CSCI–4150 Introduction to Artificial Intelligence, Fall 2005
Assignment 3 (142 points), out Thursday September 15, due Thursday September 29

Introduction

This assignment is on A* search and solving sliding block puzzles. You will implement the procedures for sliding block puzzles that allow them to be solved by a general search algorithm. The support code will include an implementation of the A* algorithm for you to use, but you will have to invent and implement heuristics.

Notes on this assignment

- There will be two web-testers for this assignment, one for Problems 1–6, and one for the electronic portion of Problem 8. The Problem 8 webtester will only run a few basic tests to make sure your heuristics work; the tests that are used to determine your score will be run offline. You will be able to upload your code for Problem 8 as many times as you like, but only the most recent file as of the late deadline will be scored.
- Problem 7 and the written portion of Problem 8 must be turned in on paper. Any late work should be turned in to the CS main office (Lally 207) or to my secretary (Shannon, in Amos Eaton 132). Make sure you give it to someone who will record the date and time you turn it in.
- There will be three files for this assignment on the assignment information page:
  - `assign3.scm` — a file of stubs for the procedures you must write.
  - `a3code.com` — support code in compiled format
  - `a3puzzles.scm` — starting and goal states for a number of puzzles
- You will need correct implementations of the procedures `valid-right?`, `block-distance`, and `empty:right-neighbors` from Assignment 2. If you did not get these to work for Assignment 2 and need assistance in finishing them, please see the instructor or TA.
- For this assignment, you will have to use `map` and `apply`, but mutators and iterative forms (i.e. `do`) are still prohibited.

A. Generating sliding block puzzle child states

In order to formulate a problem for a general search algorithm, we need three things: a start state, a goal test procedure, and a "child states" procedure. (Usually, we generate "child *nodes*," but not this time.) In this part, you will write a number of procedures leading up to the `sbp-child-states` procedure.

A.1 Terminology & representations

The following terminology and representations are used in the rest of the assignment:

- A *sliding block puzzle* has the same representation as in Assignment 2. See the Assignment 2 handout for a full description.
- A *move* will translate one block by one grid cell. It is represented as a list of length 2 where the first element is the name of the block (a single-character symbol) and the second element is the direction (which must be one of the symbols: `left`, `right`, `up`, and `down`). For example: `(B left)`.

A.2 Support code

There are a number of procedures in the support code that you will find useful for these problems:

- `(all-blocks sbp)`

  Given a sliding block puzzle state, it will return a list of all the blocks in the puzzle. For example:

  ```
  > (all-blocks example-a)
  ;Value: (a b c d e f g h j)
  ```

  This procedure will also work with "goal templates" (which are explained later in the assignment).

- `(valid-move?  sbp move)`

  This procedure consists of a `case` statement that calls of the procedures: `valid-right?`, `valid-left?`, `valid-up?`, and `valid-down?`. It depends on your implementation of these procedures.

- `(sbp-move sbp move)`

  Given a sliding block puzzle state and a move, it returns the resulting sliding block puzzle state. This procedure assumes that the move is valid; if not, it may return an invalid state or cause an error.

- `(print-sbp sbp)`

  The same procedure from the Assignment 2 support code that prints an ASCII picture of a sliding block puzzle state to the screen.

There is a parameter `error-checking` whose default value is `#t`. This causes several of the procedures above to check their arguments and signal an error if they are invalid. This is useful for debugging, but if you want the code to run faster re-`define` it to be `#f`.

A.3 Problems

1. (8 points) Write a nonrecursive procedure `(transpose M)` using `map` and `apply` that returns the transpose of the "matrix" `M` — a list where each element represents a row, and each row is represented by a list with one element per column. All rows must, of course, have the same length. For example:

   ```
   > (transpose '((a b c)
                  (d e f)))
   ;Value: ((a d)
            (b e)
            (c f))
   ```

2. (9 points) Write the procedures:

   - `(valid-left? sbp block)`
   - `(valid-up? sbp block)`
   - `(valid-down? sbp block)`

   which are analogous to the `valid-right?` procedure you wrote for Assignment 2.

