CSC384 September 19, 2011
University of Toronto

Homework Assignment #1
Due: Friday, October 14, 2011, by 3 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: A 'grace days' system is in effect, see the course sheet for full information. After you have exhausted your grace days, no late assignment is accepted!

Total Marks: There are **100 marks** available in this assignment. This assignment represents **15%** of the course grade.

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 code and all answers 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 7–12 in `a1answers.txt`. Answer Question 6 in a separate pdf file or text file. Name it `a1question6.pdf` or `a1question6.txt`, respectively. Submit all three files. **Be sure to include your name and student number as a comment in the submitted code and all other files submitted.**

To submit these files electronically, use the CDF secure Web site:
`https://www.cdf.utoronto.ca/students`
or use the CDF **submit** command. Type **man submit** for more information.

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

The questions in this assignment may be (partially) automarked and also inspected manually. Note that we may automark your code using tasks (here: puzzle configurations) that are *different* from the ones given to you here.

Clarification Page:

Important corrections (hopefully few or none) and clarifications to the assignment will be posted on the Assignment 1 Clarification page, linked from the CSC384 Assignments web site. You are responsible for monitoring the A1 Clarification page.

Questions:

Questions concerning the assignment should be directed by email to the TA of Assignment 1, Alexandra Goultiava (t7goulti@cdf.toronto.edu). Please place "384" and "A1" in your email subject header.

1 Prolog

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'`, 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.

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').

2 Introduction

In this assignment you are going to implement a solver to the N-puzzle using three different search algorithms, A^*,A^* with cycle-checking, and IDA^*. We are providing you with the generic implementations of these algorithms in Prolog. Your task will be to formulate the N-puzzle as a search problem and to run experiments with these algorithms.

First, a bit of background. The N-puzzle is the simple (one-person) game we discussed briefly in class where tiles numbered 1 through N are moved on a square grid of $N+1$ cells (i.e. the grid is $\sqrt{N+1} \times \sqrt{N+1}$). Any tile adjacent to the blank position can be moved into the blank position. By moving tiles in sequence we attempt to reach the goal configuration. For example, in the figure below, we see three game configurations: the configuration (b) can be reached from configuration (a) by sliding tile 5 up; configuration (c) can be reached from configuration (b) by sliding tile 8 to the left. Configuration (c) is the goal configuration. The objective of the game is to reach the goal configuration from some starting configuration in as few moves as possible. The goal is, independent of the size of the grid, always defined as the ordering of all tiles enumerating from left to right and top to bottom, with the blank at the lower right corner.

Note that not all starting configurations can reach the goal.

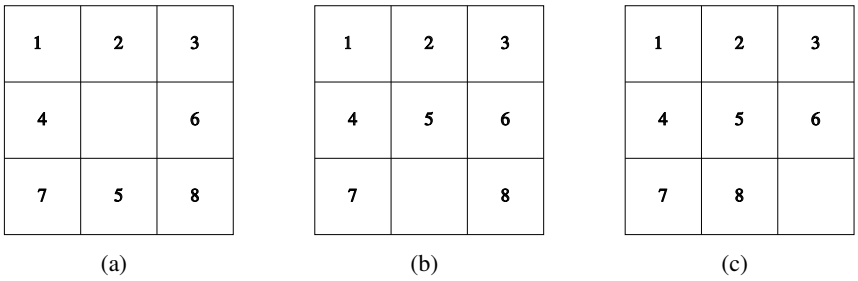

Figure 1: Three configurations of the 8-puzzle.

We provide you with the following three search algorithms implemented 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`.

These algorithms are generic: that is, they can be used to solve any problem that can be properly formalized. In order to apply them, you will need to provide the problem representation that these algorithms can use. This will include the representation of the initial state, the description of "steps" the algorithm can take, what it means to have reached the goal, when are the states considered equal, and the heuristics that the algorithm can use to estimate the distance to the goal.

Some simple examples of search spaces that show how these search routines are applied (`simpleSpace.pl`, and `waterjugs.pl`) are available.

3 The Questions

3.1 Implementation

Important Note: We require you to implement the following predicates so that they work for any valid size of puzzle (i.e. $M \times M$, $M > 1$).

The N-puzzle can be naturally considered as a grid $M \times M$ ($M = \sqrt{N+1}$) which can be implemented as a list of M lists of length M. The first list represents the first row of the puzzle, the second list the second row, etc. Use a placeholder 'b' for the blank field. For example, the puzzle represented in Figure 1(a) should be encoded as follows: `[[1,2,3], [4, b, 6], [7, 5, 8]]`

Question 1. (2 marks) Initial State Representation

Encode the initial state puzzle configurations shown in Figure 2. To do so, implement the corresponding 4 predicates in `a1handin.pl`, `init(Name, State)` where `Name` is the letter in the figure (i.e. `a`,`b`,`c`, or `d`) and `State` is the list representation of that configuration.

