T.C.

MARMARA UNIVERSITY

FACULTY of ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE4082 - Artificial Intelligence

Homework #1 Report

Group Members

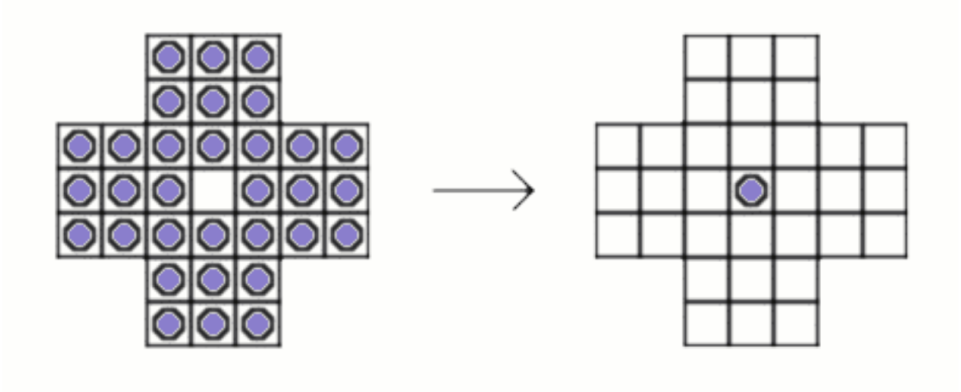
150119692 - Bedirhan SARIHAN

170517033 - Yasin Alper BİNGÜL

PEG SOLITAIRE GAME

Peg solitaire is a single-player board game that involves moving pegs around a board with the goal of leaving only one peg on the board at the end of the game. The game is typically played on a board with 33 holes arranged in a cross shape, with 32 pegs that can be moved around the holes.

There are several variations of peg solitaire, with different board shapes and different rules for making moves. Some variations allow the player to move pegs diagonally, while others require the player to jump over a peg and land on a specific spot on the board. Despite these variations, the basic goal of the game remains the same: to eliminate all but one peg from the board. In this project, we simply designed a peg solitaire like down below.



Algorithms

Our code consists of three distinct sections. The first part controls the program, the second part creates clones, and the third part utilizes the search algorithm we developed. We defined the board as a 7x7 matrix and wrote the code based on this board variable. If a particular index in the matrix has a value of 1, it indicates the presence of a peg at that index. If the value is 2, it means the index is not a valid part of the board map. If the value is 0, the index is empty and available for a move.

In the control section, we examine the board matrix to determine if a particular index contains the value 1. If a 1 is found, we move on to the second control section, which checks the neighboring indices. If a suitable destination is found, it is added to a list. If not, the program continues to search for potential moves.

In the section on creating clones, we simply evaluate the number of possible movements and create clones based on that count. When creating the clones, the board matrix will be updated with the movement information.

In the last section, we create different search algorithms that search in different ways such as Breadth-First Search, Depth-First Search, and some derived Heuristic searches.

1. **Breadth-First Search**

Breadth-first search (BFS) is an algorithm that traverses tree structure by exploring all the nodes at the current depth level before moving on to the nodes at the next depth level. We implement BFS is to use a queue as the frontier list, which stores the nodes that are waiting to be explored.

Here is an outline of the steps involved in a BFS algorithm using a queue as the frontier list:

- Initialize the queue with the starting node.
- While the queue is not empty:
    - Remove the first node from the queue and explore it.
    - Add all the unvisited neighbors of the current node to the queue and mark them as visited.
    - Repeat this process until the queue is empty.

After searching one hour, we got the results below:

Explored count	Frontier count	Max frontier count	**Best Solution**
951771	8048231	8048232	**22**

2. **Depth-First Search**

Depth-first search (DFS) is an algorithm that traverses a tree structure by exploring as far as possible along each branch before **backtracking**. We again implement DFS is to use a stack as the frontier list, which stores the nodes that are waiting to be explored. Here is an outline of the steps involved in a DFS algorithm using a stack as the frontier list:

- Initialize the stack with the starting node.
- While the stack is not empty:
    - Remove the top node from the stack and explore it.
    - Add all the unvisited neighbors of the current node to the stack and mark them as visited.
    - Repeat this process until the stack is empty.

After searching one hour, we got the results below:

Explored count	Frontier count	Max frontier count	**Best Solution**
31788558	50	51	**7**

3. Iterative Deepening Search

Iterative deepening depth-first search (IDDFS) is a *variant of depth-first search (DFS)* that repeatedly performs DFS to increasing depth limits until the goal is found. IDDFS can be implemented using a queue as the frontier list, which stores the nodes that are waiting to be explored.

Here is an outline of the steps involved in an IDDFS algorithm using a queue as the frontier list:

- Initialize a depth limit to 1 and a queue with the starting node.
- While the goal has not been found:
    - Perform a DFS search with the current depth limit, using the queue as the frontier list and marking visited nodes.
        - If the goal is found, return the path and explore it.
        - Increase the depth limit and clear the visited nodes.
        - Repeat this process until the goal is found or the depth limit exceeds the maximum depth of the tree.

