CSC421: Artificial Intelligence
Assignment 1

Rafael Solorzano

May 23, 2018

1 Introduction

This assignment of solving a map problem using uniformed and informed search algorithms. The map is a 2D 100x100 grid with 26 cities labeled 'A' to 'Z'. Each city is randomly placed in a non-repeated place in the grid. Furthermore, for each city the closes 5 cities based on Euclidean distance is calculated from which between 1 and 4 of them are chosen to be connected randomly. Experimentation was run against a bi-directional cities in which a path is added from any missing if it is included in one, and one in which the path is only considered as it was randomly at first, this is further discussed in section 2. An implementation discussion follows in section with the entire code shown and detailed in section 4.

1.1 Formulation of Problem as Search Problem

The map search problem can be formally defined as follows:

- Initial State: any city from the 26 possible cities chosen randomly.
- Action: The only possible actions is moving from one a city to an immediately adjacent one, incurring its cost. Therefore the actions available at one particular state are the immediately adjacent cities to that state.
- Transition Model: The only transition possible is go to city from city. For example In(A) Go(B) will result going from city A to city B, note that is only possible if B is in the adjacent list of A.
- Goal State: any city from the 26 possible cities chosen randomly, Initial State equaling Goal State is allowed.
- Goal Test: Checks if the current state is the goal state.
- Path Cost: The path cost is the euclidean distance between the adjacent cities.
- Solution: The list of cities to travel to reach from initial state to goal state.

The problem is represented using two dictionaries. The first contains a mapping between each city and its X and Y coordinates in the 2D grid. The second dictionaries contain a mapping between each city and another dictionary containing its immediate neighbours with the respective cost incurred by transition to that specific city.

2 Experimental Results

Results obtained from experimentation with the problem generator, uniformed search algorithms and informed search algorithms are presented in this section.

2.1 Problem Generation

The map problem is generated using Python. First, the 26 random locations in the 100x100 2D grid are generated making sure none of them use the same spot in the grid. Second, The euclidean distance for city n and all other cities is calculated, nevertheless only the 5 closest cities for each city are kept in a distance list. Finally, a random number between 1 and 4 is generated, this number will decide how many paths a city will include. Furthermore, the list is shuffled and then between 1 and 4, depending on the random number, cities are removed from its paths. This is done for each city, therefore it can result that city A has a connection to city B but city B did not generate a connection to A during the last processing, therefore to achieve bi-directionality any city that has a connection missing, will have one added when the problem gets turned intro a graph before processing, this achieves an undirected graph. Note that during experimentation both options are tested, that is the tests are ran without forcing bi-directionality and with an undirected graph.

2.1.1 Branches Generated

Random map generation was ran 10, 100 and 1000 times to generate that number of random maps. Each time the average number of paths (branches) is recorded. The results are summarized in Table 1.

Iterations	Average Paths
10	63.7
100	65.29
1000	65.252

Table 1: Average number of Paths (branches) generated

The code that generated this averages can be seen in Figure 1. The description of generate_map function is presented in section 4.

```
#Map branches test for Question 2
paths_10 = []
for i in range(10):
    _,_,total_paths = generate_map(True)
    paths_10.append(total_paths)
print(paths_10)

paths_100 = []
for j in range(100):
    _,_,total_paths = generate_map(True)
    paths_100.append(total_paths)
print(paths_100)

paths_1000 = []
for k in range(1000):
    _,_,total_paths = generate_map(True)
    paths_1000.append(total_paths)
print(paths_1000)

print("Average Number of paths for 10 iterations ")
print(np.mean(paths_10))
print("Average Number of paths for 100 iterations ")
print(np.mean(paths_100))
print("Average Number of paths for 1000 iterations ")
print(np.mean(paths_1000))
```

Figure 1: Code Generation for Branches Averages.

2.2 Uninformed Search Strategies

The results obtained from running Breadth-First Search, Depth-First Search, and Iterative Deepening Search on 100 different instances of a map for each algorithm are presented below. Note, that as previously the tests are ran twice for each algorithm, one with forced bi-directionality and one without.

The result of calling any of the three algorithms mentioned above will be a list of cities to follow from the start state to the goal state. If no such path exists the result will be an empty list. An example of the printed results of both cases are shown in Figure 2 and 3.

```
Attempting to solve from W to T
Solution from BFS
['W', 'Z', 'M', 'U', 'T']
Solution from DFS
['W', 'O', 'Z', 'M', 'U', 'T']
Solution from IDS
['W', 'Z', 'M', 'U', 'T']
```

Figure 2: Result from calling Uninformed Search with a path.

