

## Project: Part-2 — Revised Edition

Due 9:30am (sections 10, 11) 11:001m (sections 12, 13) Monday, May 16, 2005  
150 points

Part-2 of the project consists of both a high-level heuristic game-playing program and the underlying implementation of data types. It must be done independently; joint work with another person is **NOT** allowed.

### The Eight Puzzle

The Eight Puzzle, a popular sliding block puzzle, consists of eight movable square tiles and one empty square on a  $3 \times 3$  grid. A move consists of exchanging the blank tile with an adjacent tile; we view this as moving the blank tile left, right, up, or down. The object of the game is to find a sequence of moves that transforms an initial arrangement of tiles into some goal configuration. The bottom of this page illustrates an initial puzzle configuration and a goal puzzle configuration for the Eight Puzzle, and a sequence of moves comprising a solution. (There is more than one way to solve the puzzle.)

Your overall project consists of several parts, resulting in a program that uses a simple Artificial Intelligence technique to guide the search for a sequence of moves that will produce a given goal puzzle configuration from a given initial puzzle configuration. This project will also illustrate several layers of data abstraction.

Each of your procedures must be well-documented. This should include a specification of each argument to the procedure and a specification of the returned result.

## Part I

At the top level, you will work with partial solutions (referred to as *states*) and state sets (referred to as *State-Sets*), where a partial solution or *state* represents the problem after making a given sequence of moves from a given initial puzzle configuration and a *State-Set* represents a set of different problem states. Assume that you have the following constructors, selectors, and predicates for operating on States and State-Sets:

`(Combine-Disjoint-State-Sets state-set-1 state-set-2)`

Takes two state-sets as arguments and returns a state-set containing the states in the argument state-sets.

`(Remove-From-State-Set state state-set)`

Takes a state and a state-set as arguments and returns a state-set containing all the states in state-set except for the state passed as an argument.

`(Select-Best-State state-set)`

Takes a nonempty State-Set as argument and returns the state with the best rating (ie. the state that should be extended to try and reach the goal puzzle configuration)

`(Empty-State-Set? state-set)`

Takes a State-Set as argument and returns true if it contains no states.

`(Expand-State state)`

Takes a state as argument and returns a state-set containing a set of states, each constructed by making one additional move from the argument state.

`(Is-Solution? state)`

Takes a state as argument and returns true if the state represents one that has reached the goal.

`(Get-Solution state)`

Takes a state as argument and returns an ordered list of moves that solve the puzzle, ordered from first move to last move.

Using these primitives, and assuming **NOTHING** about how states and State-Sets are represented, design a procedure (`Eight-Puzzle initial-state-set max`) that takes as arguments 1) an initial state set containing a single state representing the initial state as one begins to solve the puzzle and 2) the maximum number of repetitions of procedure `Eight-Puzzle`, and returns a list of the moves that solve the problem.

I suggest you proceed as follows:

As long as the maximum number of repetitions of `Eight-Puzzle` is not exceeded and the best state in the current State-Set does not represent a solution, expand the best state so far (using procedure `Expand-State`) and form a new State-Set constructed from the states in the State-Set produced by procedure `Expand-State` and the states in the original State-Set after removing the state that was expanded. When a solution state is found, return an ordered list of the moves that solve the puzzle, as produced by `Get-Solution`.

To run your program, load file `~carberry/Project-2-init.scm` which contains the procedure `Make-Initial-State-Set` and execute

```
(Eight-Puzzle (Make-Initial-State-Set start-list goal-list) max)
```

For `start-list` and `goal-list`, use a list of 9 elements giving the numbers on the tiles from left to right in the top row, then the middle row, and then the bottom row, with `Blank` given for the blank tile. For example

```
(Eight-Puzzle (Make-Initial-State-Set
               '(4 2 6 Blank 1 3 7 8 5)
               '(2 6 Blank 4 8 3 1 7 5))
 30)
```

(Of course, your program will not run until you have implemented the constructors and selectors that it uses (See Part II).

## Part II

Now you must design the procedures for the constructors, selectors, and predicates used to operate on states and state-sets in Part I. Use the following representation for states and state-sets:

- A state will be an ordered list of four elements:
  1. The current puzzle configuration.
  2. a move-set of moves so far
  3. The rating of the state (a non-negative integer, where the rating is the sum of the rating of the move-set and 3 times the rating of the current puzzle configuration).
  4. The goal puzzle configuration.
- A state-set will be an unordered list of states.

In order to design the constructors, selectors, and predicates for states and state-sets, you need to be able to operate on moves, move-sets and puzzle-configurations. Assume that you have the following constructors, selectors, and predicates for operating on moves, move-sets and puzzle-configurations.

**(Can-Move? puzzle-configuration direction)**

Takes a puzzle configuration and a direction (Right, Left, Up, or Down) as arguments and returns true if the blank tile can be moved in that direction

**(Get-New-Puzzle-Configuration puzzle-configuration direction)**

Takes a puzzle-configuration and a direction (Right, Left, Up, or Down) and returns the new puzzle configuration that would result if the blank tile is moved in that direction.

**(Get-Adjacent-Tile tile1 puzzle-configuration direction)**

Takes a tile, a puzzle-configuration, and a direction (Right, Left, Up, or Down) and returns the tile adjacent to tile1 in the specified direction.

