REPORT ON MODIFIED KNIGHT'S PATH PROJECT - Group 7

Members:

1/ Nguyễn Tuấn Dũng - 20194427

2/ Nguyễn Việt Hoàng - 20194434

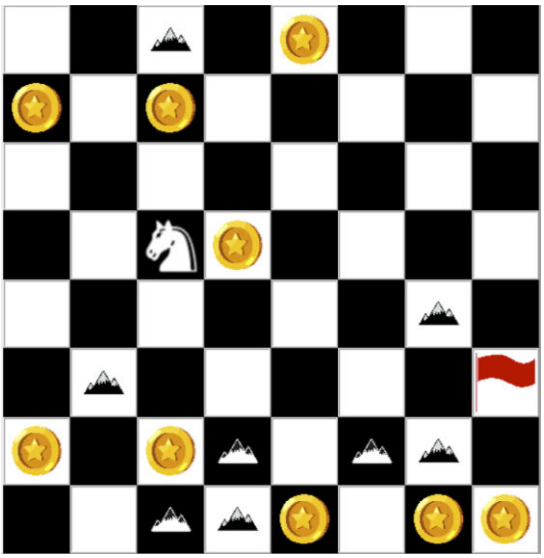
3/Phan Đức Thắng - 20194452

Table of content

1. Presentation of the subject

Given a chess board of size N x N, on the board are: a knight, m coins, p obstacles, and a destination ,all are generated on random squares of the board. The knight can move in a 2x1 ‘L’ shape like in regular Chess rule. The journey of the Knight must end on the destination square.

If the knight steps into the square with the coin, it will collect the coin and gain 3 points, the knight can not step into the square with the obstacles and if it steps into a normal square (that has nothing on it), it will lose 1 point. The program will find the path that maximizes the number of points to the destination.



2. Description of the problem

- This is a search problem with known, observable, deterministic, discrete and dynamic environments.

- Known: the agent knows the positions of obstacles, coins, destination and the knight.
- Observable: At each state, the knight can “observe” the next square.
- Deterministic: The next state is completely determined by the current state and the next move of the knight.

- Discrete: Limited number of moves of the knight can lead to the goal
- Dynamic: The coins can disappear

- Problem specification:

- Maximum branching factor (b): 8 since the knight can go to maximum 8 squares
- Maximum depth of state space (m): can be ∞ since the knight can jump back to previous square

- Problem formulation

- Initial state: the initial position of the knight and the initial positions of the coins
- Actions: the knight moves to a new position, the coin disappears if the knight moves to the square with the coin.
- Goal test: the destination
- Path cost: (additive) sum of all step cost
  - step cost = 3 if the next square has a coin

= -1 if the next square doesn't have anything

3. Selecting Algorithms

3.1. Choice of algorithms: Uniform cost search, A*, Backtrack

- Breadth first search is also expected to perform poorly because the step cost is different so it can not find an optimal solution. Moreover there exists negative step cost in the search tree, so the result is even poorer.

- Uniform cost search is chosen because the step costs are different so it is expected to find a better solution than breadth first search. However, uniform cost search also does not guarantee an optimal solution because there exists negative cost.

- A* is chosen because of similar reasons to uniform cost search that the step costs are different. Combined with a good heuristic function, A* is expected to find a solution faster than uniform cost search. However, designing a good heuristic function for this problem is very difficult.

- Backtracking is chosen because we will find all solutions to the problem and select the best one, so it guarantees an optimal solution. Compared to breadth first search and depth first search when finding all solutions, backtracking is more efficient.

