CISC352 - Assignment 02

Gabriele Cimolino - 10009098
Luis Rivera-Wong - 10142361
Zach Slater - 10196265
Tom Szendrey - 10187030

March 16, 2018

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 is done through the help of the following helper functions.

```python
def reader(input_file, output_file):
    with open(input_file) as file:
        single_grid = []
        for line in file:
            if (not((line == '\n' or line == ''))):
                a_line = line.split()
                a_list_first = a_line[0]
                the_chars = list(a_list_first)
                single_grid.append(the_chars)
            if (line == '\n' or line == ''):
                s_loc = target_finder(single_grid, 'S')
                g_loc = target_finder(single_grid, 'G')
                return_grid_greedy = greedy_a(single_grid, s_loc, g_loc)
                return_grid_a_star = a_star_a(single_grid, s_loc, g_loc)
                writer(output_file, 'Greedy', return_grid_greedy, 'A*', return_grid_a_star)
                single_grid = []
        s_loc = target_finder(single_grid, 'S')
        g_loc = target_finder(single_grid, 'G')
        return_grid_greedy = greedy_a(single_grid, s_loc, g_loc)
        return_grid_a_star = a_star_a(single_grid, s_loc, g_loc)
        writer(output_file, 'Greedy', return_grid_greedy, 'A*', return_grid_a_star)
        single_grid = []
```

```python
def writer(filename, grid1_name, grid1, grid2_name, grid2):
    with open(filename, 'a+') as f:
        f.write(grid1_name + '\n')
        for i in grid1:
            f.write(''.join(i) + '\n')
        f.write(grid2_name + '\n')
        for j in grid2:
            f.write(''.join(j) + '\n')
        f.write('\n')
```

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

```
def target_finder(grid, target):
    location = []
    for i in range(len(grid)):
        for j in range(len(grid[i])):
            if (grid[i][j] == target):
                location.append(i)
                location.append(j)
    return location
```

A*

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

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.

```
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, we initialize the needed priority queue, boolean flag to determine if a solution has been found, and variable containing the 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*.

```
def manhattan(a, b):
    return abs(a[0] - b[0]) + abs(a[1] - b[1])
```

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 spot 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*.

```
def a_star_a(tmp_grid, s_loc, g_loc):
    a_grid = copy.deepcopy(tmp_grid)
```

```
    came_from = [[None for j in range(len(a_grid[0]))] for i in range(len(a_grid))]
    cost = [[None for j in range(len(a_grid[0]))] for i in range(len(a_grid))]

    cost[s_loc[0]][s_loc[1]] = 0

    curr_loc = None
    foundGoal = False
    visited = []

    heappush(visited, (manhattan(s_loc, g_loc), s_loc))

    # Find the goal
    while(not(foundGoal)):
        curr_node = heappop(visited)
        curr_loc = curr_node[1]

        if(a_grid[curr_loc[0]][curr_loc[1]] == 'G'):
            foundGoal = True

        if(not(foundGoal)):
            # Left
            if (curr_loc[1] - 1 >= 0):
                if (a_grid[curr_loc[0]][curr_loc[1] - 1] == '_' or a_grid[curr_loc[0]][curr_loc[1] - 1] == 'G'):
                    cost_so_far = cost[curr_loc[0]][curr_loc[1]]


                    if (cost[curr_loc[0]][curr_loc[1] - 1] == None or cost_so_far + 1 < cost[curr_loc[0]][curr_loc[1] - 1]):
                        cost[curr_loc[0]][curr_loc[1] - 1] = cost_so_far + 1
                        came_from[curr_loc[0]][curr_loc[1] - 1] = [curr_loc[0], curr_loc[1]]
                        heappush(visited, (cost_so_far + 1 + manhattan([curr_loc[0], curr_loc[1] - 1], g_loc), [curr_loc[0], curr_loc[1] - 1]))
            # Right
            if (curr_loc[1] + 1 < len(a_grid[0])):
                if (a_grid[curr_loc[0]][curr_loc[1] + 1] == '_' or a_grid[curr_loc[0]][curr_loc[1] + 1] == 'G'):
                    cost_so_far = cost[curr_loc[0]][curr_loc[1]]

                    if (cost[curr_loc[0]][curr_loc[1] + 1] == None or cost_so_far + 1 < cost[curr_loc[0]][curr_loc[1] + 1]):
                        cost[curr_loc[0]][curr_loc[1] + 1] = cost_so_far + 1
                        came_from[curr_loc[0]][curr_loc[1] + 1] = [curr_loc[0], curr_loc[1]]
                        heappush(visited, (cost_so_far + 1 + manhattan([curr_loc[0], curr_loc[1] + 1], g_loc), [curr_loc[0], curr_loc[1] + 1]))

