

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

- Problem 10 is a “bonus” problem. Bonus problems will be marked by a “*” on this and future assignments.
- 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.

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 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.

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. Recall that a sliding block puzzle may be on a rectangular grid of any size. The blocks are either rectangular or L-shaped and may be any size or orientation.
- 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 symbol) and the second element is the direction (which must be one of the symbols: `left`, `right`, `up`, and `down`).

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.
- `(valid-right? sbp block)`
An implementation of the procedure you wrote in Assignment 2.
- `(empty:right-neighbors sbp)`
An implementation of the procedure you wrote in Assignment 2.

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 `redefine` 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) In Assignment 1, you wrote the procedure `valid-right?` which tested whether a block could be moved right in the given sliding block puzzle state. An implementation of this procedure is in the support code to decouple the assignments.

For this problem, write the procedures:

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

Use your `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 two lines each.

3. (9 points) In Assignment 1, you wrote the procedure `empty:right-neighbors` which returned a list of blocks on the immediate right of an empty cell in any row of the given sliding block puzzle state. An implementation of this procedure is in the support code to decouple the assignments.

For this problem, write the procedures:

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

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

4. (16 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.
- Remove the moves from this list that are not valid.

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.

The support code also contains:

- `(block-distance sbp-a sbp-b block)`
An implementation of the procedure you wrote for Assignment 2.

In addition, you should 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 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 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-defining 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.

B.4 Problems

- (12 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` and `all-blocks` procedures in the support code.
- (12 points; written) Run your Manhattan-distance heuristic on the following puzzles:
 - 8-puzzle #5, using `(ep5)` and `(ep5^2)`
 - 8-puzzle #6, using `(ep6)` and `(ep6^2)`
 - 15-puzzle #2, using `(fp2)` and `(fp2^2)`
 - Hughes puzzle, using `(hughes)` and `(hughes^2)`

You can 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.

- (48 points; written and electronic) For this problem, you will design and implement two heuristics. The first, `(c-heuristic sbp goal-template)`, should be a *consistent* heuristic; the second, `(i-heuristic sbp goal-template)`, should be an *inconsistent* (or inadmissible) heuristic. You should strive to design heuristics that do better than the Manhattan-distance heuristic (and the squared Manhattan-distance), but at the very least, you should demonstrate that you have thought carefully about the problem and your solutions.

You will turn in your code electronically to a separate web tester. This web tester will only do a syntax check and pretest on your code; we will be running tests offline.

The written component of this problem should cover the following:

- (a) a clear (prose) description of each heuristic
- (b) justify the consistency of `c`-heuristic and the inconsistency of `i`-heuristic
- (c) test your heuristic on the following puzzles:
 - 15-puzzle #2, using `(c-fp2)` and `(i-fp2)`
 - Hughes puzzle, using `(c-hughes)` and `(i-hughes)`
 - the "w1" puzzle, using `(c-w1)` and `(i-w1)`
 - the "w2" puzzle, using `(c-w2)` and `(i-w2)`
 - Ma's puzzle, using `(c-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.

The written part for this problem probably only needs to be 1–2 pages; it may not exceed 3 (single sided) pages. You may include descriptions of other tests and heuristics you ran during the development of your two heuristics, subject to the length constraint.

- *9. (10 points bonus; electronic and written) Write a heuristic functions for one or more of the following puzzles

- The Ten Block puzzle — `(ten-block-heuristic s)`
- Any of the "p" puzzles — `(p1-heuristic s), ... (p6-heuristic s)`
- The Century Puzzle — `(century-heuristic s)`
- Either of the Piano Moving puzzles — `(piano-moving-1-heuristic s)` and `(piano-moving-2-heuristic s)`

Note that these heuristics only take one argument, a sliding block puzzle state, so each will be customized for its respective problem. Your heuristic function should be able to solve the puzzle in 10 minutes or less of CPU-time and not run out of memory with the `"-heap 4000"` switch.

Turn in a written (prose) description of your heuristic, and the results for solving the respective problem (i.e. path length, number of states evaluated, and run time).