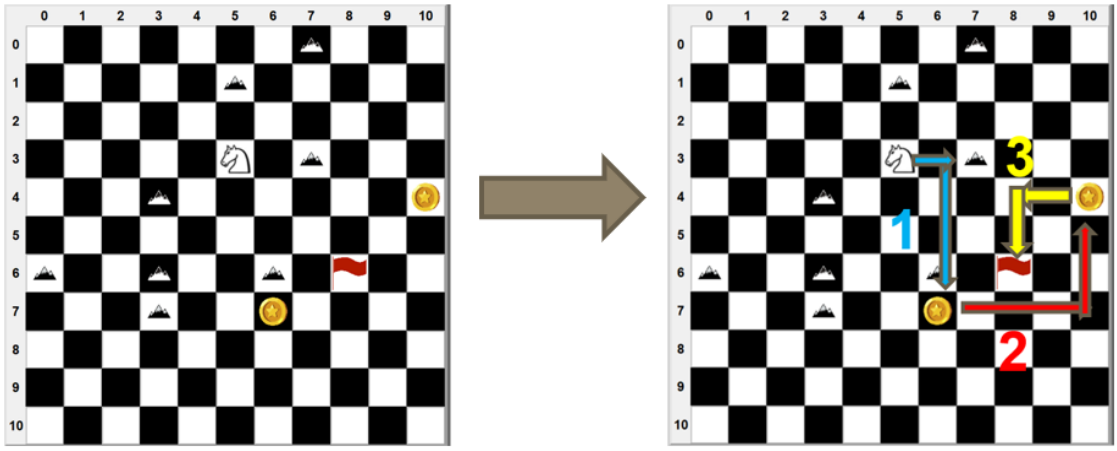
3.2. Apply algorithms to the problem

3.2.1. Uniform cost search:

Modifications to the regular UCS algorithms will be discussed in Section 4.

3.2.2. A*

- Idea: Assume the knight collects the nearest coin the board no longer has any coin then the knight goes to the destination. Estimate the cost using this path.



- For this problem, assume we grab the nearest coin first (with path1), and then grab the last coin with path 2.

- After there's no longer any point left on the board, we will jump to the destination.

-All estimations of the paths will use the manhattan distance divided by 3, rounded up plus the coin value (if we are jumping to a coin only), to estimate the cost reaching that path. (since the knight travels 3 squares per each move).

- So, the heuristic in this situation will be:

$h(n) = path\_1 + path\_2 + path\_3 = (-3+\lceil 5/3 \rceil) + (-3+\lceil 7/3 \rceil) + \lceil 4/3 \rceil = -1 + 0 + 2 = 1$

Advantage: Can reach higher cost since the knight tends to collect more coins.

Disadvantage: Sometimes it can take a very long time to find a solution and the solution is still not very good. Also it is not admissible and there exists negative cost so the A* algorithm, in theory can not find an optimal solution

3.2.3 Backtracking

- Forward checking: estimate the minimum cost possible of the current path by adding the value of coin * the number remaining coins. The algorithm will only continue to search if the estimated minimum cost does not exceed the minimum cost already found

- Ordering value: Order the next state to search in ascending order, with this method the algorithm will choose the smaller cost first thus will tend to cut the branch sooner in later iteration.

- Check feasibility: when using backtracking, if the problem does not have any solution, i.e the knight can not go to the destination because of obstacles, the algorithm can take a very long time to run, instead we use breadth first search to check the feasibility of the problem.

4. Implementing the algorithms to be used for solving the problem

4.1 Uniform Cost Search

Worst time and space complexity: O(N^2 x 2^C), where N is the size of the board, and C is the initial number of coins .

