MIMOUN Avi
PARDINI Raphaël

Paris Dauphine University
GILBERT Hugo

Sliding Block Puzzles

Artificial intelligence project report

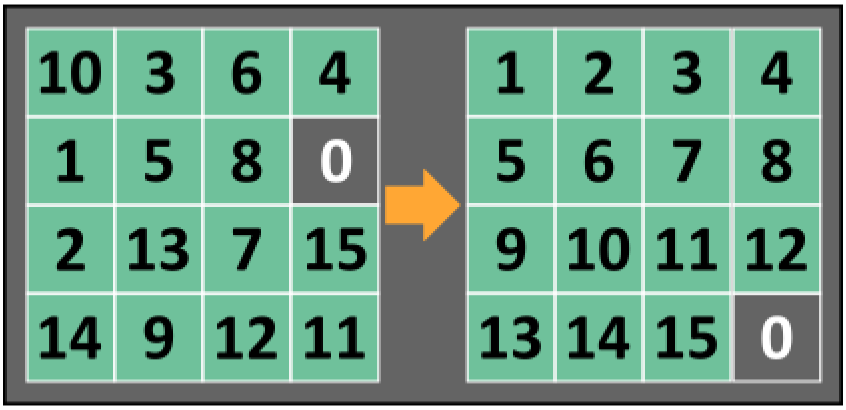


Table of contents

1. Introduction

As part of the artificial intelligence course, we chose to work on the sliding-block-puzzles project.
In order to implement the different search algorithms, we used the programming language Python, in order to create a package usable by all.
We used a Flutter application (Web, Android, IOS, Desktop) as an example for the 8-puzzle, but it is also possible to use our package through the terminal (see Running the application).
In this report, we will first describe the structure of our code, then define the different heuristics that we have developed.
Finally, we will describe our experimental protocol as well as the results obtained.

2. Code structure

We have organized our app to be available as a package.

2.1 Preview folders and files

sliding_puzzle/	Package folder
sliding_puzzle/cli.py	Module for interacting with the package using the command interface
sliding_puzzle/wsgi.py	Module allowing to interact with the package using a web service
sliding_puzzle/algorithm/	Package implementing the algorithms used for solving puzzles
sliding_puzzle/representation/	Package that represents the model of a Puzzle and the different heuristics that can be used on a Puzzle.
report/	Experimental protocol and protocol statistics
tests/	Unit tests of the package
sample/	Sample scripts
.github/	File to achieve continuous integration and

	continuous delivery
.pre-commit-config.yaml	File used to create pre-commits (action to be performed before making a commit)
Makefile	Automates common project commands
LICENSE	License we assigned to the project (MIT)

2.1 Structure details

Given the similarity of all algorithms, we decided to create a Search interface that will be implemented by all algorithms.
This class gives:

- How the solve function should be implemented
- What values are common and need to be changed by the children in the class
- Static functions that help in the development of algorithms and have useful functions for the entire resolution (example: knowing if a puzzle is solvent)

Like the Search interface, we have created a Heuristic interface that brings together all heuristics.
The Heuristic interface defines an abstract static method that must be implemented by the children.
Thus, the methods of resolution which need a heuristic, have a parameter which makes it possible to pass in parameters a heuristic class.

The most important class is of course Puzzle (which can be found in the representation package).
The class may seem long but is filled with:

- comments
- native methods redefined
- methods for manipulating / converting a Puzzle

2.2 Run the application

To install the application use:

```
python -m pip install \
git+https://github.com/av1m/sliding-block-puzzles
```

We have implemented different solutions to run the application:

1. **Command line**

   In the form of a python module, a CLI module has been programmed to interact and test the package.

   ```
   sliding_puzzle # Or python -m sliding_puzzle

   # Example
   sliding_puzzle \
     --tiles 4 1 2 3 5 6 7 11 8 9 10 15 12 13 14 0 \
     --method a_star depth_limited \
     --no-blank-at-first
   ```

2. **Web service**

   A WSGI module has been implemented in order to create a web server on which we can make HTTP GET requests

   ```
   # In development environment
   make serve
   # In production environment
   gunicorn sliding_puzzle.wsgi --reload --timeout 1000
   ```

   This makes it possible to solve n-puzzles in a different environment.
   We have to fork and implement a real example of this use case in Flutter.

