**Main Classes:**

\* Board: represents the location of the blocks (1x1, 1x2, 2x2). Contains also all the moves applied to move from one state to another.

\* Block: represents the couple Upper-left and Lower-right coordinate that defines the block

\* BlockCoordinate: represents the X/Y coordinates of either upper-left or lower-right point of a block

\* BoardFactory: creates the initial and target boards from input files

**Solver:**

\* AbstractSolver: abstract class that defines the core features for the solver engine.

\* AStarSolver: inherits from AbstractSolver, defines the specifics of the A\* algorithm. Here, it defines a PriorityQueue with a specific Comparator that allows to sort the boards based on their distance between each other.

\* BestFSSOlver: inherits from AbstractSolver, defines a naive approach that is based on the closest distance from another board

\* BFSSolver: inherits from AbstractSOlver, defines the breadth first search traversal approach which consists in exploring the neighbors first before moving to the next deepest level and so on..

\*\* Further implementation: It would be interesting to use Graph Planning approach to resolve this kind of puzzle. [1] & [2]

I will push a first implementation of this approach in my personal GitHub [3] during the next days.

[1]: <http://lig-membres.imag.fr/PPerso/membres/pellier/_media/publications/pelllier08a.pdf>

[2]: <http://lig-membres.imag.fr/PPerso/membres/pellier/_media/publications/pellier10a.pdf>

[3]: <https://github.com/ibelaidi>

A\* Algorithm in our case:

A\* algorithm goes from a start (root) state then for each action we can perform on the world we generate states for what the world would be like if we did so. If you do this until the world is in the state that we specified as a solution, then the route from the start to this goal state is the solution to your problem.

In our case, A\*, with the PriorityQueue enhanced with a custom Comparator which is based on Distance computed by the defined Heuristic for a better traversal, checks for possible moves in a board, then goes through each recommended move and branches off the fringe with that.

The PriorityQueue chooses which traversal to attend first. The stack depth first traversal will look at which moves it should make and follow the same idea based on the move with the highest priority.

BFS Algorithm in our case:

Breadth-first search allows to find all the vertices reachable from the root. BFS traverses a connected component of a given graph and defines a spanning tree. Intuitively, the basic idea of the breath-first search is this: send a wave out from root. The wave hits all vertices 1 edge from root. From there, the wave hits all vertices 2 edges from root. And so on... We use a Java LinkedList to maintain the wavefront: v is in Q if and only if wave has hit v but has not come out of v yet.

**Heuristics**:

A heuristic is like an algorithm, but with a key difference. An algorithm is a set of steps which you can follow to solve a problem, which always works for valid input. For example you could probably write an algorithm yourself for multiplying two numbers together on paper. A heuristic is not guaranteed to work but is useful in that it may solve a problem for which there is no algorithm.

We need it to help us cut down on this huge search problem. What we need is to use our heuristic at each node to make an estimate of how far we are from the goal. In pathfinding we know exactly how far we are, because we know how far we can move each step, and we can calculate the exact distance to the goal.

\* Supported Implementations:

\*\* ManhattanDistanceImpl: defines the distance between two points measured along axes at right angles.

\*\* LinearConflict: another method to compute the distance between two boards.

**Running Example:**

\* Run "mvn clean package" which will create an executable .jar file

\* then: java -jar SlidingBlock.jar -aAStar -hMH initial-board.txt target-board.txt

\*\*Arguments:

\*\* -a takes AStar, BFS, CFS

\*\* -h takes MH, LC

\*Run from Eclipse/IntelliJ

\* Main class: SlidingBlockTester which takes the same arguments as above.

if none of these arguments are defined, then A\* and Manhattan-distance will be used by default.

I included the start-br.txt and goal-br.txt files which define the boards in the PDF file of the test.

Another files start.txt and goal.txt defines 4x5 board with a moving 2x2 block

