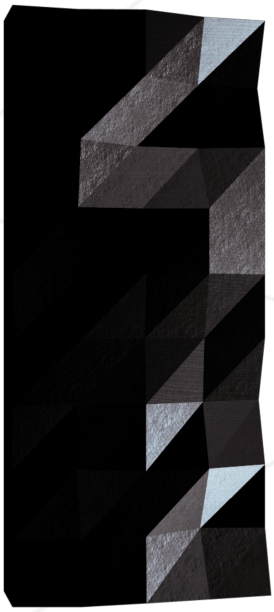

N-Puzzle

Solve it better than a 60-year-old drunkard

42 staff staff@42.fr

Abstract: *The goal of this project is to programmatically solve the N-puzzle.*

Contents

I Foreword **2**

II The project **3**
II.1 What you have to do 3
II.2 Bonuses 4
II.3 Language constraints 4
II.4 Defense sessions 4

III Appendix **6**
III.1 Pseudo-implementation of A* 6
III.2 Input file format 7

Chapter I

Foreword

Here's what Wikipedia has to say on `Beef Wellington`:

`Beef Wellington is a preparation of filet steak coated with pâté (often pâté de foie gras) and duxelles, which is then wrapped in puff pastry and baked. Some recipes include wrapping the coated meat in a crêpe to retain the moisture and prevent it from making the pastry soggy.`

`A whole tenderloin may be wrapped and baked, and then sliced for serving, or the tenderloin may be sliced into individual portions prior to wrapping and baking. Many spices may be added to enhance the flavour; some examples are allspice, any grilling mix or ginger.`

`The origin of the name is unclear.`

`There are theories that suggest that beef Wellington is named after Arthur Wellesley, 1st Duke of Wellington. Some theories go a step further and suggest this was due to his love of a dish of beef, truffles, mushrooms, Madeira wine, and pâté cooked in pastry, but there is a noted lack of evidence supporting this. In addition to the dearth of evidence attaching this dish to the famous Duke, the earliest recorded recipe to bear this name appeared in a 1966 cookbook.`

`Other accounts simply credit the name to a patriotic chef wanting to give an English name to a variation on the French filet de bœuf en croûte during the Napoleonic Wars.`

`Still another theory is that the dish is not named after the Duke himself, but rather that the finished joint was thought to resemble a Wellington boot, a brown shiny military boot named after the duke.`

Chapter II

The project

The goal of this project is to solve the N-puzzle ("taquin" in French) game using the A* search algorithm or one of its variants.

You start with a square board made up of N*N cells. One of these cells will be empty, the others will contain numbers, starting from 1, that will be unique in this instance of the puzzle.

Your search algorithm will have to find a valid sequence of moves in order to reach the final state, a.k.a the "snail solution", which depends on the size of the puzzle (Example below). While there will be no direct evaluation of its performance in this instance of the project, it has to have at least a vaguely reasonable perfomance : Taking a few second to solve a 3-puzzle is pushing it, ten seconds is unacceptable.

3-puzzle

1	2	3
8		4
7	6	5

4-puzzle

1	2	3	4
12	13	14	5
11		15	6
10	9	8	7

5-puzzle

1	2	3	4	5
16	17	18	19	6
15	24		20	7
14	23	22	21	8
13	12	11	10	9

The only move one can do in the N-puzzle is to swap the empty cell with one of its neighbors (No diagonals, of course. Imagine you're sliding a block with a number on it towards an empty space).

II.1 What you have to do

Implement the A* search algorithm (or one of its variants, you're free to choose) to solve an N-puzzle, with the following constraints:

- You have to manage various puzzle sizes (3, 4, 5, 17, etc ...). The higher your program can go without dying a horrible, horrible death, the better.
- You have to manage both randomly determined states (of your own generation of course), or input files that specify a starting board, the format of which is described in the appendix.
- The cost associated with each transition is always 1.
- The user must be able to choose between at LEAST 3 (relevant) heuristic functions. The Manhattan-distance heuristic is mandatory, the other two are up to you. By "relevant" we mean they must be admissible (Read up on what this means) and they must be something other than "just return a random value because #YOLO".
- At the end of the search, the program has to provide the following values:
  - Total number of states ever selected in the "opened" set (complexity in time)
  - Maximum number of states ever represented in memory at the same time during the search (complexity in size)
  - Number of moves required to transition from the initial state to the final state, according to the search
  - The ordered sequence of states that make up the solution, according to the search
  - The puzzle may be unsolvable, in which case you have to inform the user and exit

II.2 Bonuses

- Configure the appropriate `g(x)` and `h(x)` functions to run both the uniform-cost and greedy searches. Execute with the same output (Of course, the solution may be different. Read up on why, that's the point.)

II.3 Language constraints

You are free to use whatever language you want, but keep in mind that some languages are more time and space-efficient than others, so your choice in languages may influence the performances of your program.

II.4 Defense sessions

For the defense session, be prepared to :

- Explain your implementation of the search algorithm, most importantly explain which variant of A* it is, and why you chose it. If you can't explain it, you might as well not even implement it.

- Explain your choice of heuristics and show they're admissible. Same as the previous item : If you can't explain it, no use implementing it.

- Justify your choices in terms of data structures

- Actually run your program on examples of varied sizes that demonstrate your successful implementation of the project. You're expected to provide some examples for the defense.

Chapter III

Appendix

III.1 Pseudo-implementation of A*

```
set opened ← { initial states }
           // States to be examined and candidates to expansion
set closed ← ∅
           // States already selected by the algorithm, compared to the
           // solution, and expanded
bool succes ← false
While ( opened ≠ ∅ ) and ( not succes ) do
  state e ← select_according_to_Astar_strategy_in ( opened )
  If is_final ( e ) // Compares 'e' to a solution state
    Then succes ← true
    Else opened ← opened - e
         closed ← closed + e
         ForEach state s in expand(e) do
            If (s ∉ opened) and (s ∉ closed)
              Then opened ← opened + s
                   predecessor(s) ← e
                   g(s) ← g(e) + C(e-->s)
              Else // s is in 'opened' or in 'closed'
                   If g(s) + h(s) > g(e) + C(e-->s) + h(s)
                   // i.e. f value > 'potentially new' f value
                     Then g(s) ← g(e) + C(e-->s)
                          predecessor(s) ← e
                          If s ∈ closed
                            Then closed ← closed – s
                                 opened ← opened + s
            EndIf  EndIf EndIf
         EndForEach
  EndIf
EndWhile
If succes Then . . . Else . . . EndIf
```

III.2 Input file format

```
zaz@blackjack:~/npuzzle/$ cat -e npuzzle-3-1.txt
# this is a comment$
3$
3 2 6 #another comment$
1 4 0$
8 7 5$
zaz@blackjack:~/npuzzle/$ cat -e npuzzle-4-1.txt
# PONIES$
4$
 0 10  5  7$
11 14  4  8$
 1  2  6 13$
12  3 15  9$
zaz@blackjack:~/npuzzle/$ cat -e npuzzle-4-1.txt
# Puzzles can be aligned, or NOT. whatever. accept both.$
4$
0 10 5 7$
11 14 4 8$
1 2 6 13$
12 3 15 9$
zaz@blackjack:~/npuzzle/$
```