

CONCORDIA UNIVERSITY

COMP6721

INTRODUCTION TO ARTIFICIAL INTELLIGENCE

FALL 2018

Mini-Project 1 Report

Author:

Adrien POUPA
40059458

Professor:

Dr. Leila KOSSEIM

October 6, 2018



Contents

1	Java Implementation of the Mini-Project	2
1.1	Running the Program	2
1.2	The Solvers Package	3
1.3	The Heuristics Package	3
2	Using Heuristics to Solve Puzzles	4
2.1	Hamming Distance	4
2.2	Manhattan Distance	4
2.3	Sum of Permutation Inversions	5
3	Experimenting with algorithms, heuristics and puzzle size	6
4	Challenges Encountered	10
	References	11
A	UML Class Diagram	12
B	Expectations of Originality	13

1 Java Implementation of the Mini-Project

In this section, I will be describing my implementation of the puzzle solver. I have added code comments and a JavaDoc for a better understanding. A UML class diagram is available in figure 5. Code references: [1] [6] [7] [4] [3].

1.1 Running the Program

The "root" package contains the `Main` class, which is the entry point used to execute the program. A user can either enter numbers to create a puzzle, or run the program with the argument "random" to have solve a randomly generated Puzzle. In any case, the puzzle will be solved using the Depth-First Search algorithm, then Best-First Search algorithm with three heuristics described in section 2, and A* using the same three heuristics.

The size of the puzzle can be configured by modifying the `ROW_SIZE` and `COL_SIZE` constants of the `Puzzle` class. By default, puzzles have a dimension of 3 by 5 and are represented using a 2D array. Puzzles keep track of their parent to generate the solution path and for the $g(n)$ evaluation in the A* algorithm.

Since we want to run all the algorithms and not to be blocked by a long resolution, the `Main` class has a configurable timeout before it tries a new algorithm, `SECONDS_BEFORE_TIMEOUT`. It is set to 5 seconds by default, but it can be changed.

The solution trace generated by the solvers is available in the "results" folder as .txt files. Their generation and writing is handled by the `FileUtil` class.

1.2 The Solvers Package

The three solvers (Depth-First, Best-First, A*) implement the `Solver` interface that contains the `solve` method. It takes a `Puzzle` and a `Heuristic` as arguments, so that the same solver can be used with different heuristics.

Because the Depth-First Search algorithm can take a long time to execute (see section 2), it comes with a depth limit of 5 by default. It can be changed by modifying the `DEPTH_LIMIT` constant of the `DepthFirstSolver` class; 0 means no depth limit.

The algorithms have been implemented using the class notes:

- Depth-First Search uses an open and a close `Stack of Puzzles`;
- Best-First and A* use a `PriorityQueue` of `Puzzles` based on the heuristic for the open nodes and a `LinkedList` of `Puzzles` for the closed nodes.

For Java to recognize different instances of `Puzzle`, the `equals` and `hashCode` functions have been overridden to support the comparison on 2D arrays.

1.3 The Heuristics Package

The three heuristics (Hamming Distance, Manhattan Distance, Sum of Permutation Inversions) implement the `Heuristic` interface that contains the `evaluate` method. It takes a `Puzzle` as argument.

2 Using Heuristics to Solve Puzzles

In this section, I will be describing the heuristics that I have implemented.

2.1 Hamming Distance

The Hamming Distance is the simplest heuristic I have implemented. In general, the Hamming Distance measures the number of substitutions needed so that a state a would be equal to a state b [9]. For our puzzles, it counts the number of tiles that are misplaced on the board. I had the impression that since it is simple, it would not consume too much resources, and ultimately be faster. This heuristic is admissible since it does not overestimate the cost of reaching the goal.

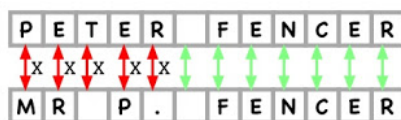


Figure 1: The Hamming Distance between those two names is 5 [5]

2.2 Manhattan Distance

The Manhattan Distance, also known as the Taxicab Geometry, has been inspired by the layout of Manhattan island, in New York City [11]. It calculates the distance from a point a to a point b by adding the distance on the x axis and the distance on the y axis. It is simple to compute, so it should be fast yet more accurate than the Hamming Distance. Many different routes could have the same heuristic value.

