**Functional Specification**

**Introduction**

**“The Taquin (puzzle) is a sliding puzzle that consists of a frame of numbered square tiles in random order with one tile missing.** The puzzle also exists in other sizes, particularly the smaller 8-puzzle. If the size is 3×3 tiles, the puzzle is called the 8-puzzle or 9-puzzle, and if 4×4 tiles, the puzzle is called the 15-puzzle or 16-puzzle named, respectively, for the number of tiles and the number of spaces. The object of the puzzle is to place the tiles in order by making sliding moves that use the empty space.

**The n-puzzle is a classical problem for modelling algorithms involving heuristics**. Commonly used heuristics for this problem include counting the number of misplaced tiles and finding the sum of the taxicab distances between each block and its position in the goal configuration. Note that both are admissible, i.e. they never overestimate the number of moves left, which ensures optimality for certain search algorithms such as A\*.” (Wikipedia)

There are several algorithms to solve the puzzle. These are:

* Breadth-first search algorithm
* Greedy
* Depth-first search algorithm
* Iterative deepening search algorithm (IDA\*)
* A\*

The three algorithms given above differs on the choice of the search path (node to node).

**In this project we will write a Lisp Program to solve the taquin puzzle (3x3) using A\*. We'll use the dialect Common Lisp with LispWorks.**

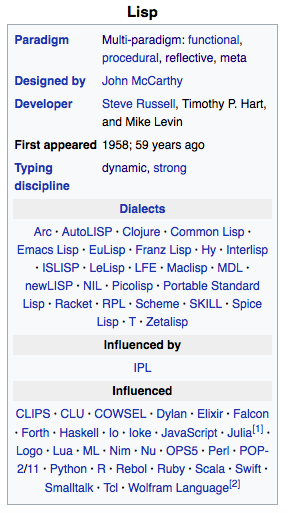
First, we are going to give some information about LISP.

**LISP**

“**Lisp is a family of computer programming languages** with a long history. Originally specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today. Only Fortran is older, by one year. Lisp has changed since its early days, and many dialects have existed over its history. Today, the best known general-purpose Lisp dialects are Clojure, Common Lisp and Scheme.

Lisp was originally created as a practical mathematical notation for computer programs, influenced by the notation of Alonzo Church's lambda calculus. It quickly became the favored programming language for artificial intelligence (AI) research. As one of the earliest programming languages, Lisp pioneered many ideas in computer science, including tree data structures, automatic storage management, dynamic typing, conditionals, higher-order functions, recursion, etc.

The name LISP derives from "LISt Processor". **Linked lists are one of Lisp's major data structures, and Lisp source code is made of lists. Thus, Lisp programs can manipulate source code as a data structure.”** (Wikipedia)



**Solution**

We will implement the A\* algorithm to solve the problem. It is a computer algorithm that is widely used in pathfinding and graph traversal, the process of plotting an efficiently traversable path between multiple points, called nodes.

It is one of the basic and the most common algorithm which you’ll see widely across the internet. It is widely used for pathfinding and various other problems which have admissible heuristics and can be converted to a graph form.

Pros:

* It will always find the optimal solution provided that it exists and that if a heuristic is supplied it must be admissible.
* Heuristic is not necessary, it is used to speed up the process.
* Various heuristics can be integrated to the algorithm without changing the basic code.
* The cost of each move can be tweaked into the algorithms as easily as the heuristic.
* It isn’t constrained to a unidirectional search.

Cons:

* Not the best algorithm for each problem in terms of memory and processing required.
* Uses a lot of memory since each node created has to be kept accounted for.

Prerequisite knowledge.

* + 1. You need convert your problem into a graph (nodes and edges)
    2. You need to have an admissible heuristic for the problem for optimal and faster search(optional). A Heuristic is a function that, when computed for a given state, returns a value that estimates the demerit of a given state, for reaching the goal state.
    3. You need to have an cost function so get the cost required to travel from node to other. It is important because the function is used in the algorithm.

