

CISC352 - Assignment 02

Gabriele Cimolino - 10009098

Luis Rivera-Wong - 10142361

Zach Slater - 10196265

Tom Szendrey - 10187030

March 16, 2018

1 Part 1: Pathfinding

Pre and Post Processing

Both of the following function implementations will need to read from an input file and write into an output file. This is done through the help of the following helper functions.

```
1 def reader(filename):
2     return_grid = []
3     with open(filename) as f:
4         # grid_data = [i.split() for i in f.readlines()]
5         for i in f.readlines():
6             a_line = i.split()
7             a_list_first = a_line[0]
8             the_chars = list(a_list_first)
9             return_grid.append(the_chars)
10            # print(the_chars)
11    return return_grid
```

```
1 def writer(filename, grid):
2     with open(filename, 'w+') as f:
3         # f.write(grid_data2)
4         for i in grid:
5             f.write(' '.join(i) + '\n')
```

In addition to this, we needed a way to find the specific locations for the *Start* cell and the *Goal* cell, this was aided through the following helper function.

```
1 def target_finder(grid, target):
2     location = []
3     for i in range(len(grid)):
4         for j in range(len(grid[i])):
5             if (grid[i][j] == target):
6                 location.append(i)
7                 location.append(j)
8    return location
```

1.1 Greedy

A greedy algorithm attempts to solve a problem by making the locally optimal choice at each stage in the hopes of finding the globally optimum. In pathfinding,

the algorithm will find the heuristics for all of its available options and choose the best option at that moment.

```

1 def greedy_a(tmp_grid, s_loc, g_loc):

1 def greedy_b(tmp_grid, s_loc, g_loc):
2     a_grid = copy.deepcopy(tmp_grid)
3     curr_loc = copy.deepcopy(s_loc)
4     prev_dir = "None"
5     stuck = False
6     while (not(stuck)):
7         print("\n")
8         for x in a_grid:
9             print(' '.join(x))
10            left_dist = math.inf
11            right_dist = math.inf
12            up_dist = math.inf
13            down_dist = math.inf
14            up_right_dist = math.inf
15            up_left_dist = math.inf
16            down_right_dist = math.inf
17            down_left_dist = math.inf
18            #
#####

19            #Get the H values for all the directions.
20            #
#####

21            # Left distance
22            if ((curr_loc[1] - 1) >= 0):
23                if(a_grid[curr_loc[0]][curr_loc[1] - 1] == 'G'):
24                    return a_grid
25                if(a_grid[curr_loc[0]][curr_loc[1] - 1] == '_'):
26                    # left_dist = (row_diff + col_diff)
27                    left_dist = cheb([curr_loc[0], (curr_loc[1]-1)],
g_loc) #(g_loc[0] - curr_loc[0]) + (g_loc[1] - (curr_loc[1]-1))
28                    #print(" left_dist = ", left_dist)
29            # Right distance
30            if ((curr_loc[1] + 1) < len(a_grid[0])):
31                if(a_grid[curr_loc[0]][curr_loc[1] + 1] == 'G'):
32                    return a_grid
33                if(a_grid[curr_loc[0]][curr_loc[1] + 1] == '_'):
34                    # right_dist = (row_diff + col_diff)
35                    right_dist = cheb([curr_loc[0], (curr_loc[1]+1)],
g_loc) #(g_loc[0] - curr_loc[0]) + (g_loc[1] - (curr_loc[1]+1))
36                    #print(" right_dist = ", right_dist)
37            # Up distance
38            if ((curr_loc[0] - 1) >= 0):
39                if(a_grid[(curr_loc[0]-1)][curr_loc[1]] == 'G'):
40                    return a_grid
41                if(a_grid[(curr_loc[0]+1)][curr_loc[1]] == '_'):
42                    # up_dist = (row_diff + col_diff)
43                    up_dist = cheb([(curr_loc[0]-1), curr_loc[1]], g_loc)
44                    #(g_loc[0] - (curr_loc[0]-1)) + (g_loc[1] - curr_loc[1])
45                    #print(" up_dist = ", up_dist)
46            # Down distance
47            if ((curr_loc[0] + 1) < len(a_grid)):
48                if(a_grid[(curr_loc[0]+1)][curr_loc[1]] == 'G'):
49                    return a_grid
50                if(a_grid[(curr_loc[0]+1)][curr_loc[1]] == '_'):
                    # down_dist = (row_diff + col_diff)

```

```

51         down_dist = cheb([(curr_loc[0]+1), curr_loc[1]],
52         g_loc) # (g_loc[0] - (curr_loc[0]+1)) + (g_loc[1] - curr_loc[1])
53         #print("down_dist = ", down_dist)
54         #Up + Right (diagonal)
55         if (((curr_loc[0] - 1) >= 0) and ((curr_loc[1] + 1) < len(
56         a_grid[0]))):
57             #
58             up
59             right
60             if (a_grid[curr_loc[0]-1][curr_loc[1]+1] == 'G'):
61                 return a_grid
62             #
63             up
64             right
65             if (a_grid[curr_loc[0]-1][curr_loc[1]+1] == '_'):
66                 up_right_dist = cheb([(curr_loc[0]-1), (curr_loc
67                 [1]+1)], g_loc)
68
69         #Up + Left (diagonal)
70         if (((curr_loc[0] - 1) >= 0) and ((curr_loc[1] - 1) < len(
71         a_grid[0]))):
72             #
73             up
74             Left
75             if (a_grid[curr_loc[0]-1][curr_loc[1]-1] == 'G'):
76                 return a_grid
77             #
78             up
79             Left
80             if (a_grid[curr_loc[0]-1][curr_loc[1]-1] == '_'):
81                 up_left_dist = cheb([(curr_loc[0]-1), (curr_loc
82                 [1]-1)], g_loc)
83
84         #Down + Right (diagonal)
85         if (((curr_loc[0] + 1) >= 0) and ((curr_loc[1] + 1) < len(
86         a_grid[0]))):
87             #
88             Down
89             right
90             if (a_grid[curr_loc[0]+1][curr_loc[1]+1] == 'G'):
91                 return a_grid
92             #
93             Down
94             right
95             if (a_grid[curr_loc[0]+1][curr_loc[1]+1] == '_'):
96                 down_right_dist = cheb([(curr_loc[0]+1), (curr_loc
97                 [1]+1)], g_loc)
98
99         #Down + Left (diagonal)
100         if (((curr_loc[0] + 1) >= 0) and ((curr_loc[1] - 1) < len(
101         a_grid[0]))):
102             #
103             Down
104             Left
105             if (a_grid[curr_loc[0]+1][curr_loc[1]-1] == 'G'):
106                 return a_grid
107             #
108             Down
109             Left
110             if (a_grid[curr_loc[0]+1][curr_loc[1]-1] == '_'):
111                 down_left_dist = cheb([(curr_loc[0]+1), (curr_loc
112                 [1]-1)], g_loc)
113
114         if (left_dist == math.inf and right_dist == math.inf and
115         up_dist == math.inf and down_dist == math.inf):
116             stuck = True
117             return False
118
119         if (not(stuck)):
120             # TODO: Improve the checking for None values before min
121             check
122             min_index = randomMinIndex([up_dist, down_dist,
123             left_dist, right_dist, up_right_dist, up_left_dist,
124             down_right_dist, down_left_dist])
125             #print("Previous direction was %s" % prev_dir)
126             if (prev_dir == "Down" and min_index == 0):
127                 return False

