

Problem 1

Part 2

Breadth-First-Search

Grid #	Total Nodes Generated	Max Nodes Stored At Once	Number of Iterations	Depth of Goal	Cost of path to goal	Length of path to goal
1	71	7	25	7	17	8
2	34	2	15	10	30	11
3	234	7	83	31	77	32
4	331	8	100	27	60	28
5	189	5	79	44	104	45
Total	859	29	302			

Depth-First-Search

Grid #	Total Nodes Generated	Max Nodes Stored At Once	Number of Iterations	Depth of Goal	Cost of path to goal	Length of path to goal
1	25	6	10	9	21	10
2	25	5	11	10	30	11
3	101	21	38	35	89	36
4	262	28	82	41	90	42
5	165	18	69	52	128	53
Total	578	78	210			

Uniform-Cost-Search

Grid #	Total Nodes Generated	Max Nodes Stored At Once	Number of Iterations	Depth of Goal	Cost of path to goal	Length of path to goal
1	54	8	19	7	17	8
2	34	2	15	10	30	11
3	164	11	61	31	77	32
4	281	22	84	27	60	28
5	195	7	82	44	104	45
Total	728	50	261			

A*-Search: Manhattan Heuristic

Robin Mehta
Artificial Intelligence, Homework 2

Grid #	Total Nodes Generated	Max Nodes Stored At Once	Number of Iterations	Depth of Goal	Cost of path to goal	Length of path to goal
1	39	8	14	7	17	8
2	29	2	13	10	30	11
3	150	11	56	35	89	36
4	109	20	39	27	60	28
5	128	10	54	44	104	45
Total	455	51	176			

A*-Search: Euclidean Heuristic

Grid #	Total Nodes Generated	Max Nodes Stored At Once	Number of Iterations	Depth of Goal	Cost of path to goal	Length of path to goal
1	31	7	12	11	27	12
2	29	2	13	10	30	11
3	149	13	56	35	89	36
4	119	21	42	27	60	28
5	126	10	53	44	104	45
Total	454	53	176			

Iterative-Deepening-Search

Grid #	Total Nodes Generated	Max Nodes Stored At Once	Number of Iterations	Depth of Goal	Cost of path to goal	Length of path to goal
1	25	6	10	9	21	10
2	25	5	11	10	30	11
3	101	21	38	35	89	36
4	262	28	82	41	90	42
5	165	18	69	52	128	53
Total	578	78	210			

2. Which Search Algorithm

- Fewest total nodes: A* Euclidean Heuristic search, just beat A* Manhattan Heuristic Search by 1 node.
- Stored fewest nodes at one time: Breadth First Search
- A* Search Algorithms

3. The informed search algorithms did much better than the uninformed because they had a better understanding of their environments, and were able to more optimally choose successor nodes based on this knowledge. This let the algorithm choose optimal paths before choosing sub-optimal paths. Having no knowledge of an environment creates the necessity to check all possibilities for successor nodes, which can inefficient either with time or space complexity.

Part 4:

1. Local search does not do well in 8-puzzle, as only 10-15 puzzles were solved out of 25000 generated. 8-puzzle creates scenarios where it might be beneficial to make a move away from the goal in order to find a solution – otherwise, it can get stuck at any local maxima. 8-queens works well with local search because it can continue placing queens until it finds a goal state. In 8-puzzle, trying to improve the search space configuration is tricky and often plateaus without any major improvement.

2. Simulated annealing can be better than local-search because it can probabilistically infer successor moves based on the search space, which can use a function to provide more information than just the current configuration. Random Restart Hill Climbing would also work better because once it reaches a local maxima, unlike a regular hill-climbing algorithm, random restart allows for a new starting point, rather than staying stuck.

