**Yousra Gaimes**

**Sofie Kalinke**

**Jackie Nobbe**

**CSC 3309 // Artificial Intelligence**

**Assignment 2 // Write-Up**

***Overview***

This program serves as a game solver shell, allowing different search methods and games to be implemented. This assignment was implemented using Java in Eclipse Mars.

***Search***

All information pertaining to a game is stored in the class file for that game. If there is more than one heuristic, the user has to simply call the specific heuristic in the GBF or A\* search, or replace/comment out the existing heuristic with a new one into the same getHeuristicCost() function.

***Heuristics***

For each of the two puzzles, look for or come up with one or two heuristics that help you reach the solution faster and implement them. Then answer the following questions.

Peg Solitaire

***Heuristic 1-weighted matrix-***

In A\* and Greedy-Best-First-Search we used a weighted matrix where we give holes in the outer corners the value 4 and holes in the inner corner the value 3. I will explain this on the example of the board (4) [[1]](#footnote-1) with the size 7x7, here the weighted board would look like the following:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 99 | 99 | 1 | 1/4 | 1 | 99 | 99 |
| 99 | 99 | 1/4 | 1/4 | 1/4 | 99 | 99 |
| 1 | 1/4 | 3/4 | 1/4 | 3/4 | 1/4 | 1 |
| 1/4 | 1/4 | 1/4 | 0 | 1/4 | 1/4 | 1/4 |
| 1 | 1/4 | 3/4 | 1/4 | 3/4 | 1/4 | 1 |
| 99 | 99 | 1/4 | 1/4 | 1/4 | 99 | 99 |
| 99 | 99 | 1 | 1/4 | 1 | 99 | 99 |

We notice that corner pegs have the lowest degree of freedom and hence make the board most likely to be unsolvable. Therefore we assign them with the highest value. If pegs are in the positions that we marked blue, they will move in future moves either towards the corner positions or to an other “blue position”, so we allot the value 3/4 to these positions. To make sure that the heuristic value at the goal state is equal to real cost, so zero, we have to set this value to zero.

We calculate the heuristic value of a state by adding up the individual values, assigned to the holes, if a peg occupies them and dividing this value by the number of pegs still left on the board. This step will increase the heuristic value the fewer pegs are left on the board, because in these situations it becomes more urgent to move the corner pegs.

1. Are they admissible/consistent? Why or why not?

We are working with a Search tree so we have to ask ourselves whether the heuristic is admissible.

The heuristic is admissible because the cost (here equal to the number of moves) to reach the goal will never be higher than the calculated heuristic function

(all n, h(n) <= h\*(n))[[2]](#footnote-2).

We can proof this through proving that the heuristic is consistent through induction.

Base: h(goal) = 0 = h\*(goal)

We assume that the general case h\*(n) >= h(n) is correct for every node n that is L steps away from the goal.

Let n’ be length (L+1) from a goal, so h\*(n) = c(n, n') + h\*(n') from compatibility we arrive tot he statement h\*(n) = c(n, n') + h\*(n') >= c(n, n') + h(n') >= h(n).

This statement is the equivalent oft he definition of a heuristic function.

1. Does it help or not? If there is a gain, is it in the cost of the search, in the cost of the solution found, both , neither? Is there a tradeoff? Do a little analysis and discuss your results.

No improvement in the cost of solution, since the cost to reach the goal state is per definition in this game always the same. The cost ( assuming a step cost of 1) would always be equal to the number of pegs on the initial board configuration – 1.

But the heuristic function based on the weighted matrix will help to reduce the cost of the search. The idea is that the heuristic will encourage board states that are less likely to be solvable. Avoiding board states with pegs in the weighted holes will cut off some branches in the search tree where the game would stall.

***Heuristic 2-Manhattan Distance***

The Manhattan Distance heuristic calculates the sum of distances from every peg to all the surrounding pegs. And following the procedure in heuristic 1, we sum these values up over the hole board and divide it through the number of pegs left on the board. This heuristic favors boards that had more clusters peg configuration. We didn’t end up implementing this heuristic but I am quite positive that this heuristic could improve the search considerably, by preventing the search algorithm to run into dead-ends. However, we have to take into account that this heuristic is quite complex and might rather slow down the search than improve it if we use it on a big board (e.g 20x20)

1. Are they admissible/consistent? Why or why not?

The use of the Manhattan distance in this example is not admissible. It will overestimate the actual cost of reaching the goal. But here this is not a problem even though the algorithm won’t be optimal, due to the fact that we don’t have a optimal goal anyway. Every goal is the same number of states (#of pegs -1) away from the initial state.

1. Does it help or not? If there is a gain, is it in the cost of the search, in the cost of the solution found, both , neither? Is there a tradeoff? Do a little analysis and discuss your results.

As I explained above the cost in of the solution doesn’t improve. But the cost of search does improve. The fact that this heuristic distinguishes very efficiently between “good” and “bad” board configurations. If a board is likely to be unsolvable because the pegs are to spread apart and it will hence lead to isolated pegs, the heuristic value increases considerably and the algorithm will not expand these states further. This will led to less expansions and a more efficient search.

