## Defining the BLOXORZ problem

### Design Components

The Bloxorz game is simulated in the application via the following components:

Bloxorz World Map

World map is defined by a static m \* n square matrix containing tiles (represented by ‘1’) and holes or no-tile spaces (represented by ‘0’). The holes or no-tile spaces are needed to represent the irregularly shaped world map into a square matrix.

The world map also contains a 1\*1 block represented by value ‘9’ called the ‘target’ or the ‘goal’ block.

The world map is static in the nature that the map itself does not change throughout the course of the game.

Brick

A brick is a 1\*1\*2 sized 3-d structure, that occupies either 1 or 2 blocks depending upon the orientation. The orientation can be standing (occupying 1 block on the 2-dimensional world map) or horizontally/vertically lying (occupying 2 blocks on the world map)

Brick component in the app holds a Position object, and offers methods for moving the brick or identifying the number and positions of the occupied blocks.

Position

A position object holds the brick’s x, y coordinates as well as its current orientation.

The position object offers low level methods for comparing current position with the target block’s position.

TreeNode

TreeNode is the unit used to represent a node in the graph. Each node contains a brick object (which composites a Position object) and some properties/attributes to link the nodes together in order to create a graph.

The nodes can be connected from parent to child node via 4 directions, namely:

left, right, up and down.

For easily navigating through the graph/tree, we also use additional properties / links like:

parent that connects a child node back to its parent, and

dir\_from\_parent to suggest what direction was picked from parent to reach a child node.

The TreeNode also contains properties specific to A\* algorithm. The properties are named:

cost - is the actual cost to reach up to the node from the head of the tree.

f\_score - is the sum of g-cost and h-cost, used for computing the estimated cost.

### States and Tree Representation

The Bloxorz game can be though off as a background layer composed of the world map, with a series of movements of Brick. Each of the nodes in the graph thus only needs to represent the brick position with respect to the map. The directional movements between the nodes thus represent the edge between the nodes.

The application, thus uses a head TreeNode, consisting of a Brick object initialised at position 2,2 (or 1,1 in a 0-based world). Each of feasible movements form a new node connected to the head or it’s child nodes, thereby making a graph ( or a tree since most of the search algorithms maintain a visited node list to restrict movements terminating on an already visited node, thereby preventing any loops.)

The world map combined with the brick position and orientation can be thought of as one state, that can transition to another state via brick movement in one of the four possible directions.

## 2. BFS and DFS implementation

BFS Search

The BFS being an uninformed search traverses through each and every state in a tree order traversal. For this, the algorithm needs to maintain a queue (nodes\_queue), using which it can expand nodes one level at a time.

BFS algorithm also maintains a visited nodes/positions list to prevent the brick from moving back to its earlier position.

|  |
| --- |
|  |

Fig. 1 - Initial moves of the BFS search tree. (order of directions - LRUD)

DFS Search

Similar to BFS, The DFS search tree is also implemented in the application. The DFS algorithm maintains a stack (internally managed via recursive function calls). It also maintains a list of visited nodes to prevent loops.

Unlike BFS and A\*, DFS is highly sensitive to the order of search directions.

A detailed analysis of the same is presented later in Section 6.

|  |
| --- |
| Right  Right  Down  Left  Left  Left  Left  Left  Right  invalid  visited  visited  invalid |

Fig. 2 - Initial moves of DFS Search tree (order of directions - LRUD)

A\* Search

The A\* search being an informed search, is passed the target position. It calculates the heuristic cost function as a distance from each node to the target. The application supports Euclidean and Manhattan distance for heuristic cost calculation, defaulting to Euclidean if no arguments specified.

The search algorithm maintains a list of expanded nodes as a min-heap priority queue. The queue is sorted by the actual cost to reach the given nodes.

At each step, as it discovers new states, it calculates the total cost f(x) = g(x) + h(x)

where g(x) is the cost to reach the new state: g(x) = cost(current\_node) + 1

and h(x) is the heuristic cost calculated by Euclidean or Manhattan distance.

The state with minimum f(x) cost is popped from the priority queue, and

|  |
| --- |
|  |

Fig. 3 - Initial moves from A\* search.

### Application output

DFS / BFS Search

When running the application with search method as dfs or bfs ( -s dfs | -s bfs )

The output shown on the terminal looks like the following:

Step: 0, Depth: 0, hash(Node): 273905053, hash(Parent): 273627595, Parent->None

⬜⬜⬜⬛⬛⬛⬛⬛⬛⬛

⬜🟧⬜⬜⬜⬜⬛⬛⬛⬛

⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜

⬛⬜⬜⬜⬜⬜⬜⬜⬜⬜

⬛⬛⬛⬛⬛⬜⬜❎⬜⬜

⬛⬛⬛⬛⬛⬛⬜⬜⬜⬛

Step: 1, Depth: 1, hash(Node): 273905089, hash(Parent): 273905053, Parent->right

⬜⬜⬜⬛⬛⬛⬛⬛⬛⬛

⬜⬜🟧🟧⬜⬜⬛⬛⬛⬛

⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜

⬛⬜⬜⬜⬜⬜⬜⬜⬜⬜

⬛⬛⬛⬛⬛⬜⬜❎⬜⬜

⬛⬛⬛⬛⬛⬛⬜⬜⬜⬛

Step: 2, Depth: 2, hash(Node): 273905110, hash(Parent): 273905089, Parent->right

⬜⬜⬜⬛⬛⬛⬛⬛⬛⬛

⬜⬜⬜⬜🟧⬜⬛⬛⬛⬛

⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜

⬛⬜⬜⬜⬜⬜⬜⬜⬜⬜

⬛⬛⬛⬛⬛⬜⬜❎⬜⬜

⬛⬛⬛⬛⬛⬛⬜⬜⬜⬛

At each step, the application shows some information about the current state and its transition from previous / parent state.