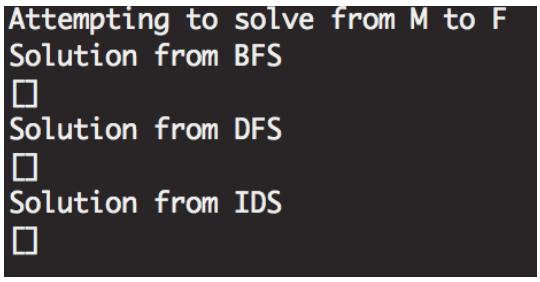

Figure 3: Result from calling Uninformed Search with no solution.

The results from from Figure 2 and 3 are generated by the code shown in Figure 4. The explanation of each function and class shown in the figure is described in Section 4.

```
# ---------- Sample result of one map for each search strategy Question 3------
#Generate cities
cities_locations, cities_distances = generate_map()
#Turn City intro Graph Object
random_map = Graph(cities_distances)
#Generate Random start and goal
start = randint(65,90)
goal = randint(65,90)
#Setup Problem
problem = MapProblem(chr(start),chr(goal),cities_distances,cities_locations)

print("Attempting to solve from " + chr(start) + " to " + chr(goal))

#Print Results for BFS
result_bfs = breadth_first_search(problem)
node, path_back = result_bfs, []
while node:
        path_back.append(node.state)
        node = node.parent
print("Solution from BFS")
print(list(reversed(path_back)))

#Print Results for DFS
result_dfs = depth_first_search(problem)
node, path_back = result_dfs, []
while node:
        path_back.append(node.state)
        node = node.parent
print("Solution from DFS")
print(list(reversed(path_back)))

#Print Results for IDS
result_ids = iterative_deepening_search(problem)
node, path_back = result_ids, []
while node:
        path_back.append(node.state)
        node = node.parent
print("Solution from IDS")
print(list(reversed(path_back)))

#-------------- End Question 3 ----------------------------------------
```

Figure 4: Code to generate the results of Uniformed Search.

The results obtained from running each search strategy including their average space complexity, average time complexity, actual running time (of the total of the 100 iterations), average path length, and numbers of problem solved are summarized in Table 2 and Table 3 below. Table 2 include the results from forced bi-directionality whereas Table 3 include the results from a non bidirectional map.

It was expected that the bidirectional will solve all problems, in some cases it will fail to solve at most 2 out of the 100 test. On the contrary, for the non-bidirectional search it was expected that not all maps will have solutions and as it can be seen from Table 3 only about 65 maps were solvable. Additionally, for the Iterative Deepening Search it took about 25 seconds to run all the 100 iterations, as it had to dig deeper each time it went through the iteration, the limit was set to 13 as anything above 15 will take a considerable amount of time to solve.

Search	Avg Space Com	Avg Time Com	Runtime	Avg Path Length	Solved
BFS	201	4.85	3.55s	3	100
DFS	315	12.52	3.59s	10	100
IDS	924	3.18	3.57s	3	100

Table 2: Results from running BFS, DFS and IDS on bi-directional map.

Search	Avg Space Com	Avg Time Com	Runtime	Avg Path Length	Solved
BFS	80	6.20	3.54s	4.03	63
DFS	106	8.18	3.60s	6.2	65
IDS	5541038	2.68	24.48s	4	67

Table 3: Results from running BFS, DFS and IDS on a non bi-directional map.

The description of each statistic and how it was calculated follows. The average space complexity was calculated simply as the max number of nodes generate between all 100 iterations of the problem. Furthermore, the average time complexity was calculated as the mean of all different number of nodes generated in the 100 problems. The running time was calculated using the Python time library. Finally, the average path length is the average of all the different length of solutions generated.

