Computer Science 384 January 26, 2014
St. George Campus University of Toronto

Homework Assignment #1
Due: Thursday February 13, 2014, by 11:59 PM

Silent Policy: *A silent policy will take effect 24 hours before this assignment is due. This means that no question about this assignment will be answered, whether it is asked on the newsgroup, by email, or in person.*

Late Policy: Late penalty of 15% per day after the use of 2 grace days.

Total Marks: There are **86 marks** available in total. This assignment represents **10%** of the course grade.

Questions worth one (1) mark: You'll notice that some of the questions are worth only one mark. The answers to these questions will be inspected but not marked. We will take them up in class. Going to the effort of answering them will help you prepare for the test and exam.

Handing in this Assignment
What to hand in on paper: For this assignment, no paper submission is required.
What to hand in electronically: You must submit your assignment electronically. Download `a1handin.pl` and `a1answers.txt` from the course web page, fill in the definitions of predicates in `a1handin.pl` in the space provided (defining helper predicates as necessary) and answer questions in `a1answers.txt`. Submit both these files.

To submit these files electronically, use the CDF secure Web site:
`https://www.cdf.utoronto.ca/students`

Warning: marks will be deducted for incorrect submission (e.g. wrong predicate names).

Since we may test your code electronically, you must:

- *make certain that your code runs on CDF*,
- use the exact predicate names and argument(s) (including the order of arguments) specified,
- include all your code in `a1handin.pl`,
- not load any file of yours from within that file,
- not produce any output in your predicates, i.e. the only output you should see when calling your predicates is the variable binding Prolog displays if the call was successful,
- follow the instructions in `a1answer.txt`.

Marking Questions in this assignment will be (partially) automarked. Note that we may automark your code using test cases (here: maze configurations) that are *different* from the ones given to you here.

Clarification Page and Newsgroup: Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the A1 Clarification page, linked from our CSC384 A1 Assignment page. You are responsible for monitoring the CSC384 A1 Clarification Page.

Questions: Questions concerning the assignment should be directed by e-mail to the Assignment 1 TA, Brent Mombourquette (bgmomb -at- cs.toronto.edu). Please place "384" and "A1" in your e-mail subject header.

Prolog Notational Conventions: Predicate and Mode Spec

We shall use the following notation when *referring* to Prolog predicates: Predicates in Prolog are distinguished by their name and their arity. The notation `name/arity` is therefore used when it is necessary to refer to a predicate unambiguously; e.g. `append/3` specifies the predicate named "`append`" that takes 3 arguments.

Sometimes, we may be interested in specifying how specific predicates are meant to be used. To that end, we shall present a predicate's usage with a *mode spec* which has the form: `name(arg1, ..., argn)` where each `argi` denotes how that argument should be instantiated when a goal to `name/n` is called. `argi` has one of the following forms:

+ArgName This argument should be instantiated to a non-variable term.

-ArgName This argument should be uninstantiated.

?ArgName This argument may or may not be instantiated.

For example `delete(+List,?Elem,?NewList)` states that, when using `delete/3`, the first argument should be instantiated whereas the second and third arguments may or may not be instantiated.

Note that these Prolog notational conventions provide a convenient way to *specify* Prolog predicates and their usage. They do not represent in any way the form of your actual code. E.g., when defining the predicates in the following questions, do not put `+/-/?` in front of the arguments in your code.

Do's and Don'ts

1. You should not use any special Prolog/SWI-Prolog features like `assert, retract, arg, ...` Put differently, the only predicates you may use apart from the ones you implement yourself are `not/1, append/3, member/2, length/2, 'is', sqrt/2, floor/2 abs/1`, arithmetic symbols (`'=', '+', '-', '*', '/'`), the symbols `'\=', '->', '!', ';', and of course ','`. Of course you may also use output predicates like `writeln/1` for testing and debugging, but these have to be removed prior to submission. If there is another built-in predicate you wish to use, please ask the TA.

2. Avoid *singleton* variables! A singleton variable is a variable that only occurs once in the arguments or the body of a predicate and is thus useless. For example in the following two definitions:

   ```
   doit( X, Y, Z) :- Y is X*2.
   doitagain( X, Y ) :- Z = doesitmatter , Y is X*2.
   ```

   `Z` is a singleton variable and should be prefixed by an underscore (`_Z`) or removed entirely if possible. Note that it usually won't be possible to simply remove a singleton argument as the arity of the predicate is often fixed:

   ```
   times( [], [], Z) :- !.
   times( [H1|L1], [H2|L2], Z ) :- H2 is H1*Z, times( L1, L2, Z).
   ```

   Here in the first definition `Z` is singleton but cannot be removed. Therefore we should replace it with '`_Z`' or just '`_`'.