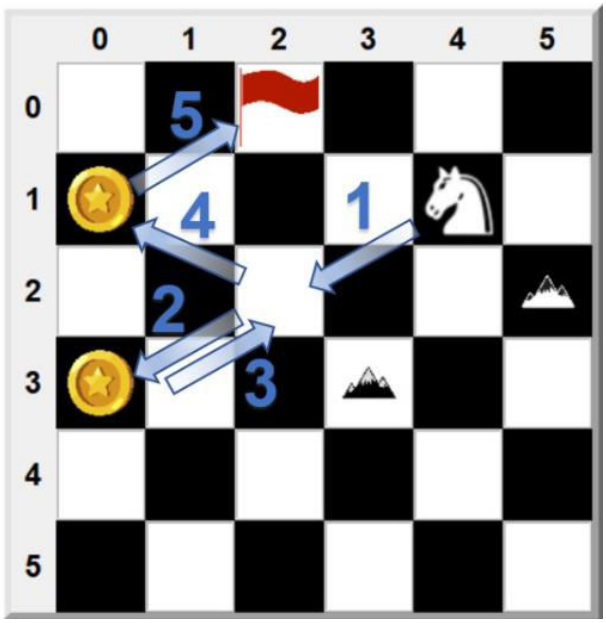
In this section, we will focus on difficulties encountered and present our solutions.

4.1.1 Negative weight edges:

The rule states as: jumping on coins gets +3 points, and empty squares gets -1 point.
Thus, negative weights are unavoidable. UCS will fail if there exists negative cycles.

- Solution: Consider the fact that coins will disappear after being 'taken'. If we let edges 'Coins' have negative costs (-3) and empty squares have positive costs (+1), the 'Coin squares' will change their cost to positive after being visited, which helps us avoid the negative cycles.
- Remark: by implementing the costs as above, the problem becomes finding the path with the lowest total cost, this also means that we can use the Min Priority Queue in the Python's built-in library Queue without having to modify it.

4.1.2 Looping paths:



Consider the board above, the optimal path contains a small loop: … (2,2) -> (3,0) -> (2,2) …

Thus, besides having to allow the Agent to return to its previous visited cell, it also seems that a state can occur more than once?

Actually no, please notice the first time the Knight visits (2,2) there is a Coin on (3,0); and when it returns to (2,2) there is no coin on (3,0). If we consider a state consisting of the board and all of its objects (positions of coins, knight), we can conclude that visiting each state at most once is enough to find the solution.

This also means that there is a finite number of states. So by closing each state after it is visited, we can make the number of possible nodes in the search tree become finite. Which means the algorithm can stop itself if there is no possible solution.

4.1.3 States distinction:

We cannot distinct states by Knight's position, and saving the whole board for each state costs a lot of space. Therefore, we need to find another way to distinguish between states.

- Idea:
  - Using Hashing
  - We can distinguish between states by the knight's position, and the positions of the remaining coins, because they are the non-static parts of the board.
- Solution:
  - Design a hash function as follow:

Hash(state) = string(Knight_x_coor + Knight_y_coor + Coin1_x_coor + Coin1_y_coor + … + CoinN_x_coor + CoinN_y_coor)

Then, we only have to use the keys to represent the states without having to save the whole state.

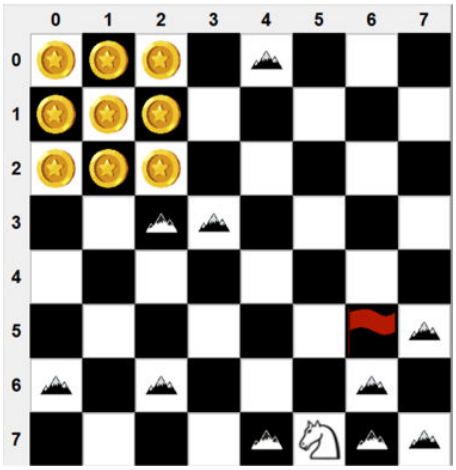
In the implementation, we will use the keys in 'explored' set and 'frontier_set'[1] and as the index of 'parent'[2] dictionary.

[1] *Will be discussed more in II.7*

[2] *Will be discussed more in II.5*

4.1.4 Goal cost

Since there are negative costs, the first solution to reach the goal found in UCS is not always optimal. That is, the agent can take a longer path, and grab more coins on its way.



Considering the board above, the optimal solution is that the Knight will 'grab' all the coins before getting back to the destination.

