Ant Colony System for the Travelling Salesman Problem

Matteo Marcuzzo
1207249
University of Padua

Course of Methods and Models for Combinatorial Optimization

A.Y. 2019/2020

Department of Mathematics "Tullio Levi-Civita"
Master's Degree in Computer Science

Table of Contents

1 - Abstract

The aim of this report is to account for the development of two solvers for the Travelling Salesman Problem (TSP) in a real-world scenario.
In particular, the report is a summary of the development and testing phases for both an exact-method solution (implemented via CPLEX) and the Ant Colony System (ACS) approximate solution approach. A side-by-side comparison of the two is given, evaluating both time and accuracy. The heuristic method is then further explored, discussing pros, cons and possible improvements.

2 - Introduction

The below figure (Fig. 1) presents the structure of the code produced in the realization of this project. The figure only presents relevant source code (cmake/make files and data folders have been omitted). The content of each file is fairly self-explicative, but for the sake of clarity:

- All files in *"Instance Generators"* deal with the generation of TSP instances;
- All files in *"Solvers"* deal with the solving of such instances;
- All files in *"Utilities"* deal with any necessity not specific to any such categories.

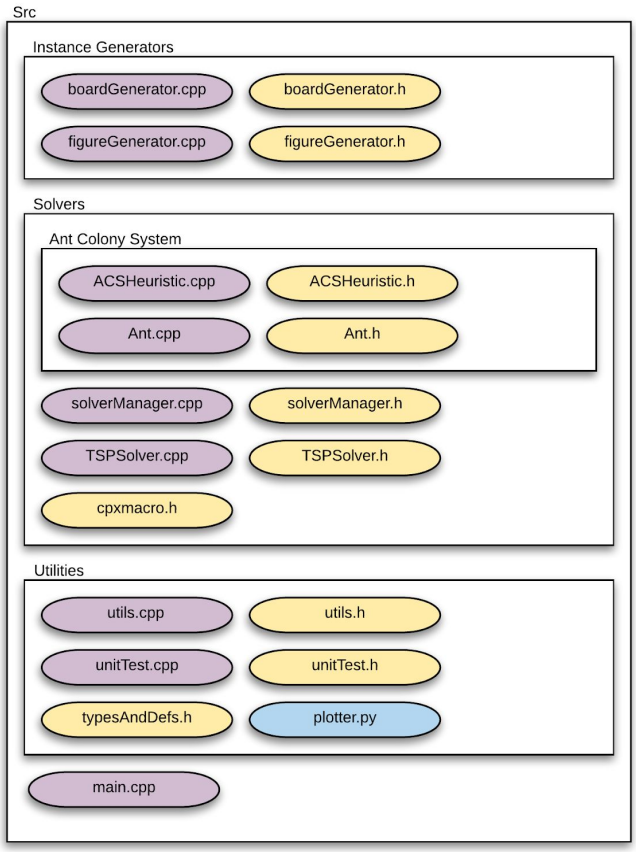

Fig. 1. Structure of source code.

2.1 - Dependencies

The project in itself only depends on the CPLEX libraries. In particular, this project was developed using IBM ILOG CPLEX Optimization Studio 12.8.0. The code was written and run on Windows OS, though it has been tested to work on Linux machines. The *CMakeLists.txt* file handles compilation (on both Windows and Linux), and takes care of finding CPLEX using the script found in *cmake/FindCplex.cmake*. The cmake version used was 3.10. Compilation can be easily achieved by running the cmake script, and then the makefile generated.

The python script used to display grids was tested with python 3.6, though this was not tested on Linux machines.

2.2 - System specifics

The table (Table 1) below contains specifics on the systems used for testing, as well as some information on compilation. **Any table/graph in this report will also specify which machine it was run on (either [1] or [2])**.

Specific	**Machine [1] (laptop)**	**Machine [2] (PC)**
OS	Microsoft Windows 10 Home	Microsoft Windows 10 Enterprise
RAM	16 GB LPDDR4X @3.733 MHz	16 GB DDR4 @2.400MHz
Processor	Intel Core i7-1065G7 (1.30 GHz/3.90 GHz w/ Turbo, 4 core)	AMD Ryzen 5 1600X (3.60 GHz/4 GHz Turbo, 6 core)
Compiler	Visual Studio 2015 (Version 14, x86_amd64)	Visual Studio 2015 (Version 14, amd64)
Flags	-Ox (enable most optimizations)	

Table 1. System specifics of the machines where the project has been tested.

