Application of Branch and Bound Algorithm to Solve Rider-Driver Batched Matching Problem

Puti Nabilla Aidira - 13521088
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13521088@std.stei.itb.ac.id

***Abstract*—The Rider-Driver Batched Matching Problem is a significant challenge in online transportation services, where a group of riders needs to be effectively matched with available drivers to minimize average waiting time. This paper explores the application of the Branch and Bound Algorithm to address this problem. This paper also discusses testing and analysis of the Branch and Bound algorithm compared to Exhaustive Search.**

Keywords—Branch and Bound; Rider-Driver Batched Matching; Online Transportation Services

I. Introduction

Rider-Driver Batched Matching is a technique used in online transportation services where a set of riders is matched with a set of available drivers. This batched matching is proven to reduce the average wait time more effectively compared to closest-pair matching, where a single rider immediately matched with the closest driver. However, this technique leads to a new problem which this paper address as the Rider-Driver Batched Matching Problem.

The Rider-Driver Batched Matching Problem's goal is to effectively and efficiently match a set of riders with available drivers in order to minimize average waiting time. With the same nature of minimizing average cost and the same matching process involved, this problem can be viewed as a variation of the popular job assignment problem. One of the generally used algorithms to solve this kind of optimization problem is Branch and Bound Algorithm.

The Branch and Bound Algorithm is an algorithm usually used to solve optimization problems. The algorithm considers a set of potential solutions as a rooted tree and systematically explores different combinations. The algorithm also systematically prunes unproductive branches, leading to an efficient search.

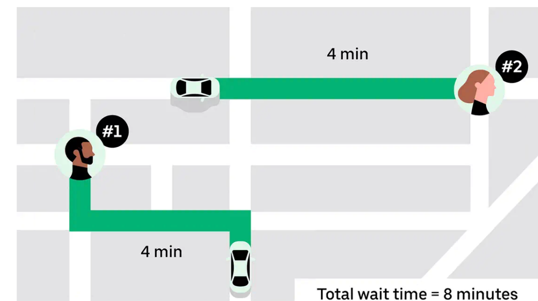

Fig 1. Example of Batched Matching Problem's Solution
(Source: www.uber.com/gb/en/marketplace/matching)

II. Theory

A. *Branch and Bound Algorithm*

The branch and Bound Algorithm is an algorithm commonly used for optimization problems. Optimization problems are problems related to either minimizing or maximizing an objective function of some variables under a certain constraint. As seen in [1], this definition can be written in formal notation:

$$\min_{x \in S} f(x) \text{ or } \max_{x \in S} f(x) \quad (1)$$

where, x : variables $(x_1 \cdots x_n)$,

f : objective function,

S : a region of feasible solutions.

Some terms and techniques used in Branch and Bound Algorithm are discussed below.

1. State Space Search

The Branch and Bound Algorithm conceptualizes the set of potential solutions as a rooted-tree structure, where the complete set of solutions serves as the root node. This rooted-tree structure is often called a state-space tree, while this searching process is usually called a state-space search. An example of a state-space tree constructed using Branch and Bound Algorithm is shown in Figure 2.

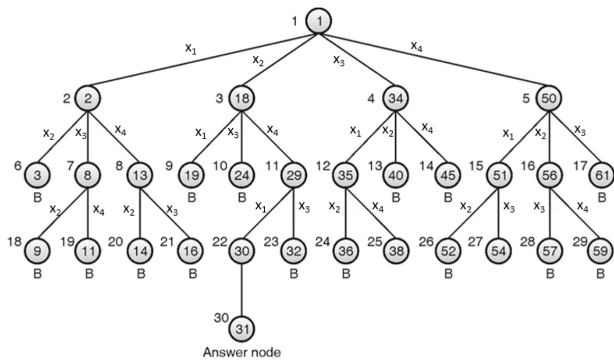

Fig 2. Example of a State-Space Tree Constructed Using Branch and Bound Algorithm
(Source: codecrucks.com/branch-and-bound-the-dummies-guide/)

2. Best-First Search

The traversal approach of the state-space tree can be done in a breadth-first, depth-first, or best-first manner. However, the most commonly used approach is a best-first search based on bounding estimation cost calculation. In this approach, each node is assigned a cost value $\hat{c}(i)$ calculated by a certain bounding estimation approach. Then, the next expanded nodes will be chosen from the least-cost node (for minimizing problems) or the most-cost node (for maximizing problems).