            # Up
            if (curr_loc[0] - 1 >= 0):
                if (a_grid[curr_loc[0] - 1][curr_loc[1]] == '_' or a_grid[curr_loc[0] - 1][curr_loc[1]] == 'G'):
                    cost_so_far = cost[curr_loc[0]][curr_loc[1]]

                    if (cost[curr_loc[0] - 1][curr_loc[1]] == None or cost_so_far + 1 < cost[curr_loc[0] - 1][curr_loc[1]]):
```

```python
                        cost[curr_loc[0] - 1][curr_loc[1]] = cost_so_far + 1
                        came_from[curr_loc[0] - 1][curr_loc[1]] = [curr_loc[0], curr_loc[1]]
                        heappush(visited, (cost_so_far + 1 + manhattan([curr_loc[0] - 1, curr_loc[1]], g_loc), [curr_loc[0] - 1, curr_loc[1]]))

            # Down
            if (curr_loc[0] + 1 < len(a_grid)):
                if (a_grid[curr_loc[0] + 1][curr_loc[1]] == '_' or a_grid[curr_loc[0] + 1][curr_loc[1]] == 'G'):
                    cost_so_far = cost[curr_loc[0]][curr_loc[1]]

                    if (cost[curr_loc[0] + 1][curr_loc[1]] == None or cost_so_far + 1 < cost[curr_loc[0] + 1][curr_loc[1]]):
                        cost[curr_loc[0] + 1][curr_loc[1]] = cost_so_far + 1
                        came_from[curr_loc[0] + 1][curr_loc[1]] = [curr_loc[0], curr_loc[1]]
                        heappush(visited, (cost_so_far + 1 + manhattan([curr_loc[0] + 1, curr_loc[1]], g_loc), [curr_loc[0] + 1, curr_loc[1]]))

    while(came_from[curr_loc[0]][curr_loc[1]] != None):
        curr_loc = came_from[curr_loc[0]][curr_loc[1]]
        if(a_grid[curr_loc[0]][curr_loc[1]] != 'S'):
            a_grid[curr_loc[0]][curr_loc[1]] = 'P'

    return a_grid
```

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 takes the same inputs, a grid, and the location of the *Start* and *Goal* cells.

```python
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 function to allow for the diagonal movement as it works as a radial distance measure.

```python
def cheb(a, b):
    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. We then pop the next available node from the queue and look for empty spaces or the *Goal* cell 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 second version of the A* pathfinder.

```python
def a_star_b(tmp_grid, s_loc, g_loc):
    a_grid = copy.deepcopy(tmp_grid)
    came_from = [[None for j in range(len(a_grid[0]))] for i in range(len(a_grid))]
    cost = [[None for j in range(len(a_grid[0]))] for i in range(len(a_grid))]

    cost[s_loc[0]][s_loc[1]] = 0

    curr_loc = None
    foundGoal = False
    visited = []

    heappush(visited, (cheb(s_loc, g_loc), s_loc))

    # Find the goal
    while(not(foundGoal)):
        curr_node = heappop(visited)
        curr_loc = curr_node[1]

        if (a_grid[curr_loc[0]][curr_loc[1]] == 'G'):
            foundGoal = True

        if(not(foundGoal)):
            # Left
            if (curr_loc[1] - 1 >= 0):
                if (a_grid[curr_loc[0]][curr_loc[1] - 1] == '_' or a_grid[curr_loc[0]][curr_loc[1] - 1] == 'G'):
                    cost_so_far = cost[curr_loc[0]][curr_loc[1]]

                    if(cost[curr_loc[0]][curr_loc[1] - 1] == None or cost_so_far + 1 < cost[curr_loc[0]][curr_loc[1] - 1]):
                        cost[curr_loc[0]][curr_loc[1] - 1] = cost_so_far + 1
                        came_from[curr_loc[0]][curr_loc[1] - 1] = [curr_loc[0], curr_loc[1]]
                        heappush(visited, (cost_so_far + 1 + cheb([curr_loc[0], curr_loc[1] - 1], g_loc), [curr_loc[0], curr_loc[1] - 1]))

