MODIFIED KNIGHT'S PATH PROBLEM

Group 7

MEMBERS

Phan Đức Thắng
20194452

Nguyễn Việt Hoàng
20194434

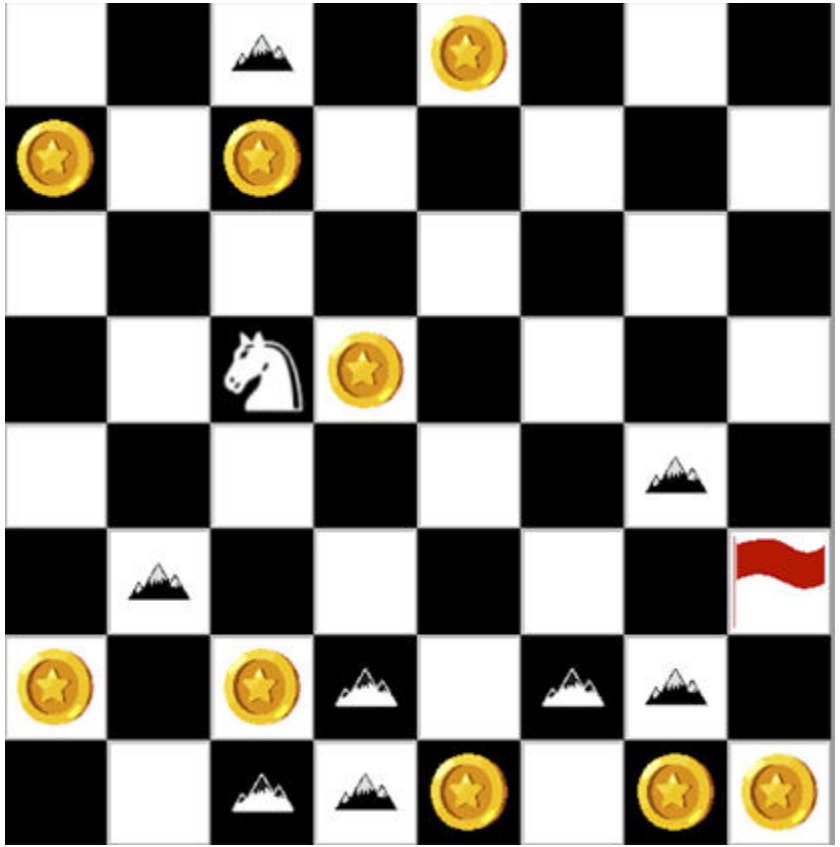
Nguyễn Tuấn Dũng
20194427

Subject Presentation 1

Summary

Knight need to go to the destination with max points

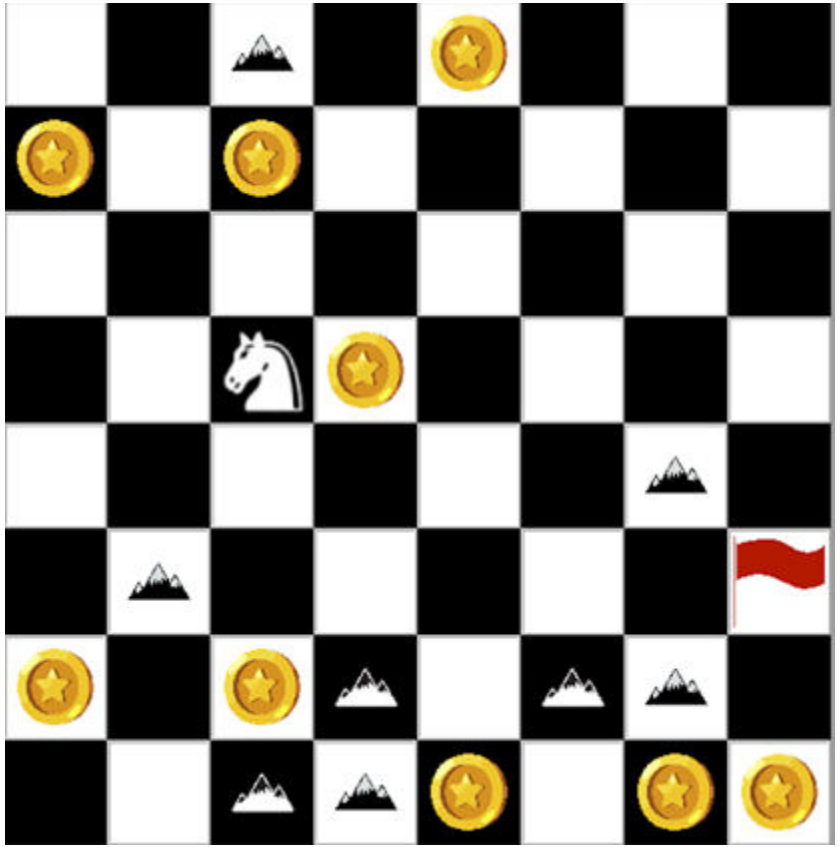
- Jumps to square with coin gain 3 points.
- Jumps to normal square, lose 1 point.
- Can not jump to square with obstacle



