1. Defining the BLOXORZ Problem 2

Design Components 2

State and Tree Representation 2

2. Implementation of Search Algorithms 3

BFS Search 3

DFS Search 4

A* Search 5

3. Application Output 6

4. Output Examples 8

Verbose output: 8

Cost Method 12

Order of Search Directions 12

Search Method 12

Display Style 13

5. Analysis 14

Conclusions 15

1. Defining the BLOXORZ Problem

Design Components

The Bloxorz game is simulated in the application using 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 tree from the state space 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 to connect 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, like:
f_cost - is the sum of g-cost and h-cost, used for computing the estimated cost.

State and Tree Representation

The Bloxorz game can be thought 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 uses a head TreeNode, consisting of a Brick object initialised at position 2,2 (1,1 in a 0-based world). Each of the feasible movements form a new node connect the node to its child nodes, thereby making a tree (as 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. Implementation of Search Algorithms

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.

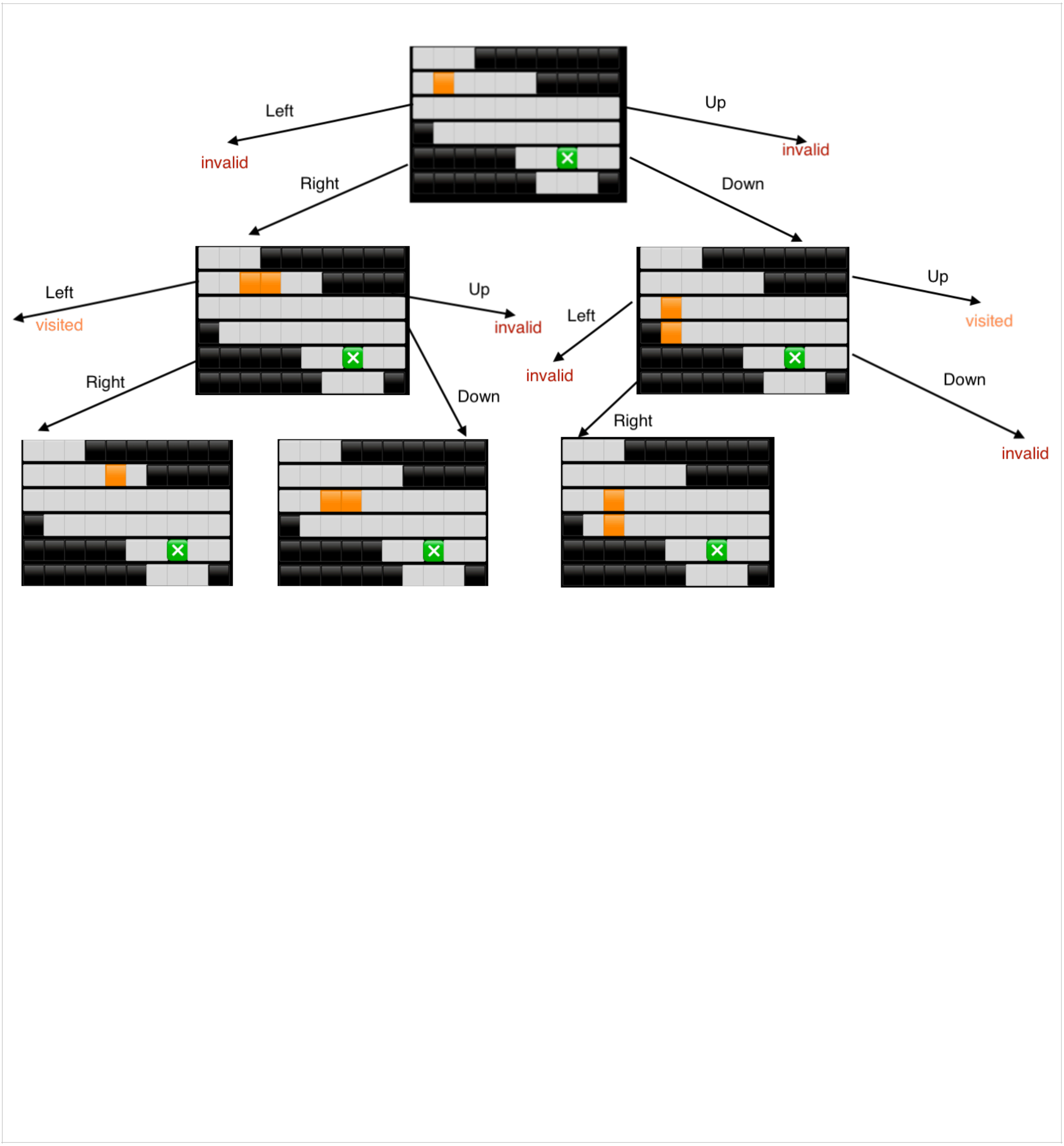


Fig. 1 - Initial moves of the BFS search tree. (order of directions - LRUD)

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. (see Section 5. Analysis for the results.)

