CSCI 4150 Introduction to Artificial Intelligence, Fall 2002
Assignment 3 (146 points): out Friday September 13, due Friday September 27

A. 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 implement and invent heuristics.

Notes on this assignment

- Problems 1–6 and 8 are to be turned in electronically through the web tester; problems 7 and 9 are written and must be turned in on paper. Problem 10 has both an electronic and written component.
- 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
- For this assignment, you will have to use `map` and `apply`, but mutators and iterative forms ("named lets" and `do`) are still prohibited.
- There is one "bonus" problem on this assignment which is marked by a "⋆".

B. A few useful procedures

The procedures you will write in this part will be used in other parts of the assignment.

Problems

1. (10 points) Write the 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. For example:

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

2. (10 points) Write the procedure `(pair-combinations A B)` where `A` and `B` are both lists. It should return a list containing all possible "pairs" of an element from list `A` and an element from list `B`. Each "pair" should be a list where the first element is an element from list `A` and the second from list `B`. For example:

```
(pair-combinations '(a b c) '(1 2))
;Value: ((a 1) (a 2) (b 1) (b 2) (c 1) (c 2))
```

The order of the elements in the return value is not important.

C. 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 procedure to test whether a given state is a goal, and a procedure to generate the child states from a given state. (Usually, we do this in terms of nodes instead of states, but not this time.)

In this part, you will write a number of procedures leading up to the `sbp-child-states` procedure. 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 block dir)`

  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 block dir)`

  Given a sliding block puzzle state `sbp`, a `block` (which must be a symbol), and a direction `dir` (which must be one of the symbols: `left`, `right`, `up`, and `down`), this procedure returns the resulting state. It assumes that the move is valid; it may otherwise return an invalid state or cause an error.

- `(block-equal?  x y)`

  Returns `#t` if `x` and `y` represent the same block. They might both be symbols, one might be a list containing the symbol for the block, or they both might be lists. For example:

  ```
  > (block-equal '(a) 'b)
  ;Value: #f
  > (block-equal 'a '(a))
  ;Value: #t
  > (block-equal '(q) '(q))
  ;Value: #t
  ```

- `(print-sbp sbp)`

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

Problems

3. (18 points) In the first assignment, you wrote the procedure `valid-right?` which tested whether a block could be moved right in the given sliding block puzzle state. I will provide an implementation of this procedure in the support code, so that everyone will have a correct implementation to work with. (Therefore, you should not include your `valid-right?` code in your Assignment 3 code.)

   For this problem, write the procedures:

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

   Use the `transpose` procedure (from Problem 1) and the `reverse` procedure in conjunction with the `valid-right?` procedure instead of writing these from scratch. The procedures for this problem need only be a few lines each.

4. (24 points) Write the procedure `(sbp-child-states sbp)` which returns a list of sliding block puzzle states.

   The simple approach to this problem is:

   - generate a list of all possible moves (i.e. combination of block and move direction) using the `pair-combinations` procedure from Problem 2)
   - remove the moves from this list which are not valid
   - then use `map` on this list with a procedure that takes a "move" and returns the state after making that move.

   However, you may implement this procedure in any way you see fit. See the information earlier in this section for support code procedures that will be useful for this problem.

D. Solving sliding block puzzles

The support code will contain a reasonably efficient implementation of the A* algorithm and all the other code needed to solve sliding block puzzles except for a heuristic function. This section details the support code, and the problems will ask you to implement several heuristic functions.

Checking for goal states

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. In order to check for a goal state in a general way, the support code provides a procedure that returns a "goal-checking procedure," given a "template" for the goal state.

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 "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 exact block representation (i.e. the symbol or the list containing the symbol are different). The cells that don't matter are represented by the value `#f`.

The `sbp-compare` procedure compares a state with a goal "template." For example, here is the `hughes-goal?` procedure:

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

The A* algorithm

The A* algorithm in the support code will be "hardcoded" to use your `sbp-child-states` procedure. (It is also customized in some other ways to this problem.)

To run the A* search on a sliding block puzzle, use the procedure:

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

where `sbp` is the starting state, `at-goal?` is a predicate that takes a sliding block puzzle state and returns `#t` if that state is a goal state, and `heuristic` is a procedure of the form: `(heuristic sbp)`. The goal state (or goal template) must be implicit in the `at-goal?` and `heuristic` procedures.

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 not mutually exclusive — you can set them all to `#t` or all to `#f`.

Other support code

- `(block-distance sbp-a sbp-b block)`

  An implementation of the procedure you wrote for Assignment 2.

- Also be sure to look through the `a3puzzles.scm` file.

Problems

6. (18 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 take advantage of the `block-distance` procedure implementation provided in the support code.

7. (18 points) The Assignment 3 information page will detail a number of puzzles for you to run using the Manhattan-distance and the square of the Manhattan-distance as a heuristic. Turn in a (written) table of your results, indicating the number of states examined, the running time, and whether the search produced an optimal solution.

8. (18 points) Write a procedure `(sbp-heuristic sbp)` that implements a heuristic of your own design for the sliding block puzzles such as the "Hughes puzzle" which only specify the location of a few blocks in the goal state. You should aim for something that does better than the Manhattan-distance heuristic (or the squared Manhattan-distance).

9. (30 points) Turn in the following written information about your `sbp-heuristic` procedure:

   - a clear (prose) description of your heuristic
   - whether it is admissible and monotonic or not and justify your answer
   - results from running your heuristic on tests as described on the Assignment 3 information page: number of states evaluated, running time, and whether the solution is optimal or not.

⋆10. (10 points) There will be a number of challenge puzzles on the Assignment 3 information page. Write a heuristic function that can solve one or more of these puzzles and turn in written information as for the previous problem.