3. **In a Python application/script**

   By using the package as a dependency and importing it.
   A series of examples are available in the "sample" folder (located at the root of the project).

To configure the development environment (creation of a virtual environment, installation of dependencies, etc.), run the command:

`make install`

It is possible to test (unit test, black) the application by running the command:
`make serve`

💡 It is possible to deploy the application on Heroku using the Procfile file and the command `make deploy`

3. Definition of heuristics

In order to guide informed search methods, we have implemented four heuristics, which we rank in order of increasing dominance:

- **Misplaced** : this heuristic is the sum of the cost of a trip for each piece of the puzzle that is misplaced
- **Manhattan** : this heuristic is the sum of the cost of moving from the initial position to the final position for each piece of the puzzle that is misplaced
- **Linear Conflict** : this heuristic uses the Manhattan heuristic, to which it adds the cost of moving the pieces of the puzzle that are in conflict.
  Two boxes tj and tk are in linear conflict (i.e. linear conflict) if tj and tk are on the same row, the goals positions of tj and tk are both in this row, tj is to the right of tk, and the position of tj's goal is to the left of tk's goal position. In order for the heuristic to be admissible, a piece that conflicts with two or more pieces must only be counted once.
  It’s a relaxation of the game of teasing.

4. Experimental protocol

For each research method, we will analyze and compare the following:

- The number of nodes generated (time complexity)
- The number of nodes stored in memory (memory complexity)
- The maximum size that a puzzle can reach

In order to obtain reliable figures, we will apply the following experimental protocol.
First, we'll randomly generate a list of five hundred 8-puzzles.
To build them, we will start from the solution and move the empty square a certain number of times (without going back and forth between two movements), a number which will increase from 1 to 100. This list will therefore contain puzzles of different complexity. .
For each research method, we will give it this list to solve, and for each puzzle we will store information 1 and 2 mentioned above.
Finally, we will take the average of these.
If the method takes more than five seconds to solve the puzzle, it is considered to have failed and we move on to the next puzzle.
Then we will repeat this experiment increasing the size n of the n-puzzle by 1 each time until the method takes too long and the solve rate is low.

4.1 Results

4.1.1 Research methods

Here are the settings we used in our testing:

Parameters	
Number of generated puzzles	100
Number of mutations on a Puzzle	[1, 2, ..., 99, 100]
Heuristic (A*, IDA*, GBF, Bidirectional)	Linear conflict
Limit (Depth Limited)	100

Limit (Iterative Deepening)	10
Step (Iterative Deepening)	10
Timeout	5 secondes

And here are the results obtained:

	8-puzzles (n=3)		
Methods	Time complexity	Memory complexity	% of resolution
Bread-first	3961	786	19.00%
Uniform Cost	4547	902	17.00%
Depth-first	-	-	0.00%
Depth-limited	34149	100	13.00%
Iterative-Deepening	29159	11	13.00%
Greedy best-first	824	142	100.00%
A*	4200	683	93.00%
Iterative-lengthening	39445	7	11.00%
Bidirectional	2190	296	100.00%
Iterative-Deepening A*	12827	14.5	46.00%

	15-puzzles (n=4)		
Methods	Time complexity	Memory complexity	% of resolution
Bread-first	2902	647	11.00%
Uniform Cost	3733	829	9.00%
Depth-first	24	6	1.00%
Depth-limited	25983	100	8.00%

Iterative-Deepening	43832	10	11.00%
Greedy best-first	4654	847	100.00%
A*	2341	419	40.00%
Iterative-lengthening	18435	6	8.00%
Bidirectional	4388	586	53.00%
Iterative-Deepening A*	4956	13.8	30.00%

	24-puzzles (n=5)		
Methods	Time complexity	Memory complexity	% of resolution
Bread-first	5100	1180	10.00%
Uniform Cost	1789	416	7.00%
Depth-first	721	155	2.00%
Depth-limited	23811	100	6.00%
Iterative-Deepening	18417	10	7.00%
Greedy best-first	7267	1360	64.00%
A*	2269	420	35.00%
Iterative-lengthening	5342	5	7.00%
Bidirectional	3352	448	44.00%
Iterative-Deepening A*	3146	14	26.00%

Now, let's analyze the figures obtained.
First of all, the averages were only carried out on the puzzles that were solved, so this bias must be taken into account. For example, if Depth-First Search has a lower temporal complexity than A*, it is because DFS solved only a puzzle with one mutation, while A* solved around fifty puzzles with a much higher number of mutations. important. It is therefore always necessary to compare the number of puzzles solved. The graphs in the Appendices provide information on this subject, we can see the cost of solving each method for a given puzzle (with N = 4).