If we force the program to find more solutions, it will keep reaching for the goal node many times before it finds the optimal solution. Hence, we don't know how many more solutions it has to find.

- Idea: Make the Agent "wants" to explore more nodes in its vicinity before exploring the goal node. (Which means, make the goal node be dequeued later in the Priority Queue.)
- Solution: Assign higher cost to the goal node:

$$Goal\_cost = -L,$$

with: $L = ceil(N*2)$, where N is the size of the board.

The higher the value of L we set, the more time we have to trade off. By practical, we found out that this implementation helps us find the optimal solution for most of the problems while still relatively quick.

4.1.5 Retracking the solution's path:

- Idea: we will traverse from-leaf-to-root of the search tree that has been built while UCS was performing the search. The tree with nodes as states.

- Solution: for each node created, assign a 'pointer' to it's parent node. Then, when the goal node has been found, we use the Solution() function to traverse backward.

4.1.6 When to Enqueue a Child node?

We have a remark: due to negative costs, the shortest path to a node isn't always the one with the lowest cost. Consider the example below:

Let the first time state A is visited with the cost a, we then add state A to the 'explored' set. However, there is a chance that state A will be visited later by a different path, with a lower cost a'. In this case, we then have to update state A in the queue with the lower cost and the second path.

Thus, we add a new logical branch elif to the algorithm when considering to add a child node to the queue:

```
        elif state in explored and new_point < explored[state]:
    explored[state] = point
    queue.put((new_point, id(child), child))
    frontier[state] = new_point
```

4.1.7 Removing objects from Priority Queue:

In the UCS algorithm, the a new created Child.state is already in frontier, but with higher cost, then we will have to replace it with this new state. However, PriorityQueue using Heap structure doesn't allow us to remove a node in the middle.

Solution: we will keep a set called frontier_set. When a new state is enqueued, it will also be added to the set; when a state is dequeue, it will be removed, no matter if there is another similar state in the queue. So, when a similar Child.state is found with lower cost:

- Add the new Child.state along with the new cost to the queue, and keep the old Child.state.
- Since frontier_set already had Child.state stored, it will not be changed when adding a second Child.state.
- Since the new Child.state has a lower cost, it will be dequeued sooner, and the state will be removed from the set.

- When the old Child.state is dequeued, an error will occur when trying to remove the state from the frontier_set. By using Python's Exception Handling[3] , we will ignore the error, as well as ignore the node, and dequeue the next one.

⇨ By this implementation, we can disregard the old Child.state without having to delete them.

4.2 A* Search.

4.2.1. Similarities to UCS

- Negative weight edges: We also let edges 'Coins' have negative costs (-3) and empty squares have positive costs (+1) to avoid negative cycles.
- Retracking the solution's path: We will also travel from child nodes to parent nodes until we've reached the initial node,and store all the knight's position of these nodes in an array from the first node to the destination.

4.2.2. Goal cost

Since we want the heuristic to give a decent estimation of the cost from a point to the destination, we will not give the destination too big like in the UCS implementation so that we don't overestimate the heuristic.

Also, we want to give the destination a positive cost, since we tend to avoid arriving at the destination too early on with few coins collected (as demonstrated by the image on page 8). Thus, we choose the value (+3) for the goal cost.

4.2.3. Nodes limit and solutions list

Since all of our heuristics are not strictly admissible so the solution will not be optimal. Also, the state space is infinite, and the heuristic can be quite ineffective for larger cases, the program might iterate through hundreds of thousands of nodes and still find no solution.

Therefore, we will establish a very large node limit, in our program we set it to 400000 (which takes about from 30 seconds to more than 1 minute, depending on the problem).

[3] More about Python's Exception Handling:
https://www.w3schools.com/python/python_try_except.asp

When the nodes limit has been reached, we will stop the searching process, since we don't want the process to go on for too long and take too much memory for the computer to handle.

