**Artificial Intelligence I** , *prof. Pasquale Caianiello 19.10.16*

Homework 1: implementing heuristic search in puzzles. *Presented by: Yuna Frolov*

Hidato

**Hidato** (Hebrew: חידאתו‎‎, originating from the Hebrew word *Hida* = Riddle) is a logic puzzle game invented by Dr. Gyora M. Benedek, an Israeli mathematician. The goal of Hidato is to fill the grid with consecutive numbers that connect horizontally, vertically, or diagonally.

In Hidato, a grid of cells is given. It is usually square-shaped, like Sudoku or Kakuro, but it can also include irregular shaped grids like hearts, skulls, and so forth. It can have inner holes (like a disc), but it has to be made of only one piece.

The goal is to fill the grid with a series of consecutive numbers adjacent to each other vertically, horizontally, or diagonally.

In every Hidato puzzle the smallest and the highest numbers are given on the grid. There are also other given numbers on the grid (with values between the smallest and the highest) to help direct the player how to start the solution and to ensure that Hidato has a single solution.

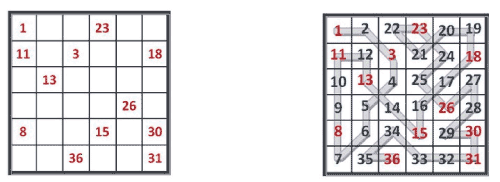
*Solving techniques:*

As in many logic puzzles, the basic resolution technique consists of analyzing the possibilities for each number of being present in each cell. When a cell can contain only one number (Naked Single) or when a number has only one possible place (Hidden Single), it can be asserted as belonging to the solution.

One key to the solution is it does not have to be built in ascending (or descending) order; it can be built piecewise, with pieces starting from different givens.

As in the Sudoku case, the resolution of harder Hidato or Numbrix puzzles requires the use of more complex techniques - in particular of various types of chain patterns.

[Explanation taken from Wikipedia – [Hidato](https://en.wikipedia.org/wiki/Hidato)]



**\_\_init\_\_.py**

For my homework I have used the MissCannPython implementation provided, as an example of using Python, because it is my first time working with this language.

I have worked with 3 python files:

**Main.py:**

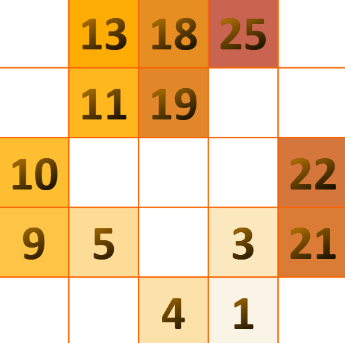
This file is for the main program, which runs the game, and also has the *breadth\_first\_search*, *path* and *pick* functions (*pick* uses *argMax*). In the file I have given three examples and run them to show that the algorithm works on different sizes, and at different times.

The *pick* and *path* functions are almost as given in the class example, just with a max preference instead of min. When working with the example, I noticed that the ‘dictOfStates’ never updates the heuristic value, so I added a function called *updateState*. The search is breadth first search as learned in class (changed it from the example to fit my code, and decided to ‘remove’ explored states from the horizon).

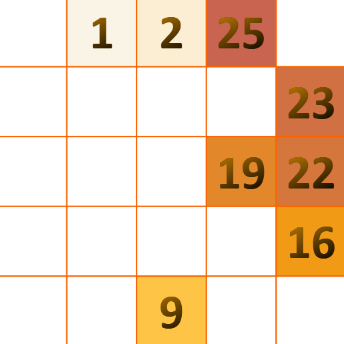
The examples print out the initial board, the final solution board (should be the same each time, if the board is designed correctly), how many states were explored until a solution was found, how many steps you need to take to reach the solution, and the time it takes to execute each example.