Problem 2

Part 1:

1.
 - a) A B C D E F
 - b) A D C F E B
 - c) A D C F E B (assuming $d = 1$, until algorithm stops at $d = 3$)
 - d) A D B C E F
 - e) A C F E B D
 - f) A C E F B D
2.
 - a) Creating a $4 \times 4 \times 2$ boolean array to mark states visited creates 32 representable states.
Example: Visit(true, 1, 2) marks that 1 cannibal and 2 missionaries on the right (goal) side of the bank has occurred so far.
 - b) 32 representable states
 - c) Initial state: Visit(true, 0, 0), indicating that there are 0 cannibals and 0 missionaries on the left side of the bank at the start.
 - d) Visit(true, 3, 3), indicating the goal state of 3 cannibals and 3 missionaries on the right side of the bank.

e) Potential successors of an arbitrary state will include all possibilities of (false, cannibal+1, or missionary+1). A transition will be defined by marking one of these potential successor states as true.

f)

Visit(true, 0, 0)

Visit(true, 2, 0)

Visit(true, 1, 0)

Visit(true, 3, 0)

Visit(true, 2, 0)

Visit(true, 2, 2)

Visit(true, 1, 1)

Visit(true, 1, 3)

Visit(true, 0, 3)

Visit(true, 2, 3)

Visit(true, 3, 3)

3.

Let $k(n)$ denote the least expensive path from a (starting node) to the goal node.

Prove: Every consistent heuristic is admissible, i.e. $h(n) \leq k(n)$.

Base Case: if starting point is the goal, then $h(n) = 0 \leq k(n)$.

Inductive Step: if the starting point to the goal consists of i steps, the best path from n to n' (successor node) has $i-1$ steps. This gives us a proof by consistency: $h(n') \leq k(n')$.

$h(n) \leq c(n, a, n') + h(n') \leq c(n, a, n') + k(n') = k(n)$.

4. Local Search in general

a) Two possible advantages of local search over graph / tree search are that it is very space efficient, only saving 1 node in memory at a time, and always improving, which allows it to often solve large, continuous problems (in addition to optimization problems). Two possible disadvantages of local search vs. graph / tree search are that when you hit a maxima or plateau, there's no better successor. Additionally, there's no clear answer on how often to restart or try to "repair" by choosing successor moves randomly (Lecture, Slide 13).

b) Hill-climbing search is a loop that continually move in direction of increasing value, and terminates when reaching a peak. Genetic algorithms start with k randomly generated states, represent each state as a string, and rate each state using an objective function (Lecture 6, Slide 21). Genetic algorithms are different than Hill-climbing algorithms, because they have an uphill tendency but not expectation, they explore randomly (mutations occur), and there is an exchange of information across parallel search threads.

c) Simulated annealing search picks a random move and accepts with a probability if it improves. Random restart searches independently, without passing information about probability.

5.

a)

```
if L:
    pole = NudgeRight()
else:
    pole = NudgeLeft()
# if initial state is V, do nothing.
```

b)

```
while(pole != vertical):
    # Establishes a new state for the pole after being nudged.
    randomNudge = randint(0, 1)
    if randomNudge == 0:
        pole = NudgeLeft
    else:
        pole = NudgeRight
```

Problem 3: Adversarial Search

Minimax: minimize the possible loss for a worst-case scenario.

B(4)
C(3)
D(7)
A(7)
The leaf nodes are the same.