2.3 - Configuration

The program accepts either **1** or **4 command line arguments:**

1. Path to a "*.dat*" file to load and solve (averaged over 10 tests),

or

1. The number of problems to be generated for each size;
2. The number of intervals to test between 0 a given upper bound on the instance size;
3. An upper bound on the number of holes of the biggest instances;
4. The number of tests to run on each individual instance.

In other words, (1) instances are generated for each number of holes; (2) different number sizes between 0 and (3) are generated, and each instance is tested (4) times (average time/objective value is calculated). Instances are generated utilizing the *“regular polygons”* method, explained in the following sections.

The parameters for the Ant Colony System have been set to be the best found through experimentation (see below), though these can be easily modified in the *main.cpp* file. When run, the program runs the exact solver followed by the ACS solver, and compares statistics.

3 - Problem definition

The project aims to resolve instances of the Travelling Salesman Problem (TSP) with two different approaches: an exact-method solution and an approximate heuristic solution. In particular, the work done was aimed at a particular real-life situation of calculating the least time-consuming path for an industrial drill to create holes on electric panels. It is assumed that boring time for each hole is constant, and is therefore ignored.

This scenario is easily representable as a classical instance of the symmetrical TSP in which the N cities are the holes, while the distance between cities is the time required to move between such holes. The problem is therefore modeled with a complete undirected graph, which is represented by a matrix of “times” to move between holes (the matrix is symmetric). This graph representation also bodes well with the chosen heuristic of the *Ant Colony System*, which works on exactly such structure.

4 - Instance Generation

> What is described here relates to code in the “Instance Generators” folder.

The program is able to generate various problem instances, such as to test the effectiveness of the the developed solution methods. Instances can be created in different fashions, though the most appropriate try to simulate the real-world scenario proposed. The main method generates an electric panel where holes are part of various non-intersecting polygons (see the pertaining section). The *Coin Toss* generation method is also presented, as it presents regularities that might be worth exploring.

4.1 - Coin Toss grid

This method generates a regular grid where holes are positioned at the intersections between lines. In order to generate potentially different boards each time, the grid is created so that it may fit twice as many holes as requested. Then, holes are applied at each intersection with probability 0.5; this way, the expected number of holes is exactly what was required. In any case, generation is repeated until the instance contains the required number of holes to avoid unbalanced comparisons. A few examples are shown in Figure 2.

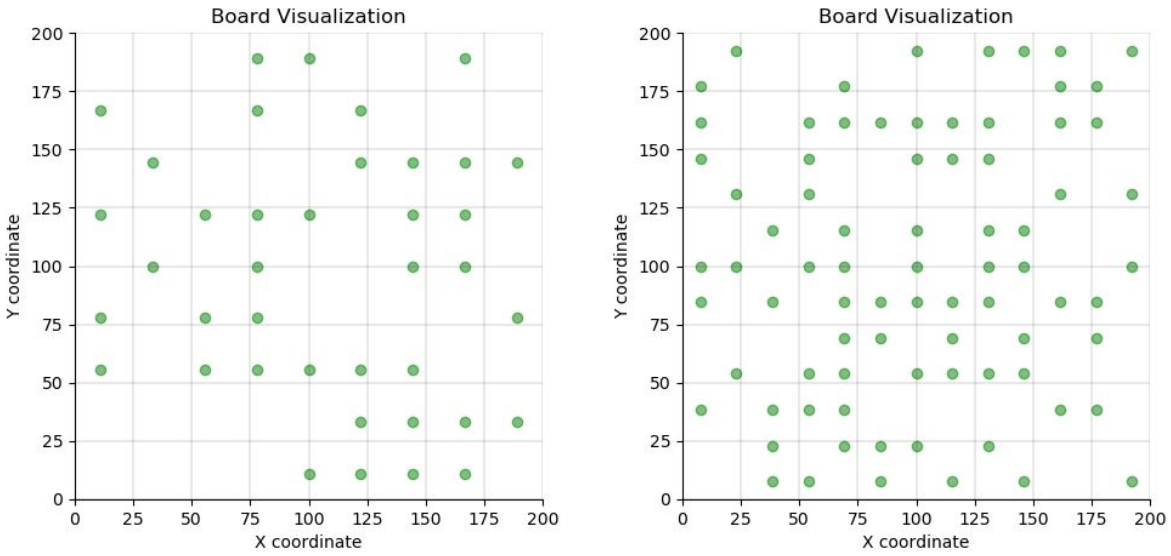

Fig. 2. Visualization of Coin Toss grid (40 and 80 holes).

