Daniel Van Der Maden

Blocks Project Write-up

Question: Description of your data structures for the tray and blocks.

My blocks were represented as an object with a Location associated with it. A Location is also an object for easy block movement calculations. All possible locations are instantiated once at the start of the puzzle, then all following blocks reference the already-instantiated location objects. Blocks object contains methods to calculate free moves, determine if it can make a move, and actually move itself. My tray (I called it board in my project) is also an object. The tray object contains a HashSet of blocks as its layout as well as a HashSet of locations to represent all the free locations in the tray. The tray also contains auxiliary object reference information for displaying a successful path. The tray object contains methods to check for a win condition, generate new trays, as well as get the auxiliary information for the output.

Question: A list of the operations on blocks, boards, and the collection of boards seen earlier in the solution search

Board Operations:

- setPath(parent board, moved block, moved direction) → Void "Sets print info"
- move(block, direction) → Board "Generates a new board with the block moved in the direction"
- checkGoal(goal) → Boolean "Checks if the current board matches the goal argument"
- Equals(other)
- hashCode()
- toString()
- copy()

Block Operations:

- getFreeMoves(direction) → Location[] "Check how many free moves a block can make in the given direction"
- movable(Block) → Boolean "Check if a block is movable"
- move(direction, occupied slots, empty slots) → void "Moves a block"
- Equals(other)
- hashCode()
- toString()
- copy()

Seen boards are stored in a hashSet and contains is called after each board movement to check if the board has been seen before.

Question: A list of alternative strategies for selecting moves and alternatives for representing blocks, boards, and the collection of already-seen boards. As well as an analysis of the advantages and disadvantages of each alternative in these lists.

An alternative for selective moves is through some sort of heuristic. Currently, I have it generating moves in a brute force manner. Instead, I could have it prioritise movement on blocks that are close to being solved, thus the need of a heuristic and a priority queue. A simple implementation would use the manhattan distance for the heuristic.

An alternative for representing blocks in a board would be to use some sort of reference hashSet of blocks that all boards could use. This would be useful because most of the blocks on the boards are redundant objects. Have it reference blocks that already exist and only generate new blocks when needed would save memory.

Question: A description of the alternatives you adopted and of evidence you collected that support your choices.

An efficiency change that I did that was radically different than my first attempt at the program was using breadth first search (BFS) to testing out all the possible combinations instead of recursion. This allowed me to solve puzzles without changing the maximum thread size of java.

Another efficiency change that I did was implementing an array of location objects to be reference instead of creating new locations at every board. This resulted in a ~7% reduction in memory usage when solving a large board (seen in task manager).

Another efficiency change that I did was changing the hash codes. Previously I just had the hashCode hash the String representation of the object and this resulted in a ~1300ms lookup time for a contains method check on a hashSet of 25,000 boards. I changed the hash code to be something more unique per board and the same look up took ~300ms (done on the same machine).

NOTE: I implemented a JAVA DOC style of comments for all significant methods. The generated java docs can be found in the "Javadoc" folder.