   Instead of writing each of these from scratch, you should use your `transpose` procedure from Problem 1 and the `reverse` procedure to transform the problem so that you can use your `valid-right?` procedure. The procedures for this problem need only be two lines each.

3. (9 points) Write the procedures:

   - `(empty:left-neighbors sbp)`
   - `(empty:up-neighbors sbp)`
   - `(empty:down-neighbors sbp)`

   which should be analogous to the `empty:right-neighbors` procedure you wrote for Assignment 2.

   As in previous problem, you should use your `transpose` procedure and the `reverse` procedure to transform the problem so that you can use `empty:right-neighbors` instead of writing these from scratch. The procedures for this problem need only be two lines each.

4. (14 points) Write the procedure `(sbp-child-moves sbp)` which returns a list of all possible moves from the given sliding block puzzle state.

   I suggest the following approach to this problem:

   - Generate a list of possible moves using the `empty:right-neighbors` procedures and its compatriots from Problem 3. Recall that `empty:right-neighbors` returns a list of blocks that have at least one empty cell on their left side; this does not mean that the entire block can be moved to the left.
   - Remove the moves from this list that are not valid (by checking with the `valid-right?` procedure and its compatriots from Problem 2).

   The order of the moves in the resulting list is not important.

5. (6 points) Write the procedure `(sbp-child-states sbp)` which returns a list of all possible states that can be reached by a single move from the given sliding block puzzle state. I suggest using your `sbp-child-moves` procedure from the previous problem. The order of the states in the resulting list is not important.

B. Solving sliding block puzzles

The remaining things needed to solve sliding block puzzles are a way of testing for the goal state, the A* search algorithm, and heuristics. The support code will provide most of the first two; you will have to provide the last.

B.1 Support code

The following procedures are discussed in detail in the following sections:

- `(sbp-compare sbp goal-template)`

  Returns `#t` if the sliding block puzzle state satisfies the goal template.

- `(solve sbp at-goal? heuristic)`

  Runs an A* search starting from the given sliding block puzzle state `sbp`, using the procedure `at-goal?` to determine if the goal has been reached, and the procedure `heuristic` as the heuristic to guide the search.

You should also look through the `a3puzzles.scm` file.

B.2 Goal templates

For some of the sliding block puzzles we will solve, the goal state only requires a few of the blocks to be in specific locations; the remaining blocks may be anywhere. To recognize a goal state in a general way, we will use a "goal template."

For example, here is the "Hughes puzzle" starting state:

Q		A
Q		B
C	D	D
E	E	F

The goal for this puzzle states that Block Q must be in the lower right hand corner. The goal template for this goal is then:

```
(define hughes-goal '((#f #f #f)
                      (#f #f #f)
                      (#f #f  Q)
                      (#f #f  Q)))
```

The cells that require a certain block must contain the name of that block. The cells that don't matter are represented by the value `#f`. Recall that we assume each block in the puzzle has a unique name.

The `sbp-compare` procedure returns `#t` if the given sliding block puzzle state satisfies the goal template. For example, here is the `hughes-goal?` procedure:

```
(define (hughes-goal? s)
  (sbp-compare s hughes-goal))
```

Incidentally, this naming convention (a goal template ends with "`-goal`" and a goal testing procedure ends with "`-goal?`") is used throughout the `a3puzzles.scm` file.

B.3 The A* algorithm

The `(solve sbp at-goal? heuristic)` procedure is a reasonably efficient implementation of the A* algorithm. Its arguments are:

- `sbp` — a sliding block puzzle state
- `at-goal?` — a procedure of the form `(at-goal? sbp)` which returns `#t` if the given state is a goal state
- `heuristic` — a procedure of the form `(heuristic sbp)` which returns a nonnegative number used as the heuristic in the A* search.

The goal template (or state) must be implicit in the `at-goal?` and `heuristic` procedures.

Dependency The `solve` procedure is "hardcoded" to use your `sbp-child-states` procedure. (It is also customized in some other ways to this problem.)

Parameters There are several parameters that control what output the `solve` procedure prints:

- `progress-messages` — default value is `#t`

  Prints a message every 100 nodes so that you can see how the search is progressing.