Subject Presentation 1

Problem Formulation

- Initial state: initial position of the knight and coins
- Actions: The knight jumps to new square, collect coin if possible
- Path cost: sum of all step cost
- Step cost = 3 if next square has a coin, = -1 if next square does not



UNIFORM COST SEARCH 2

2.1 Reasons

- Difficult to find efficient heuristics for this problem.
- Faster than BFS and DFS, and find better solutions.

2.2 Main differences

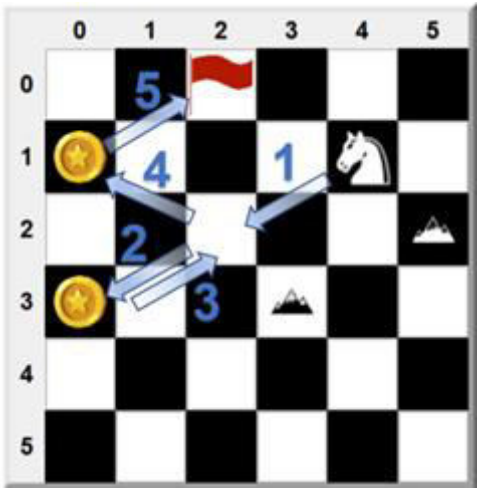
2.2.1 Looping paths & States distinction

- Looping paths must be enabled to find better solutions.
- Therefore, we have to distinguish states by both the Knight's position and the position of the remaining coins. (Non-static components.)

For easy accessing/ storing a state, we use hash function:

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

The hash key will be used to replace it's state in 'explored' set and 'frontier_set' set, and as the index of 'parent' dictionary, this helps us reduce the space needed.

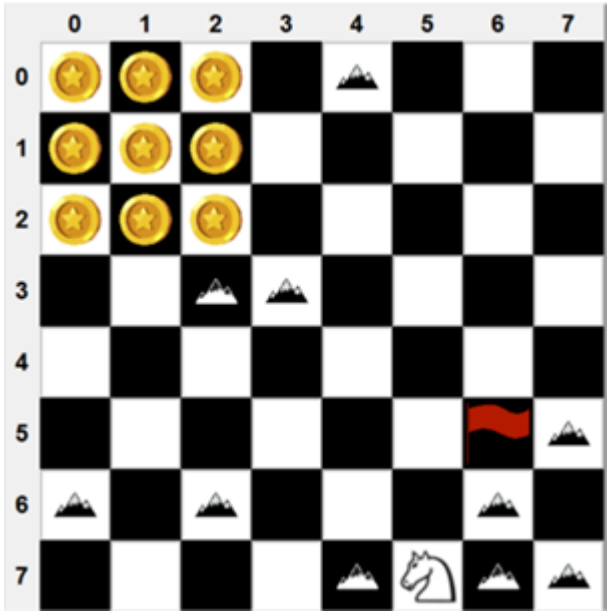


UNIFORM COST SEARCH 2

2.2 Main differences

2.2.2 Negative weight edges:

We let: **Cost = - Point** , so that the cost of coins will have negative values, which helps us avoid infinite negative cycles.



2.2.3 Goal cost

Since there are negative costs, the first solution to reach the goal found in UCS is not always optimal.

