	0.05	553674	0.02	1
ni_nopr	500	62375	0.68	0.14	0.00	0.00	0	0.00	0	0.00	537027	0.12	5
hybrid	600	89850	1.61	0.05	—	—	0	0.00	5	0.28	—	—	5
ni	600	89850	0.97	0.01	0.23	0.02	0	0.00	507	0.03	802661	0.02	1
ni_nopr	600	89850	1.06	0.08	0.00	0.00	0	0.00	0	0.00	838423	0.09	5
hybrid	700	122325	2.54	0.06	—	—	0	0.00	5	0.25	—	—	6
ni	700	122325	1.41	0.02	0.31	0.03	0	0.00	620	0.02	1132793	0.03	1
ni_nopr	700	122325	1.72	0.09	0.00	0.00	0	0.00	0	0.00	1348331	0.10	6
hybrid	800	159800	3.26	0.07	—	—	0	0.00	7	0.31	—	—	5
ni	800	159800	1.83	0.04	0.42	0.01	0	0.00	686	0.04	1448069	0.03	1
ni_nopr	800	159800	2.09	0.11	0.00	0.00	0	0.00	0	0.00	1585064	0.13	5
hybrid	900	202275	4.54	0.05	—	—	0	0.00	7	0.09	—	—	6
ni	900	202275	2.36	0.02	0.53	0.01	0	0.00	802	0.02	1847438	0.01	1
ni_nopr	900	202275	2.94	0.07	0.00	0.00	0	0.00	0	0.00	2244850	0.09	6
hybrid	1000	249750	5.94	0.03	—	—	0	0.00	5	0.37	—	—	7
ni	1000	249750	3.02	0.01	0.66	0.01	0	0.00	902	0.01	2355983	0.01	1
ni_nopr	1000	249750	3.89	0.03	0.00	0.00	0	0.00	0	0.00	2943297	0.05	7

Table 52: NOI1 family, NI codes

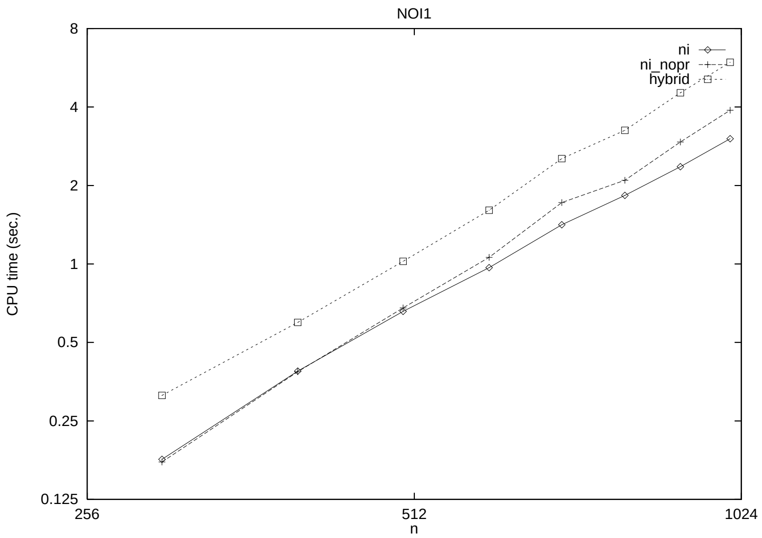

Figure 40: NI variants on NOI1 family

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	300	22425	0.27	0.03	—	—	0	0.00	13	1.20	—	—	2
ni	300	22425	0.17	0.00	0.05	0.00	0	0.00	166	0.08	184785	0.01	1
ni_nopr	300	22425	0.13	0.03	0.00	0.00	0	0.00	0	0.00	122409	0.03	3
hybrid	400	39900	0.53	0.02	—	—	0	0.00	22	0.85	—	—	3
ni	400	39900	0.38	0.05	0.10	0.16	0	0.00	230	0.06	329887	0.01	1
ni_nopr	400	39900	0.27	0.02	0.00	0.00	0	0.00	0	0.00	219950	0.02	3
hybrid	500	62375	0.92	0.03	—	—	0	0.00	35	0.84	—	—	3
ni	500	62375	0.63	0.04	0.15	0.05	0	0.00	302	0.05	521470	0.01	1
ni_nopr	500	62375	0.48	0.05	0.00	0.00	0	0.00	0	0.00	351768	0.03	3
hybrid	600	89850	1.37	0.02	—	—	0	0.00	28	0.64	—	—	3
ni	600	89850	0.92	0.02	0.22	0.02	0	0.00	357	0.03	745424	0.00	1
ni_nopr	600	89850	0.72	0.00	0.00	0.00	0	0.00	0	0.00	498428	0.01	3
hybrid	700	122325	2.01	0.02	—	—	0	0.00	71	0.21	—	—	3
ni	700	122325	1.31	0.02	0.31	0.04	0	0.00	436	0.02	1021707	0.01	1
ni_nopr	700	122325	1.02	0.03	0.00	0.00	0	0.00	0	0.00	693191	0.01	3
hybrid	800	159800	2.73	0.01	—	—	0	0.00	90	0.13	—	—	3
ni	800	159800	1.73	0.01	0.42	0.03	0	0.00	506	0.01	1339026	0.00	1
ni_nopr	800	159800	1.36	0.01	0.00	0.00	0	0.00	0	0.00	912200	0.01	3
hybrid	900	202275	3.59	0.02	—	—	0	0.00	115	0.11	—	—	3
ni	900	202275	2.24	0.02	0.54	0.04	0	0.00	582	0.03	1702206	0.01	1
ni_nopr	900	202275	1.77	0.01	0.00	0.00	0	0.00	0	0.00	1169033	0.02	3
hybrid	1000	249750	4.58	0.02	—	—	0	0.00	132	0.07	—	—	3
ni	1000	249750	2.77	0.01	0.66	0.03	0	0.00	652	0.02	2102361	0.00	1
ni_nopr	1000	249750	2.22	0.02	0.00	0.00	0	0.00	0	0.00	1451949	0.01	3

Table 53: NOI2 family, NI codes

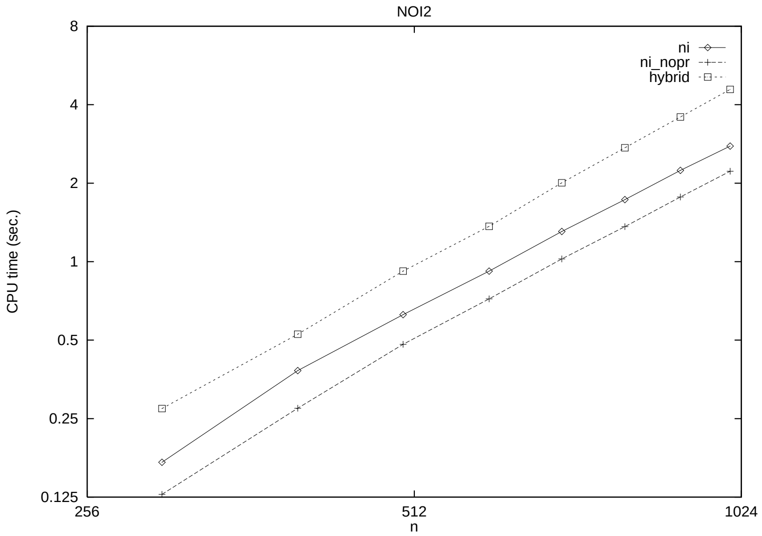

Figure 41: NI variants on NOI2 family

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	1000	24975	0.85	0.03	—	—	0	0.00	0	2.00	—	—	2
ni	1000	24975	0.24	0.04	0.07	0.00	0	0.00	490	0.08	249472	0.01	1
ni_nopr	1000	24975	0.19	0.05	0.00	0.00	0	0.00	0	0.00	166832	0.06	2
hybrid	1000	49950	1.36	0.02	—	—	0	0.00	1	0.31	—	—	3
ni	1000	49950	0.55	0.03	0.15	0.05	0	0.00	667	0.06	504398	0.01	1
ni_nopr	1000	49950	0.47	0.04	0.00	0.00	0	0.00	0	0.00	383303	0.06	3
hybrid	1000	124875	3.02	0.03	—	—	0	0.00	4	0.37	—	—	4
ni	1000	124875	1.50	0.02	0.35	0.00	0	0.00	821	0.02	1238518	0.02	1
ni_nopr	1000	124875	1.58	0.02	0.00	0.00	0	0.00	0	0.00	1187545	0.04	4
hybrid	1000	249750	5.96	0.03	—	—	0	0.00	5	0.37	—	—	7
ni	1000	249750	3.04	0.01	0.66	0.03	0	0.00	902	0.01	2355983	0.01	1
ni_nopr	1000	249750	3.87	0.04	0.00	0.00	0	0.00	0	0.00	2943297	0.05	7
hybrid	1000	374625	8.74	0.03	—	—	0	0.00	6	0.43	—	—	8
ni	1000	374625	4.36	0.01	0.94	0.02	0	0.00	931	0.01	3128006	0.00	1
ni_nopr	1000	374625	6.45	0.04	0.00	0.00	0	0.00	0	0.00	4840275	0.05	8
hybrid	1000	499500	11.46	0.06	—	—	0	0.00	10	0.19	—	—	10
ni	1000	499500	5.69	0.02	1.17	0.01	0	0.00	945	0.01	3917042	0.02	1
ni_nopr	1000	499500	9.04	0.08	0.00	0.00	0	0.00	0	0.00	6752355	0.10	10