Missionaries and Cannibals

***Heuristic 1-NumberOfPeopleOnInitialSide -1***

The heuristic function in this example would be the number of people on the initial shore-1. This would be a solution to the relaxed problem, if we don’t consider the number of cannibals and missionaries on each shore and just consider the fact that with every boat trip forth and back (two people fit into the boat; one has to row back) at most one person can reach the goal shore.

1. Are they admissible/consistent? Why or why not?

This heuristic is admissible, because every boat trip over the river (except the last one) would result into at most one person to change the shore, since always one person has to row the boat back.

1. Does it help or not? If there is a gain, is it in the cost of the search, in the cost of the solution found, both , neither? Is there a tradeoff? Do a little analysis and discuss your results.

This heuristic will definitely improve the cost of the solution found, since it favors states with less people on the initial shore. Therefore the algorithm will find a more optimal solution. The path to a solution that has less people after fewer expansions is more likely to be optimal than one that has more people on the initial shore after the same number of expansions. Additionally, we reduce the cost in search since we cut of branches that will not lead to an optimal goal.

Remark:

This heuristic function has a local minimum because every return trip would increase h(n) instead of decreasing our function.

***Heuristic 2-NumberofPeopleOnTheInitialShore/BoatCapacity***

Another approach would be to take the number of people on the initial shore and divide it through the capacity of the boat as a heuristic function. This is a good indicator for the distance to the goal, since a bigger boat capacity decreases the distance to the goal and a smaller one respectively increases it.

1. Are they admissible/consistent? Why or why not?

It is admissible because this heuristic is a solution to the relaxed problem that no cannibal or missionary has to row the boat back. So it respectively always underestimates the real problem. The cost of the found solution should always be reduced since this heuristic encourages the algorithm to find the optimal solution, the fastest possible way to get all the people to the goal shore.

***Problem Reporting***

One problem experienced (and not overcome) in the programming process was attempting to use an organized closed list, indexed by the number of pegs left on the board. In concept, the search would look in the part of an array with the index of the pegs remaining, and only search game states in that corresponding list, thus cutting down on board states to search, and consequently, cutting down on search time.

The weighted Matrix heuristic we implemented in the code does only work for the board shape 4[[3]](#footnote-3) in this configuration. However, the idea would be easily applicable to other Board shapes as well, we would simply need to define a new Matrix in the getHeuristicCost() function, based on the ideas we presented above (Heuristic 1- weighted matrix).

***Extra Credit***

The Triangle Puzzle is a collection of 15 holes organized into a triangle, into which pegs are placed, leaving at least one empty hole. Typically, it is the same rules of the other boards the only difference are the moves. A legal move can only be made if there is a peg flanked by both another peg and a hole, arranged in a line. So now you can jump diagonally.

The board will remain the same we can represent it using the same 7by7 2D array, but we will change the placement of the “2”s, “0”s, “1”s in the board.

The Goal state is the same as the other boards, one peg should be left on the board.

The Successor function will be modified because the moves are no more the same:

public Vector expand()

{

// For each operation, check if it is valid and will change

// the state. If so, generate a child and add it to the

// children Vector.

Vector children = new Vector();

// For each line of positions, and ...

for (int i = 0; i < LINEAR\_JUMP\_POSITIONS.length; i++) {

int[] lineOfPos = LINEAR\_JUMP\_POSITIONS[i];

// ... for each three positions along that line

for (int j = 0; j < (lineOfPos.length - 2); j++)

// ... check if a jump can occur.

if (pegs[lineOfPos[j+1]] == 'O' && (pegs[lineOfPos[j]] != pegs[lineOfPos[j+2]])) {

// If it can, create the child node and add it.

PegSolitaireNode child = (PegSolitaireNode)this.childClone();

for (int k = j; k < j + 3; k++)

child.pegs[lineOfPos[k]] =

(child.pegs[lineOfPos[k]] == 'O') ? ' ' : 'O';

children.add(child);

}

}

return children;

}[[4]](#footnote-4)

We will keep the same idea when it comes to the search strategies.

Path cost function will be the same as well.

As we already mentioned we would have to adapt the heuristic 1(weighted matrix) to work for a triangular problem. Nevertheless, Heuristic 2 would still work for the new board shape.

Note: if there is difficulty using the code in the zip file, it can be found in a git repository at (<https://github.com/jacnobbe/GameSearchSell>).

1. Source: wikipedia: https://en.wikipedia.org/wiki/Peg\_solitaire [↑](#footnote-ref-1)
2. h(n) = estimated cost to reach the goal from n; h\*(n) = actual cost to reach the goal from n [↑](#footnote-ref-2)
3. Source: wikipedia: https://en.wikipedia.org/wiki/Peg\_solitaire [↑](#footnote-ref-3)
4. http://cs.gettysburg.edu/~tneller/resources/ai-search/uninformed-java/code/PegSolitaireNode.java [↑](#footnote-ref-4)