```

```

100     if (prev_dir == "Up" and min_index == 1):
101         return False
102     if (prev_dir == "Right" and min_index == 2):
103         return False
104     if (prev_dir == "Left" and min_index == 3):
105         return False
106
107     #Diagonal Directions
108     if (min_index == 4):
109         curr_loc[0] -= 1
110         curr_loc[1] += 1
111         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
112         #print('up-right')
113     elif (min_index == 5):
114         curr_loc[0] -= 1
115         curr_loc[1] -= 1
116         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
117         #print('up-left')
118     elif (min_index == 6):
119         curr_loc[0] += 1
120         curr_loc[1] += 1
121         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
122         #print('down-right')
123     elif (min_index == 7):
124         curr_loc[0] += 1
125         curr_loc[1] -= 1
126         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
127         #print('down-left')
128
129     #Cardinal Directions
130     elif (min_index == 2):
131         curr_loc[1] -= 1
132         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
133         #print('left')
134
135     elif (min_index == 3):
136         curr_loc[1] += 1
137         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
138         #print('right')
139     elif (min_index == 0):
140         curr_loc[0] -= 1
141         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
142         #print('up')
143     elif (min_index == 1):
144         curr_loc[0] += 1
145         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
146         #print('down')

```

1.2 A*

A* is an informed-search algorithm, very similar to Dijkstra's algorithm for finding the shortest paths in graph, except for the fact that the priority isn't just sorted by the distance but by the distance of the path and the distance it has to go.

1.2.1 Standard (A*_a)

The `a_star_a` function takes 3 parameters; a grid, the location of the start and goal as an array in row, column format.

```
1 def a_star_a(tmp_grid, s_loc, g_loc):
```

Our solution for the non-diagonal movement for pathfinding starts by creating a local copy of the grid, an array to hold the visited nodes that the current node has come from as well as an array to hold the associated costs. Both of the mentioned arrays are multidimensional arrays initialized to the values *None* representing the entire grid, this is to have a 1:1 representation of costs and visited nodes to our original grid. Lastly, the initialize the needed priority queue, boolean flag to determine if a solution has been found, and variable containing the our current location.

We start the search for the goal by first pushing the root node onto the priority queue using a priority value determined by the Manhattan distance heuristic from the *Start Node* to the *Goal Node*.

```
1 lastlinelastline
2 def manhattan(a, b):
3     return abs( a[0] - b[0] ) + abs( a[1] - b[1] )
4
5 #Will be used to find the h values for up down left right and
6   diagonals
7 def cheb(a, b):
8     return max( abs( a[0] - b[0] ), abs( a[1] - b[1] ) )
9
10 def printMatrix(matrix):
11     print("\n")
12     for x in matrix:
13         print(x)
14
15 def a_star_b(tmp_grid, s_loc, g_loc):
16     a_grid = copy.deepcopy(tmp_grid)
17     came_from = [[None for j in range(len(a_grid[0]))] for i in
18                   range(len(a_grid))]
19     cost = [[None for j in range(len(a_grid[0]))] for i in range(
20             len(a_grid))]
21
22     cost[s_loc[0]][s_loc[1]] = 0
23
24     curr_loc = None #copy.deepcopy(s_loc)
25     foundGoal = False
26     visited = []
27
28     heappush(visited, (cheb(s_loc, g_loc), s_loc))
29
30     #Find the goal
31     while(not(foundGoal)):
32         curr_node = heappop(visited)
33         curr_loc = curr_node[1]
34
35         if(a_grid[curr_loc[0]][curr_loc[1]] == 'G'):
36             foundGoal = True
37
38         if(not(foundGoal)):
39             #Left
40             if(curr_loc[1] - 1 >= 0):
41                 if(a_grid[curr_loc[0]][curr_loc[1] - 1] == '_' or
42                    a_grid[curr_loc[0]][curr_loc[1] - 1] == 'G'):
43                     cost_so_far = cost[curr_loc[0]][curr_loc[1]]
44
45                     if(cost[curr_loc[0]][curr_loc[1] - 1] == None
46                        or cost_so_far + 1 < cost[curr_loc[0]][curr_loc[1] - 1]):
```

```

43         cost[curr_loc[0]][curr_loc[1] - 1] =
cost_so_far + 1
44         came_from[curr_loc[0]][curr_loc[1] - 1] = [
curr_loc[0], curr_loc[1]]
45         heappush(visited, (cost_so_far + 1 + cheb([
curr_loc[0], curr_loc[1] - 1], g_loc), [curr_loc[0], curr_loc
[1] - 1]))
46
47         #Right
48         if(curr_loc[1] + 1 < len(a_grid[0])):
49             if(a_grid[curr_loc[0]][curr_loc[1] + 1] == '_' or
a_grid[curr_loc[0]][curr_loc[1] + 1] == 'G'):
50                 cost_so_far = cost[curr_loc[0]][curr_loc[1]]
51
52
53             if(cost[curr_loc[0]][curr_loc[1] + 1] == None
or cost_so_far + 1 < cost[curr_loc[0]][curr_loc[1] + 1]):
54                 cost[curr_loc[0]][curr_loc[1] + 1] =
cost_so_far + 1
55                 came_from[curr_loc[0]][curr_loc[1] + 1] = [
curr_loc[0], curr_loc[1]]
56                 heappush(visited, (cost_so_far + 1 + cheb([
curr_loc[0], curr_loc[1] + 1], g_loc), [curr_loc[0], curr_loc
[1] + 1]))
57
58                 #Up
59                 if(curr_loc[0] - 1 >= 0):
60                     if(a_grid[curr_loc[0] - 1][curr_loc[1]] == '_' or
a_grid[curr_loc[0] - 1][curr_loc[1]] == 'G'):
61                         cost_so_far = cost[curr_loc[0]][curr_loc[1]]
62
63
64                     if(cost[curr_loc[0] - 1][curr_loc[1]] == None
or cost_so_far + 1 < cost[curr_loc[0] - 1][curr_loc[1]]):
65                         cost[curr_loc[0] - 1][curr_loc[1]] =
cost_so_far + 1
66                         came_from[curr_loc[0] - 1][curr_loc[1]] = [
curr_loc[0], curr_loc[1]]
67                         heappush(visited, (cost_so_far + 1 + cheb([
curr_loc[0] - 1, curr_loc[1]], g_loc), [curr_loc[0] - 1,
curr_loc[1]]))
68
69                     #Down
70                     if(curr_loc[0] + 1 < len(a_grid)):
71                         if(a_grid[curr_loc[0] + 1][curr_loc[1]] == '_' or
a_grid[curr_loc[0] + 1][curr_loc[1]] == 'G'):
72                             cost_so_far = cost[curr_loc[0]][curr_loc[1]]
73
74
75                             if(cost[curr_loc[0] + 1][curr_loc[1]] == None
or cost_so_far + 1 < cost[curr_loc[0] + 1][curr_loc[1]]):
76                                 cost[curr_loc[0] + 1][curr_loc[1]] =
cost_so_far + 1
77                                 came_from[curr_loc[0] + 1][curr_loc[1]] = [
curr_loc[0], curr_loc[1]]
78                                 heappush(visited, (cost_so_far + 1 + cheb([
curr_loc[0] + 1, curr_loc[1]], g_loc), [curr_loc[0] + 1,
curr_loc[1]]))
79
80                             #Up Left
81                             if(curr_loc[0] - 1 >= 0 and curr_loc[1] - 1 >= 0):
82                                 if(a_grid[curr_loc[0] - 1][curr_loc[1] - 1] == '_'