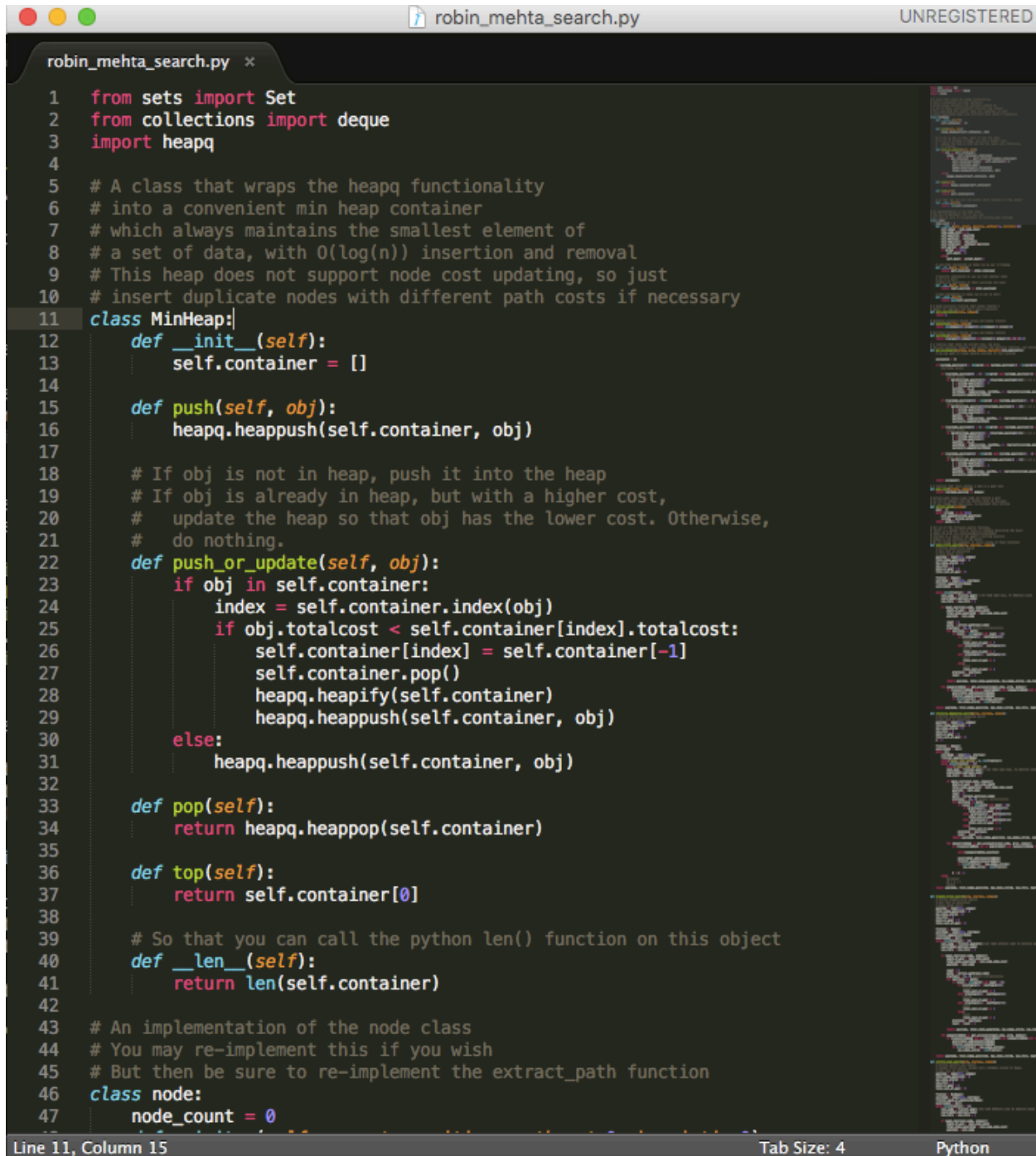
If we are using alpha-beta pruning left to right, we can skip child-nodes of D, which are: K, L, and M.

Apply minimax to the tree with chance nodes:

B(10)
C(20)
Chance node: 15

D(30)
E(20)
Chance node: 22

Problem 1, Part 1: Appended code



```
1  from sets import Set
2  from collections import deque
3  import heapq
4
5  # A class that wraps the heapq functionality
6  # into a convenient min heap container
7  # which always maintains the smallest element of
8  # a set of data, with O(log(n)) insertion and removal
9  # This heap does not support node cost updating, so just
10 # insert duplicate nodes with different path costs if necessary
11 class MinHeap:
12     def __init__(self):
13         self.container = []
14
15     def push(self, obj):
16         heapq.heappush(self.container, obj)
17
18     # If obj is not in heap, push it into the heap
19     # If obj is already in heap, but with a higher cost,
20     # update the heap so that obj has the lower cost. Otherwise,
21     # do nothing.
22     def push_or_update(self, obj):
23         if obj in self.container:
24             index = self.container.index(obj)
25             if obj.totalcost < self.container[index].totalcost:
26                 self.container[index] = self.container[-1]
27                 self.container.pop()
28                 heapq.heapify(self.container)
29                 heapq.heappush(self.container, obj)
30         else:
31             heapq.heappush(self.container, obj)
32
33     def pop(self):
34         return heapq.heappop(self.container)
35
36     def top(self):
37         return self.container[0]
38
39     # So that you can call the python len() function on this object
40     def __len__(self):
41         return len(self.container)
42
43     # An implementation of the node class
44     # You may re-implement this if you wish
45     # But then be sure to re-implement the extract_path function
46     class node:
47         node_count = 0
```