The figure 2 shows different routes from a starting state to the goal state. In a study [2], it has been shown that the Manhattan distance performs better than the euclidean distance with a dimension of 20.

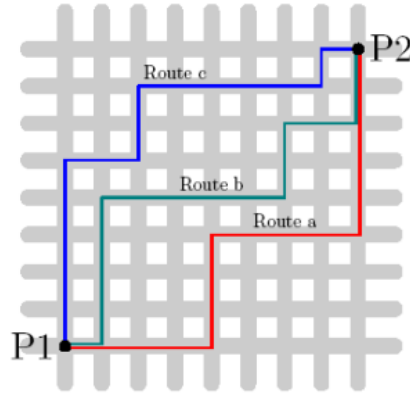


Figure 2: Different routes to reach P2 using the Manhattan Distance [8]

2.3 Sum of Permutation Inversions

In computer science, a sequence is said to have an inversion when two elements do not follow their natural order [10]. For sliding puzzles, the natural order would be a sequence from 0 to $N-1$, where N is the total number of tiles.

Therefore, in the sliding puzzle context, the Sum of Permutation Inversions is a heuristic that flattens the puzzle array, and adds the sum of tiles on the right of each tile that should be placed on its left. The figure 3 shows how permutation inversions are calculated.

Puzzle	1	2	4	5	3	9	8	6	7	10	11	0	Sum
Sum of Permutation Inversions	0	0	1	1	0	3	2	0	0	0	0	0	7

Figure 3: Permutation inversions for a 3 by 4 sliding puzzle

3 Experimenting with algorithms, heuristics and puzzle size

I experimented by changing the size of the puzzles and observing the time taken by the algorithms to solve them, as well as the number of moves they found in the solution path. This is why the puzzle solver is implemented in such a way that it is possible to plug new heuristics to the solvers without changing them, and the puzzle dimensions are dynamic.

I did not think that running the algorithms on a few puzzles could make a valid experiment. Thus, to make my observations more valid, I decided to create a **Benchmark** class that would run all the algorithms on N randomly generated puzzles. One can configure the number of puzzles to be solved by setting the `NUMBER_OF_PUZZLES` and the timeout before another algorithm is tried by setting the `SECONDS_BEFORE_TIMEOUT` static variables in the **Benchmark** class. By default, 100 puzzles will be generated with a 5 seconds timeout. The results are then stored in a CSV (comma-separated values) file. An excerpt of the file is shown in figure 4.

	A	B	C	D	E	F
1	Puzzle Number	Puzzle	DFS-Duration	DFS-Moves	BFS-h1-Duration	BFS-h1-Moves
2		1 9-0-8-2-4-1-10-3-5-11-6-7	67	0	443	63
3		2 5-2-11-4-3-9-8-0-7-1-10-6	29	0	192	43
4		3 9-3-8-4-0-11-6-1-5-7-2-10	21	0	24	41
5		4 9-6-5-3-11-8-4-1-10-0-2-7	14	0	373	66
6		5 2-7-1-11-0-8-3-6-9-10-5-4	18	0	131	70
7		6 7-5-4-9-11-6-10-8-2-1-0-3	16	0	238	71
8		7 10-2-11-6-5-9-7-1-0-8-3-4	19	0	164	47
9		8 7-4-6-2-8-10-11-3-0-5-1-9	14	0	2000	0

Figure 4: Excerpt of the benchmark.csv file

The information stored in the benchmark.csv file are the following:

- The puzzle number, ranging from 1 to `NUMBER_OF_PUZZLES`;
- The puzzle itself, displayed as a flatten array, each tile being separated by dashes;
- The time taken by the Depth-First Search algorithm in milliseconds
- The number of moves of the solution found, if any; if no solution is found with the configured cutoff, the number of moves is 0;

- For the three heuristics of the Best-First Search and A* Search algorithms:
 - The time taken by the Best-First Search algorithm in milliseconds; if the time taken by the algorithm exceeds `SECONDS_BEFORE_TIMEOUT`, it will be `SECONDS_BEFORE_TIMEOUT * 1000`, so 5000 for a 5 seconds timeout;
 - The number of moves of the solution found, if any; if no solution is found before the `SECONDS_BEFORE_TIMEOUT` timeout, the number of moves is 0.