- `print-boards` — default value is `#t`

  After finding a solution, prints the sequence of boards from the start to a goal.

- `print-moves` — default value is `#f`

  After finding a solution, prints the sequence of moves from the start to a goal.

- `print-states` — default value is `#f`

  After finding a solution, prints the sequence of states (in Scheme form) from the start to a goal.

You can change the values of these variables by re-`define`-ing them. They are all independent — you can set them all to `#t`, all to `#f`, or to any combination.

Memory For this assignment you will need to run Scheme with more memory than it usually runs with.

- If you run Scheme through gnu-emacs, make sure the following lines are in your `.emacs` file:

```
(load-library "xscheme")
(setq scheme-program-arguments "-heap 4000")
```

- If you run Scheme or Edwin from a UNIX shell, just add "`-heap 4000`" (without quotes) to your command line.

- Under Windows, you need to edit the shortcut or properties for Scheme or Edwin to add the switch "`-heap 4000`" (without quotes) to the actual command.

This sets the size of the heap to be 4000 blocks of 1024 words each; this should be more than enough for this assignment. The default heap size is around 200 or 400 blocks.

B.4 Problems

6. (10 points) Write the procedure `(manhattan-dist sbp goal-template)` which returns the sum over all blocks in the `goal-template` of the Manhattan distance between that block's position in the `goal-template` and its position in the given sliding block puzzle state `sbp`.

   You should use your `block-distance` procedure from Assignment 2 and the `all-blocks` procedures from the support code.

7. (6 points; written) Solve the following puzzles using the Manhattan distance heuristic and the Manhattan distance squared heuristic. Note that there are predefined procedures in `a3puzzles.scm` that call your `manhattan-dist` procedure, as noted below.

   - the "w2" puzzle, using `(w2)` and `(w2^2)`
   - Hughes puzzle, using `(hughes)` and `(hughes^2)`
   - the "y3" puzzle, using `(y3)` and `(y3^2)`
   - the "y5" puzzle, using `(y5)` and `(y5^2)`
   - Ma's puzzle, using `(mas)` and `(mas^2)`

   You may abort any test that takes more than 3 minutes.

   Turn in a (written) table of your results, indicating: the length of the solution, the number of states examined, and the running time.

8. (60 points written; 20 points electronic) For this problem, you will design and implement two heuristics for solving sliding block puzzles where the position of all the blocks need not be specified in the goal template.

   Your first heuristic, `(a-heuristic sbp goal-template)`, must be an *admissible* heuristic; the second, `(i-heuristic sbp goal-template)`, must be an *inadmissible* heuristic.

   You should strive to design heuristics that are better than the Manhattan-distance heuristic (and the squared Manhattan-distance heuristic), not just tinker with those heuristics. You should think carefully about the problem and your solutions.

   The written component of this problem should cover the following:

   (a) a clear (prose) explanation of each heuristic

   (b) Prove that your `a-heuristic` is admissible and that your `i-heuristic` is inadmissible. You need not give a "formal" proof, but it should be both convincing and concise.

   (c) test your heuristic on the following puzzles:

   - the "w2" puzzle, using `(a-w2)` and `(i-w2)`
   - the Hughes puzzle, using `(a-hughes)` and `(i-hughes)`
   - the "y3" puzzle, using `(a-y3)` and `(i-y3)`
   - the "y5" puzzle, using `(a-y5)` and `(i-y5)`
   - Ma's puzzle, using `(a-mas)` and `(i-mas)`

   You may abort any test that takes longer than 3 minutes.

   Turn in a table of the results showing: the length of the solutions, the numbers of states examined, and the running times.

   (d) evaluate the performance of your heuristics in comparison to each other and to the Manhattan-distance (and Manhattan-distance squared) heuristics using your results from Problem 7.

   The written part for this problem probably only needs to be 2–3 pages; it may not exceed 4 (single sided) pages. You may include descriptions of other tests and heuristics you tried during the development of your two heuristics, subject to the length constraint.

   The electronic part of this problem will be based on offline tests of your heuristic functions.