Solution: Assign cost to the goal node:

Goal_cost = - ceil (N x 2)

UNIFORM COST SEARCH 2

2.2 Main differences

2.2.4 Retracking the solution's path:

For each node created, we will assign a 'pointer' to it's parent node. Then, when the goal node has been found, we use the Solution() function to traverse the search tree from-leaf-to-root, and return the path.

```
def solution(node, point):
    path = []
    path.append(node.state.knight_pos)
    while node.parent != None:
        node = node.parent
        path.append(node.state.knight_pos)
    path = path[::-1]
    return path
```

2.2.5 When to Enqueue a Child node.

Due to negative costs, the shortest path to a node isn't always the one with the lowest cost
We add a new case where an explored child node can be added to the Queue:

```
elif state in 'explored' but with higher costs:
    replace the old state in the PriorityQueue with the new state
```

A* SEARCH 3

3.1 Reasons

- Coupled with an efficient and admissible heuristic , A* can be fast and (nearly) optimal.
- However, choosing such a heuristic that is admissible and efficient for all cases is very difficult.

3.2 Initialization

3.2.1 Negative weight edges

- The weight of edges is established similarly to UCS Search

3.2.2 Goal costs

- We need the heuristic to not overestimate too much
  - We establish Goal_cost = - Coin_Value

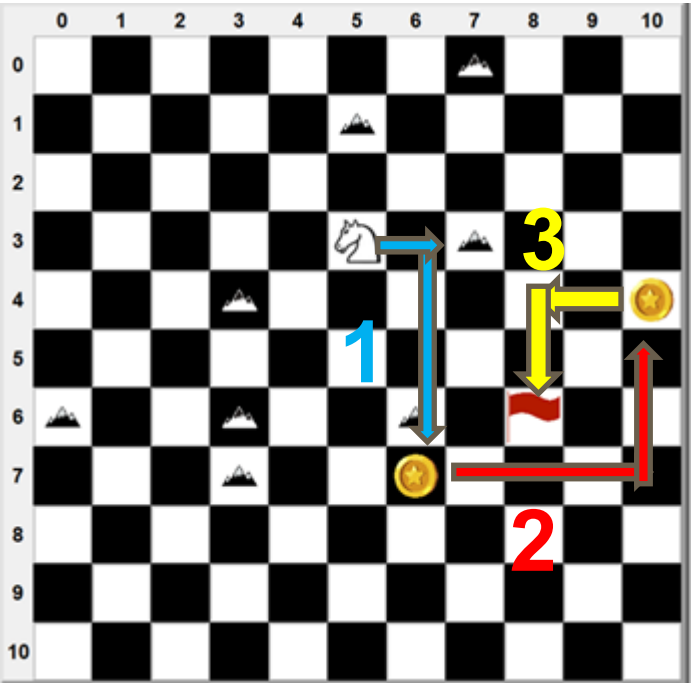
3.3 Heuristic function

Assume the knight collects the nearest coin the board one by one until there's no more coins left, then the knight goes to the destination
We estimate the cost:

- *Assume the coin jumps to its nearest coin, then we will estimate that distance, which is the Manhattan distance divided by 3, rounded up(since every time the knight moves, it travels 3 squares).*
- *We keep jumping from coins to the remaining nearest coins and adding to the heuristic costs until there's no more coins left on the board.*
- *Finally, when there's no more coins left, we jump to the destination(the estimated distance is calculated similarly).*
- *The sum of all these distances will be our heuristics.*

*Comment: This heuristic is **not** strictly admissible, so it will not be optimal.*

Visual Representation



- The manhattan distance from the knight to the nearest coin is 5, thus we have:

- Sub-path: from initial point to nearest coin: coin_value + $\left\lceil \frac{manhattan(n1)}{3} \right\rceil = -3 + \left\lceil \frac{5}{3} \right\rceil$ = -1

- Sub-path from coin 1 to coin 2: coin_value + $\left\lceil \frac{manhattan(n2)}{3} \right\rceil$ = $-3 + \left\lceil \frac{7}{3} \right\rceil = -3 + 3 = 0$

- Sub-path from the last coin to the destination : $\left\lceil \frac{manhattan(n3)}{3} \right\rceil = \left\lceil \frac{4}{3} \right\rceil = 2$

→ ***The heuristic of this node is: h(n) = -1 + 0 + 2 = 1***

3.4 Implementation

3.4.1 Nodes limit

- Since the searching space is infinite,there are cases where we can't consider all the nodes.
- Thus we store all the solutions we come across in a list.
- We set a really large nodes limit, and stop searching after the nodes limit has been reached.
- Finally, we return the best solution we found.

