The SaveWestros problem is a search problem involving Jon Snow as the A.I. search agent. In this problem, Jon should navigate a world with obstacles, dragon stones and white walkers. Jon’s goal is to kill all walkers. Some constraints include the concept of dragon glass. In simplest terms, Jon can only kill walkers if he has a dragon glass, which he can get from dragon stones in the world. Other constraints, Jon cannot walk into obstacles or alive-walkers, but can walk into dead walkers and dragon stones.

For this search problem, the implemented search tree node(SearchTreeNode) has the following attributes:

* State state
  + Denotes the stored state of the node
  + State contains the number of walkers left in the world and a boolean to keep track of whether Jon needs to pick up dragon glass or kill walker at any point
* SearchTreeNode parent
  + Denotes the parent of any node. The root node has null parent
* Operator operatorApplied
  + Denotes the operator applied on a parent to lead to a node
* int depth
  + Depth level of a node
* int pathcost
  + The cumulative cost from a root to a node
* MiniMap worldState
  + A data structure implemented to keep track of the world state after applying a specific sequence of operator
  + Important grid info is stored here, such as grid dimensions, locations of remaining walkers, obstacles and dragon stones. It also has the number of dragon glass Jon is carrying and the maximum he can carry. Finally, a MiniMap has Jon’s current position in the world
* long seqnum
  + An identifier for a node
* int heursticCost
  + The estimate cost till the nearest goal according to a heuristic function

The search problem starts after the grid is generated. After preparing the initial settings for the world, the MiniMap and State, all search algorithms begin execution in different ways. However, the underlying common factors are as follows. All algorithms are implemented as sub-classes to an abstract Search class. This class contains the root node of the search tree(SearchTreeNode), cumulative counters for cost and number of expanded nodes. It also has static methods:

* boolean isGoal(SearchTreeNode node)
  + Checks if ‘node’ satisfies the goal condition
* ArrayList<SearchTreeNode> expandNode(SearchTreeNode node)
  + Expands ‘node’ into all possible children according to whether applying an Operator is a valid action in the world(eliminates most unwanted expansions and cycles)
* LinkedList<SearchTreeNode> backTrack(SearchTreeNode node)
  + Backtracks to the root node back from the starting ‘node’
* int heuristic1(SearchTreeNode node) and int heuristic2(SearchTreeNode node)
  + Two distinct heuristic functions that estimate the cost to the nearest goal from ‘node’

All algorithms contain a data structure that acts as a queue for search tree nodes. Each algorithm however uses a different data structure.

Our implementation of the SaveWestros problem includes all of the previously mentioned classes in the following interaction scheme:

1. Grid is generated, allowing for configuration of initial settings(Jon’s initial position, amount of dragon glasses initially held, initial State and world state, root of search tree, … etc) and hand-over to the appropriate search algorithm
2. Search algorithm class is created, initializing appropriate data structures
3. Algorithm starts execution using the initial root created
4. During execution, if algorithm queue contains a node, it checks whether it satisfies the goal. If yes, algorithm returns. Else, algorithm expands the node into all it’s possible children and adds them to the queue appropriately

\\\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Karim\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

There are two heuristic functions implemented, heuristic1 and heuristic2. Heuristic2 assumes that Jon doesn’t need to move at all. It only counts one ‘Pickup’ action and the number of ‘Kill’ actions required. This heuristic function is very weak but admissible.

Heuristic1, however, calculates the shortest distances from Jon’s current position to a ‘Pickup’ (if needed) and the distances to a sequence of ‘Kill’ actions. This function can overestimate the number of moves since Jon can ‘Kill’ without moving in some situations, but it always underestimates the actual cost of the operations applied since it only counts the operations, and doesn’t consider actual costs. This fact ensures that even if the actual number of actions needed are less than expected, the final estimated cost will always be less than the actual, hence, Heuristic1 is admissible.

\\\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Karim\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//

According to our implementation of the search algorithms, our performance measure is to reach the goal with minimal average expansions and/or with minimal average cost. For a sample 7\*4 grid, one walker at (3,2), one stone at (2,1) and two obstacles at (3,3)&(5,3). Here’s how the different algorithms run(using heuristic1):

* Breadth First Search(BFS):
  + Path Cost = 60, Expansions = 48
* Depth First Search(DFS):
  + Path Cost = 24, Expansions = 702
* Iterative Depth Search(IDS):
  + Path Cost = 24, Expansions = 10(for final run only)
* Uniform Cost Search(UCS):
  + Path Cost = 24, Expansions = 707
* Greedy Search (GR):
  + Path Cost = 24, Expansions = 26
* A\* (AS):
  + Path cost = 24, Expansions = 238

It is important to note that for this grid, only one walker is present. This allowed IDS, BFS and DFS to run without following any infinite branches or having to run for too long. For grids with more spread-out targets, the following outcome is observed:

* Breadth First Search(BFS):
  + Fails
* Depth First Search(DFS):
  + Fails
* Iterative Depth Search(IDS):
  + Path Cost = 69, Expansions = 2152
* Uniform Cost Search(UCS):
  + Path Cost = 48, Expansions = 20707
* Greedy Search (GR):
  + Path Cost = 48, Expansions = 13450
* A\* (AS):
  + Path cost = 48, Expansions = 12942

From these observations, we can conclude that A\* is optimal for either small or large grids and grid configurations

\\\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*Karim\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*//