Eight Puzzle

The puzzle consists of eight sliding tiles, numbered by digits from 1 to 8, and arranged in a 3 by 3 array of nine cells. One of the cells is always empty, and any adjacent tile can be moved into the empty cell. We can say that the empty cell is allowed to move around, swapping its place with any of the adjacent tiles.

The initial state is some arbitrary arrangement of the tiles. An initial state and the goal state
are shown next.

an initial state

1	2	3
8		5
7	4	6

goal state

1	2	3
8		4
7	6	5

For the initial state above, there are four possible moves. Tile 2, 8, 5 or 4 can be moved into the empty cell, e.g. one state transitions is shown by:

current state

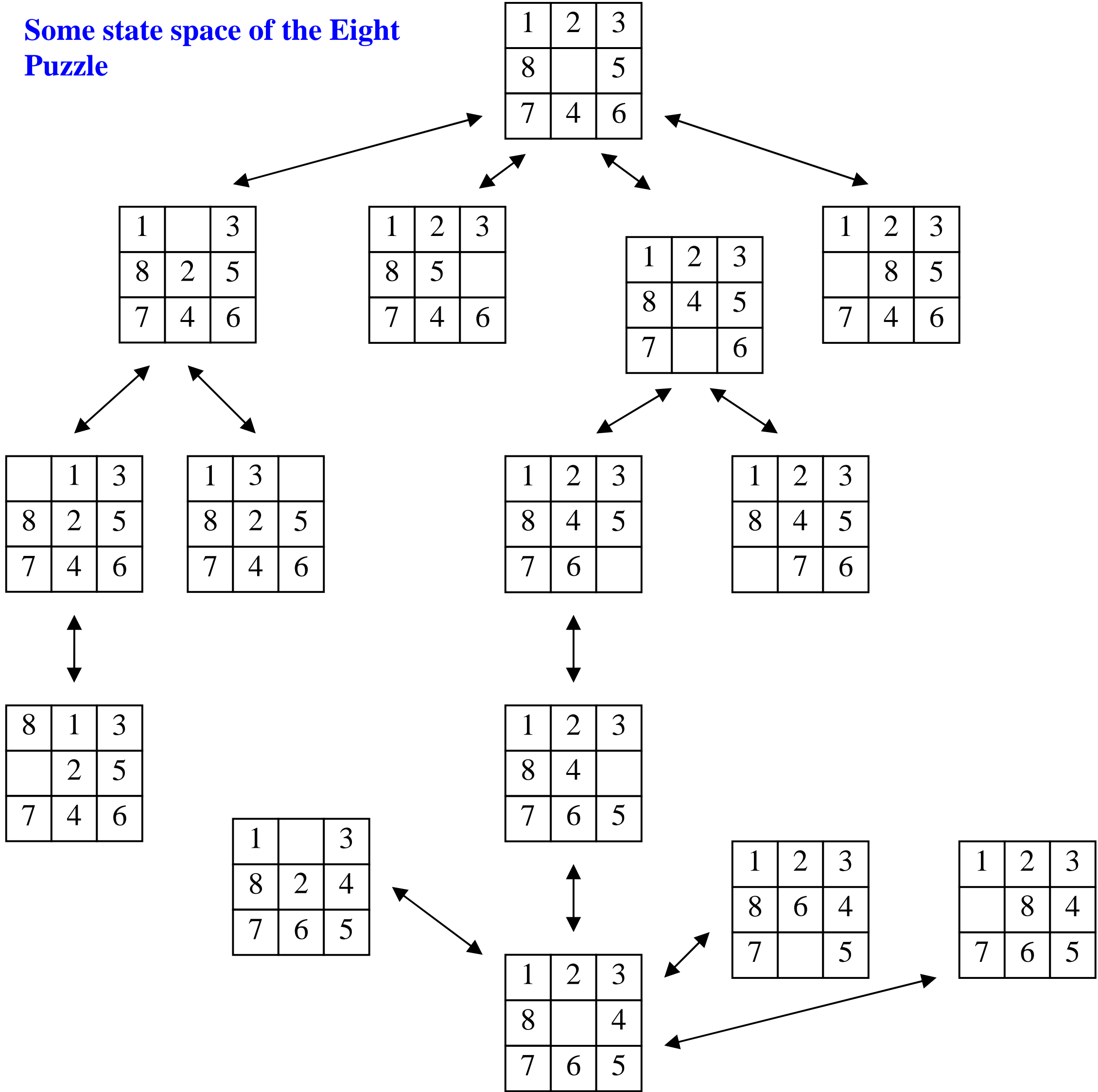
1	2	3
8		5
7	4	6

a successor or child state

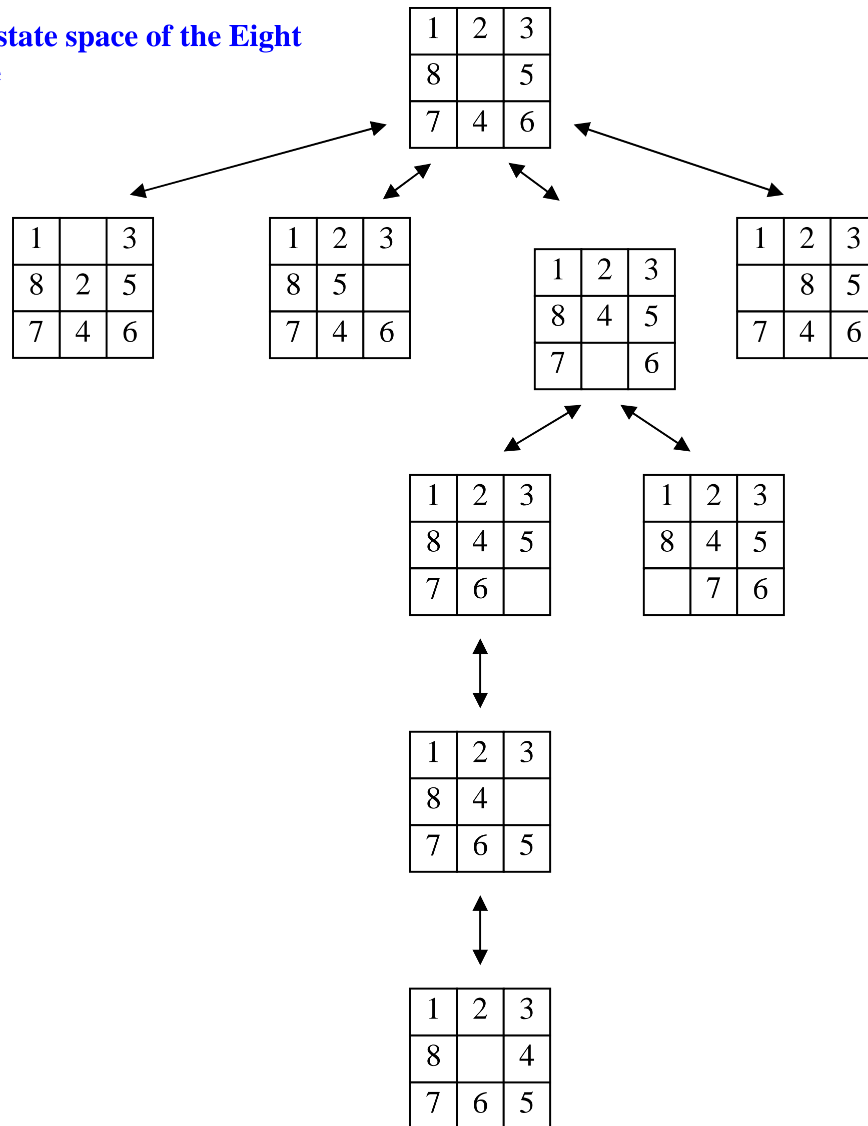
1	2	3
8	4	5
7		6

A move or state transition is only allowed if the ***Manhattan*** distance between the current and new states is 1. The problem is to find a sequence of moves from the initial state to the goal state.

