BLG 435-E Project Report

Project No	Project 1
Lecturer’s Name	Sanem Sarıel
Student’s Name	Alperen KANTARCI – 150140140 – Computer Engineering
Delivery Date	09.11.2018

1. Question 1 and 2

Q1-) I choose domestic service robot.

Performance measure: Successfully complete domestic services for the people such as washing dishes, empty trash, bring items, do not spoil foods or drinks while bringing, tasks should be finished in reasonable time (washing dishes shouldn't take 10 hours)

Environment: Partially observable, Stochastic, Sequential, Dynamic, Continous and multi agents

Sensors: Microphones, video camera, depth sensor.

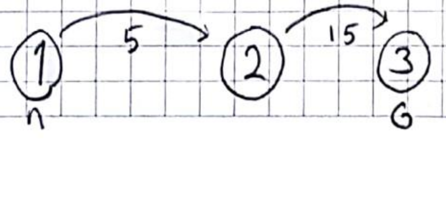
Actuators: Grasping arms, moving mechanism, height changer body (for crouching or reaching items at the top shelves)

I select goal based agent model for this agent. Because there will be variable goals such as clean clothes, bring food, wash dishes. It should form a plan to achieve that goal. Also state of the world and it's changes is important for this agent.

2-) Heuristic is admissable if the estimated cost is never more than the actual cost from the current node to the goal node.

Consistent heuristic is a heuristic that has this cost property: If estimated cost from current state to goal is always less than or equal to real cost from current state to one of it's following state and the estimated cost from that following state to the goal.

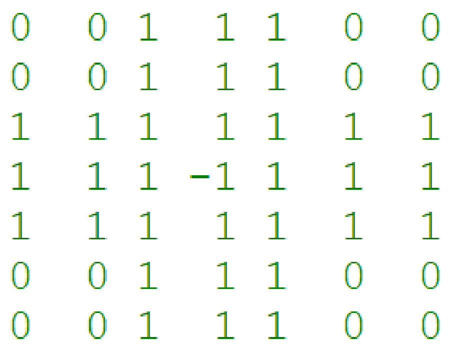
The graph at the below shows that inconsistent heuristic. Consider heuristic, h(n) = minimum cost from successor of n to one of it's own successor, If successor is goal node than it's 0 so it's excluded. $h(1) = 15$, but $h(2) = 0$ so $h(1) = 15 > 5 = c(1,2) + h(2)$ so it violates consistency. But it's still admissable because $h(1) = 15 < 20 = h^*(1)$ and $h(2) = 0 \leq 15 = h^*(2)$



2. Question 3

2. a)

Each state has 7x7 board, for each element in this grid. Spaces are represented as -1 , pegs are 1 and outside of the board is 0 in this representation. Each peg can have action if following conditions holds. If a peg and two location ahead of the peg is empty, also if there is another peg between empty location and the peg then this peg can move and remove the peg that jumpeg over. Actions are represented as this (i,j,Direction) . In this representation i and j are the location of the peg where i is row indexed with 0 and j is the column with index 0. Direction can be one of these Direction = {UP,DOWN,RIGHT,LEFT}. You can see the initial state in the Figure 1.