An example of how the data was analyzed for the BFS case is shown in Figure 5 below. All the search strategies follow an almost identical strategy.

```
#-------------Statistic Generation for BFS ---------------------
start_time_bfs = time.time()
nodes_generated = []
nodes_visited = []
path_lengths = []
problems_solved = 0
for i in range(100):
    #Generate cities
    cities_locations, cities_distances = generate_map()
    #Turn City intro Graph Object
    random_map = Graph(cities_distances,True)
    #Generate Random start and goal
    start = randint(65,90)
    goal = randint(65,90)
    #Setup Problem
    problem = MapProblem(chr(start),chr(goal),cities_distances,cities_locations)

    print("Attempting to solve from " + chr(start) + " to " + chr(goal))

    if breadth_first_search(problem) is not None:
        solution, explored, nodes_gen = breadth_first_search(problem,stats=True)
    else:
        continue

    node = solution
    solution_length = 0
    while node:
        solution_length+=1
        node = node.parent

    if solution_length != 0:
        problems_solved+=1
        path_lengths.append(solution_length)
    nodes_generated.append(nodes_gen)
    nodes_visited.append(len(explored))


print("Average Space Complexity (max nodes generated) %f" %np.max(nodes_generated))
print("Average Time Complexity (nodes visited) %f" %np.mean(nodes_visited))
print("Total BFS Execution Time %s" % (time.time() - start_time_bfs))
print("Average path length %f" %np.mean(path_lengths))
print("Total problems solved %f" %problems_solved)

#------------ End Static Generation for BFS ---------------------------------------
```

Figure 5: Code generation for BFS stats.

2.3 Informed Search Strategies

This section presents the results obtained from running greedy best-first search as well as A* search on 100 different instances of a map for each search strategy. As before, two different instances of each 100 maps is ran. Furthermore, the three heuristics used for the informed search strategies are detailed.

2.3.1 Search Heuristics

The search heuristics used for informed searching are:

- he: straight-line Euclidean distance to goal state
- h0: just the literal value '0'
- hm: Manhattan Distance to goal state.

The heuristics functions are acceptable as they each provide an estimated cost from the current state to the goal state based on different calculations. he provides a heuristic based on the straight line Euclidean-distance from the current state to the goal state which makes it admissible as it helps choose the best strategy to get to the goal with the fewest cost possible. Similarly, the hm which uses the Manhattan Distance is also admissible for the same reason as it will just provide an informed cost based on the sum of horizontal and vertical distances to the goal state. Finally, the h0 simply states that the informed cost to reach the goal is simply 0, therefore this heuristic is acceptable as it is nonnegative, it will satisfy the constraint that h(n) if n is a goal node = 0, and it is essentially just want to find a solution without regarding cost.

In terms of dominance two different results were obtained according to the search function being used. For Greedy best-first search the dominance is shown to be he ¿ hm ¿ h0 where he is the dominating one. On the contrary, for the A* search the dominance is shown to be hm ¿ he ¿ h0 where hm is the dominating one. The results of the experiments that led to this conclusion are shown in section 2.3.2 below.

2.3.2 Statistics

The results obtained from running each search strategy include the same statistics as with uniformed case. The results are summarized in Table 4 and 5 below. Table 4 include the results from forced bi-directionality whereas Table 5 include the results from non bidirectional map.

The results found where interesting, as it was shown that the dominance of a heuristic function changed depending on the search algorithm used. As A* incorporates the added path cost to its heuristic function its dominant heuristic was the hm which calculates the Manhattan Distance, on the other hand Greedy Best first search uses only the heuristic function, for which the most dominant was the he which calculates the euclidean distance. Finally, as it was expected the worst heuristic function h0, resulted in a large space and time complexity for both cases, this was expected mainly as it simply was not informing anything and just only cared about finding a solution.

Search Type	Avg Space Com	Avg Time Com	Runtime	Avg Path Length	Solved
GBFS he	92	2.67	3.57s	3.42	100
GBFS h0	753	11.91	3.79s	5.01	100
GBFS hm	117	2.78	3.55s	3.41	100
A* he	82	3.73	3.69s	4.0	100
A* h0	503	12.64	3.62s	6.64	100
A* hm	68	2.77	3.48s	3.36	100

Table 4: Informed Search results on bi-directional map.

Search Type	Avg Space Com	Avg Time Com	Runtime	Avg Path Length	Solved
GBFS he	46	3.93	3.66s	4.30	59
GBFS h0	106	8.68	3.49s	4.95	70
GBFS hm	71	3.68	3.63s	3.94	57
A* he	67	5.11	3.62s	4.73	63
A* h0	95	9.14	3.43s	5.10	64
A* hm	61	4.22	3.56s	4.21	75

Table 5: Informed Search results on non bi-directional map.

