Extended Wolf, Goat, Cabbage Problem

1. Presentation of the subject

In the original problem, there are three objects: a goat, a wolf, and a cabbage.

In the expanded problem, we add two more objects: a stick and a torch.

At the beginning, all five objects and the shepherd are on the first side. Each time the shepherd crosses the river, he can only bring up to two objects with him. And he needs to bring all the objects to the other side. But there are some constraints that he must follow:

1, Without the shepherd, if the wolf and the goat is together, the wolf will eat the goat

2, Without the shepherd, if the cabbage and the goat is together, the goat will eat the cabbage

3, Without the shepherd, if the wolf and the stick is together the stick will beat the wolf

4, Without the shepherd, if the torch and the stick is together, the torch will burn the stick

In this problem, we will analyze which algorithms are suitable to solve this problem and which ones are not.

The program will have several outputs:

- Time complexity (number of nodes expanded in order to solve the problem)
- Space complexity (number of nodes kept in memory)
- The sequence of actions that the shepherd must take in order to bring all objects/animals on the other side of the river or announce that no solution is found if the algorithm cannot find any possible solution.

2. Description of the problem

Type of Problem: This is a single-state search problem which means it is deterministic, fully observable, static and discrete. Every state can be observed fully and remains unchanged if no action is taken. The number of actions is also limited. Besides, given an action and current state, the next state of the problem is completely determined.

SEARCH PROBLEM FORMULATION

Initial State: All the objects on one side. We expand the problem that the initial state can be automatically generated in any state, but not a conflict one.

Action: Change only the shepherd or the shepherd with 1 or 2 objects to the other side

Transition Model: (state, action, new_state)

e.g: S((cabbage goat wolf stick torch shepherd), ( )) = {<(shepherd goat stick: A → B), ((cabbage wolf torch), (goat stick shepherd))>,......}

Goal Test: All the objects are moved to the other side

Path Cost: Each step costs 1 and the cost is uniform.

3. Algorithms

3.1. Uninformed Search

3.1.1. Uniform-cost Search

The Uniform-cost Search Algorithm is not suitable for this problem because all of the step costs are the same which makes the uniform-cost search in this problem be similar to the breadth-first search

3.1.2. Depth First Search

If we use DFS, there will be a problem that we must deal with.
In the worst case, the algorithms will expand the state in the infinite branch. Therefore, there is no way to find out the solution, the algorithm can't complete.
If we store the nodes already expanded, the algorithm can find a way to the goal, but will not have the optimal solution => we don't choose this algorithm

3.1.3. Depth Limited Search

Depth-limited Search = Depth first search with a depth-limit
The problem with Depth-limited Search is that if the depth of the goal node is beyond the depth-limit , the solution is not found
It is very hard to predict exactly the depth of the goal state so we can't choose the depth limit reasonably.
The algorithm is not complete. => we don't choose this algorithm

3.1.4. Iterative Deepening Search

The IDS is the DLS which is iterative with a depth-limit from 1 to the depth of the goal state. This algorithm fixes the drawback of Depth-limited search and it's not necessary to predict the depth of the goal state => We choose this algorithm

3.1.5. Breadth First Search

This algorithm is complete (we always find the goal state) => We choose this algorithm

3.2. Informed and advanced search strategies

3.2.1. Simplified Memory - bounded A* Search (SMA*)

The Simplified Memory-bounded A* is not the good choice for this problem as the step costs are uniform => we don’t choose this algorithm

3.2.2. A* Search

A* search makes the best use of all the available information => we choose this algorithm

3.2.3. Iterative Deepening A* Search (IDA*)

It is a variant of Iterative Deepening Search(IDS) that borrows the idea of A*. IDA* is complete and optimal under the same conditions as A* search => we choose this algorithm

4. Implementing the algorithms used for solving the problem

4.1. Breadth-First Search

In this algorithm, we will use the fringe as a queue. We will expand the first node of the queue and every new node will be put at the back of the queue. We also notice that every state of the problem is fully observable, therefore, we could save all the visited nodes in memory. This could be implemented as:

```
if new node is visited:
    continue
else:
    set new node as visited
    ...
```

4.2. Iterative Deepening Search

To implement IDS, we will iterate over the depth limit. For each depth limit, we will use depth first search to check whether we could reach the goal given that every path length could not exceed the limit. If the goal is reached, we will return the depth limit as the solution. The DFS algorithm could be implemented using a stack as the fringe. But in our program, we will use a recursive function.

4.3. A* Search & IDA* Search

The implementation of A* will be mostly the same as BFS. The main difference is that in A* we will use the priority queue as the fringe. And the node with lower priority will be expanded first because we want to minimize the sum of current cost and heuristic function in this problem.

The implementation of IDA* will also be the same as IDS but now we will not iterate over the depth limit, we will use the heuristic functions of the leaf nodes as the limit in each iteration instead.

4.4. About A* Search & IDA* Search heuristic function

- This is the main difficulty we encounter when implementing A* and IDA*.
- How we find the heuristic:

In this problem, we use the heuristic function h(n) which estimates the minimum cost from current node n to the goal, ignoring all conflicting states.