Table 54: NOI3 family, NI codes

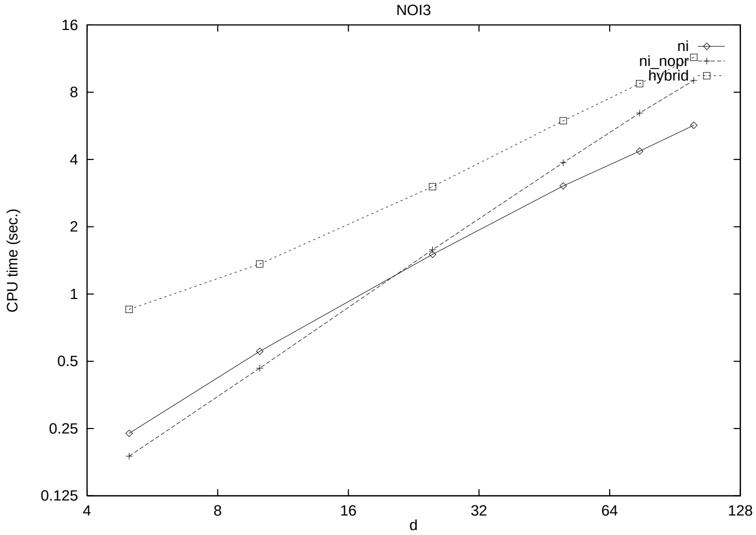

Figure 42: NI variants on NOI3 family

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	1000	24975	0.83	0.03	—	—	0	0.00	0	2.00	—	—	2
ni	1000	24975	0.23	0.03	0.07	0.06	0	0.00	220	0.23	237406	0.01	1
ni_nopr	1000	24975	0.17	0.00	0.00	0.00	0	0.00	0	0.00	138971	0.03	2
hybrid	1000	49950	1.28	0.01	—	—	0	0.00	0	0.50	—	—	2
ni	1000	49950	0.52	0.01	0.13	0.06	0	0.00	429	0.09	476370	0.01	1
ni_nopr	1000	49950	0.38	0.03	0.00	0.00	0	0.00	0	0.00	299764	0.02	3
hybrid	1000	124875	2.70	0.04	—	—	0	0.00	4	0.40	—	—	3
ni	1000	124875	1.42	0.02	0.35	0.00	0	0.00	589	0.06	1150995	0.01	1
ni_nopr	1000	124875	1.10	0.02	0.00	0.00	0	0.00	0	0.00	767900	0.03	3
hybrid	1000	249750	4.58	0.02	—	—	0	0.00	132	0.07	—	—	3
ni	1000	249750	2.80	0.02	0.68	0.03	0	0.00	652	0.02	2102361	0.00	1
ni_nopr	1000	249750	2.22	0.02	0.00	0.00	0	0.00	0	0.00	1451949	0.01	3
hybrid	1000	374625	6.17	0.01	—	—	0	0.00	0	1.46	—	—	3
ni	1000	374625	4.09	0.01	0.94	0.02	0	0.00	687	0.00	2889799	0.00	1
ni_nopr	1000	374625	3.30	0.01	0.00	0.00	0	0.00	0	0.00	2040534	0.00	3
hybrid	1000	499500	7.19	0.01	—	—	0	0.00	0	0.00	—	—	3
ni	1000	499500	5.26	0.01	1.17	0.01	0	0.00	691	0.01	3519347	0.00	1
ni_nopr	1000	499500	4.26	0.02	0.00	0.00	0	0.00	0	0.00	2499005	0.00	3

Table 55: NOI4 family, NI codes

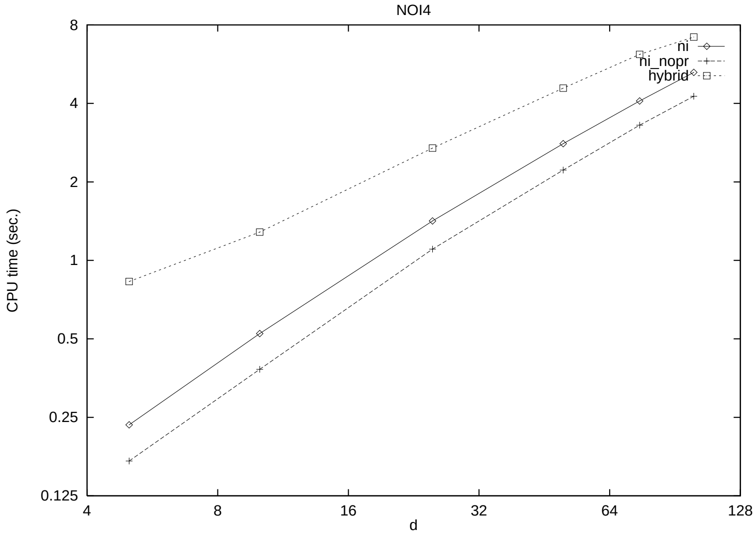

Figure 43: NI variants on NOI4 family

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	1000	249750	5.95	0.03	—	—	0	0.00	5	0.37	—	—	7
ni	1000	249750	3.02	0.01	0.66	0.01	0	0.00	902	0.01	2355983	0.01	1
ni_nopr	1000	249750	3.91	0.04	0.00	0.00	0	0.00	0	0.00	2943297	0.05	7
hybrid	1000	249750	4.55	0.02	—	—	0	0.00	132	0.07	—	—	3
ni	1000	249750	2.76	0.01	0.66	0.01	0	0.00	652	0.02	2102361	0.00	1
ni_nopr	1000	249750	2.24	0.02	0.00	0.00	0	0.00	0	0.00	1451949	0.01	3
hybrid	1000	249750	4.45	0.03	—	—	0	0.00	18	1.92	—	—	4
ni	1000	249750	3.10	0.02	0.67	0.02	0	0.00	430	0.12	2344959	0.02	2
ni_nopr	1000	249750	2.36	0.04	0.00	0.00	0	0.00	0	0.00	1513487	0.05	4
hybrid	1000	249750	4.41	0.02	—	—	0	0.00	0	0.00	—	—	5
ni	1000	249750	3.12	0.02	0.67	0.03	0	0.00	245	0.07	2356937	0.02	2
ni_nopr	1000	249750	2.24	0.02	0.00	0.00	0	0.00	0	0.00	1407624	0.02	5
hybrid	1000	249750	4.37	0.02	—	—	0	0.00	9	1.26	—	—	5
ni	1000	249750	3.02	0.04	0.65	0.01	0	0.00	325	0.50	2276407	0.06	2
ni_nopr	1000	249750	2.20	0.02	0.00	0.00	0	0.00	0	0.00	1343615	0.04	5
hybrid	1000	249750	4.18	0.01	—	—	0	0.00	6	1.54	—	—	3
ni	1000	249750	2.58	0.02	0.75	0.08	47	0.58	355	0.15	1875425	0.03	1
ni_nopr	1000	249750	2.01	0.02	0.00	0.00	0	0.00	0	0.00	1185005	0.03	3
hybrid	1000	249750	4.18	0.03	—	—	0	0.00	3	0.65	—	—	2
ni	1000	249750	2.36	0.18	0.80	0.04	235	1.62	119	0.59	1588310	0.27	0
ni_nopr	1000	249750	1.87	0.02	0.00	0.00	0	0.00	0	0.00	1072472	0.02	2
hybrid	1000	249750	4.20	0.01	—	—	0	0.00	1	0.85	—	—	3
ni	1000	249750	2.29	0.17	0.82	0.05	276	1.31	66	0.52	1505651	0.26	0
ni_nopr	1000	249750	1.86	0.01	0.00	0.00	0	0.00	0	0.00	1057830	0.01	3
hybrid	1000	249750	4.24	0.02	—	—	0	0.00	0	0.94	—	—	3
ni	1000	249750	1.81	0.23	0.93	0.06	810	0.46	27	2.00	1012838	0.38	0
ni_nopr	1000	249750	1.85	0.01	0.00	0.00	0	0.00	0	0.00	1055309	0.01	3
hybrid	1000	249750	4.21	0.01	—	—	0	0.00	0	1.22	—	—	3
ni	1000	249750	1.66	0.12	1.02	0.20	998	0.00	0	0.00	890687	0.25	0
ni_nopr	1000	249750	1.86	0.01	0.00	0.00	0	0.00	0	0.00	1050967	0.01	2
hybrid	1000	249750	4.22	0.01	—	—	0	0.00	0	0.00	—	—	2
ni	1000	249750	1.55	0.01	0.89	0.02	998	0.00	0	0.00	748224	0.01	0
ni_nopr	1000	249750	1.85	0.00	0.00	0.00	0	0.00	0	0.00	1045515	0.00	2

Table 56: NOI5 family, NI codes

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	1000	249750	4.24	0.01	—	—	0	0.00	0	0.00	—	—	2
ni	1000	249750	1.55	0.01	0.91	0.01	998	0.00	0	0.00	758571	0.00	0
ni_nopr	1000	249750	1.88	0.03	0.00	0.00	0	0.00	0	0.00	1045948	0.00	2
hybrid	1000	249750	4.47	0.01	—	—	0	0.00	6	0.48	—	—	2
ni	1000	249750	1.63	0.01	0.99	0.02	998	0.00	0	0.00	872232	0.00	0
ni_nopr	1000	249750	1.86	0.01	0.00	0.00	0	0.00	0	0.00	1056072	0.00	2
hybrid	1000	249750	4.73	0.02	—	—	0	0.00	74	0.82	—	—	2
ni	1000	249750	1.83	0.02	1.18	0.02	998	0.00	0	0.00	1131720	0.01	0
ni_nopr	1000	249750	1.97	0.02	0.00	0.00	0	0.00	0	0.00	1157751	0.02	2
hybrid	1000	249750	4.93	0.01	—	—	0	0.00	0	2.00	—	—	2
ni	1000	249750	1.80	0.00	1.15	0.00	998	0.00	0	0.00	1128304	0.01	0
ni_nopr	1000	249750	2.10	0.02	0.00	0.00	0	0.00	0	0.00	1306353	0.02	2
hybrid	1000	249750	5.00	0.01	—	—	0	0.00	5	0.74	—	—	2
ni	1000	249750	1.82	0.00	1.18	0.01	998	0.00	0	0.00	1183976	0.01	0
ni_nopr	1000	249750	2.26	0.02	0.00	0.00	0	0.00	0	0.00	1395803	0.01	2
hybrid	1000	249750	5.14	0.03	—	—	0	0.00	10	0.70	—	—	3
ni	1000	249750	1.84	0.01	1.20	0.00	998	0.00	0	0.00	1215231	0.00	0
ni_nopr	1000	249750	2.44	0.07	0.00	0.00	0	0.00	0	0.00	1567454	0.07	3

Table 57: NOI5 family, NI codes (continued)

