Project Report

Topic: *Optimal Routing in Wavelength Division Multiplexing Networks*
Name: *Cahid Enes Keleş*
Student Number: *2019400201*

Problem Definition

Wavelength-division Multiplexing (WDM) is a technology which multiplexes a number of optical carrier signals onto a single optical fiber by using different wavelengths (i.e., colors) of laser light (Cai, Parks, 2015). **Wavelength Division Multiplexing Network** is a network that uses WDM in channels between links.

In our problem, there is a Wavelength Division Multiplexing Network. This network's links (nodes) and connections (edges) are given. There are some fixed number of source and destination pairs given, which are need to be connected through paths. These paths need to be assigned a wavelength. The constraint on the paths are such that each pair of paths should either be disjoint (share no edges) or should be assigned different wavelengths.

The objective is to satisfy as much source-destination path and wavelength assignments as possible given a fixed number of available wavelengths.

Brief Search

In the literature, this problem is solved using Integer Programming. Although this method finds the optimum solution, it is not scalable. *Ramaswami and Sivarajan (1994)* proposed a method to find an upper bound for this problem. *Chen and Banerjee (1996)* proposed a more efficient method to solve this problem. *Lee, Lee and Park (2000)* proposed a method to solve the corresponding integer programming problem of ring networks efficiently under some conditions. *Liimatainen and Honkanen (2008)* proposed a method for a specific network topology (torus network). *He, Brandt-Pearce and Subramaniam (2008)* proposed a method to solve the problem optimally with route selection rather than wavelength assignment.

Intended Solving Technique

I am planning to calculate the different paths between sources and destinations. Then select a random path for each source and destination, and assign random wavelengths. Then iteratively change the wavelength and paths to come to a better solution.

I am planning to use **OptaPlanner** (an open source constraint solver) to solve this problem. OptaPlanner contains various algorithms such as Tabu Search and Simulated Annealing, and only requires the constraints to work. OptaPlanner also has great report system which can graph optimization state history or compare methods.

I am planning to use different algorithms that are already implemented in OptaPlanner and compare these algorithms. At the and, I will choose the best algorithm and compare it with previous work in the literature.

Definitions

Problem: A network topology, a set of tasks, and available wavelength count.
Dense Problem: A problem in which all tasks cannot be satisfied.
Sparse Problem: A problem in which tasks can easily be satisfied.
Task: A source and a destination node in the network which should be connected through a path.
Path: A task, a wavelength, and a set of adjacent nodes. These nodes are meant to connect source and destination nodes but do not have to (may be in progress).
Complete Path: a path that connects source and destination.
State: A set of paths.
Move: A state change that turns a state into another state.

Implementation Progress

Moves

Previously it was planned to implement two kinds of moves:

- Add a new complete path
- Remove an existing path

Removing a path is easy but adding a new complete path turned out to be impossible to implement. While there are exponential order of complete paths from one node to another, choosing some set of them is not sensible. So the moves are changed as follows:

- Add a new node to a path (extend the path by 1 node)
- Remove the last node of an existing path (shorten the path by 1 node)
- Change the wavelength of the path

These moves are possible to implement.

Score

To score a state, 3-level scoring system is used: **hard**, **medium**, and **soft** score.
Hard Score: This score indicates the number of duplicate uses of channels. If this score is not 0, the state is not feasible.
Medium Score: This score indicates the number of satisfied tasks or equivalently number of complete paths. This score also corresponds to the objective function.
Soft Score: For a path, the soft score of the path indicates how many nodes need to be added to satisfy the task if channel capacity is ignored. The overall soft score is the sum of all the path's soft scores.

Two scores are compared according to **hard** scores first, then **medium** scores, then **soft** scores. The **hard** score is used to eliminate infeasible solutions. The **medium** score is used to maximize the objective function. The **soft** score is used to lead individual moves towards sensible moves.

First Tests

After these implementations, **Hill Climbing** is observed to be working well. But when **Tabu Search** was tried, it was seen that **Tabu Search** resulted in a similar performance to **Hill Climbing.** The reason for this is found to be that current move types cause deep local maxima which **Tabu Search** couldn't escape. To eliminate this, a new move type is introduced:

- Remove any number of nodes from an existing path.

After this change, it was seen that Tabu Search performed much better than **Hill Climbing**.

Improvements

The score calculation process is changed to calculate scores incrementally. This means the score for a state is not calculated from scratch, but only calculated changed variable effects. This made the algorithm significantly faster.