3. Bounding Function

One of the key processes in the Branch and Bound algorithm is the pruning of sub-problems that do not lead to the optimal solution. This pruning process is done using a bounding function. According to [1] the value of a bounding function for a given sub-problem should be an estimation of the best feasible solution to the problem. Some general criteria for bounding function, according to [2], are:

- The last unexpanded node of a given sub-problem has a cost whose value is not better than the value of the best solution found so far.
- The sub-problem does not represent a feasible solution due to the violation of some constraints.
- The solution at the given node consists of only one point.

Using the terms and techniques explained above, the pseudocode of the Branch and Bound algorithm is shown below.

```
function BranchAndBound(problem)
begin
      create an empty priority queue Q
      create an initial node
      compute initial node’s cost
      add the initial node to Q

      while Q is not empty do
          select the node with the best cost
          from Q
          if the selected node represents a
          complete solution
          then
              update the current best
              solution
          else
              generate child nodes
              for each child node do
                compute the cost of the
                child node
                if the cost is better than
                   the current best
                   solution
                then
                   add the child node to Q
                else
                   do nothing {bounding}
      return the best solution found
end
```

B. *Rider-Driver Batched Matching Problem*

Rider-Driver Batched Matching Problem is an optimization problem that arises from the Rider-Driver Batched Matching technique. Rider-Driver Batched Matching is a technique used in online transportation services where a set of riders is matched with a set of available drivers. In contrast to closest-pair matching, where a single rider immediately matched with the closest driver, batched matching involves batching time. The batching time is a few seconds of waiting time used to accumulate a batch of potential rider-driver matches. This batching time is important to avoid potentially long wait times for other slightly later requests. The illustration of a case where batch matching is minimizing average wait time more effectively compared to closest-pair matching is shown in Figure 3.

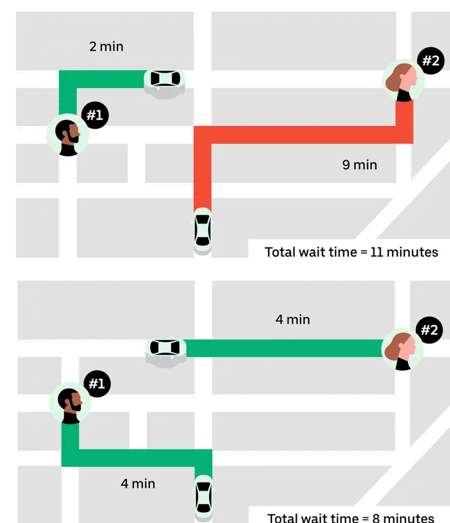

Fig 3. Batch Matching (top) vs Closest-Pair Immediate Matching (bottom)
(Source: www.uber.com/gb/en/marketplace/matching/)

In Figure 3, the second rider's request is a few seconds late than the first rider's request. So, with Closest-Pair Immediate Matching the first rider will immediately get the driver with a two-minute waiting time first. However, that match extremely increases the waiting time for the second rider. Resulting in a longer total waiting time hence longer average waiting time. This problem is solved with a few seconds of batching time, waiting for the second rider's request. After some batching time, a specific matching optimization can be used to minimize the average waiting time considering both riders. Even though this solution involves extra wait time, it is still considered the most effective in average cases.

The matching optimization in the Rider-Driver Batched Matching technique is the goal of the Rider-Driver Batched Matching Problem. However, the problem discussed in this paper will be limited to the problem where in one batch the number of riders is equal to the number of drivers. Hence, every rider will be matched to one driver and vice versa.

III. Implementation

The detailed implementation of the Branch and Bound algorithm to solve the Rider-Driver Batched Matching Problem is discussed below. A Java program containing the implementation of Branch and Bound algorithm as well as the simulation of its application in Rider-Driver Batched Matching is provided in [3]. The program will be walked through in the following discussion.