Line 11, Column 15

Tab Size: 4

Python

Robin Mehta
Artificial Intelligence, Homework 2



```
47     node_count = 0
48     def __init__(self, parent, position, pathcost=0, heuristic=0):
49         self.idnum = node.node_count
50         node.node_count += 1
51         self.position = position
52         self.pathcost = pathcost
53         self.heuristic = heuristic
54         self.totalcost = pathcost+heuristic
55         self.parent = parent
56         if parent is None:
57             self.depth = 0
58         else:
59             self.depth = parent.depth+1
60
61     # Less-than implemented so nodes can be put in MinHeap
62     def __lt__(self, other):
63         return self.totalcost < other.totalcost
64
65     # Equality implemented so you can test whether nodes
66     # are in a Set()
67     # Nodes are equivalent if their positions are equal
68     def __eq__(self, other):
69         return (self.position == other.position)
70
71     # Hash implemented so nodes can be put in Set()
72     def __hash__(self):
73         return hash(self.position)
74
75     # A dummy heuristic function that always returns 0
76     # Useful for searches that don't have a heuristic
77     def zero_heuristic(curpos, endpos):
78         return 0
79
80     # Manhattan distance between curpos and endpos (tuples)
81     def manhattan(curpos, endpos):
82         return abs(endpos[0]-curpos[0])+abs(endpos[1]-curpos[1])
83
84     # Euclidean distance between curpos and endpos (tuples)
85     def euclidean(curpos, endpos):
86         return ((curpos[0]-endpos[0])**2+(curpos[1]-endpos[1])**2)**(0.5)
87
88     # A function that takes the current node, the grid,
89     # The goal position (endpos), and possibly the heuristic function, and returns a
90     def get_successors(curnode, grid, endpos, heuristic=zero_heuristic):
91         # You may want to create several versions of this function
92
93         successors = []
```

Line 11, Column 15

Tab Size: 4

Python

Robin Mehta
Artificial Intelligence, Homework 2

```
robin_mehta_search.py x
95     if (curnode.position[0] < len(grid) and curnode.position[1] < len(grid[0])):
96         #curnode exists
97
98         if ((curnode.position[0] - 1) < len(grid) and (curnode.position[1] < len(
99             #north exists
100             if (grid[curnode.position[0] - 1][curnode.position[1]]): # not a wall
101                 i = curnode.position[0] - 1
102                 j = curnode.position[1]
103                 northPos = (i,j)
104                 northNode = node(curnode, northPos, 1 + heuristic(curnode.position,
105                     successors.append(northNode)
106
107         if ((curnode.position[0]) < len(grid) and (curnode.position[1] + 1) < len(
108             #east exists
109             if (grid[curnode.position[0]][curnode.position[1] + 1]): # not a wall
110                 i = curnode.position[0]
111                 j = curnode.position[1] + 1
112                 eastPos = (i,j)
113                 eastNode = node(curnode, eastPos, 2 + heuristic(curnode.position,
114                     successors.append(eastNode)
115
116         if ((curnode.position[0] + 1) < len(grid) and (curnode.position[1] < len(
117             #south exists
118             if (grid[curnode.position[0] + 1][curnode.position[1]]): # not a wall
119                 i = curnode.position[0] + 1
120                 j = curnode.position[1]
121                 southPos = (i,j)
122                 southNode = node(curnode, southPos, 3 + heuristic(curnode.position,
123                     successors.append(southNode)
124
125         if ((curnode.position[0]) < len(grid) and (curnode.position[1] - 1) < len(
126             #west exists
127             if (grid[curnode.position[0]][curnode.position[1] - 1]): # not a wall
128                 i = curnode.position[0]
129                 j = curnode.position[1] - 1
130                 westPos = (i,j)
131                 westNode = node(curnode, westPos, 4 + heuristic(curnode.position,
132                     successors.append(westNode)
133
134     return successors
135
136     # A function that tests whether a node is a goal node
137     def goal_test(curnode, endpos):
138         return (curnode.position == endpos)
139
140     # extract_path takes a goal node and returns a path
141     # (a list of tuples) from the initial state to the goal
```