            # Right
            if(curr_loc[1] + 1 < len(a_grid[0])):
                if (a_grid[curr_loc[0]][curr_loc[1] + 1] == '_' or a_grid[curr_loc[0]][curr_loc[1] + 1] == 'G'):
                    cost_so_far = cost[curr_loc[0]][curr_loc[1]]

                    if (cost[curr_loc[0]][curr_loc[1] + 1] == None or cost_so_far + 1 < cost[curr_loc[0]][curr_loc[1] + 1]):
                        cost[curr_loc[0]][curr_loc[1] + 1] = cost_so_far + 1
                        came_from[curr_loc[0]][curr_loc[1] + 1] = [curr_loc[0], curr_loc[1]]
                        heappush(visited, (cost_so_far + 1 + cheb([curr_loc[0], curr_loc[1] + 1], g_loc), [curr_loc[0], curr_loc[1] + 1]))

            #Up
            if (curr_loc[0] - 1 >= 0):
                if (a_grid[curr_loc[0] - 1][curr_loc[1]] == '_' or a_grid[curr_loc[0] - 1][curr_loc[1]] == 'G'):
                    cost_so_far = cost[curr_loc[0]][curr_loc[1]]

```

```
                if (cost[curr_loc[0] - 1][curr_loc[1]] == None or cost_so_far + 1 < cost[curr_loc[0] - 1][curr_loc[1]]):
                    cost[curr_loc[0] - 1][curr_loc[1]] = cost_so_far + 1
                    came_from[curr_loc[0] - 1][curr_loc[1]] = [curr_loc[0], curr_loc[1]]
                    heappush(visited, (cost_so_far + 1 + cheb([curr_loc[0] - 1, curr_loc[1]], g_loc), [curr_loc[0] - 1, curr_loc[1]]))

        #Down
        if (curr_loc[0] + 1 < len(a_grid)):
            if (a_grid[curr_loc[0] + 1][curr_loc[1]] == '_' or a_grid[curr_loc[0] + 1][curr_loc[1]] == 'G'):
                cost_so_far = cost[curr_loc[0]][curr_loc[1]]


                if (cost[curr_loc[0] + 1][curr_loc[1]] == None or cost_so_far + 1 < cost[curr_loc[0] + 1][curr_loc[1]]):
                    cost[curr_loc[0] + 1][curr_loc[1]] = cost_so_far + 1
                    came_from[curr_loc[0] + 1][curr_loc[1]] = [curr_loc[0], curr_loc[1]]
                    heappush(visited, (cost_so_far + 1 + cheb([curr_loc[0] + 1, curr_loc[1]], g_loc), [curr_loc[0] + 1, curr_loc[1]]))

        #Up Left
        if (curr_loc[0] - 1 >= 0 and curr_loc[1] - 1 >= 0):
            if (a_grid[curr_loc[0] - 1][curr_loc[1] - 1] == '_' or a_grid[curr_loc[0] - 1][curr_loc[1] - 1] == 'G'):
                cost_so_far = cost[curr_loc[0]][curr_loc[1]]


                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]):
                    cost[curr_loc[0] - 1][curr_loc[1] - 1] = cost_so_far + 1
                    came_from[curr_loc[0] - 1][curr_loc[1] - 1] = [curr_loc[0], curr_loc[1]]
                    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]))

        # Up Right
        if(curr_loc[0] - 1 and curr_loc[1] + 1 < len(a_grid[0])):
            if(a_grid[curr_loc[0] - 1][curr_loc[1] + 1] == '_' or a_grid[curr_loc[0] - 1][curr_loc[1] + 1] == 'G'):
                cost_so_far = cost[curr_loc[0]][curr_loc[1]]


                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]):
                    cost[curr_loc[0] - 1][curr_loc[1] + 1] = cost_so_far + 1
                    came_from[curr_loc[0] - 1][curr_loc[1] + 1] = [curr_loc[0] - 1, curr_loc[1]]
                    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]))

            # Down Left
            if (curr_loc[0] + 1 < len(a_grid) and curr_loc[1] - 1 >= 0):
                if (a_grid[curr_loc[0] + 1][curr_loc[1] - 1] == '_' or a_grid[curr_loc[0] + 1][curr_loc[1] - 1] == 'G'):
                    cost_so_far = cost[curr_loc[0]][curr_loc[1]]