**(Swap-Tiles puzzle-configuration tile-1 tile-2)**

Takes a puzzle-configuration and two tiles as arguments and returns a puzzle configuration in which tile-1 and tile-2 are swapped.

**(Get-Move-Directions puzzle-configuration)**

Takes a puzzle configuration as argument and returns a list of the directions in which the blank tile could be moved, where the directions are Right, Left, Up, or Down

**(Create-Move puzzle-configuration direction)**

Takes a puzzle configuration and a direction (Right, Left, Up, or Down) as arguments, and returns a move in that direction

**(Add-To-Move-Set move move-set)**

Takes a move and a move-set as arguments, and returns an expanded move-set that includes the new move.

**(Get-Moves-From-Start-To-Finish move-set)**

Takes a move-set as argument and returns an ordered list of moves from first move to last move.

**(Eval-Move-Set move-set)**

Takes a move-set as argument and returns its rating as a non-negative

number, where lower ratings are better.

(Eval-Puzzle-Configuration puzzle-configuration goal-configuration)  
 Takes two puzzle configurations as arguments and returns a non-negative number that rates how hard it will be to get from the first puzzle configuration to the second one, where lower ratings are better.

(Get-Tile-In-Position row column puzzle-configuration)  
 Takes a row, a column, and a puzzle-configuration as arguments and returns the tile in the specified row and column of the puzzle.

(Get-Tile-With-Value value puzzle-configuration)  
 Takes a value and a puzzle configuration as arguments and returns the tile in puzzle-configuration with the specified face value.

(Get-Blank-Tile puzzle-configuration)  
 Takes a puzzle configuration as argument and returns the blank tile.

(Blank-At-Left? puzzle-configuration)  
 Takes a puzzle configuration as argument and returns true if the blank tile is in the leftmost column

(Blank-At-Right? puzzle-configuration)  
 Takes a puzzle configuration as argument and returns true if the blank tile is in the rightmost column

(Blank-At-Top? puzzle-configuration)  
 Takes a puzzle configuration as argument and returns true if the blank tile is in the top row

(Blank-At-Bottom? puzzle-configuration)  
 Takes a puzzle configuration as argument and returns true if the blank tile is in the bottom row

Using these constructors and selectors for operating on moves, move-sets and puzzle-configurations, and assuming **NOTHING** about how moves, move-sets and puzzle-configurations are represented, design the constructors and selectors given in Part I for operating on states and State-Sets.

### Part III

Now you must design the procedures for the constructors, selectors, and predicates used to operate on moves, move-sets and puzzle-configurations in Part II. Use the following representation for a move-set:

A move will be represented as R, L, U, or D telling whether to move the blank tile right, left, up, or down respectively.

A move-set will be represented as an ordered list of moves, ordered from most recent move to oldest move.

A puzzle-configuration will be represented as an unordered list of nine tiles, where one tile is the blank tile.

The rating of a move-set will be the number of moves in it.

The rating of a puzzle-configuration will be the sum of the distance that each tile in the current puzzle configuration (except for the blank tile) is from its position in the goal puzzle configuration.

Assume that you have the following constructors and selectors for operating on tiles.

`(Is-Blank-Tile? tile)`

Takes a tile as argument and returns true if it is the blank tile

`(Create-Tile row column value)`

Takes a row (1, 2, or 3), a column (1, 2, or 3), and a value and returns a tile in the specified location.

`(Get-Row-Of-Tile tile)`

Takes a tile as argument and returns the row in which it is located

`(Get-Column-Of-Tile tile)`

Takes a tile as argument and returns the column in which it is located

`(Get-Value-Of-Tile tile)`

Takes a tile as argument and returns its value (ie., the number on its face).

`(Get-Distance-Between-Tiles tile1 tile2)`

Takes two tiles as arguments and returns the sum of the row and column distances between them.

Using these constructors and selectors for operating on tiles in puzzle configurations, and assuming **NOTHING** about how tiles are represented, design the constructors, selectors, and predicates given in Part II for operating on moves, move-sets and puzzle configurations.

## Part IV

Now you must design the procedures for the constructors and selectors used in Part III to operate on tiles.

A tile will be represented as an ordered list of three items: the tile's row location (1, 2, or 3), its column location (1, 2, or 3), and the face value of the tile (with 'Blank' for the blank tile). Thus the tile in the upper left corner of the goal puzzle configuration on the first page would be represented as (1 1 4).

Design the constructors, selectors, and predicates for operating on tiles.

Test the various parts of your program. Your code should be submitted as follows — (do not include my code for `Make-Initial-State-Set` in your submission):

1. Your code for all of the procedures should be submitted to `Project-2a`.
2. Your top-level procedure `Eight-Puzzle` should be submitted to `Project-2b`
3. Your procedures listed under Part I should be submitted to `Project-2c`
4. Your procedures listed under Part II should be submitted to `Project-2d`
5. Your procedures listed under Part III should be submitted to `Project-2e`
6. Your procedures listed under Part II should be submitted again to `Project-2f`

Each of the submissions tests your code for several test cases, two of which are hidden from view. If you pass the second hidden test case on a submission, then you have passed that submission. But do not submit a final version (ie., do not click on `Confirm`) until you have everything working, since the code you submit for each independent part must be the same code submitted for `Project-2a`.