Laboratory 1

Variant 2

Group 12

By Dimitrios Georgousis and Andrea Amato

**Introduction**

Task description: The maze is a 2D grid with empty spaces, walls, a start, and an end position. Our task is to write an algorithm, a computer program, which will manage to solve the maze – arrive at the end position – using greedy best-first search algorithm.

Greedy best-first search algorithm is an algorithm which heavily depends on a heuristic function. On each step or iteration of the algorithm, it takes into account heuristic values and expands the path with the best such value.

This algorithm is usually used for problems where one might want to find the optimal path or solution in a large search space. It can be used for shortest path problems, but, also, for optimization or constraint satisfaction problems if the heuristic function is adjusted properly.

We shall mention again that this algorithm depends very heavily on the value of the heuristic function. The structure of the algorithm has some side effects.

Advantages: it typically searches a narrower part of the search space thus finding a candidate solution quite fast and also by not using up that many resources. This behavior can be explained by considering that at each step only the node with the lowest heuristic value is expanded and this, often, leads to nodes with lower heuristic values, hence, closer to a solution. We can say that the ‘search-beam’ of the greedy best-first algorithm is narrower than others.

Disadvantages: the greediness of the approach has the obvious drawback that the solution found is not guaranteed to be the best or optimal, it just usually is good enough. Also, in some cases, considering only the heuristic value can lead the algorithm down a path which seems good for quite some time but ends up being a dead-end. The algorithm will need to back-track and/or follow alternative paths which could end up being longer than what could have otherwise been found.

**Implementation**

We have done a basic implementation of the greedy best-first search algorithm. There is one difference between our code and the guidelines which is that our greedy function has an extra argument which is the heuristic function to be used. We present the code:

def greedy(maze, start, finish, heuristic):

    """

    Greedy best-first search

    Parameters:

    - maze: The 2D matrix that represents the maze with 0 represents empty space and 1 represents a wall

    - start: A tuple with the coordinates of starting position

    - finish: A tuple with the coordinates of finishing position

    - heuristic: A function that takes two tuples and returns a number,

      representing the estimated cost to reach the finish from the start

    Returns:

    - Number of steps from start to finish, equals -1 if the path is not found

    - Viz - everything required for step-by-step vizualization

    """

    frontier = PriorityQueue()

    frontier.put((0, start))

    predecessor = {}

    explored = set()

    predecessor[start] = None

    no\_expanded\_nodes = 0

    search\_history = [(copy.deepcopy(frontier.queue), copy.deepcopy(explored))]

    while not frontier.empty():

        current = frontier.get()[1]

        if current == finish:

            break

        no\_expanded\_nodes += 1

        for next in get\_neighbors(maze, current):

            if next not in explored and all(next != pos for \_, pos in frontier.queue):

                priority = heuristic(next, finish)

                frontier.put((priority, next))

                predecessor[next] = current

        explored.add(current)

        search\_history.append((copy.deepcopy(frontier.queue), copy.deepcopy(explored)))

    path = []

    while current != start:

        path.append(current)

        current = predecessor[current]

    path.append(start)

    path.reverse()

    return (

        len(path) - 1 if path[-1] == finish else -1, # -1 if the path is not found

        (maze, path, finish, search\_history, no\_expanded\_nodes), # no\_expanded\_nodes can be

                                                                 # used for analysis

    )

We use a priority queue to keep track of the frontier (positions in the maze to be explored). This queue gives priority to tuples with lowest heuristic value, which is the behavior we want. A predecessor dictionary is used to keep track of nodes in the solution “path” that may be eventually found. The explored set keeps track of nodes we’ve already explored, so we don’t repeat the same actions and the search-history list is a list that keeps track of the states of the frontier and the explored set at each iteration of the algorithm. This list is used for display purposes in the visualization function.

Step by step explanation of the solution: The frontier contains maze cells, which are neighbors of explored cells, the algorithm picks the lowest priority (lowest heuristic value) cell and looks up its neighbors. If a new neighbor is discovered, it is added to the queue with priority equal to its heuristic value. The current node is marked as explored and the search continues until either the finish/target node/cell is found or all cells have been explored (a dead-end is reached). The search of course starts at the start cell of the maze/grid.

Finally, as is evident from the code, the first return value of the “greedy” function is the path length, if a path-solution is indeed found. We understood the “number of steps from start to finish” requirement as meaning the path length.

