Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Cristea Tudor Iosif
Group: 30433
Email: cristeatudor6@gmail.com

Teaching Assistant: Adrian Groza
Adrian.Groza@cs.utcluj.ro

Contents

Table 1: Lab scheduling

Activity	**Deadline**
Searching agents, Linux, Latex, Python, Pacman	W_1
Uninformed search	W_2
Informed Search	W_3
Adversarial search	W_4
Propositional logic	W_5
First order logic	W_6
Inference in first order logic	W_7
Knowledge representation in first order logic	W_8
Classical planning	W_9
Contingent, conformant and probabilistic planning	W_{10}
Multi-agent planing	W_{11}
Modelling planning domains	W_{12}
Planning with event calculus	W_{14}

Lab organisation.

1. Laboratory work is 25% from the final grade.
2. There are three deliverables in total: 1. Search, 2. Logic, 3. Planning.
3. Before each deadline, you have to send your work (latex documentation/code) at moodle.cs.utcluj.ro
4. We use Linux and Latex
5. Plagiarism: Don't be a cheater! Cheating affects your colleagues, scholarships and a lot more.

Chapter 1

A1: Search

In this chapter, the work/code is done on a modified version of the *Pac-Man* game. This alternative version was developed in Python and for Linux based operating systems by the University of California, Berkeley.

1.1 Search Agent Problem

The first problem that needs to be solved is the so-called *"Search Agent"* problem which consists of a starting position from which the *Pac-Man* character begins the search and an ending position which needs to be reached. Multiple searching algorithms are implemented and presented below as solutions to this problem.

1.1.1 Random Search

Firstly, a simple, but naïve solution is to choose a random position from one of the four available successors. Evidently, this makes the number of steps taken to reach the *ending position* completely unpredictable, even if the map remains unchanged.

1.1.2 Depth First Search

Secondly, an intuitive solution is inspired from graphs and that is the *Depth First Search* algorithm. For this algorithm, a separate class for representing the nodes that will be *pushed* and *popped* from a *stack* is used, since a Last-In-First-Out approach needs to be adopted. Starting from the last node which was popped from the stack (which also contains the ending position), the "resulting graph" is traversed until the root (which has no parent) of the graph is reached. While doing so, every node's corresponding move is inserted into the list of moves. However, since the moves are needed in the opposite order, the list of moves is reversed. This solution is significantly better than the first one, when it comes to the execution time.

1.1.3 Breadth First Search

Another solution inspired from graphs is the other well-known searching algorithm, namely *Breadth First Search*. The solution is almost identical to the last one with the simple modification of the order in which the nodes are extracted from the array, which actually is now a queue (a First-In-First-Out approach needs to be adopted). Basically, we don't extract the last element inserted but the first. In the restricted set of test cases, *BFS* performed equally or better than *DFS*.

1.1.4 Iterative Deepening Search

A modified version of the *BFS* algorithm can also be used to solve this problem, namely the *Iterative Deepening Search* algorithm, which uses a *depth* parameter in order to perform numerous *DFS* searches at different depths, until the first good one is reached. Basically, this parameter tells the algorithm how many nodes should the path have. The algorithm incorporates the advantages of the two search algorithms, by combining the speed and the use of less memory of the *DFS* algorithm while being an optimal algorithm thanks to the optimality property of the *BFS* algorithm.

1.1.5 Uniform Cost Search

The last uninformed search algorithm is the *Uniform Cost Search* algorithm which is inspired by *Dijkstra's Algorithm*. It implies the usage of a priority queue in which a modified version of the *Node* class is used, the only modification being that the cost of the path from the start and up to that node is also stored. Then, for every node that is not yet visited, we append to the priority queue, the successors of the node and continue the search by selecting the node that has the smallest cost associated to it, until we reach the goal.

1.1.6 A* Search

Moving on to solutions that are commonly known in the vast domain of AI, the first solution for this problem comes in the form of the *A* Search* algorithm, an informed search algorithm. This is a modified version of the *Uniform Cost Search* algorithm, in which a heuristic is added to every node, so $f(n) = g(n) + h(n)$, where g is the cost of the path from the start node to the current node and h represents the heuristic from the current node to the goal node. This heuristic is a number which represents an estimate of the cost that it takes the agent to reach the goal from that node, and for it to be a good/admissible heuristic, it has to be an optimistic one, meaning that it has to underestimate the real cost of getting from the node to the goal. Therefore, the algorithm keeps track of two different costs, one without the heuristic added and one with the heuristic added and the priority queue will use the latter. The rest of the algorithm is similar to the previous one.

Heuristics

The heuristics that can be used for this algorithm (and for some of the following ones) are the *Manhattan distance* and *Euclidean distance*. However, I have added two more, namely the *Chebyshev distance* and the *Octile distance*. These last two expand less nodes than the first two. Moreover, I also implemented a modified version of all four heuristics starting from the idea of comparing the heuristic from the current node to the goal node with the heuristic from the current node to the starting node and taking the minimum value. Unfortunately, these modified versions of the heuristics are less efficient than their unmodified counterparts.

1.1.7 Weighted A* Search

A modified version of the *A* Search* algorithm is the *Weighted A* Search* algorithm, which multiplies every node's cost by a weight and the heuristic value by another weight in order to make the algorithm run a little faster (works only in some cases). These weights have the purpose of putting emphasis on $g(n)$ or $h(n)$, respectively. They can also be dynamic in the sense that, throughout a running instance of the program, they can change their values depending on the distance from the current state to the goal state. Thus, the previous formula

for f becomes: *f(n) = w1 * g(n) + w2 * h(n).* Other than this, the algorithm works based on the same principles.

1.1.8 BEAM Search

Another version of the *A* Search* algorithm is the *BEAM Search* algorithm which restricts the size of the priority queue to be at most a certain constant, β, and so, the number of choices for the next node is somewhat pruned. The main advantage of this algorithm is the fact that it uses less memory, but the main disadvantage is the fact that it is not a complete algorithm due to the fact that the global optimal choice could be pruned without being expanded. Therefore, the value for the β parameter has to be chosen carefully and specifically for the problem that needs to be solved. In my own testing on the *Pac-Man* search problem, a value of $\beta = 6$ is good for the small and medium mazes, but for the tiny maze a value of $\beta = 4$ is more appropriate whereas for the big maze a value of $\beta = 11$ is good enough (the heuristic used for all of these values is the null heuristic).