First, there is a big difference between the Tree Search methods and the one in Graph Search. Indeed, the temporal complexity of Tree Search is much greater: of the order of 10^4 against an order of magnitude of 10^3 for Graph Search.
Conversely, the memory complexity is very low for Tree Search (generally between 0 and 100) and higher for Graph Search (on average between 500 and 1000).

These results confirm what we saw in the course: Tree Search methods will have the advantage of a lower memory complexity O (b*m), but with a greater time complexity in $O(b^m)$. Here, with a timeout of 5 seconds and relatively simple problems, the Graph Search is therefore at an advantage. Indeed, they have a memory and time complexity in $O(b^d)$. However, we know that when the problem becomes more complex, the memory will fail before the execution time.

One result surprised us first: the fact that Greedy Best First is the best algorithm, ahead of A*. Indeed, greedy has better stats, and completed some puzzles in seconds while A* takes an hour. But after re-reading the course, we understood why: A* will look for the optimal solution, and will therefore take longer. In addition, it is difficult to scale.
Finally, we see that the four methods that stand out in terms of resolution rate are the informed methods: A*, Bidirectional, Greedy-Best First and Iterative Deepening A*.
This clearly highlights the effectiveness of heuristics on resolution time.
However, these results may vary depending on the choice of heuristics.

4.1.2 Heuristics

In order to compare the heuristics, we counted the sum of the differences between the cost of resolution estimated by them and the cost of effective resolution of each method.
Thus, we can classify the heuristics by dominances: those whose sum of the deviations is the smallest will be the best.
Here are the results obtained:

	8-puzzles (n=3)		
Methods	Heuristic Misplaced : Average deviations	Heuristic Manhattan : Average deviations	Heuristique Linear Conflict : Average deviations
Bread-first	63	34	30

Uniform Cost	45	22	20
Depth-first	-	-	-
Depth-limited	1698	1692	1690
Iterative-Deepening	96	90	88
Greedy best-first	3848	3104	3004
A*	1499	830	742
Iterative-lengthening	12	10	10
Bidirectional	1794	1050	950
Iterative-Deepening A*	509	226	190

	15-puzzles (n=4)		
Methods	Heuristique Misplaced : Average deviations	Heuristique Manhattan : Average deviations	Heuristique Linear Conflict : Average deviations
Bread-first	7	10	10
Uniform Cost	-6	0	0
Depth-first	-1	0	0
Depth-limited	953	958	958
Iterative-Deepening	69	72	72
Greedy best-first	10677	9232	9104
A*	384	168	152
Iterative-lengthening	-5	0	0
Bidirectional	828	390	362
Iterative-Deepening A*	197	82	72

Methods	24-puzzles (n=5)		
	Heuristique Misplaced : Average deviations	Heuristique Manhattan : Average deviations	Heuristique Linear Conflict : Average deviations
Bread-first	0	4	4
Uniform Cost	-2	4	4
Depth-first	208	210	210
Depth-limited	676	682	682
Iterative-Deepening	47	54	54
Greedy best-first	5052	4266	4212
A*	262	106	98
Iterative-lengthening	-2	4	4
Bidirectional	511	230	216
Iterative-Deepening A*	124	34	34

These are consistent. Indeed, the heuristic Linear conflict is dominant in the other two and close to Manhattan. This is because it is a relaxation of the problem with more restrictive rules than Misplaced, and close to Manhattan.

Conclusion

To conclude, there is a clear difference between informed methods and others. They have a higher resolution rate and complexities on average.
In addition, the results confirm the theoretical complexities: Tree Search methods have lower memory complexity than Graph Search methods, but greater time complexity.