Line 11, Column 15 Tab Size: 4 Python

Robin Mehta
Artificial Intelligence, Homework 2

```
robin_mehta_search.py x
142 # If you change the node class, re-implement this function
143 def extract_path(curnode):
144     path = []
145     while curnode is not None:
146         path.append(curnode.position)
147         curnode = curnode.parent
148     return path[::-1]
149
150 # For all of the following search functions,
151 # "grid" is a python list of lists of integers describing the board
152 # Index the grid via grid[rowindex][columnindex]
153 # startpos is a tuple of the agent's starting position
154 # endpos is the location of the batter
155 # Do not change the arguments or return values of these functions
156 def depth_first_search(grid, startpos, endpos):
157     # Perform depth-first search
158     # Goal test at generation
159     # Stack based search
160     goalnode = node(None, endpos)
161     total_nodes_generated = 0
162     max_nodes_stored = 0
163     num_iters = 0
164     depth_of_goal = 0
165     total_cost_of_goal = 0
166
167     frontier = deque()
168     startNode = node(None, startpos)
169     frontier.append(startNode)
170     exploredSet = Set()
171
172     while (len(frontier) > 0):
173         next_node = frontier.pop() # POP FROM SAME SIDE, TO IMITATE STACK
174         exploredSet.add(next_node)
175         num_iters = num_iters + 1
176
177         if (goal_test(next_node, endpos)):
178             depth_of_goal = next_node.depth
179             total_nodes_generated = next_node.node_count
180             goalnode = next_node
181
182             count = 0
183             paths = extract_path(next_node)
184             prevTuple = (0, 0) #safety initialization
185             for nextTuple in paths:
186                 if (count < len(paths) and count > 0):
187                     if (prevTuple[0] < nextTuple[0]):
188                         #south
```

Robin Mehta
Artificial Intelligence, Homework 2

```
robin_mehta_search.py x
189         total_cost_of_goal += 3
190     elif (prevTuple[0] > nextTuple[0]):
191         #north
192         total_cost_of_goal += 1
193     elif (prevTuple[1] < nextTuple[1]):
194         #east
195         total_cost_of_goal += 2
196     else:
197         #west
198         total_cost_of_goal += 4
199     prevTuple = nextTuple
200     count = count + 1
201
202     return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
203
204     for unexploredNode in get_successors(next_node, grid, endpos):
205         if ((unexploredNode not in exploredSet) and (unexploredNode not in frontier)):
206             exploredSet.add(unexploredNode)
207             frontier.append(unexploredNode)
208             if (len(frontier) > max_nodes_stored):
209                 max_nodes_stored = len(frontier)
210
211     return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
212
213 def iterative_deepening_search(grid, startpos, endpos):
214     # Perform iterative deepening search
215     # Goal test at generation
216     goalnode = node(None, endpos)
217     total_nodes_generated = 0
218     max_nodes_stored = 0
219     num_iters = 0
220     depth_of_goal = 0
221     total_cost_of_goal = 0
222     d = 1
223
224     frontier = deque()
225     exploredSet = Set()
226     while (d):
227         startNode = node(None, startpos)
228         frontier.append(startNode)
229         print('outer loop called', d, len(frontier))
230         while (len(frontier) > 0):
231             print('inner loop called', d)
232             next_node = frontier.pop() # POP FROM SAME SIDE, TO IMITATE STACK
233             exploredSet.add(next_node)
234             num_iters = num_iters + 1
235
236             if (goal_test(next_node, endpos)):
237                 depth_of_goal = next_node.depth
```