We need to realize the search space as a graph. Each state in the graph is represented with it’s puzzle configuration, thus each node is a separate puzzle state which is produced by sliding a tile to the blank space on the previous state. Let’s take the following case:

The above figure is considered as a single node in a graph, let’s take it as the starting node. The following will be the expanded graph in the next epoch when the available moves are implemented.

Each node can have maximum of 4 children, the graph fill further expand in a similar fashion with each child pointing to the parent using pointer.

The g score here, which is the cost of the move will increase by 1 at each depth since each tile sliding to the blank space represents one move and in the end we need to get the optimal moves for the puzzle instance, this basically mean that we have to add 1 to the g score of the parent before storing it in the child nodes.

We can use the following Heuristics:

* Hamming Distance/Misplaced Tiles: Just as the name suggests, this heuristics returns the number of tiles that are not in their final position. Let’s take this puzzle instance.

Here we see that all the tiles are not in their final position except ‘7’ thus the heuristic value(h) will be 7 for this instance. Remember we don’t consider the blank space as a tile while calculating this heuristic value.

This heuristic is however the slowest and a huge amount of nodes will be explored to reach the goal state compared to other heuristics.

* Manhattan Distance/Taxicab geometry: Manhattan Distance of a tile is the distance or the number of slides/tiles away it is from it’s goal state.Thus, for a certain state the Manhattan distance will be the sum of the Manhattan distances of all the tiles except the blank tile.

Let’s take the following example.

Here we see the tiles 5, 3, 4, 8 are misplaced so we calculate the Manhattan distances of each of these tiles which are 2, 3, 2, 1 respectively, thus the Manhattan heuristic value for this state will be 8 (h=8).

Manhattan Distance is faster than Hamming distance explained above but it’s still slow for higher values of N.

* Linear Conflict + Manhattan Distance/Taxicab geometry: Two tiles ‘a’ and ‘b’ are in a linear conflict if they are in the same row or column, also their goal positions are in the same row or column and the goal position of one of the tiles is blocked by the other tile in that row.

Let’s take the following example:

In this instance we see that tile 4 and tile 1 are in a linear conflict since we see that tile 4 is in the path of the goal position of tile 1 in the same column or vice versa, also tile 8 and tile 7 are in a linear conflict as 8 stands in the path of the goal position of tile 7 in the same row. Hence here we see there are 2 linear conflicts.

As we know that heuristic value is the value that gives a theoretical least value of the number of moves required to solve the problem we can see that one linear conflict causes two moves to be added to the final heuristic value(h) as one tile will have to move aside in order to make way for the tile that has the goal state behind the moved tile and then back resulting in 2 moves which retains the admissibility of the heuristic.

Linear conflict is always combined with the Manhattan distance to get the heuristic value of that state and each linear conflict will add 2 moves to the Manhattan distance as explained above, so the ‘h’ value for the above state will be

Manhattan distance + 2\*number of linear conflicts

Manhattan distance for the state is: 10  
 Final h: 10 + 2\*2= 14

Linear Conflict combined with Manhattan distance is significantly way faster than the heuristics explained above and 4 x 4 puzzles can be solved using it in a decent amount of time.Just as the rest of the heuristics above we do not consider the blank tile when calculating linear conflicts. We'll implemented this heuristic.

So first of all we need to decide what all information is needed to be kept in the node.

* + We need to store the heuristic value h and the cost node g.
  + We need to save the puzzle state at that instance, thus a 3x3 grid.
  + The pointer that points to the parent node.

We'll use classes (paradigm object oriented) to represent graphs and nodes. The nodes are saved in a table-hash.

The heuristic value h and the cost g are saved in each node. Use hash-table because the search is more efficient than other data structure. The search is O(1) or constant.

