Assignment submitted by Harsh Kumar Bhatia (2019AD04086), Alkesh Chaturvedi (2019AD04082) and Raza Abbas (2019AD04095)

1) **Formulate the Bloxorz problem as a search problem by depicting its states representation, initial state, actions, transition model, goal state.**

   In the given problem, it is mentioned that there is a single agent (path finding) which must reach its given goal state from its given initial state.

   This agent is an example of a goal-based agent a.k.a problem-solving agent which will use a search strategy (a.k.a search algorithm) to solve the given problem to reach its goal. Such problems are solved by formulating the goal (**Goal formulation**) followed by formulating the problem (**Problem formulation**).

   Goal formulation: State *In*(*Bloxorz[5,8,5,8]*[a]*)* is the goal state to be found by the agent.

   Problem formulation: Problem can be defined formally by five components:

   - **States:** Any arrangement (standing, lying horizontally or lying vertically) of the 1 x 1 x 2 size block anywhere on the given terrain while ensuring that its ends are always within the terrain boundary is a state. These are called *Legal states*.
   - **Initial state:** Any one state from the set of above obtained states can be designated as the initial state. In the given problem, state *In*(*Bloxorz[2,2,2,2])* is the initial state where 1 x 1 x 2 block is standing straight on the given terrain.
   - **Actions:** In this environment, each state has just 4 actions: *Left, Right, Up, or Down*. Different subsets of these actions or moves are possible depending on where the block or agent is. These are called *Legal moves* such that block's ends always remain within the terrain boundary.
   - **Transition model:** Given a (legal) state and (legal) action/move, this returns the resulting (legal) state; for example, if we apply *Right* move to the initial state (as given in the problem statement), the resulting state has the block lying horizontally on next 2 cells right of cell *Bloxorz[2,2,2,2]* (Initial state), that is, *(In(Bloxorz[2,2,2,2]), Right)* gives state *In(Bloxorz[2,3,2,4])*.
   - **Goal test:** This checks whether state has become *In*(*Bloxorz[5,8,5,8])*. This state is also called *Goal State*.

   Though it is not asked in the given problem, but we can have **"Path Cost"** as well as part of the problem formulation.

   - **Path Cost:** Each step costs 1, so the path cost is the number of steps in the path.

2) **Can BFS/DFS be used to solve this problem? If so, explain how it can be used by providing an algorithm/pseudocode.**

   Yes, both BFS and DFS can be used to solve the given problem. We would be needing *"Frontier aka Open List"* to store the nodes/states which are to be expanded next and we would need *"Explored Set aka Closed List"* to store the nodes which have already been expanded/explored.

   Please note that in case of BFS and DFS search strategies, both *Frontier* and *Explored Set* are going to be simple *list* data structures. In case of BFS, *Frontier* will be implemented as a *FIFO queue* while in case of DFS, *Frontier* will be implemented as a *LIFO stack*. In both the algorithms, we will get *Sequence of Actions* as the result of search strategy which will then be executed to reach the final *Goal State*.

BFS (Breadth First Search)

As mentioned earlier, in BFS, both *Frontier & Explored Set* will be implemented using a simple list data structure. In case of BFS specifically, *Frontier* will be operating as a *FIFO queue* so that nodes or states which are added first in the *Frontier* will be taken out first for expansion or exploration.

To remove the *repeated states* from search (generated due to *loopy paths*, a special case of more general concept of *redundant paths)*, *Explored Set* is used to keep track of visited states so that the states, which are already explored, are not explored again.

Having use of Explored set makes this BFS search a graph search versus a tree search.

In general, in BFS search strategy, *shallowest* unexpanded node is chosen for expansion.

Pseudocode for *Breadth First Search (BFS)*[a]

1. Formulate the problem by clearly defining the *problem.Initial_State* (given initial position of the 1 x 1 x 2 block on specified terrain or user specified initial position) *& problem.Goal_Test* (given position of the hole on specified terrain)
2. *node*[b] ← a node with *STATE = problem.Initial_State*
3. Initialize *node.SequenceOfActions* variable to *Null*.
4. Check if *problem.Goal_Test(node.STATE) is True, return SOLUTION(node)*[d] *& terminate*
5. *Frontier* ← a FIFO queue with *node* as the only element /* *node* enqueued in *Frontier* */
6. *Explored* ← an empty set
7. Loop do
    a. If *Frontier is Empty*, then return *failure*.
    b. *node* ← dequeue(*Frontier*) /* expands the shallowest node in *Frontier* */
    c. add *node.STATE* to *Explored*
    d. for each *action* in *problem.ACTIONS*(*node.STATE*) do /* only legal actions allowed */
        i. generate *child* node[c] /* child node represents new tile position of agent */
        ii. Check if *child.STATE* is not in *Explored* or *Frontier* then
            - ✓ Check if *problem.Goal_Test*(*child.STATE*) then return *SOLUTION(child)*[d] *& terminate*
            - ✓ Else *Frontier* ← enqueue(*child, Frontier*)