A. *Cost Matrix Representation*

The problem is represented in the form of a cost matrix whose element[i][j] is the amount of waiting time between rider i and driver j. Figure 4 illustrates the cost matrix with n riders and n drivers.

$$C = \begin{bmatrix} driver\ 1 & driver\ 2 & \dots & driver\ n \\ x_{1,1} & x_{1,2} & \dots & x_{1,n} \\ x_{2,1} & x_{2,2} & \dots & x_{2,n} \\ \vdots & \vdots & & \vdots \\ x_{n,1} & x_{n,2} & \dots & x_{n,n} \end{bmatrix} \begin{matrix} rider\ 1 \\ rider\ 2 \\ \vdots \\ rider\ n \end{matrix}$$

Fig 4. Cost Matrix with *n* Riders and *n* Drivers

To better illustrate the implementation, the case represented by the cost matrix in Figure 5 shall be used throughout this implementation section.

$$C_{ex} = \begin{bmatrix} driver\ 1 & driver\ 2 & driver\ 3 & driver\ 4 \\ 65 & 2 & 31 & 37 \\ 32 & 26 & 30 & 14 \\ 35 & 15 & 22 & 32 \\ 43 & 72 & 9 & 74 \end{bmatrix} \begin{matrix} rider\ A \\ rider\ B \\ rider\ C \\ rider\ D \end{matrix}$$

Fig 5. Cost Matrix for Example Case

In the simulation program [3], the waiting time was calculated using the Manhattan distance. While the cost matrix is constructed randomly using this snippet of code:

MainSimulation.java

```
Position posFirstRider = new Position(firstRiderX,
firstRiderY); // form input
costMatrix[0][0]  = posFirstRider.manhattanDist(new
                    Position()); // random position
int lastAdded = 1;
// batchtime from input
for(int i = 0; i < batchTime; i++){
int thisAdded = rand.nextInt(4); // generate random num
                                  of drivers & riders
   for(int j = lastAdded ; j < lastAdded +
       thisAdded; j++){
     for(int k = 0 ; k < lastAdded +
         thisAdded; k++){
         // generate random position
         Position riderPos = new Position();
         costMatrix[j][k] = riderPos.manhattanDist(new
                            Position());
         }
     }
   for(int j = 0 ; j < lastAdded; j++){
     for(int k = lastAdded ; k < lastAdded + thisAdded;
         k++){
         // generate random position
         Position riderPos = new Position();
         costMatrix[j][k] = riderPos.manhattanDist(new
                            Position());
         }
     }
    lastAdded += thisAdded;
}
```

B. *Bounding Function*

The bounding function is based on a lower bound cost ĉ(i). The pruning is done implicitly using a priority queue of least cost, nodes with bigger cost will not be accessed and hence be pruned. The lower bound estimation used to calculate the cost ĉ(i) is the sum of the minimum value in each of the cost matrix's rows. The formal notation is shown in (2).

$$\hat{c}(i) = \sum_{i=0}^{n} \min\left(costMatrix_{i,j}\right) \quad (2)$$

As mentioned in [4], the basic idea is that in any given solution, including the optimal solution, the total matching cost is not less than the sum of all the smallest values in each row. Other than that, for any legitimate solution (with no overlapping matching) if a rider is matched to a driver, then the cost of the matching is calculated as one of the smallest value components in the sum. The code implementation in [3] for this cost calculation is shown below.

BatchedMatchingBnB.java

```
private int calculateCost(int riderIdx, boolean[]
                          assigned)
{
 int cost = 0;
 for (int i = riderIdx + 1; i < N; i++) {
   int min = Integer.MAX_VALUE;
   for (int j = 0; j < N; j++) {
     if (!assigned[j] && costMatrix[i][j] < min) {
         min = this.costMatrix[i][j];
     }
    }
   cost += min;
  }
 return cost;
}
```

C. *State-Space Tree Construction*

The state-space tree is implemented as a group of nodes with a parent. The root node has null as its parent. The node class is implemented with the code below.

Node.java

```
class Node {
   // parent node
   private Node parent;

   // cost for ancestors nodes including current node
   private int pathCost;

   // least promising cost
   private int cost;

   // matched rider number
   private int riderID;

   // matched driver number
   private int driverID;

   // available driver info
   private boolean[] assigned;

   // .. constructor, getter/setter
}
```

The implementation of state-space tree construction is shown in the below code.