Next we present the display function used:

def vizualize(viz):

    """

    Vizualization function. Shows step by step the work of the search algorithm.

    Symbols used in visualization:

        S: start

        F: finish

        #: wall

        .: empty

        \*: explored

        o: frontier

        P: maze cell in the final path

    Parameters:

    - viz: everything required for step-by-step vizualization

    """

    maze, path, finish, search\_history, \_ = viz

    print("Maze:")

    for i, row in enumerate(maze):

            for j, cell in enumerate(row):

                if (i, j) == path[0]:

                    print('S', end = ' ')

                elif (i, j) == finish:

                    print('F', end = ' ')

                elif cell == 1:

                    print('#', end = ' ')

                else:

                    print('.', end = ' ')

            print()

    time.sleep(1.6)

    print('\n' \* 5)

    for phase, (frontier, explored) in enumerate(search\_history):

        print(f"Phase: {phase}")

        for i, row in enumerate(maze):

            for j, cell in enumerate(row):

                if (i, j) == path[0]:

                    print('S', end = ' ')

                elif (i, j) == finish:

                    print('F', end = ' ')

                elif (i, j) in explored:

                    print('\*', end = ' ')

                elif any((i, j) ==  position for \_, position in frontier):

                    print('o', end = ' ')

                elif cell == 1:

                    print('#', end = ' ')

                else:

                    print('.', end = ' ')

            print()

        time.sleep(0.2)

        print('\n' \* 5)

    # Finally, display the final path

    if path[-1] == finish:

        print("Final Path:")

        for i, row in enumerate(maze):

                for j, cell in enumerate(row):

                    if (i, j) in path:

                        if (i, j) == path[0]:

                            print(f'{YELLOW}S{RESET}', end = ' ')

                        elif (i, j) == path[-1]:

                            if (i, j) == finish:

                                print(f'{YELLOW}F{RESET}', end = ' ')

                            else:

                                print(f'\*', end = ' ')

                        else:

                            print(f'{YELLOW}P{RESET}', end = ' ')

                    elif cell == 1:

                        print('#', end = ' ')

                    elif (i, j) in explored:

                        print('\*', end = ' ')

                    elif any((i, j) ==  position for \_, position in frontier):

                        print('o', end = ' ')

                    else:

                        print('.', end = ' ')

                print()

    else:

        print("No path found.")

This function has a very simple structure. The output is explained in the description comment at its start. For the sake of conciseness the input argument of “visualize” is of the same form as the second return value of the “greedy” function.

A visualization of the code follows. It shows the steps or, as described in our code, phases of the algorithm for a 5x5 maze, start = (0, 0) and finish = (4, 4).

Using euclidean heuristic:

Path from (0, 0) to (4, 4) using greedy best-first search is 12 steps.

Maze:

S . . . .

. # # # .

. # . . .

. # . # #

. . . . F

Phase: 0

S . . . .

. # # # .

. # . . .

. # . # #

. . . . F

Phase: 1

S o . . .

o # # # .

. # . . .

. # . # #

. . . . F

Phase: 2

S \* o . .

o # # # .

. # . . .

. # . # #

. . . . F

Phase: 3

S \* \* o .

o # # # .

. # . . .

. # . # #

. . . . F

Phase: 4

S \* \* \* o

o # # # .

. # . . .

. # . # #

. . . . F

Phase: 5

S \* \* \* \*

o # # # o

. # . . .

. # . # #

. . . . F

Phase: 6

S \* \* \* \*

o # # # \*

. # . . o

. # . # #

. . . . F

Phase: 7

S \* \* \* \*

o # # # \*

. # . o \*

. # . # #

. . . . F

Phase: 8

S \* \* \* \*

o # # # \*

. # o \* \*

. # . # #

. . . . F

Phase: 9

S \* \* \* \*

o # # # \*

. # \* \* \*

. # o # #

. . . . F

Phase: 10

S \* \* \* \*

o # # # \*

. # \* \* \*

. # \* # #

. . o . F

Phase: 11

S \* \* \* \*

o # # # \*

. # \* \* \*

. # \* # #

. o \* o F

Phase: 12

S \* \* \* \*

o # # # \*

. # \* \* \*

. # \* # #

. o \* \* F

Final Path:

S P P P P

o # # # P

. # P P P

. # P # #

. o P P F

We came up with test cases such as:

* Maze with dead ends
* Maze with sparse walls
* An empty maze (also a corner case)
* Some larger mazes
* Mazes where one of the heuristics performs better than the other.
* An unsolvable maze with only walls (except for the start and finish positions)

**Additional Code**

In order to gather some statistical information and be able to make better comparisons between the chosen heuristics, we implemented a simple maze generator and then we use that to gather information on path lengths and corresponding cells/nodes expanded by the different algorithms. This helps in making conclusions.