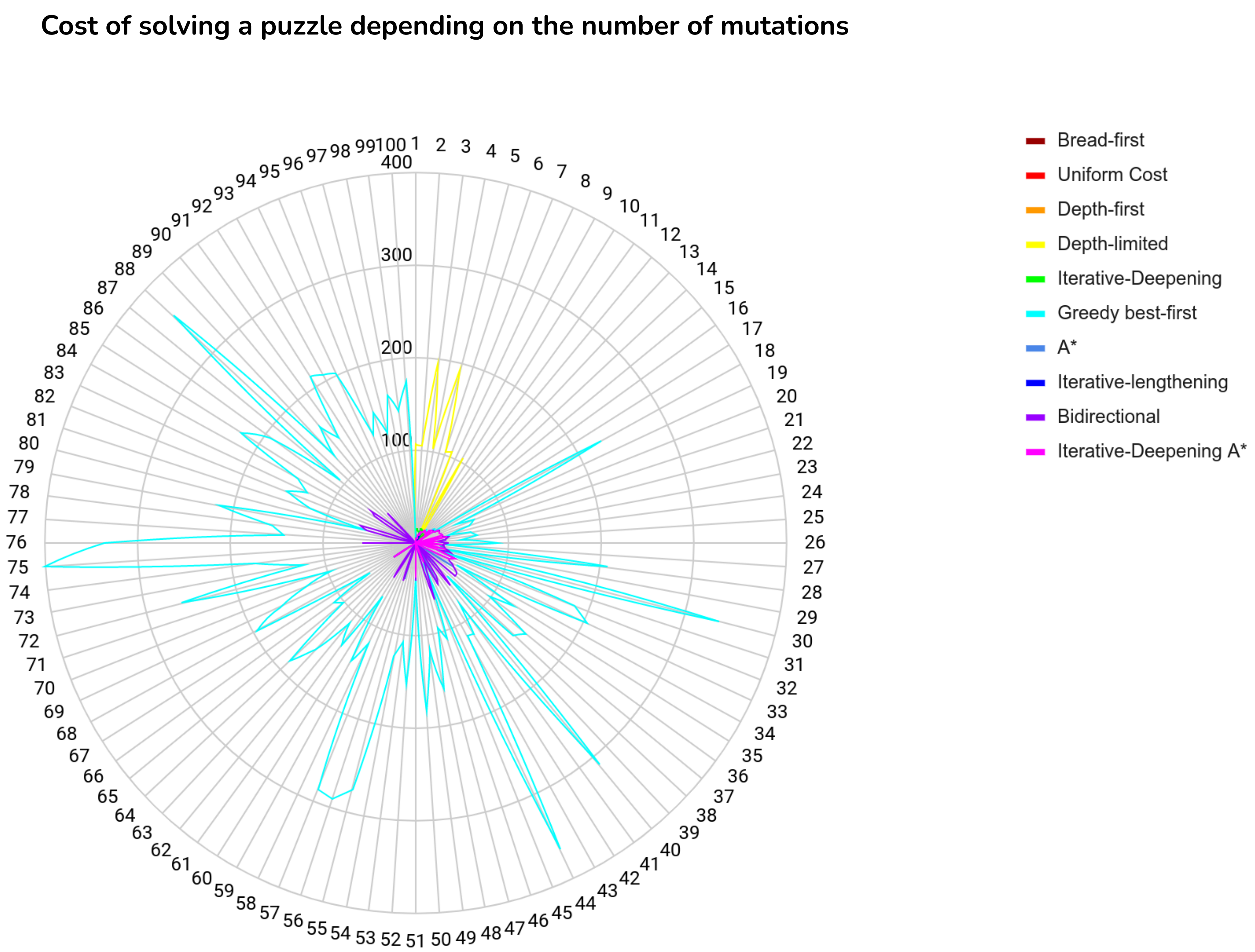
Project links

- Source code (Github) : https://github.com/av1m/sliding-block-puzzles
- Client source code (example) : https://github.com/av1m/slide_puzzle
- Application demo : https://av1m.github.io/slide_puzzle

