Understanding Question

Raphael Haddad - 342810835
Daniel Perretz - 316153386

1.1 Depth First Search

No, the solution doesn't use all the moves that were checked.

1.2 Breadth First Search

Yes, BFS found the shortest path through pacman's maze.

2 BFS vs DFS

2.1 Optimality

1) DFS: Not optimal. It can find suboptimal solutions due to its nature of exploring deeply before considering breadth. In our case, DFS didn't find the optimal solution.
BFS: Optimal for finding the shallowest solution (fewest steps) when step costs are equal. Not guaranteed to be optimal in general cases where step costs vary.In packman, BFS finds the best solution.

2)

Cost	Blokus	Pacman
DFS	score: 17	total cost of 130
BFS	score: 17	total cost of 68

From these results, we can conclude for sure that DFS is not optimal, because in Pacman the cost is very expensive compared to BFS.
For BFS, we can't conclude that he is optimal.

3) Yes, the theoretical and empirical results are consistent with each other.

2.2 Running Time

b - Branching factor
d - depth of the shallowest solution
m - maximum depth of the search tree
1) DFS - O(b^m) because DFS explores as deeply as possible before backtracking.Thus, the worst-case time complexity is proportional to b^m.

BFS - O(b^d) because BFS explores all nodes at the present depth level before moving on to nodes at the next depth level. Thus, if the shallowest solution is at depth d, the number of nodes expanded is proportional to b^d.

Since d is the depth of the shallowest solution, we can conclude that d <= m. In general case, BFS has a better running time than DFS
2)

Expanded Nodes	Blokus	Pacman
DFS	Expanded nodes: 128	nodes expanded: 146
BFS	Expanded nodes: 3119	nodes expanded: 269

From these results, we can conclude that DFS expanded fewer nodes compared to BFS in the Pacman maze problem, indicating that DFS was faster in terms of the number of nodes expanded but BFS found the optimal path.
In the Blokus game scenario, where both DFS and BFS achieved the same score but DFS expanded much fewer nodes compared to BFS.

In the context of the Pacman maze problem, DFS might quickly find a path towards a goal state by exploring one path deeply before considering other paths. This nature allows DFS to explore less nodes but he doesn't guarantee that this path is optimal.

In the context of the Pacman maze problem, BFS systematically explores the maze level by level, ensuring that the shortest path is found before exploring deeper into the maze. BFS guarantees that the path is optimal but expands to higher nodes.

3) Blokus Game : BFS expanded significantly more nodes than DFS - 3119 to 128
DFS has a better running time in this case
Pacman Game : In this case, DFS expanded less nodes than BFS but not significantly - 146 to 269, it’s probably due to the branching factor that is low here.

In conclusion, the theoretical and empirical results are not always consistent with each other. While DFS might excel in certain scenarios with lower branching factors or shallower solutions, BFS could be more effective in cases with higher branching factors or when the shallowest solution is desired.

4.1 Blokus Corner Heuristic
Expanded nodes: 2633, score: 17

5.1 A valid Heuristic

1) Our heuristic estimates the minimum number of moves required to reach all the unreached corners from any filled position on the board by leveraging the infinity norm distance (Chebyshev distance).
   To ensure the heuristic is both admissible and consistent, we initialize `min_moves` and `min_distance` with the perimeter value, an upper bound on the actual cost. This guarantees that the heuristic never overestimates the cost.
   The function begins by identifying the corners of the board and filtering out those that have already been reached.
   If all corners are reached, it returns 0, indicating no further moves are needed.
   For each unreached corner, the heuristic calculates the minimum infinity norm distance from any occupied cell to that corner, ensuring it captures the closest approach to each corner.

   *It's the heuristic of question 4.1.

2) Let's define some notation:

- **n** be the number of unreached corners.
- **c** be an unreached corner, where c is a tuple (row, col) representing the coordinates of the corner.
- **p** be a filled position on the board, where p is a tuple (row, col) representing the coordinates of the position.
- **d(c, p)** is the infinity norm distance (Chebyshev distance) between an unreached corner **c** and a filled position **p**
- **h(state)** is the heuristic function, which estimates the minimum number of moves required to reach the nearest unreached corner from the current state.
- **h*(state)** is the true minimum number of moves required to reach the nearest unreached corner from the current state.

Proof:

If **n = 0**, then **h(state) = 0**. In this case, all corners are already reached, and no further moves are required. Therefore, **h(state) = h*(state) = 0**, and the admissibility condition holds.

