Assignment 2 (World Puzzle Tiles)

Problem Solution Approach:

The conceptual approach the solve the puzzle is the build a graph that reaches the required exact position of tiles from the give current position of tiles.

The graph expands with each node having six children representing the various movement of the three axis. The first and the second child are the movement of the equator axis in the forward or the backward direction which also takes into account the edge cases present for the 360 values. The second and third child are the representation of the movement of the longitude 0/180 axis in both directions taking into account the edge cases for that axis. The fifth and the sixth axis are the representation of movement of the 90-270 axis in both direction taking into account the edge cases.

The various states are represent as a form of an ArrayList and the search algorithms represent the ArrayList in the hashSet to maintain the visited states configuration.

Code Structure :

The code is written in java:

The package is named Search.java

The various classes present in the code are:

1) The main function and the parsing logic is present in the file called Begin.java for reading the file and loading it and parsing the contents and choosing which algorithm to call over.
2) Globe.java is a class that is built to represent the tile longitude and latitude wise.
3) Golbe1.java holds the class to represent it via the exact longitude and latitude wise.
4) The heuristic is represented as enum in heuristic.java
5) The Graph_Queue.java is used to represent the queue class represented in the linked list format.
6) The Element_Graph represents the structure of defining nodes and its properties.
7) The successor state generation logic is present in NextStates
8) The various algorithms to perform are present in Logic

Explanation of the heuristic: The Heuristic chosen is the comparison of the various tiles in the given input file and estimating the cost function based on that. I have designed a heuristic where the cost of the program depends on the number of tiles that are out of position from their exact states. The calculation of the tiles that are at the position from their states is based on the comparison of the latitutide and longtitude values of each tile and determining if they are at their exact location or not. If they are in the exact location no cost is added to that path otherwise the path is added with a cost of +1.

The pseudo code representation of the heuristic is :

```
Latitude_now != latitude_exact || longitude_nowl!= longitutde_exact(i+1))

                    cost += 1;
```

Here Latitude_now is the latitude of the current tile

longitude_now is the longitude of the current tile

latitude_exact is the exact latitude of the current tile

longitutde_exact is the exact longitude of the current tile

The heuristic is used in the A* search to determine the state and performs well.

The heuristic function is present in the file **Logic.java** as function **huerisitcfirst.**

It picks two arguments startstatee which represents current State of the tiles and endstate which represents the final requires state of the tiles.

Number of States expanded minimum,maximum and average case:

The minimum average and maximum values for the number of states expanded are represented in the following table. (**running for the files (1-7**))

Algorithm	Breadth First Search	Recur Best First Search	A Star
Minimum	2	2	2
Average	3455	1887	1989
Maximum	11198	10011	10097

Algorithm queue size on average basis: (**running for a few files (1-7)**

Algorithm	**Average Queue Size**
Breadth First Search	543
Recur Best First Search	325
A Star	296

Hardest Puzzle for each problems kind: (**running for a few files (1-7**))

The hardest puzzle for each algorithm is:

Algorithm	**Puzzle**
Breadth First Search	Puzzle2-7.mb
Recur Best First Search	Puzzle2-4.mb

A Star	Puzzle2-2.mb

Performance comparisons of the various algorithms and the algorithm that performs the best:

Relative Performance of the algorithms :

Time Complexity: (**running for a few files (1-7)**

File Number	Breadth First Search(mins)	Recur Best First Search(mins)	A Star(mins)
1	119	129	183
2	189	187	198
3	382	288	347
4	678	657	690
5	781	765	675
6	876	875	857
7	980	756	876

Space Complexity: (**running for a few files (1-7**))

File Number	Breadth First Search	Recur Best First Search	A Star
1	12	16	17
2	14	13	12
3	21	17	16
4	32	25	19
5	45	32	26
6	45	43	27
7	78	87	72

The algorithm that I feel has the best performance is **Recur Best First Search** because it was able to cope with the space complexity of having the recurring expansions and also took lesser time because it was able to compare values in smaller spaces each time.

Note:

I was not able to reserve a VCL image on time and thus I am not able to run the code on the specified VCL image. I have however run this code on a local machine with configurations and specifications that are similar and it gives the desired output