Here's the pseudo code to illustrate what we will do:

```
nodes = 1

   While Queue not empty:
      Node = Queue.Pop(0)
      nodes += 1
      if nodes = nodes_limit: break out and return None
      '' Explore child nodes here  ''
```

4.3 Backtracking

4.3.1. Store the path

We use a list of Node objects to store the path. Since we don't know how many it takes to arrive at the destination, using a list makes more sense than using a numpy array.

Each time we call Try() function, we will append the Node into the list and we will remove the Node from the list (list.pop() method) when we explored all its children (explored all its path to the destination).

Another list best_path is also used each time the algorithm finds a solution. It will be updated if the algorithm finds a better solution.

4.3.2 Check if state is already explored

We can use the "in" method of python but python will only compare the id of the object to return True or False, if we keep it this way python will always return True because each time we create a child node, we create a new instance of the class Node.

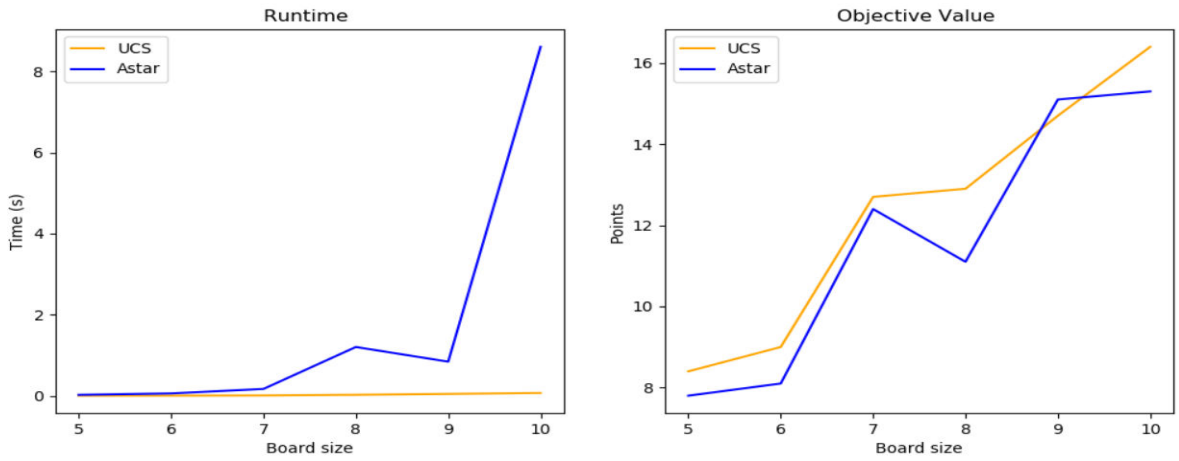
To fix this behavior, we define __eq__ method that returns True if and only if two Node objects have the same knight_pos and coin_pos_set attributes.

5. Comparing the results of the algorithms used for solving the problem

5.1. Uniform Cost Search and A*:

5.1.1. Testing data & results:

We tested the 2 algorithms on 120 randomly generated instances, 20 instances for each n *n board size, with n ranging from 5 to 10. On each board, there would be n coins and 2n obstacles. We obtain the results:



The average runtime of UCS is 0.022610s, with average objective value: 12.33

The average runtime of A* is 1.384533s, with average objective value: 11.63

5.1.2. Remarks & explanation.

UCS has a much smaller average run time for all board sizes tested.

As the size of board increases, the growth rate of UCS's runtime is also much smaller than A*'s.

Generally, UCS can find solutions with higher objective value.

Thus, UCS outperforms A* in the tested data.

