**Blocks Project**

**Design**

Store the content of the tray as a hash table. Keys are the size of the blocks, representing in m\*n format, and values are the position of the upper left corner of the blocks. If a size has more than one block, store the positions as a linked list to resolve collisions.

Here is the declaration of the hash table for states:

**HashTable** configuration = **HashTable**<**String**, **ArrayList**<**int[]**> >

where int[] is a 4-element array storing the width, length of the block and x, y position of the block.

Blocks Project Solver Classes:

class **Solver** {

**bPair** size of the tray;

**HashTable** initial state;

**ArrayList<int[]>** goal state;

**HashTable** seenStates;

}

class **States** {

**HashTable** state;

}

class **Block** {

**int** width;

**int** length;

}

class **BlockDes** {

**Block** b;

**int** x;

**int** y;

}

**1. Check if the goal configuration has been achieved.**

After reading the goal.txt file, store the goal as a hash table of sizes and positions. Every time a new state is generated, first check if goal has been reached by iterating through key-value pairs in goal state and check if the the blocks in current state satisfy the requirement.

**Pseudocode:**

for each value in the hash table:

check if the state configuration has that 4-element array as one of the value.

**2. Generate moves possible in the current configuration, and choose the move to explore next.**

In the explore function, compute the empty block(s) of the state. And then search the blocks that are possible, compute their new position and put them into a temporary list, create new states for each selected block. For each new state, call explore function recursively and compute the next explore state in the explore function. Keep the same routine until the end of the path.

**3. Make and unmake moves.**

Use depth-first search and recursive algorithm to make moves, keep exploring the states until there is no possible moves left. One case would be the state satisfies the goal state, which will return true, the other case would be the state generate a duplicated states or the state can not generate any move, which will return false. It only return the path that reach the goal with the least numbers of moves.

**4. Detect previously visited configurations in order to avoid infinite cycling.**

Define a hash table to store the seenStates. Before exploring a state, first check if that state has been seen before. If it is in the seenStates hashtable, stop explorating, otherwise, continue exploring that state.