CPSC470/570 Artificial Intelligence Win13 Programming Project A

Goal-Based Agents: Pursuit World

**Due Monday, 2013-02-11**

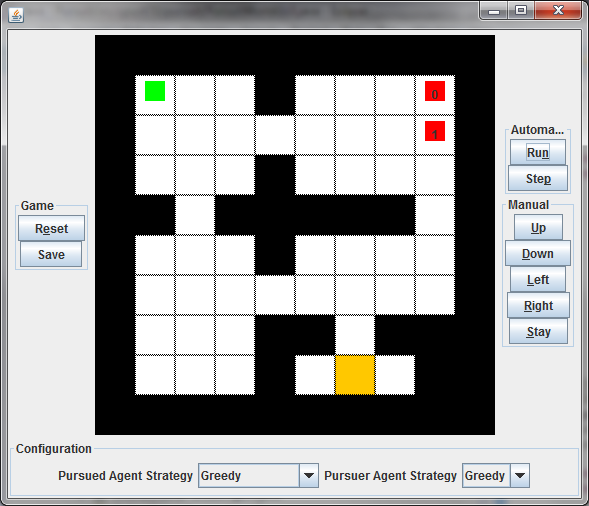
**Name(s)**: \_\_Anthony Marcheselli, Ly Nguyen\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**REMINDER:** You must solve these problems only with your team members. You may not reference the work of others, solutions from previous offerings of this or related courses here at Seattle University or elsewhere, or any other solutions.

# Introduction

This project involves building simple (simulated) agents that exists in a *Pursuit World* environment. There are two types of agents: a pursued agent (referred to as “the hobbit” from now on) and the pursuer agents (referred to as “the orcs” from now on). The hobbit’s goal is to get to safety while avoiding orcs trying to capture it. You will implement search-based strategies for both types of agents to use.

## The Pursuit World

To get started you must first understand the environment the agents will navigate. A ***sample*** environment is shown in the screenshot below: 

I describe each component below. Note that this is just a sample and not the only environment your agents may need to navigate. I will provide a few environments as part of this assignment; your code should be general enough to work in these or any other environment meeting the specifications below.

## The Maze

Each environment is a two-dimensional grid world which is divided into *cells* which appear as the individual squares in the grid above. The cells in the grid world are numbered according to a Cartesian coordinate system where the y-axis is inverted, with the cell (0,0) being in the upper left-hand corner of the grid. Squares that are entirely black denote walls or obstacles (blocked cells) in the environment. The agents are free to move into any non-blocked cell. As you can see, the cell (0,0) is blocked, and in general you can assume that the perimeter of the world will always be enclosed by blocked cells.

The orange-colored cell is safety and is the goal location for the hobbit. In the example above, the safety location is at (6, 8). For this assignment, you may assume that every environment has exactly **one** safety cell.

## The Hobbit

The hobbit occupies one cell within the environment and is denoted graphically by a smaller green box, which lies completely inside a cell. In the example above, the hobbit is located in cell (1, 1).

The hobbit can perform five simple movement actions in the grid world: Up, Down, Left, Right, and Stay. The goal of the agent is to get to the safety square.

## The Orcs

The orcs look just like the hobbit, but are colored red and are numbered. In the example above, the orcs are located in cells (8,1) and (8,2). It is possible for multiple orcs to be on the same square.

Orcs can perform the same movements the hobbit can. After each move the hobbit performs, each orc gets one move of its own. The goal of the orcs is to get to the hobbit (share the same cell), thus capturing it. Note that if an orc intercepts the hobbit at the safety location, the hobbit is still considered safe.

## How the Game Ends

If the hobbit reaches safety, or an orc captures the hobbit, then the game ends. The game will also end if 150 moves go by without either of these events occurring.

## The Controls

There are three sets of controls: game controls, movement controls, and configuration controls.

### Game controls

* Reset button: Resets the game to the initial state, clears the game transcript.
* Save button: Opens a dialogue to save current game transcript to a file.

### Configuration controls

* Dropdown menus for selecting strategy used by each category of agent. Note that changing one of these values resets the game. The NoOp strategy leaves the agent where it is throughout the game and can be useful for disabling agents while you test.