The statistics in the tables above were calculated in the same way as in the uniformed case.

3 Implementation Overview

The code created for this assignment is written entirely in Python. The code is ordered by first defining global variables, classes, functions, and the use of all the aforementioned. The code contain three classes:

- MapProblem which creates the problem, this class includes initial state, goal state, cities distances, cities locations and cities locations. Additionally it includes functions for goal test, path cost, action and results, and if needed heuristic functions.
- Graph this class entire purpose is to hold the map in an object, and turn into an undirected graph if requested
- Node the class that allows navigation in the search, it includes state, parent, action, cost, and depth. Additionally, it includes the functions for expanding a node and generating the child nodes.

The rest of the code is contained outside classes and is distributed as follows. Map generation is done from a function with an option parameter to return how many paths were generated, this is to report average paths as required by this assignment. Each search algorithm is implemented as a function that takes a problem object, additionally it contains an optional parameter to return stats for this assignment. All the stats and execution of the search algorithms are performed after the entire declaration of classes and functions. Some screenshots of this code can be observed in Figures 1, 4, and 5 of this report.

The code uses data structures provided natively by Python as well as the PriorityQueue and Memoize as provided by the AIMA book, this data structures are stored in a separate utils.py

file and are a exact copy of the AIMA books implementation. Furthermore, for the calculation of Euclidean and Manhattan distance the distance module from scipy.spatial is used. Additionally, to calculate averages and max in lists numpy is used. Finally, all the randomization is done by the randint and shuffle modules from the random Python package.

4 Code Listing

The following section describes the code along with accompanying screenshots. First, the three classes used are shown. Second, each search algorithm is presented. Finally, the utils used in the code are shown. The code for testing is repetitive and is not shown in this section, it follows the same format as shown in Figure 5 and 4.

4.1 Classes

The code for all the classes used in the code is shown in this section. This code is based on the AIMA book code repository for Python.

Figure 6 shows the MapProblem which is used as the base problem for this assignment. As it was mentioned before it includes initial state, goal state, cities distances, cities locations and a heuristic function. The heuristic function defaults to he, but it can be changed to h0 or hm if specifying heuristic z or m respectively. Furthermore, the class includes the functions to check if the desired goal has been reached, this is done by directly comparing the string of state with goal state. Additionally, the path cost is simply computed from the dictionary of cities distances, the function takes two cities and will return the cost of travel between both of them. The actions function will return the possible cities to travel to from the current state, as that is the only possible thing to do in this problem. The result function, will yield what would be result if traveling to a city (action) from the current state, this function is unused in the entire code but was left for completeness. Finally, the heuristic functions do as described in section 2, with the only not described being h_g, this function is only used by A and it has the difference of returning the result of the current heuristic function plus the cost of the path cost of the node.

Figure 7 shows the Graph class, this class only purpose is to turn the map into an object. Additionally, the Graph class provides an option of forcing bi-directionality in every city by making it an undirected graph. The way it works is for example if A has a path to Z of cost 11, but Z may not have that path in its internal dictionary, therefore the code will loop over every city and add it if its not there. In this case as A is the first city in the dictionary it will loop over all its neighbours, and loop on them to check their paths, if any is missing it will add them. This is achieved by using the setdefault Python function.

Figure 8 shows the Node class, this class is very important in this program as it is the class that allows navigation in the graph. It contains all necessary information to traverse, state, parent, action, path_cost, and depth. The class only has expand and child node functions which are interconnected, as only expand is called in the code which in turn calls child node which is the function to generate the child node and populate it accordingly.

```python
class MapProblem:
    def __init__(self, initial, goal, cities_distances, cities_locations, heuristic='e'):
        self.initial = initial
        self.goal = goal
        self.cities_distances=cities_distances
        self.cities_locations=cities_locations
        #Set default h to he
        self.h = self.h_e
        #change heuristic accordingly
        if heuristic == 'e':
            self.h = self.h_e
        elif heuristic == 'z':
            self.h = self.h_zero
        elif heuristic == 'm':
            self.h = self.h_m

    def goal_test(self, state):
        return state == self.goal

    def path_cost(self,city1,city2):
        return self.cities_distances[city1][city2]

    def actions(self, state):
        #The possible actions is go to any city that has a path
        #in one hop.
        possible_actions = list(cities_distances[state].keys())
        return possible_actions


    def result(self,state,action):
        #result of action would be new city based on current state
        new_state = cities_distances[state][action]
        return new_state

    def h_e(self,node):
        return round(distance.euclidean(self.cities_locations[node.state],self.cities_locations[self.goal]),2)

    def h_zero(self,node):
        return 0

    def h_m(self,node):
        return round(distance.cityblock(self.cities_locations[node.state],self.cities_locations[self.goal]),2)

    #Used for A* Search computes h(n) + g(n)
    def h_g(self,node):
        return (self.h(node) + node.path_cost)
```