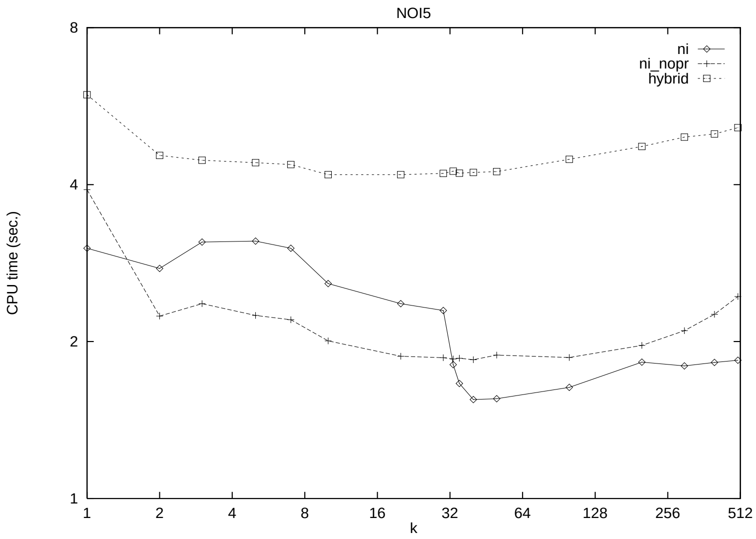

Figure 44: NI variants on NOI5 family

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	1000	249750	4.02	0.01	—	—	0	0.00	393	0.04	—	—	2
ni	1000	249750	2.57	0.01	0.66	0.01	0	0.00	423	0.03	1954265	0.00	1
ni_nopr	1000	249750	1.99	0.02	0.00	0.00	0	0.00	0	0.00	1262871	0.01	2
hybrid	1000	249750	4.48	0.01	—	—	0	0.00	11	1.19	—	—	2
ni	1000	249750	2.65	0.01	0.67	0.03	0	0.00	512	0.03	1999281	0.01	1
ni_nopr	1000	249750	2.07	0.02	0.00	0.00	0	0.00	0	0.00	1319241	0.01	3
hybrid	1000	249750	4.57	0.01	—	—	0	0.00	132	0.07	—	—	3
ni	1000	249750	2.77	0.02	0.68	0.02	0	0.00	652	0.02	2102361	0.00	1
ni_nopr	1000	249750	2.21	0.02	0.00	0.00	0	0.00	0	0.00	1451949	0.01	3
hybrid	1000	249750	5.19	0.05	—	—	0	0.00	59	1.13	—	—	5
ni	1000	249750	3.03	0.02	0.65	0.00	0	0.00	838	0.04	2326294	0.03	1
ni_nopr	1000	249750	3.12	0.09	0.00	0.00	0	0.00	0	0.00	2213552	0.11	6
hybrid	1000	249750	5.23	0.06	—	—	0	0.00	76	0.82	—	—	5
ni	1000	249750	3.02	0.02	0.66	0.01	0	0.00	848	0.04	2312736	0.02	1
ni_nopr	1000	249750	3.17	0.07	0.00	0.00	0	0.00	0	0.00	2284205	0.10	5
hybrid	1000	249750	4.98	0.03	—	—	0	0.00	3	0.67	—	—	4
ni	1000	249750	2.94	0.01	0.66	0.01	0	0.00	808	0.03	2246478	0.01	1
ni_nopr	1000	249750	2.93	0.04	0.00	0.00	0	0.00	0	0.00	2010144	0.06	4
hybrid	1000	249750	5.12	0.03	—	—	0	0.00	3	0.47	—	—	5
ni	1000	249750	2.94	0.01	0.66	0.03	0	0.00	832	0.02	2226590	0.01	1
ni_nopr	1000	249750	3.06	0.06	0.00	0.00	0	0.00	0	0.00	2146069	0.06	5
hybrid	1000	249750	5.91	0.07	—	—	0	0.00	5	0.43	—	—	6
ni	1000	249750	2.99	0.02	0.66	0.01	0	0.00	909	0.02	2325945	0.02	1
ni_nopr	1000	249750	3.86	0.11	0.00	0.00	0	0.00	0	0.00	2897920	0.14	6
hybrid	1000	249750	5.85	0.05	—	—	0	0.00	7	0.42	—	—	6
ni	1000	249750	2.96	0.02	0.67	0.03	0	0.00	892	0.02	2279187	0.02	1
ni_nopr	1000	249750	3.75	0.07	0.00	0.00	0	0.00	0	0.00	2830016	0.09	6

Table 58: NOI6 family, NI codes

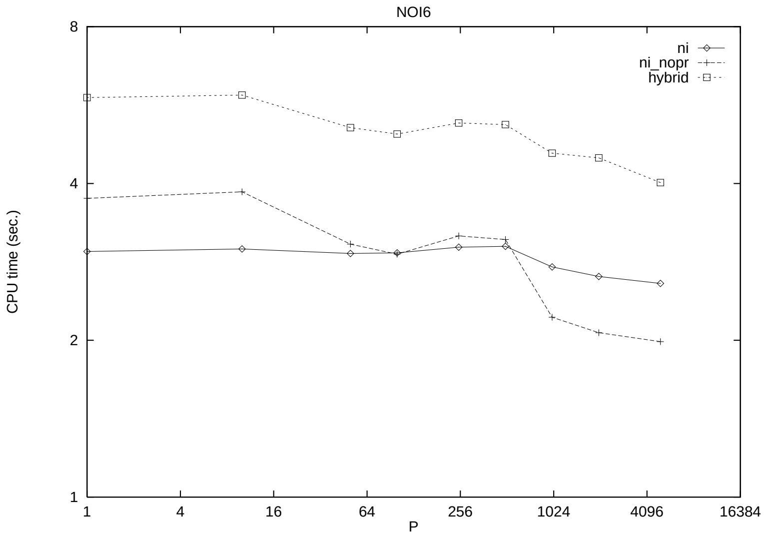

Figure 45: NI variants on NOI6 family

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	1000	8000	0.62	0.04	—	—	0	0.00	8	0.10	—	—	16
ni	1000	8000	0.13	0.00	0.01	0.82	0	0.00	867	0.01	223903	0.01	4
ni_nopr	1000	8000	0.25	0.03	0.00	0.00	0	0.00	0	0.00	374431	0.01	16
hybrid	1000	16000	1.82	0.35	—	—	0	0.00	15	0.10	—	—	35
ni	1000	16000	0.36	0.44	0.03	0.00	0	0.00	921	0.04	514597	0.47	4
ni_nopr	1000	16000	1.13	0.46	0.00	0.00	0	0.00	0	0.00	1492401	0.46	35
hybrid	1000	32000	6.51	0.48	—	—	0	0.00	29	0.13	—	—	79
ni	1000	32000	1.22	0.49	0.09	0.11	0	0.00	962	0.01	1357742	0.52	6
ni_nopr	1000	32000	5.97	0.54	0.00	0.00	0	0.00	0	0.00	6549302	0.55	78
hybrid	1000	62500	10.26	0.37	—	—	0	0.00	42	0.17	—	—	76
ni	1000	62500	1.63	0.17	0.18	0.02	0	0.00	973	0.01	1605205	0.19	3
ni_nopr	1000	62500	9.41	0.38	0.00	0.00	0	0.00	0	0.00	9591881	0.39	76
hybrid	1000	125000	40.45	0.02	—	—	0	0.00	73	0.07	—	—	154
ni	1000	125000	3.58	0.09	0.35	0.04	0	0.00	987	0.00	3317215	0.09	3
ni_nopr	1000	125000	36.73	0.05	0.00	0.00	0	0.00	0	0.00	35768082	0.05	152
hybrid	1000	250000	72.80	0.01	—	—	0	0.00	87	0.04	—	—	173
ni	1000	250000	5.35	0.19	0.69	0.02	0	0.00	990	0.00	4412996	0.22	2
ni_nopr	1000	250000	68.27	0.02	0.00	0.00	0	0.00	0	0.00	63337291	0.02	171
hybrid	1000	499500	149.77	0.26	—	—	0	0.00	95	0.13	—	—	222
ni	1000	499500	8.61	0.24	1.23	0.02	0	0.00	992	0.00	6151137	0.27	2
ni_nopr	1000	499500	139.40	0.25	0.00	0.00	0	0.00	0	0.00	124293627	0.25	220

Table 59: RAND1 family, NI codes

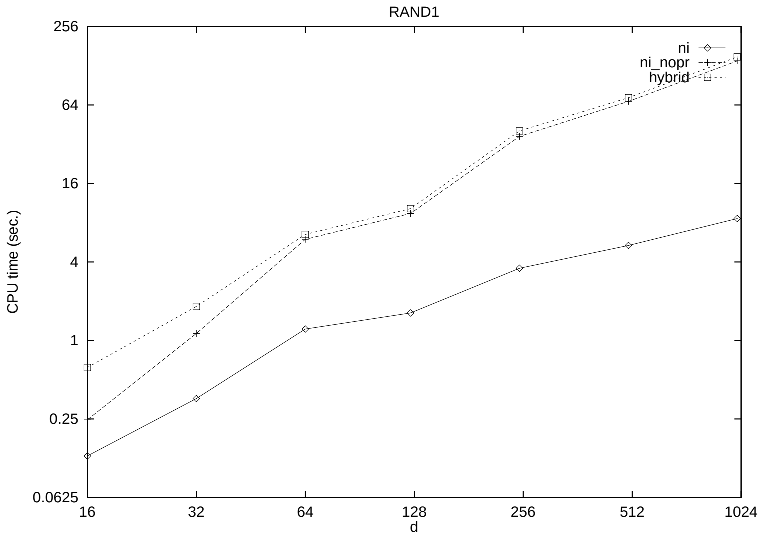

Figure 46: NI variants on RAND1 family

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	500	12500	1.79	0.32	—	—	0	0.00	16	0.08	—	—	59
ni	500	12500	0.33	0.42	0.02	0.00	0	0.00	466	0.02	711803	0.43	8
ni_nopr	500	12500	1.14	0.37	0.00	0.00	0	0.00	0	0.00	2059881	0.36	59
hybrid	708	17700	2.59	0.46	—	—	0	0.00	17	0.06	—	—	55
ni	708	17700	0.48	0.52	0.03	0.00	0	0.00	674	0.01	664953	0.53	5
ni_nopr	708	17700	1.88	0.52	0.00	0.00	0	0.00	0	0.00	2561200	0.54	54
hybrid	1000	25000	4.95	0.23	—	—	0	0.00	25	0.08	—	—	73
ni	1000	25000	1.04	0.29	0.07	0.06	0	0.00	954	0.01	1218246	0.31	7
ni_nopr	1000	25000	4.38	0.28	0.00	0.00	0	0.00	0	0.00	4974015	0.28	73
hybrid	1414	35350	5.90	0.43	—	—	0	0.00	27	0.14	—	—	56
ni	1414	35350	1.13	0.38	0.11	0.09	0	0.00	1353	0.01	1164292	0.41	4
ni_nopr	1414	35350	5.07	0.56	0.00	0.00	0	0.00	0	0.00	5077948	0.57	57
hybrid	2000	50000	11.44	0.40	—	—	0	0.00	33	0.19	—	—	69
ni	2000	50000	2.16	0.42	0.16	0.06	0	0.00	1900	0.01	2140653	0.45	6
ni_nopr	2000	50000	9.06	0.47	0.00	0.00	0	0.00	0	0.00	8815422	0.48	68
hybrid	2828	70700	23.94	0.02	—	—	0	0.00	44	0.06	—	—	95
ni	2828	70700	3.75	0.11	0.23	0.00	0	0.00	2707	0.01	3529980	0.12	7
ni_nopr	2828	70700	18.90	0.01	0.00	0.00	0	0.00	0	0.00	17318872	0.01	95
hybrid	4000	100000	33.40	0.30	—	—	0	0.00	49	0.27	—	—	85
ni	4000	100000	5.66	0.44	0.34	0.03	0	0.00	3798	0.02	5063980	0.47	7
ni_nopr	4000	100000	24.37	0.37	0.00	0.00	0	0.00	0	0.00	21117716	0.37	84
hybrid	5656	141400	52.90	0.22	—	—	0	0.00	58	0.18	—	—	89
ni	5656	141400	8.04	0.17	0.51	0.02	0	0.00	5419	0.01	6404503	0.18	6
ni_nopr	5656	141400	41.33	0.28	0.00	0.00	0	0.00	0	0.00	31409265	0.29	90
hybrid	8000	200000	95.02	0.01	—	—	0	0.00	70	0.03	—	—	104
ni	8000	200000	13.29	0.04	0.77	0.01	0	0.00	7699	0.00	9785313	0.04	7
ni_nopr	8000	200000	71.45	0.01	0.00	0.00	0	0.00	0	0.00	50274600	0.01	104