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

3.4.2. Retracking the solution's path

The same as UCS!

BACKTRACKING 4

4.1 Reasons

Guarantee optimal solution

More efficient in terms of space

Faster than BFS and DFS when find all solutions

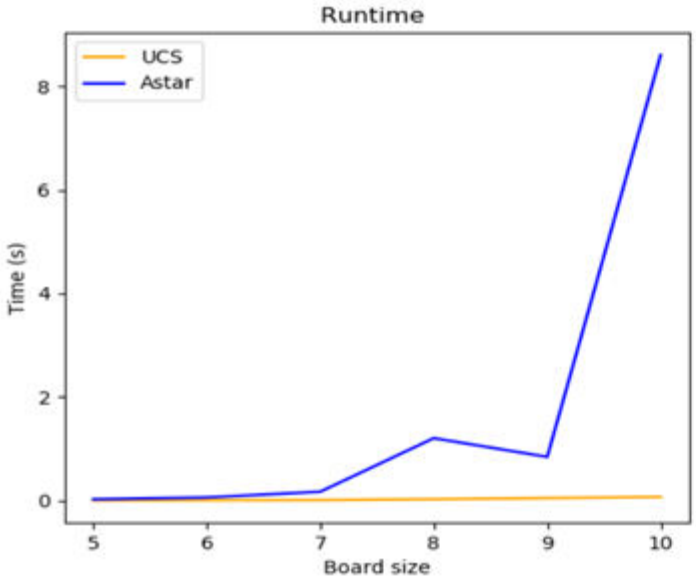
4.2 Apply to the problem

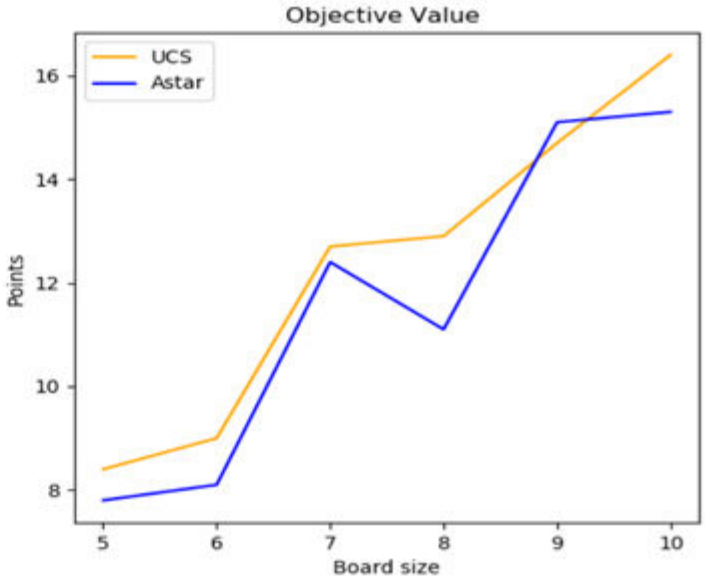
- Forward checking:
  estimated_min_cost = current_cost + coin_value * num_coins

- Ordering value: Select the step that has the smallest cost so that it is more likely to cut more branch in later iteration.

ANALYSIS 5

5.1 UCS vs A*





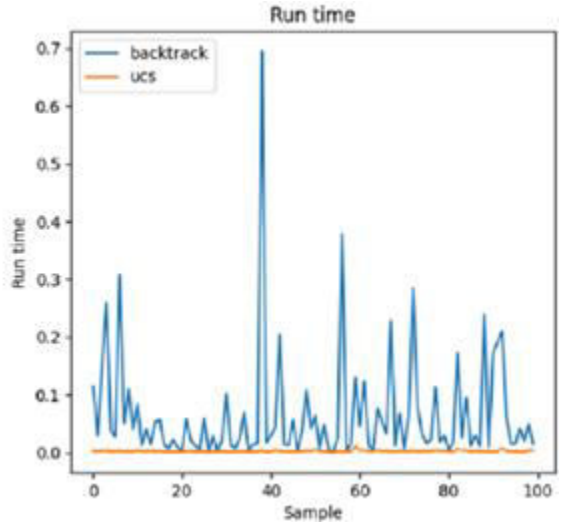
The average runtime of UCS is 0.022610s, with average objective value: 12.33
The average runtime of A* is 1.384533s, with average objective value: 11.63