Figure 6: The MapProblem class.

```python
class Graph:

    def __init__(self, cities_distances=None,undirected=False):
        self.cities_distances = cities_distances or {}
        if undirected:
            self.make_undirected()

    def make_undirected(self):
        for city in self.cities_distances:
            for (toCity, dist) in self.cities_distances[city].items():
                self.cities_distances.setdefault(toCity,{})[city] = dist
```

Figure 7: The Graph class.

```python
class Node:

    def __init__(self, state, parent=None, action=None, path_cost=0):
        self.state=state
        self.parent=parent
        self.action=action
        self.path_cost=path_cost
        self.depth=0
        if parent:
            self.depth = parent.depth+1

    def __lt__(self,node):
        return self.state < node.state

    def expand(self,problem):
        #print(self.state)
        #print(list(problem.cities_distances[self.state].keys()))
        return [self.child_node(problem,action) for action in problem.actions(self.state)]
        #keys = list(cities_distances[city].keys())
        #return keys

    def child_node(self,problem,action):
        #next_node = problem.cities_distances[self.state]
        #print(self.state)
        #print(action)
        return Node(action,self,None,problem.path_cost(self.state,action))
```

Figure 8: The Node class.

4.2 Map Generation

```python
def generate_map(report_paths=False):

    '''
    Generate random location for Cities in a 100x100 grid
    '''
    total_paths = 0

    cities_locations = dict()
    curr_city = ord('A')

    seen=set()
    x, y = randint(0,99), randint(0,99)
    cities_locations[chr(curr_city)] = (x,y)
    for i in range (26):
        seen.add((x,y))
        x, y = randint(0,99), randint(0,99)
        while (x,y) in seen:
            x, y = randint(0,99), randint(0,99)
        cities_locations[chr(curr_city)] = (x,y)
        curr_city += 1
```

Figure 9: Initial Map Generation Function, which generates X and Y coordinates.

Figure 9 shows the initial generate_map function, it provides an optional report_paths which will include an extra return with the number of paths generated, this is added due to the requirements of this assignment. The first thing done is generate 26 random cities, and their corresponding X and Y positions. This is achieved with pure Python and the randint module from random.

Figure 10 shows the euclidean generation of each city to its closes 5 neighbours based on it. This part starts with a tuple of lists which contain the city along its cost, it is initially instantiated to 999 as to be replaced later with the 5 closes paths. The result of this will be each city will contain an ordered list in ascending of its closest 5 cities with its corresponding cost.

Figure 11 builds upon the result from Figure 10, this code will randomly decide how many paths of the 5 the city will keep, this is done by first generating a number between 1 and 4 (as the max path a city can have is 4), and furthermore shuffling the keys as to keep randomization. This shuffled keys are then iterated over and some of them are removed according to the number of paths. Finally, the return of the function can provide 2 or 3 results depending on the optional report paths seen in Figure 9.

```
'''
Calculate Euclidean Distances between cities and keep 5 closest
'''
distances = (['city',999], ['city',999], ['city',999], ['city',999], ['city',999])
#print(distance.euclidean(cities_locations['A'], cities_locations['B']))
cities_distances=dict()
for city1 in cities_locations:
    cities_distances[city1] = dict()
    for city2 in cities_locations:
            for i in range(len(distances)):
                euclidean = distance.euclidean(cities_locations[city1],cities_locations[city2])
                if euclidean == 0:
                    break
                if euclidean < distances[i][1]:
                    distances[i][0] = city2
                    distances[i][1] = round(euclidean,2)
                    break
    #print(distances)
    for i in range(len(distances)):
        cities_distances[city1][distances[i][0]] = distances[i][1]
    distances = (['city',999], ['city',999], ['city',999], ['city',999], ['city',999])
```

Figure 10: Map Generation Function, calculates Euclidean distance and keep 5 closest.

```
'''
Randomly decide between having 1 or 4 paths between cities
and randomly choose the paths
'''
for city in cities_locations:
    paths = randint(1,4)
    total_paths += paths
    keys = list(cities_distances[city].keys())
    shuffle(keys)
    for i in range(len(keys)-paths):
        cities_distances[city].pop(keys[i],None)
#print(cities_distances)
if report_paths:
    return (cities_locations,cities_distances,total_paths)
return (cities_locations,cities_distances)
```

Figure 11: Map Generation Function, decide how many paths each city will have.