Table 60: RAND2 family, NI codes

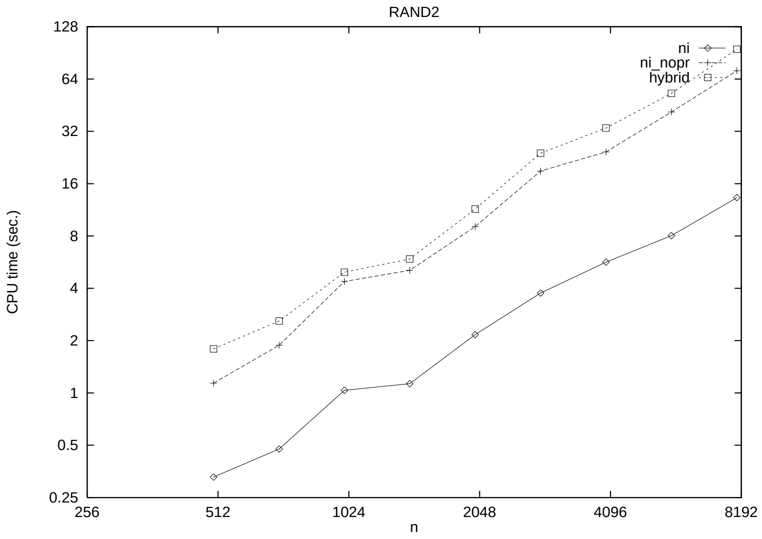

Figure 47: NI variants on RAND2 family

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	1001	1001	1.54	0.02	—	—	0	0.00	300	0.00	—	—	699
ni	1001	1001	0.01	1.22	0.00	0.00	999	0.00	0	0.00	5001	0.00	0
ni_nopr	1001	1001	1.62	0.02	0.00	0.00	0	0.00	0	0.00	2007993	0.00	999
hybrid	1001	3003	3.28	0.02	—	—	0	0.00	3	0.54	—	—	266
ni	1001	3003	3.04	0.02	0.00	2.00	0	0.00	411	0.16	4456979	0.02	250
ni_nopr	1001	3003	2.37	0.03	0.00	0.00	0	0.00	0	0.00	2863192	0.03	264
hybrid	1001	10010	12.91	0.02	—	—	0	0.00	77	0.06	—	—	425
ni	1001	10010	10.02	0.02	0.02	0.50	0	0.00	594	0.12	19899570	0.02	332
ni_nopr	1001	10010	7.96	0.01	0.00	0.00	0	0.00	0	0.00	13278259	0.01	425
hybrid	1001	33033	37.21	0.02	—	—	0	0.00	156	0.02	—	—	477
ni	1001	33033	51.66	0.01	0.08	0.10	0	0.00	681	0.01	61711219	0.01	311
ni_nopr	1001	33033	36.90	0.00	0.00	0.00	0	0.00	0	0.00	40531738	0.00	478
hybrid	1001	100100	111.02	0.01	—	—	0	0.00	195	0.01	—	—	494
ni	1001	100100	151.94	0.00	0.28	0.00	0	0.00	710	0.00	166466428	0.00	286
ni_nopr	1001	100100	103.67	0.00	0.00	0.00	0	0.00	0	0.00	104767321	0.00	494
hybrid	1001	333333	289.28	0.01	—	—	0	0.00	196	0.01	—	—	499
ni	1001	333333	12.65	0.07	0.85	0.01	0	0.00	993	0.00	11099620	0.07	5
ni_nopr	1001	333333	267.34	0.00	0.00	0.00	0	0.00	0	0.00	249318885	0.00	499
hybrid	1001	1001000	442.91	0.01	—	—	0	0.00	404	0.06	—	—	297
ni	1001	1001000	18.14	0.02	1.79	0.01	0	0.00	996	0.00	12777462	0.02	3
ni_nopr	1001	1001000	456.84	0.00	0.00	0.00	0	0.00	0	0.00	397887207	0.00	500

Table 61: REG1 family, NI codes

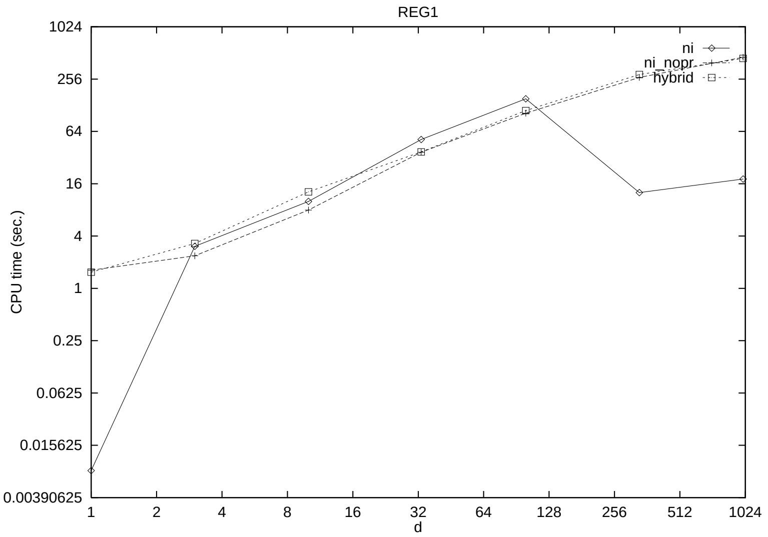

Figure 48: NI variants on REG1 family

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	50	2500	0.04	0.19	—	—	0	0.00	10	0.07	—	—	20
ni	50	2500	0.02	0.00	0.00	2.00	0	0.00	46	0.00	20664	0.01	2
ni_nopr	50	2500	0.03	0.14	0.00	0.00	0	0.00	0	0.00	60584	0.01	25
hybrid	100	5000	0.25	0.03	—	—	0	0.00	14	0.07	—	—	49
ni	100	5000	0.05	0.17	0.00	2.00	0	0.00	95	0.00	89037	0.03	3
ni_nopr	100	5000	0.18	0.03	0.00	0.00	0	0.00	0	0.00	355237	0.01	49
hybrid	200	10000	1.34	0.05	—	—	0	0.00	32	0.03	—	—	98
ni	200	10000	0.17	0.15	0.00	2.00	0	0.00	191	0.01	376816	0.17	6
ni_nopr	200	10000	0.90	0.01	0.00	0.00	0	0.00	0	0.00	1847583	0.01	98
hybrid	400	20000	6.64	0.03	—	—	0	0.00	66	0.01	—	—	196
ni	400	20000	9.06	0.02	0.04	0.26	0	0.00	277	0.01	13649236	0.02	119
ni_nopr	400	20000	6.24	0.00	0.00	0.00	0	0.00	0	0.00	8485443	0.00	195
hybrid	800	40000	33.25	0.03	—	—	0	0.00	139	0.02	—	—	390
ni	800	40000	48.19	0.01	0.10	0.00	0	0.00	557	0.00	56706079	0.01	239
ni_nopr	800	40000	34.07	0.00	0.00	0.00	0	0.00	0	0.00	36615665	0.00	389

Table 62: REG2 family, NI codes

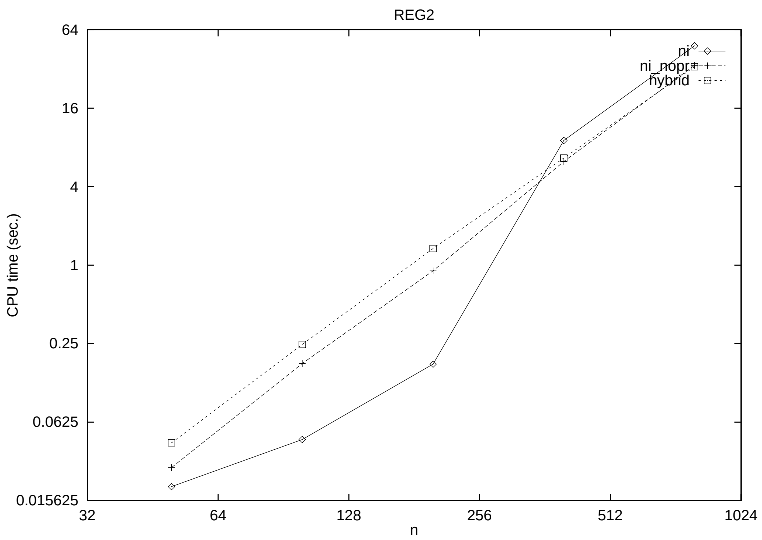

Figure 49: NI variants on REG2 family

	nodes	arcs	total time avg	total time dev %	discovery time avg	discovery time dev %	initial PR avg	initial PR dev %	internal PR avg	internal PR dev %	edge scans avg	edge scans dev %	phases avg
hybrid	256	512	0.13	0.11	—	—	0	0.00	1	0.73	—	—	63
ni	256	512	0.10	0.13	0.00	0.00	4	0.44	125	0.22	142274	0.07	47
ni_nopr	256	512	0.09	0.20	0.00	0.00	0	0.00	0	0.00	103300	0.04	60
hybrid	512	1024	0.45	0.04	—	—	0	0.00	2	0.42	—	—	105
ni	512	1024	0.39	0.07	0.00	0.00	4	0.27	231	0.12	529756	0.08	89
ni_nopr	512	1024	0.31	0.12	0.00	0.00	0	0.00	0	0.00	350072	0.05	100
hybrid	1024	2048	1.80	0.03	—	—	0	0.00	4	0.42	—	—	184
ni	1024	2048	1.46	0.05	0.00	0.00	6	0.32	358	0.22	1954614	0.05	163
ni_nopr	1024	2048	1.14	0.05	0.00	0.00	0	0.00	0	0.00	1311582	0.05	181
hybrid	2048	4096	8.62	0.04	—	—	0	0.00	2	0.50	—	—	331
ni	2048	4096	5.95	0.03	0.01	1.22	6	0.30	626	0.35	7326869	0.03	304
ni_nopr	2048	4096	4.60	0.01	0.00	0.00	0	0.00	0	0.00	4873826	0.02	329
hybrid	4096	8192	39.82	0.02	—	—	0	0.00	2	0.42	—	—	600
ni	4096	8192	26.06	0.02	0.03	0.00	5	0.36	972	0.32	28309175	0.03	585
ni_nopr	4096	8192	19.00	0.03	0.00	0.00	0	0.00	0	0.00	17954220	0.03	595
hybrid	8192	16384	177.96	0.01	—	—	0	0.00	1	0.31	—	—	1099
ni	8192	16384	167.89	0.03	0.08	0.00	5	0.22	2275	0.25	105627355	0.02	1086
ni_nopr	8192	16384	122.83	0.03	0.00	0.00	0	0.00	0	0.00	67001317	0.00	1111
hybrid	16384	32768	843.45	0.00	—	—	0	0.00	1	1.10	—	—	2027
ni	16384	32768	780.60	0.01	0.21	0.05	5	0.61	4665	0.28	392336486	0.01	2012
ni_nopr	16384	32768	557.04	0.01	0.00	0.00	0	0.00	0	0.00	247195863	0.01	2028