**Game -> \_\_init\_\_.py:**

This file contains the representation of the game and the states, which help to play the game from the main. I am using a grid representation for the board, where ‘0’ represents an empty space, and a number is a fixed number in the ‘snake’ of numbers to fill. I also keep track of the current number I’m working with, and the row and column of that number I’m searching around. For example, in this case:



I am starting with the number 1, in row = 4 col = 3 and searching its *neighbors* for *admissible* states – which are the empty spaces top-left and mid-right, with the new number to set – 2. Also, there is a function that checks what should be the next number. Because, if you start with 1 in the following example:



The next number to insert should be 2, but it is already on the board, so the function ‘*nextNum*’ checks for the next available number and starts the search from there.

Each time a state is found *admissible* – which means the space was free, and also it was adjacent to a higher number if it existed on the board – it copies the board, places the next number on it, and sets the new board in the new state that is added to the horizon.

**The solution** of the game is found when there are no more ‘0’, which means we’ve filled the board, climbing the numbers to the top.

**Heuristics -> \_\_init\_\_.py:**

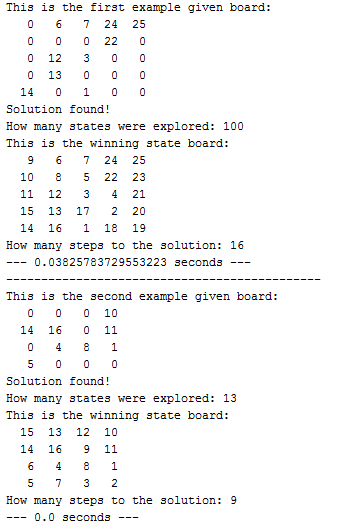
The heuristic file only contains the heuristic *initialization* and the *HidatoHeuristic* I’ve chosen to use – in my research I could not find the best heuristic solution for the Hidato game, but it is recommended to go with a state that can guarantee a correct placement (in case of only one open spot / corner piece). My heuristic counts how many taken spaces are there, and returns the number as the weight of this state. The pick function in main chooses the maximum weight to be picked, that way the state with least possible moves brings me closer to the solution.

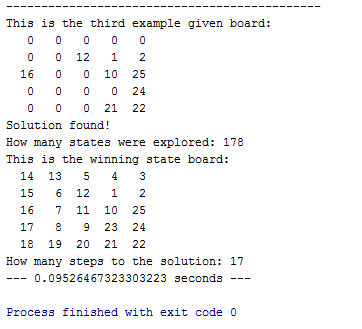
**I have compared it with picking the minimum weight (meaning more possible moves): In easier cases the min choice explored fewer states, but in harder puzzles the states and time execution increased dramatically.**

**\_\_init\_\_.py**

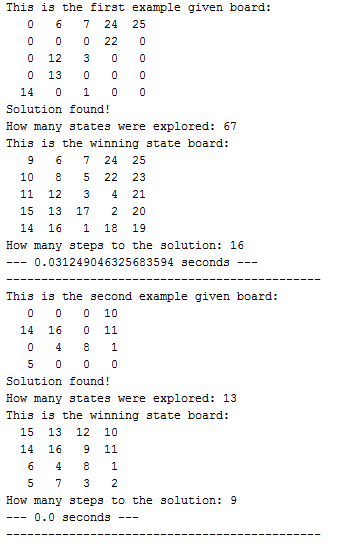
Here are some runs of the program on my machine:

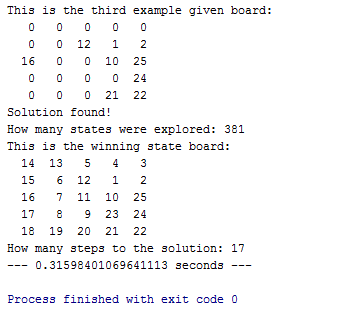
* With the ‘max’ choice:





* With the ‘min’ choice:





**\_\_init\_\_.py**

In my research I found those sites most useful:

* [Hidoku Solver](http://hidoku-solver.appspot.com/#0) – used this to play the game and find solvable examples
* [Wikipedia page](https://en.wikipedia.org/wiki/Hidato) – to study about the game
* [Talk:Solve a Hidato Game](https://rosettacode.org/wiki/Talk:Solve_a_Hidato_puzzle) – to learn about the tactics of solving the Hidato game
* [Heuristics](http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html) – a page that explains different kinds of searches and their uses

Thank you for your time.