CS 440 MP1
Section Q4

Christian Howard
howard28@illinois.edu

Luke Pitstick
pitstck2@illinois.edu

Liuyi Shi
liuyis2@illinois.edu

Abstract

Within this report, we investigate creating agents that use different path planning algorithms and do an analysis of how they compare for a set of sample problems. **Insert summary of results**. We later apply A^* algorithms to guide an agent to solving the Sokoban puzzle. **Insert summary of results**.

Contents

1 Part 1 - Multi-Goal Search

Within this section, we investigate using search algorithms to tackle single-goal and multi-goal objectives. To handle this problem, the provided maze is processed into a graph, G_m, that can be used to figure out what actions a given agent can take and when they reach some goal node. To model the motion of some agent, we define the state, $\boldsymbol{x}$, as the following:

$$\boldsymbol{x} = (p, b_1, b_2, \cdots, b_n)$$

where p is an index representing the graph node the agent is on and b_k represents a Boolean variable that is 1 (true) when the agent has passed through the k^{th} goal point at some point prior and 0 (false) otherwise. Note that this state is always implicitly dependent on the maze graph, G_m. We know an agent has completed a maze when it reaches a state that satisfies the following:

$$b_1 \wedge b_2 \wedge \cdots \wedge b_n = 1$$

To model the step by step motion of an agent, the transition function is defined as the following:

Algorithm 1: Transition Function - Multi-goal

```
Data: State, $\boldsymbol{x}$
Data: Action, $a$
Data: Maze Graph, $G_m$
Data: Goal Node Set, $V_g = \{p_1, p_2, \cdots, p_n\}$
Result: New State, $\boldsymbol{x}_n$

// Set the new state with the
// current state's boolean configuration
for $k \in \{1, 2, \cdots, |V_g|\}$ do
    $\boldsymbol{x}_n(b_k) = \boldsymbol{x}(b_k)$
end

// Update position in maze given the action $a$
$\boldsymbol{x}_n(p) = \text{getNextPosition}(\boldsymbol{x}, a, G_m)$

// Check if new position is in goal node set
// If so, find index associated with goal node and
// set corresponding boolean to 1
if $\boldsymbol{x}_n(p) \in V_g$ then
    Get $m \ni \boldsymbol{x}_n(p) = p_m$
    Set $\boldsymbol{x}_n(b_m) = 1$
end

return $\boldsymbol{x}_n$
```

Given the state representation and transition function described above, we are now able to implement and test our search-based agents. That said, we are now able to dive into solving each challenge of Part 1.

1.1 Part 1.1 - Basic Pathfinding

In this first challenge for Part 1, the goal is to implement search algorithms for navigating through a maze environment to a single goal location. For this challenge, we have implemented the following search algorithms:

- Depth-First Search
- Breadth-First Search
- Greedy Best-First Search
- A* Search

For Greedy and A*, we have also set their Heuristic Function as the Manhattan Distance metric, defined as the following:

$$d_M(u, v) = |u - v|_1$$

where $u, v \in \mathbb{R}^2$ represent positions in the maze we want the distance between and $|\cdot|_1$ is the L_1 norm. Using this formula, the heuristic function will be:

$$h(n) = d_M(n, g)$$

where n is the 2-D position of the current agent's position and g is the 2-D position of the goal point. Given the above search strategies and heuristic, Table 1 and Table 2 show the stats between each method for each of the provided mazes:

	Medium Maze	Big Maze	Open Maze
Depth-First Search	124	474	59
Breadth-First Search	68	148	45
Greedy Best-First Search	68	222	77
A* Search	68	148	45

Table 1: Solution Path Cost

As we can see in the table, Depth-First Search (DFS) tended to arrive at a solution to a given maze quickly but at the expense of a sub-optimal path cost. We can note that Breadth First Search (BFS) and A* both, as expected, produced optimal path costs but it is interesting to note that for the Big Maze and Open Maze, BFS actually reached the optimal solution faster. It seems

	Medium Maze	Big Maze	Open Maze
Depth-First Search	198	1029	319
Breadth-First Search	345	1259	523
Greedy Best-First Search	77	311	27761
A^* Search	202	1495	667

Table 2: Number of Nodes Expanded

feasible that the Manhattan distance heuristic would, at times, take A* in a path that would lead to dead ends, in turn requiring A* to expand more nodes than BFS would have to.

One other thing to note was the interesting characteristics of the Greedy Search. We can see that on one end, it reached an optimal result in a very low number of expanded nodes for the Medium Maze. On the other end, it reaches a sub-optimal path cost after expanding a huge number of states in the Open Maze. Greedy obviously lacks robustness and this can be suspected due to the fact it does not take into account how far it has come but only thinks about how far away it thinks it is. Given this one sided thinking, the agent can get too excited being "close" to the goal even though they may be winding through a lot of poor sub-paths.