Table 63: REG3 family, NI codes

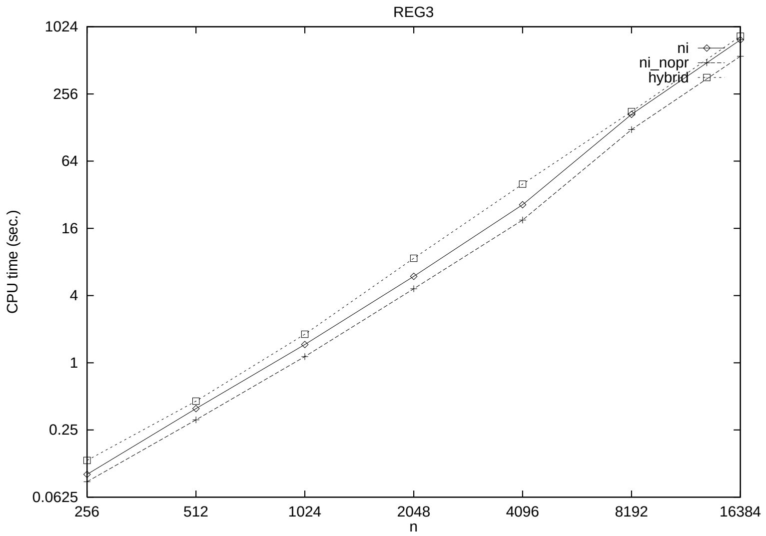

Figure 50: NI variants on REG3 family

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	128	1024	0.14	0.06	—	—	0	0.00	4	0.18	—	—	56
ni	128	1024	0.12	0.08	0.00	2.00	0	0.00	79	0.02	268717	0.04	44
ni_nopr	128	1024	0.10	0.13	0.00	0.00	0	0.00	0	0.00	175125	0.03	56
hybrid	256	4096	0.93	0.01	—	—	0	0.00	26	0.04	—	—	119
ni	256	4096	0.87	0.01	0.01	1.22	0	0.00	168	0.01	2049661	0.01	84
ni_nopr	256	4096	0.68	0.01	0.00	0.00	0	0.00	0	0.00	1328580	0.01	120
hybrid	512	16384	7.73	0.02	—	—	0	0.00	76	0.02	—	—	245
ni	512	16384	8.85	0.00	0.03	0.14	0	0.00	352	0.00	15173694	0.00	156
ni_nopr	512	16384	6.56	0.01	0.00	0.00	0	0.00	0	0.00	9899512	0.01	246
hybrid	1024	65536	74.20	0.02	—	—	0	0.00	187	0.01	—	—	501
ni	1024	65536	103.96	0.01	0.19	0.05	0	0.00	719	0.00	116991994	0.01	301
ni_nopr	1024	65536	73.54	0.00	0.00	0.00	0	0.00	0	0.00	75807247	0.00	501
hybrid	2048	262144	725.92	0.01	—	—	0	0.00	418	0.01	—	—	1011
ni	2048	262144	857.65	0.01	0.81	0.01	0	0.00	1466	0.00	896240114	0.01	577
ni_nopr	2048	262144	600.55	0.00	0.00	0.00	0	0.00	0	0.00	582230705	0.00	1010

Table 64: REG4 family, NI codes

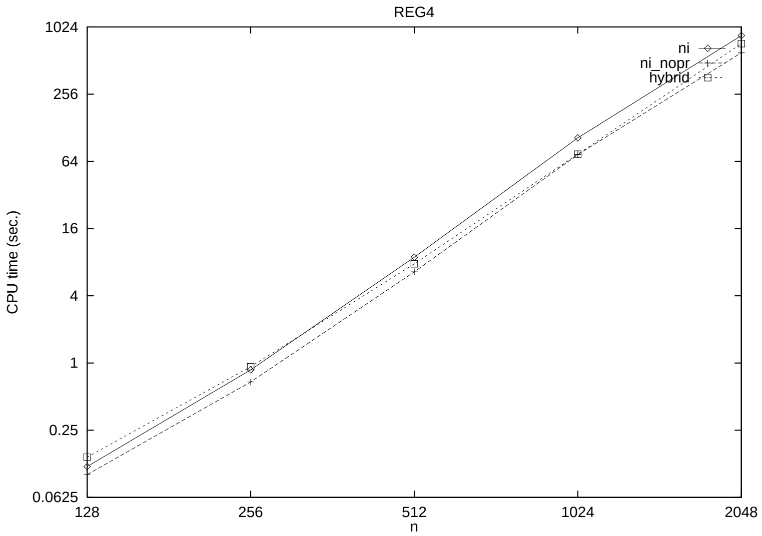

Figure 51: NI variants on REG4 family

	nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
			avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
hybrid	1024	2044	5.18	0.00	—	—	0	0.00	17	0.00	—	—	1002
ni	1024	2044	1.73	0.01	0.00	0.00	0	0.00	764	0.00	2879946	0.00	257
ni_nopr	1024	2044	2.34	0.03	0.00	0.00	0	0.00	0	0.00	3168281	0.00	766
hybrid	2048	4092	23.08	0.00	—	—	0	0.00	107	0.00	—	—	1936
ni	2048	4092	7.11	0.00	0.01	0.82	0	0.00	1532	0.00	11480028	0.00	513
ni_nopr	2048	4092	9.49	0.01	0.00	0.00	0	0.00	0	0.00	12692757	0.00	1536
hybrid	4096	8188	105.98	0.01	—	—	0	0.00	322	0.00	—	—	3770
ni	4096	8188	28.95	0.00	0.02	0.20	0	0.00	3067	0.00	45877881	0.00	1026
ni_nopr	4096	8188	38.27	0.01	0.00	0.00	0	0.00	0	0.00	50714457	0.00	3071
hybrid	8192	16380	510.13	0.00	—	—	0	0.00	682	0.00	—	—	7506
ni	8192	16380	201.12	0.00	0.07	0.00	0	0.00	6140	0.00	183242209	0.00	2049
ni_nopr	8192	16380	255.10	0.00	0.00	0.00	0	0.00	0	0.00	202861701	0.00	6142

Table 65: BIKEWHE family, NI codes

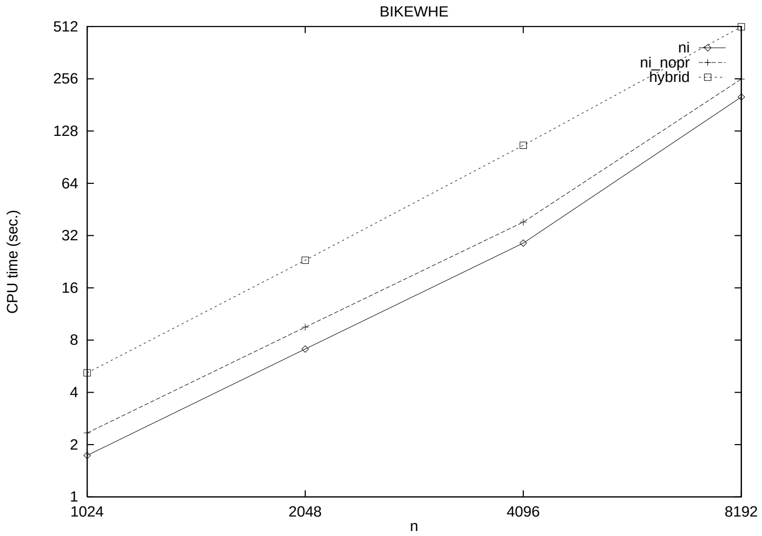

Figure 52: NI variants on BIKEWHE family

problem		nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
				avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
1	hybrid	532	787	0.60	0.00	—	—	0	0.00	136	0.00	—	—	364
1	ni	532	787	0.00	0.00	0.00	0.00	487	0.00	35	0.00	10042	0.00	7
1	ni_nopr	532	787	0.62	0.00	0.00	0.00	0	0.00	0	0.00	699475	0.00	491
2	hybrid	1291	1942	3.57	0.00	—	—	0	0.00	326	0.00	—	—	823
2	ni	1291	1942	0.05	0.00	0.00	0.00	976	0.00	257	0.00	61047	0.00	29
2	ni_nopr	1291	1942	4.33	0.00	0.00	0.00	0	0.00	0	0.00	4659892	0.00	1213
3	hybrid	1400	2231	2.37	0.00	—	—	0	0.00	240	0.00	—	—	658
3	ni	1400	2231	0.05	0.00	0.02	0.00	1091	0.00	238	0.00	64451	0.00	26
3	ni_nopr	1400	2231	2.95	0.00	0.00	0.00	0	0.00	0	0.00	3420515	0.00	980
4	hybrid	76	90	0.02	0.00	—	—	0	0.00	27	0.00	—	—	45
4	ni	76	90	0.00	0.00	0.00	0.00	74	0.00	0	0.00	691	0.00	0
4	ni_nopr	76	90	0.02	0.00	0.00	0.00	0	0.00	0	0.00	12821	0.00	70
5	hybrid	5934	7287	20.93	0.00	—	—	0	0.00	787	0.00	—	—	1217
5	ni	5934	7287	0.07	0.00	0.03	0.00	5651	0.00	169	0.00	88043	0.00	16
5	ni_nopr	5934	7287	24.70	0.00	0.00	0.00	0	0.00	0	0.00	21081457	0.00	2274
6	hybrid	5934	7627	84.68	0.00	—	—	0	0.00	1090	0.00	—	—	3961
6	ni	5934	7627	0.12	0.00	0.05	0.00	5444	0.00	360	0.00	139309	0.00	37
6	ni_nopr	5934	7627	69.07	0.00	0.00	0.00	0	0.00	0	0.00	60054461	0.00	4939
7	hybrid	1323	2169	4.37	0.00	—	—	0	0.00	338	0.00	—	—	850
7	ni	1323	2169	0.05	0.00	0.02	0.00	1115	0.00	162	0.00	70018	0.00	40
7	ni_nopr	1323	2169	5.33	0.00	0.00	0.00	0	0.00	0	0.00	5690479	0.00	1279
8	hybrid	1323	2195	4.70	0.00	—	—	0	0.00	368	0.00	—	—	883
8	ni	1323	2195	0.03	0.00	0.00	0.00	1167	0.00	113	0.00	62461	0.00	39
8	ni_nopr	1323	2195	5.67	0.00	0.00	0.00	0	0.00	0	0.00	5890041	0.00	1293
9	hybrid	1084	1252	1.72	0.00	—	—	0	0.00	336	0.00	—	—	655
9	ni	1084	1252	0.02	0.00	0.00	0.00	1052	0.00	23	0.00	11403	0.00	5
9	ni_nopr	1084	1252	2.23	0.00	0.00	0.00	0	0.00	0	0.00	2344365	0.00	1003
10	hybrid	1748	2336	5.87	0.00	—	—	0	0.00	535	0.00	—	—	1105
10	ni	1748	2336	0.02	0.00	0.02	0.00	1611	0.00	109	0.00	37891	0.00	19
10	ni_nopr	1748	2336	6.57	0.00	0.00	0.00	0	0.00	0	0.00	7127789	0.00	1576
11	hybrid	15112	19057	392.22	0.00	—	—	0	0.00	1948	0.00	—	—	5349
11	ni	15112	19057	0.52	0.00	0.17	0.00	13912	0.00	793	0.00	423025	0.00	67
11	ni_nopr	15112	19057	452.17	0.00	0.00	0.00	0	0.00	0	0.00	231649252	0.00	7761