### Movement controls

* Automation controls
  + Run button: Runs game to completion with each agent making moves based on its selected strategy. (Note: If the run speed is not to your liking, you can change it via the PursuitWorldGUI.NUM\_MILLISECONDS\_BETWEEN\_STEPS constant.)
  + Step button: Move each agent one step based on its selected strategy.
* Manual controls
  + Up, down, left, right, and stay buttons allowing you to override the automated behavior of the hobbit. (Arrow keys also work as expected, and space will activate a stay action.) Each orc will automatically make one move after a manual hobbit move.

# Code Infrastructure

For this assignment, most of the code will be given to you. Pursuit world is written on top of the [aima-java](http://code.google.com/p/aima-java/) implementation of the algorithms in the textbook.

## Setup

Follow the instructions at [aima-java website](http://code.google.com/p/aima-java/) to download and setup the latest version.

Download the pursuit world project provided in the compressed file labeled “programming project a – provided code - pursuit.zip” in the assignments folder under the Lessons section of the course Angel website. Extract the Pursuit project and add it to your workspace.

The program takes one input argument, the path to a world configuration file, described below. I have provided you with configuration files in the project inputs subfolder for some sample worlds (e.g., the world shown above is specified in small.wld), but these are not necessarily the only worlds I will test your code on.

## Pursuit Project

The project has three packages: cpsc470.pursuit, cpsc470.pursuit.environment, and cpsc470.pursuit.agent.

### cpsc470.pursuit

This package contains the classes with the program’s main entrypoint and the GUI control:

* PursuitWorldGame
* PursuitWorldGUI

You shouldn’t need to change anything in this package to solve the assignment.

### cpsc470.pursuit.environment

This package contains components of the environment external to the agent:

* PursuitWorldEnvironment
* Maze
* PursuitWorldPercept
* PursuitWorldAction

You shouldn’t need to change anything in this package to solve the assignment.

### cpsc470.pursuit.agent

This package contains the only files you should need to modify to solve the assignment. My design uses extensive subclassing to relate the various agents so as to capture the natural relationships between them and to maximize code reuse. First, there are a few interfaces used mainly for type control. I’ve listed them according to their subclass relationship:

* PursuitWorldAgent
  + PursuedAgent
  + PursuerAgent

Here is the list agent classes, again ordered by subclass relationship:

* AbstractPursuitWorldAgent
  + AbstractOnlineGreedyPursuitWorldAgent (+ AbstractOnlineGreedyPursuitWorldAgentProgram)
    - OnlineGreedyPursuedAgent (+ OnlineGreedyPursuedAgentProgram)
      * OnlineGreedyTabuPursuedAgent
    - OnlineGreedyPursuerAgent (+ OnlineGreedyPursuerAgentProgram)
  + AbstractAStarPursuedAgent
    - AStarPursuedAgent
      * SmartAStarPursuedAgent
  + AbstractNoOpPursuitWorldAgent
    - NoOpPursuedAgent
    - NoOpPursuerAgent
  + (optional) OpenPursuedAgent
  + (optional) OpenPursuerAgent

These contain the “smart” parts of the code, those that control the agents. Many of these have pieces “chopped out” to allow you to implement them. However, you don’t have to feel constrained to precisely follow the structure I’ve set up. Feel free to create new classes and/or changes existing ones if you prefer, as long as you meet the requirements of the assignment.

## aima-java Project

The aima-java project contains implementations of all the algorithms in the book, including the search algorithms we want to experiment with. Feel free to browse through the relevant packages for useful stuff. I’ve listed below some to take particular note of.

### aima.core.agent, aima.core.agent.impl, aima.core.agent.impl.aprog

Interfaces and classes for a variety of agent types.

### aima.core.search.framework

The core infrastructure for search.

### aima.core.search.informed

Implementations for informed search algorithms, including A\*.

### aima.core.util.datastructure

A collection of useful data structures. We use the XYLocation class extensively in this assignment.

## aima.test.core.unit.\*

A comprehensive collection of unit tests organized in a similar way to the code they test. These can be very useful to see examples of how to use the aima-java API.