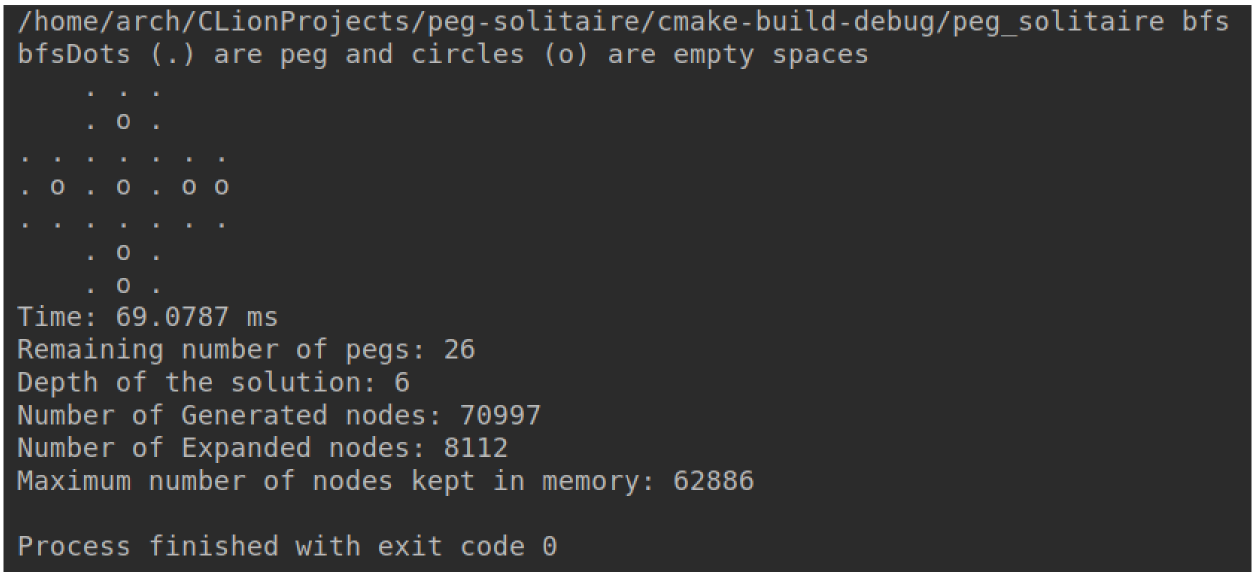
```
0  0 1  1 1  0  0
0  0 1  1 1  0  0
1  1 1  1 1  1  1
1  1 1 -1 1  1  1
1  1 1  1 1  1  1
0  0 1  1 1  0  0
0  0 1  1 1  0  0
```

Figure 1: Initial state

Goal state is the state that none of the pegs have any move which means that possible number of action is 0. Path cost is the uniform for each movement in the board. So every peg movement costs same.

2. b)

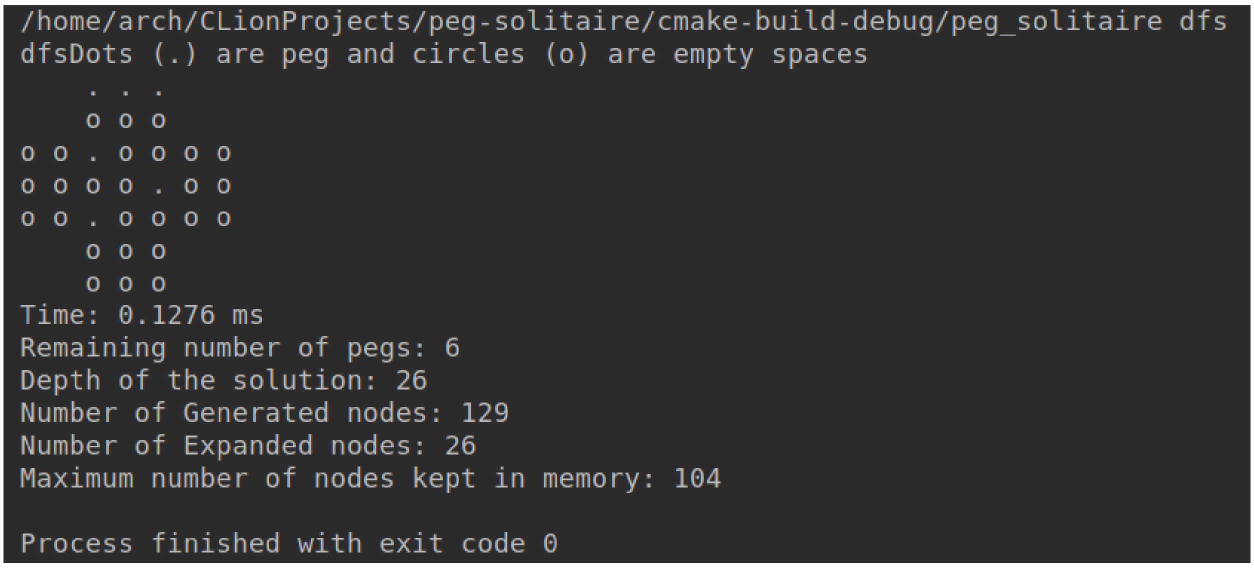
In Figure 2 and 3 you can see the BFS and DFS tree search results for the problem. In the implementation of these search algorithms I have used https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/ for BFS but modified it for the tree search and adopted it for our problem. I have used https://www.geeksforgeeks.org/iterative-depth-first-traversal/ for the DFS and also I changed it to tree search and adopted it for our problem.



```
/home/arch/CLionProjects/peg-solitaire/cmake-build-debug/peg_solitaire bfs
bfsDots (.) are peg and circles (o) are empty spaces
    . . .
    . o .
. . . . . . .
. o . o . o o
. . . . . . .
    . o .
    . o .
Time: 69.0787 ms
Remaining number of pegs: 26
Depth of the solution: 6
Number of Generated nodes: 70997
Number of Expanded nodes: 8112
Maximum number of nodes kept in memory: 62886

Process finished with exit code 0
```