DFS (Depth First Search)

In DFS (unlike BFS), *Frontier* will be operating as a *stack (LIFO)* so that nodes or states which are added last in the *Frontier* will be taken out first for expansion or exploration.

In DFS (like BFS), to remove the *repeated states* from search (generated due to *loopy paths*, a special case of more general concept of *redundant paths)*, *Explored Set* is used to keep track of visited states so that the states, which are already explored, are not explored again.

Having use of Explored set makes this DFS search a graph search versus a tree search.

Also, explored nodes with no descendants in the *Frontier* are removed from memory thus making DFS memory efficient when compared with BFS.

In general, in DFS search strategy, it is always that the *deepest* node in current *Frontier* is chosen for expansion.

Pseudocode for *Depth First Search (DFS)*[a]

1. Formulate the problem by clearly defining the *problem.Initial_State* (given initial position of the 1 x 1 x 2 block on specified terrain or user specified initial position) *& problem.Goal_Test* (given position of the hole on specified terrain)
2. *node*[b] ← a node with *STATE = problem.Initial_State*
3. Initialize *node*.*SequenceOfActions* variable to *Null*.
4. *Frontier* ← a LIFO stack with *node* as the only element /* *node* pushed in *Frontier* */
5. *Explored* ← an empty set
6. Loop do
    a. If *Frontier is Empty*, then return *failure*.
    b. *node* ← pop(*Frontier*) /* expands the topmost node in *Frontier* */
    c. Check if *problem.Goal_Test(node.STATE) is True, return SOLUTION(node)*[d] *& terminate*
    d. add *node.STATE* to *Explored*
    e. for each *action* in *problem.ACTIONS*(*node.STATE*) do /* only legal actions allowed */
        i. generate *child* node[c] /* child node represents new tile position of agent */
        ii. Check if *child.STATE* is not in *Explored* or *Frontier* then
            - ✓ *Frontier* ← push(*child, Frontier*)

Note: [a] *Here 1 x 1 x 2 block (or agent) can lay horizontally or vertically on the tiles (assuming each 1 x 1 tile is alike a matrix element where every matrix element is represented by it corresponding row and column position), hence every legal state (in the state space of this problem) can be defined by specifying a legal position for agent using two x-y co-ordinates, that is, In(Bloxorz[c1,r1,c2,r2]) where r1 = row 1, c1= column 1, r2 = row 2 & c2 = column 2. If agent is standing straight, with its longer edge being perpendicular to the terrain, we will have r1=r2 & c1=c2. Note that c1 or c2 are corresponding to x co-ordinate of agent on given board and r1 or r2 correspond to y-co-ordinate of agent on given board.*

[b] *For each node, we have a structure that contains following components:*

node.STATE: the state in the state space to which the node corresponds;
node.PARENT: the node in the search tree that generated this node;
node.SequenceOfActions: the action that was applied to the parent to generate the node. Its value is equal to its parent's node SequenceOfActions plus the move or action it is generated from.

[c] *Child node or node gets generated by applying only legal moves to the parent node. Child node will have its own child.SequenceOfActions which is equal to its parent's node SequenceOfActions plus the move or action it is generated from.*

[d] *SequenceOfActions variable of node or child discovered as goal state or goal node gives the answer.*

3) Can A* search be used to solve this problem? If so, explain how it can be used by providing an algorithm/pseudocode

Yes, we can use A* search algorithm to solve the given problem, however, A* being an informed search, we need an appropriate (both consistent & admissible) heuristic for it to work. Once we have the heuristic function *h(n)* available for every node or state, we can go on to calculate an evaluation function *f(n)* (*f(n)*[7] = *g(n)*[5] + *h(n)*[6]) for every node and the node with least value for *f(n)* will be picked for expansion. This is how A* is different from blind search (BFS or DFS) and *prune* a major chuck of search state.