Table 66: TSP misc family, NI codes

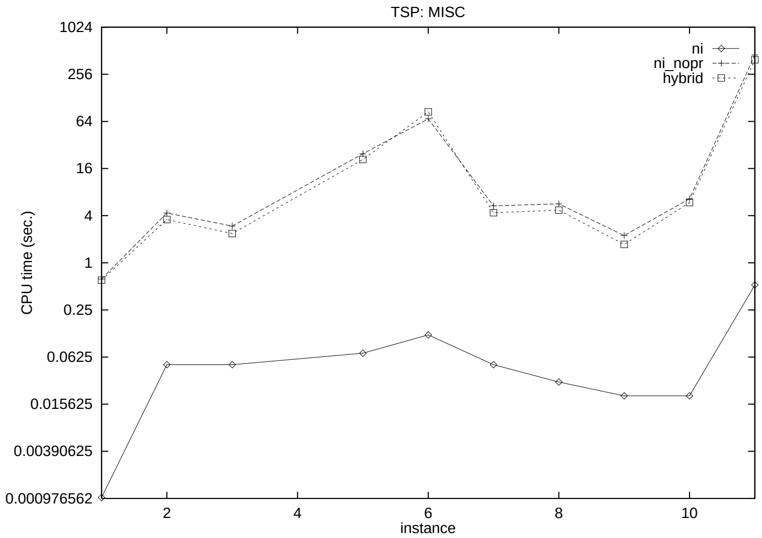

Figure 53: NI variants on TSP misc family

problem		nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
				avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
12	hybrid	33810	38600	3.27	0.00	—	—	0	0.00	0	0.00	—	—	7
12	ni	33810	38600	0.43	0.00	0.28	0.00	33808	0.00	0	0.00	271168	0.00	0
12	ni_nopr	33810	38600	0.95	0.00	0.00	0.00	0	0.00	0	0.00	394977	0.00	6
13	hybrid	33810	39367	11.37	0.00	—	—	0	0.00	320	0.00	—	—	20
13	ni	33810	39367	0.47	0.00	0.32	0.00	33808	0.00	0	0.00	303274	0.00	0
13	ni_nopr	33810	39367	5.10	0.00	0.00	0.00	0	0.00	0	0.00	2164353	0.00	18
14	hybrid	33810	39456	3.98	0.00	—	—	0	0.00	0	0.00	—	—	11
14	ni	33810	39456	0.45	0.00	0.30	0.00	33808	0.00	0	0.00	295707	0.00	0
14	ni_nopr	33810	39456	2.07	0.00	0.00	0.00	0	0.00	0	0.00	872200	0.00	12
15	hybrid	85900	102596	16.62	0.00	—	—	0	0.00	5	0.00	—	—	10
15	ni	85900	102596	1.37	0.00	1.02	0.00	85898	0.00	0	0.00	810396	0.00	0
15	ni_nopr	85900	102596	6.75	0.00	0.00	0.00	0	0.00	0	0.00	2704618	0.00	10
16	hybrid	85900	102934	21.18	0.00	—	—	0	0.00	22	0.00	—	—	13
16	ni	85900	102934	1.43	0.00	1.08	0.00	85898	0.00	0	0.00	855358	0.00	0
16	ni_nopr	85900	102934	8.62	0.00	0.00	0.00	0	0.00	0	0.00	3499783	0.00	14
17	hybrid	85900	102988	21.30	0.00	—	—	0	0.00	11	0.00	—	—	15
17	ni	85900	102988	1.43	0.00	1.08	0.00	85830	0.00	25	0.00	868777	0.00	1
17	ni_nopr	85900	102988	8.70	0.00	0.00	0.00	0	0.00	0	0.00	3502234	0.00	16

Table 67: TSP PLA family, NI codes

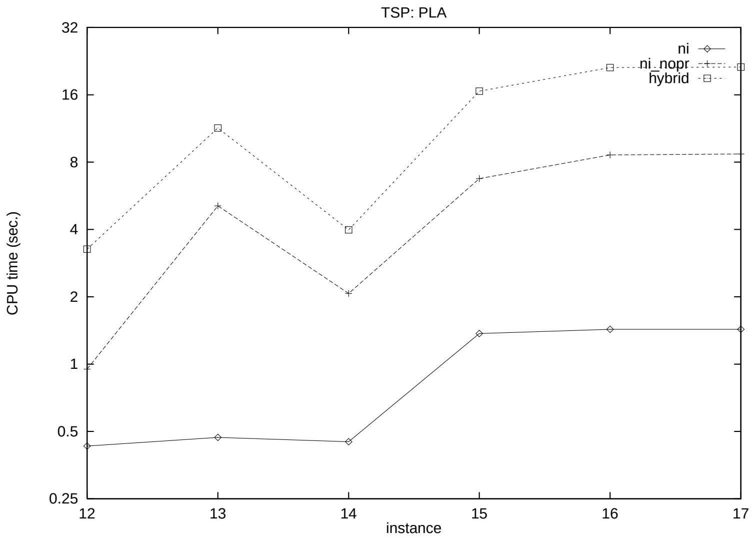

Figure 54: NI variants on TSP PLA family

problem		nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
				avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
18	hybrid	13509	15631	175.25	0.00	—	—	0	0.00	1222	0.00	—	—	2932
18	ni	13509	15631	0.20	0.00	0.12	0.00	12976	0.00	281	0.00	164535	0.00	15
18	ni_nopr	13509	15631	133.40	0.00	0.00	0.00	0	0.00	0	0.00	76095006	0.00	4222
19	hybrid	13509	17048	428.88	0.00	—	—	0	0.00	2591	0.00	—	—	6122
19	ni	13509	17048	0.42	0.00	0.13	0.00	12492	0.00	678	0.00	351284	0.00	59
19	ni_nopr	13509	17048	453.97	0.00	0.00	0.00	0	0.00	0	0.00	253341400	0.00	9015
20	hybrid	13509	17079	364.72	0.00	—	—	0	0.00	2099	0.00	—	—	5959
20	ni	13509	17079	0.42	0.00	0.13	0.00	12456	0.00	732	0.00	354701	0.00	58
20	ni_nopr	13509	17079	439.13	0.00	0.00	0.00	0	0.00	0	0.00	244780331	0.00	8929
21	hybrid	13509	17111	404.72	0.00	—	—	0	0.00	2140	0.00	—	—	6029
21	ni	13509	17111	0.43	0.00	0.13	0.00	12707	0.00	526	0.00	360997	0.00	67
21	ni_nopr	13509	17111	426.70	0.00	0.00	0.00	0	0.00	0	0.00	234260485	0.00	8822
22	hybrid	13509	17130	47.42	0.00	—	—	0	0.00	662	0.00	—	—	752
22	ni	13509	17130	0.25	0.00	0.13	0.00	12725	0.00	406	0.00	213941	0.00	14
22	ni_nopr	13509	17130	76.17	0.00	0.00	0.00	0	0.00	0	0.00	39957660	0.00	1627
23	hybrid	13509	17156	413.37	0.00	—	—	0	0.00	2410	0.00	—	—	6372
23	ni	13509	17156	0.45	0.00	0.15	0.00	12649	0.00	522	0.00	382971	0.00	66
23	ni_nopr	13509	17156	394.18	0.00	0.00	0.00	0	0.00	0	0.00	222172978	0.00	8662
24	hybrid	13509	17156	339.32	0.00	—	—	0	0.00	2151	0.00	—	—	5288
24	ni	13509	17156	0.40	0.00	0.15	0.00	12684	0.00	529	0.00	345126	0.00	54
24	ni_nopr	13509	17156	417.18	0.00	0.00	0.00	0	0.00	0	0.00	229023267	0.00	8127
25	hybrid	13509	17183	374.07	0.00	—	—	0	0.00	2122	0.00	—	—	4990
25	ni	13509	17183	0.35	0.00	0.13	0.00	12419	0.00	716	0.00	296352	0.00	35
25	ni_nopr	13509	17183	316.28	0.00	0.00	0.00	0	0.00	0	0.00	176724553	0.00	6640
26	hybrid	13509	17193	415.75	0.00	—	—	0	0.00	2319	0.00	—	—	6222
26	ni	13509	17193	0.48	0.00	0.15	0.00	12427	0.00	745	0.00	398840	0.00	71
26	ni_nopr	13509	17193	406.97	0.00	0.00	0.00	0	0.00	0	0.00	225077560	0.00	8669
27	hybrid	13509	17210	376.00	0.00	—	—	0	0.00	2300	0.00	—	—	5424
27	ni	13509	17210	0.47	0.00	0.15	0.00	12645	0.00	547	0.00	408682	0.00	75
27	ni_nopr	13509	17210	400.70	0.00	0.00	0.00	0	0.00	0	0.00	220310346	0.00	8345
28	hybrid	13509	17303	407.12	0.00	—	—	0	0.00	2115	0.00	—	—	6501
28	ni	13509	17303	0.45	0.00	0.15	0.00	12402	0.00	745	0.00	381080	0.00	59
28	ni_nopr	13509	17303	406.78	0.00	0.00	0.00	0	0.00	0	0.00	227860804	0.00	8746
29	hybrid	13509	17358	440.32	0.00	—	—	0	0.00	1990	0.00	—	—	6938
29	ni	13509	17358	0.47	0.00	0.15	0.00	12582	0.00	610	0.00	401551	0.00	71
29	ni_nopr	13509	17358	447.00	0.00	0.00	0.00	0	0.00	0	0.00	246194212	0.00	9248
30	hybrid	13509	17375	403.95	0.00	—	—	0	0.00	1875	0.00	—	—	5761
30	ni	13509	17375	0.42	0.00	0.15	0.00	12714	0.00	485	0.00	364594	0.00	61
30	ni_nopr	13509	17375	390.30	0.00	0.00	0.00	0	0.00	0	0.00	218803729	0.00	8614
31	hybrid	13509	17386	388.87	0.00	—	—	0	0.00	2086	0.00	—	—	5824
31	ni	13509	17386	0.43	0.00	0.15	0.00	12343	0.00	765	0.00	363960	0.00	48
31	ni_nopr	13509	17386	357.77	0.00	0.00	0.00	0	0.00	0	0.00	198333291	0.00	7791
32	hybrid	13509	17390	396.90	0.00	—	—	0	0.00	2435	0.00	—	—	6484
32	ni	13509	17390	0.47	0.00	0.15	0.00	12385	0.00	751	0.00	400896	0.00	61
32	ni_nopr	13509	17390	371.70	0.00	0.00	0.00	0	0.00	0	0.00	210429473	0.00	8295
33	hybrid	13509	17494	435.72	0.00	—	—	0	0.00	1981	0.00	—	—	6574
33	ni	13509	17494	0.43	0.00	0.17	0.00	12626	0.00	566	0.00	366605	0.00	52
33	ni_nopr	13509	17494	356.50	0.00	0.00	0.00	0	0.00	0	0.00	199561651	0.00	7668