1.1.9 Hill Climbing Search

Another searching algorithm that can be implemented for this problem is the *"Hill Climbing"* algorithm, which uses a fitness function in order to select the best possible successor. Unfortunately, in our *Pac-Man* game, since all the successors have the same fitness function value, the algorithm is not sufficient, and therefore, incomplete. However, in order to make it complete, I implemented one of its variants, namely the *"Random Restart Hill Climbing"*, that simply chooses randomly from the grid (excluding the previous starting positions, the goal position and wall positions), a new starting position from which the algorithm restarts and tries to find the goal. This version of the algorithm is very inefficient, but it is complete. Another aspect that is worth mentioning is the fact that due to the current settings of the game, even though the path to the goal is found successfully and the list of moves is returned, these moves will not be rendered visually since the problem has a starting state which contains a starting position that cannot be changed (I tried to change it, but I was unsuccessful).

1.1.10 Genetic Algorithm/Programming Search

The last searching algorithm that I chose to implement is based on *Genetic Algorithms*. In the beginning of the algorithm, a finite number of valid paths (6 in this case) are randomly generated. Then, for a certain number of generations, the following actions are performed on the population:

- for every population, the fitness score is computed; the scores are computed using the octile distance between each position in the path list with the position of the goal

- using random members from the current population, an attempt at crossovering them is made, and if successful, a child is created

- for the crossover procedure, the middle index of the list of coordinates of parent1 and the middle index of the list of coordinates of parent2 are considered and the coordinates found at those indexes are compared; if they are equal, then the new path is constructed by concatenating the first half of parent1 with the second half of parent2 (or vice-versa if we are at the second batch); if they are not equal, then a potential common neighbour is searched; if this fails too, then the first middle index is decremented and the second one is incremented and the last two steps are repeated; when a common neighbour is found, than the two subpaths are concatenated and in between them, the common neighbour is inserted

- afterwards, a mutation is applied on the child and its fitness score is generated; the mutation basically consists of a small improvement that is applied on the child's path by

removing one redundant repetition of coordinates that occurs (e.g.: $(1,2) \rightarrow (1,3) \rightarrow (1,2)$, in this example (1,3) and (1,2) are deleted from the list)

- if the child's fitness score is better (smaller) than the "best parent's" score, then the search is stopped and the child is returned from the function; otherwise, the search continues until a suitable child is eventually found; if a suitable child cannot be obtained no matter what, then those parents cannot create a child, and so the process continues with a new pair of parents

- in the end, the path has to be created in order to display it inside the game; for this purpose, the list of population members is ordered ascendingly by the fitness score, so that the first member (the best one) is selected to be displayed

1.1.11 Comparing The Performances Of All Algorithms (Four Directions: N, S, W, E)

In the following three tables (one for each of the three maps: small, medium and big), all of the performances of the search algorithms that were implemented are compared in terms of Nodes Expanded (N.E.), Cost and Time. For the genetic algorithm, only the results for the small map were put in the table, but instead of randomly generating the paths as part of the algorithm, they were randomly pregenerated, but the same ones were used as population members for the genetic algorithm.

(C.W. = Cost Weight; H.W. = Heuristic Weight; B.W. = Beam Width)

Algorithm	Heuristic/Generations	C.W.	H.W.	B.W.	N.E.	Cost	Time
Random	-	-	-	-	1788	1787	0.0
DFS	-	-	-	-	149	57	0.0
BFS	-	-	-	-	186	19	0.0
IDS	-	-	-	-	11027	57	0.9
UCS	-	-	-	-	186	19	0.0
A*	manhattan	-	-	-	76	19	0.0
A*	euclidean	-	-	-	100	19	0.0
A*	chebyshev	-	-	-	102	19	0.0
A*	octile	-	-	-	59	19	0.0
A*	bidirectionalManhattan	-	-	-	127	19	0.0
A*	bidirectionalEuclidean	-	-	-	135	19	0.0
A*	bidirectionalChebyshev	-	-	-	142	19	0.0
A*	bidirectionalOctile	-	-	-	116	19	0.0
Weighted A*	octile	1	2	-	69	19	0.0
Weighted A*	octile	2	1	-	132	19	0.0
Weighted A*	octile	1	3	-	71	19	0.0
Weighted A*	octile	3	1	-	149	19	0.0
Weighted A*	octile	2	3	-	67	19	0.0
Weighted A*	octile	3	2	-	112	19	0.0
BEAM	octile	-	-	6	54	37	0.0
BEAM	octile	-	-	8	54	35	0.0
BEAM	octile	-	-	10	62	33	0.0
Hill Climb	-	-	-	-	367	999999	0.0
Genetic	5	-	-	-	35764	295	0.1
Genetic	10	-	-	-	36506	289	0.2
Genetic	50	-	-	-	32072	113	0.2
Genetic	100	-	-	-	26806	213	0.1
Genetic	500	-	-	-	26790	119	0.1

Table 1.1: For small map

Algorithm	Heuristic	C.W.	H.W.	B.W.	N.E.	Cost	Time
Random	-	-	-	-	24785	24784	0.3
DFS	-	-	-	-	551	148	0.2
BFS	-	-	-	-	548	68	0.2
IDS	-	-	-	-	151526	148	36.4
UCS	-	-	-	-	548	68	0.1
A*	manhattan	-	-	-	414	68	0.1
A*	euclidean	-	-	-	429	68	0.1
A*	chebyshev	-	-	-	435	68	0.1
A*	octile	-	-	-	419	19	0.0
A*	bidirectionalManhattan	-	-	-	449	68	0.2
A*	bidirectionalEuclidean	-	-	-	459	68	0.2
A*	bidirectionalChebyshev	-	-	-	468	68	0.2
A*	bidirectionalOctile	-	-	-	435	19	0.2
Weighted A*	octile	1	2	-	380	68	0.1
Weighted A*	octile	2	1	-	490	68	0.2
Weighted A*	octile	1	3	-	111	74	0.0
Weighted A*	octile	3	1	-	498	68	0.2
Weighted A*	octile	2	3	-	438	68	0.2
Weighted A*	octile	3	2	-	478	68	0.2
BEAM	octile	-	-	11	302	152	0.1
BEAM	octile	-	-	15	306	152	0.1
BEAM	octile	-	-	19	360	134	0.1
Hill Climb	-	-	-	-	3314	999999	0.1