Another optimization was the initial state. Initially, each task was connected through the shortest path (ignoring channel capacities). After that, the algorithm was run to resolve capacity issues. This improvement made the algorithm %200 faster.

Other Optimization Methods

Other optimization methods are tried and results are compared with Tabu Search. The methods performed as follows:

- **Hill Climbing** performed very poorly.
- **Variable Neighborhood Descent** performed poorly.
- **Simulated Annealing** performed poorly.
- **Late Acceptance** performed poorly.
- **Great Deluge** performed poorly.
- **Step Counting Hill Climbing** performed poorly.
- **Strategic Oscillation** performed much better than Tabu Search

According to these results, **Strategic Oscillation** is selected.

Strategic Oscillation

Strategic Oscillation is an add-on, which works especially well with Tabu Search. Instead of picking the accepted move with the highest score, it employs a different mechanism: If there's an improving move, it picks it. If there's no improving move, however, it prefers moves that **improve a softer score level**, over moves that break a harder score level less.

```
while not timeout and score(current_state) != best_score:
	candidate_move = move which max{score(move)} for possible_moves()
	if score(candidate_move) > score(current_state):
		pick(move)
	candidate_move = move which max{score(move).medium} for possible_moves()
	if score(candidate_move) > score(current_state):
		pick(move)
```

```
        candidate_move = move which max{score(move).soft} for possible_moves()
        if score(candidate_move) > score(current_state):
                pick(move)
```

In our algorithm, **Strategic Oscillation** is used along with **Tabu Search**. To implement this, the **possible_moves** function is modified:

```
function possible_moves():
        possible_moves = [move if move not in last_n_moves]
```

Implemented Algorithm

When implementing the algorithm, the open-source optimization tool **OptaPlanner** is used. To use **OptaPlanner** in our problem, planning entities, a move generator, and a scoring system need to be implemented.

Planning Entities

Planning Entities are implemented as two classes: **Network** and **Path**. The **Network** class contains problem givens and path variables. The **Path** class contains source-destination pairs, a path that connects them (or does not connect), and an assigned wavelength. OptaPlanner fills the **Path** instances in the **Network** class while solving the problem.

```
public class Network {

   private ArrayList<ArrayList<Integer>> topology = new ArrayList<>();

   @PlanningEntityCollectionProperty
   private ArrayList<Path> paths = new ArrayList<>();

   @PlanningScore
   private HardMediumSoftScore score;

   public Network(String filename) throws FileNotFoundException {
      // read the problem from file
   }
}
```

```
public class Path {
   @PlanningId
   private int id;

   @PlanningVariable(valueRangeProviderRefs = {"wavelengthRange"})
   private Integer wavelength;
```

```
    @PlanningListVariable(valueRangeProviderRefs = {"nodeRange"})
    private ArrayList<Integer> path;
    private int source;
    private int destination;

    public Path(int id, int source, int destination) {
        this.id = id;
        this.source = source;
        this.destination = destination;
        wavelength = 0;
        path = new ArrayList<>();
        path.add(source);
    }
```

Move Generator

Given a state, **OptaPlanner** should know which moves are available. For that, 4 types of move classes are implemented: **ExtendPathMove**, **MultiExtendMove**, **RemovePathMove**, and **ChangeWavelengthMove**. Each move class should extend **AbstractMove** class, and implement some functions.
On top of these classes, a move list generator from a given state should be implemented and for that, the **MoveFactory** class is implemented.

```
public class MoveFactory implements MoveListFactory<Network> {
    @Override
    public List<? extends Move<Network>> createMoveList(Network network) {
        List<Move<Network>> moveList = new java.util.ArrayList<>();

        for (Path path: network.getPaths()) {
            if (!path.isDone()) {
                for (int neighbor:
network.getTopology().get(path.getPath().get(path.getPath().size()-1))) {
                    if (path.getPath().contains(neighbor)) continue;
                    moveList.add(new ExtendPathMove(path, neighbor));
                }
            }

            for (int i = 1; i < path.getPath().size(); i++) {
                moveList.add(new RemovePathMove(path, i));
            }

            for (int i = 0; i < network.getAvailableWavelengths(); i++) {
                if (path.getWavelength() == i) continue;
                moveList.add(new ChangeWavelengthMove(path, i));
            }
        }

        return moveList;
```

```
    }
}
```