My first experiment consisted in running the benchmark to test 1000 3 by 4 puzzles, with a 2 seconds timeout and a Depth-First Search cutoff of 5. I have found that no puzzle was solved using Depth-First Search. The results are available in table 1.

	Puzzles Solved Before Timeout	Average Duration	Average Number of Moves
BFS-h1	885	328 ms	67
BFS-h2	995	58 ms	60
BFS-h3	1000	59 ms	68
A*-h1	14	534 ms	14
A*-h2	913	258 ms	25
A*-h3	80	547 ms	22

Table 1: Benchmark results for 3 by 4 puzzles with a timeout of 2 seconds

"h1" is the Hamming Distance, "h2" is the Manhattan Distance and "h3" is the Sum of Permutation Inversions. The average duration and the average number of moves are only calculated for puzzles that were solved before the timeout.

One can see that except for the Best-First Search algorithm that uses the Hamming Distance, there is a trade-off between performance and accuracy. Overall, Best-First Search algorithms are fast and manage to solve almost all puzzles within 2 seconds. On the other hand, A* Search algorithms are noticeably slower, with less than 8% of the puzzles solved within 2 seconds for the Sum of Permutation Inversions heuristic and roughly a mere 1% for the Hamming Distance, but they find a solution with on average 45 less moves.

However, it is hard to generalize these conclusions because the number of puzzles actually solved by h1 and h3 remains low. That being said, the best compromise between performance and accuracy seemed to be the A* Search algorithm that uses the Manhattan Distance.

Because the 2 seconds timeout seemed to be an issue, I launched a benchmark of 1000 3 by 4 puzzles with a timeout of 5 seconds to see if there was any difference with the first. Once again, no puzzle was solved using Depth-First Search with a cutoff of 5. The results are available in table 2.

	Puzzles Solved Before Timeout	Average Duration	Average Number of Moves
BFS-h1	939	519 ms	67
BFS-h2	999	130 ms	46
BFS-h3	1000	35 ms	66
A*-h1	33	1904 ms	15
A*-h2	967	380 ms	25
A*-h3	133	1373 ms	23

Table 2: Benchmark results for 3 by 4 puzzles with a timeout of 5 seconds

One can see a small increase in terms of puzzles solved before timeout for A* Search algorithms that use the Hamming Distance and the Sum of Permutation Inversions: 150% of time increase leads to 135% increase of puzzles solved for the Hamming Distance, 66% for the Sum of Permutation Inversions.

The results are not fundamentally different, except for BFS-h2 compared to the previous run that seems to be more effective. The previous observations remain the same: BFS is faster but less precise while A* runs for a long time but finds better solution, with A* using the Manhattan Distance outperforming the others.

I relaunched the benchmark for 3 by 3 puzzles, to see if there was a difference with a smaller grid. The timeout was set to 2 seconds. Once again, no puzzle was solved using Depth-First Search with a cutoff of 5. The results are available in table 3.

	Puzzles Solved Before Timeout	Average Duration	Average Number of Moves
BFS-h1	1000	16 ms	32
BFS-h2	1000	3 ms	21
BFS-h3	1000	6 ms	28
A*-h1	608	481 ms	13
A*-h2	1000	5 ms	15
A*-h3	845	275 ms	15

Table 3: Benchmark results for 3 by 3 puzzles with a timeout of 2 seconds

Because the grid is smaller, all the algorithms have a higher rate of grids solved than earlier. BFS really shines by its speed whereas A* finds better solutions. Like before, solutions found by A* algorithms are roughly twice as good as BFS' ones, but take much longer to be found. One more time, the exception is A* with the Manhattan Distance that is as good as the other A* heuristics in terms of moves but only takes 5 milliseconds on average while solving all the puzzles before the timeout.

As a conclusion, BFS is faster than A* but A* finds better solutions (less moves to make). The best heuristic is the Manhattan Distance, that offers the best trade-off between performance and accuracy.

4 Challenges Encountered

I began by implementing the common "framework" for the project (that is, the `Puzzle` class and the `FileUtil` class). I found challenging to write the .txt files gradually in a results folder. I also had to change the architecture several times before finding one that is flexible enough to add new solvers and new heuristics without duplicating code.