   Although not harmful, it is useful not to have any singleton variables to keep the code easy. SWI-Prolog will point out any singleton variables you have. This is very useful information because it often helps you finding typos, a common source of bugs in Prolog. For instance if we want to increase a number we could write:

   ```
   inc( FirstNr , Result ) :- Result is Firstnr+1.
   ```

   Here both `FirstNr` and `Firstnr` are singletons and SWI-Prolog will tell you so, which in turn will make you realize that you have a typo (`Firstnr` should be spelled with a capital 'N').

Solving a Maze

In this assignment you are going to implement a maze solver using three different search algorithms, A^*,A^* with cycle-checking, and IDA^*. We are providing you with generic implementations of these algorithms in SWI-Prolog:

1. A^* search with path checking (`astar.pl`).
2. A^* search with cycle checking (`astarCC.pl`).
3. IDA^* search with path checking (`idastar.pl`).

All of the above files use some common code from `astarcommon.pl`. Also, some simple examples of search spaces that show how these search routines are applied (`simpleSpace.pl`, and `waterjugs.pl`) are available.

Your task will be to formulate the following maze problem as a search problem and to run experiments with these algorithms. Suppose that there is a robot on an MxN-grid (M rows and N columns). The robot can only move one position (right/left/up/down) at a time with uniform cost of 1 for each direction. The robot is located on the starting position S (e.g. 1/1) and has to move to the goal position G (e.g. 9/9).

Besides the robot there exist several obstacles `X` on different positions on the grid that prevent the robot from moving into these positions. Figure 1 shows one possible instance of the search problem for a 9x9 grid:

```
S-----X--
--X-----X
XX----XX-
----XX---
---X-----
--X----X-
--X---X--
-X--X-X--
-----X--G
```

Figure 1: A sample search problem instance.

The state of the robot is represented as a single C/R coordinate (C being the column number and R being the Row number), with $1/1$ being the upper left corner, $N/1$ the upper right corner, $1/M$ the lower left corner, and N/M the lower right corner. Obstacle are represented by a list of C/R-pairs corresponding to the coordinates of each obstacle. Each maze is represented as `maze(MazeName, M, N, O, S, G)`, where `MazeName` is the name of the maze, M and N are the number of the rows and columns respectively, `O` is the list of obstacles coordinates, and `S` and `G` are the start and goal states. For instance, the example maze of Fig. 1 would be encoded as follows:

`maze(sample1` *, 9, 9, [7/1,3/2,9/2,1/3,2/3,7/3,8/3,5/4,6/4,4/5,3/6,8/6,3/7,8/7,2/8,5/8,7/8,6/9], 1/1, 9/9*)

Objective: The objective of your implementation is to determine a *shortest path* from the start position to the goal that the robot can take. You are to set up the problem as a search problem, using the given representation of the state space, and using the provided search routines to find a solution. Therefore, your implementation of this search problem will consist of the following items.

Successor Predicate Implement the predicate `successors(+State, -Neighbors)` that holds if and only if `Neighbors` is a list of elements `(Cost, NewState)` where `NewState` is a state *reachable* from `State` and `Cost` is the cost of doing so (and is constant $= 1$ in this problem). You must consider the following when implementing this predicate:

- For a given maze your successor state function should use the maze dimensions and its list of obstacle coordinates to correctly compute the successors of the given state (use the `maze/6` predicate to obtain this information).
- The list returned by `successors` must contain only the reachable states (i.e. the robot can neither move off the grid nor to a position that contains an obstacle) obtained by the moves *right, left, up, down* in that order. **It is extremely important for you to follow this order**.

  For example, in the search problem above from the state 1/1, the robot can move right to 2/1 or down to 1/2. In this case the list of successors would be `[(1, 2/1), (1, 1/2)]`, where each successor state is represented by a pair containing the cost of the move and the successor state (see the documentation of the search routines for more details).

State Equality Predicate You will need to write a state equality predicate that holds iff two states are considered the same state. *Hint: In defining this predicate, think about what aspects of the state representation need to be equal for your purposes.*

Heuristic Predicates To guide the search, you also need to specify a heuristic function. A trivial heuristic is to assign zero to all states (this is called the Null heuristic). This will cause A^* to perform as ordinary uniform cost search. We have provided you with its definition: `hfnUniform(_,0)`. We ask you to implement *three* other different heuristic functions:

Rounded Euclidean-Distance The distance from state x_1/y_1 to x_2/y_2 is calculated by the following formula: $\lfloor\sqrt{(x_1-x_2)^2+(y_1-y_2)^2}\rfloor$. In our example, the value of Rounded Euclidean-Distance for state S is 11 (i.e. $\lfloor\sqrt{(1-9)^2+(1-9)^2}\rfloor$). Note that it's been rounded down (you can use the `floor` function in Prolog for this purpose). Use the predicate `hfnEuclid(+State,-HVal)` to define this.