4.2 - Regular polygons

This method generates random, regular polygons throughout the board without intersection between figures. In order to achieve this, the board is gradually split into squares (hereby referred to as “blocks”), all of which may contain at most one figure.

The board is initially split into four blocks, and figures are gradually added until no free blocks are available or the required number of holes has been positioned. If more figures need to be added but no free block is available, one is deleted (as well as the figure it contains) and split into four smaller ones.

Blocks and figure coordinates are contained in a double ended queue, so that the first block to be removed is always the oldest (and largest) that can be split. New figures and blocks are added to the end of the queue. If only one or two holes are needed to complete the instance, some “filler holes” are added - either a point at the center of the board or a diagonal straight through it.

The geometric figures themselves have a number of vertices that varies between a minimum of three and a maximum equal to the number of holes still required (such number is bounded by the constant *MAX_POLY_SIZE*, which was set to 8). Notably, the figures are also randomly rotated around their center.

Once the board has been completed, the coordinate queue is converted to a weight matrix representation of the graph. Figure 3 provides an example board with 80 holes.

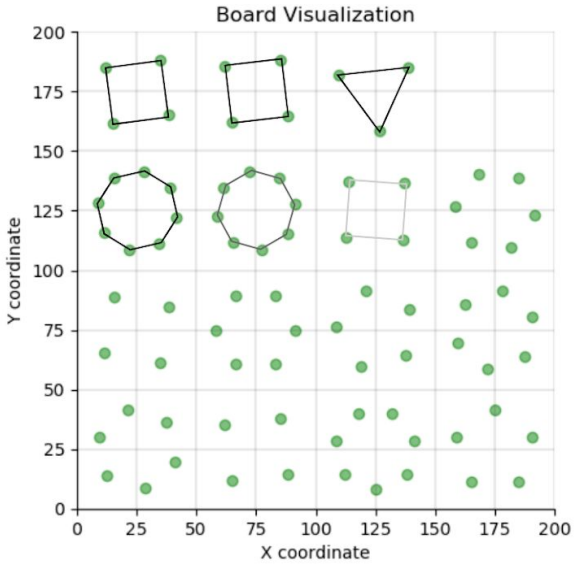

Fig 3. Visualization of geometric board with 80 holes.

4.4 - Existing instances

The program is also able to load and save previous instances of the problem. The ***Utils*** class provides two methods: storing and loading from *.csv* files (list of coordinates) and from *.dat* files (the number of holes followed by the matrix of costs).

5 - Optimization Methods

> What is described here relates to code in the "Solvers" folder.

5.1 - CPLEX Exact method

The CPLEX exact method was developed following specifications given in the first part of the assignment. In particular, this method formulates the TSP as a network flow model.

This solver is handled by the ***TSPSolver*** class, which accepts as parameters the required CPLEX environment variables as well as a "*Data"* structure. This structure contains the weight matrix (implemented as a vector of vectors, renamed "doubleMap") as well as the number of holes of the instance. Instantiation of an object of this class also initializes the index maps of variables (see next paragraph).

The linear programming problem may be set up with the ***initLP*** method, which initializes the variables required by CPLEX as well as storing their indices in the aforementioned maps.
More specifically, the x_{ij} variables (referred to as *unit flow*) represent the amount of flow between hole pairs i and j, while y_{ij} (referred to as *binary flow*) represents whether hole i ships anything to hole j. Variables are added to CPLEX individually, and whenever a

variable is inserted its index is saved in the related index map for easier access. The method also adds the necessary constraints specified by the mathematical model. The constraints are the following (names are arbitrary):

1. **Unit flow constraint:** For any hole k (node 0 excluded), the difference between the sum of incoming and outgoing flows must be one (i.e. the node must receive and keep a unit of flow).
2. **Sum of binary flow constraint:** The sum of binary flows from a hole i to all other holes must be exactly one (only one outgoing edge must be active).
3. **Dual binary flow constraint:** The sum of binary flows to a hole i from all other holes must be exactly one (only one incoming edge must be active).
4. **Flow quantity constraint:** The amount of flow shipped from any i to j is never greater than the maximum unit flow. If the edge is not active, the flow shipped must be 0.