We show the code for maze generation:

# start position should be on the border of the maze and size should be an odd number...

# To see why, read comments bellow.

def generate\_maze(size, start, finish, threshold=0.08):

    """

    size: integer that indicates the size of the NxN grid of the maze

    start: pair of integers that indicates the coordinates of the starting point (S)

    finish: pair of integers that indicates the coordinates of the finish point (F)

    You can add any other parameters you want to customize maze creation (e.g. variables that

    control the creation of additional paths)

    """

    assert size > 2

    # Given a tuple, this algorithm works by finding its neighbors by skipping one cell in each direction. Thus, if

    # the start is on the border of the maze, then the algorithm will never consider turning the cells on some of the borders

    # into empty cells, leading to problems and bugs. To combat this, we add the following assertion.

    assert size % 2 == 1

    ## Initialize grid

    grid = np.ones((size, size), dtype=bool)

    def neighbors(node, size, visited, threshold):

        """

        Returns all neighbors of a node that are either unvisited, or they are visited but

        there is a wall between the node and the neighbor and the neighbor passes a random test.

        """

        l = []

        x, y = node

        # first condition in all checks is for boundaries

        # neighbors are +-2 in x or y

        # walls are +-1

        if x > 1 and (visited[x-2, y] or (visited[x-1,y] and random.uniform(0,1) <= threshold)):

            l.append((x-2, y))

        if x < size-2 and (visited[x+2, y] or (visited[x+1, y] and random.uniform(0,1) <= threshold)):

            l.append((x+2, y))

        if y > 1 and (visited[x, y-2] or (visited[x, y-1] and random.uniform(0,1) <= threshold)):

            l.append((x, y-2))

        if y < size-2 and (visited[x, y+2] or (visited[x, y+1] and random.uniform(0,1) <= threshold)):

            l.append((x, y+2))

        return l

    stack = []

    stack.append(start)

    grid[start] = False

    while stack:

        current\_node = stack.pop()

        # get all unvisited neighbors (and some visited ones with a random chance)

        n = neighbors(current\_node, size, grid, threshold)

        if len(n):

            stack.append(current\_node)

            # select a random neighbor

            next\_node = random.choice(n)

            # break the wall between current and next node

            # find wall!

            # the wall is the block between current and next, so we find that

            (x, y) = current\_node

            (x\_n, y\_n) = next\_node

            wall = ((x+x\_n)//2, (y+y\_n)//2)

            grid[wall] = False

            # mark next node as visited and add it to the stack

            grid[next\_node] = False

            stack.append(next\_node)

    grid[start] = False

    grid[finish] = False

    # turn grid to ints

    grid = grid.astype(int)

    return grid

The above code uses the following idea in order to generate a maze. Start with a grid that is full of walls. Consider an empty cell X, cells that are immediate neighbors of X are walls and cells that are two (2) steps away from X are just neighbors. Randomly turn some neighbors from walls to empty cells and also randomly turn the wall between two neighbors into an empty cell (thus connecting them). Continue doing so until you can’t find any more neighbors.

**Discussion**

Test cases:

We have implemented test cases where the maze has solutions and the starting or finishing positions are on the borders of the maze or not. We, also, included some test cases where the finish is unreachable just to observe the behavior of our algorithm in those cases.

An interesting phenomenon appears in mazes such as this:

Maze:

S . . . .

. # # # .

. # . . .

. # . # #

. . . . F

This maze also happens to be the one included as a visualization example earlier in the report. The path that is ultimately found is the following:

Final Path:

S P P P P

o # # # P

. # P P P

. # P # #

. o P P F

One may notice here that the path is indeed not the optimal. The optimal would be to go straight down and then move right. This test case, also, shows the weakness of the algorithm. By relying solely on the heuristic as an evaluation method for each cell, it ends up making a ‘mistake’ in the path it follows, which results in it following a longer path than was needed.