Table 68: TSP USA family, NI codes

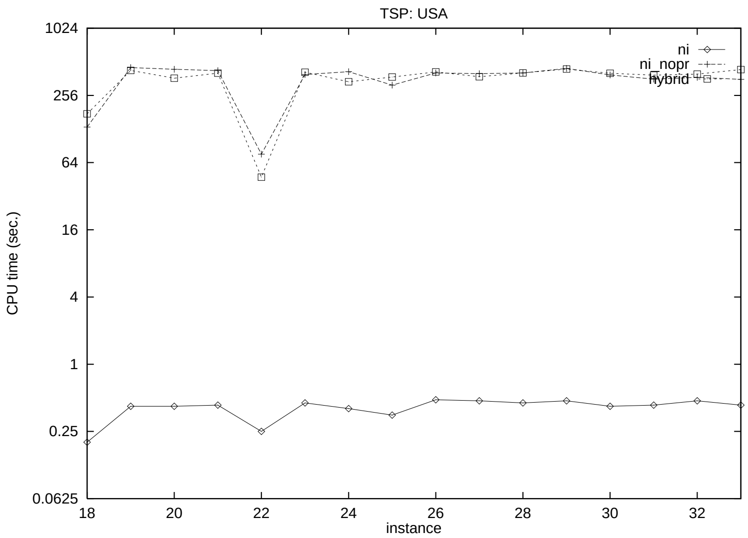

Figure 55: NI variants on TSP USA family

problem		nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
				avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
1	hybrid	20	38	0.00	0.00	—	—	0	0.00	3	0.00	—	—	14
1	ni	20	38	0.00	0.00	0.00	0.00	0	0.00	16	0.00	647	0.00	2
1	ni_nopr	20	38	0.00	0.00	0.00	0.00	0	0.00	0	0.00	1570	0.00	17
2	hybrid	88	185	0.02	0.00	—	—	0	0.00	11	0.00	—	—	53
2	ni	88	185	0.02	0.00	0.00	0.00	0	0.00	63	0.00	16960	0.00	20
2	ni_nopr	88	185	0.03	0.00	0.00	0.00	0	0.00	0	0.00	30592	0.00	81
3	hybrid	148	300	0.03	0.00	—	—	0	0.00	10	0.00	—	—	63
3	ni	148	300	0.02	0.00	0.00	0.00	0	0.00	94	0.00	21620	0.00	21
3	ni_nopr	148	300	0.03	0.00	0.00	0.00	0	0.00	0	0.00	55331	0.00	106
4	hybrid	2	1	0.00	0.00	—	—	0	0.00	0	0.00	—	—	0
4	ni	2	1	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00	0
4	ni_nopr	2	1	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00	0
5	hybrid	150	292	0.05	0.00	—	—	0	0.00	16	0.00	—	—	78
5	ni	150	292	0.02	0.00	0.00	0.00	0	0.00	95	0.00	23816	0.00	25
5	ni_nopr	150	292	0.03	0.00	0.00	0.00	0	0.00	0	0.00	48571	0.00	92
6	hybrid	261	517	0.13	0.00	—	—	0	0.00	19	0.00	—	—	143
6	ni	261	517	0.03	0.00	0.00	0.00	0	0.00	162	0.00	53429	0.00	37
6	ni_nopr	261	517	0.10	0.00	0.00	0.00	0	0.00	0	0.00	120138	0.00	150
7	hybrid	113	221	0.05	0.00	—	—	0	0.00	3	0.00	—	—	83
7	ni	113	221	0.03	0.00	0.00	0.00	0	0.00	71	0.00	34228	0.00	37
7	ni_nopr	113	221	0.03	0.00	0.00	0.00	0	0.00	0	0.00	51325	0.00	107
8	hybrid	106	208	0.03	0.00	—	—	0	0.00	3	0.00	—	—	80
8	ni	106	208	0.02	0.00	0.00	0.00	0	0.00	68	0.00	29148	0.00	32
8	ni_nopr	106	208	0.03	0.00	0.00	0.00	0	0.00	0	0.00	43842	0.00	99
9	hybrid	19	36	0.00	0.00	—	—	0	0.00	2	0.00	—	—	14
9	ni	19	36	0.00	0.00	0.00	0.00	0	0.00	15	0.00	612	0.00	2
9	ni_nopr	19	36	0.00	0.00	0.00	0.00	0	0.00	0	0.00	1212	0.00	15
10	hybrid	77	131	0.02	0.00	—	—	0	0.00	4	0.00	—	—	52
10	ni	77	131	0.00	0.00	0.00	0.00	0	0.00	54	0.00	9331	0.00	14
10	ni_nopr	77	131	0.02	0.00	0.00	0.00	0	0.00	0	0.00	16913	0.00	61
11	hybrid	605	1162	0.32	0.00	—	—	0	0.00	15	0.00	—	—	121
11	ni	605	1162	0.18	0.00	0.00	0.00	0	0.00	332	0.00	204810	0.00	65
11	ni_nopr	605	1162	0.32	0.00	0.00	0.00	0	0.00	0	0.00	323070	0.00	151

Table 69: PRETSP misc family, NI codes

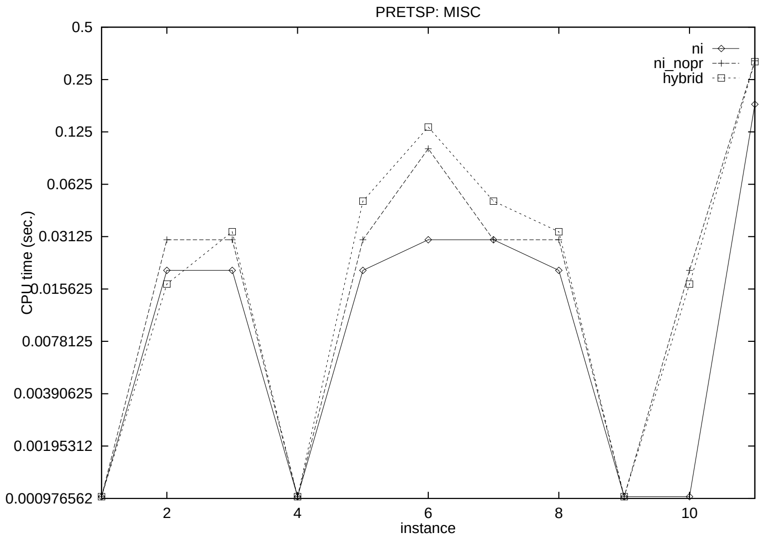

Figure 56: NI variants on PRETSP misc family

problem		nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
				avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
12	hybrid	2	1	0.00	0.00	—	—	0	0.00	0	0.00	—	—	0
12	ni	2	1	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00	0
12	ni_nopr	2	1	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00	0
13	hybrid	2	1	0.02	0.00	—	—	0	0.00	0	0.00	—	—	0
13	ni	2	1	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00	0
13	ni_nopr	2	1	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00	0
14	hybrid	2	1	0.00	0.00	—	—	0	0.00	0	0.00	—	—	0
14	ni	2	1	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00	0
14	ni_nopr	2	1	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00	0
15	hybrid	2	1	0.00	0.00	—	—	0	0.00	0	0.00	—	—	0
15	ni	2	1	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00	0
15	ni_nopr	2	1	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00	0
16	hybrid	2	1	0.00	0.00	—	—	0	0.00	0	0.00	—	—	0
16	ni	2	1	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00	0
16	ni_nopr	2	1	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2	0.00	0
17	hybrid	52	90	0.02	0.00	—	—	0	0.00	0	0.00	—	—	11
17	ni	52	90	0.02	0.00	0.00	0.00	0	0.00	31	0.00	2329	0.00	5
17	ni_nopr	52	90	0.00	0.00	0.00	0.00	0	0.00	0	0.00	2054	0.00	11

Table 70: PRETSP PLA family, NI codes

problem		nodes	arcs	total time		discovery time		initial PR		internal PR		edge scans		phases
				avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg	dev %	avg
18	hybrid	325	561	0.03	0.00	—	—	0	0.00	8	0.00	—	—	23
18	ni	325	561	0.02	0.00	0.00	0.00	0	0.00	135	0.00	28537	0.00	16
18	ni_nopr	325	561	0.03	0.00	0.00	0.00	0	0.00	0	0.00	28725	0.00	39
19	hybrid	477	920	0.30	0.00	—	—	0	0.00	26	0.00	—	—	122
19	ni	477	920	0.12	0.00	0.00	0.00	0	0.00	260	0.00	147999	0.00	57
19	ni_nopr	477	920	0.23	0.00	0.00	0.00	0	0.00	0	0.00	233164	0.00	144
20	hybrid	449	861	0.27	0.00	—	—	0	0.00	19	0.00	—	—	120
20	ni	449	861	0.13	0.00	0.00	0.00	0	0.00	226	0.00	149716	0.00	59
20	ni_nopr	449	861	0.23	0.00	0.00	0.00	0	0.00	0	0.00	202715	0.00	130
21	hybrid	454	886	0.27	0.00	—	—	0	0.00	11	0.00	—	—	128
21	ni	454	886	0.12	0.00	0.00	0.00	0	0.00	241	0.00	139677	0.00	60
21	ni_nopr	454	886	0.28	0.00	0.00	0.00	0	0.00	0	0.00	266286	0.00	167
22	hybrid	403	786	0.18	0.00	—	—	0	0.00	12	0.00	—	—	94
22	ni	403	786	0.13	0.00	0.00	0.00	0	0.00	224	0.00	126807	0.00	54
22	ni_nopr	403	786	0.17	0.00	0.00	0.00	0	0.00	0	0.00	184603	0.00	143
23	hybrid	532	1029	0.37	0.00	—	—	0	0.00	28	0.00	—	—	140
23	ni	532	1029	0.13	0.00	0.00	0.00	0	0.00	291	0.00	174077	0.00	55
23	ni_nopr	532	1029	0.25	0.00	0.00	0.00	0	0.00	0	0.00	260787	0.00	164
24	hybrid	501	950	0.27	0.00	—	—	0	0.00	16	0.00	—	—	133
24	ni	501	950	0.18	0.00	0.00	0.00	0	0.00	273	0.00	218389	0.00	71
24	ni_nopr	501	950	0.35	0.00	0.00	0.00	0	0.00	0	0.00	370106	0.00	202
25	hybrid	492	946	0.33	0.00	—	—	0	0.00	19	0.00	—	—	136
25	ni	492	946	0.15	0.00	0.00	0.00	0	0.00	256	0.00	149765	0.00	52
25	ni_nopr	492	946	0.20	0.00	0.00	0.00	0	0.00	0	0.00	208720	0.00	122
26	hybrid	549	1072	0.35	0.00	—	—	0	0.00	12	0.00	—	—	142
26	ni	549	1072	0.15	0.00	0.00	0.00	0	0.00	303	0.00	178225	0.00	63
26	ni_nopr	549	1072	0.33	0.00	0.00	0.00	0	0.00	0	0.00	334354	0.00	185
27	hybrid	465	926	0.27	0.00	—	—	0	0.00	17	0.00	—	—	117
27	ni	465	926	0.12	0.00	0.00	0.00	0	0.00	254	0.00	147139	0.00	57
27	ni_nopr	465	926	0.23	0.00	0.00	0.00	0	0.00	0	0.00	259703	0.00	158
28	hybrid	542	1064	0.33	0.00	—	—	0	0.00	28	0.00	—	—	114
28	ni	542	1064	0.13	0.00	0.02	0.00	0	0.00	293	0.00	145787	0.00	53
28	ni_nopr	542	1064	0.23	0.00	0.00	0.00	0	0.00	0	0.00	240893	0.00	136
29	hybrid	573	1104	0.38	0.00	—	—	0	0.00	21	0.00	—	—	149
29	ni	573	1104	0.20	0.00	0.00	0.00	0	0.00	313	0.00	198529	0.00	66
29	ni_nopr	573	1104	0.38	0.00	0.00	0.00	0	0.00	0	0.00	387008	0.00	207
30	hybrid	476	945	0.28	0.00	—	—	0	0.00	17	0.00	—	—	121
30	ni	476	945	0.15	0.00	0.00	0.00	0	0.00	247	0.00	177708	0.00	67
30	ni_nopr	476	945	0.30	0.00	0.00	0.00	0	0.00	0	0.00	319329	0.00	190
31	hybrid	607	1150	0.28	0.00	—	—	0	0.00	14	0.00	—	—	116
31	ni	607	1150	0.13	0.00	0.00	0.00	0	0.00	301	0.00	160409	0.00	46
31	ni_nopr	607	1150	0.23	0.00	0.00	0.00	0	0.00	0	0.00	224223	0.00	113
32	hybrid	557	1091	0.38	0.00	—	—	0	0.00	14	0.00	—	—	122
32	ni	557	1091	0.17	0.00	0.00	0.00	0	0.00	307	0.00	191347	0.00	59
32	ni_nopr	557	1091	0.32	0.00	0.00	0.00	0	0.00	0	0.00	308765	0.00	161
33	hybrid	505	971	0.30	0.00	—	—	0	0.00	10	0.00	—	—	124
33	ni	505	971	0.15	0.00	0.00	0.00	0	0.00	270	0.00	169536	0.00	68
33	ni_nopr	505	971	0.28	0.00	0.00	0.00	0	0.00	0	0.00	278663	0.00	159