Explanation:

These factors contribute to the unstability and inefficiency of the implementation of A* in this problem:

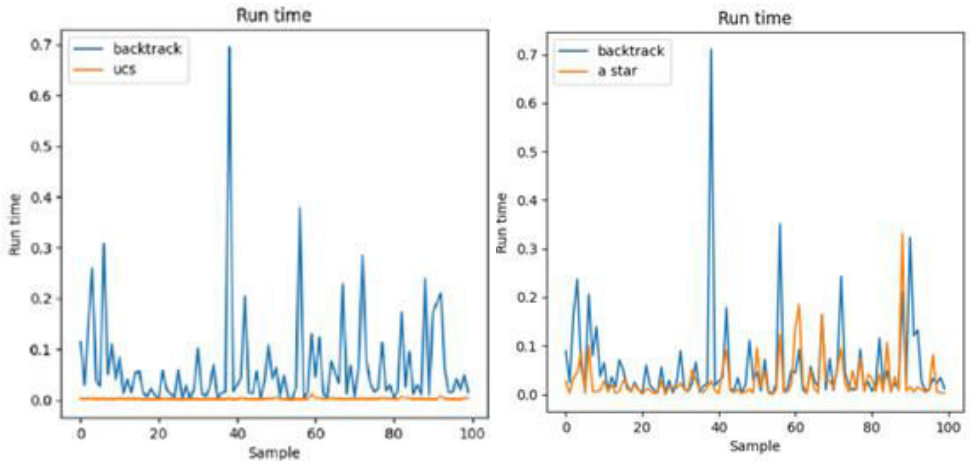
- Heuristic is not admissible and inconsistent.
- Unbalanced assignment of step costs.
- Repeatedness in states is enabled in A*'s algorithm.

5.2. Backtracking

5.2.1. Running time

Backtracking is very inefficient for large problem sizes. Because of this reason, we only compare backtracking to other algorithms for small instances of the problem, specifically, size of board = 5, number of coins = 5, number of obstacles = 7.

On average of 100 instances of the problem, backtracking takes about 0.07s to solve a problem while uniform cost search only takes about 0.002s, and A* about 0.06s, to find a solution.



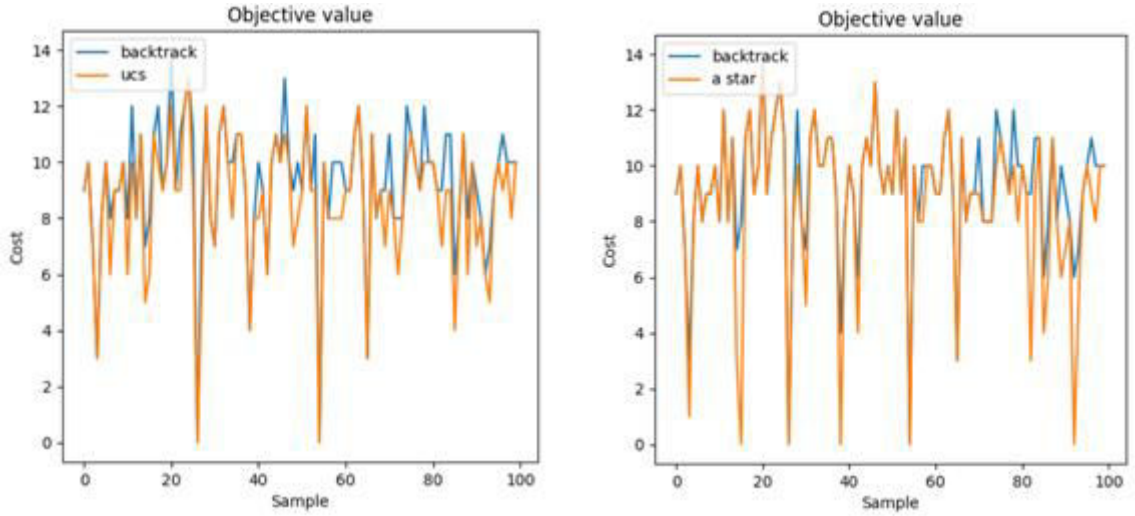
Moreover, backtracking varies greatly in terms of running time, some case can take up to 0.7s to find a solution, the standard deviation of this particular sample is 0.1s (even greater than the mean running time).