# Data

The world configuration files specify the maze (including safety location) and the locations of the agents. Below is configuration for the small world shown above:

// maze

// - 2x2 array of 1s (walls) and 0s (spaces)

// - space-delimited rows

0 0 0 1 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 1 0 0 0 0

1 0 1 1 1 1 1 0

0 0 0 1 0 0 0 0

0 0 0 0 0 0 0 0

0 0 0 1 1 0 1 1

0 0 0 1 0 0 0 1

// locations of goal and agents

// - top left corner of maze is (1, 1)

goal: (6, 8)

pursued: (1, 1)

pursuer: (8, 1)

pursuer: (8, 2)

Empty rows and rows beginning with “//” are ignored. Also, while one and only one pursued agent is required, it is possible to have any number of pursuer agents including none at all.

# Tasks

## Problem 1 (10 points)

Implement a simple reflex agent that implements a greedy hill-climbing strategy for both the hobbit and the orcs. The rules for the greedy strategy are as follows:

1. Make no illegal moves (never try to walk through walls).
2. Use greedy hill-climbing with a Manhattan distance heuristic function to the goal (for the orcs, the goal is the hobbit).
3. If there is a tie between two axes, try to minimize the distance along the axis which is farther from the goal. For example, if the hobbit is at (2,2), safety is at (3,10), and the hobbit can move either right or up, choose up.
4. If still tied, prefer vertical movement to horizontal movement.
5. If all one-step moves take the agent away from the goal, then the agent moves nowhere.

Your job is to write the code that returns the correct move from the five possibilities (Up, Down, Left, Right, Stay) for each agent. It should be sufficient to fill in the placeholders in AbstractOnlineGreedyPursuitWorldAgentProgram and its subclasses OnlineGreedyPursuedAgentProgram and OnlineGreedyPursuerAgentProgram to solve this problem.

You can test your code by setting both agent strategies to “Greedy”. “smallGreedy.out” in the “inputs\expected outputs” folder is the transcript you should expect if you save to file after running the program on “small.wld.” If you want only the orcs to move, set the pursued agent strategy to “NoOp”; “smallPursuers.out” is the expected output in this case.

Answer the following question:

1. List and describe the changes you made to the code. Confirm whether or not your program transcripts on the provided worlds matched the provided output. If it does not, also turn in the transcripts it did produce.

The changes made were as follows:

**In AbstractOnlineGreedyPursuitWorldAgentProgram.java**

The function getGreedyMove() was updated to include a list which is filled up with all the spots of the valid available moves.

Next, another list is filled up in parallel to the moves list which contains the manhattan distance from the potential move to the the goal location. The function then selects the action with the lowest hueristic value and takes that as its moves.

In the case of ties, a tie breaker section was included to highlight all the general cases that need to be considered.

**For the manhattan distances, a separate class called ManhattanDistance.java was included.**

While this could have been implemented as a simple member function, we chose to use a class because we knew that the aStar agent would make use of the manhattan distance as well.

In addition to the getGreedyMove() additions, we also included a helper function called checkAxis() which was used for the tie breaking section of the get greedy move function.

Program transcripts do match the provided output for pursued agent and the pursuer agents.

## Problem 2 (15 points)

One extension of greedy hill-climbing search is called *tabu search*. This involves storing some history of where the agent has been so as to avoid some of the inadequacies of greedy hill-climbing (like local optima and plateaus).

Extend the greedy strategy to use a tabu list. We are going to use a naïve implementation of the tabu list. Keep a list of the last *n* states that the agent visited (including the state that the agent is currently in). The agent will now choose the best move (according to the greedy strategy) that is not in the tabu list. If there are no legal moves that are not in the tabu list, clear the tabu list before selecting your move. The variable *n* is constant throughout a game, but you should be able to easily recompile your program with a new value of *n*.

Make sure that you implement the third greedy rule correctly when the agent is moving *away* from the goal. For example, if the agent is at (2,2), the goal is at (3,10), and the agent can only move either *left* or *down*, then choose *left*. You may also have to choose between two directions along the same axis. In this case, choose left over right, and down over up. For example, if the agent is at (4,6), the goal is at (4,10), and the agent cannot move up, then choose left.

For this problem, you only need to make tabu search available to the hobbit. It is sufficient to fill in the placeholders in OnlineGreedyTabuPursuedAgentProgram to solve this problem. You will need to choose an appropriate data structure strategy for the tabu list and also implement the clearing of the list.