```

```

83     or a_grid[curr_loc[0] - 1][curr_loc[1] - 1] == 'G':
84         cost_so_far = cost[curr_loc[0]][curr_loc[1]]
85
86         if(cost[curr_loc[0] - 1][curr_loc[1] - 1] ==
87 None or cost_so_far + 1 < cost[curr_loc[0] - 1][curr_loc[1] -
88 1]):
89             cost[curr_loc[0] - 1][curr_loc[1] - 1] =
90 cost_so_far + 1
91             came_from[curr_loc[0] - 1][curr_loc[1] - 1]
92 = [curr_loc[0], curr_loc[1]]
93             heappush(visited, (cost_so_far + 1 + cheb([
94 curr_loc[0] - 1, curr_loc[1] - 1], g_loc), [curr_loc[0] - 1,
95 curr_loc[1] - 1]))
96
97             #Up Right
98             if(curr_loc[0] - 1 and curr_loc[1] + 1 < len(a_grid[0])
99 ):
100                 if(a_grid[curr_loc[0] - 1][curr_loc[1] + 1] == '- '
101 or a_grid[curr_loc[0] - 1][curr_loc[1] + 1] == 'G'):
102                     cost_so_far = cost[curr_loc[0]][curr_loc[1]]
103
104                     if(cost[curr_loc[0] - 1][curr_loc[1] + 1] ==
105 None or cost_so_far + 1 < cost[curr_loc[0] - 1][curr_loc[1] +
106 1]):
107                         cost[curr_loc[0] - 1][curr_loc[1] + 1] =
108 cost_so_far + 1
109                         came_from[curr_loc[0] - 1][curr_loc[1] + 1]
110 = [curr_loc[0] - 1, curr_loc[1]]
111                         heappush(visited, (cost_so_far + 1 + cheb([
112 curr_loc[0] - 1, curr_loc[1] + 1], g_loc), [curr_loc[0] - 1,
113 curr_loc[1] + 1]))
114
115                         #Down Left
116                         if(curr_loc[0] + 1 < len(a_grid) and curr_loc[1] - 1 >=
117 0):
118                             if(a_grid[curr_loc[0] + 1][curr_loc[1] - 1] == '- '
119 or a_grid[curr_loc[0] + 1][curr_loc[1] - 1] == 'G'):
120                                 cost_so_far = cost[curr_loc[0]][curr_loc[1]]
121
122                                 if(cost[curr_loc[0] + 1][curr_loc[1] - 1] ==
123 None or cost_so_far + 1 < cost[curr_loc[0] + 1][curr_loc[1] -
124 1]):
125                                     cost[curr_loc[0] + 1][curr_loc[1] - 1] =
126 cost_so_far + 1
127                                     came_from[curr_loc[0] + 1][curr_loc[1] - 1]
128 = [curr_loc[0], curr_loc[1]]
129                                     heappush(visited, (cost_so_far + 1 + cheb([
130 curr_loc[0] + 1, curr_loc[1] - 1], g_loc), [curr_loc[0] + 1,
131 curr_loc[1] - 1]))
132
133                                     #Down Right
134                                     if(curr_loc[0] + 1 < len(a_grid) and curr_loc[1] + 1 <
135 len(a_grid[0])):
136                                         if(a_grid[curr_loc[0] + 1][curr_loc[1] + 1] == '- '
137 or a_grid[curr_loc[0] + 1][curr_loc[1] + 1] == 'G'):
138                                             cost_so_far = cost[curr_loc[0]][curr_loc[1]]
139
140                                             if(cost[curr_loc[0] + 1][curr_loc[1] + 1] ==

```

```

120     None or cost_so_far + 1 < cost[curr_loc[0] + 1][curr_loc[1] +
121         1]):
122         cost[curr_loc[0] + 1][curr_loc[1] + 1] =
123             cost_so_far + 1
124         came_from[curr_loc[0] + 1][curr_loc[1] + 1]
125         = [curr_loc[0], curr_loc[1]]
126         heappush(visited, (cost_so_far + 1 + cheb([
127             curr_loc[0] + 1, curr_loc[1] + 1], g_loc), [curr_loc[0] + 1,
128             curr_loc[1] + 1]))
129
130     while(came_from[curr_loc[0]][curr_loc[1]] != None):
131         curr_loc = came_from[curr_loc[0]][curr_loc[1]]
132         if(a_grid[curr_loc[0]][curr_loc[1]] != 'S'):
133             a_grid[curr_loc[0]][curr_loc[1]] = 'P'
134
135     return a_grid
136
137 def a_star_a(tmp_grid, s_loc, g_loc):
138     a_grid = copy.deepcopy(tmp_grid)
139     came_from = [[None for j in range(len(a_grid[0]))] for i in
140         range(len(a_grid))]
141     cost = [[None for j in range(len(a_grid[0]))] for i in range(
142         len(a_grid))]
143
144     cost[s_loc[0]][s_loc[1]] = 0
145
146     curr_loc = None #copy.deepcopy(s_loc)
147     foundGoal = False
148     visited = []
149
150     heappush(visited, (manhattan(s_loc, g_loc), s_loc))
151
152     #Find the goal
153     while(not(foundGoal)):
154         curr_node = heappop(visited)
155         curr_loc = curr_node[1]
156
157         if(a_grid[curr_loc[0]][curr_loc[1]] == 'G'):
158             foundGoal = True
159
160         if(not(foundGoal)):
161             #Left
162             if(curr_loc[1] - 1 >= 0):
163                 if(a_grid[curr_loc[0]][curr_loc[1] - 1] == '-' or
164                     a_grid[curr_loc[0]][curr_loc[1] - 1] == 'G'):
165                     cost_so_far = cost[curr_loc[0]][curr_loc[1]]
166
167                     if(cost[curr_loc[0]][curr_loc[1] - 1] == None
168                     or cost_so_far + 1 < cost[curr_loc[0]][curr_loc[1] - 1]):
169                         cost[curr_loc[0]][curr_loc[1] - 1] =
170                             cost_so_far + 1
171                         came_from[curr_loc[0]][curr_loc[1] - 1] = [
172                             curr_loc[0], curr_loc[1]]
173                         heappush(visited, (cost_so_far + 1 +
174                             manhattan([curr_loc[0], curr_loc[1] - 1], g_loc), [curr_loc[0],
175                             curr_loc[1] - 1]))
176
177             #Right
178             if(curr_loc[1] + 1 < len(a_grid[0])):
179                 if(a_grid[curr_loc[0]][curr_loc[1] + 1] == '-' or
180                     a_grid[curr_loc[0]][curr_loc[1] + 1] == 'G'):