Unlike BFS or DFS (where we used a simple *list* data structure for *Frontier* implementation), in case of A* search algorithm, we will use a *Priority queue* data structure for *Frontier*. *Priority queue* will be used to order the nodes with least *f(n)* value on top (*highest priority*) of the *Frontier* to pick them first for expansion.

Regarding choosing an appropriate heuristic, many calculations can be used. h_{SLD} (Straight Line Distance) using Euclidean distance is one of the options. Another option is to use absolute distances between current node and goal node.

Formula for heuristic function *h(n)* using absolute distances can be as below:

$$x(n) = abs(C_{current\ node} - C_{goal\ node})$$

$$y(n) = abs(R_{current\ node} - R_{goal\ node})$$

where $R_{current\ node}$ is the row number (corresponding to y co-ordinate of the agent on the given terrain) and $C_{current\ node}$ is the column number (corresponding to x co-ordinate of the agent on the given terrain) of the current node. Similarly, $R_{goal\ node}$ and $C_{goal\ node}$ are defined. As the given block or agent occupies two tiles when laid horizontally or vertically on terrain, its position is represented by two x-y co-ordinates. In case agent is standing straight, both x-y co-ordinate values are same. First, the *x(n) & y(n)* values for each tile (occupied by block/agent) or each x-y co-ordinate pair is calculated as shown below:

$$x1(n) = abs(C_{current\ node} - C_{goal\ node})$$

$$y1(n) = abs(R_{current\ node} - R_{goal\ node})$$

$$x2(n) = abs(C_{current\ node} - C_{goal\ node})$$

$$y2(n) = abs(R_{current\ node} - R_{goal\ node})$$

Then do the summation of *x(n)* and *y(n)* values for each of the two tiles and pick the maximum value out of two for the final heuristic calculation using below formula.

$$h(n) = max(x1(n)+y1(n),\ x2(n)+y2(n))$$

For example, the heuristic of the state *In(Bloxorz[2,3,2,4])* when goal state is *In*(*Bloxorz[5,8,5,8])* is *h(n) = 8.* Calculation of the same is shown below:

$$x1(n) = abs(2-5) = 3$$

$$y1(n) = abs(3-8) = 5$$

$$x2(n) = abs(2-5) = 3$$

$$y2(n) = abs(4-8) = 4$$

$$h(n) = max(3+5,3+4) = max(8,7) = 8$$

Pseudocode for *A* search*[1]

1. Formulate the problem by clearly defining the *problem.Initial_State* (given initial position of the 1 x 1 x 2 block on specified terrain or user specified initial position) *& problem.Goal_Test* (given position of the hole on specified terrain)
2. *node*[2] ← a node with *STATE = problem.Initial_State, Path_Cost*[7] *= 0*
3. Initialize *node.SequenceOfActions* variable to *Null*.
4. *Frontier* ← a priority queue ordered by *Path_Cost*[7], with *node* as the only element
5. *Explored* ← an empty set
6. Loop do
    a. If *Frontier is Empty*, then return *failure*.
    b. *node* ← pop(*Frontier*) /* choose the *lowest-cost* node in *Frontier* for expansion */
    c. Check if *problem.Goal_Test(node.STATE) is True, return SOLUTION(node)*[4] *& terminate*
    d. add *node.STATE* to *Explored*
    e. for each *action* in *problem.ACTIONS*(*node.STATE*) do /* only legal actions allowed */
        i. generate *child* node[3] /* child node represents new tile position of agent */
        ii. Check if *child.STATE* is not in *Explored* or *Frontier* then
            ✓ *Frontier* ← insert(*child, Frontier*)
        iii. Else if *child.STATE* is *Frontier* with higher *Path_Cost*[7] then
            ✓ replace that *Frontier* node with *Child* node

As a result of this search algorithm, we will get *Sequence of Actions* which will then be executed to reach the final *Goal State*.

***Note*:** [1] *Here 1 x 1 x 2 block (or agent) can lay horizontally or vertically on the tiles (assuming each 1 x 1 tile is alike a matrix element where every matrix element is represented by it corresponding row and column position), hence every legal state (in the state space of this problem) can be defined by specifying a legal position for agent using two x-y co-ordinates, that is, In(Bloxorz[c1,r1,c2,r2]) where r1 = row 1, c1= column 1, r2 = row 2 & c2 = column 2. If agent is standing straight, with its longer edge being perpendicular to the terrain, we will have r1=r2 & c1=c2. Note that c1 or c2 are corresponding to x co-ordinate of agent on given board and r1 or r2 correspond to y-co-ordinate of agent on given board.*

[2] *For each node, we have a structure that contains following components:*

node.STATE: the state in the state space to which the node corresponds;
node.PARENT: the node in the search tree that generated this node;
node.Path_Cost: the cost denoted by f(n) is the estimated path cost from initial state to goal node via node n. f(n) = g(n) + h(n).
node.SequenceOfActions: the action that was applied to the parent to generate the node. Its value is equal to its parent's node SequenceOfActions plus the move or action it is generated from.

