# READ ME

Author: Andrew C. Haynes

## Purpose

This Search Simulator software will create a simulated robot agent in an environment that is essentially a two-dimensional grid that contains obstacles and enemy agents. The amount of obstacles and enemy agents can be modified by the user and environments that you wish to test a search algorithm on again can be saved and loaded back onto the search simulator. There are several different search algorithms that can be used in order to compare which performs better in certain environments.

## Contents

* Head.java
* Environment.java
* Block.java
* Agent.java
* BlindSearch.java
* DepthFirstSearch.java
* BreadthFirstSearch.java
* SCDepthFirstSearch.java
* SCBreadthFirstSearch.java
* HillClimbingSearch.java
* RRHillClimbingSearch.java
* IterativeDeepeningSearch.java
* AStarSearch.java
* MinConflict.java
* SearchSimulator (application)
* A Report on the Performances of the Search Algorithms

## Installation

To install this software on your computer, you have three options.

1) Compile the source codes through command line:

* + Head.java
  + Environment.java
  + Block.java
  + Agent.java
  + BlindSearch.java
  + DepthFirstSearch.java
  + BreadthFirstSearch.java
  + SCDepthFirstSearch.java
  + SCBreadthFirstSearch.java
  + HillClimbingSearch.java
  + RRHillClimbingSearch.java
  + IterativeDeepeningSearch.java
  + AStarSearch.java
  + MinConflict.java

To compile the source codes, use the command “javac [file name]”. For example, “javac Head.java” will compile the “Head.java” file into a “Head.class” file for you. My recommendation is to type in “javac \*.java”. This will compile all of the source codes in the folder with one line! Once they are all compiled, run the software by calling “java Head”. The search simulator should now appear on your computer for your experimentation.

2) Use an IDE like Netbeans, Eclipse, or jGrasp. Put all source code listed above in the same project in your IDE and when you want to run the simulator, compile and run the project. The main class is “Head.java”.

3) Double click on the jar file titled “SearchSimulator” and the search simulator software will start up for your experimentation.

## Using the Software

This search simulator software has several features that will allow it to be adaptable and durable for many uses during run-time.

### Load Grid

You will be able to load up a grid file that can be put into your simulator software for research use of comparing the different results from different search algorithms that you can initiate on the grid. The starting space of the robot agent will always be at the same spot, as well as the spot where the goal object is to be picked up. The file format is built in the search simulator software and should not be messed with or the grid will not load.

\*DO NOT select Load Grid if you have not saved the current grid, if you wanted that grid in your grid archive. To prevent problems, when Load Grid is selected, the current grid on board is deleted.

### Save Grid

The grid that you randomly created can also be saved whether you initiated a search procedure on it or not. You will have the option to give it a name and store it anywhere you’d like on your hard drive. There is no need to add an extension onto the file name for the formatting of the files will take care of that. These files can be called back onto the search simulator software by selecting the “Load Grid” button.

### Create Grid

On the search simulator software, you will notice four input areas called Grid’s Length, Grid’s Width, Grid’s Obstacle %, and Grid’s Agent %. These inputs can manipulate the types of grids you can get for your search algorithms to itself test on. If you want to have a 50x50 grid with a 20 per cent chance of seeing an obstacle and a 10 per cent chance of seeing an enemy agent, you input 50 for Grid’s Length and Grid’s Width and 20 and 10 respectively in Grid’s Obstacle % and Grid’s Agent %. When you input your grid parameters, select the “Create Grid” button and you will see your grid created on the right side of the search simulator software.

\* Only 3x3 grids and up can be created. It was decided that 2x2 and 1x1 grids should not be created since they are too small to really test out a search algorithm’s performance.

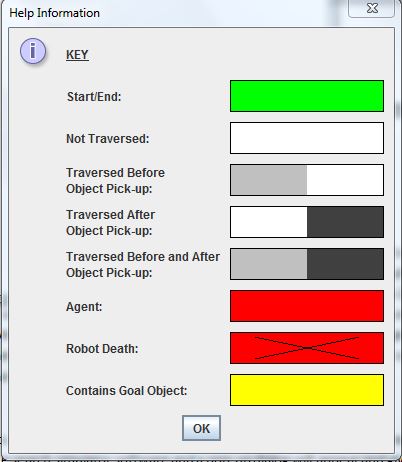
### Start

To initiate a specific search algorithm, select the “Start” button on the search simulator software and a pop-up dialog will reference you to the current search algorithms implemented in the software. When the search algorithm from the robot is finished, another pop-up dialog will appear and give you a record of its performance in that grid.

\*No search algorithm will start if there is no grid on the search simulator software.

### Help

There is also a key that explains the various states that a block in the grid represents. Select the “Help” button on the search simulator software and a pop-up dialog will appear and give reference to the various states the blocks in the grid will be after a search algorithm is initiated. For internal documentation use, here is the visual aide of the key:



## Search Algorithms

Random Search (Blind Search) – Randomly chooses its next direction blindly. It can even decide to bump into a wall twenty times because the random number generator decided it so.

Depth-First Search [Tree] – The tree based version goes down into the depths of its leftmost block. It does not check for repeated blocks. Therefore, this search is going to perform horribly but it is here to show how the graph version performs much better.

Depth-First Search [Graph] – The graph based version does the same movement as the tree based depth-first search but it performs better because it checks for repeated blocks and does not visit these blocks again so now it will not get stuck.

Breadth-First Search [Tree] – The tree based version will go through all of its neighbors before visiting the neighbor’s neighbors. This version is without state-checking.

Breadth-First Search [Graph] – The graph based version does the same as the tree based version but performs better by doing repeated block checks.

Hill Climbing Search – The environment will be in a complete state for the search to go through. It uses straight-line heuristic to determine the quickest path to the treasure block and back. It can get stuck because of obstacles.

Random Restart Hill Climbing Search – Improves the Hill Climbing Search by restarting it if it gets stuck. It does not restart if it has the treasure picked up or else it ruins the search’s performance/purpose overall.

Iterative Deepening Search – All of the blocks in the environment will know their depth before this search begins. This search has a limit as to what depth it can go to each time it goes through the environment. If the search does not find the treasure or the home block on the way back from the treasure, it increases the limit by one and restarts the search.

A\* Search – The environment will be in a complete state for the search algorithm to go through. The environment will use a straight-line formula for its heuristic to determine how far a block is from the current objective. Each time the robot agent moves, it increases the cost to go to that current block by one in order to not revisit that block in future movements.

Min-Conflict – This is a constraint-satisfaction problem search (CSP Search) where the constraints are to allow no obstacles or agents in the path the robot agent takes. The robot agent will also never re-visit a block during the search for the treasure and then for the search back to the entry point.