Table 1.2: For medium map

Algorithm	Heuristic	C.W.	H.W.	B.W.	N.E.	Cost	Time
Random	-	-	-	-	105937	105936	1.2
DFS	-	-	-	-	1029	312	0.7
BFS	-	-	-	-	1237	210	1.0
IDS	-	-	-	-	528907	312	245.9
UCS	-	-	-	-	1237	210	0.3
A*	manhattan	-	-	-	1077	210	0.8
A*	euclidean	-	-	-	1101	210	0.8
A*	chebyshev	-	-	-	1146	210	0.9
A*	octile	-	-	-	1065	210	0.8
A*	bidirectionalManhattan	-	-	-	1077	210	0.8
A*	bidirectionalEuclidean	-	-	-	1101	210	0.8
A*	bidirectionalChebyshev	-	-	-	1146	210	0.9
A*	bidirectionalOctile	-	-	-	1067	210	0.8
Weighted A*	octile	1	2	-	950	210	0.7
Weighted A*	octile	2	1	-	1153	210	0.9
Weighted A*	octile	1	3	-	910	210	0.6
Weighted A*	octile	3	1	-	1207	210	1.0
Weighted A*	octile	2	3	-	959	210	0.7
Weighted A*	octile	3	2	-	1121	210	0.9
BEAM	octile	-	-	11	1203	212	1.0
BEAM	octile	-	-	16	1234	210	1.0
BEAM	octile	-	-	21	1238	210	1.1
Hill Climb	-	-	-	-	5628	999999	0.1

Table 1.3: For big map

1.1.12 Comparing The Performances Of All Algorithms (Eight Directions: N, S, W, E, NE, NW, SE, SW)

In order for *Pac-Man* to be able to go on diagonal directions as well, the following modifications have to be made to the game:

Listing 1.1: In the *game.py* file

```
class Directions:
    NORTH = 'North'
    SOUTH = 'South'
    EAST = 'East'
    WEST = 'West'
    NORTH_WEST = 'North-West'
    NORTH_EAST = 'North-East'
    SOUTH_WEST = 'South-West'
    SOUTH_EAST = 'South-East'
    STOP = 'Stop'
    ...
    ...
    class Actions:
    """A collection of static methods for manipulating move
       actions."""
```

```
    # Directions
    _directions = {Directions.NORTH:       (0, 1),
                   Directions.SOUTH:       (0, -1),
                   Directions.EAST:        (1, 0),
                   Directions.WEST:        (-1, 0),
                   Directions.NORTH_WEST:  (-1, 1),
                   Directions.NORTH_EAST:  (1, 1),
                   Directions.SOUTH_WEST:  (-1, -1),
                   Directions.SOUTH_EAST:  (1, -1),
                   Directions.STOP:        (0, 0)}
    ...
    ...
    def vectorToDirection(vector):
        dx, dy = vector
        if dy > 0 and dx < 0:
            return Directions.NORTH_WEST
        if dy > 0 and dx > 0:
            return Directions.NORTH_EAST
        if dy < 0 and dx < 0:
            return Directions.SOUTH_WEST
        if dy < 0 and dx > 0:
            return Directions.SOUTH_EAST
        if dy > 0:
            return Directions.NORTH
        if dy < 0:
            return Directions.SOUTH
        if dx < 0:
            return Directions.WEST
        if dx > 0:
            return Directions.EAST
        return Directions.STOP
```

Listing 1.2: In the *searchAgents.py* file

```
    def getSuccessors2(self, state):
        successors = []
        for action in [Directions.NORTH, Directions.SOUTH,
           Directions.EAST, Directions.WEST, Directions.
           NORTH_EAST, Directions.NORTH_WEST, Directions.
           SOUTH_EAST, Directions.SOUTH_WEST]:
            x,y = state
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                nextState = (nextx, nexty)
                cost = self.costFn(nextState)
                successors.append( ( nextState, action, cost) )

        # Bookkeeping for display purposes
        self._expanded += 1 # DO NOT CHANGE
        if state not in self._visited:
```

```
            self._visited[state] = True
            self._visitedlist.append(state)

        return successors
```

Below, another three tables, which compare the performance metrics of the implemented algorithms, are presented. Their format is the same as before, with the exception that the genetic algorithm is not part of the comparison in this case.

Algorithm	**Heuristic**	**C.W.**	**H.W.**	**B.W.**	**N.E.**	**Cost**	**Time**
Random	-	-	-	-	1242	1241	0.0
DFS	-	-	-	-	217	109	0.1
BFS	-	-	-	-	260	15	0.1
IDS	-	-	-	-	23437	109	3.1
UCS	-	-	-	-	260	15	0.0
A*	manhattan	-	-	-	72	15	0.0
A*	euclidean	-	-	-	82	15	0.0
A*	chebyshev	-	-	-	109	15	0.0
A*	octile	-	-	-	68	15	0.0
A*	bidirectionalManhattan	-	-	-	171	15	0.0
A*	bidirectionalEuclidean	-	-	-	179	15	0.0
A*	bidirectionalChebyshev	-	-	-	192	15	0.0
A*	bidirectionalOctile	-	-	-	141	15	0.0
Weighted A*	octile	1	2	-	77	15	0.0
Weighted A*	octile	2	1	-	165	15	0.0
Weighted A*	octile	1	3	-	74	23	0.0
Weighted A*	octile	3	1	-	210	15	0.0
Weighted A*	octile	2	3	-	75	15	0.0
Weighted A*	octile	3	2	-	93	15	0.0
BEAM	octile	-	-	6	72	28	0.0
BEAM	octile	-	-	8	64	24	0.0
BEAM	octile	-	-	10	64	23	0.0
Hill Climb	-	-	-	-	779	999999	0.0

