Overview:

This practical asked us to develop and implement a java program for various searching algorithms. The implementation of this program required the use of different methods, loops, conditional statements, queues, and unit testing to create an output that is functional and formatted to the specification.

My solution achieved all the required aspects of the practical, and it passed both the checks provided by the documentation as well as the various ones I created to test its validity. My method contains several files within the implementation which all work together to create best first, a star and additional search methods. My program also contains various tests to test the different functionalities of the above-mentioned implementation using a sort of unit type testing.

Achievements:

The following goals were accomplished in my submission of this practical:

- Requirement A (Basic Functionality)
    - Best First (BestF) algorithm
    - A* (AStar) algorithm
- Requirement B
    - Evaluation of algorithms
- Requirement C
    - Alternative Search Algorithm
    - Opt Heuristic for BestF & A*
- Testing of Program

Design:

Search States

For modeling and implementing search states, the program contains a file, *Node.java*, to represent each state. This record contains information on the position, cost, heuristic, visited cell

count, and parent node for path details. This structure supports efficient path tracing through recursive parent links and enables the tracking of cost and heuristic for each state. This design simplifies path recovery and provides flexibility in defining heuristics and costs dynamically for both Best First and A* algorithms. Using a linked structure allows for efficient recursive path retrieval.

Frontier

To model the frontier a priority queue was chosen. The java built in library for this was used as it is quite extensive and provides all necessary functions. The queue uses the heuristic value stored in the nodes as a comparison value for priority. For BestF the queue is ordered by heuristic value alone, pushing lower values first. For AStar the queue incorporates both the heuristic value and the accumulated cost for optimal path discovery. Priority queue ensures optimal performance by maintaining an ordered structure which in turn enables the efficient retrieval of nodes with the lowest heuristic values.

Heuristic Functions

Implementation for the search algorithms all use a heuristic based on the remaining cells required to meet the coverage demand. Specifically, the heuristic calculates the difference between coverage and visited cells, scaled by a constant of three. In AStar, the heuristic combines the path cost and estimated remaining cost. This allows the heuristic to be a more optimal pathfinding strategy by considering both the path cost and the remaining estimated cost. This implementation is efficient in balancing progress toward the goal and maintain computational simplicity. Using a consistent heuristic structure across both algorithms simplifies comparisons while enabling optimal path selection.

Additional Functionality

To better help with the below evaluation, *Comparison.java*, was created to run the search algorithms against each other. The program tests BestF and AStar in multiple scenarios with various board sizes and coverage values. For each of these the cost, number of nodes searched and the time it took is stored. This allows evaluation between the different search algorithms for comparison. There is also an implementation for different heuristic considerations for BestFOpt

and AStarOpt. These functions prefer a horizontal move rather than a diagonal move. Lastly an alternative search algorithm, depth first search was implemented.

Evaluation:

The evaluation involves testing both BestF and AStar searches on sets of board sizes and coverage requirements, with time limits applied to assess completions time and path costs. The key metrics explored include time to completion, the number of nodes explored, and total path cost. In BestF the data correlated to a faster node exploration but often resulted in non-optimal paths. On boards with higher coverage amounts, BestF completed searches faster than AStar due to the less computation regarding the lack of cost consideration in the heuristic. This was at the expense of higher path costs. For AStar, the results consistently produced optimal paths due to a more rigorous heuristic. This is particularly apparent in larger board sizes and higher coverage amounts. The algorithms performance, while slower in nodes explored compared to BestF, consistently yielded lower path costs and optimal coverage. Of note for AStar, with extra-large board sizes and coverage amounts the program would often not procure any output given the 30 second CPU timeout. For smaller board sizes with a medium amount of coverage, both these algorithms perform similar with often little to no difference in output. This limitation became apparent with board sizes larger than 8x8 and coverage requirements near or above 80%. As the board sizes increased upwards of 10x10 the algorithms started failing at lower coverage requirements, going down to as far as 40%. For BestF implementing a pruning technique, such as memorization of visited states and dynamically adjusting exploration based on coverage density would greatly improve node exploration. For AStar, improvements could be made to the heuristic which consider distance to the target and previously covered areas. If given more time, I would consider adding a hybrid approach which initializes a heuristic driven path search but falls back on systematic node exploration in high coverage, large grid cases. The Opt searches ended up failing more than both the basic implementations. I believe this is due to the extra computation required. For the alternative search algorithm, depth first, the program requires significantly less storage for the nodes explored. This does come at a drop in cost to find the goal path, however.

Testing & Output:

This practical submission has been tested against Stacscheck as well as manual testing to ensure the output is valid.

References:

Java Documentation: Oracle's Java API documentation, particularly for Priority Queue and Duration.

Practical Documentation: Practical guidelines specifically for implementation of heuristic and program functionality.

Provided Starter Code: P1main.java

Baeldung CS: Article comparing Depth-First and Breadth-First Searches.
https://www.baeldung.com/cs/dfs-vs-bfs