Meanwhile, the other 2 algorithms prove to be more consistent. A* search has a standard deviation of approximately 0.05s, only half as much as that of backtracking. And backtracking seems to be quite inferior to UCS in terms of

consistency, with UCS having a standard deviation of only about 0.01s, which is 10 times less than that of backtracking.

5.2.2. Optimality

Since backtracking guarantees optimality, it is expected for it to outperform the other two algorithms in this criteria.



However, there are many instances where we can also find the optimal solution using UCS or A*. Out of 100 tested instances, there are 74 where uniform cost search is able to find the optimal solution. A* also proves to be quite efficient when it comes to optimality, with 78 optimal instances where it is able to find the optimal solution out of 100 tested instances.

5.2.3. Explanation

The running time of backtracking can vary greatly because the algorithm depends on how early it can detect failures. If the algorithm can find a more optimal solution early, it can detect failures sooner in later iterations.

For small instances of the problem, backtracking algorithm is considered the best algorithm since it can guarantee an optimal solution in a reasonable time.

6. Conclusion and possible extensions

Upon solving this problem, there are several conclusions:

- For smaller instances (board size <= 5), Backtracking is the best algorithm despite its running time being significantly worse than the others, since optimality is guaranteed, and the time it takes to run the problem might be slower and less consistent, the running time is still quite reasonable.
- A* and UCS performs quite well with small instances, both with relatively fast and consistent running time. Even though both of the algorithms themselves are not optimal, they still manage to achieve optimal or near optimal solutions in a majority of instances.
- A* and UCS run faster than backtracking on larger instances of the problem with UCS find better solution than A* in shorter amount of time.
- A* lacks in consistency, and is generally inefficient in large instances(board size >= 10).When running these large instances, A* can take a lot of time to find a solution, or even no solution at all, while UCS outperforms it in terms of both optimality and running time.

Possible extensions:

- Improve the heuristic function for A* algorithm. For larger problem size, A* performs much worse than uniform cost search. This is highly due to the ineffectiveness of the heuristic function. Given time, we hope to find a better heuristic function to this problem.
- Another flaw of the A* algorithm is, unlike uniform cost search and backtracking, we have not restricted the repeated state when exploring the path, making the space complexity much worse. Thus, we hope to apply the same method used for uniform cost search to reduce the running time as well as the space.

7.List of tasks

7.1. Programming Tasks

- Task 1: Implement class Board and class Problem: Phan Duc Thang

- Task 2: Generate instances of the problem: Phan Duc Thang 50%, Nguyen Viet Hoang 50%.

- Task 2: Implement uniform cost search: Nguyen Viet Hoang.

- Task 3: Design and implement heuristic functions for A*: Phan Duc Thang 50%, Nguyen Tuan Dung 50%.

- Task 4: Implement A* algorithm: Nguyen Tuan Dung.

- Task 5: Implement Backtracking: Phan Duc Thang

- Task 6: Visual demonstrate module: Nguyen Viet Hoang

7.2. Analytic Tasks

- Subject Proposition: Phan Duc Thang

- Selecting Algorithms: Phan Duc Thang 50%, Nguyen Viet Hoang 50%

- Write part 1, 2 of the report: Phan Duc Thang

- Write part 3 of the report: Phan Duc Thang (3.1, 3.2.3), Nguyen Tuan Dung (3.2.2)

- Write part 4 of the report: Nguyen Viet Hoang (4.1), Nguyen Tuan Dung (4.2), Phan Duc Thang (4.3)

- Write part 5 of the report: Nguyen Viet Hoang (5.1), Phan Duc Thang (50% of 5.2), Nguyen Tuan Dung (50% of 5.2)

- Write part 6, 7 of the report: Phan Duc Thang

- Slides part 1, 4, 5.2: Phan Duc Thang

- Slides part 2, 5.1: Nguyen Viet Hoang

- Slides part 3: Nguyen Tuan Dung