Robin Mehta
Artificial Intelligence, Homework 2

```
robin_mehta_search.py ✕
235
236     if (goal_test(next_node, endpos)):
237         depth_of_goal = next_node.depth
238         total_nodes_generated = next_node.node_count
239         goalnode = next_node
240         count = 0
241         paths = extract_path(next_node)
242         prevTuple = (0, 0) #safety initialization
243         for nextTuple in paths:
244             if (count < len(paths) and count > 0):
245                 if (prevTuple[0] < nextTuple[0]):
246                     total_cost_of_goal += 3
247                 elif (prevTuple[0] > nextTuple[0]):
248                     total_cost_of_goal += 1
249                 elif (prevTuple[1] < nextTuple[1]):
250                     total_cost_of_goal += 2
251                 else:
252                     total_cost_of_goal += 4
253                 prevTuple = nextTuple
254                 count = count + 1
255         return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
256
257     for unexploredNode in get_successors(next_node, grid, endpos):
258         if ((unexploredNode not in exploredSet) and (unexploredNode not in frontier) and (unexploredNode.depth <=
259
260             print(unexploredNode.position)
261
262             exploredSet.add(unexploredNode)
263             frontier.append(unexploredNode)
264             if (len(frontier) > max_nodes_stored):
265                 max_nodes_stored = len(frontier)
266
267             d = d + 1
268         break
269         #print(d)
270         #d = d + 1
271         #break
272     return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
273
274
275 def breadth_first_search(grid, startpos, endpos):
276     # Perform breadth-first search
277     # Goal test at generation
278     # Queue based search
279     goalnode = node(None, endpos)
280     total_nodes_generated = 0
281     max_nodes_stored = 0
282     num_iters = 0
283     depth_of_goal = 0
```

Robin Mehta
Artificial Intelligence, Homework 2

```
283     depth_of_goal = 0
284     total_cost_of_goal = 0
285
286     frontier = deque()
287     startNode = node(None, startpos)
288     frontier.append(startNode)
289     exploredSet = Set()
290     while (len(frontier) > 0):
291         next_node = frontier.popleft() #POP FROM OPPOSITE SIDE TO IMITATE QUEUE
292         exploredSet.add(next_node)
293         num_iters = num_iters + 1
294
295         if (goal_test(next_node, endpos)):
296             depth_of_goal = next_node.depth
297             total_nodes_generated = next_node.node_count
298             goalnode = next_node
299
300             count = 0
301             paths = extract_path(next_node)
302             prevTuple = (0, 0) #safety initialization
303             for nextTuple in paths:
304                 if (count < len(paths) and count > 0):
305                     if (prevTuple[0] < nextTuple[0]):
306                         #south
307                         total_cost_of_goal += 3
308                     elif (prevTuple[0] > nextTuple[0]):
309                         #north
310                         total_cost_of_goal += 1
311                     elif (prevTuple[1] < nextTuple[1]):
312                         #east
313                         total_cost_of_goal += 2
314                     else:
315                         #west
316                         total_cost_of_goal += 4
317             prevTuple = nextTuple
318             count = count + 1
319
320             return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
321
322         for unexploredNode in get_successors(next_node, grid, endpos):
323             if ((unexploredNode not in exploredSet) and (unexploredNode not in frontier)):
324                 exploredSet.add(unexploredNode)
325                 frontier.append(unexploredNode)
326                 if (len(frontier) > max_nodes_stored):
327                     max_nodes_stored = len(frontier)
328
329     return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
330
331 if __name__ == '__main__':
    #if you want to run the code from the command line, you can use the following code
    #python3 a_star.py 0 0 3 3
```