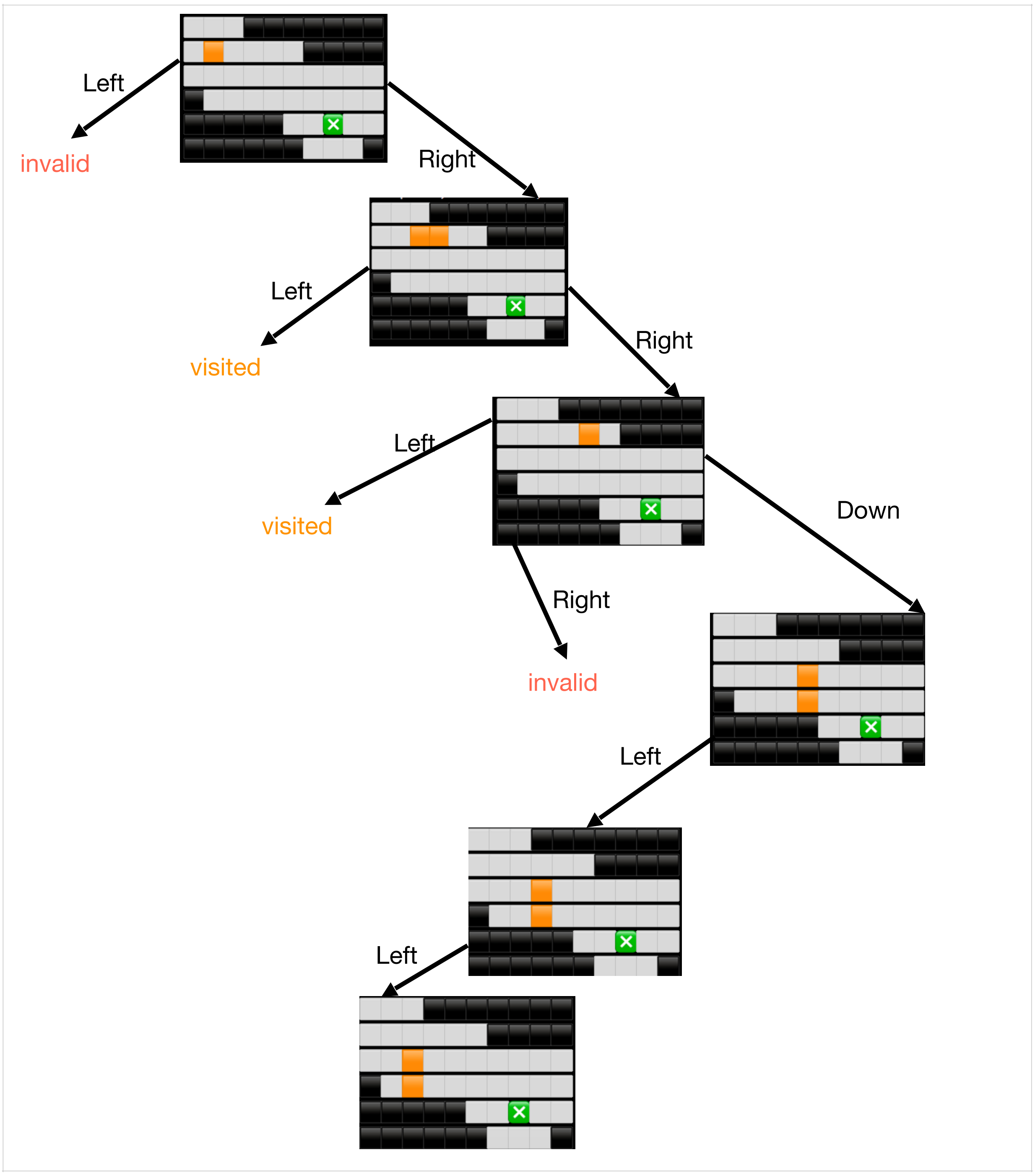


Fig. 2 - Initial moves of DFS Search tree (order of directions - LRUD)

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 estimated cost f(x) = g(x) + h(x) where g(x) is the actual cost to reach the new state: g(x) = actual_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 used to generate the new state.