IDDFS is useful when the depth of the goal node is not known in advance, as it allows the search to gradually increase the depth limit until the goal is found. However, it may be less efficient than other search algorithms, such as breadth-first search (BFS), in cases where the depth of the goal is known and the tree is shallow.

After searching one hour, we got the results below:

Limit	Explored count	Frontier count	Max frontier count	**Best Solution**
1		1	2	**32**
2		2	3	**31**
3		3	4	**29**
4		4	5	**28**
5		5	6	**26**
6		6	7	**25**
7		7	8	**23**
8		8	9	**22**

4. **Depth-First Search with Random Selection**

Depth-first search with random selection (DFSRS) is a variation of the depth-first search (DFS) algorithm, a tree traversal algorithm that explores as far as possible along each branch before backtracking.

In DFSRS, instead of following a predetermined order when adding nodes to the frontier list (the list of nodes that have been visited but not yet fully explored), the next node to be explored is chosen randomly from the frontier list. This random

selection helps to avoid getting stuck in a locally optimal solution and allows the algorithm to explore more of the search space.

Here is an example of how DFSRS works:

- Start at a given node, called the root node.
- Mark the root node as visited and add it to the frontier list.
- Choose a random node from the frontier list and remove it from the list. This is the current node.
- If the current node has any unvisited neighbors, choose one at random and mark it as visited. Add it to the frontier list.
- Repeat step 3 until the frontier list is empty, then backtrack to the last node that has unvisited neighbors and repeat the process.
- Continue this process until all nodes in the tree have been visited.

DFSRS can be useful in situations where the goal is to find any solution to a problem, rather than the optimal solution. It can also be useful when the search space is large and it is not feasible to explore all possible solutions.

After searching one hour, we got the results below:

Explored count	Frontier count	Max frontier count	**Best Solution**
142442544	35	36	**6**

5. **Depth-First Search with a Node Selection Heuristic**

Depth-First Search (DFS) is a traversal algorithm that starts at the root node of a tree and explores as far as possible along each branch before backtracking. One way to implement DFS is to use a frontier list to keep track of the nodes that are yet to be explored. The frontier list can be implemented using a stack or a queue.

A heuristic is a function that estimates the distance from a given node to the goal. When using a heuristic in DFS, we can use it to guide the search by selecting the most promising node to explore next. This can be done by using the heuristic to rank the nodes in the frontier list and selecting the node with the lowest heuristic value.

We have identified a heuristic method that can be used in DFS to solve a problem. The method involves taking nodes from the frontier list (which is a list of nodes that have yet to be explored) and checking if they are a solution. To do this, we start by selecting the first and last nodes in the frontier list. Then, we iteratively choose the first and last nodes and try to reduce the number of pegs as much as possible. If a

node is a solution, we stop the search. Otherwise, we continue the search by exploring the remaining nodes in the frontier list.

After searching one hour, we got the results below:

Explored count	Frontier count	Max frontier count	**Best Solution**
47855938	50	51	**7**

Conclusion

Search Algorithm	Explored count	Frontier count	Max frontier count	**Best Solution**
Breadth-First Search	951771	8048231	8048232	**22**
Depth-First Search	31788558	50	51	**7**
Iterative Deepening Search		8	9	**22**
Depth-First Search with Random Selection	142442544	35	36	**6**
Depth-First Search with a Node Selection Heuristic	47855938	50	51	**7**

Breadth-First Search (BFS), Depth-First Search (DFS), Iterative Deepening Search (IDS), Depth-First Search with Random Selection (DFS-RS), and Depth-First Search with a Node Selection Heuristic (DFS-NSH) are all tree traversal algorithms that are used to explore the nodes of a trees.

BFS is a complete and optimal search algorithm that explores the nodes of a trees in a breadth-first manner. It guarantees to find the shortest path between the start and goal nodes, but it has a high memory requirement and may not be suitable for large trees.

DFS is an incomplete and non-optimal search algorithm that explores the nodes of a trees in a depth-first manner. It may not find the shortest path between the start and goal nodes, but it has a low memory requirement and is suitable for large trees.

IDS is a complete and optimal search algorithm that combines the depth-first search strategy of DFS with the iterative deepening technique.

DFS-RS is a variant of DFS that randomly selects the next node to explore from the frontier list. It has a low time complexity and may find a solution faster than DFS, but it may not find the optimal solution. But we actually did, we found the best solution with Depth-First Search with Random Selection.

DFS-NSH is a variant of DFS that uses a heuristic function to guide the search by selecting the most promising node to explore next. It may find a solution faster than DFS, but it may not find the optimal solution. It depends on the specified heuristic. So in that case we couldn't specified heuristic that behaves better than DFS-RS.

In conclusion, each of these search algorithms has its own strengths and weaknesses, and the best algorithm to use will depend on the specific problem and the resources (such as time and memory) available. At the end of 1 hour the best solution that we found was **Depth-First Search with Random Selection.**