                    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]):
                        cost[curr_loc[0] + 1][curr_loc[1] - 1] = cost_so_far + 1
                        came_from[curr_loc[0] + 1][curr_loc[1] - 1] = [curr_loc[0], curr_loc[1]]
                        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]))

            # Down Right
            if(curr_loc[0] + 1 < len(a_grid) and curr_loc[1] + 1 < len(a_grid[0])):
                if(a_grid[curr_loc[0] + 1][curr_loc[1] + 1] == '_' or a_grid[curr_loc[0] + 1][curr_loc[1] + 1] == 'G'):
                    cost_so_far = cost[curr_loc[0]][curr_loc[1]]

                    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]):
                        cost[curr_loc[0] + 1][curr_loc[1] + 1] = cost_so_far + 1
                        came_from[curr_loc[0] + 1][curr_loc[1] + 1] = [curr_loc[0], curr_loc[1]]
                        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]))

    while (came_from[curr_loc[0]][curr_loc[1]] != None):
        curr_loc = came_from[curr_loc[0]][curr_loc[1]]
        if (a_grid[curr_loc[0]][curr_loc[1]] != 'S'):
            a_grid[curr_loc[0]][curr_loc[1]] = 'P'

    return a_grid
```

Greedy

A greedy algorithm attempts to solve a problem by making the locally optimal choice at each stage in the hopes of finding the global optimum. In pathfinding, the algorithm will find the heuristic values for all of its available options and choose the best option. Greedy search may make choices that lead to a dead-end. This is why this algorithm is considered incomplete.

Greedy_a

This version of the greedy algorithm takes 3 inputs. The first being a 2-dimensional array representing the grid, or layout of the stage. The second being a tuple or list, of 2 elements, for the coordinates of the start location. The last being a tuple, or list of 2 for the coordinates of the goal location.

This function must follow these steps.

Step 1:
Set all cardinal directions heuristic value to infinite. This is done so that any possible heuristic value calculated later on will be better than the initial state.

```
def greedy_a(tmp_grid, s_loc, g_loc):
    a_grid = copy.deepcopy(tmp_grid)
    curr_loc = copy.deepcopy(s_loc)
    prev_dir = "None"
    stuck = False
    while (not(stuck)):
        for x in a_grid:
            left_dist = math.inf
            right_dist = math.inf
            up_dist = math.inf
            down_dist = math.inf
```

Step 2:
Get all of the heuristic values for the standard directions. This is done by using the Manhattan distance helper function.

```
            # Left distance
            if ((curr_loc[1] - 1) >= 0):
                if (a_grid[curr_loc[0]][curr_loc[1] - 1] == 'G'):
                    return a_grid
                if (a_grid[curr_loc[0]][curr_loc[1] - 1] == '_'):
                    left_dist = manhattan([curr_loc[0],(curr_loc[1]-1)],g_loc)
            # Right distance
            if ((curr_loc[1] + 1) < len(a_grid[0])):
                if (a_grid[curr_loc[0]][curr_loc[1] + 1] == 'G'):
                    return a_grid
                if (a_grid[curr_loc[0]][curr_loc[1] + 1] == '_'):
                    right_dist = manhattan([curr_loc[0],(curr_loc[1]+1)],g_loc)
            # Up distance
            if ((curr_loc[0] - 1) >= 0):
                if(a_grid[(curr_loc[0]-1)][curr_loc[1]] == 'G'):
                    return a_grid
                if(a_grid[(curr_loc[0]-1)][curr_loc[1]] == '_'):
                    up_dist = manhattan([(curr_loc[0]-1),curr_loc[1]],g_loc)
            # Down distance
            if ((curr_loc[0] + 1) < len(a_grid)):
                if (a_grid[(curr_loc[0]+1)][curr_loc[1]] == 'G'):
                    return a_grid
                if (a_grid[(curr_loc[0]+1)][curr_loc[1]] == '_'):
                    down_dist = manhattan([(curr_loc[0]+1),curr_loc[1]],g_loc)
```

Step 3:
Now that we have the heuristic values for each direction all we must do is select the lowest possible value. If there is a tie select any one of the lowest values. The previous direction taken is stored to allow the function to know whether or not it is backtracking. If the greedy algorithm attempts to go backwards it will instead finish executing as the algorithm has then failed to find a solution. Repeat the previous steps until you are either stuck, or at a goal.