Test this code by selecting the “GreedyPlusTabu” strategy for the hobbit. Test the agent on both tabu1.wld and tabu2.wld (note that there are no orcs in these worlds). tabu1.out is the correct output for the former when the size of the tabu list is 1000 (making the tabu list a closed list). Then answer the following questions:

1. List and describe the changes you made to the code. Confirm whether or not your program transcripts on the provided worlds matched the provided output. If it does not, also turn in the transcripts it did produce.
2. In each of the two worlds above, what is the minimal size of the tabu list such that the agent finds the goal? Explain the difference between these two numbers, and try to make a general statement about how properties of the search space affect how large the tabu list needs to be.
3. Given this greedy strategy and implementation of the tabu list, can the agent ever fail to reach the goal (assuming that a path does exist) if the size of the tabu list is greater than the entire state space? Either give an example where this would happen, or briefly describe why the agent will always find the goal.

**A**

This function makes use of the same code from getGreedyMove() but extended to now make use of a list that stores previous spots.

Function clearTabuList() filled out to empty the implemented list of previous locations.

In getGreedyMove, new tiebreaking rules and conditions have been added to match the requirements. Additionally the method in which available moves are stored has been change to allow for temporary removal of moves which can then be restored in the case that there are no valid moves.

Function checkAxis() was resused from the part 1 implementation and function awayFromGoal() was added to check for addional tie breaking conditions.

Program transcript does match the provided output for the tabu agent.

**B**

tabu1.wld smallest tabu list size = 5

tabu2.wld smallest tabu list size = 14

In tabu world 1 the minimum list size is so small because of the fact that all the wall boundaries are disjoint and only affect the immediate area around them. Essentially, they become 3 sparate difficulty areas for which the tabu agent can deal with separately because once it is past them, they are no longer part of its path to the goal. On the other hand in tabu world 2, the wall forms a single contigous barrier which the agent needs to be able to search along the entire side before being able to find a better greedy spot.

As a generality, the search space needs to be at least as large as the area taken up by the largest obstacle that gets in the way of the agent and its goal. In these two specific cases, the minimum list size was slightly smaller but that has to do with the shape and initial postion in relation to the goal position.

**C**

The greedy strategy with tabu list can fail in any case where the agent is located in an area that is square shaped and only has one exit. If the exit is placed on the opposite direction of the goal location in relation to the agent, it will be unable to find the exit. Additionally the exit must not be situated near a corner.

for example the map:

1 1 1 1 1 1 1 1 1

1 0 0 0 0 0 0 0 1

1 0 0 0 G 0 0 0 1

1 0 0 0 0 0 0 0 1

1 0 1 1 1 1 1 0 1

1 0 1 0 0 0 1 0 1

1 0 1 0 0 0 1 0 1

1 0 1 0 @ 0 1 0 1

1 0 1 0 0 0 1 1 1

1 0 1 0 0 0 1 0 1

1 0 1 1 0 1 1 0 1

1 0 0 0 0 0 0 0 1

Where @ is the agent location

and G is the goal, the agent will never find the path out because it will never stop near the exit, and even if it does, the exit path will always be one cost more than any of the other valid moves. The size of the tabu list plays no role because of the fact that its greedy habits will never let it consider the exit move unless if it traversed the entire room and then it landed on the last room spot which happened to be right next to the exit (this is the reason why I said that the exit cannot be next to a corner or the tabu agent can escape from the room.

## Problem 3 (10 points)

Implement a simple A\* search strategy with the Manhattan distance heuristic, that is guaranteed to get the hobbit from the initial state to the goal with the shortest path if there are no orcs. States in our search will be simplified environment states extracted from percepts; they are AbstractAStarPursuitWorldAgent. AStarPursuitWorldState objectes. For this problem, A\* should treat states in the search as only differing in the position of the agent (even though more information may be available in the objects themselves).

It is sufficient to fill in the placeholders in AbstractAStarPursuitWorldAgent and AStarPursuedAgent to solve this problem. These changes include filling out the agent and search problem components needed for this problem (e.g., actions function, result/transition function, goal test, etc.).

In order to perform a search, select the “AStar” strategy for the hobbit. “tabu1AStar.out” is the correct output when this code is run on “tabu1.wld.” Running the search will plan a path and then move the agent. Note that nothing will happen for a moment as the agent executes the search to find a path. Once it has the path, it will execute it on the UI. The initial node for the search is the current environment. The goal for the search is the environment in which the hobbit has reached safety. The transcript of a game will include time and space metrics for the search performed.