[3] *Child node or node gets generated by applying only legal moves to the parent node. Child node will have its own child.SequenceOfActions which is equal to its parent's node SequenceOfActions plus the move or action it is generated from.*

[4] *SequenceOfActions variable of node or child discovered as goal state or goal node gives the answer.*

[5] *g(n) is the actual path cost from the initial state to the state defined in the current node. Also, each step costs 1, so the path cost is the number of steps in the path.*

[6] *h(n) is the estimated path cost from the state in current node to the state in goal node.*

[7] *f(n) or Path_Cost is the estimated path cost from initial state to goal node via node n. f(n) = g(n) + h(n).*

4) Implement an agent to solve level-1 of Bloxorz game using python.

Please find associated python file for the code of this problem. Screenshots of the program output are pasted below.

```
Please Note: Board starts (Topmost Left tile) at x-coordinate = 0 & y-coordinate = 0
Any references to x and y coordinates hereon should be read accordingly
************************************************
Input board is displayed below such that 1 = valid tile, 0 = void location & 9 = goal location
************************************************
[1, 1, 1, 0, 0, 0, 0, 0, 0, 0]
[1, 1, 1, 1, 1, 1, 0, 0, 0, 0]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 0]
[0, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 0, 0, 0, 1, 1, 9, 1, 1]
[0, 0, 0, 0, 0, 0, 1, 1, 1, 0]
************************************************

Starting position of agent on playing board is at ( x-coordinate = 2 and y-coordinate = 2 )
************************************************
Checking for Agent's starting position...
Agent's start position found to be legal. Proceeding further...
Agent solves Bloxorz game problem using Breadth First Search strategy as below
************************************************

Step ( 1 )
Agent is on board:  standing straight  at x-coordinate =  2 and y-coordinate =  2
************************************************
  1 1 1
  1 1 1 1 1 1
  1 1 x 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1
            1 1 9 1 1
              1 1 1

Step ( 2 )
Agent  goes right on board:  laying horizontally  at x-coordinate =  3 and y-coordinate =  2
************************************************
  1 1 1
  1 1 1 1 1 1
  1 1 1 x x 1 1 1 1
    1 1 1 1 1 1 1 1 1
            1 1 9 1 1
              1 1 1

Step ( 3 )
Agent  goes up on board:  laying horizontally  at x-coordinate =  3 and y-coordinate =  1
************************************************
  1 1 1
  1 1 1 x x 1
  1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1
            1 1 9 1 1
              1 1 1

Step ( 4 )
Agent  goes right on board:  standing straight  at x-coordinate =  5 and y-coordinate =  1
************************************************
  1 1 1
  1 1 1 1 1 x
  1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1
            1 1 9 1 1
              1 1 1

Step ( 5 )
Agent  goes down on board:  laying vertically  at x-coordinate =  5 and y-coordinate =  2
************************************************
  1 1 1
```

```
  1 1 1 1 1 1
  1 1 1 1 1 x 1 1 1
    1 1 1 1 x 1 1 1 1
            1 1 9 1 1
              1 1 1

Step ( 6 )
Agent  goes right on board:  laying vertically  at x-coordinate =  6 and y-coordinate =  2
************************************************
  1 1 1
  1 1 1 1 1 1
  1 1 1 1 1 1 x 1 1
    1 1 1 1 1 x 1 1 1
            1 1 9 1 1
              1 1 1

Step ( 7 )
Agent  goes right on board:  laying vertically  at x-coordinate =  7 and y-coordinate =  2
************************************************
  1 1 1
  1 1 1 1 1 1
  1 1 1 1 1 1 1 x 1
    1 1 1 1 1 1 x 1 1
            1 1 9 1 1
              1 1 1

Step ( 8 )
Agent  goes down on board:  standing straight  at x-coordinate =  7 and y-coordinate =  4
************************************************
  1 1 1
  1 1 1 1 1 1
  1 1 1 1 1 1 1 1 1
    1 1 1 1 1 1 1 1 1
            1 1 x 1 1
              1 1 1

Agent took 8 step(s) to solve this problem using BFS
```