BatchedMatchingBnB.java

```
public int branchAndBound() {
   // initialize priority queue based on less cost
   PriorityQueue<Node> pq = new PriorityQueue<>(
                              newNodeComparator());
   // set root
   Node root = new Node(-1, -1, new boolean[N], null);
   root.setPathCost(0);
   root.setCost(0);
   pq.add(root);
   while (!pq.isEmpty()) {
     Node min = pq.poll();
     int i = min.getRiderID() + 1;
     if (i == N) {
        // all rider has been matched
        // solution found
        printAssignments(min); // print solution
        return min.getCost();
     }
     for (int j = 0; j < N; j++) {
       if (!min.getAssigned()[j]) {
        // create child node with
        // riderId = i, driverId = j, parrent = min,
        // and boolean array assigned passed from
        // parent nodes
        Node child = new Node(i, j,
                          min.getAssigned(), min);
        // set path cost to child node

        child.setPathCost(min.getPathCost() +
                            this.costMatrix[i][j]);
        // set cost to child node

        child.setCost(child.getPathCost() +
                calculateCost(i, child.getAssigned()));
          pq.add(child); // add child to prioqueue
        }
      }
    }
    return -1; // solution not found
}
```

The algorithm iteratively explores nodes in the priority queue until it finds a complete matching (all riders are matched). In each iteration, it retrieves the node with the minimum cost from the priority queue. If the last retrieved nodes indicate that all riders are matched (min.getRiderID() + 1 = N), the solution found thus it will be printed.

However, if it doesn't, the algorithm will generate child nodes for each unassigned driver at the current level. It creates a child node by cloning the assigned array and setting the assigned status for the current driver. The child node's path cost is updated by adding the cost of assigning the rider i to the driver j. The total cost of the child node is computed by adding the path cost and the lower bound estimated cost for the remaining unassigned riders.

The child node is then added to the priority queue for further exploration. The algorithm continues this process until the priority queue is empty, indicating that all possible assignments have been explored without finding a complete assignment.

Using the case in Figure 5, the state-space tree construction as well as the node cost calculation and priority queue condition in each iteration is shown below.

1. Iteration 1: Rider A Matching

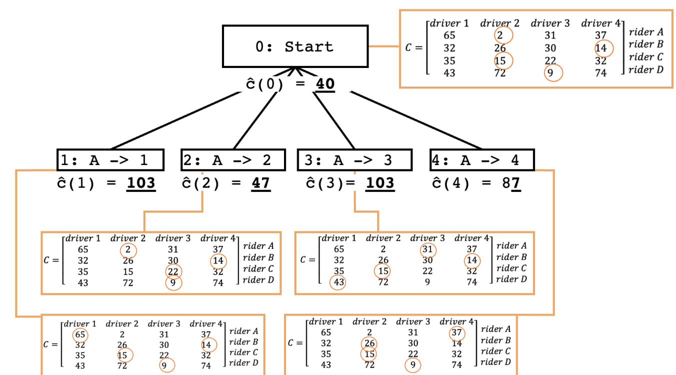

Fig 6. State-Space Tree for The 1st Iteration with Cost Calculation

The circled numbers in the matrices indicate the minimum cost chosen for each row. Based on the cost ĉ(i), the priority queue will look like this:

```
// start node added then immediately polled
pq = [40]
```

```
// queue after 1st iteration finished
pq = [47, 87, 103, 103]
```