Otherwise, let's break this down step by step:

- The perimeter of the board is calculated as **2 * state.board_w + 2 * state.board_h,** which represents the total number of positions along the edges of the board. This serves as an upper bound on the initial cost.
- For each unreached corner **c**, the heuristic function finds the minimum infinity norm distance from any filled position **p** to that corner. This is done by iterating over all filled positions **p** and calculating **d(c, p)** using the infinity norm distance formula.
- The minimum distance found for each unreached corner c is stored in **min_distance**, and the overall minimum distance among all unreached corners is stored in **min_moves**.
- By taking the minimum value of **min_moves** and **min_distance** for each unreached corner, the heuristic function ensures that it is always considering the shortest path to reach any unreached corner.
- The infinity norm distance **d(c, p)** represents the minimum number of moves required to reach the unreached corner **c** from the filled position **p**, assuming no obstacles. It is a lower bound on the actual number of moves required.
- Since the heuristic function considers the minimum infinity norm distance from any filled position to each unreached corner, it ensures that:

$$\forall c,\ \text{min\_distance} \leq \min(d(c, p) + h^*(p))$$

where **h*(p)** is the actual minimum number of moves required to reach the nearest unreached corners from the filled position **p**.

- Therefore, **min_moves** is always less than or equal to the actual minimum number of moves required to reach the nearest unreached corners from the current state:

$$\text{min\_moves} \leq h^*(\text{state})$$

- Finally, **h(state) = min_moves**, so we have:

$$h(\text{state}) \leq h^*(\text{state})$$

which proves the admissibility of the heuristic function.

Overall, the heuristic function **h(state)** is admissible because it never overestimates the actual cost to reach the goal state. It achieves this by considering the minimum infinity norm distance from any filled position to each unreached corner, which provides a lower bound on the actual number of moves required. The perimeter of the board serves as an upper bound on the initial cost, ensuring that the heuristic function remains admissible throughout the computation.

5.2 A New Heuristic

1) Changing the infinity norm to an Euclidean distance will make it a non-admissible heuristic.

Modified non-admissible heuristic with tile multiplication:

```
def euclidean_distance(point1, point2):
    """
    Calculate the Euclidean distance between two points
represented as (row, col).
    """
    row_diff = point1[0] - point2[0]
    col_diff = point1[1] - point2[1]
    distance = math.sqrt(row_diff ** 2 + col_diff ** 2)
    return distance
```

```
def blokus_corners_heuristic(state, problem):
    corners = [(0, 0), (0, state.board_w - 1), (state.board_h - 1, 0), (state.board_h - 1, state.board_w - 1)]
    unreached_corners = [corner for corner in corners if state.state[corner[0]][corner[1]] == -1]

    if not unreached_corners:
        return 0

    perimeter = 2 * state.board_w + 2 * state.board_h
    min_moves = perimeter
    for corner in unreached_corners:
        min_distance = perimeter
        for row in range(state.board_h):
            for col in range(state.board_w):
                if state.state[row][col] != -1:
                    distance = euclidean_distance(corner, (row, col))
                    min_distance = min(min_distance, distance)
        min_moves = min(min_moves, min_distance)

    return min_moves
```

Explanation:

Let's consider an example to illustrate the effect of this modification:

Suppose we have a Blokus game board of size 5x5 with the following state:

5.2

2) Counter - Example:

```
[
    [0, -1, -1, -1, 0],
    [-1, 0, -1, 0, -1],
    [-1, -1, 0, -1, -1],
    [-1, 0, -1, 0, -1],
    [0, -1, -1, -1, -1],
]
```

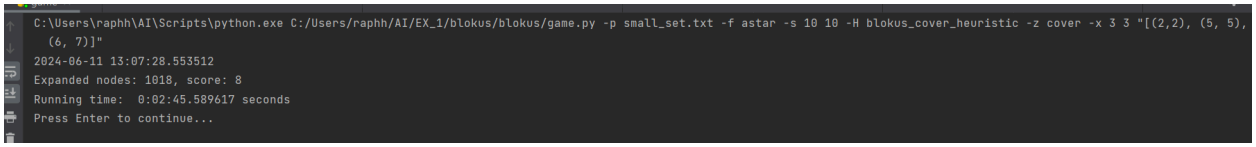
Tiles available : 0

Let's apply the modified heuristic that multiplies by the size of the smallest tile:

1. Identify the unreached corners:
    - The bottom-right corner (4, 4) are unreached.
2. Calculate the minimum Euclidean distance from any filled position to the unreached corners:
    - For the bottom-right corner (4, 4):
        - The minimum distance is sqrt(2), considering the occupied position (3,3).

However, in this counterexample, the actual minimum number of Euclidean distance required to reach all corners is sqrt(2) but the true cost is 1.

6.2


C:\Users\raphh\AI\Scripts\python.exe C:/Users/raphh/AI/EX_1/blokus/blokus/game.py -p small_set.txt -f astar -s 10 10 -H blokus_cover_heuristic -z cover -x 3 3 "[(2,2), (5, 5), (6, 7)]"
2024-06-11 13:07:28.553512
Expanded nodes: 1018, score: 8
Running time: 0:02:45.589617 seconds
Press Enter to continue...