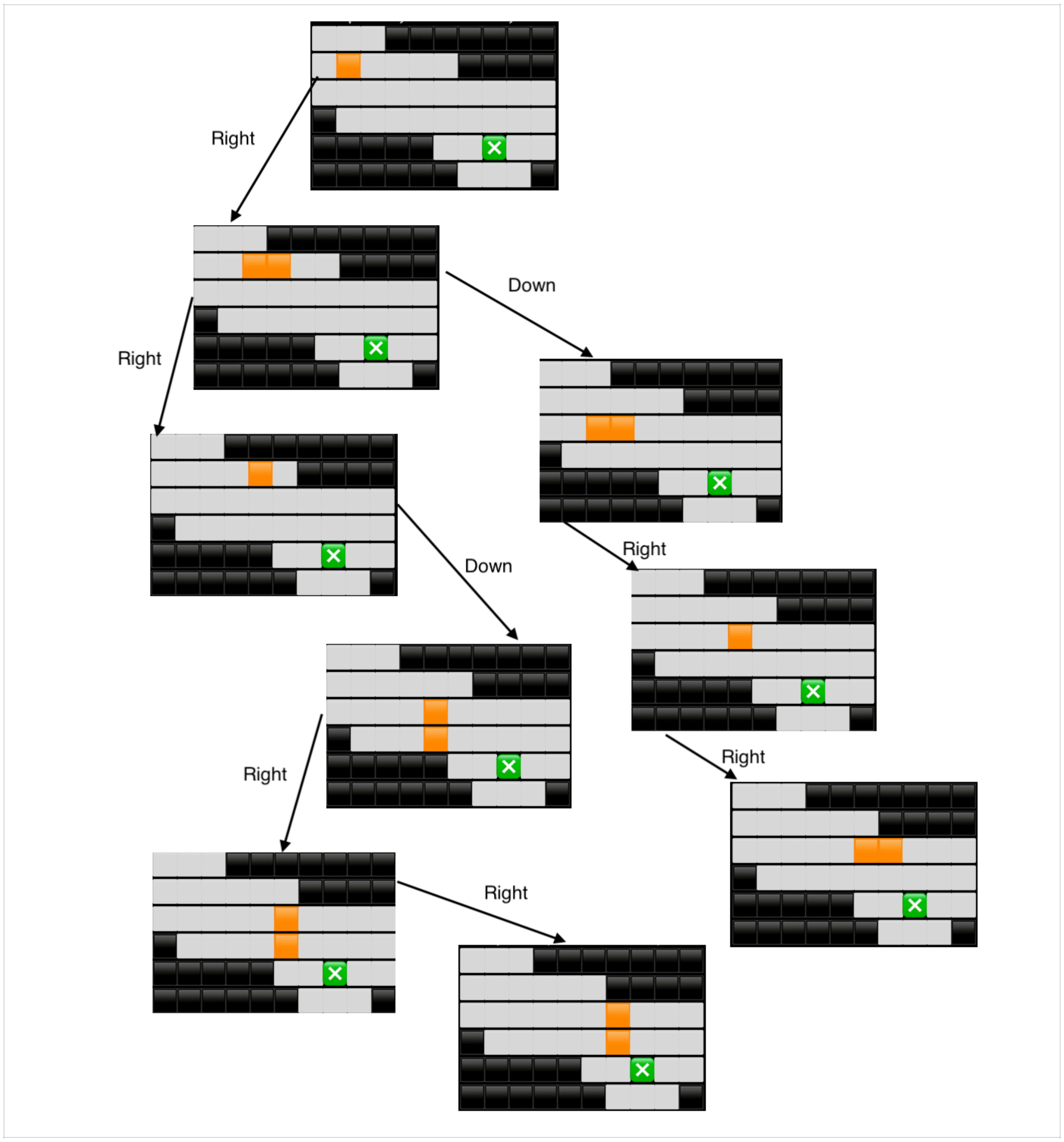


Fig. 3 - Initial moves from A* search.