When we consider the case where there are no conflicts, the Shepherd can directly bring up to 2 objects from the initial river side to the goal side. So for every node n, the value of the heuristic function h(n) will be obviously less than or equal to the true cost. Thus, the heuristic is admissible:

$$\forall\ n,\ 0 \le h(n) \le h^*(n)$$

Pseudocode:

```
  if 'Shepherd' is in side A:
          return ceil((number of objects in side A - 1) / 2) * 2 - 1
else:
          return ceil((number of objects in side A) / 2) * 2
```

(The heuristic function returns the minimum number of steps to bring all objects in A to side B without mentioning the conflicts)

The following table shows the h(n) of each valid state:

Cabbage Goat Sheperd Stick Torch Wolf		5
Cabbage Goat Sheperd Stick Torch	Wolf	3
Cabbage Sheperd Stick Torch Wolf	Goat	3
Goat Sheperd Stick Torch Wolf	Cabbage	3
Goat Sheperd Stick Torch	Wolf Cabbage	3
Cabbage Goat Sheperd Torch Wolf	Stick	3
Cabbage Sheperd Torch Wolf	Goat Stick	3
Cabbage Torch Wolf	Sheperd Goat Stick	4
Torch	Sheperd Wolf Goat Cabbage Stick	2
Cabbage Goat Sheperd Stick Wolf	Torch	3
Cabbage Goat Sheperd Stick	Wolf Torch	3
Cabbage Sheperd Stick Wolf	Goat Torch	3
Cabbage Stick	Sheperd Wolf Goat Torch	2
Goat Sheperd Stick Wolf	Cabbage Torch	3
Goat Sheperd Stick	Wolf Cabbage Torch	1
Goat Stick	Sheperd Wolf Cabbage Torch	2

Cabbage Torch	2
Sheperd Wolf Goat Stick	

Goat Sheperd Torch Wolf	3
Cabbage Stick	

Goat Torch	2
Sheperd Wolf Cabbage Stick	

Torch Wolf	2
Sheperd Goat Cabbage Stick	

Wolf	2
Sheperd Goat Cabbage Stick Torch	

Stick	2
Sheperd Wolf Goat Cabbage Torch	

Cabbage Wolf	2
Sheperd Goat Stick Torch	

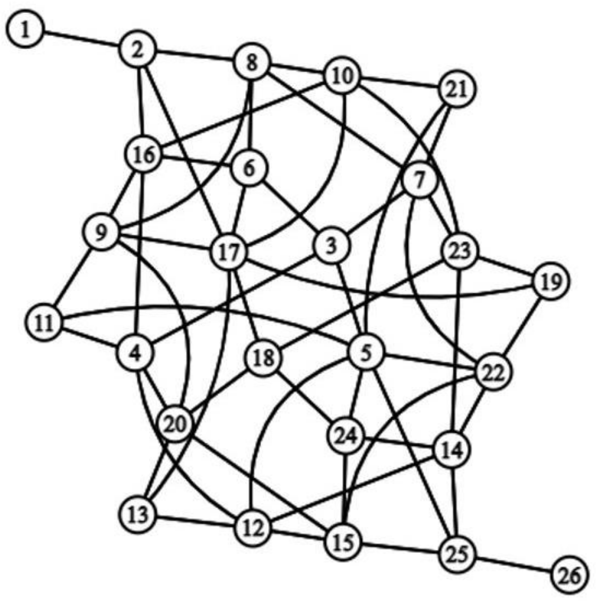
Cabbage	2
Sheperd Wolf Goat Stick Torch	

Goat	2
Sheperd Wolf Cabbage Stick Torch	

	0
Sheperd Wolf Goat Cabbage Stick Torch	

In the previous section, we already knew that the BFS tree search could be improved by saving all the visited nodes but this happens because it is an optimal search algorithm. But this is not the case for A* search. For A* Graph search to be optimal, we need to prove that the heuristic function is consistent.

('Cabbage', 'Goat', 'Shepherd', 'Stick', 'Torch', 'Wolf')	1
('Cabbage', 'Torch', 'Wolf')	2
('Cabbage', 'Goat', 'Shepherd', 'Stick', 'Torch')	3
('Goat', 'Torch')	4
('Goat', 'Stick')	5
('Cabbage', 'Torch')	6
('Cabbage', 'Stick')	7
('Cabbage', 'Shepherd', 'Stick', 'Torch', 'Wolf')	8
('Torch', 'Wolf')	9
('Cabbage', 'Wolf')	10
('Goat', 'Shepherd', 'Stick', 'Torch', 'Wolf')	11
('Goat', 'Shepherd', 'Stick', 'Torch')	12
('Torch')	13
('Stick')	14
('Goat')	15
('Cabbage', 'Goat', 'Shepherd', 'Torch', 'Wolf')	16
('Cabbage', 'Shepherd', 'Torch', 'Wolf')	17
('Wolf')	18
('Cabbage')	19
('Goat', 'Shepherd', 'Torch', 'Wolf')	20
('Cabbage', 'Goat', 'Shepherd', 'Stick', 'Wolf')	21
('Cabbage', 'Goat', 'Shepherd', 'Stick')	22
('Cabbage', 'Shepherd', 'Stick', 'Wolf')	23
('Goat', 'Shepherd', 'Stick', 'Wolf')	24
('Goat', 'Shepherd', 'Stick')	25
()	26



T/N: We display the objects only on beginning side

According to the previous table of heuristics and the graph of states, this heuristics satisfied the consistency.