Score Calculator

For the score calculator, an incremental score calculator is implemented to gain efficiency which is mentioned before.

```
@Override
    public void beforeVariableChanged(Object o, String s) {
        Path path = (Path) o;

        // hard
        lastSize = path.getPath().size();
        lastwl = path.getWavelength();
        lastPath = new ArrayList<>(path.getPath());

        // medium
        if (path.getPath().get(path.getPath().size()-1) ==
path.getDestination()) {
            medium -= 1;
        }

        // soft
        soft +=
dist[path.getPath().get(path.getPath().size()-1)][path.getDestination()];
    }

    @Override
    public void afterVariableChanged(Object o, String s) {
        Path path = (Path) o;

        // hard
        int cursize = path.getPath().size();
        if (cursize > lastSize) {
            // extended
            for (int i = lastSize-1; i < cursize-1; i++) {
                int src = path.getPath().get(i);
                int dst = path.getPath().get(i + 1);
                String edge = getEdge(src, dst, path.getWavelength());
                if (!occupancy.containsKey(edge)) occupancy.put(edge, 0);
                occupancy.put(edge, occupancy.get(edge) + 1);
                if (occupancy.get(edge) > 1) {
                    hard--;
                }
            }
        } else if (cursize < lastSize) {
            // removed
            for (int i = cursize-1; i < lastSize-1; i++) {
                int src = lastPath.get(i);
                int dst = lastPath.get(i + 1);
```

```
                String edge = getEdge(src, dst, lastwl);
                if (occupancy.get(edge) > 1) {
                    hard++;
                }
                occupancy.put(edge, occupancy.get(edge) - 1);
            }
        } else {
            // wavelength
            for (int i = 0; i < cursize-1; i++) {
                // add new wl
                int src = path.getPath().get(i);
                int dst = path.getPath().get(i + 1);
                String edge = getEdge(src, dst, path.getWavelength());
                if (!occupancy.containsKey(edge)) occupancy.put(edge, 0);
                occupancy.put(edge, occupancy.get(edge) + 1);
                if (occupancy.get(edge) > 1) {
                    hard--;
                }

                // remove old wl
                edge = getEdge(src, dst, lastwl);
                if (occupancy.get(edge) > 1) {
                    hard++;
                }
                occupancy.put(edge, occupancy.get(edge) - 1);
            }
        }

        // medium
        if (path.getPath().get(cursize-1) == path.getDestination()) {
            medium += 1;
        }

        // soft
        soft -= dist[path.getPath().get(cursize-1)][path.getDestination()];
    }
```

To generate sample problems to test the algorithms, a **python script** is used. This script generates a connected graph with **n** nodes and **m** edges, **k** source-destination pairs, and **w** wavelength channel count.

```
import random, pyperclip

n = 30
m = 40
k = 600
w = 3

s = ''
s += str(n) + ' ' + str(m) + '\n'

for i in range(n-1):
```

```
    s += str(random.randint(0, i)) + ' ' + str(i+1) + '\n'

for i in range(m-n+1):
    u = random.randint(0, n-1)
    v = u
    while v == u:
        v = random.randint(0, n-1)
    s += str(u) + ' ' + str(v) + '\n'

s += str(k) + '\n'
for i in range(k):
    u = random.randint(0, n-1)
    v = u
    while v == u:
        v = random.randint(0, n-1)
    s += str(u) + ' ' + str(v) + '\n'
s += str(w)
pyperclip.copy(s)
print('Copied to clipboard!')
```

The full project can be found in this repo:
https://github.com/cahidenes/optimal-wavelength-path-assignment/tree/main

Planned Procedure For Evaluating The Algorithm

I am planning to test the algorithm in 2 different sets of problems: **Sparse Problems** and **Dense Problems**.

Sparse Problems that our algorithm can solve optimally will be given to the algorithm and the time it takes for the algorithm to finish will be measured. The measured time will be compared with two optimal solving algorithms: **Integer Programming** (R. Ramaswami and K. N. Sivarajan, 1994) and **Max Flow Algorithm** (Gurzì, Nowé, Colitti, Steenhaut, 2009).

Dense Problems will be given to the algorithm and the resulting score will be compared with upper-bound calculated using **Integer Programming Relaxation** (R. Ramaswami and K. N. Sivarajan, 1994).

For each set, different sizes of problems will be used. The problems will be generated using the above algorithm.

