Heuristic and Genetic Algorithms for $O_{\parallel} C_{\text{max}}$

Xiao Fu, Samuel Karasik, Do Yeong Tak
Problem Statement

\[ M = \#machines \]
\[ J = \#jobs \]

- Each job may require processing on multiple machines
- Each job can be processed on at most one machine at any given time
- Jobs can be processed in any order
Real-life Applications

Repair Shop
- Vehicle with different repairs needed = jobs
- Mechanics = machines

Hospital Patient Scheduling
- Patient tests/examinations = jobs
- Doctors = machines
NP-Completeness of On||C_{max}, n>2

- On: operation to be done
  - p: integer duration;
  - Cn: completion time;
  - LB: max of job lengths and machine load
    - Null duration possible, if a job does not need a particular machine
    - Any instance of the problem can be coded by m, n, and a matrix of processing times, P: m x n
    - Any feasible schedule can be defined by a matrix C of completion times
- Objective: compute the makespan C_{max}
- NP-hardness proven for n = 3

Heuristic Algorithms

• We tested three algorithms:
  ○ List Scheduling
  ○ Longest Processing Time First (LPT)
  ○ Shortest Processing Time First (SPT)

• Algorithm procedure:
  ○ From the set of unscheduled jobs:
  ○ Sort the jobs according to the selected heuristic
  ○ Choose the process with the earliest possible start time
Genetic Algorithms: Overview

- Class of meta-heuristic algorithms based on the evolution of dominant gene sequences in a population
- Steps:
  - Encode candidate solutions as “chromosomes” - i.e. integer arrays
  - Select two “parent” solutions based on weights given by a fitness function
  - From the two parent solutions, create a child solution:
    - Crossover: taking a combination of integers from each parent
    - Mutation: random swaps in the sequence of integers
  - Compute fitness of child solution
  - Recalculate weights based on the quality of the new solution, repeat
  - Once population reaches terminal size, output best solution
Genetic Algorithms: Encoding

Encoding:
- We need to encode a solution as a “gene”
- Permutation of linear indices of the job matrix

E.g. for a 2 x 2 instance: C = [4 1 3 2] represents the solution where the jobs are scheduled in that order by the makespan subroutine.

<table>
<thead>
<tr>
<th>Sample Instance</th>
</tr>
</thead>
<tbody>
<tr>
<td>J1</td>
</tr>
<tr>
<td>M1            13 (1)</td>
</tr>
<tr>
<td>M2            19 (2)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>J1 (0, 13)</th>
<th>J2 (15, 41)</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M2</td>
<td>J2 (0, 15)</td>
<td>J1 (15, 34)</td>
</tr>
</tbody>
</table>
Roulette Wheel Selection:

- Each solution is given a weight according to its fitness:
  - \( \text{weight}_i = \frac{\text{sum(makespan}_j, j = 1 : \text{pop_size})}{\text{makespan}_i} \)
  - i.e. small makespans are given a large weight/roulette wheel section
- Normalize all weights to sum to 1, compute CDF
- Generate 2 uniform(0, 1) random numbers
- Select the 2 parent solutions according to the inverse of the CDF
Genetic Algorithms: Evolution Procedure

Uniform Crossover
- For the first parent solution, a uniform(0, 1) random number (u) is generated for each index and if u < 0.5 then we accept that index in the child solution.
- The remaining indices are taken from the second parent solution in the order that they appear.

Single Point Crossover
- We generate a uniform(0, N) random integer (v) and assign all indices from 1 to v from the first parent solution to the child solution.
- The remaining indices are taken from the second parent solution in the order that they appear.

Mutation
- Set a mutation probability (ours is fixed at 0.1)
- Generate a uniform(0, 1) random (w) if w < mutation_prob, then select two indices from the child solution at random and swap them.
Genetic Algorithms: Fitness Function

Makespan

○ select job to add to schedule, get job # (j), machine # (m), processing time (p)
○ calculate idle time slots on machine m and idle time slots for job j
○ select earliest available idle time on machine
○ compare with earliest time job is idle
○ if \( \min(j\text{-idle\_end, m\text{-idle\_end}) - \max(j\text{-idle\_start, m\text{-idle\_start})} \geq p \)
    ○ then schedule job \((m, j)\) to start at \(\max(j\text{-idle\_start, m\text{-idle\_start})\), STOP
○ else
    ○ move to next time slot that the job is idle
○ if all job idle times conflict with the machine idle time, move to next machine idle time
Genetic Algorithms: Performance

Running Time vs. Solution Quality

○ 4 x 4 instance:
  Number of candidate solutions:
  \(16! = 20,922,789,888,000\)
  pop_size = 10,000
  avg_gap = 0.0196

○ 7 x 7 instances:
  Number of candidate solutions:
  \(49! = 60825866643346582602953163655214780153766309783744100910902811425699774880000719507670832679013726758
    \)
  pop_size = 15,000
  avg_gap = 0.1537
Results for Benchmark Instances

- For smaller problem instances, GA gives a better result than heuristic algorithms.
- For large size instances, heuristic algorithms give better result than GA (only have 10,000 population size).
- We are restricted by computational power (my sad old MacBook Pro) so we were unable to simulate a population of size greater than 20,000 for large instances.
Benchmark Instance: 4 jobs and 4 machines

- GA performs best in all 10 instances, optimally solve 7 instances out of 10
- Heuristic algorithm performance depends highly on the instance
- Hard to claim that LPT is better than the other two heuristic algorithms
Extreme Problem Instances

Two extreme cases
- Number of jobs >> Number of machines
- Number of machines >> Number of jobs

Easier to solve
- LPT gives optimal solution for all instances in this situation
Performance of heuristics under different variances

- Instance size: 20 jobs and 20 machines
- Randomly generate 50 instances for one distribution range
- Uniformly distributed processing times
- LPT gives better results than List scheduling and SPT in this situation
- For LPT, the higher the variance of processing time, the harder to solve the problem optimally
Next Steps

- Further experimentation with the genetic algorithm by perturbing `pop_size`, `mutation_prob`, `crossover_type`
- Thorough analysis of GA run-time (pseudo-polynomial in `pop_size` and $M^J$?)
- Try some overnight computing on larger instances/obtain a supercomputer (Best Buy?)
- Finding more types of instances where the heuristic algorithms perform well
- Open source the GA package we wrote