```
            if (left_dist == math.inf and right_dist == math.inf and up_dist == math.inf and down_dist == math.inf):
                stuck = True
                return False

            if (not(stuck)):
                min_index = randomMinIndex([up_dist, down_dist, left_dist, right_dist])
                if (prev_dir == "Down" and min_index == 0):
                    return False
                if (prev_dir == "Up" and min_index == 1):
                    return False
                if (prev_dir == "Right" and min_index == 2):
                    return False
                if (prev_dir == "Left" and min_index == 3):
                    return False

                if (min_index == 2):
                    if (prev_dir == "Right"):
                        return False
                    prev_dir = "Left"
                    curr_loc[1] -= 1
                    a_grid[curr_loc[0]][curr_loc[1]] = 'P'
                elif (min_index == 3):
                    if (prev_dir == "Left"):
                        return False
                    prev_dir = "Right"
                    curr_loc[1] += 1
                    a_grid[curr_loc[0]][curr_loc[1]] = 'P'
                elif (min_index == 0):
                    if (prev_dir == "Down"):
                        return False
                    prev_dir = "Up"
                    curr_loc[0] -= 1
                    a_grid[curr_loc[0]][curr_loc[1]] = 'P'
                elif (min_index == 1):
                    if (prev_dir == "Up"):
                        return False
                    prev_dir = "Down"
                    curr_loc[0] += 1
                    a_grid[curr_loc[0]][curr_loc[1]] = 'P'
```

Greedy_b

This variant is almost identical to greedy_a. The main differences are: instead of using Manhattan to find the heuristic values we use Chebyshev distance. Also now when calculating heuristic values, and which direction to go we consider four more options (up_right,up_left,down_right,down_left).

```
def greedy_b(tmp_grid, s_loc, g_loc):
    a_grid = copy.deepcopy(tmp_grid)
    curr_loc = copy.deepcopy(s_loc)
    prev_dir = "None"
    stuck = False
    while (not(stuck)):
        for x in a_grid:
            left_dist = math.inf
            right_dist = math.inf
            up_dist = math.inf
            down_dist = math.inf
            up_right_dist = math.inf
```

```
        up_left_dist = math.inf
        down_right_dist = math.inf
        down_left_dist = math.inf
        # Get the H values for all the directions.
        # Left distance
        if ((curr_loc[1] - 1) >= 0):
            if(a_grid[curr_loc[0]][curr_loc[1] - 1] == 'G'):
                return a_grid
            if(a_grid[curr_loc[0]][curr_loc[1] - 1] == '_'):
                left_dist = cheb([curr_loc[0],(curr_loc[1]-1)], g_loc)
        # Right distance
        if ((curr_loc[1] + 1) < len(a_grid[0])):
            if(a_grid[curr_loc[0]][curr_loc[1] + 1] == 'G'):
                return a_grid
            if(a_grid[curr_loc[0]][curr_loc[1] + 1] == '_'):
                right_dist = cheb([curr_loc[0],(curr_loc[1]+1)],g_loc)
        # Up distance
        if ((curr_loc[0] - 1) >= 0):
            if (a_grid[(curr_loc[0]-1)][curr_loc[1]] == 'G'):
                return a_grid
            if (a_grid[(curr_loc[0]+1)][curr_loc[1]] == '_'):
                up_dist = cheb([(curr_loc[0]-1),curr_loc[1]], g_loc)
        # Down distance
        if ((curr_loc[0] + 1) < len(a_grid)):
            if (a_grid[(curr_loc[0]+1)][curr_loc[1]] == 'G'):
                return a_grid
            if (a_grid[(curr_loc[0]+1)][curr_loc[1]] == '_'):
                down_dist = cheb([(curr_loc[0]+1),curr_loc[1]], g_loc)
        #Up + Right (diagonal)
        if (((curr_loc[0] - 1) >= 0) and ((curr_loc[1] + 1) < len(a_grid[0]))):
        #                       up           right
            if (a_grid[curr_loc[0]-1][curr_loc[1]+1] == 'G'):
                return a_grid
        #                       up           right
            if (a_grid[curr_loc[0]-1][curr_loc[1]+1] == '_'):
                up_right_dist = cheb([(curr_loc[0]-1),(curr_loc[1]+1)],g_loc)
        #Up + Left (diagonal)
        if (((curr_loc[0] - 1) >= 0) and ((curr_loc[1] - 1) < len(a_grid[0]))):
        #                       up           Left
            if (a_grid[curr_loc[0]-1][curr_loc[1]-1] == 'G'):
                return a_grid
        #                       up           Left
            if (a_grid[curr_loc[0]-1][curr_loc[1]-1] == '_'):
                up_left_dist = cheb([(curr_loc[0]-1),(curr_loc[1]-1)],g_loc)
        #Down + Right (diagonal)
        if (((curr_loc[0] + 1) >= 0) and ((curr_loc[1] + 1) < len(a_grid[0]))):
        #                       Down           right
            if (a_grid[curr_loc[0]+1][curr_loc[1]+1] == 'G'):
                return a_grid
        #                       Down           right
            if (a_grid[curr_loc[0]+1][curr_loc[1]+1] == '_'):
                down_right_dist = cheb([(curr_loc[0]+1),(curr_loc[1]+1)],g_loc)
```