3. Application Output

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): 269055907, hash(Parent): none, Parent->none , row: 2, col: 2]
⬜⬜⬜⬛⬛⬛⬛⬛⬛⬛
⬜🟧⬜⬜⬜⬜⬛⬛⬛⬛
⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜
⬛⬜⬜⬜⬜⬜⬜⬜⬜⬜
⬛⬛⬛⬛⬛⬜⬜❎⬜⬜
⬛⬛⬛⬛⬛⬛⬜⬜⬜⬛

Step: 1, Depth: 1 - [hash(Node): 269055970, hash(Parent): 269055907, Parent->right, row: 2, col: 3]
⬜⬜⬜⬛⬛⬛⬛⬛⬛⬛
⬜⬜🟧🟧⬜⬜⬛⬛⬛⬛
⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜
⬛⬜⬜⬜⬜⬜⬜⬜⬜⬜
⬛⬛⬛⬛⬛⬜⬜❎⬜⬜
⬛⬛⬛⬛⬛⬛⬜⬜⬜⬛

Step: 2, Depth: 2 - [hash(Node): 269055991, hash(Parent): 269055970, Parent->right, row: 2, col: 5]
⬜⬜⬜⬛⬛⬛⬛⬛⬛⬛
⬜⬜⬜⬜🟧⬜⬛⬛⬛⬛
⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜
⬛⬜⬜⬜⬜⬜⬜⬜⬜⬜
⬛⬛⬛⬛⬛⬜⬜❎⬜⬜
⬛⬛⬛⬛⬛⬛⬜⬜⬜⬛
```

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 various terms:

Step - is the total number of steps / moves taken by the algorithm so far.

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

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

hash(Parent) - hash value of the a node's parent or "none" if there is no 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. The value is "none" if a node does not have any parent node.

row/col - Row and column position of the brick, if in standing orientation.
For horizontal lying orientation, the col value represents the leftmost block position of the brick.
For vertical lying orientation, the row value represents the topmost bloc position of the brick.

The row/col values are 1-based. The application internally uses 0-based index, but for display purposes the values are converted to 1-based index.

With A* search method, the application prints some additional fields, namely Cost and f_cost.

Cost - The actual cost to reach the current node.

f_cost - The estimated cost calculated using g_cost (or actual cost to reach the next node) and h_cost (heuristic cost distance from the next node to the target node).

Additional debugging information can be obtained when running the application with -v (verbose) option. The verbose output provides insight into the algorithm's decisions based on validity or cost of the moves. (see Section 4.)

4. Output Examples

Verbose output:

With verbose flag enabled, the application shows the configuration in use, before the algorithm steps are performed.

```
$ python3 ./bloxorz.py  -v
cost-method: euclidean
order: LRUD
search: a-star
style: unicode
verbose: True
```

During the execution, it prints extra debug lines before each step / state.
The debug lines follow the syntax:
Action : Reason   -   Information about the node on which action was taken.

BFS SEARCH - VERBOSE OUTPUT

An example below shows the output of bfs search with verbose option on.
($ python3 ./bloxorz.py -s bfs -o LRUD -v)

```
Step: 53, Depth: 6, - [hash(Node): 282172404, hash(Parent): 282171597, Parent->right, row: 3, col: 6]
⬜⬜⬜⬛⬛⬛⬛⬛⬛⬛
⬜⬜⬜⬜⬜⬜⬛⬛⬛⬛
⬜⬜⬜⬜⬜🟧⬜⬜⬜⬜
⬛⬜⬜⬜⬜⬜⬜⬜⬜⬜
⬛⬛⬛⬛⬛⬜⬜❎⬜⬜
⬛⬛⬛⬛⬛⬛⬜⬜⬜⬛

rejected  : visited node  - [hash(Parent): 282172404, Parent->left ]
added     : new node    - [hash(Node): 282173175, hash(Parent): 282172404, Parent->right, row: 3, col: 7]
rejected  : invalid move  - [hash(Parent): 282172404, Parent->up   ]
added     : new node   - [hash(Node): 282173194, hash(Parent): 282172404, Parent->down , row: 4, col: 6]
removed   : frontier node - [hash(Node): 282152355, hash(Parent): 282171597, Parent->down , row: 4, col: 4]
Step: 54, Depth: 6, - [hash(Node): 282152355, hash(Parent): 282171597, Parent->down , row: 4, col: 4]
⬜⬜⬜⬛⬛⬛⬛⬛⬛⬛
⬜⬜⬜⬜⬜⬜⬛⬛⬛⬛
⬜⬜⬜⬜⬜⬜⬜⬜⬜⬜
⬛⬜⬜🟧🟧⬜⬜⬜⬜⬜
⬛⬛⬛⬛⬛⬜⬜❎⬜⬜
⬛⬛⬛⬛⬛⬛⬜⬜⬜⬛
```

Explanation:

From the given state (53), The 4 search directions left, right, up and down (in that order) were tried.

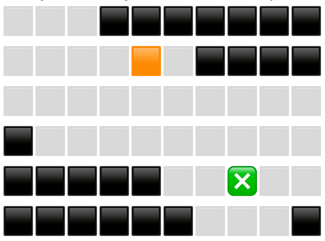
The move to left was rejected, as the node corresponding to that state is already visited.
The move to right is a valid move, so it was added to the queue for later expansion in BFS search.
The move to up direction is invalid, as it would cause the brick to fall off the world map.
The move down is valid, and it is added to the BFS queue.

Finally, a node is removed from the BFS queue which belongs to a previous state (note that frontier node's parent hash does not match the previous node's hash value)
and new state is generated from it.

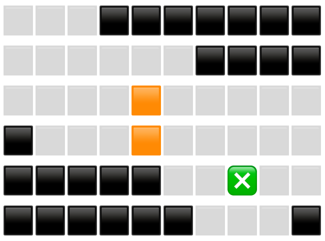
DFS SEARCH - VERBOSE OUTPUT

