Remy Orans Ð cs61bl-dt
Quoc Thai Nguyen Truong cs61bl - on
Luis Torres cs61bl - in
Anna Carey cs61bl-ii
ReadMe
Division of Labor:

Our process for this project involved a lot of outlining. We found last project that jumping headfirst into coding was a poor strategy so we all sat around

and argued over which algorithms and data structures we would use until we arrived at a consensus. We subsequently sat around and decided to figure out

exactly which methods we would use. We all agreed upon input arguments and output arguments and the purpose of each method and documented it in our outline

together. After this process we felt it was safe to code separately because we had spent so long discussing our ideas and were all on the same page. Luis

and Thai did the majority of the coding for checker, while Anna and Remy tackled most of solver. Everyone pitched in for debugging and testing.

Design:

We represent Tray configurations with 2DArrays of ÒCoordinatesÓ, which are essentially similar to java.util.points, with a few added instance variables. We

also have a HashMap that maps Coordinates to Blocks objects which contain information about the size of the block the coordinate potentially represents.

The most important methods are canMakeMove and the direction methods: top, left, right, and bottom. CanMakeMoves works by iterating through the path the

block will eventually take and checking if those coordinates are mapped to any block. If it isnÕt it will call one of the direction methods to actually do

the work of changing the data structures to represent the new configuration after the move. In the direction methods we first hash the new coordinate to

its recently moved block object then we remove from our the old coordinate from the map.

Solver was solved through the use of graphs. The vertices of the graph are tray configurations and the edges are moves between different configurations. The

most important data structure for the solver is a priority queue. We push the initial configuration onto the queue and generate all the possible moves for

this board, looping through blocks that can move and generating new nodes from each possible node for each possible block. We push all of these new vertices

onto the queue. The way the queue orders the vertices is based on A* search. A* search is an algorithm that helps find the shortest path between two

vertices, by adding a ÒcostÓ to each vertice. The cost is the sum of the distance to the start and how far the current vertice is from the solution. A fair

way of determining distance to solution for our vertices is summing the distances of each block to where they reside in the goal configuration. In this

manner we prioritize trays that look similar to the goal while also making sure we explore some breadth of the vertices. We tweaked our Checker class in order to generate Node objects which correspond to the vertices of our graph.