Answer the following question:

1. List and describe the changes you made to the code. Provide transcripts for your agent’s performance on the tabu1 and tabu2 worlds. Confirm whether the former matches the expected output.

-Filled out AbstractAStarPursuitWorldAgent.PursuitActionsFunction.actions() to provide a set of actions that would provide all the available adjacent squares of the agent using getAvailableAdjacentLocations().

-Filled out AbstractAStarPursuitWorldAgent.PursuitResultFunction.result() to provide a new state where the only difference from the last is the pursued agent’s location.

-Filled out AbstractAStarPursuitWorldAgent.PursuitWorldManhattanHeuristicFunction.h() to provide a manhattan distance heuristic using out ManhattanDistance class methods.

-Filled out AStarPursuedAgent.PursuedAgentGoalTest.isGoalState() to tell whether the pursued agent is at its goal state, which aligns with the coordinates of the safety location on the maze resulting from getSafetyLocation().

-Outputs on tabu1 and tabu2 attached in outputs/ directory

-The former matches the expected output in terms of path cost, but the path itself is slightly different and the number of expanded nodes it several nodes greater. That is ok, because the actual path depends on the order of the actions presented in the actions function.

## Problem 4 (15 points)

Now run the hobbit with A\* in the world large.wld, and see what happens when orcs are added that follow the greedy strategy you wrote above.

In this problem, you will create a smarter version of the A\* agent so that A\* is guaranteed to find a path to the goal if one exists. Thus, your agent must (if possible) avoid all orcs, **knowing that the orcs will use a greedy strategy**.

It is sufficient to fill in the placeholders in SmartAStarPursuedAgent to solve the problem. As needed, override your implementations from the previous question. (You might not have to override them all.) For example, in the original A\*, child nodes were constructed by applying a move for the hobbit. You may want to override the result function to make any other changes you want to occur in a state as it advances one timestep. (Tip: If you want the search not to examine a state, you might consider transforming it into an illegal state – e.g., one where the hobbit is at (0,0) – and define a step cost function that assigns infinite cost to going to such nodes.)

In order to perform a search, select the “SmartAStar” strategy for the hobbit.

Answer the following question:

1. List and describe the changes you made to the code. Provide transcripts for your agent’s performance on the tabu1 and tabu2 worlds. Confirm whether the former matches the expected output.

-Transcript on outputs for Tabu1, Tabu2, and Large worlds attached in outputs/ directory.

-Filled out SmartAStarPursuedAgent.SmartAStarPursuedAgentState.equals() to check the sameness of locations of all the agents

-Filled out SmartAStarPursuedAgent.SmartAStarPursuerResultFunction.result() to return the resulting state for an action that the pursued agent takes – it returns a new state where the pursuer agents’ locations have also moved based on the greedy algorithm and the pursued agent’s new location.

-Filled out SmartAStarPursuedAgent.SmartAStarStepCostFunction.c() to return 1 as a normal step cost, and positive infinity as a step cost function of a step that results in the pursued agent’s capture.

-The former test run on the world matches the expected output. Like with the dumber A Star state model, the path cost was also 29, but the actual path taken is slightly different because the actions function returns a different order of actions available. The number of expanded nodes is also a little bit higher.

## Extra Credit (up to 10 points)

You have the option of submitting more intelligent strategies for both the hobbit and the orcs, and you will receive extra credit points based on how clever your strategies are. (FYI, simply transferring the tabu list or A\* extensions you implemented above for the hobbit to the orcs is *not* likely to net you a lot of extra credit. ☺) Provide a written description of your approach; the description should be short but sufficiently detailed to allow me to reconstruct what you did. Feel free to construct your own worlds for testing purposes following the syntax described earlier. Include in your submission the worlds which you think best demonstrate your idea, and tell me the filenames for these worlds in your write-up.

Your strategies for the hobbit and the orcs should be written in OpenPursuedAgent and OpenPursuerAgent, respectively. In order to run any extra credit code you write, you can select the “Open” strategy for either type of agent. Note that the default open strategy is one in which the agent does not move.

# Deliverables

Select a team member as contact point to do the submission. Have that person turn in the following to the “programming project a drop box”:

1. A compressed copy of your pursuit project. This should include any world files you created as well as the transcripts for any worlds you ran the game on.
2. Your write-up for questions above, including your extra credit write-up (if applicable). Make sure to include the names of all team members in your write-up.