Reference

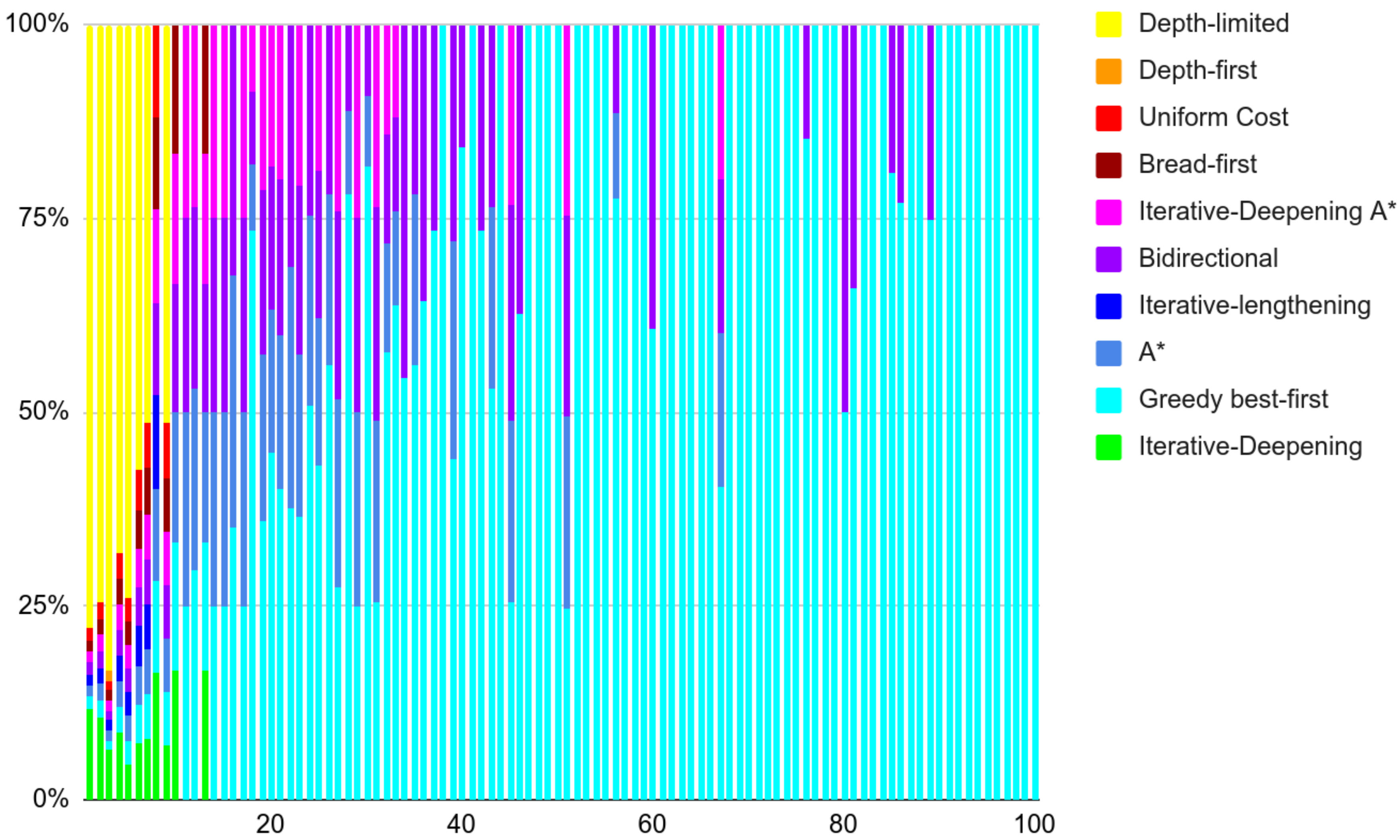
- Criticizing Solutions to Relaxed Models Yields Po. werful Admissible Heuristics
  https://cse.sc.edu/~mgv/csce580sp15/gradPres/HanssonMayerYung1992.pdf
- Graphical interface used : https://github.com/kevmoo/slide_puzzle
  Comparison with the Fork :
  https://github.com/kevmoo/slide_puzzle/compare/master...av1m:master

APPENDICES



Cost of solving a puzzle depending on the number of mutations
Bread-first
Uniform Cost
Depth-first
Depth-limited
Iterative-Deepening
Greedy best-first
A*
Iterative-lengthening
Bidirectional
Iterative-Deepening A*
100
200
300
400

Percentage of the cost of solving a puzzle among the sum of the costs according to the number of mutations



Cost of solving a puzzle (abscissa) according to the number of mutations (ordinate)