```



```

167         cost_so_far = cost[curr_loc[0]][curr_loc[1]]
168
169
170         if (cost[curr_loc[0]][curr_loc[1] + 1] == None
171 or cost_so_far + 1 < cost[curr_loc[0]][curr_loc[1] + 1]):
172             cost[curr_loc[0]][curr_loc[1] + 1] =
173             cost_so_far + 1
174             came_from[curr_loc[0]][curr_loc[1] + 1] = [
175             curr_loc[0], curr_loc[1]]
176             heappush(visited, (cost_so_far + 1 +
177             manhattan([curr_loc[0], curr_loc[1] + 1], g_loc), [curr_loc[0],
178             curr_loc[1] + 1]))
179
180         #Up
181         if (curr_loc[0] - 1 >= 0):
182             if (a_grid[curr_loc[0] - 1][curr_loc[1]] == '_' or
183             a_grid[curr_loc[0] - 1][curr_loc[1]] == 'G'):
184                 cost_so_far = cost[curr_loc[0]][curr_loc[1]]
185
186         if (cost[curr_loc[0] - 1][curr_loc[1]] == None
187 or cost_so_far + 1 < cost[curr_loc[0] - 1][curr_loc[1]]):
188             cost[curr_loc[0] - 1][curr_loc[1]] =
189             cost_so_far + 1
190             came_from[curr_loc[0] - 1][curr_loc[1]] = [
191             curr_loc[0], curr_loc[1]]
192             heappush(visited, (cost_so_far + 1 +
193             manhattan([curr_loc[0] - 1, curr_loc[1]], g_loc), [curr_loc[0]
194             - 1, curr_loc[1]]))
195
196         #Down
197         if (curr_loc[0] + 1 < len(a_grid)):
198             if (a_grid[curr_loc[0] + 1][curr_loc[1]] == '_' or
199             a_grid[curr_loc[0] + 1][curr_loc[1]] == 'G'):
200                 cost_so_far = cost[curr_loc[0]][curr_loc[1]]
201
202         if (cost[curr_loc[0] + 1][curr_loc[1]] == None
203 or cost_so_far + 1 < cost[curr_loc[0] + 1][curr_loc[1]]):
204             cost[curr_loc[0] + 1][curr_loc[1]] =
205             cost_so_far + 1
206             came_from[curr_loc[0] + 1][curr_loc[1]] = [
207             curr_loc[0], curr_loc[1]]
208             heappush(visited, (cost_so_far + 1 +
209             manhattan([curr_loc[0] + 1, curr_loc[1]], g_loc), [curr_loc[0]
210             + 1, curr_loc[1]]))
211
212         while (came_from[curr_loc[0]][curr_loc[1]] != None):
213             curr_loc = came_from[curr_loc[0]][curr_loc[1]]
214             if (a_grid[curr_loc[0]][curr_loc[1]] != 'S'):
215                 a_grid[curr_loc[0]][curr_loc[1]] = 'P'
216
217         return a_grid
218
219 # uses manhattan distance (up down left right solution)
220 def greedy_a(tmp_grid, s_loc, g_loc):
221     a_grid = copy.deepcopy(tmp_grid)
222     curr_loc = copy.deepcopy(s_loc)
223     prev_dir = "None"
224     stuck = False
225     while (not(stuck)):
226         print("\n")

```

```

212     for x in a_grid:
213         print(''.join(x))
214     left_dist = math.inf
215     right_dist = math.inf
216     up_dist = math.inf
217     down_dist = math.inf
218     # Left distance
219     if ((curr_loc[1] - 1) >= 0):
220         if(a_grid[curr_loc[0]][curr_loc[1] - 1] == 'G'):
221             return a_grid
222         if(a_grid[curr_loc[0]][curr_loc[1] - 1] == '_'):
223             left_dist = manhattan([curr_loc[0], (curr_loc[1]-1)
], g_loc) #(g_loc[0] - curr_loc[0]) + (g_loc[1] - (curr_loc
[1]-1))
224     # Right distance
225     if ((curr_loc[1] + 1) < len(a_grid[0])):
226         if(a_grid[curr_loc[0]][curr_loc[1] + 1] == 'G'):
227             return a_grid
228         if(a_grid[curr_loc[0]][curr_loc[1] + 1] == '_'):
229             # right_dist = (row_diff + col_diff)
230             right_dist = manhattan([curr_loc[0], (curr_loc[1]+1)
], g_loc) #(g_loc[0] - curr_loc[0]) + (g_loc[1] - (curr_loc[1]+1)
)
231             #print("right_dist = ", right_dist)
232     # Up distance
233     if ((curr_loc[0] - 1) >= 0):
234         if(a_grid[(curr_loc[0]-1)][curr_loc[1]] == 'G'):
235             return a_grid
236         if(a_grid[(curr_loc[0]-1)][curr_loc[1]] == '_'):
237             # up_dist = (row_diff + col_diff)
238             up_dist = manhattan([(curr_loc[0]-1), curr_loc[1]],
g_loc) #(g_loc[0] - (curr_loc[0]-1)) + (g_loc[1] - curr_loc[1])
239             #print("up_dist = ", up_dist)
240     # Down distance
241     if ((curr_loc[0] + 1) < len(a_grid)):
242         if(a_grid[(curr_loc[0]+1)][curr_loc[1]] == 'G'):
243             return a_grid
244         if(a_grid[(curr_loc[0]+1)][curr_loc[1]] == '_'):
245             # down_dist = (row_diff + col_diff)
246             down_dist = manhattan([(curr_loc[0]+1), curr_loc
[1]], g_loc) #(g_loc[0] - (curr_loc[0]+1)) + (g_loc[1] -
curr_loc[1])
247             #print("down_dist = ", down_dist)
248
249
250     if (left_dist == math.inf and right_dist == math.inf and
up_dist == math.inf and down_dist == math.inf):
251         stuck = True
252         return False
253
254     if (not(stuck)):
255         # TODO: Improve the checking for None values before min
check
256         min_index = randomMinIndex([up_dist, down_dist,
left_dist, right_dist])#min(up_dist, down_dist, left_dist,
right_dist)
257         #print("Previous direction was %s" % prev_dir)
258         if(prev_dir == "Down" and min_index == 0):
259             return False
260         if(prev_dir == "Up" and min_index == 1):
261             return False
262         if(prev_dir == "Right" and min_index == 2):