2. Iteration 2: Rider B Matching

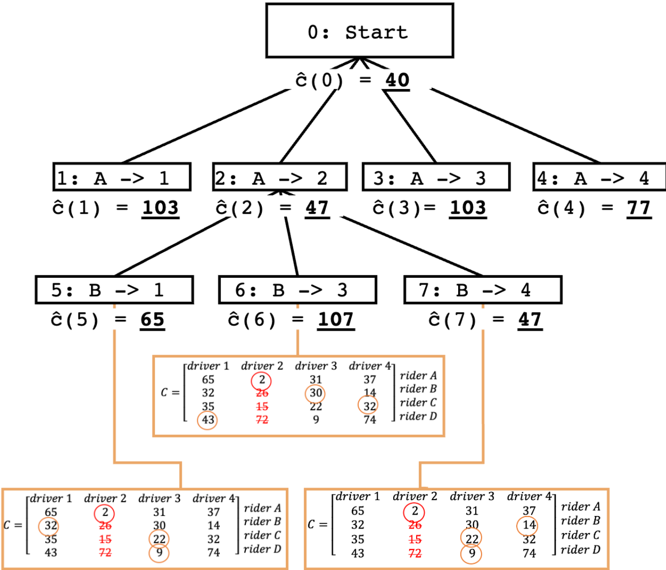

Fig 7. State-Space Tree for The 2nd Iteration with Cost Calculation

The numbers marked in red indicate the matching from the previous iteration. The driver that is already matched can not be chosen for minimum cost. Based on the cost ĉ(i), the priority queue will look like this:

```
// queue after polling
pq = [87, 103, 103]

// queue after 2nd iteration finished
pq = [47, 65, 87, 103, 103, 107]
```

3. Iteration 3: Rider C Matching

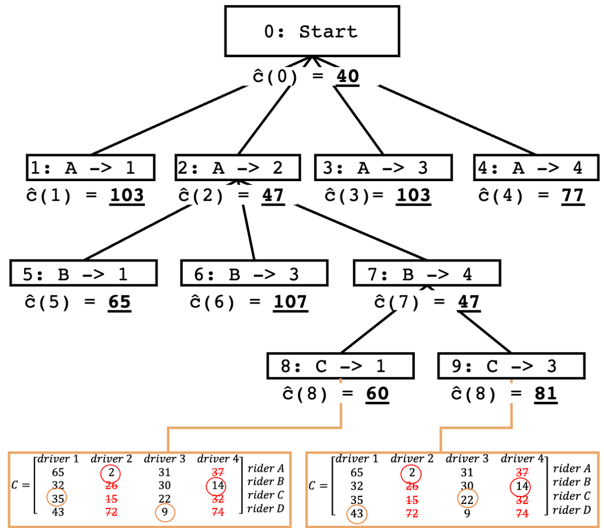

Fig 8. State-Space Tree for The 3rd Iteration with Cost Calculation

Based on the cost ĉ(i), the priority queue will look like this:

```
// queue after polling
pq = [65, 87, 103, 103, 107]

// queue after 3rd iteration finished
pq = [60, 65, 81, 87, 103, 103, 107]
```

4. Iteration 4: Rider D Matching

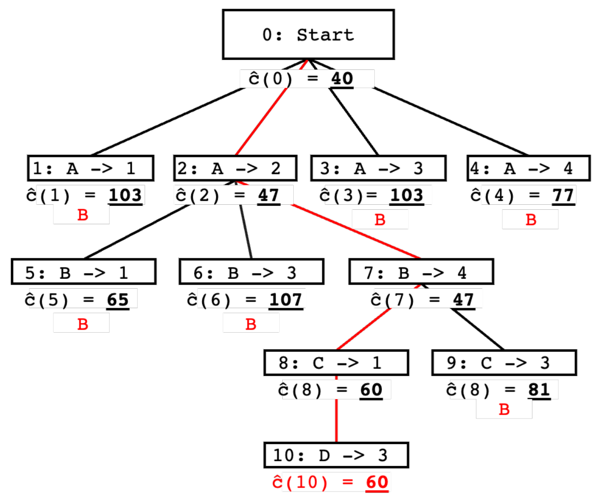

Fig 9. State-Space Tree for The 4th Iteration with Cost Calculation

Since there are no other nodes that cost less than 60 and all riders are already matched, the other nodes are pruned. The pruned nodes are marked with a red "B" (bounded). The solution path is marked red, which is A →

2, B → 4, C →1, D → 3. The total waiting time is 60 minutes. Hence, the minimum average waiting time is 60/4 = 15 minutes.

IV. Testing and Analysis

To evaluate the efficiency of the Branch and Bound application to the Rider-Driver Batched Matching Problem, the program is being tested on various cases and against Brute Force exhaustive search. The detailed testing and analysis are discussed below.

A. *Time Complexity Analysis*

In Branch and Bound, the algorithm iterates through the matrix with a double loop. Since the matrix size is N x N, the time complexity of Branch and Bound algorithm is $O(n^2)$. In Exhaustive Search, the algorithm generates all permutations of matching, hence the time complexity is O(n!).