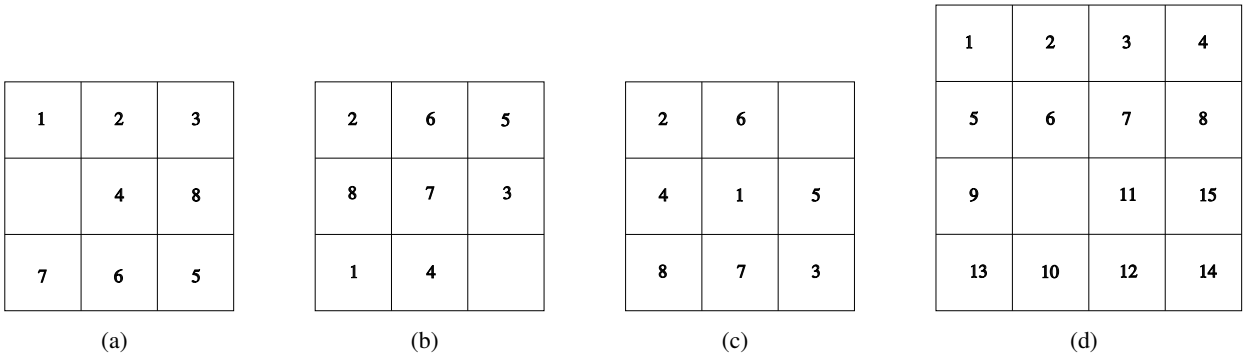

Figure 2: Four problems you will solve. Please use these names, i.e. `a`, `b`, `c`, and `d`.

Question 2. (6 marks) Goal Predicate

Implement the predicate `goal(+State)` that holds if and only if `State` is a goal state.

Question 3. (15 marks) Successors 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` by moving a tile down, left, right, or up (into the blank) and `Cost` is the cost of doing so (and is constant $= 1$ in this problem).

Question 4. (2 marks) Equality Predicate

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

Question 5. (26 marks) Heuristic Predicates

In `a1handin.pl` the null heuristic `hfn_null/2` (uniform-cost heuristic) is already given. In addition, implement the following two heuristics:

- `hfn_misplaced(+State, -V)` where `V` is the number of misplaced tiles (excluding the blank) in `State`, that is, the number of tiles which are not at the position where they should be according to the goal configuration.
- `hfn_manhattan(+State, -V)` where `V` is the sum over all the manhattan distances between the current and the designated position of every tile (except the blank). That is, instead of just counting the number of misplacements, we also take the 'distance' of each misplaced tile to its destination into account. This is more informative and should guide our search better. The manhattan distance between two positions is simply the sum of the absolute differences in x and in y direction.

Question 6. (8 marks) Testing

Run each of the 4 test cases using the two heuristics you implemented and the uniform-cost heuristic with the 3 search routines. You can do so by calling the predicates `runastar/2, runastarCC/2, runidastar/2`. For instance, `runastar(a, hfn_misplaced)` will try to solve the first problem using A^* search together with the heuristic made up from the number of misplaced tiles. Note that not all problems may be solvable with all heuristics with the expansion node limit as set in the code for `astar.pl`, `astarCC.pl`, and `idastar.pl`. Those are 10,000 for the astar searches and 20,000 for the idastar search.

Draw up a table that compares the number of nodes expanded by each of the provided search algorithms for all three heuristics, for all test cases. Use the format depicted in Table 1. If one search 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).

	*A**	*A*-CC*	*IDA**
Problem (a), hfn_null			
Problem (a), hfn_misplaced			
Problem (a), hfn_manhattan			
Problem (b), hfn_null			
Problem (b), hfn_misplaced			
Problem (b), hfn_manhattan			
Problem (c), hfn_null			
Problem (c), hfn_misplaced			
Problem (c), hfn_manhattan			
Problem (d), hfn_null			
Problem (d), hfn_misplaced			
Problem (d), hfn_manhattan			

Table 1: Results for all problems, search routines and heuristics.

3.2 Understanding

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

Question 7. (3 marks) Cost Function

What is the cost function for this problem? Which part of the problem description specified it? (Explain in less than 100 words)

Consider a forth heuristic defined in terms of *inversions*: For a puzzle configuration we say that a pair of tiles a and b are *inverted* if $a < b$ but the position of b is before a in the left-to-right,top-to-bottom ordering described through the goal state. For instance, in the configuration in Figure 2 (a) the pairs $(7,8)$, $(6,8)$, $(5,8)$, $(6,7)$, $(5,7)$, and $(5,6)$ are inverted. We define a new heuristic *hfn_inversions* as the number of inversions in a configuration. So for the said configuration in Figure 2 (a) hfn_inversions = 6.

Important: You do not need to implement this heuristic.

Question 8. (6 marks) Heuristics I

Which of the four heuristics are admissible? If it is not admissible, explain why. It suffices to give a counterexample.

Question 9. (6 marks) Heuristics II

Suppose we would change the cost for sliding a tile downwards to 2 and leave the cost of all the other movements at 1. Does this affect the admissibility of the four heuristics? Which of the heuristics are admissible now? For any which is not, why not?

Question 10. (12 marks) Heuristics III

Now suppose for sliding a tile to the left we would change the cost from 1 to 0.5 and leave all the other moves at cost 1.

a) Does this affect the admissibility of the heuristics? Again, which of them are admissible now? For any which is not, why not?

b) Pick one of the heuristics which are admissible in the original problem, but are no longer admissible. How would you modify it so that it becomes admissible?

Question 11. (4 marks) Performance

In the last modification (sliding to the left costs 0.5), can you say for sure which heuristic will be the fastest (expand the least number of states) in finding a (not necessary optimal) solution? Explain.

Question 12. (10 marks) State Representation

Imagine a variant of the problem where, in addition to sliding the tiles, you were allowed to swap any two tiles. The "swap" move would cost 1, just as any other move, but you are allowed to use it at most three times for each problem.

a) How would you minimally represent the states for this problem?

b) What changes to the successors predicate would you have to make? (Do not write any code for this question, just briefly explain in English)

Have fun and good luck!