The heuristics themselves were a challenge to implement. To ensure their correctness, I shared the solution trace of some grids with a classmate and we ensured to have the same results.

I had trouble setting up a timeout mechanism that does not break the remaining tests while shutting down the algorithm gracefully. But this was worth it since it allowed me to build a benchmark mechanism and ultimately to exploit the results.

References

- [1] adarshr. *How do I convert a number to a letter in Java?* 2012. URL: <https://stackoverflow.com/a/10813256> (visited on 10/05/2018).
- [2] Charu C. Aggarwal, Alexander Hinneburg, and Daniel A. Keim. “On the Surprising Behavior of Distance Metrics in High Dimensional Space”. In: *Database Theory — ICDT 2001 Lecture Notes in Computer Science* (2001), pp. 420–434. DOI: [10.1007/3-540-44503-x_27](https://doi.org/10.1007/3-540-44503-x_27).
- [3] Arnaldo Pérez Castaño. *Solving Sliding Tiles with Artificial Intelligence (and Some C#)*. 2015. URL: <https://visualstudiomagazine.com/Articles/2015/10/30/Sliding-Tiles-C-Sharp-AI.aspx> (visited on 09/25/2018).
- [4] Java2S. *Clone two dimensional array*. 2010. URL: <http://www.java2s.com/Code/Java/Collections-Data-Structure/clonetwodimensionalarray.htm> (visited on 10/05/2018).
- [5] Jeremy Kubica. *The Hamming Distance Option: Part 3 of Peter and the Postal Service*. 2011. URL: <http://computationaltales.blogspot.com/2011/07/hamming-distance-option-part-3-of-peter.html> (visited on 10/05/2018).
- [6] MadProgrammer. *How to print 2D Arrays to look like a grid/matrix?* 2016. URL: <https://stackoverflow.com/a/34846615> (visited on 10/01/2018).
- [7] martinus. *Number of lines in a file in Java*. 2010. URL: <https://stackoverflow.com/a/453067> (visited on 09/29/2018).
- [8] Lambert R. *Quelle distance choisir en machine learning*. 2018. URL: <http://penseeartificielle.fr/choisir-distance-machine-learning/> (visited on 10/05/2018).
- [9] Wikipedia. *Hamming Distance*. 2018. URL: https://en.wikipedia.org/wiki/Hamming_distance (visited on 10/03/2018).
- [10] Wikipedia. *Inversion (discrete mathematics)*. 2018. URL: [https://en.wikipedia.org/wiki/Inversion_\(discrete_mathematics\)](https://en.wikipedia.org/wiki/Inversion_(discrete_mathematics)) (visited on 10/02/2018).
- [11] Wikipedia. *Taxicab Geometry*. 2018. URL: https://en.wikipedia.org/wiki/Taxicab_geometry (visited on 10/04/2018).

