**Assignment One**

Peter Fredatovich & Leonard Phillips

***BFS***

BFS searches every element in a tree starting from the root going down moving left to right.

We decided on the STL data structure Queue as our Q because its functionality complements the algorithm. We can add newly expanded child states to the back of a Queue and pop the next state to be expanded off the front very easily.

Pseudo Algorithm Implementation:

queue<Puzzle\*> Q;

Puzzle = new Puzzle(initialState, goalState);

Q.push(begin);

While (Q not empty):

New Puzzle = front of Q

if(Puzzle.state is our goal state):

Return path

Else delete state in Q

If(child state of Puzzle possible and not in search node):

Create all child states of Puzzle and add them to the end of Q

Increase numOfStateExpansions by 1

Update maxQLength

***BFS w/ VL***

BFS w/ VL has the same algorithmic steps as BFS expect for the deciding what is added to the Q. For this reason, we have selected the STL data Structure Queue as our Q. For the VL we have selected the STL data structure unordered\_set<long long>. Since the VL only contains stringified states of Puzzles, we decided that converting that string into a hash long long would make for fast indexing than that of a string comparison. The unordered\_set structure provides very fast element lookup time which is critical for searching through a potential worst-case space of !9

Pseudo Algorithm Implementation:

queue<Puzzle\*> Q;

unordered\_set Visited\_List;

Puzzle = new Puzzle(initialState, goalState);

Q.push(Puzzle);

Visited\_List = begin->getSearchNode();

While (Q not empty):

New Puzzle = front of Q

if(Puzzle.state is our goal state):

Return path

Else delete state in Q

If(child state of Puzzle possible and not in search node):

if(Puzzle.state not in Visited\_List):

Create all child states of Puzzle and add them to the end of Q

And to Visited\_List

Increase numOfStateExpansions by 1

Update maxQLength

***PDS wo/ VL***

PDS wo/ VL expands the left most state starting at the root state, traversing down until it reaches a pre set interval depth initially set at 0. If a goal is found it is returned, if not, the interval depth is incremented, and the search begins again. This is repeated until a goal is found, or the maximum depth is reached. The best data structure to represent our Q is a STL stack. As a stack has LIFO (last in first out) properties, the order is reversed to push the queued states, so they can be retrieved in the correct order.

Pseudo Algorithm Implementation:

stack Q

int intervalDepth = 0

while(intervalDepth < maximum depth):

Puzzle = new Puzzle(initialState, goalState);

stackQueuedStates.push(Puzzle);

while(Q is not empty):

New Puzzle = top of Q

if(Puzzle.state is our goal state):

Return path

Else delete state in Q

if(child state of Puzzle possible):

Add child state to stack

Update maxQLength

Increase intervalDepth by one

***UCS w/ EL***

The UCS finds the shortest path length in Q and expands it if the children are valid. The STL data structure vector provides the functionality we need for fast iterative indexing enabling O(n) complexity when search for shortest path length.

For the EL we have selected the STL data structure unordered\_set. Since the EL only contains stringified states of Puzzles, we decided that converting that string into a hash long long would make for fast indexing than that of a string comparison. The unordered\_set structure provides very fast element lookup time which is critical for searching through a potential worst-case space of !9

Pseudo Algorithm Implementation:

vector<Puzzle> Q

unordered\_set EL

Puzzle = new Puzzle(initialState, goalState)

Q.push\_back(Puzzle)

While (Q not empty):

New Puzzle = front of Q

for (i < Q size):

if(Q[i].pathLength is less than Puzzle.pathLength()):

Puzzle = Q[i].pathLength

if(Puzzle.state is our goal state):

Return path

Else delete state in Q

if(EL does not contains Puzzle state):

Insert Puzzle into EL

If(child state of Puzzle possible and not in search node):

if(child state of Puzzle not in EL):

if(child state of Puzzle not in Q):

Create all child states of Puzzle and add them to Q

Increase numOfStateExpansions by 1

Else(child state of Puzzle is in Q):

if(child state of Puzzle path length < state in Q)

Remove state in Q

Add child state of Puzzle to Q

Increase numOfStateExpansions by 1

Else:

Dont add child state of Puzzle to Q

Update maxQLength

***A\* w/ EL***

A\* is similar to UCS in the way that they both expand the shortest path length in Q.The main difference is how that shortest path length is calculated.

A\* uses a heuristic function that estimates the distance from the current state to the goal state. The two functions are manhattan distance and misplaced tiles. Manhattan distance calculates the distance each tile is from is goal state using the formula |X1 - X2| + |Y1 - Y2|. Misplaced tiles counts the number of tiles in the current state that are not in their goal state position. These heuristic values are then added with the shortest path length to create a F cost. A\* expands the smallest F cost in Q.

A STL vector will provide the functionality we need for O(n) shortest path length lookup.

For the EL we have selected the STL data structure unordered\_set. Since the EL only contains stringified states of Puzzles, we decided that converting that string into a hash long long would make for fast indexing than that of a string comparison. The unordered\_set structure provides very fast element lookup time which is critical for searching through a potential worst-case space of !9

Pseudo Algorithm Implementation:

vector<Puzzle> Q

unordered\_set EL

Puzzle = new Puzzle(initialState, goalState)

Q.push\_back(Puzzle)

While (Q not empty):

New Puzzle = front of Q

for (i < Q size):

if(Q[i].pathLength is less than Puzzle.pathLength()):

Puzzle = Q[i].pathLength

if(Puzzle.state is our goal state):

Return path

Else delete state in Q

if(EL does not contains Puzzle state):

Insert Puzzle into EL

If(child state of Puzzle possible and not in search node):

if(child state of Puzzle not in EL):

if(child state of Puzzle not in Q):

Create all child states of Puzzle and add them to Q

Increase numOfStateExpansions by 1

Else(child state of Puzzle is in Q):

if(child state of Puzzle path length < state in Q)

Remove state in Q

Add child state of Puzzle to Q

Increase numOfStateExpansions by 1

Else:

Dont add child state of Puzzle to Q

else(child state of Puzzle in EL):

if(child state of Puzzle F cost < state in EL F cost):

Remove state in EL

Else:

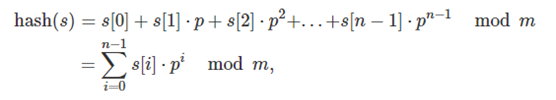
Don’t add child state of Puzzle to EL or Q

Update maxQLength

**Hash Function**

The hash implementation we have chosen is a polynomial rolling hash function. As the states are stored as a string, this has function is a good fit for our model.

The definition for the hash with a string *s*of length *n* is:



where *p* and *m* are some chosen, positive, polynomial number.

We have made p equal to 11. As there are 9 states on the board, we have selected the next prime number after 9, being 11. Testing with this number has shown that at 9 or below, the chance of a collision increases significantly.

m should be as large as possible to avoid collisions, while avoiding integer overflows. A large prime number is ideal, small enough to perform calculation two long longs (64 bit) integers, without overflowing. We have used m = 29 +9.

To compute the hash value, each letter in the string is converted to an int (u->21, r->18, d->4, l->12).

Thus, the returned int is a unique hash of the string.

The collision rate is so for m = 29 +9 the probability of a collision is 2-9. In our model, we have not detected any collisions in the experiments.