1.2 Part 1.2 - Search with Multiple Dots

The goal of this part was to build heuristics for the A* algorithm that would allow an agent to tackle a set of multi-goal problems in a reasonable time frame. The heuristic being used for this section is one based on the Convex Hull.

Let us define p as the current location of the agent and define $V_u = \{\hat{g}_1, \hat{g}_2, \cdots, \hat{g}_m\}$ as the set of m unvisited goal points. Let us then compute the convex hull of $\{p\} \bigcup V_u$, defined as:

$$(V_{ch}, E_{ch}) = \text{ConvexHull}\left(\{p\} \bigcup V_u\right)$$

where V_{ch} are the vertices on the convex hull of the input set and E_{ch} is the edge set of 2-tuples such that $E_{ch} = \{(x, y) : x, y \text{ are ordered vertices of some edge}\}$. For convenience, let us remove the largest edge touching point p, if any such edges exist, within the edge set E_{ch}. We will define this resulting set as $\hat{E}_{ch}$. With these results, the heuristic we choose is then defined as:

$$h_{ch}(p) = \sum_{(x,y) \in \hat{E}_{ch}} \beta d_E(x, y) + (1 - \beta)\, d_M(x, y) + |V_u|$$

1.3 Part 1 Extra Credit - Suboptimal Search

2 Part 2 - Sokoban

Within this section, the goal is to develop an agent that is capable of solving the Sokoban puzzle problem for a set of input puzzles. To handle this problem, the state representation has been defined as the following:

$$\boldsymbol{x} = (p, b_1, b_2, \cdots, b_n)$$

where in this case p is the position of the agent, represented as a graph node index, and b_k represents the position of the k^{th} box's position, represented as a graph node index, within the puzzle environment. Note that, as before, the input puzzle is processed to construct a graph representation which is in turn an implicit part of the state and environment representation for this problem. To know when the agent has found a Goal State, the following condition must be met:

$$(b_1 \in V_g) \wedge (b_2 \in V_g) \wedge \cdots \wedge (b_n \in V_g) = 1$$

where $V_g = \{g_1, g_2, \cdots, g_n\}$ is the set of goal points within the puzzle we can place a box. The next step in formulating this problem is describing the Transition Model for the agent and what actions an agent can take based on its state. Algorithm 2 describes the Transition Model for Sokoban given our state representation, while Algorithm 3 refers to how we get the feasible action set for an agent given its current state. With these pieces described, we can readily approach solving Sokoban using the search methods discussed earlier.

Algorithm 2: Sokoban Transition Model Algorithm

```
Data: State, $\boldsymbol{x}$
Data: Action, $a$, assumed to be valid
Data: Maze Graph, $G_m$
Result: New State, $\boldsymbol{x}_n$

// Set the new state with the
// current state's box configuration
for $k \in \{1, 2, \cdots, |V_g|\}$ do
    $\boldsymbol{x}_n(b_k) = \boldsymbol{x}(b_k)$
end

// Update position in maze given the action $a$
$\boldsymbol{x}_n(p) = \text{getNextPosition}\,(\boldsymbol{x}, a, G_m)$

// Check if new position is in equivalent to
// the position of one of the boxes.  If so update
// the location of the box using the same action
if $\boldsymbol{x}_n(p) \in \{\boldsymbol{x}_n(b_1), \boldsymbol{x}_n(b_2), \cdots, \boldsymbol{x}_n(b_n)\}$ then
    Get $i \ni \boldsymbol{x}_n(p) = \boldsymbol{x}_n(b_i)$
    Set $\boldsymbol{x}_n(b_i) = \text{getNextPosition}\,(\boldsymbol{x}_n(b_i), a, G_m)$
end

return $\boldsymbol{x}_n$
```

Algorithm 3: Sokoban Action Set Retrieval Algorithm

```
Data: State, $\boldsymbol{x}$
Data: Maze Graph, $G_m$
Data: Goal Node Set, $V_g = \{g_1, g_2, \cdots, g_n\}$
Result: Action Set, $A$

// Get set of box positions
$B_p = \text{getBoxPositions}\,(\boldsymbol{x})$

// Initialize Action Set as the one based on the Connectivity
$A = \text{getActionsFromConnectivity}\,(\boldsymbol{x}(p), G_m)$

// Loop through possible actions and check they are valid
// given the rules of Sokoban
for $a \in A$ do
    // Get the next node in the graph the
    // agent would be going to
    $p_n = \text{getNextPosition}\,(\boldsymbol{x}, a, G_m)$

    // Check if $p_n$ is the position of a box
    if $p_n \in B_p$ then
        // Update position of matching box using action $a$
        // and check if it is a valid position
        $u = \text{getNextPosition}\,(p_n, a, G_m)$

        // If new box position interferes with another box
        // or the wall, remove $a$ from the action set $A$
        if $u \in B_p$ or $isWall(u, G_m)$ then
            $A \leftarrow A \setminus \{a\}$
        end
    end
end

// Return the desired Action Set
return $A$
```