Table 1.4: For small map

Algorithm	Heuristic	C.W.	H.W.	B.W.	N.E.	Cost	Time
Random	-	-	-	-	4755	4754	0.1
DFS	-	-	-	-	719	210	0.5
BFS	-	-	-	-	728	58	0.5
IDS	-	-	-	-	258122	210	108.6
UCS	-	-	-	-	728	58	0.2
A*	manhattan	-	-	-	548	59	0.1
A*	euclidean	-	-	-	545	58	0.1
A*	chebyshev	-	-	-	562	58	0.1
A*	octile	-	-	-	529	59	0.1
A*	bidirectionalManhattan	-	-	-	576	58	0.1
A*	bidirectionalEuclidean	-	-	-	588	58	0.1
A*	bidirectionalChebyshev	-	-	-	604	58	0.1
A*	bidirectionalOctile	-	-	-	560	58	0.1
Weighted A*	octile	1	2	-	110	59	0.0
Weighted A*	octile	2	1	-	641	58	0.1
Weighted A*	octile	1	3	-	110	59	0.0
Weighted A*	octile	3	1	-	660	58	0.1
Weighted A*	octile	2	3	-	116	59	0.0
Weighted A*	octile	3	2	-	620	58	0.1
BEAM	octile	-	-	11	124	60	0.0
BEAM	octile	-	-	15	131	60	0.0
BEAM	octile	-	-	34	247	80	0.0
Hill Climb	-	-	-	-	1554	999999	0.0

Table 1.5: For medium map

Algorithm	Heuristic	C.W.	H.W.	B.W.	N.E.	Cost	Time
Random	-	-	-	-	88007	88006	1.1
DFS	-	-	-	-	1511	668	2.3
BFS	-	-	-	-	1763	156	3.3
IDS	-	-	-	-	1140806	668	1087.0
UCS	-	-	-	-	1763	156	0.9
A*	manhattan	-	-	-	1364	156	0.6
A*	euclidean	-	-	-	1396	156	0.6
A*	chebyshev	-	-	-	1401	156	0.6
A*	octile	-	-	-	1351	156	0.6
A*	bidirectionalManhattan	-	-	-	1377	156	0.6
A*	bidirectionalEuclidean	-	-	-	1396	156	0.6
A*	bidirectionalChebyshev	-	-	-	1401	156	0.6
A*	bidirectionalOctile	-	-	-	1367	156	0.6
Weighted A*	octile	1	2	-	1262	156	0.5
Weighted A*	octile	2	1	-	1557	156	0.8
Weighted A*	octile	1	3	-	1196	156	0.5
Weighted A*	octile	3	1	-	1631	156	0.8
Weighted A*	octile	2	3	-	1293	156	0.5
Weighted A*	octile	3	2	-	1513	156	0.7
BEAM	octile	-	-	40	1156	160	0.4
BEAM	octile	-	-	50	1285	159	0.5
BEAM	octile	-	-	60	1291	159	0.5
Hill Climb	-	-	-	-	15779	999999	0.2

Table 1.6: For big map

1.2 Corners Problem

The second problem that needs to be solved is the *"Corners Problem"* which consists of a starting position from which *Pac-Man* begins the search and four ending positions represented by the four corners of the maze's map, all of which need to be reached. The searching algorithms have already been implemented (the same ones from the first problem also work here) and so, all that needs to be done is to define a new representation of the state of this search problem, meaning that the following functions need to be implemented: *getStartState()*, *isGoalState()* and *getSuccessors()*. Here, in order for the already implemented algorithms to work on this new problem without modifying them, a new structure needs to be added alongside the coordinates of a state. for this, I have chosen an appearance list meaning that, whenever a new node is created, at the beginning, of the search, it will contain a list of four zeros ([0,0,0,0]), but, once a corner has been reached, the list changes one of the four values to one ([1,0,0,0]), corresponding to that corner. In this way, we avoid two problems:

1) The search algorithm does not get stuck in a corner due to the fact that the positions around it were already visited

2)There is a distinction between every corner of the map since we have four different elements in the list.

Additionally, another aspect which can be implemented is the *cornersHeuristic()* function which has to be an admissible and efficient heuristic that is specific only to this search problem. Something like the Manhattan distance heuristic can also be used here, but the most appropriate one that I could find is a heuristic which is equal to the maximum octile distance to the unvisited

corners. To this number, the number of unvisited corners is added, at the end. For reference, the number of nodes expanded (on the *mediumCorners* map), using the Manhattan distance heuristic, is 2266, while using the octile distance heuristic, this number drops to about 2048, which is not a big difference, but as the map gets larger, so does the gap between these two numbers (for the *bigCorners* map, Manhattan expands 8902 nodes while octile expands 6974 nodes).

1.3 All Food Problem

The third problem that needs to be solved is the *"All Food Problem"* which consists of a starting position from which *Pac-Man* begins the search and there are multiple food pellets across the map, all of which need to be eaten by the game's character. for this problem, there is no need to define another representation for its state because the three functions which were mentioned in the previous paragraph are already implemented. The challenge for this problem is too find an admissible and good heuristic, that can decrease the number of expanded nodes as much as possible. Starting from the idea of the previous problem's heuristic, I first tried the maximum octile distance summed up with the number of remaining food pellets. However, this did not work so well for this problem (in fact it was so bad, that the game was frozen for a very long time, until I got bored and manually closed it), so instead of taking the maximum distance, I decided to take the sum of all octile distances to the remaining food pellets summed up with the number of remaining food pellets. This yielded a very good result in comparison with the previous heuristic (10915 expanded nodes), but it still wasn't enough, so I came up with another idea which turned out to be quite good. Since the octile distance is the largest out of the four distances that I implemented, namely the Manhattan distance, the Euclidean distance, the Chebyshev distance and the octile distance itself, I decided to multiply the latter by two and subtract the Manhattan one. Again, a visible improvement was achieved (9462 expanded nodes), so then I continued this idea and thus, multiplied the octile distance by four and subtracted the sum of all the other distances. The number of expanded nodes continue to decrease (down to 8520). The last improvement that I found was to multiply the final result with a weight that I kept increasing until I found the smallest possible number of expanded nodes that I could obtain (4646 expanded nodes obtained with a weight equal to 500). Increasing the weight further did not affect the performance in any way. Thus, this last improvement (with the weight) is the same as having a *Weighted A** algorithm and setting the weight of the heuristic to 500. In addition, this heuristic can also be applied to the previous problem, since the *"All Food Problem"* is essentially a generalization of the *"Corners Problem"*.

