# EECS 391: Introduction to AI (Spring 2016) Programming Assignment 1 (Max Points: 100)

**Assigned Tuesday January 21, due midnight Thursday February 11. Turn in your code using blackboard. Please comment your code extensively so we can understand it, and use sensible variable names. If two people worked on a submission, please include both your names and IDs in a README. You may only turn in an assignment as a pair if both of you have contributed equally to it.**

In this assignment, you will implement the A* search algorithm for pathfinding in SEPIA. SEPIA is a real-time strategy (RTS) game (though we will not use the real-time aspects in these assignments). RTS games generally have "economic" and "battle" components. In the economic game, the goal is to collect different types of resources in the map. Typical resources are "Gold" and "Wood." Resources are collected using units called "Peasants." Having resources allows the player to build other buildings (Peasants can be used to build things) and produce more units. Some of these units are battle units that can be used to fight the opponent. Games generally end when one player has no more units left; however, in SEPIA, a variety of victory conditions can be declared through XML configuration files. For example we can declare a victory condition to be when a certain amount of Gold and Wood have been collected, some number of units of a certain type built, etc.

From the website/blackboard, download PA1.zip. You will find some agent source code (in the src/ subdirectory) and maps and configuration files (in the data/subdirectory) for this assignment.

## Pathfinding (70 points)

Write an agent that can move around a given map by implement the A* search algorithm discussed in class. In the file AstarAgent.java, find the function AstarSearch.java and fill it in. This function should return a path from the starting location to the goal. The rest of the agent code has already been filled in so that once you implement the search, the agent will execute it in the game. During this step, it will output its progress with helpful messages that should let you debug your code. You can also watch VisualAgent (the GUI window showing the map) to see how your found path is being followed. The terminal output will also show the total time taken by the process. You should try to reduce this as much as possible (i.e. write efficient code!). For a proper estimate of the time taken, you can stop VisualAgent from running by deleting or commenting out the corresponding <agentclass> lines from the configuration file. We may award up to 10 bonus points if (i) your code is clean and comprehensible and (ii) your total runtimes are among the top three in the class. Please do **NOT** use map-specific or heuristic-specific optimizations in your A* implementation to get your code to run faster. This means your implementation should be able to handle any map and any heuristic.

For A*, you will need a heuristic function. The Chebyshev distance is a good heuristic for this purpose. This is defined as follows: $D((x_1,y_1),(x_2,y_2))=max(|x_2-x_1|,|y_2-y_1|)$. To test your algorithm, use the maze maps provided. In each map, a Footman is trapped in a maze. Somewhere in the maze is an enemy Townhall. The provided code will use your implementation to find a path that takes the Footman to the Townhall and attack it. When the Townhall is destroyed, the game will end. If it is not possible to guide the Footman to the Townhall, print "No available path." in the terminal and quit (call System.exit(0)).

You can use VisualAgent to check that your agent is behaving correctly. You can run the maze maps just using VisualAgent, left click on the Footman and right click on the Townhall to see the solution according to the built in pathfinding routines. Note that we will test your code on other maps and heuristics as well, so ensure nothing is specific to the maps or heuristics provided.

## Pathfinding with Interference (30 points)

In a real game, many units will be wandering around, and pathfinding is complicated. We will simulate this using a simple scenario.

One of the provided maps (maze_16x16h_dynamic) is an environment with an enemy footman, controlled by the wicked EnemyBlockerAgent. This agent will try to prevent your agent from destroying the enemy Townhall by blocking their path (it won't attack you otherwise). To get around this, use the "shouldReplanPath()" function in AstarAgent. Here, you can check to see if the current path is blocked. If so, you should return true and the agent will redo the search from that point. Try to write a nice function which is smart about when it needs to redo the search so you minimize the total time spent in searching and execution.

## What to turn in

Prepare a ZIP file with AstarAgent.java and a README (do NOT include any class files, or any SEPIA code). Include a README with your name(s) and ID(s) and any other comments you have. Name your file as "yournetworkID(s)_PA1.zip" and use Blackboard to submit it. If you worked as a pair, ensure that both your IDs appear in your submission. Only one submission is required if you worked as a pair.