Some state space of the Eight Puzzle



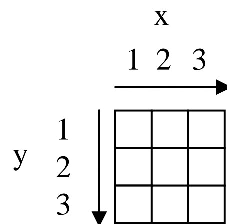
Some state space of the Eight Puzzle



Representing and Solving the 8-Puzzle in Haskell

Note that most of the Haskell program code below is outside the scope of this course. It is only included to show the power of Haskell for this type of programming and to show how a state might be represented, and of course for your interest.

First we need a way to represent and store a state. We can think of the tiles as being numbered from 1 to 8. One possibility is to assign each tile an (x, y) pair of co-ordinates.



So in the following state

1	5	3
8		2
7	4	6

tile1 is at (1, 1), tile2 at (3, 2), tile3 at (3, 1). Empty position or tile0 if you like is at (2,2).

We wish to store all the tile positions together and a natural way to do this is put them in a list like

[t0, t1, t2, t3, t4, t5, t6, t7, t8].

In the above state t0 = (2,2), t1 = (1,1) etc. The whole state could be written as:

[(2,2), (1,1), (3,2), (3,1), (2,3), (2,1), (3,3), (1,3), (1,2)]

Similarly, the goal state

1	2	3
8		4
7	6	5

could be represented by

[(2,2), (1,1), (2,1), (3,1), (3,2), (3,3), (2,3), (1,3), (1,2)].

We can use the following Haskell code to simplify this.

```
type TilePos  =  (Int, Int)
type State  =  [TilePos]
goal :: State
goal = [(2,2),(1,1),(2,1),(3,1),(3,2),(3,3),(2,3),(1,3),(1,2)]
```

Generating Successor States

First of all a function with the signature

```
      moves :: State -> [State]
```

is required. One way to implement this is to initially separate the empty tile position from the other tile positions and employ an auxiliary function

```
      moves (e:ts)        =  movs [e] ts
```

The auxiliary function *movs* that recursively iterates thru the list of tile positions comparing each tile position with that of the empty tile. If for any tile the distance equals 1, a successor state can be constructed by swapping the two positions, and this new state is inserted into the list of successor states. As each tile position is examined, it is moved from the second to the first list. This process terminates when all tile positions have been examined, i.e. second list becomes empty.

Code for this follows.

```
movs :: [TilePos] -> [TilePos] -> [State]
movs (e:ts) []        =  []
movs (e:ts1) (t:ts2)
  | dist e t == 1   =  move : movs (e:ts1++[t]) ts2
  | otherwise       =  movs (e:ts1++[t]) ts2
    where
      move = t:ts1 ++ e:ts2
```