Chapter 2

A2: Logics

Chapter 3

A3: Planning

Bibliography

Berkeley's version of the Pac-Man game
AI course slides
Video Tutorials on YouTube
Various tutorials found on the internet
Additional information about heuristics

Appendix A

Your original code

Don't be a cheater! Cheating affects your colleagues, scholarships and a lot more. This section should contain only code developed by you, without any line re-used from other sources. This section helps me to correctly evaluate your amount of work and results obtained.

A.1 Search

A.1.1 Search Agent Problem

Random Search

Listing A.1: Random Search Algorithm

```
def randomSearch(problem):
    current_state = problem.getStartState()
    list1 = []
    while (problem.isGoalState(current_state) == False):
        successors = problem.getSuccessors(current_state)
        random_successor = random.choice(successors)
        current_state = random_successor[0]
        list1.append(random_successor[1])

    return list1
```

Depth First Search

Listing A.2: Node Class

```
class Node:
    def __init__ (self, triplet, parent):
        self.triplet = triplet
        self.parent = parent
    def getTriplet(self):
        return self.triplet
    def getParent(self):
        return self.parent
    def __cmp__(self, that):
        return cmp(self.triplet, that.getTriplet())
```

Listing A.3: DFS Algorithm

```
def depthFirstSearch(problem):
    start_position = problem.getStartState()

    current_node = Node((start_position, None, 0), None)
    stack = [current_node]
    visited = []

    while stack:
        current_node = stack.pop()
        current_position = current_node.getTriplet()[0]

        if (problem.isGoalState(current_position) == True):
            break

        if (current_node not in visited):
            visited.append(current_node)
            successors = problem.getSuccessors2(current_position)
            for successor in successors:
                successorNode = Node(successor, current_node)
                if (successorNode not in visited):
                    stack.append(successorNode)

    list2 = []
    while current_node.getParent():
        list2.append(current_node.getTriplet()[1])
        current_node = current_node.getParent()

    list2.reverse()
    return list2
```

Breadth First Search

Listing A.4: BFS Algorithm

```
def breadthFirstSearch(problem):
    start_position = problem.getStartState()

    current_node = Node((start_position, None, 0), None)
    queue = [current_node]
    visited = []

    while queue:
        current_node = queue.pop(0)
        current_position = current_node.getTriplet()[0]

        if (problem.isGoalState(current_position) == True):
            break

        if (current_node not in visited):
```

```
            visited.append(current_node)
            successors = problem.getSuccessors2(current_position)
            for successor in successors:
                successorNode = Node(successor, current_node)
                if (successorNode not in visited):
                    queue.append(successorNode)

    list3 = []
    while current_node.getParent():
        list3.append(current_node.getTriplet()[1])
        current_node = current_node.getParent()

    list3.reverse()
    return list3
```

Iterative Deepening Search

Listing A.5: Depth Limited Search

```
def depthLimitedSearch(problem, start_position, depth):
    """Variant of DFS, but has a certain depth until which is
       allowed to go"""
    """This is only a helper function for the IDS algorithm"""

    current_node = Node((start_position, None, 0), None)
    stack = [current_node]
    visited = []
    foundGoal = False

    while stack:
        current_node = stack.pop()
        current_position = current_node.getTriplet()[0]

        if (problem.isGoalState(current_position) == True):
            foundGoal = True
            break

        if (current_node not in visited):
            visited.append(current_node)
            depth -= 1
            if (depth < 0):
                return None
            successors = problem.getSuccessors2(current_position)
            for successor in successors:
                successorNode = Node(successor, current_node)
                if (successorNode not in visited):
                    stack.append(successorNode)

    list3 = []
    if (foundGoal == True):
```

```
        while current_node.getParent():
            list3.append(current_node.getTriplet()[1])
            current_node = current_node.getParent()

        list3.reverse()

    return list3
```

Listing A.6: IDS Algorithm

```
def iterativeDeepeningSearch(problem):
    start_position = problem.getStartState()

    depth = 1
    list3 = []
    while not list3:
        list3 = depthLimitedSearch(problem, start_position, depth
            )
        depth += 1

    return list3
```

Uniform Cost Search

Listing A.7: Node2 Class

```
class Node2:
    def __init__ (self, triplet, parent, cost):
        self.triplet = triplet
        self.parent = parent
        self.cost = cost
    def getTriplet(self):
        return self.triplet
    def getParent(self):
        return self.parent
    def getCost(self):
        return self.cost
    def __cmp__(self, other):
        return cmp(self.triplet, other.getTriplet())
    def __eq__(self, other):
        return self.triplet == other.getTriplet()
```

Listing A.8: UCS Algorithm

```
def uniformCostSearch(problem):
    start_position = problem.getStartState()

    current_node = Node2((start_position, None, 0), None, 0)
    priority_queue = [current_node]
    visited = []

```

```
    while priority_queue:

        priority_queue = sorted(priority_queue, key=lambda x: x.
           getCost())
        current_node = priority_queue.pop(0)
        current_position = current_node.getTriplet()[0]

        if (problem.isGoalState(current_position) == True):
            break

        if (current_node not in visited):
            visited.append(current_node)
            successors = problem.getSuccessors2(current_position)
            for successor in successors:
                successorNode2 = Node2(successor, current_node,
                   current_node.getCost() + successor[2])
                if (successorNode2 not in visited):
                    priority_queue.append(successorNode2)

    list4 = []
    while current_node.getParent():
        list4.append(current_node.getTriplet()[1])
        current_node = current_node.getParent()

    list4.reverse()
    return list4
```