```
($ python3 ./bloxorz.py -s dfs -o LRUD -v )
```

```
Step: 2, Depth: 2 - [hash(Node): 277081597, hash(Parent): 277081579, Parent->right, row: 2, col: 5]
```



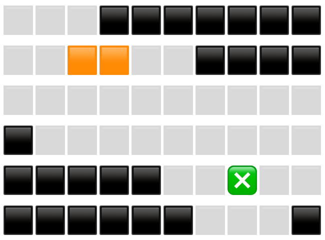
```
rejected  : visited node        - [hash(Parent): 277081597, Parent->left ]
rejected  : invalid move         - [hash(Parent): 277081597, Parent->right]
rejected  : invalid move         - [hash(Parent): 277081597, Parent->up   ]
to visit  : new node            - [hash(Node): 277081582, hash(Parent): 277081597, Parent->down , row: 3, col: 5]
Step: 3, Depth: 3 - [hash(Node): 277081582, hash(Parent): 277081597, Parent->down , row: 3, col: 5]
```



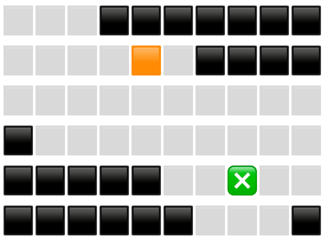
From the given state (step 2), the 4 directions LRUD (in that order) were tried.
The state corresponding to left move is already visited, hence rejected.
The states corresponding to right and up are invalid, as they take the brick off world map.
The move to down direction is valid, and will be used in the recursive call to dfs search method, leading to Step 3.
Also it can be confirmed from the hash values, that Step 2 node is the parent node of step 3.

A* SEARCH - VERBOSE OUTPUT

```
Step: 1, Depth: 1, Cost: 1 - [hash(Node): 284759019, hash(Parent): 284758980, Parent->right, row: 2, col: 3] [f_cost:
6.00]
```



```
rejected  : visited & costly      - [hash(Parent): 284759019, Parent->left] [Cost now: 2, earlier: 0]
added     : new | visited & cheap - [hash(Node): **284759037**, hash(Parent): 284759019, Parent->right, row: 2, col: 5] [f_-
cost: **6.24** = **2** + **4.24**]
rejected  : invalid move          - [hash(Parent): 284759019, Parent->up   ]
added     : new | visited & cheap - [hash(Node): 284795661, hash(Parent): 284759019, Parent->down , row: 3, col: 3]
[f_cost: 6.47 = 2 + 4.47]
removed   : frontier node         - [hash(Node): 284759037, hash(Parent): 284759019, Parent->right, row: 2, col: 5]
Step: 2, Depth: 2, Cost: 2 - [hash(Node): **284759037**, hash(Parent): 284759019, Parent->right, row: 2, col: 5] [f_cost:
6.24]
```



From Step 1, the following moves were tried:

- Move from State 1 to left is already visited at cost 0, visiting it again would make the cost 2, hence it is rejected.
- Move to right leads to a new state, hence it is added to the priority queue at f_cost = 6.24 (or if it were visited previously at a higher actual cost than current actual cost.)
- Move to up direction is invalid.
- Move to down is again a new state, and added to priority queue with f_cost = 6.47

- Off all the items in priority queue, the move [State 1 -> right] has the lowest f_cost, so it is removed from the queue, and yields the next state.

The f_cost in the above examples is composed of g_cost or the actual cost to reach the node (g_cost = 2), and the heuristic distance (euclidean distance) from the new state to the target state (h_cost = sqrt(3*3 + 3*3) = 4.24)

Cost Method

A* search can be performed with heuristic cost method based on Euclidean distance or Manhattan distance.