B. *Test Results*

TABLE I. Test Case 1

Test Case 1 (same as Figure 5.)	
First rider position (x, y): (0, 0)	Batch time (s): 2
Cost Matrix Result	

Cost Matrix (waiting time in minute(s)):

	Driver 1	Driver 2	Driver 3	Driver 4
Rider A	65	2	31	37
Rider B	32	26	30	14
Rider C	35	15	22	32
Rider D	43	72	9	74

Branch and Bound	
Matching Result	
Matching: Rider A -> Driver 2 (2 min) Rider B -> Driver 4 (14 min) Rider C -> Driver 1 (35 min) Rider D -> Driver 3 (9 min) Average waiting time is 15 min	Execution time: 4.053834 ms
Brute Force (Exhaustive Search)	
Matching Result	Execution time:
Average waiting time is 15 min	0.8275 ms

TABLE II. Test Case 1

Test Case 2	
First rider position (x, y): (0, 0)	Batch time (s): 2
Cost Matrix Result	

Cost Matrix (waiting time in minute(s)):

	Driver 1	Driver 2	Driver 3	Driver 4	Driver 5	Driver 6	Driver 7
Rider A	41	22	29	35	10	39	25
Rider B	37	9	33	14	40	7	56
Rider C	29	22	9	9	30	38	75
Rider D	32	72	28	67	37	44	56
Rider E	41	24	18	23	22	43	73
Rider F	60	47	24	54	6	46	37
Rider G	18	23	44	5	19	54	38

Branch and Bound	
Matching Result	
Matching: Rider A -> Driver 7 (25 min) Rider B -> Driver 6 (7 min) Rider C -> Driver 3 (9 min) Rider D -> Driver 1 (32 min) Rider E -> Driver 2 (24 min) Rider F -> Driver 5 (6 min) Rider G -> Driver 4 (5 min) Average waiting time is 15 min	Execution time: 2.764208 ms
Brute Force (Exhaustive Search)	
Matching Result	Execution time:
Average waiting time is 15 min	3.096084 ms

TABLE III. Test Case 1

Test Case 3	
First rider position (x, y): (1, 1)	Batch time (s): 5
Cost Matrix Result	

Cost Matrix (waiting time in minute(s)):

	Driver 1	Driver 2	Driver 3	Driver 4	Driver 5
Rider A	8	36	39	11	30
Rider B	41	37	55	47	57
Rider C	42	29	49	46	52
Rider D	30	8	65	51	24
Rider E	21	27	27	30	40
Rider F	50	18	39	40	44
Rider G	39	11	19	51	11
Rider H	18	36	78	65	44
Rider I	33	24	43	35	41
Rider J	36	24	21	44	17
Rider K	14	24	28	21	18

Driver 6	Driver 7	Driver 8	Driver 9	Driver 10	Driver 11
19	45	7	64	24	5
10	50	32	29	26	28
42	25	38	76	20	27
21	43	32	4	29	22
23	22	48	23	1	16
18	33	27	56	15	23
16	64	42	9	56	4
57	24	37	34	49	66
50	33	26	27	62	34
7	43	39	14	18	36
9	43	17	51	22	19

Branch and Bound	
Matching Result	
Matching: Rider A -> Driver 4 (11 min) Rider B -> Driver 6 (10 min) Rider C -> Driver 7 (25 min) Rider D -> Driver 9 (4 min) Rider E -> Driver 10 (1 min) Rider F -> Driver 2 (18 min) Rider G -> Driver 11 (4 min) Rider H -> Driver 1 (18 min) Rider I -> Driver 8 (26 min) Rider J -> Driver 3 (21 min) Rider K -> Driver 5 (18 min) Average waiting time is 14 min	Execution time: 6.474417 ms
Brute Force (Exhaustive Search)	
Matching Result	Execution time:

Average waiting time is 14 min	834.39675 ms

In all test cases, both Branch and Bound and Exhaustive Search succeed in generating the least average waiting time. The difference between Branch and Bound and Exhaustive Search on those test cases is in terms of execution time needed.