4.3 Algorithms

All the algorithms presented in this section are based of the AIMA book Python code repository. They have been modified to add statistics required for this assignment and some to fit with the specific problem being solved.

```
def breadth_first_search(problem,stats=False):
    node = Node(problem.initial)
    explored = set()
    nodes_generated = 1 #the initial state
    if problem.goal_test(node.state):
        if stats:
            return (node,explored,nodes_generated)
        return node
    frontier = deque([node])
    while frontier:
        node = frontier.popleft()
        explored.add(node.state)
        for child in node.expand(problem):
            nodes_generated+= 1
            if child.state not in explored and child not in frontier:
                if problem.goal_test(child.state):
                    if stats:
                        return (child,explored,nodes_generated)
                    return child
                frontier.append(child)
    return None
```

Figure 12: Breadth-first Search algorithm.

Figure 12 and 13 presents Breadth-first Search and Depth-first search algorithms respectively. The code includes an option stats parameter to not only return the node which contains the solution, when found but also all the explored nodes, and the number of nodes generated during the execution of the code.

Figure 14 includes the code used for Iterative Deepening Search, this code uses a recursive Depth Limited Search in order to find a solution. It will iterate over a fixed depth, each time increasing by one its limit, if a solution is found the code will stop and the solution will be returned. The depth was chosen to be 13 as anything above 15 will take a considerable amount of time for a problem without a solution, the initial testing was done with 99 but in a problem with no solution the code never terminates. To keep statistics for this code as the code is recursive global variables were used to keep tracks of the visited and generated nodes along with it, these results are later used when generating the statistics of the code very similar to Figure 5.

```
def depth_first_search(problem,stats=False):
    frontier = [(Node(problem.initial))]
    explored = set()
    nodes_generated = 1
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            if stats:
                return (node,explored,nodes_generated)
            return node
        explored.add(node.state)
        for child in node.expand(problem):
            nodes_generated+=1
        frontier.extend(child for child in node.expand(problem)
                        if child.state not in explored and
                        child not in frontier)
    return None
```

Figure 13: Depth-first Search algorithm.

```python
def recursive_dls(node,problem,limit):
    global ids_visited
    global ids_generated
    if problem.goal_test(node.state):
        ids_visited+=1
        return node
    elif limit == 0:
        return 0
    else:
        cutoff_occured = False
        for child in node.expand(problem):
            ids_generated += 1
            result = recursive_dls(child,problem,limit-1)
            if result == 0:
                cutoff_occured = True
            elif result is not None:
                ids_visited+=1
                return result
        return 0 if cutoff_occured else None

def depth_limited_search(problem,limit):
    return recursive_dls(Node(problem.initial),problem,limit)

def iterative_deepening_search(problem):
    for depth in range(13):
        result = depth_limited_search(problem,depth)
        if result != 0:
            return result
```

Figure 14: Iterative Deepening Search algorithm.

```
def greedy_best_first_search(problem,stats=False):
    h = memoize(problem.h, 'h')
    node = Node(problem.initial)
    nodes_generated = 1
    explored = set()
    if problem.goal_test(node.state):
        if stats:
            return (node,explored,nodes_generated)
        return node
    frontier = PriorityQueue('min', h)
    frontier.append(node)
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            if stats:
                return (node,explored,nodes_generated)
            return node
        explored.add(node.state)
        for child in node.expand(problem):
            nodes_generated += 1
            if child.state not in explored and child not in frontier:
                frontier.append(child)
            elif child in frontier:
                incumbent = frontier[child]
                if h(child) < h(incumbent):
                    del frontier[incumbent]
                    frontier.append(child)
    return None
```

Figure 15: Greedy best-first search algorithm.