**Algorithm A\*:**

The key feature of the A\* algorithm is that it keeps a track of each visited node which helps in ignoring the nodes that are already visited, saving a huge amount of time.It also has a list that holds all the nodes that are left to be explored and from this list it chooses the most optimal node thus saving time not exploring unnecessary or less optimal nodes.

So we use two lists namely ‘open list‘ and ‘closed list‘ the open list contains all the nodes that are being generated and are not existing in the closed list and each node explored after it’s neighboring nodes are discovered is put in the closed list and the neighbors are put in the open list this is how the nodes expand. Each node has a pointer to it’s parent so that at any given point it can retrace the path to the parent. Initially the open list holds the start(Initial) node. The next node chosen from the open list is based on it’s f score, the node with the least f score is picked up and explored.

F score is nothing but the sum of the cost to reach that node and the heuristic value of that node. For any give node the f score is defined as:

###### **f(x)=h(x)+g(x)**

Where g(x) is the cost of that node, h(x) is the calculated heuristic of that node and x is the current node

The algorithm in natural language is the follow:

Start = initial node;

Goal = final node;

open\_list.insert(Start); //list of nodes, we'll implement a table hash with key equal to //function f and the value is the node (numbers concatenated)

// (setq ope\_list ( make-hash-table ))

close\_list = empty set; // list of nodes, we'll implemet a set of nodes

// (defparameter \*myset\* ())

Start.g = 0;

Start.h = heuristic(Start);

Start.f = Start.g + Start.h;

Begin

Repeat hasta (No open\_list.empty()) // run until open\_list is empty

process = open\_list(node with min(f(x)))

if (process == Goal)

return (function path(process)) //return path to goal found

endif

open\_list.remove(process)

closed\_list.insert(process)

foreach (node in nextnodes(process)) //run for all nodes possible from current

if (closed\_list.count(node) != 0) // it already exists in closed list

continue with the next loop

endif

if (open\_list.count(node) == 0)

open\_list.insert(node)

else

actual\_node = open\_list.find(node) //find node in open list

if (node.g < actual\_node.g)

actual\_node.g = node.g

actual\_node.f = node.f

actual\_node.parent = node.parent //change of parent // because this father

// is better

endif

endif

endforeach

endrepeat

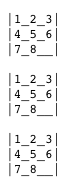
Fin

We have four options to run the program:

1. Use the console to show the result. We can run the program by calling:

CL-USER > (program\_main (1,3,4,6,2,8,5,7,9), 3) ; 3 is the dimension of the pu

This starts the search with a list containing the start cell and an empty route. The result comes back:



* + 1. Use GUI toolkit CAPI (This will take more development time).
    2. Interface with Java and then. Interface with Java and then visualize the result in Java using the facilities of the tools or IDES eclipse (This will take more development time).
    3. Load and Compile the program doing the main logic into the file

There are other solutions that use the database. They are the fastest compared to the otherheuristics. In general a pattern database consists of a creating a database of all patterns occurring in a selected area of the puzzle and the minimum cost/Heuristic value required for it to reach the goal state from every possible permutation of positions of tile in that group i.e. It stores a collection of solutions to sub-problems that must be achieved to solve the problem. It’s used as a table lookup to get the ‘h’ value of the current state.

A consistent heuristic is when it never overestimates the cost, keeping this in mind , imagine if 2 tiles are in Linear conflict , for 1 tile to reach its goal position in the best of the best cases ( imagine no tiles in between and empty places all around ) one of the pieces would have to move aside to make way for the other piece and back , which is 2 moves at the least in the best of the best case. This makes it consistent as it’ll never over estimate the cost.

References:

* + - 1. <https://en.wikipedia.org/wiki/15_puzzle>
      2. <https://algorithmsinsight.wordpress.com/graph-theory-2/a-star-in-general/implementing-a-star-to-solve-n-puzzle/>
      3. <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>
      4. <https://algorithmsinsight.wordpress.com/graph-theory-2/a-star-in-general/implementing-a-star-to-solve-n-puzzle/>