```

```

263         return False
264     if (prev_dir == "Left" and min_index == 3):
265         return False
266
267     if (min_index == 2):
268         if (prev_dir == "Right"):
269             return False
270         prev_dir = "Left"
271         curr_loc[1] -= 1
272         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
273         #print('left')
274     elif (min_index == 3):
275         if (prev_dir == "Left"):
276             return False
277         prev_dir = "Right"
278         curr_loc[1] += 1
279         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
280         #print('right')
281     elif (min_index == 0):
282         if (prev_dir == "Down"):
283             return False
284         prev_dir = "Up"
285         curr_loc[0] -= 1
286         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
287         #print('up')
288     elif (min_index == 1):
289         if (prev_dir == "Up"):
290             return False
291         prev_dir = "Down"
292         curr_loc[0] += 1
293         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
294         #print('down')
295
296 # uses cheb distance (up down left right diagonal solution)
297 def greedy_b(tmp_grid, s_loc, g_loc):
298     a_grid = copy.deepcopy(tmp_grid)
299     curr_loc = copy.deepcopy(s_loc)
300     prev_dir = "None"
301     stuck = False
302     while (not(stuck)):
303         print("\n")
304         for x in a_grid:
305             print(' '.join(x))
306         left_dist = math.inf
307         right_dist = math.inf
308         up_dist = math.inf
309         down_dist = math.inf
310         up_right_dist = math.inf
311         up_left_dist = math.inf
312         down_right_dist = math.inf
313         down_left_dist = math.inf
314         #
315         #####
316
317         #Get the H values for all the directions.
318         #
319         #####
320
321         # Left distance
322         if ((curr_loc[1] - 1) >= 0):
323             if (a_grid[curr_loc[0]][curr_loc[1] - 1] == 'G'):
324                 return a_grid

```

```

321         if (a_grid[curr_loc[0]][curr_loc[1] - 1] == '_'):
322             # left_dist = (row_diff + col_diff)
323             left_dist = cheb([curr_loc[0], (curr_loc[1]-1)],
g_loc) #(g_loc[0] - curr_loc[0]) + (g_loc[1] - (curr_loc[1]-1))
324             #print(" left_dist = ", left_dist)
325         # Right distance
326         if ((curr_loc[1] + 1) < len(a_grid[0])):
327             if (a_grid[curr_loc[0]][curr_loc[1] + 1] == 'G'):
328                 return a_grid
329             if (a_grid[curr_loc[0]][curr_loc[1] + 1] == '_'):
330                 # right_dist = (row_diff + col_diff)
331                 right_dist = cheb([curr_loc[0], (curr_loc[1]+1)],
g_loc) #(g_loc[0] - curr_loc[0]) + (g_loc[1] - (curr_loc[1]+1))
332                 #print(" right_dist = ", right_dist)
333         # Up distance
334         if ((curr_loc[0] - 1) >= 0):
335             if (a_grid[(curr_loc[0]-1)][curr_loc[1]] == 'G'):
336                 return a_grid
337             if (a_grid[(curr_loc[0]+1)][curr_loc[1]] == '_'):
338                 # up_dist = (row_diff + col_diff)
339                 up_dist = cheb([(curr_loc[0]-1), curr_loc[1]], g_loc)
340                 #(g_loc[0] - (curr_loc[0]-1)) + (g_loc[1] - curr_loc[1])
341                 #print(" up_dist = ", up_dist)
342         # Down distance
343         if ((curr_loc[0] + 1) < len(a_grid)):
344             if (a_grid[(curr_loc[0]+1)][curr_loc[1]] == 'G'):
345                 return a_grid
346             if (a_grid[(curr_loc[0]+1)][curr_loc[1]] == '_'):
347                 # down_dist = (row_diff + col_diff)
348                 down_dist = cheb([(curr_loc[0]+1), curr_loc[1]],
g_loc) #(g_loc[0] - (curr_loc[0]+1)) + (g_loc[1] - curr_loc[1])
349                 #print(" down_dist = ", down_dist)
350         #Up + Right (diagonal)
351         if (((curr_loc[0] - 1) >= 0) and ((curr_loc[1] + 1) < len(
a_grid[0]))):
352             #
353             up
354             right
355             if (a_grid[curr_loc[0]-1][curr_loc[1]+1] == 'G'):
356                 return a_grid
357
358             #
359             up
360             right
361             if (a_grid[curr_loc[0]-1][curr_loc[1]+1] == '_'):
362                 up_right_dist = cheb([(curr_loc[0]-1), (curr_loc
[1]+1)], g_loc)
363
364         #Up + Left (diagonal)
365         if (((curr_loc[0] - 1) >= 0) and ((curr_loc[1] - 1) < len(
a_grid[0]))):
366             #
367             up
368             Left
369             if (a_grid[curr_loc[0]-1][curr_loc[1]-1] == 'G'):
370                 return a_grid
371
372             #
373             up
374             Left
375             if (a_grid[curr_loc[0]-1][curr_loc[1]-1] == '_'):
376                 up_left_dist = cheb([(curr_loc[0]-1), (curr_loc
[1]-1)], g_loc)
377
378         #Down + Right (diagonal)
379         if (((curr_loc[0] + 1) >= 0) and ((curr_loc[1] + 1) < len(
a_grid[0]))):
380             #
381             Down
382             right
383             if (a_grid[curr_loc[0]+1][curr_loc[1]+1] == 'G'):
384                 return a_grid
385
386             #
387             Down
388             right
389             if (a_grid[curr_loc[0]+1][curr_loc[1]+1] == '_'):

```

```

374         down_right_dist = cheb([(curr_loc[0]+1),(curr_loc
[1]+1)],g_loc)
375
376         #Down + Left (diagonal)
377         if (((curr_loc[0] + 1) >= 0) and ((curr_loc[1] - 1) < len(
a_grid[0]))):
378             #                               Down           Left
379             if (a_grid[curr_loc[0]+1][curr_loc[1]-1] == 'G'):
380                 return a_grid
381             #                               Down           Left
382             if (a_grid[curr_loc[0]+1][curr_loc[1]-1] == '_'):
383                 down_left_dist = cheb([(curr_loc[0]+1),(curr_loc
[1]-1)],g_loc)
384
385
386         if (left_dist == math.inf and right_dist == math.inf and
up_dist == math.inf and down_dist == math.inf):
387             stuck = True
388             return False
389
390         if (not(stuck)):
391             # TODO: Improve the checking for None values before min
check
392             min_index = randomMinIndex([up_dist, down_dist,
left_dist, right_dist, up_right_dist, up_left_dist,
down_right_dist, down_left_dist])
393             #print("Previous direction was %s" % prev_dir)
394             if (prev_dir == "Down" and min_index == 0):
395                 return False
396             if (prev_dir == "Up" and min_index == 1):
397                 return False
398             if (prev_dir == "Right" and min_index == 2):
399                 return False
400             if (prev_dir == "Left" and min_index == 3):
401                 return False
402
403             #Diagonal Directions
404             if (min_index == 4):
405                 curr_loc[0] -= 1
406                 curr_loc[1] += 1
407                 a_grid[curr_loc[0]][curr_loc[1]] = 'P'
408                 #print('up-right')
409             elif (min_index == 5):
410                 curr_loc[0] -= 1
411                 curr_loc[1] -= 1
412                 a_grid[curr_loc[0]][curr_loc[1]] = 'P'
413                 #print('up-left')
414             elif (min_index == 6):
415                 curr_loc[0] += 1
416                 curr_loc[1] += 1
417                 a_grid[curr_loc[0]][curr_loc[1]] = 'P'
418                 #print('down-right')
419             elif (min_index == 7):
420                 curr_loc[0] += 1
421                 curr_loc[1] -= 1
422                 a_grid[curr_loc[0]][curr_loc[1]] = 'P'
423                 #print('down-left')
424
425             #Cardinal Directions
426             elif (min_index == 2):
427                 curr_loc[1] -= 1
428                 a_grid[curr_loc[0]][curr_loc[1]] = 'P'