One of the reasons our algorithm finds the above path can be found in the way we get the neighbors of a cell. More specifically, the following function:

# Returns valid neighbors

def get\_neighbors(maze, position):

    x, y = position

    # The four traversable directions (up, down, left, right)

    directions = [(-1, 0), (1, 0), (0, 1), (0, -1)]

    neighbors = []

    for d in directions:

        neighbor = (x + d[0], y + d[1])

        x\_n, y\_n = neighbor

        if 0 <= x\_n < len(maze) and 0 <= y\_n < len(maze[0]) and not maze[x\_n][y\_n]:

            neighbors.append(neighbor)

    return neighbors

It is obvious that the returned “neighbors” list will have the horizontal neighbors first and then the vertical neighbors will follow.

This leads to our algorithm showing preference to following horizontal paths first. Finally, this explains the observed behavior in the above example.

One way to combat this intrinsic bias would be to randomly shuffle the neighbors list before returning it.

Last but not least, we used the maze generator to get some statistic information. We will show the code and results for that and then comment upon it.

random.seed(17) # so that results are reproducible

results = {}

for size in [41, 61, 81, 101]:

    results[size] = {}

    for \_ in range(20):

        maze = generate\_maze(size, (0, 0), (size-1, size-1), 0.04)

        for heur\_name, heur in [('euclidean', euclidean), ('manhattan', manhattan)][:]:

            path\_length, (maze, path, finish, search\_history, no\_expanded\_nodes) = greedy(maze, (0, 0), (size-1, size-1), heur)

            if heur\_name not in results[size]:

                results[size][heur\_name] = (path\_length, no\_expanded\_nodes)

            else:

                pl, no\_e\_n = results[size][heur\_name]

                results[size][heur\_name] = (pl + path\_length, no\_e\_n + no\_expanded\_nodes)

    # get an average of the results

    for heur\_name in ['euclidean', 'manhattan']:

        results[size][heur\_name] = (results[size][heur\_name][0] / 20, results[size][heur\_name][1] / 20)

    print(f"Results for {size}x{size} maze:")

    print(f"Average path length using euclidean heuristic: {results[size]['euclidean'][0]}")

    print(f"Average number of expanded nodes using euclidean heuristic: {results[size]['euclidean'][1]}")

    print(f"Average path length using manhattan heuristic: {results[size]['manhattan'][0]}")

    print(f"Average number of expanded nodes using manhattan heuristic: {results[size]['manhattan'][1]}")

    print()

Results for 41x41 maze:

Average path length using euclidean heuristic: 134.0

Average number of expanded nodes using euclidean heuristic: 176.65

Average path length using manhattan heuristic: 131.6

Average number of expanded nodes using manhattan heuristic: 184.0

Results for 61x61 maze:

Average path length using euclidean heuristic: 210.2

Average number of expanded nodes using euclidean heuristic: 292.65

Average path length using manhattan heuristic: 209.2

Average number of expanded nodes using manhattan heuristic: 286.95

Results for 81x81 maze:

Average path length using euclidean heuristic: 284.0

Average number of expanded nodes using euclidean heuristic: 391.15

Average path length using manhattan heuristic: 295.8

Average number of expanded nodes using manhattan heuristic: 405.15

Results for 101x101 maze:

Average path length using euclidean heuristic: 370.4

Average number of expanded nodes using euclidean heuristic: 531.15

Average path length using manhattan heuristic: 364.6

Average number of expanded nodes using manhattan heuristic: 521.75

Both heuristic functions provide similar results with the Manhattan distance being slightly better sometimes (giving smaller solution paths with also less expanded nodes). The similarity between the results of the two heuristic functions can be attributed up to a degree to the restricted part of the search space, that the greedy best-first search algorithm usually likes to look into as discussed in the ‘Introduction’ of this report, which forces the algorithm to follow similar paths no matter the heuristic used.

**Conclusion**

We learned how the greedy best-first search algorithm works and what behaviors it usually displays in search spaces. We looked up heuristic functions and found ones that are appropriate for maze problems. Also, we implemented such functions and observed the results of our algorithm when using those.

There was also the challenge of coming up with appropriate test cases, discovering corner cases and methods for gathering both useful and usable statistics on our data.

Our code could perhaps accept some changes to become more modular and, also, the issue with the ‘get\_neighbors’ function we mentioned could be improved upon.