Manhattan-Distance This is the sum of the horizontal and vertical distances. In our example, S and G have Manhattan Distance 16. Use the predicate `hfnManhattan(+State,-HVal)` to define this.

Your own heuristic Here is a simple idea for one possible heuristic: a modified Manhattan distance that takes the dead-ends of the local neighborhood of the robot into account, where dead-end is a position with at least 3 obstacles around it. Use the predicate `hfnMyHeuristic(+State,-HVal)` to define this.

Further Information: Refer to the comments included in the provided search routines and the simple examples for more information. Note that there is no need to change any the given files except for the file containing the starter code `a1handin.pl` (you must add your implementation to this file).

Your Assignment

Part 1: Implementation

Implement the following predicates inside `a1handin.pl`*. Submit this file electronically. No paper copy needed.*

Important Note: Your code should work for any valid size maze.

Question 1. (18 marks) Successors Predicate
Implement the `successor` predicate as specified above.

Question 2. (2 marks) Equality Predicate
Implement the predicate `equality(+State1, +State2)` which holds if and only if `State1` and `State2` denote the same state.

Question 3. (30 marks) Heuristic Predicates
Implement the three heuristics as specified above (`hfnManhattan`, `hfnEuclid`, and `hfnMyHeuristic`).

To run your code simply call any of the given predicates `go/2, goCC/2, goIDA/2`. These predicates need the maze name and the name of a heuristic predicate. For instance, `go(maze1,hfnUniform)` will try to solve the first maze using A^* search together with the `hfnUniform` heuristic.

In the starter code we have provided you with three different sample mazes named `maze1, maze2`, and `maze3`. They are stored in mazesDatabase. This is where you should add any mazes you want to use for testing.

To choose which maze is used, call loadMazeInfo(+MazeName). This call is already included in go/2, goCC/2 and goIDA. It sets the maze/6 predicates (you don't need to worry about the details). After the call to loadMazeInfo, whenever you need to access the information about the maze being solved simply call maze(MazeName,M,N,O,S,G) with all variables uninstantiated. This will bind all these variables to appropriate values.

Also, to make your life easier, assume that the maze information is always correct, i.e. the dimensions are positive integers, the start, goal, and obstacle coordinates are all within the maze boundaries. Moreover, the start and goal positions do not contain an obstacle. You do NOT need to check for these conditions.

If you are curious, note that `astar`, `astarCC`, and `idastar` predicates all need the following inputs to generate the path to a solution: the starting state, the heuristic predicate, the `successors`, `goal`, and `equality` predicates. Since the definition of the *goal* predicate changes for each instance of the problem, it should be defined dynamically but we have already done this for you (its implementation is in helper predicate `prepareGoalPredicate`).

Part 2: Heuristics - properties and performance

Edit the file `a1answers.txt` *to answer the following questions. Submit it electronically. No paper copy needed.*

Question 4. (1 marks) Heuristics I
Explain in a few sentences how your heuristic works.

Question 5. (8 marks) Heuristics II
Which of the four heuristics (Null heuristic, Manhattan distance, Rounded Euclidean distance, and your own heuristic) are admissible? For any which is not admissible, give a counter example.

Question 6. (8 marks) Heuristics III
Suppose we modify the cost function so that moving to the left now costs 0.5, and all other moves still cost 1. Which of the heuristics are admissible now? For any which is not admissible give a counter example.

Question 7. (8 marks) Heuristics IV
For the original problem, imagine the robot would also be able to move diagonally (with the same cost 1). Which of the four heuristics will be admissible for this new problem? For any which is not admissible, give a counter example.

Question 8. (10 marks) Performance
Run your implementation on each of the 3 given mazes in the starter code (i.e. `maze1`, `maze2`, and `maze3`) and 3 provided search routines. In addition, for each of the 9 maze/search routine combinations use each of the 4 heuristics (i.e., totalling 36 runs). Use the given predicates `go/2`, `goCC/2`, `goIDA/2`. Note that not all problems may be solvable with all heuristics with the given node limit. (The limit is 10k nodes for `astar` and `astarCC`, and 30k nodes for `idastar`)

Fill in the table at the bottom of `a1answer.txt` that compares the numbers of nodes expanded by each of the provided search algorithms for each of the four heuristics. If one of the searches does not terminate within the given node limit, indicate it with "$> limit$."

Based on the results, draw your conclusions about the quality of the heuristics and the search routines (in less than 200 words).

Question 9. (1 marks) State Representation
Imagine a variant of the maze problem where the player initially has a a number of JumpOnObstacle pills. The player can move on top of an obstacle position if he still has at least one of these pills available (in effect this work as if there were no obstacle there). Note that moving on top of an obstacle will consume one pill. How would you minimally represent the states for this problem and what changes to the successors predicate will you need to make? (Do not write any code, briefly explain in English.)

Have fun and good luck!