Test Results

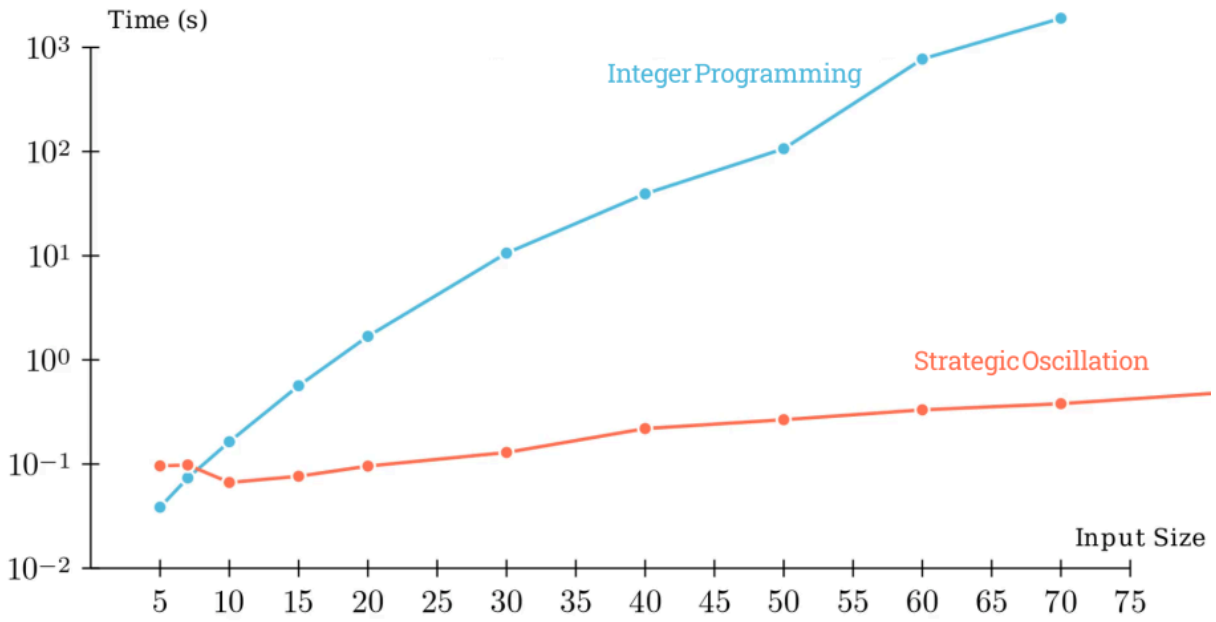
Sparse Problems

Max Flow Algorithm is not included in tests due to implementation overhead and the **Integer Programming** solution is more widespread. Integer Programming algorithm is implemented using **pulp** library with default solver. The implementation can be seen in the provided repository.

The solution proposed here and the **integer programming solution** are tested against various sizes of sparse problems. Input parameters for sparse inputs are determined as follows: For the input size **n**, there are

- **n** nodes
- **3n** edges (links)
- **n** source-destination pairs
- **5** available wavelengths

Both solutions are tested against different input sizes and they found the optimal answers. Both solutions' runtimes are measured and plotted. The comparison of the runtimes can be seen below.



Note that the y-axis of this plot is log-scaled. As can be seen from the plot, **My solution** works much faster than **integer programming**.

Dense Problems

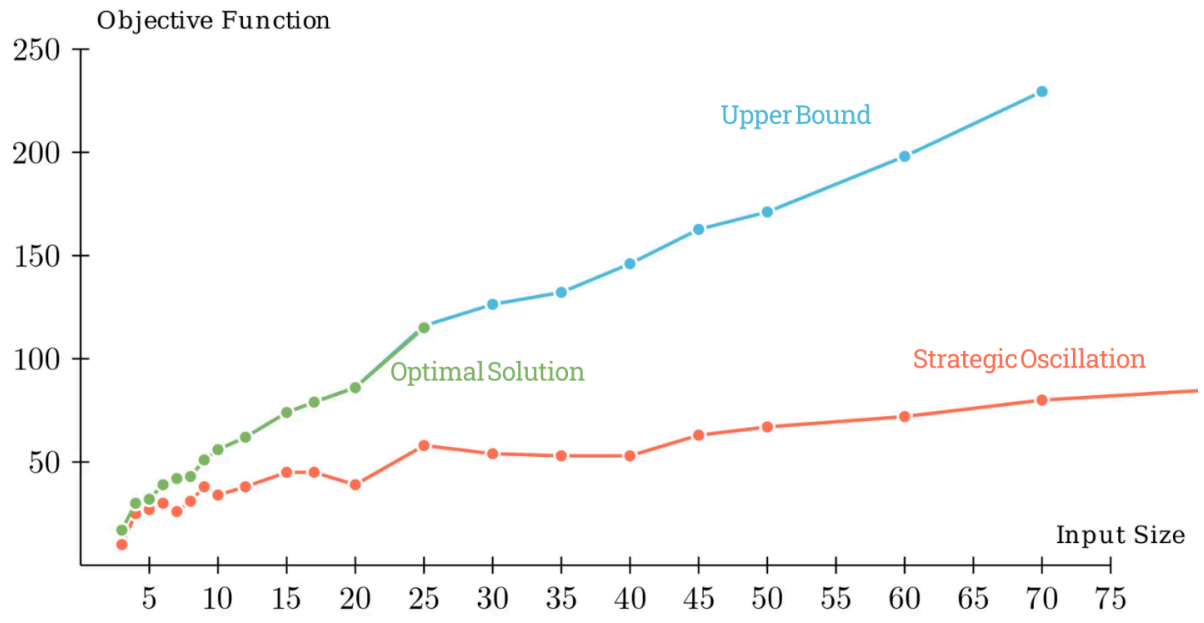
Linear relaxation of the integer programming solution is used to compare my solution with **dense problems.** The **linear relaxation** is implemented using pulp library with default solver. The implementation can be seen in the provided repository.