```
        #Down + Left (diagonal)
        if (((curr_loc[0] + 1) >= 0) and ((curr_loc[1] - 1) < len(a_grid[0]))):
        #                       Down            Left
            if (a_grid[curr_loc[0]+1][curr_loc[1]-1] == 'G'):
                return a_grid
        #                       Down            Left
            if (a_grid[curr_loc[0]+1][curr_loc[1]-1] == '_'):
                down_left_dist = cheb([(curr_loc[0]+1),(curr_loc[1]-1)],g_loc)

        if (left_dist == math.inf and right_dist == math.inf and up_dist == math.inf and down_dist == math.inf):
            stuck = True
            return False

        if (not(stuck)):
            min_index = randomMinIndex([up_dist, down_dist, left_dist, right_dist, up_right_dist, up_left_dist, down_right_dist, down_left_dist])
            if (prev_dir == "Down" and min_index == 0):
                return False
            if (prev_dir == "Up" and min_index == 1):
                return False
            if (prev_dir == "Right" and min_index == 2):
                return False
            if (prev_dir == "Left" and min_index == 3):
                return False

            #Diagonal Directions
            if (min_index == 4):
                curr_loc[0] -= 1
                curr_loc[1] += 1
                a_grid[curr_loc[0]][curr_loc[1]] = 'P'
            elif (min_index == 5):
                curr_loc[0] -= 1
                curr_loc[1] -= 1
                a_grid[curr_loc[0]][curr_loc[1]] = 'P'
            elif (min_index == 6):
                curr_loc[0] += 1
                curr_loc[1] += 1
                a_grid[curr_loc[0]][curr_loc[1]] = 'P'
            elif (min_index == 7):
                curr_loc[0] += 1
                curr_loc[1] -= 1
                a_grid[curr_loc[0]][curr_loc[1]] = 'P'

            #Cardinal Directions
            elif (min_index == 2):
                curr_loc[1] -= 1
                a_grid[curr_loc[0]][curr_loc[1]] = 'P'
            elif (min_index == 3):
                curr_loc[1] += 1
                a_grid[curr_loc[0]][curr_loc[1]] = 'P'
            elif (min_index == 0):
                curr_loc[0] -= 1
                a_grid[curr_loc[0]][curr_loc[1]] = 'P'
            elif (min_index == 1):
                curr_loc[0] += 1
                a_grid[curr_loc[0]][curr_loc[1]] = 'P'
```

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:

```python
class Node:
    def __init__(self, letter, minmax, value=-1):
        self.letter = letter
        self.min = minmax
        self.values = []
        self.children = []

    def valueSetter(self, value):
        self.values.append(value)

    def childrenSetter(self, value):
        self.children.append(value)

    def alpha_beta(self, a, b):
        examined = 0
        if(len(self.children) == 0):
            if self.min:
                for x in self.values:
                    examined += 1
                    b = min(b, x)
                    if b <= a:
                        return (b, examined)
                return (b,examined)
            else:
                for x in self.values:
                    examined += 1
                    a = max(a, x)
                    if a >= b:
                        return (a, examined)
                return (a, examined)
        else:
            if self.min:
                for child in self.children:
                    childValue = child.alpha_beta(a, b)
                    best = childValue[0]
                    examined += childValue[1]
                    b = min(b, best)
                    if b <= a:
                        return (b, examined)
                return (b, examined)
            else:
                for child in self.children:
                    childValue = child.alpha_beta(a, b)
                    best = childValue[0]
                    examined += childValue[1]
                    a = max(a, best)
                    if a >= b:
                        return (a, examined)
                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.