Figure 15 and 16 include the code for Greedy best-first search and A* respectively. Greedy best first search uses a memoize and PriorityQueue which are presented in the following subsection. They use heuristic functions, for their informed searching. With the difference in A* search being that is will use problem.h_g which is a special function described above that will compute the sum of the heuristic function and the path cost of the current node and will use this combination as its heuristic function. Aside from that, the code is identical. Furthermore, as before the code includes the optional stats which will return in addition to the node with the solution path, all the explored nodes, and the number of nodes generated during its execution.

```
'''
Same algorithm as GBFS, with the difference that uses the
h_g function in the problem class instead of h
h_g adds the path cost to the selected heuristic
'''
def a_star_search(problem,stats=False):
    h = memoize(problem.h_g, 'h')
    node = Node(problem.initial)
    nodes_generated = 1
    explored = set()
    if problem.goal_test(node.state):
        if stats:
            return (node,explored,nodes_generated)
        return node
    frontier = PriorityQueue('min', h)
    frontier.append(node)
    while frontier:
        node = frontier.pop()
        if problem.goal_test(node.state):
            if stats:
                return (node,explored,nodes_generated)
            return node
        explored.add(node.state)
        for child in node.expand(problem):
            nodes_generated += 1
            if child.state not in explored and child not in frontier:
                frontier.append(child)
            elif child in frontier:
                incumbent = frontier[child]
                if h(child) < h(incumbent):
                    del frontier[incumbent]
                    frontier.append(child)
    return None
```

Figure 16: A* search algorithm.

4.4 Utils

The following two Figures 17 and 18 present the memoize and PriorityQueue implementations. This code is entirely from the AIMA book Python repository and are used only for the Greedy best-first search and A*. Memoization allows for a function to be remembered with a name for the entire execution of a function, in this case it is used to bind h to the corresponding heuristic function assigned in the Problem class. The PriorityQueue is used as a min queue, on both Greedy best-first search and A* search.

```
#_____ Memoization _____
'''
Memoization, From AIMA Python Github Repository utils.py
Lines 367 -384
'''

def memoize(fn, slot=None, maxsize=32):
    """Memoize fn: make it remember the computed value for any argument list.
    If slot is specified, store result in that slot of first argument.
    If slot is false, use lru_cache for caching the values."""
    if slot:
        def memoized_fn(obj, *args):
            if hasattr(obj, slot):
                return getattr(obj, slot)
            else:
                val = fn(obj, *args)
                setattr(obj, slot, val)
                return val
    else:
        @functools.lru_cache(maxsize=maxsize)
        def memoized_fn(*args):
            return fn(*args)

    return memoized_fn

#_____ Memoization Ends _____
```

Figure 17: Memoization code.

```python
#-------- PriorityQueue ----------------
'''
PriorityQueue from AIMA Python Github Repository utils.py
Lines 688-739
'''
class PriorityQueue:
    """A Queue in which the minimum (or maximum) element (as determined by f and
    order) is returned first.
    If order is 'min', the item with minimum f(x) is
    returned first; if order is 'max', then it is the item with maximum f(x).
    Also supports dict-like lookup."""

    def __init__(self, order='min', f=lambda x: x):
        self.heap = []

        if order == 'min':
            self.f = f
        elif order == 'max':  # now item with max f(x)
            self.f = lambda x: -f(x)  # will be popped first
        else:
            raise ValueError("order must be either 'min' or max'.")

    def append(self, item):
        """Insert item at its correct position."""
        heapq.heappush(self.heap, (self.f(item), item))

    def extend(self, items):
        """Insert each item in items at its correct position."""
        for item in items:
            self.heap.append(item)

    def pop(self):
        """Pop and return the item (with min or max f(x) value
        depending on the order."""
        if self.heap:
            return heapq.heappop(self.heap)[1]
        else:
            raise Exception('Trying to pop from empty PriorityQueue.')

    def __len__(self):
        """Return current capacity of PriorityQueue."""
        return len(self.heap)

    def __contains__(self, item):
        """Return True if item in PriorityQueue."""
        return (self.f(item), item) in self.heap

    def __getitem__(self, key):
        for _, item in self.heap:
            if item == key:
                return item

    def __delitem__(self, key):
        """Delete the first occurrence of key."""
        self.heap.remove((self.f(key), key))
        heapq.heapify(self.heap)

#--------- End Priority Queue --------------------
```

Figure 18: PriorityQueue code.