In test case 1, the execution time needed for Branch and Bound is longer than Exhaustive Search. That is because the matrix size is relatively small. Exhaustive Search, with time complexity O(n!), is better with a smaller sized matrix compared to Branch and Bound, with $O(n^2)$. The graph that shows how O(n!) behaves compared to $O(n^2)$ in small n is shown in Figure 10.

For a relatively average-sized matrix, like the one shown in test case 2, the execution time needed for Branch and Bound is shorter than Exhaustive Search. However, the difference is not significant and often similar.

However, for the large-sized matrix, like the one in test case 3, the execution time needed for Exhaustive Search is extremely longer than Branch and Bound. This is consistent with the extremely steep graph of O(n!). The graph that shows how O(n!) behaves compared to $O(n^2)$ in large n is shown in Figure 11.

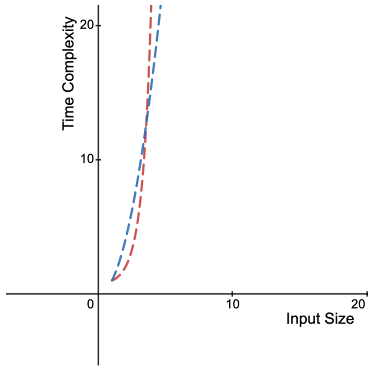

Fig 10. O(n!) (red) compared to $O(n^2)$ (blue) in smaller input size

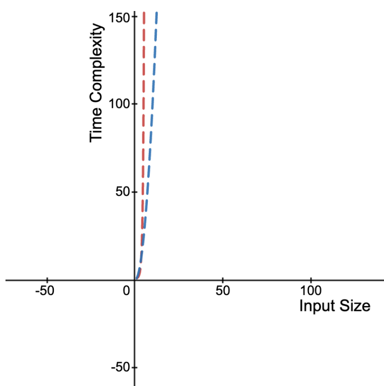

Fig 11. O(n!) (red) compared to $O(n^2)$ (blue) in larger input size

V. Conclusion

In conclusion, Branch and Bound algorithm is a promising algorithm to solve Rider-Driver Batched Matching Problem. The algorithm succeeds in generating a correct and complete solution to the problem. It has $O(n^2)$ time complexity which is generally better compared to Exhaustive Search in a large input. However other limitations such as space complexity also need to be further analyzed. Other real-life factors in Rider-Driver Batched Matching such as driver acceptance rates, traffic, etc. also need to be considered in a real Rider-Driver Batched Matching Algorithm.

Video Link at YouTube

https://youtu.be/AHhN96L-uaE

Acknowledgment

First and foremost, I would like to thank Allah Swt. as without his blessing I wouldn't be able to finish this paper. Second, I would also like to thank Dr. Nur Ulfa Maulidevi, S.T., M.Sc. as my Discrete Mathematics lecturer for her lectures that inspire me to write this paper. Not to forget, I wish to show my appreciation for Dr. Ir. Rinaldi Munir, MT. as his lecture materials help me to compose this paper. Lastly, I also want to thank all of my friends and colleagues who always support me.

References

[1] Clausen, Jens (1999). Branch and Bound Algorithms—Principles and Examples (PDF) (Technical report). University of Copenhagen, pp 4-13. Archived from the original (PDF) on 2015/09/23. [Accessed May 21, 2023].

[2] Munir, Rinaldi. "Algoritma Branch & Bound (Bagian 1)". Program Studi Teknik Informatika STEI ITB: 2021, https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Algoritma-Branch-and-Bound-2021-Bagian1.pdf. [Accessed May 21, 2023].

[3] P.N. Aidira, "Rider-Driver-Batched-Matching-Simulation," Github. [Online]. Available: https://github.com/Putinabillaa/Rider-Driver-Batched-Matching-Simulation. [Accessed May 21, 2023].

[4] R. Munir and Masayu L.K. "Algoritma Branch & Bound (Bagian 4)". Program Studi Teknik Informatika STEI ITB: 2021, https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2021-2022/Algoritma-Branchand-Bound-2022-Bagian4.pdf. [Accessed May 21, 2023].

STATEMENT

With this I acknowledge that this paper is a writing of my own, and neither a copy, translation of other papers, nor a plagiarism.

Bandung, 22 Mei 2023

Puti Nabilla Aidira 13521088