A* Search

Listing A.9: Node3 Class

```
class Node3:
    def __init__ (self, triplet, parent, cost, costWithHeuristic)
       :
        self.triplet = triplet
        self.parent = parent
        self.cost = cost
        self.costWithHeuristic = costWithHeuristic
    def getTriplet(self):
        return self.triplet
    def getParent(self):
        return self.parent
    def getCost(self):
        return self.cost
    def getCostWithHeuristic(self):
        return self.costWithHeuristic
    def __cmp__(self, that):
        return cmp(self.triplet, that.getTriplet())
    def __eq__(self, other):
        return self.triplet == other.getTriplet()
```

Listing A.10: A* Search Algorithm

```
def aStarSearch(problem, heuristic=nullHeuristic):
    start_position = problem.getStartState()

    current_node = Node3((start_position, None, 0), None, 0, 0)
    priority_queue = [current_node]
    visited = []

    while priority_queue:

        priority_queue = sorted(priority_queue, key=lambda x: x.
           getCostWithHeuristic())
        current_node = priority_queue.pop(0)
        current_position = current_node.getTriplet()[0]

        if (problem.isGoalState(current_position) == True):
            break

        if (current_node not in visited):
            visited.append(current_node)
            successors = problem.getSuccessors2(current_position)
            for successor in successors:
                successorNode3 = Node3(successor, current_node,
                   current_node.getCost() + successor[2],
                   current_node.getCost() + successor[2] +
                   heuristic(successor[0], problem))
                if (successorNode3 not in visited):
                    priority_queue.append(successorNode3)

    list5 = []
    while current_node.getParent():
        list5.append(current_node.getTriplet()[1])
        current_node = current_node.getParent()

    list5.reverse()
    return list5
```

Heuristics (besides Manhattan and Euclidean)

Listing A.11: Chebyshev Heuristic

```
def chebyshevHeuristic(position, problem, info={}):
    xy1 = position
    xy2 = problem.goal
    dx = abs(xy1[0] - xy2[0])
    dy = abs(xy1[1] - xy2[1])
    return max(dx, dy)
```

Listing A.12: Octile Heuristic

```
def octileHeuristic(position, problem, info={}):
    xy1 = position
    xy2 = problem.goal
    dx = abs(xy1[0] - xy2[0])
    dy = abs(xy1[1] - xy2[1])
    return (dx + dy) - ((2 ** 0.5) - 2) * min(dx, dy)
```

Listing A.13: Bidirectional Manhattan Heuristic

```
def bidirectionalManhattanHeuristic(position, problem, info={}):
    xy1 = position
    xy2 = problem.goal
    xy3 = problem.getStartState()
    h1 = abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])
    h2 = abs(xy1[0] - xy3[0]) + abs(xy1[1] - xy3[1])
    return min(h1, h2)
```

Listing A.14: Bidirectional Euclidean Heuristic

```
def bidirectionalEuclideanHeuristic(position, problem, info={}):
    xy1 = position
    xy2 = problem.goal
    xy3 = problem.getStartState()
    h1 = ( (xy1[0] - xy2[0]) ** 2 + (xy1[1] - xy2[1]) ** 2 ) **
       0.5
    h2 = ( (xy1[0] - xy3[0]) ** 2 + (xy1[1] - xy3[1]) ** 2 ) **
       0.5
    return min(h1, h2)
```

Listing A.15: Bidirectional Chebyshev Heuristic

```
def bidirectionalChebyshevHeuristic(position, problem, info={}):
    xy1 = position
    xy2 = problem.goal
    xy3 = problem.getStartState()
    dx1 = abs(xy1[0] - xy2[0])
    dy1 = abs(xy1[1] - xy2[1])
    dx2 = abs(xy1[0] - xy3[0])
    dy2 = abs(xy1[1] - xy3[1])
    h1 = max(dx1, dy1)
    h2 = max(dx2, dy2)
    return min(h1, h2)
```

Listing A.16: Bidirectional Octile Heuristic

```
def bidirectionalOctileHeuristic(position, problem, info={}):
    xy1 = position
    xy2 = problem.goal
    xy3 = problem.getStartState()
    dx1 = abs(xy1[0] - xy2[0])
    dy1 = abs(xy1[1] - xy2[1])
    dx2 = abs(xy1[0] - xy3[0])
```

```
    dy2 = abs(xy1[1] - xy3[1])
    h1 = (dx1 + dy1) - ((2 ** 0.5) - 2) * min(dx1, dy1)
    h2 = (dx2 + dy2) - ((2 ** 0.5) - 2) * min(dx2, dy2)
    return min(h1, h2)
```

Weighted A*

Listing A.17: Weighted A* Search Algorithm

```
def weightedAStarSearch(problem, heuristic=nullHeuristic,
   cost_weight=1.0, heuristic_weight=2.0):
    start_position = problem.getStartState()

    current_node = Node3((start_position, None, 0), None, 0, 0)
    priority_queue = [current_node]
    visited = []

    while priority_queue:

        priority_queue = sorted(priority_queue, key=lambda x: x.
           getCostWithHeuristic())
        current_node = priority_queue.pop(0)
        current_position = current_node.getTriplet()[0]

        if (problem.isGoalState(current_position) == True):
            break

        if (current_node not in visited):
            visited.append(current_node)
            successors = problem.getSuccessors2(current_position)
            for successor in successors:
                successorNode3 = Node3(successor, current_node,
                   current_node.getCost() + successor[2],
                   cost_weight * (current_node.getCost() +
                   successor[2]) + heuristic_weight * heuristic(
                   successor[0], problem))
                if (successorNode3 not in visited):
                    priority_queue.append(successorNode3)

    list5 = []
    while current_node.getParent():
        list5.append(current_node.getTriplet()[1])
        current_node = current_node.getParent()

    list5.reverse()
    return list5
```