Indeed, two arbitrary adjacent nodes have heuristic function less (more) than each other exactly 1, and the cost between them is uniformly 1. Therefore, the triangle inequality: h(n) ≤ c(n, a, n’) + h(n’) is correct for two arbitrary adjacent nodes. Also, h(goal) = 0

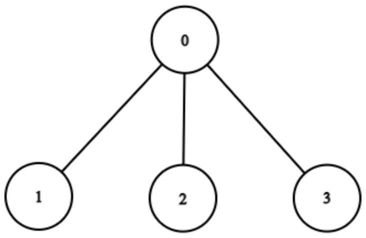
5. Comparing and Explaining the results

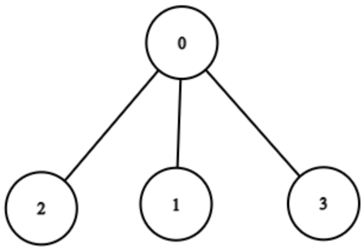
5.1. Quantitative performance indicators

We calculate time complexity by the number of nodes generated and calculate space complexity by the maximum number of nodes in the memory

In this problem, we only have 26 valid states that can be used as the initial states. Because the way the program searches for the solution is different and dependent on the order of the next states, then in order to create more problem instances, the order of the set of the next nodes of the current nodes which is being expanded is shuffled randomly.

For example, the way the algorithm searches in the set (1,2,3) which is led from the node 0, is different from the way it searches in the set (2,1,3), which is also led from the node 0.





By this way, we can create more than 50 test cases for the problem.

The following table shows the quantities of each algorithm:

Algorithm	BFS tree	BFS	IDS	A* tree	A*	IDA*
Max time	928	26	4643	272	20	1186
Min time	1	1	1	1	1	1
Avg time	134.7	21.96	672.72	29.88	8.36	216.38
Max space	2440	39	83	712	38	83
Min space	1	2	1	1	2	1
Avg space	355.34	33.58	43.36	78.6	29.58	41.68

Number of tests	50

BFS tree: BFS with reopening closed nodes

BFS = BFS(Graph): BFS without reopening closed nodes

A tree: A* Tree Search with reopening nodes*
A : A* Graph Search without reopening closed nodes*
IDA: IDA* with reopening closed nodes*

5.2. Explaining the results

5.2.1. Time complexity

- First, you can see that BFS(Graph) and BFS(Tree) have much smaller time complexities than the IDS, because the IDS has to run all over again after exploring each depth limit.

- Both A* Graph Search and A* Tree Search have smaller time complexity than IDA* Since IDA* runs from the root again after each iteration .

- A* Graph Search has the smaller average time complexity than A* Tree Search, which is (8.36 compared to 29.8) . The main reason for this is in graph-search we don't reopen closed nodes, which makes the algorithm less time-consuming.

- Finally , A* Graph Search is better than BFS(Graph) in terms of time complexity for the reason that A* Graph Search has a strategy to search based on the heuristics and it will expand nodes, which could potentially lead to the optimal solution.

5.2.2. Space complexity

- The BFS (Tree) has higher space complexity than IDS while BFS(Graph) has smaller space complexity than that of both BFS(Tree) and IDS. BFS(Tree) expands all the nodes in the trees in the worst case. Meanwhile, IDS trades off time to keep lower space complexity; IDS in some cases doesn't need to expand as many nodes as BFS(Tree) and with each new depth-limit, IDS removes memory of the old depth-limit. BFS(Graph) doesn't reopen nodes like BFS(Tree).

- A* Tree Search has higher space complexity than IDS because uninformed search finds and expands nodes that may not be the optimal solution and in the worst case, uninformed search will expand all the nodes in the tree.

- Because IDA* is a depth-first search algorithm and Memory-bounded heuristic search which is used to address the space cost problem of the A* Tree Search, so the space complexity of IDA* is much smaller than the A* Tree Search algorithm.

- A* Graph Search has the smallest average space complexity because this strategy search uses heuristics and does not reopen closed nodes. A* Graph Search has smaller space complexity than A* Tree Search with the same reason as BFS Graph Search and BFS Tree Search

6. Conclusion and possible extensions

Conclusion:

- In general , the A* Graph Search without reopening closed nodes is the most suitable strategy search for the problem in both time and space complexity aspects

Possible Extensions :

Each object has a different weight so The cost of the shepherd carrying objects is also different .

For example, we may assume that

→ Transporting the shepherd only, or both the shepherd and the cabbage costs 1

→ Transporting the wolf and shepherd costs 3 etc…

↳ the main difficulties of the extensions : We must find a more advanced heuristic function

7. References

1. Stuart J. Russell and Peter Norvig, Artificial Intelligence : A Modern Approach ,Third Edition

2. Muriel Visani, Slides of Introduction to Artificial Intelligence Course