Below is an explanation for the same:

**Step** - is the total number of steps or valid states encountered by the application so far.

**Depth** - Distance of the node from root node, root node is at depth 0.

**hash(Node)** - hash value of the current node. This can be used to link the node with its child node(s).

**hash(Parent)** - hash value of the current node’s parent node. This can be used to link the current node to its parent node.

**Parent->{left|right|up|down}** - The direction taken from the parent node to reach the current node.

When running the application with A\* algorithm, it shows some additional fields (like, Cost, Distance)

Step: 0, Depth: 0, Cost: 0, hash(Node): 285123997, Distance (current): 6.71

⬜⬜⬜⬛⬛⬛⬛⬛⬛⬛

⬜🟧⬜⬜⬜⬜⬛⬛⬛⬛

⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜

⬛⬜⬜⬜⬜⬜⬜⬜⬜⬜

⬛⬛⬛⬛⬛⬜⬜❎⬜⬜

⬛⬛⬛⬛⬛⬛⬜⬜⬜⬛

Step: 1, Depth: 1, Cost: 1, hash(Node): 285123991, hash(Parent): 285123997, Distance (parent -> current): 6.71 -> 5.00, Parent->right

⬜⬜⬜⬛⬛⬛⬛⬛⬛⬛

⬜⬜🟧🟧⬜⬜⬛⬛⬛⬛

⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜

⬛⬜⬜⬜⬜⬜⬜⬜⬜⬜

⬛⬛⬛⬛⬛⬜⬜❎⬜⬜

⬛⬛⬛⬛⬛⬛⬜⬜⬜⬛

Step: 2, Depth: 2, Cost: 2, hash(Node): 285124060, hash(Parent): 285123991, Distance (parent -> current): 5.00 -> 4.24, Parent->right

⬜⬜⬜⬛⬛⬛⬛⬛⬛⬛

⬜⬜⬜⬜🟧⬜⬛⬛⬛⬛

⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜

⬛⬜⬜⬜⬜⬜⬜⬜⬜⬜

⬛⬛⬛⬛⬛⬜⬜❎⬜⬜

⬛⬛⬛⬛⬛⬛⬜⬜⬜⬛

**Cost** - The total/actual cost to reach the current node.

**Distance (parent -> current)** - Euclidean / Manhattan distance from parent node to target, and current node to target.

### Analysis

Running the algorithms through various permutations of the search directions

|  | BFS | DFS | A\* (euclidean) | A\* (manhattan) |
| --- | --- | --- | --- | --- |
| LURD | 56, 7 | 75, 65 | 13, 7 | 11, 7 |
| LUDR | 65, 7 | 75, 58 | 14, 7 | 15, 7 |
| LRUD | 56, 7 | 70, 58 | 13, 7 | 11, 7 |
| LRDU | 56, 7 | 30, 28 | 13, 7 | 11, 7 |
| LDUR | 65, 7 | 68, 62 | 15, 7 | 15, 7 |
| LRDU | 64, 7 | 42, 37 | 13, 7 | 15, 7 |
| ULRD | 56, 7 | 64, 58 | 13, 7 | 11, 7 |
| ULDR | 65, 7 | 52, 45 | 14, 7 | 15, 7 |
| URLD | 56, 7 | 45, 43 | 13, 7 | 12, 7 |
| URDL | 56, 7 | 80, 48 | 15, 7 | 13, 7 |
| UDLR | 65, 7 | 64, 56 | 15, 7 | 15, 7 |
| UDRL | 65, 7 | 78, 59 | 14, 7 | 15, 7 |
| RLUD | 56, 7 | 65, 56 | 13, 7 | 12, 7 |
| RLDU | 56, 7 | 76, 54 | 13, 7 | 12, 7 |
| RULD | 56, 7 | 73, 60 | 13, 7 | 12, 7 |
| RUDL | 56, 7 | 81, 44 | 15, 7 | 13, 7 |
| RDLU | 56, 7 | 44, 37 | 15, 7 | 13, 7 |
| RDUL | 56, 7 | 46, 38 | 15, 7 | 13, 7 |
| DLUR | 65, 7 | 43, 42 | 14, 7 | 12, 7 |
| DLRU | 64, 7 | 20, 19 | 14, 7 | 16, 7 |
| DULR | 65, 7 | 44, 43 | 14, 7 | 12, 7 |
| DURL | 65, 7 | 13, 13 | 14, 7 | 16, 7 |
| DRLU | 64, 7 | 25, 24 | 14, 7 | 14, 7 |
| DRUL | 64, 7 | 25, 24 | 14, 7 | 14, 7 |

Table. 1 - Results of BFS, DFS and A\* (with Euclidean and Manhattan distance based cost heuristics). Each cell entry represents (Total Steps, Depth of node at target state).

|  |  |
| --- | --- |
|  | Minimum steps |
|  | Maximum steps |

* All the algorithms are able to reach the goal state.
* A\* with both Euclidean and Manhattan distance cost heuristics yields optimal path with depth=7.
* A\* with Manhattan distance based cost heuristics results in the lowest number of steps (11).
* DFS is highly sensitive to order of search directions. The number of steps taken vary between 13 (order=DURL) to 81 (order=RUDL)
* BFS is less sensitive to the order of search directions and takes between 56 to 65 steps to reach the target state.