The input parameters of the dense inputs are defined as follows: For the input size **n**, there are

- **n** nodes
- **3n** edges (links)
- **15n** source-destination pairs
- **5** available wavelengths

The **linear relaxation** does not give a feasible solution but gives an upper bound. Both solutions are tested against various sizes of inputs and my solution's resulting objective function is compared with the upper bound.

For small inputs, the **integer programming** solution described above can also be used. For the small enough inputs in which **integer programming** runs in a reasonable time, **integer programming** solution is also run and the resulting objective function (which is the optimal solution) is measured. The comparison of the objective functions can be seen below.



As can be seen from the plot, my method works pretty bad compared to the optimal solution and the upper bound.

Conclusion

The Strategic Oscillation Method for the Path-Wavelength Assignment Problem works very fast. If many solutions exist in the search space, an optimal solution is obtained very quickly. However, if there are not many solutions, the method is not good at optimizing and finding good solutions. As a result, the **Strategic Oscillation Method** can be used to find a feasible but not very good solution very fast.

References

[1] Cai, Hong; Parks, Joseph. W (2015). "Optofluidic wavelength division multiplexing for single-virus detection". *Proceedings of the National Academy of Sciences of the United States of America*. 112 (42): 12933–12937. Bibcode:2015PNAS..11212933O. doi:10.1073/pnas.1511921112.

[2] R. Ramaswami and K. N. Sivarajan, "Optimal routing and wavelength assignment in all-optical networks," Proceedings of INFOCOM '94 Conference on Computer Communications, Toronto, ON, Canada, 1994, pp. 970-979 vol.2, doi: 10.1109/INFCOM.1994.337639.

[3] J. -P. Liimatainen and R. T. Honkanen, "Work-Optimal Routing in Wavelength-Division Multiplexed Dense Optical Tori," 2008 11th IEEE International Conference on Computational Science and Engineering, Sao Paulo, Brazil, 2008, pp. 9-14, doi: 10.1109/CSE.2008.34.

[4] Lee, Taehan, K. Lee, and Sungsoo Park. “Optimal Routing and Wavelength Assignment in WDM Ring Networks.” IEEE Journal on Selected Areas in Communications 18, no. 10 (2000): 2146–54. doi:10.1109/49.887934.

[5] Chien Chen and S. Banerjee, "A new model for optimal routing and wavelength assignment in wavelength division multiplexed optical networks," Proceedings of IEEE INFOCOM '96. Conference on Computer Communications, San Francisco, CA, USA, 1996, pp. 164-171 vol.1, doi: 10.1109/INFCOM.1996.497890.

[6] J. He, M. Brandt-Pearce and S. Subramaniam, "Optimal RWA for Static Traffic in Transmission-Impaired Wavelength-Routed Networks," in IEEE Communications Letters, vol. 12, no. 9, pp. 693-695, September 2008, doi: 10.1109/LCOMM.2008.080606.

[7] P. Gurzì, A. Nowé, W. Colitti and K. Steenhaut, "Maximum flow based Routing and Wavelength Assignment in all-optical networks," 2009 International Conference on Ultra Modern Telecommunications & Workshops, St. Petersburg, Russia, 2009, pp. 1-6, doi: 10.1109/ICUMT.2009.5345545.

[8] OptaPlanner [software]. Retrieved from https://www.optaplanner.org/