Figure 2: BFS result



```
/home/arch/CLionProjects/peg-solitaire/cmake-build-debug/peg_solitaire dfs
dfsDots (.) are peg and circles (o) are empty spaces
    . . .
    o o o
o o . o o o o
o o o o . o o
o o . o o o o
    o o o
    o o o
Time: 0.1276 ms
Remaining number of pegs: 6
Depth of the solution: 26
Number of Generated nodes: 129
Number of Expanded nodes: 26
Maximum number of nodes kept in memory: 104

Process finished with exit code 0
```

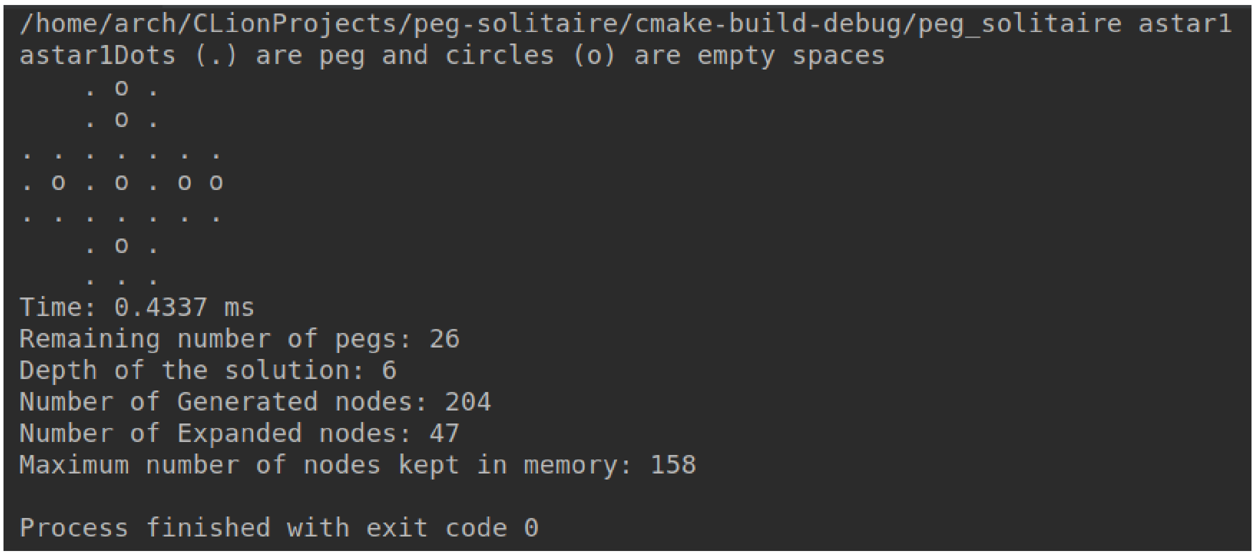
Figure 3: DFS result

As you can see from the results

- BFS generated much more nodes than DFS because of our goal is at the deeper levels because we need to find the state which there is no move, so DFS quickly reach to the goal state.
- Expanded nodes of the DFS again less than BFS because it deepens from one branch so it quickly expands the deeper nodes and find the result.
- Maximum number of nodes kept in the memory in DFS is less than BFS because of the same reasons on the above.
- Running time of the DFS is 60 times less than BFS so DFS found the goal faster.
- Number of the pegs in the final state is less in the DFS but it means that this is not optimal solution because path cost is increasing with each move and with the each move one peg will be removed so if there are lots of pegs at the solution than it's more optimal. Since BFS search each depth one by one it's guaranteed that BFS found the optimal solution but DFS didn't find the optimal.