BEAM Search

Listing A.18: BEAM Search Algorithm

```
def beamSearch(problem, heuristic=nullHeuristic, beta=11):
    start_position = problem.getStartState()

    current_node = Node3((start_position, None, 0), None, 0, 0)
    priority_queue = [current_node]
    visited = []

    found_goal = False
    while priority_queue and not found_goal:

        priority_queue = sorted(priority_queue, key=lambda x: x.
           getCostWithHeuristic())
        while len(priority_queue) > beta:
            priority_queue.pop()
        current_node = priority_queue.pop(0)
        current_position = current_node.getTriplet()[0]

        if (problem.isGoalState(current_position) == True):
            found_goal = True

        if (current_node not in visited):
            visited.append(current_node)
            successors = problem.getSuccessors2(current_position)
            for successor in successors:
                successorNode3 = Node3(successor, current_node,
                   current_node.getCost() + successor[2],
                   current_node.getCost() + successor[2] +
                   heuristic(successor[0], problem))
                if (successorNode3 not in visited):
                    priority_queue.append(successorNode3)

    list6 = []
    while current_node.getParent():
        list6.append(current_node.getTriplet()[1])
        current_node = current_node.getParent()

    list6.reverse()
    return list6
```

Hill Climbing (Random Restart)

Listing A.19: Fitness Function

```
def fitnessFunction(current_node):
    #in this problem, it is just the cost
    return current_node.getCost()
```

Listing A.20: Hill Climbing Search Algorithm (Random Restart)

```
def hillClimbingSearch(problem):
    start_position = problem.getStartState()
    wallList = problem.getAllWalls().asList()
    maximum_coordinate_x = max(wallList)[0]
    maximum_coordinate_y = max(wallList)[1]
    number_of_iterations = 0

    forbidden_positions = [(1, 1)] #add the goal position to the
       forbidden list of positions from which the hill-climbing
       algorithm cannot restart
    list7 = []
    foundGoal = False
    while (not foundGoal):

        number_of_iterations += 1
        while list7:
            list7.pop()

        current_node = None
        if number_of_iterations > 1:
            new_starting_position = None
            while True:
                random_x = random.randint(1, maximum_coordinate_x
                   )
                random_y = random.randint(1, maximum_coordinate_y
                   )
                new_starting_position = (random_x, random_y)
                if new_starting_position not in
                   forbidden_positions and new_starting_position
                   not in wallList:
                    break

            current_node = Node2((new_starting_position, None, 0)
               , None, 0)
        else:
            current_node = Node2((start_position, None, 0), None,
                0)

        visited = []
        forbidden_positions.append(current_node.getTriplet()[0])

        while True:
            current_position = current_node.getTriplet()[0]
            visited.append(current_node)
            current_fitness = fitnessFunction(current_node)

            if (problem.isGoalState(current_position) == True):
                foundGoal = True
                break

```

```
            best_fitness = 0x3f3f3f3f
            best_successor = None

            successors = problem.getSuccessors2(current_position)
            for successor in successors:
                successorNode = Node2(successor, current_node,
                    current_fitness + successor[2])
                neighbour_fitness = fitnessFunction(successorNode
                    )
                if (neighbour_fitness < best_fitness and
                    neighbour_fitness >= current_fitness and
                    successorNode not in visited):
                     best_fitness = neighbour_fitness
                     best_successor = successorNode
            if not best_successor:
                break
            else:
                current_node = best_successor
                list7.append(current_node.getTriplet()[1])

    return list7
```

Genetic Algorithm

Listing A.21: Population Class

```
class Population:
    def __init__ (self, listOfCoordinates, fitnessScore):
        self.listOfCoordinates = listOfCoordinates
        self.fitnessScore = fitnessScore
    def getListOfCoordinates(self):
        return self.listOfCoordinates
    def setListOfCoordinates(self, listOfCoordinates):
        self.listOfCoordinates = listOfCoordinates
    def getFitnessScore(self):
        return self.fitnessScore
    def setFitnessScore(self, fitnessScore):
        self.fitnessScore = fitnessScore
    def __cmp__(self, that):
        return cmp(self.listOfCoordinates, that.
            getListOfCoordinates())
    def __eq__(self, other):
        return self.listOfCoordinates == other.
            getListOfCoordinates()
```

Listing A.22: Initialize Population Function (creates randomly generated valid paths)

```
def initializePopulation(problem):
    current_position = problem.getStartState()
    list1 = [current_position]
    while (problem.isGoalState(current_position) == False):
```

```
        successors = problem.getSuccessors(current_position)
        random_successor = random.choice(successors)
        current_position = random_successor[0]
        list1.append(current_position)

    population = Population(list1, 0)
    print list1
    return population
```

Listing A.23: Evaluate Population Function (computes the fitness score of a path)

```
def evaluatePopulation(population, goal_position):
    fitness_score = 0
    for individual in population:
        if isinstance(individual, tuple) and len(individual) ==
           2:
            dx = abs(individual[0] - goal_position[0])
            dy = abs(individual[1] - goal_position[1])
            fitness_score += (dx + dy) - ((2 ** 0.5) - 2) * min(
               dx, dy)
        else:
            return 0x3f3f3f3f

    return fitness_score
```

Listing A.24: Common Neighbour Function (returns a common neighbour of two nodes)

```
def commonNeighbour(problem, element1, element2):
    successors1 = problem.getSuccessors(element1)
    successors2 = problem.getSuccessors(element2)
    for successor1 in successors1:
        for successor2 in successors2:
            if successor1[0] == successor2[0]:
                return successor1[0]

    return None
```

Listing A.25: Mutation Function

```
def mutation(child):
    n = len(child)
    for i in range(0, n - 3):
        if child[i] == child[i + 2]:
            new_child = []
            new_child.append(child[:i])
            new_child.append(child[(i + 2):])
            return new_child

    return child
```