Table 71: PRETSP USA family, NI codes

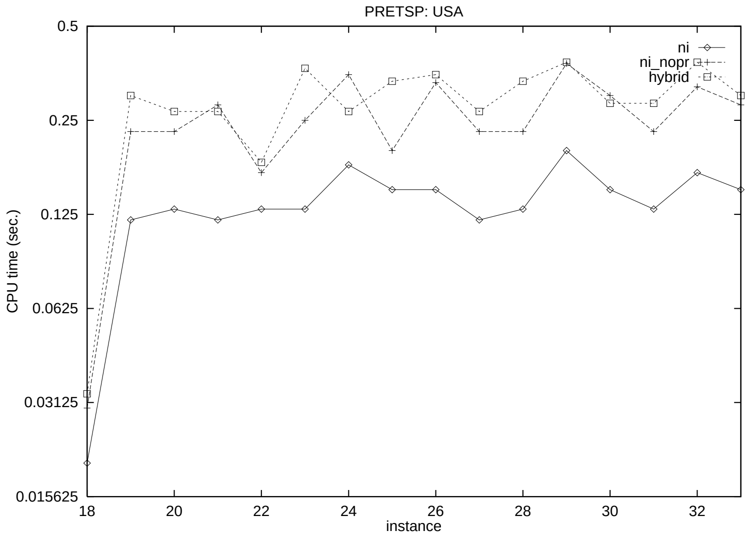

Figure 57: NI variants on PRETSP USA family

Acknowledgments

We would like to thank David Applegate, who provided the TSP instances, and Toshihide Ibaraki, who provided the `hybrid` code.

References

[1] R. K. Ahuja, J. B. Orlin, and R. E. Tarjan. Improved Time Bounds for the Maximum Flow Problem. *SIAM J. Comput.*, 18:939–954, 1989.

[2] R. J. Anderson and J. C. Setubal. Goldberg's Algorithm for the Maximum Flow in Perspective: a Computational Study. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 1–18. AMS, 1993.

[3] D. L. Applegate and W. J. Cook. Personal communication. 1996.

[4] R. A. Botafogo. Cluster Analysis for Hypertext Systems. In *Proc. of the 16-th Annual ACM SIGIR Conference of Res. and Dev. in Info. Retrieval*, pages 116–125, 1993.

[5] S. Chatterjee, J. R. Gilbert, R. Schreiber, and T. J. Sheffler. Array Distribution in Data-Parallel Programs. In *Languages and Compilers for Parallel Computing*, pages 76–91. Lecture Notes in Computer Science series, vol. 896, Springer-Verlag, 1996.

[6] J. Cheriyan and T. Hagerup. A randomized maximum flow algorithm. In *Proc. 30th IEEE Annual Symposium on Foundations of Computer Science*, pages 118–123, 1989.

[7] J. Cheriyan, T. Hagerup, and K. Mehlhorn. Can a Maximum Flow be Computed in $o(nm)$ Time? In *Proc. ICALP*, 1990.

[8] J. Cheriyan and S. N. Maheshwari. Analysis of Preflow Push Algorithms for Maximum Netwrok Flow. *SIAM J. Comput.*, 18:1057–1086, 1989.

[9] B. V. Cherkassky. A Fast Algorithm for Computing Maximum Flow in a Network. In A. V. Karzanov, editor, *Collected Papers, Vol. 3: Combinatorial Methods for Flow Problems*, pages 90–96. The Institute for Systems Studies, Moscow, 1979. In Russian. English translation appears in AMS Trans., Vol. 158, pp. 23–30, 1994.

[10] B. V. Cherkassky and A. V. Goldberg. On Implementing Push-Relabel Method for the Maximum Flow Problem. Technical Report STAN-CS-94-1523, Department of Computer Science, Stanford University, 1994.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[12] G. B. Dantzig, D. R. Fulkerson, and S. M. Johnson. Solution of a Large-Scale Traveling Salesman Problem. *Oper. Res.*, 2:393–410, 1954.

[13] U. Derigs and W. Meier. Implementing Goldberg's Max-Flow Algorithm — A Computational Investigation. *ZOR — Methods and Models of Operations Research*, 33:383–403, 1989.

[14] U. Derigs and W. Meier. An Evaluation of Algorithmic Refinements and Proper Data-Structures for the Preflow-Push Approach for Maximum Flow. In *ASI Series on Computer and System Sciences*, volume 8, pages 209–223. NATO, 1992.

[15] P. Elias, A. Feinstein, and C. E. Shannon. Note on Maximum Flow Through a Network. *IRE Transactions on Information Theory*, IT-2:117–199, 1956.

[16] L. R. Ford, Jr. and D. R. Fulkerson. Maximal Flow Through a Network. *Canadian Journal of Math.*, 8:399–404, 1956.

[17] H. N. Gabow. A Matroid Approach to Finding Edge Connectivity and Packing Arborescences. *J. Comp. and Syst. Sci.*, 50:259–273, 1995.

[18] A. V. Goldberg and R. E. Tarjan. A New Approach to the Maximum Flow Problem. *J. Assoc. Comput. Mach.*, 35:921–940, 1988.

[19] A. V. Goldberg and R. E. Tarjan. Finding Minimum-Cost Circulations by Canceling Negative Cycles. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 388–397, 1988.

[20] R. E. Gomory and T. C. Hu. Multi-terminal network flows. *J. SIAM*, 9:551–570, 1961.

[21] D. Gusfield. Very Simple Methods for All Pairs Network Flow Analysis. *SIAM Journal on Computing*, 19:143–155, 1990.

[22] J. Hao. A Faster Algorithm for Finding the Minimum Cut of a Graph. Unpublished manuscript, 1991.

[23] J. Hao and J. B. Orlin. A Faster Algorithm for Finding the Minimum Cut in a Directed Graph. *J. Algorithms*, 17:424–446, 1994.

[24] D. R. Karger. A randomized fully polynomial approximation scheme for the all terminal network reliability problem. In *Proc. 27th Annual ACM Symposium on Theory of Computing*, pages 11–17, 1995.

[25] D. R. Karger. Minimum Cuts in Near-Linear Time. In *Proc. 28th Annual ACM Symposium on Theory of Computing*, pages 56–63, 1996.

[26] D. R. Karger and C. Stein. An $\tilde{O}(n^2)$ Algorithm for Minimum Cuts. In *Proc. 25th Annual ACM Symposium on Theory of Computing*, pages 757–765, 1993.

[27] D.R. Karger. Random Sampling in Cut, Flow, and Network Design Problems. In *Proc. 26th Annual ACM Symposium on Theory of Computing*, pages 648–657, 1994. Submitted to *Math. of Oper. Res.*

[28] V. King, S. Rao, and R. Tarjan. A Faster Deterministic Maximum Flow Algorithm. *J. Algorithms*, 17:447–474, 1994.

[29] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnoy Kan, and D. B. Shmoys. *The Traveling Salesman Problem.* Wiley & Sons, 1985.

[30] M.V. Lomonosov and V.P. Poleskii. Lower bound of network reliability. *Problems of Information Transmission*, 7:118–123, 1971.

[31] D. W. Matula. A Linear Time $2 + \epsilon$ Approximation Algorithm for Edge Connectivity. In *Proc. 4th ACM-SIAM Symposium on Discrete Algorithms*, pages 500–504, 1993.

[32] H. Nagamochi and T. Ibaraki. Computing Edge-Connectivity in Multigraphs and Capacitated Graphs. *SIAM J. Disc. Meth.*, 5:54–66, 1992.

[33] H. Nagamochi, T. Ono, and T. Ibaraki. Implementing an Efficient Minimum Capacity Cut Algorithm. *Math. Prog.*, 67:297–324, 1994.

[34] Q. C. Nguyen and V. Venkateswaran. Implementations of Goldberg-Tarjan Maximum Flow Algorithm. In D. S. Johnson and C. C. McGeoch, editors, *Network Flows and Matching: First DIMACS Implementation Challenge*, pages 19–42. AMS, 1993.

[35] M. Padberg and G. Rinaldi. An Efficient Algorithm for the Minimum Capacity Cut Problem. *Math. Prog.*, 47:19–36, 1990.

[36] S. A. Plotkin, D. Shmoys, and É. Tardos. Fast Approximation Algorithms for Fractional Packing and Covering. In *Proc. 32nd IEEE Annual Symposium on Foundations of Computer Science*, 1991.

[37] A. Ramanathan and C. Colbourn. Counting Almost Minimum Cutsets with Reliability Applications. *Math. Prog.*, 39:253–261, 1987.

[38] D. D. Sleator and R. E. Tarjan. A Data Structure for Dynamic Trees. *J. Comput. System Sci.*, 26:362–391, 1983.