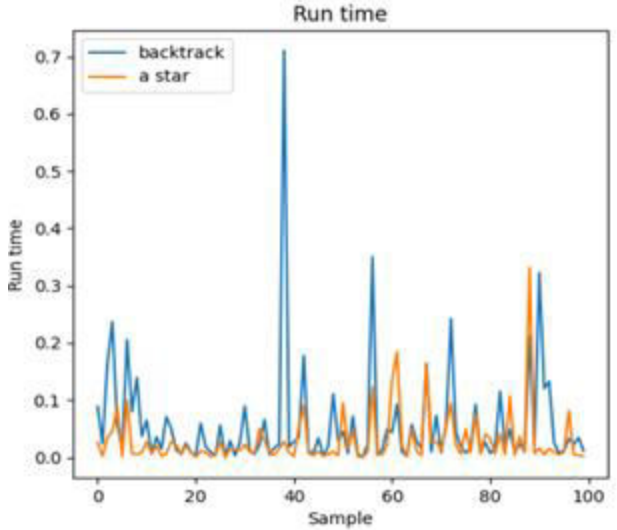
CONCLUSION: The implementation of UCS gives us better results in both running time and optimality critiques.

ANALYSIS 5

5.2 Backtracking vs UCS and A*

Board size = 5, num coins = 5, num obstacles = 7

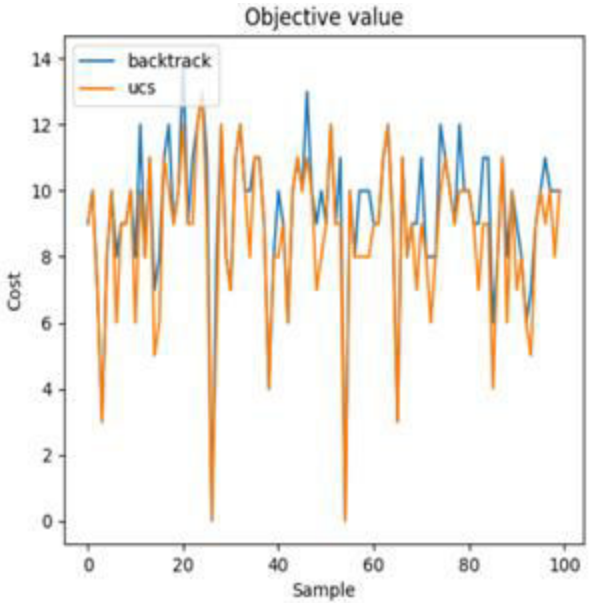


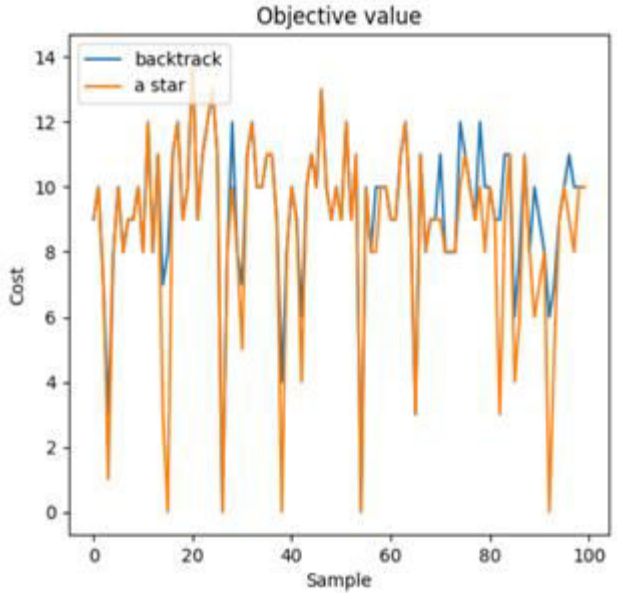


- The running time of backtracking on average is longest
- The running time of backtracking varies greatly

ANALYSIS 5

5.2 Backtracking vs UCS and A*





- Backtracking outperforms other two in terms of optimality

CONCLUSION: Backtracking is the best algorithm for small instances of the problem