High-Level Description of Software

The software has three main stages.

1) **The User Interface/Input side:** In this stage, the program obtains the grid, number of units and their starting locations as input from the user through the terminal. Please refer to the README.md for details on how-to-use.
2) **The Path Finding Algorithm:** The default path finding algorithm is **A***. However, BFS (which will give a similar result as Dijkstra in this case can also be used by passing "BFS" as an input argument)
3) **The Visualization Side:** Visualization of the grid happens automatically before and after the path finding algorithm is executed. For the latter scenario, it shows the shortest path if one is available.

High-Level Description of Algorithm

A-Star (A*):

Single Unit Case

The default algorithm used for path finding is the A* algorithm. It finds the shortest path to the goal by prioritizing its search based on the cost to reach current position (g-score) and some heuristic (h-score). For the current program, the heuristic was set to be the Manhattan distance from the current position to the goal. Specifically, the implementation is as follows

1. The algorithm begins from the initial/start node. The g-score of the start node is set to be 0.
2. From the current node, it then checks the adjacent neighbors if they are reachable and gives them a g-score. The g-score of each node is *1+current node's g-score*.
3. It then adds them to a priority queue (minimum heap) to decide which node to further search first. The position of each new node in the priority queue is determined by its f-score which is given by the sum of its g-score and h-score. The lower the f-score, the higher the priority since it implies, we are closer to our goal whilst also taking the shortest path in our search.
4. We then repeat 2 and 3 until our goal node is reached.
5. If during our search, we arrive at a previously visited node and can give it a lower g-score, we update the heap.
6. If our heap is empty, meaning there are no further nodes to explore, then there is no path to the goal.
7. If we arrive at the goal node, we can simply backtrack the parents to reach the start node and that gives us the path from the start to the goal.

Multiple Unit (Optional) Case:

In the scenario of N units, for each unit *i* the A* algorithm checks the paths of previous units 0,1…,i-1 for current time-step *t*. If the *path(n,t)* uses the same node as *path(i,t),* it deems that specific cell as unreachable for the given *t* and it is not added to the heap. Here *n* is the n^{th} previous unit.

Breadth-First-Search(BFS)/Dijkstra:

Although Dijkstra and BFS are different algorithms, for the current path finding problem their results would be the same. Unlike A*, neither uses the h-score in deciding the location of next node to search.

The Dijkstra does use a cost to jump between nodes. However, if the cost for jumping between nodes is the same, as it is in our case, then there is no advantage to Dijkstra over BFS which uses a simple queue. Although they will give the shortest path, but they will explore more before reaching the goal. To run BFS, the user can pass an argument *BFS* whilst running the code.

(My) High-Level Take on Path Planning

A*, Dijkstra and BFS are not the only path planning algorithms but are one of the complete ones. Specifically, if there are multiple paths to the goal, these algorithms will always find the shortest one. A* can be scaled to D* if the environment is dynamically changing.

One can also use probabilistically complete algorithms such as RRT and RRT* which are faster because they randomly sample instead of using each cell in the grid, but they lack guarantees of finding the shortest or smoothest path if sufficient samples are not drawn. They also struggle in large rooms with small exits, requiring more samples to bring their performance closer to the complete algorithms.

However, one challenge that both complete and probabilistically complete algorithms face is that they rely on sample-based planners. This implies that their accuracy is limited by the sample size which may not always be dynamically feasible. For example, A* might suggest that to go from A-B-C, the agent needs to make a 90 degree turn at B. Making a 90 degree turn about a point may not be safe or even possible for something like a car which has dynamic constraints that these algorithms do not account for.

I was unable to implement this due to the lack of time but do review Graphs of Convex Sets (https://groups.csail.mit.edu/robotics-center/public_papers/Marcucci21.pdf), this promises to be an efficient way of solving a path planning problem with smooth trajectories that handle dynamic constraints.