```
$ python3 ./bloxorz.py -c euclidean  -s a-star
Or
$ python3 ./bloxorz.py -c Manhattan  -s a-star
```

At each step, for both euclidean and Manhattan cost methods, output shows the f_cost:

Step: 1, Depth: 1, Cost: 1 - [hash(Node): 270217927, hash(Parent): 270217867, Parent->right, row: 2, col: 3] [**f_cost: 6.00**]

Verbose options can be added to show the g_cost and h_cost, that will print additional lines as:
added   : new | visited & cheap - [hash(Node): 270217927, hash(Parent): 270217867, Parent->right, row: 2, col: 3] [**f_cost: 6.00 = 1 + 5.00**]

Order of Search Directions

The default order for search directions is LRUD (left, right, up, down), alternate orders can be specified using the -o option.

```
$ python3 ./bloxorz.py -o DURL -v
cost-method: euclidean
order: DURL
search: a-star
style: unicode
verbose: True…
…
```

Search Method

Search method can be specified using -s option, default is A* (a-star).

```
$ python3 ./bloxorz.py -s dfs -v
cost-method: euclidean
order: LRUD
search: dfs
style: unicode
verbose: True
…
…
```

Display Style

Display style can be changed to **ascii** using -t option. The default style uses unicode characters which may not be suitable for some old terminals.

In the **ascii** style, the application uses the character ‘**1’** for tiles, ‘**0**’ for no-tile positions, ‘**X**’ for brick positions and ‘**+**’ for the target block.

```
$ python3 ./bloxorz.py -t ascii -v
cost-method: euclidean
order: LRUD
search: a-star
style: ascii
verbose: True

Step: 0, Depth: 0, Cost: 0 - [hash(Node): 282143877, hash(Parent): none, Parent->none , row: 2, col: 2]
1110000000
1X11110000
1111111111
0111111111
0000011+11
0000001110
```

5. **Analysis**

Running the algorithms through various permutations of the search directions

Order	**BFS (Steps, Depth)**	**DFS (Steps, Depth)**	**A* (euclidean) (Steps, Cost)**	**A* (manhattan) (Steps, Cost)**
LURD	56, 7	75, 65	12, 7	11, 7
LUDR	65, 7	75, 58	11, 7	12, 7
LRUD	56, 7	70, 58	12, 7	11, 7
LRDU	56, 7	30, 28	10, 7	11, 7
LDUR	65, 7	68, 62	11, 7	12, 7
LDRU	64, 7	42, 37	11, 7	12, 7
ULRD	56, 7	64, 58	12, 7	11, 7
ULDR	65, 7	52, 45	11, 7	12, 7
URLD	56, 7	45, 43	12, 7	11, 7
URDL	56, 7	80, 48	12, 7	11, 7
UDLR	65, 7	64, 56	11, 7	12, 7
UDRL	65, 7	78, 59	11, 7	12, 7
RLUD	56, 7	65, 56	12, 7	11, 7
RLDU	56, 7	76, 54	10, 7	11, 7
RULD	56, 7	73, 60	12, 7	11, 7
RUDL	56, 7	81, 44	12, 7	11, 7
RDLU	56, 7	44, 37	10, 7	11, 7
RDUL	56, 7	46, 38	10, 7	11, 7
DLUR	65, 7	43, 42	11, 7	12, 7
DLRU	64, 7	20, 19	11, 7	12, 7
DULR	65, 7	44, 43	11, 7	12, 7
DURL	65, 7	13, 13	11, 7	12, 7
DRLU	64, 7	25, 24	11, 7	12, 7
DRUL	64, 7	25, 24	11, 7	12, 7

Table. 1 - Results of BFS, DFS and A* with Euclidean and Manhattan distance based cost heuristics. For BFS/DFS cell entries represent [Total Steps, Depth of node when at target state], and for A* [Total Steps, Optimal Cost].

🟦	Minimum steps	🟨	Maximum steps

Conclusions

- All the algorithms (BFS, DFS, A*) are able to reach the goal state.
- BFS gives optimal path with depth=7.
- A* with both Euclidean and Manhattan distance cost heuristics yields optimal path with depth / cost = 7.
- 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 compared to DFS and takes between 56 to 65 steps to reach the target state.