Haskell Code

The code below is well commented in order to aid your understanding.

```
-----------------------------------------
--                                     --
--      Eight Puzzle in Haskell        --
--                                     --
-----------------------------------------

-- Eight tiles, numbered from 1 to 8 and one empty cell
-- We can think of the empty cell as a "missing" tile and
-- call it tile 0.

-----------------------------------------
--                                     --
--       State Representation and      --
--       Generating Successor States   --
--       or Moves                      --
--                                     --
-----------------------------------------

-- To represent a tile position
```

```
-- TilePos  =  (x,y)

type TilePos  =  (Int, Int)

-- We represent the state of the puzzle thus:
-- the position of the empty cell e and each
-- tile is stored in a list in a fixed order, i.e.
-- state  =  [e, t1, t2, t3, t4,t5, t6, t7, t8]

type State  =  [TilePos]


-- Declare and define some initial states and the goal state.

nil, s1, s2, goal :: State
s1 = [(2,2),(1,1),(2,1),(3,1),(2,3),(3,2),(3,3),(1,3),(1,2)]
s2 = [(2,3),(1,1),(2,1),(3,1),(2,2),(3,2),(3,3),(1,3),(1,2)]
goal = [(2,2),(1,1),(2,1),(3,1),(3,2),(3,3),(2,3),(1,3),(1,2)]

-- nil is a special non existent state which will be useful below.
nil = []


-- Manhattan distance between two tile positions
-- A tile can only be moved from one position to
-- another if the Manhattan distance == 1.

dist :: TilePos -> TilePos -> Int
dist (x,y) (u,v)  = abs(x-u) + abs(y-v)



-- Function moves generates a list a possible successor states
-- or moves from the current state. It uses a helper func movs.

moves :: State -> [State]
moves (e:ts)        =  movs [e] ts

movs :: [TilePos] -> [TilePos] -> [State]
movs (e:ts) []        =  []
movs (e:ts1) (t:ts2)
  | dist e t == 1   =  move : movs (e:ts1++[t]) ts2
  | otherwise       =  movs (e:ts1++[t]) ts2
    where
      move = t:ts1 ++ e:ts2


-----------------------------------------
--                                     --
--       State Space Search with       --
--       Breadth First Search          --
--                                     --
-----------------------------------------

-- Function solve sets the program going with initial
-- state as input. Returns the path to goal state.
-- Here it uses BFS.
```

```
solve :: State -> [State]
solve  s  =  bfsSolv [(s,nil)] []


-- Breadth first search keeps two lists of states,
-- os or "open" states records new states which have
-- been generated but not yet examined; and cs or
-- "closed" states records those states which have
-- been examined and proved not to be the goal.


-- So that the soln path can be recovered when the goal state
-- is reached, we record each state with its parent state.
-- We make nil the parent of the starting state.

bfsSolv :: [(State,State)] -> [(State,State)] -> [State]
bfsSolv [] cs           =   []
bfsSolv ((s,p):os) cs
  |s == goal         =   findSolnPath s ((s,p):cs)
  |otherwise         =   bfsSolv (os ++ newSs) ((s,p):cs)
   where
     newStates1   =  moves s
     newStates2   =  remAlreadyVisited newStates1 cs
     newStates    =  remAlreadyVisited  newStates2 os
     newSs        =  addParent s newStates
```

```
remAlreadyVisited [] old  =  []
remAlreadyVisited (s:ss) old
  | notIn s old           =  s : remAlreadyVisited ss old
  | otherwise             =  remAlreadyVisited ss old


notIn s []             =  True
notIn s ((x,p):ss)
  | s == x || s == p   =  False
  | otherwise          =  notIn s ss


findSolnPath  s cs
  | p == nil     =  [s]
  | otherwise    =  findSolnPath p cs ++ [s]
    where
      p = parent s cs

parent :: State -> [(State, State)] -> State
parent s ((x,p):cs)
  | s == x    =  p
  |otherwise  =  parent s cs

addParent p []      =  []
addParent p (s:ss)  =  (s,p) : addParent p ss
```

```
-----------------------------------------
--                                     --
--    Pretty Printing the Solution     --
--                                     --
-----------------------------------------


-- Code to find and pretty print the solution path.
-- Function    go s    runs the solve function from
-- initial state s which returns the solution path.
-- It then converts the soln path to a string and
-- finally uses the Haskell function putStr to display
-- the solution string.

go :: State -> IO ()
go s = putStr solnStr
   where
     solnStr = toStr solnPath
     solnPath = solve s


-- Convert a soln path to a long string.
toStr :: [State] -> String
toStr []      =  ""
toStr (s:ss)  =  toString s ++ toStr ss


-- Convert a state to a string
toString :: State -> String
toString s  = row 1 s ++ row 2 s ++ row 3 s ++ "\n"


-- Convert one row of a state to a string.
-- Need to know which tile is at each pos in the row.

row :: Int -> State -> String
row n s =  " " ++ t 1 ++ " " ++ t 2 ++ " " ++ t 3 ++ "\n"
  where
    t m = show (whichTileAt (m,n) s)


-- Tiles are numbered from 1 to 8, 0 is for 'missing' tile
-- Function to find out which tile is at some position.

whichTileAt :: TilePos -> State -> Int
whichTileAt pos (tilePos:tilePs)
 | pos == tilePos     =   0
 | otherwise          =   1 + whichTileAt pos tilePs
```

Exercise

They following starting states require 4 steps, 5 steps and 18 steps respectively to the goal state. Represent the them in Haskell and run the eight-puzzle program on them.

Check the number of steps in the solution of each. What happens with the last one?

1	3	4
8		2
7	6	5

2	8	3
1	6	4
7		5

2	1	6
4		8
7	5	3