Listing A.26: Crossover Function

```
def crossover(problem, parent1, parent2):
    childFinal = Population([], 0x3f3f3f3f)
    child = []

    coordinates1 = parent1.getListOfCoordinates()
    coordinates2 = parent2.getListOfCoordinates()
    n1 = len(coordinates1)
    n2 = len(coordinates2)
    middle1 = n1 // 2
    middle2 = n2 // 2
    index1 = middle1
    index2 = middle2
    restart = False
    second_batch = False

    best_parent_fitness = min(parent1.getFitnessScore(), parent2.
       getFitnessScore())

    while True:
        child = []
        if restart:
            index1 = middle1
            index2 = middle2
            restart = False
            second_batch = True

        element1 = coordinates1[index1]
        element2 = coordinates2[index2]
        while True:
            if not second_batch:
                if (index1 > 0):
                    index1 -= 1
                if (index2 < n2 - 1):
                    index2 += 1
                if index1 == 0 and index2 == n2 - 1:
                    break
            else:
                if (index1 < n1 - 1):
                    index1 += 1
                if (index2 > 0):
                    index2 -= 1
                if index1 == n1 - 1 and index2 == 0:
                    break
            element1 = coordinates1[index1]
            element2 = coordinates2[index2]
            if element1 == element2 or commonNeighbour(problem,
               element1, element2) is not None:
                break

        if element1 == element2:
```

```
            if not second_batch:
                child.append(coordinates1[:index1])
                child.append(coordinates2[index2:])
            else:
                child.append(coordinates1[:index2])
                child.append(coordinates2[index1:])
        else:
            neighbour = commonNeighbour(problem, element1,
               element2)
            if neighbour is not None:
                if not second_batch:
                    child.append(coordinates1[:index1])
                    child.append(neighbour)
                    child.append(coordinates2[index2:])
                else:
                    child.append(coordinates1[:index2])
                    child.append(neighbour)
                    child.append(coordinates2[index1:])
            else:
                if second_batch:
                    restart = True
                else:
                    return []

        if child:
            child = [item for sublist in child for item in
               sublist]
            if len(child) >= 3:
                child = mutation(child)
                child = [item for sublist in child for item in
                   sublist]
                childFinal.setListOfCoordinates(child)
                child_fitness = evaluatePopulation(child, problem
                   .goal)
                childFinal.setFitnessScore(child_fitness)
                if (child_fitness < best_parent_fitness):
                    break

    return childFinal
```

Listing A.27: Genetic Search Algorithm

```
def geneticSearch(problem):
    start_position = problem.getStartState()

    populations = []
    for i in range(1,6):
        populations.append(initializePopulation(problem))

    current_generation = 1
    maximum_number_of_generations = 500
```

```
    while (current_generation <= maximum_number_of_generations):

        for population in populations:
            population.setFitnessScore(evaluatePopulation(
               population.getListOfCoordinates(), problem.goal))

        number_of_populations = len(populations)
        if number_of_populations == 1:
            break
        maximum_number_of_combinations = number_of_populations *
           (number_of_populations - 1) / 2
        new_populations = []
        parent_combinations = []
        while len(new_populations) < number_of_populations:
            random_int1 = 0
            random_int2 = 0
            while True:
                random_int1 = random.randint(0,
                   number_of_populations - 1)
                random_int2 = random.randint(0,
                   number_of_populations - 1)
                if (random_int1 < random_int2 and (random_int1,
                   random_int2) not in parent_combinations) or
                   len(parent_combinations) ==
                   maximum_number_of_combinations:
                    break

            if len(parent_combinations) !=
               maximum_number_of_combinations:
                parent_combinations.append((random_int1,
                   random_int2))
                parent1 = populations[random_int1]
                parent2 = populations[random_int2]
                child = crossover(problem, parent1, parent2)
                if child:
                    new_populations.append(child)
            else:
                break

        if new_populations:
            populations = new_populations
            if len(populations) == 1:
                break
        else:
            break
        current_generation += 1

    list8 = []
    populations = sorted(populations, key=lambda x: x.
       getFitnessScore())
```

```
    population_list = populations[0].getListOfCoordinates()
    n = len(population_list)
    for i in range(0, n - 1):
        if problem.isGoalState(population_list[i]):
            break
        successors = problem.getSuccessors(population_list[i])
        for successor in successors:
            if successor[0] == population_list[i + 1]:
                list8.append(successor[1])

    return list8
```

A.1.2 Corners Problem

Modified Functions For The New Environment

Listing A.28: In the *searchAgent.py* file

```
def getStartState(self):
    return self.startingPosition

def isGoalState(self, state):
    return state[1] == [1, 1, 1, 1]

def getSuccessors(self, state):
    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions
       .EAST, Directions.WEST]:
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        hitsWall = self.walls[nextx][nexty]
        if not hitsWall:
            nextState = (nextx, nexty)
            cost = self.costFn(nextState)
            new_state = list(state[1])
            if nextState in self.corners:
                new_state[self.corners.index(nextState)] = 1
            successors.append(((nextState, new_state), action,
               cost))

    self._expanded += 1 # DO NOT CHANGE
    return successors
```

Listing A.29: Corners Heuristic

```
def cornersHeuristic(state, problem):
    result = 0
    x1, y1 = state[0]
    goals = state[1]
    number_of_unvisited_corners = 0
```

```
    for corner in corners:
        if (goals[corners.index(corner)] == 0):
            number_of_unvisited_corners += 1
            dx = abs(x1 - corner[0])
            dy = abs(y1 - corner[1])
            octileDistance = (dx + dy) - ((2 ** 0.5) - 2) * min(
               dx, dy)
            result = max(result, octileDistance)

    return result + number_of_unvisited_corners
```

A.1.3 All Food Search Problem

Listing A.30: All Food Heuristic

```
def foodHeuristic(state, problem):
    result = 0
    x1, y1 = position
    foodList = foodGrid.asList()
    for food in foodList:
        dx = abs(x1 - food[0])
        dy = abs(y1 - food[1])
        octileDistance = (dx + dy) - ((2 ** 0.5) - 2) * min(dx,
           dy)
        manhattanDistance = dx + dy
        euclideanDistance = (dx ** 2 + dy ** 2) ** 0.5
        chebyshevDistance = max(dx, dy)
        result += (4 * octileDistance - (manhattanDistance +
           euclideanDistance + chebyshevDistance))

    return 500 * (result + len(foodList))
```