Robin Mehta
Artificial Intelligence, Homework 2

```
330
331 def uniform_cost_search(grid, startpos, endpos):
332     # Perform uniform cost search
333     # breadth first search, except with a MinHeap instead of deque.
334     # Goal test at expansion
335     goalnode = node(None, endpos)
336     total_nodes_generated = 0
337     max_nodes_stored = 0
338     num_iters = 0
339     depth_of_goal = 0
340     total_cost_of_goal = 0
341
342     frontier = MinHeap()
343     startNode = node(None, startpos)
344     frontier.push_or_update(startNode)
345     exploredSet = Set()
346     while (len(frontier) > 0):
347         next_node = frontier.pop() #POP FROM OPPOSITE SIDE TO IMMITATE QUEUE
348         exploredSet.add(next_node)
349         num_iters = num_iters + 1
350
351         if (goal_test(next_node, endpos)):
352             depth_of_goal = next_node.depth
353             total_nodes_generated = next_node.node_count
354             goalnode = next_node
355
356             count = 0
357             paths = extract_path(next_node)
358             prevTuple = (0, 0) #safety initialization
359             for nextTuple in paths:
360                 if (count < len(paths) and count > 0):
361                     if (prevTuple[0] < nextTuple[0]):
362                         #south
363                         total_cost_of_goal += 3
364                     elif (prevTuple[0] > nextTuple[0]):
365                         #north
366                         total_cost_of_goal += 1
367                     elif (prevTuple[1] < nextTuple[1]):
368                         #east
369                         total_cost_of_goal += 2
370                     else:
371                         #west
372                         total_cost_of_goal += 4
373                 prevTuple = nextTuple
374                 count = count + 1
375
376     return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
377
```

Robin Mehta
Artificial Intelligence, Homework 2

```
377         for unexploredNode in get_successors(next_node, grid, endpos):
378             if (unexploredNode not in exploredSet):
379                 exploredSet.add(unexploredNode)
380                 frontier.push_or_update(unexploredNode)
381                 if (len(frontier) > max_nodes_stored):
382                     max_nodes_stored = len(frontier)
383
384     return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
385
386 def a_star_search(grid, startpos, endpos, heuristic=manhattan):
387     # Perform A* search
388     # Goal test at expansion
389     goalnode = node(None, endpos)
390     total_nodes_generated = 0
391     max_nodes_stored = 0
392     num_iters = 0
393     depth_of_goal = 0
394     total_cost_of_goal = 0
395
396     frontier = MinHeap()
397     startNode = node(None, startpos)
398     frontier.push_or_update(startNode)
399     exploredSet = Set()
400     while (len(frontier) > 0):
401         next_node = frontier.pop() #POP FROM OPPOSITE SIDE TO IMITATE QUEUE
402         exploredSet.add(next_node)
403         num_iters = num_iters + 1
404
405         if (goal_test(next_node, endpos)):
406             depth_of_goal = next_node.depth
407             total_nodes_generated = next_node.node_count
408             goalnode = next_node
409
410             count = 0
411             paths = extract_path(next_node)
412             prevTuple = (0, 0) #safety initialization
413             for nextTuple in paths:
414                 if (count < len(paths) and count > 0):
415                     if (prevTuple[0] < nextTuple[0]):
416                         #south
417                         total_cost_of_goal += 3
418                     elif (prevTuple[0] > nextTuple[0]):
419                         #north
420                         total_cost_of_goal += 1
421                     elif (prevTuple[1] < nextTuple[1]):
422                         #east
423                         total_cost_of_goal += 2
424                     else:
425                         #west
426                         total_cost_of_goal += 4
427                     prevTuple = nextTuple
428                     count = count + 1
429
430             return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
431
432     for unexploredNode in get_successors(next_node, grid, endpos, heuristic):
433         if (unexploredNode not in exploredSet):
434             exploredSet.add(unexploredNode)
435             frontier.push_or_update(unexploredNode)
436             if (len(frontier) > max_nodes_stored):
437                 max_nodes_stored = len(frontier)
438
439     return goalnode, total_nodes_generated, max_nodes_stored, num_iters, depth_of_goal, total_cost_of_goal
440
441
442
```