```

```

429         #print('left')
430
431     elif (min_index == 3):
432         curr_loc[1] += 1
433         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
434         #print('right')
435     elif (min_index == 0):
436         curr_loc[0] -= 1
437         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
438         #print('up')
439     elif (min_index == 1):
440         curr_loc[0] += 1
441         a_grid[curr_loc[0]][curr_loc[1]] = 'P'
442         #print('down')
443
444
445
446 def randomMinIndex(array):
447     minValue = min(array)
448     minIndices = []
449
450     for i in range(len(array)):
451         if(array[i] == minValue):
452             minIndices.append(i)
453
454     return minIndices[random.randint(0, len(minIndices) - 1)]
455
456 main()

```

Next, we continuously loop through the main pathfinding part until a goal is found. Within this main block we start off by popping off the first item in the priority queue and perform a check to see if the newly popped node is the *Goal Node*. If it is not we check for an open spots or the *Goal* in the four available directions; up, down, left and right, and on those spots we then perform checks to make sure that they are within the grid. We then update the costs for the open spots and the *came_from* grid and push the results onto our priority queue.

Lastly, we look through the *came_from* grid and write into our local grid instance, the path we took from the *Start Node* to the *Goal Node*, and return the grid.

Here is the entire main function for this version of A*.

```

1 def a_star_a(tmp_grid, s_loc, g_loc):
2     a_grid = copy.deepcopy(tmp_grid)
3     came_from = [[None for j in range(len(a_grid[0]))] for i in
4                 range(len(a_grid))]
5     cost = [[None for j in range(len(a_grid[0]))] for i in range(
6             len(a_grid))]
7
8     cost[s_loc[0]][s_loc[1]] = 0
9
10    curr_loc = None #copy.deepcopy(s_loc)
11    foundGoal = False
12    visited = []
13
14    heappush(visited, (manhattan(s_loc, g_loc), s_loc))
15
16    #Find the goal
17    while(not(foundGoal)):
18        curr_node = heappop(visited)
19        curr_loc = curr_node[1]

```

```

19         if(a_grid[curr_loc[0]][curr_loc[1]] == 'G'):
20             foundGoal = True
21
22         if(not(foundGoal)):
23             #Left
24             if(curr_loc[1] - 1 >= 0):
25                 if(a_grid[curr_loc[0]][curr_loc[1] - 1] == '_' or
a_grid[curr_loc[0]][curr_loc[1] - 1] == 'G'):
26                     cost_so_far = cost[curr_loc[0]][curr_loc[1]]
27
28
29                 if(cost[curr_loc[0]][curr_loc[1] - 1] == None
or cost_so_far + 1 < cost[curr_loc[0]][curr_loc[1] - 1]):
30                     cost[curr_loc[0]][curr_loc[1] - 1] =
cost_so_far + 1
31                     came_from[curr_loc[0]][curr_loc[1] - 1] = [
curr_loc[0], curr_loc[1]]
32                     heappush(visited, (cost_so_far + 1 +
manhattan([curr_loc[0], curr_loc[1] - 1], g_loc), [curr_loc[0],
curr_loc[1] - 1]))
33
34             #Right
35             if(curr_loc[1] + 1 < len(a_grid[0])):
36                 if(a_grid[curr_loc[0]][curr_loc[1] + 1] == '_' or
a_grid[curr_loc[0]][curr_loc[1] + 1] == 'G'):
37                     cost_so_far = cost[curr_loc[0]][curr_loc[1]]
38
39
40                 if(cost[curr_loc[0]][curr_loc[1] + 1] == None
or cost_so_far + 1 < cost[curr_loc[0]][curr_loc[1] + 1]):
41                     cost[curr_loc[0]][curr_loc[1] + 1] =
cost_so_far + 1
42                     came_from[curr_loc[0]][curr_loc[1] + 1] = [
curr_loc[0], curr_loc[1]]
43                     heappush(visited, (cost_so_far + 1 +
manhattan([curr_loc[0], curr_loc[1] + 1], g_loc), [curr_loc[0],
curr_loc[1] + 1]))
44
45             #Up
46             if(curr_loc[0] - 1 >= 0):
47                 if(a_grid[curr_loc[0] - 1][curr_loc[1]] == '_' or
a_grid[curr_loc[0] - 1][curr_loc[1]] == 'G'):
48                     cost_so_far = cost[curr_loc[0]][curr_loc[1]]
49
50
51                 if(cost[curr_loc[0] - 1][curr_loc[1]] == None
or cost_so_far + 1 < cost[curr_loc[0] - 1][curr_loc[1]]):
52                     cost[curr_loc[0] - 1][curr_loc[1]] =
cost_so_far + 1
53                     came_from[curr_loc[0] - 1][curr_loc[1]] = [
curr_loc[0], curr_loc[1]]
54                     heappush(visited, (cost_so_far + 1 +
manhattan([curr_loc[0] - 1, curr_loc[1]], g_loc), [curr_loc[0]
- 1, curr_loc[1]]))
55
56             #Down
57             if(curr_loc[0] + 1 < len(a_grid)):
58                 if(a_grid[curr_loc[0] + 1][curr_loc[1]] == '_' or
a_grid[curr_loc[0] + 1][curr_loc[1]] == 'G'):
59                     cost_so_far = cost[curr_loc[0]][curr_loc[1]]
60
61

```

```

62         if (cost[curr_loc[0] + 1][curr_loc[1]] == None
63     or cost_so_far + 1 < cost[curr_loc[0] + 1][curr_loc[1]]):
64             cost[curr_loc[0] + 1][curr_loc[1]] =
65             cost_so_far + 1
66             came_from[curr_loc[0] + 1][curr_loc[1]] = [
67             curr_loc[0], curr_loc[1]]
68             heappush(visited, (cost_so_far + 1 +
69     manhattan([curr_loc[0] + 1, curr_loc[1]], g_loc), [curr_loc[0]
70     + 1, curr_loc[1]]))
71
72     while (came_from[curr_loc[0]][curr_loc[1]] != None):
73         curr_loc = came_from[curr_loc[0]][curr_loc[1]]
74         if (a_grid[curr_loc[0]][curr_loc[1]] != 'S'):
75             a_grid[curr_loc[0]][curr_loc[1]] = 'P'
76
77     return a_grid