2. c)

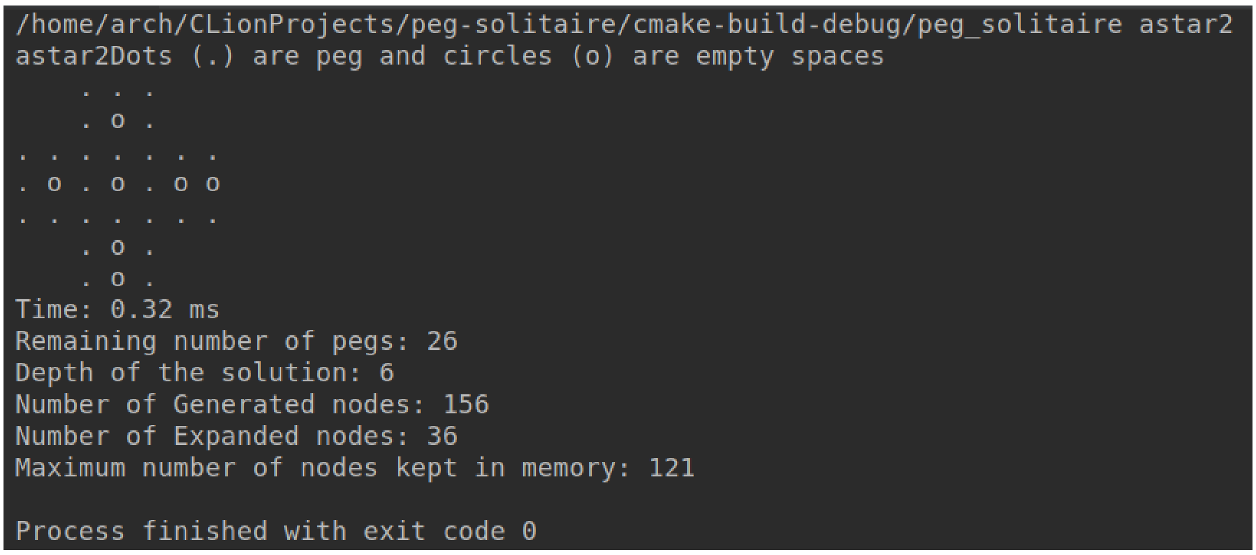
In the A* algorithm implementations I have used for "Artificial Intelligence: A Modern Approach" book pseudocode and changed it to adapt our problem. There are two heuristics that have been used. Both heuristics are admissible therefore A* algorithm is guaranteed to find the optimal (minimum number of moves). First heuristic is the number of possible moves that can be made on that state. Second heuristic is the number of the pegs that can move. As you can understand second heuristic always give less than or equal to first heuristic. Here you can see the results of the two heuristics of the A* algorithm. Number of the possible moves heuristic is in the Figure 4 and number of the pegs that can move is in the Figure 5.



```
/home/arch/CLionProjects/peg-solitaire/cmake-build-debug/peg_solitaire astar1
astar1Dots (.) are peg and circles (o) are empty spaces
    . o .
    . o .
. . . . . . .
. o . o . o o
. . . . . . .
    . o .
    . . .
Time: 0.4337 ms
Remaining number of pegs: 26
Depth of the solution: 6
Number of Generated nodes: 204
Number of Expanded nodes: 47
Maximum number of nodes kept in memory: 158

Process finished with exit code 0
```