A UML Class Diagram

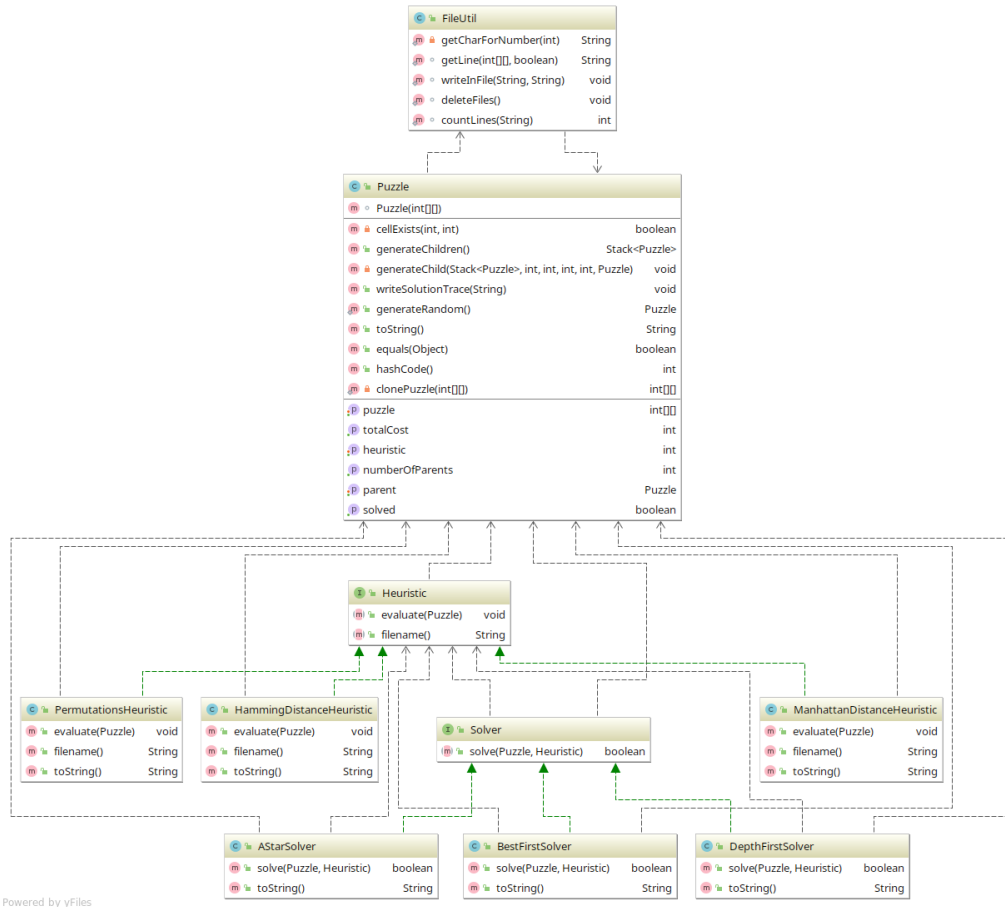


Figure 5: UML Class diagram of the Puzzle solver

B Expectations of Originality

Faculty of Engineering and Computer Science Expectations of Originality

This form sets out the requirements for originality for work submitted by students in the Faculty of Engineering and Computer Science. Submissions such as assignments, lab reports, project reports, computer programs and take-home exams must conform to the requirements stated on this form and to the Academic Code of Conduct. The course outline may stipulate additional requirements for the course.

1. Your submissions must be your own original work. Group submissions must be the original work of the students in the group.
2. Direct quotations must not exceed 5% of the content of a report, must be enclosed in quotation marks, and must be attributed to the source by a numerical reference citation¹. Note that engineering reports rarely contain direct quotations.
3. Material paraphrased or taken from a source must be attributed to the source by a numerical reference citation.
4. Text that is inserted from a web site must be enclosed in quotation marks and attributed to the web site by numerical reference citation.
5. Drawings, diagrams, photos, maps or other visual material taken from a source must be attributed to that source by a numerical reference citation.
6. No part of any assignment, lab report or project report submitted for this course can be submitted for any other course.
7. In preparing your submissions, the work of other past or present students cannot be consulted, used, copied, paraphrased or relied upon in any manner whatsoever.
8. Your submissions must consist entirely of your own or your group's ideas, observations, calculations, information and conclusions, except for statements attributed to sources by numerical citation.
9. Your submissions cannot be edited or revised by any other student.
10. For lab reports, the data must be obtained from your own or your lab group's experimental work.
11. For software, the code must be composed by you or by the group submitting the work, except for code that is attributed to its sources by numerical reference.


You must write one of the following statements on each piece of work that you submit:

For individual work: **"I certify that this submission is my original work and meets the Faculty's Expectations of Originality"**, with your signature, I.D. #, and the date.

For group work: **"We certify that this submission is the original work of members of the group and meets the Faculty's Expectations of Originality"**, with the signatures and I.D. #s of all the team members and the date.

A signed copy of this form must be submitted to the instructor at the beginning of the semester in each course.

I certify that I have read the requirements set out on this form, and that I am aware of these requirements. I certify that all the work I will submit for this course will comply with these requirements and with additional requirements stated in the course outline.

Course Number: COMP6721
Name: Adrien Poupa
Signature: 

Instructor: Dr Leila Kosseim
I.D. # 40059458
Date: October, 5th 2018

¹ Rules for reference citation can be found in "Form and Style" by Patrich MacDonagh and Jack Bordan, fourth edition, May, 2000, available at <http://www.encs.concordia.ca/scs/Forms/Form&Style.pdf>.
Approved by the ENCS Faculty Council February 10, 2012