```

1.2.2 Standard and Diagonals (A*_b)

The second version of this solution, one in which we are able to move in diagonal directions in addition to the standard directions, is very similar to the first version. The function the same inputs, a grid, and the location of the *Start* and *Goal* cell.

```

1 def a_star_b(tmp_grid, s_loc, g_loc):

```

We then create a local copy of the grid, initialize our *came_from* and *cost* grid, setup the cost of the *Start* cell as 0, set our current location and lastly setup our looping boolean flags. Afterwards, we initialize our priority queue. This time we use the Chebyshev distance metric as our heuristic value to allow for the diagonal movement as it works as a radial distance measure.

```

1 def cheb(a, b):
2     return max( abs( a[0] - b[0] ), abs( a[1] - b[1] ) )

```

The same looping as the first solution then occurs, looking for a completion flag. Next, we pop the next available node from the queue and look for empty spaces or the *Goal* cell in a radial sense while running grid validation checks, and then update the *cost* grid and *came_from* grid. This is then followed by the pushing of the available spots into the priority queue. This looping then repeats until the *Goal* cell is found. Lastly, the local grid instance is then written with the path based on the *came_from* grid. The local grid instance is then returned from the function to be written to the text output file by the writer function.

Here is the code for the section version of the A* pathfinder.

```

1 def a_star_b(tmp_grid, s_loc, g_loc):
2     a_grid = copy.deepcopy(tmp_grid)
3     came_from = [[None for j in range(len(a_grid[0]))] for i in
4     range(len(a_grid))]
5     cost = [[None for j in range(len(a_grid[0]))] for i in range(
6     len(a_grid))]
7
8     cost[s_loc[0]][s_loc[1]] = 0
9
10    curr_loc = None #copy.deepcopy(s_loc)
11    foundGoal = False
12    visited = []
13
14    heappush(visited, (cheb(s_loc, g_loc), s_loc))

```



```

13
14 #Find the goal
15 while(not(foundGoal)):
16     curr_node = heappop(visited)
17     curr_loc = curr_node[1]
18
19     if(a_grid[curr_loc[0]][curr_loc[1]] == 'G'):
20         foundGoal = True
21
22     if(not(foundGoal)):
23         #Left
24         if(curr_loc[1] - 1 >= 0):
25             if(a_grid[curr_loc[0]][curr_loc[1] - 1] == '_' or
26 a_grid[curr_loc[0]][curr_loc[1] - 1] == 'G'):
27                 cost_so_far = cost[curr_loc[0]][curr_loc[1]]
28
29                 if(cost[curr_loc[0]][curr_loc[1] - 1] == None
30 or cost_so_far + 1 < cost[curr_loc[0]][curr_loc[1] - 1]):
31                     cost[curr_loc[0]][curr_loc[1] - 1] =
32 cost_so_far + 1
33                     came_from[curr_loc[0]][curr_loc[1] - 1] = [
34 curr_loc[0], curr_loc[1]]
35                     heappush(visited, (cost_so_far + 1 + cheb([
36 curr_loc[0], curr_loc[1] - 1], g_loc), [curr_loc[0], curr_loc
37 [1] - 1]))
38
39                 #Right
40                 if(curr_loc[1] + 1 < len(a_grid[0])):
41                     if(a_grid[curr_loc[0]][curr_loc[1] + 1] == '_' or
42 a_grid[curr_loc[0]][curr_loc[1] + 1] == 'G'):
43                         cost_so_far = cost[curr_loc[0]][curr_loc[1]]
44
45                         if(cost[curr_loc[0]][curr_loc[1] + 1] == None
46 or cost_so_far + 1 < cost[curr_loc[0]][curr_loc[1] + 1]):
47                             cost[curr_loc[0]][curr_loc[1] + 1] =
48 cost_so_far + 1
49                             came_from[curr_loc[0]][curr_loc[1] + 1] = [
50 curr_loc[0], curr_loc[1]]
51                             heappush(visited, (cost_so_far + 1 + cheb([
52 curr_loc[0], curr_loc[1] + 1], g_loc), [curr_loc[0], curr_loc
53 [1] + 1]))
54
55                         #Up
56                         if(curr_loc[0] - 1 >= 0):
57                             if(a_grid[curr_loc[0] - 1][curr_loc[1]] == '_' or
58 a_grid[curr_loc[0] - 1][curr_loc[1]] == 'G'):
59                                 cost_so_far = cost[curr_loc[0]][curr_loc[1]]
60
61                                 if(cost[curr_loc[0] - 1][curr_loc[1]] == None
62 or cost_so_far + 1 < cost[curr_loc[0] - 1][curr_loc[1]]):
63                                     cost[curr_loc[0] - 1][curr_loc[1]] =
64 cost_so_far + 1
65                                     came_from[curr_loc[0] - 1][curr_loc[1]] = [
66 curr_loc[0], curr_loc[1]]
67                                     heappush(visited, (cost_so_far + 1 + cheb([
68 curr_loc[0] - 1, curr_loc[1]], g_loc), [curr_loc[0] - 1,
69 curr_loc[1]]))
70
71                                 #Down

```

```

57         if (curr_loc[0] + 1 < len(a_grid)):
58             if (a_grid[curr_loc[0] + 1][curr_loc[1]] == '_' or
a_grid[curr_loc[0] + 1][curr_loc[1]] == 'G'):
59                 cost_so_far = cost[curr_loc[0]][curr_loc[1]]
60
61
62             if (cost[curr_loc[0] + 1][curr_loc[1]] == None
or cost_so_far + 1 < cost[curr_loc[0] + 1][curr_loc[1]]):
63                 cost[curr_loc[0] + 1][curr_loc[1]] =
cost_so_far + 1
64                 came_from[curr_loc[0] + 1][curr_loc[1]] = [
curr_loc[0], curr_loc[1]]
65                 heappush(visited, (cost_so_far + 1 + cheb([
curr_loc[0] + 1, curr_loc[1]], g_loc), [curr_loc[0] + 1,
curr_loc[1]]))
66
67         #Up Left
68         if (curr_loc[0] - 1 >= 0 and curr_loc[1] - 1 >= 0):
69             if (a_grid[curr_loc[0] - 1][curr_loc[1] - 1] == '_'
or a_grid[curr_loc[0] - 1][curr_loc[1] - 1] == 'G'):
70                 cost_so_far = cost[curr_loc[0]][curr_loc[1]]
71
72
73             if (cost[curr_loc[0] - 1][curr_loc[1] - 1] ==
None or cost_so_far + 1 < cost[curr_loc[0] - 1][curr_loc[1] -
1]):
74                 cost[curr_loc[0] - 1][curr_loc[1] - 1] =
cost_so_far + 1
75                 came_from[curr_loc[0] - 1][curr_loc[1] - 1]
= [curr_loc[0], curr_loc[1]]
76                 heappush(visited, (cost_so_far + 1 + cheb([
curr_loc[0] - 1, curr_loc[1] - 1], g_loc), [curr_loc[0] - 1,
curr_loc[1] - 1]))
77
78         #Up Right
79         if (curr_loc[0] - 1 and curr_loc[1] + 1 < len(a_grid[0])
):
80             if (a_grid[curr_loc[0] - 1][curr_loc[1] + 1] == '_'
or a_grid[curr_loc[0] - 1][curr_loc[1] + 1] == 'G'):
81                 cost_so_far = cost[curr_loc[0]][curr_loc[1]]
82
83
84             if (cost[curr_loc[0] - 1][curr_loc[1] + 1] ==
None or cost_so_far + 1 < cost[curr_loc[0] - 1][curr_loc[1] +
1]):
85                 cost[curr_loc[0] - 1][curr_loc[1] + 1] =
cost_so_far + 1
86                 came_from[curr_loc[0] - 1][curr_loc[1] + 1]
= [curr_loc[0] - 1, curr_loc[1]]
87                 heappush(visited, (cost_so_far + 1 + cheb([
curr_loc[0] - 1, curr_loc[1] + 1], g_loc), [curr_loc[0] - 1,
curr_loc[1] + 1]))
88
89         #Down Left
90         if (curr_loc[0] + 1 < len(a_grid) and curr_loc[1] - 1 >=
0):
91             if (a_grid[curr_loc[0] + 1][curr_loc[1] - 1] == '_'
or a_grid[curr_loc[0] + 1][curr_loc[1] - 1] == 'G'):
92                 cost_so_far = cost[curr_loc[0]][curr_loc[1]]
93
94
95             if (cost[curr_loc[0] + 1][curr_loc[1] - 1] ==