Figure 4: A with heuristic 1*



```
/home/arch/CLionProjects/peg-solitaire/cmake-build-debug/peg_solitaire astar2
astar2Dots (.) are peg and circles (o) are empty spaces
    . . .
    . o .
. . . . . . .
. o . o . o o
. . . . . . .
    . o .
    . o .
Time: 0.32 ms
Remaining number of pegs: 26
Depth of the solution: 6
Number of Generated nodes: 156
Number of Expanded nodes: 36
Maximum number of nodes kept in memory: 121

Process finished with exit code 0
```

Figure 5: A with the heuristic 2*

- Number of the generated nodes in the heuristic 2 is less than the heuristic 1. I think it is because of it doesn't select the possible states that has more pegs to move so it finds faster than the first heuristic.
- Number of the expanded nodes in the heuristic 2 is again less than the heuristic 1 because of the same reason above.
- Maximum number of nodes kept in the memory is less in the heuristic 2 because it expanded less nodes than heuristic 1 so the frontier was smaller than heuristic 1.
- Running time of the heuristic 2 is smaller than heuristic 1 because it expanded less nodes.
- Number of the pegs in the solution state is same for both heuristics because both heuristics are admissible and therefore they are giving the optimal solution. As you can see two boards are not the same but symmetric so their heuristic costs and real costs are

the same so path costs are the same for both solutions so they both found optimal solutions.

2.d)

If objective was finding the state where only one peg remains then DFS would be performing worse than current result because it will find so many paths that is not the answer. However DFS will be performing worse too because changed objective definitely will be in the deepest level of the tree. However comparing between DFS and BFS would be hard. It is correlated with factors b, d and m which are branching factors and depth of the optimal solution. But at the worst case scenario BFS and DFS would find at the same time because BFS worst case running time is b^d and in this changed goal d=m because one peg remaining is equal to the maximum depth so DFS worst case running time would be b^m and since d=m they would perform same in this changed objective. A* algorithm would perform better in this objective if heuristic is admissible because A* will move towards to the optimal with some information gathered before. It will find optimal like BFS and DFS because d = m but memory usage would be less than others.

3. EXPLANATION OF THE SOURCE CODE

There are 2 classes and one main.cpp file in the source codes. To compile the code you need to execute *g++ -std=c++11 main.cpp Node.cpp Action.cpp* then you can run the output file. You need to give an argument from the command line for the search algorithm. You have 4 options for search algorithms: bfs , dfs , astar1 , astar2 . If you run the program one of these arguments then you will see the result of the problem. You can see the solution board and more detailed information about the solution too.

In Action class there is a Direction enum which specify the action direction of the peg which can be Up,Down,Left and Right. For creating an object from this class you need to give x,y and Direction of the action where x and y are position of the peg with indexed 0 and 0,0 is the left top of the board.

In Node class there is a board vector of vectors which defines the state of the Node. Number of the peg is number of the peg on the board. action attribute holds the latest action that have been made by the parent node. As you can expect it is empty object if node is root node. Node* parent is the pointer to the parent Node. Heuristic holds the heuristic value of the Node according to the heuristic_type. heuristic_type can be either 1,2 or -1. If it is -1 then heuristic calculation wouldn't be made this is useful when you use DFS and BFS because you don't need any heuristic. Function initial_node() should be called when initial board Node is wanted. It basically sets the initial board, depth, heuristic and so on. make_move function only called inside from the object. It takes parent board and using past_action it calculates new board and number of pegs. print_state function returns beautified version of the board instead of -1,0 and 1. check_goal_state function checks if any move can be made on the board, if there is no move then it is a goal state. check_move_valid function checks if move can be made with given action. operator > and operator < overloads are important for the priority queue. Since priority queue places biggest priority to the top we need small heuristic+pathcost at the top so if depth(path_cost) + heuristic is smaller than other than this Node is more prioritized than other.