```

```

None or cost_so_far + 1 < cost[curr_loc[0] + 1][curr_loc[1] - 1]):
96         cost[curr_loc[0] + 1][curr_loc[1] - 1] =
cost_so_far + 1
97         came_from[curr_loc[0] + 1][curr_loc[1] - 1]
= [curr_loc[0], curr_loc[1]]
98         heappush(visited, (cost_so_far + 1 + cheb([
curr_loc[0] + 1, curr_loc[1] - 1], g_loc), [curr_loc[0] + 1,
curr_loc[1] - 1]))
99
100         #Down Right
101         if (curr_loc[0] + 1 < len(a_grid) and curr_loc[1] + 1 <
len(a_grid[0])):
102             if (a_grid[curr_loc[0] + 1][curr_loc[1] + 1] == '-'
or a_grid[curr_loc[0] + 1][curr_loc[1] + 1] == 'G'):
103                 cost_so_far = cost[curr_loc[0]][curr_loc[1]]
104
105
106             if (cost[curr_loc[0] + 1][curr_loc[1] + 1] ==
None or cost_so_far + 1 < cost[curr_loc[0] + 1][curr_loc[1] + 1]):
107                 cost[curr_loc[0] + 1][curr_loc[1] + 1] =
cost_so_far + 1
108                 came_from[curr_loc[0] + 1][curr_loc[1] + 1]
= [curr_loc[0], curr_loc[1]]
109                 heappush(visited, (cost_so_far + 1 + cheb([
curr_loc[0] + 1, curr_loc[1] + 1], g_loc), [curr_loc[0] + 1,
curr_loc[1] + 1]))
110
111         while (came_from[curr_loc[0]][curr_loc[1]] != None):
112             curr_loc = came_from[curr_loc[0]][curr_loc[1]]
113             if (a_grid[curr_loc[0]][curr_loc[1]] != 'S'):
114                 a_grid[curr_loc[0]][curr_loc[1]] = 'P'
115
116         return a_grid

```

2 Part 2: Alpha-Beta Pruning

Our alpha-beta pruning solution first reads in a line from its input file and builds a dictionary of Node objects using the information it finds about each node. It then adds references to other nodes by finding the child node in the dictionary and adding a reference to it in the parent. For nodes which only have leaf nodes as children, we append the value of each leaf node to a list of child values. The Node class is implemented in Python as such:

```

1 class Node:
2     node_count = 0
3
4     def __init__(self, letter, minmax, value=-1):
5         self.letter = letter
6         self.min = minmax
7         self.values = []
8         self.children = []
9         Node.node_count += 1
10
11     def valueSetter(self, value):
12         self.values.append(value)
13
14     def childrenSetter(self, value):

```

```

15         self.children.append(value)
16
17     def alpha_beta(self, a, b):
18         print("\nNode %s" % self.letter)
19         print("Min: %s" % self.min)
20         print("Alpha: %s" % a + "\nBeta: %s" % b)
21         examined = 0
22         if (len(self.children) == 0):
23             if self.min:
24                 for x in self.values:
25                     examined += 1
26                     b = min(b, x)
27                     if b <= a:
28                         print("\nNode: %s" % self.letter + "\nBeta
29 %s" % b + " is better than alpha %s" % a)
30                         return (b, examined)
31                 return (b, examined)
32             else:
33                 for x in self.values:
34                     examined += 1
35                     a = max(a, x)
36                     if a >= b:
37                         print("\nNode: %s" % self.letter + "\nAlpha
38 %s" % a + " is better than beta %s" % b)
39                         return (a, examined)
40                 return (a, examined)
41             if self.min:
42                 for child in self.children:
43                     childValue = child.alpha_beta(a, b)
44                     best = childValue[0]
45                     examined += childValue[1]
46                     b = min(b, best)
47                     if b <= a:
48                         print("\nNode: %s" % self.letter + "\nBeta
49 %s" % b + " is better than alpha %s" % a)
50                         return (b, examined)
51                 return (b, examined)
52             else:
53                 print("Max node")
54                 for child in self.children:
55                     childValue = child.alpha_beta(a, b)
56                     best = childValue[0]
57                     examined += childValue[1]
58                     a = max(a, best)
59                     if a >= b:
60                         print("\nNode: %s" % self.letter + "\nAlpha
61 %s" % a + " is better than beta %s" % b)
62                         return (a, examined)
63                 return (a, examined)

```

In order to determine the max, or min, value of the game tree, the alpha_beta function of the root node can be called with the arguments negative infinity, for alpha, and positive infinity, for beta. The root node will then call the same function in its children, as will they on their children. Once a node which only has leaf nodes as children has its alpha_beta function invoked, if it is a min node then it will look for the leaf with the smallest value less than beta, stopping the search and returning its minimum value if it ever encounters a value less than alpha. For max nodes, the same process is followed, trying to find a new

maximum value which is greater than alpha while still less than beta, returning its maximum value encountered if it finds such a value.

Nodes employ the same strategy all the way back up the tree, using the values returned by its children as its potential max or min values instead of the static evaluation of leaf nodes. Once the search of all children has completed, although not all children have necessarily been examined, the root node returns the best value it has encountered, alpha if it is a max node and beta if it is a min node.

Our solution also keeps track of the number of examined leaf nodes. A node whose children are all leaf nodes counts how many it examines and returns that value once it is done. Nodes with non-leaf children sum the number of leaf nodes examined by its children. Since nodes stop their search if they encounter a value which is better for their opponent than their current best, not all leaf nodes need to be examined. Therefore, the total number of leaf nodes examined may be fewer than the number of leaf nodes in the game tree.

Once the tree has been solved, and the solution values written to the output file, successive lines are read in and solved until no trees remain.