The problem may then be solved through the ***solveLP*** method, which optimizes and times the problem. The *mode* parameter is only needed to set the informativeness level of output messages (silent, descriptive, verbose).

5.2 - Ant Colony System

> *What is described here relates to code in the "Ant Colony System" subfolder.*

The implemented heuristic is that of the *Ant Colony System* (ACS), which in itself is an enhanced version of the classical *Ant Colony Optimization* (ACO) algorithm. In general, this is a class of algorithms which can solve computational problems that can be reduced to finding good paths through graphs.

Summary

The idea of the algorithm is the following. A set of agents (ants) explores the graph in parallel. These agents cooperate indirectly through *pheromones*, which are - roughly speaking - deposited on good edges. Ants explore nodes of the graph following two strategies: a greedy *exploitation strategy*, which selects deterministically the best edge based on its pheromone value as well as an heuristic value (the reciprocal of the distance between nodes), and an *exploration strategy*, which chooses edges stochastically. The selection is not entirely random; higher quality edges have higher probabilities (where quality is determined in the same way as during exploitation). Exploitation and exploration may be seen as intensification and diversification phases, respectively.

Implementation

In terms of implementation, the heuristic is handled by the ***ACSHeuristic*** class. Instances of this class require:

1. A set of parameters (which will be discussed in the next section);
2. A matrix of times (distances) between holes (i.e. the graph representation);

3. (Optional) the best solution for the problem, in order to calculate error rates;
4. A boolean flag to determine whether ants will be run in parallel or sequentially.

The constructor also calls the *init* function, which allocates the required structures and initializes pheromone trails. Pheromones are stored in a matrix, much in the same way as distances are, and all trails are initialized to the same small arbitrary value. A vector of ants is also initialized (namely, *antColony*).

Two functions of this class are worthy of note: *optimize* and *globalPheromoneUpdate*.

The optimization process is fairly simple at this level, as it is mostly handled by the ants. The *optimize* method takes care of placing ants on random starting positions and starting their work in parallel (each ant is launched as a *future*), and ultimately collecting their results. Once all ants have terminated their path, a global update to the pheromone trails is issued. The whole process is iterated as many times as specified in the parameters struct

The aforementioned global update is handled by the *globalPheromoneUpdate* method, which decays all pheromone trails by a small factor while enhancing the trail of the best path that was found in the current iteration.

Agent instances are described by the ***Ant*** class. Other than the heuristic's parameters, each Ant needs access to the pheromone trail matrix as well as the best solution found so far (so that it might compare its own to it and possibly update it). An ant can be launched with the *execute* method, which asks for a starting position and an empty route to fill. Once started, an ant generates a path step by step, following one of two strategies: *exploitation* and *exploration*. These strategies are described in the omonimous functions, which return the index of the hole chosen by the procedure.

As mentioned, the *exploitation strategy* is deterministic, and simply picks the edge with the best product between the pheromone trail value and heuristic value. The *exploration strategy*, on the other hand, generates a set of probabilities over the yet unvisited holes. These probabilities are proportional to the same product used in the exploitation strategy - ideally, this should favor good solution components. Stochastic selection is achieved by gradually summing the calculated probability values until the sum exceeds a randomly generated number between 0 and 1 (uniform probability). Then, the index of the last hole considered is returned.

Lastly, ants perform an important *local pheromone update* (performed by the omonimous function) each time they traverse an edge. This amounts to the agent consuming part of the pheromone trail while walking the graph, and is meant to encourage a "shuffling" of the tours. To put this explicitly, whenever an ant utilizes a good edge, this becomes less desirable. As a consequence, other ants will be encouraged to try different solution components.

5.2.1 - Parameters

The Ant Colony System heuristic uses the following parameters. What follows is a brief overview of such parameters, though a more in-depth explanation of their role is given in the next section.

1. **Number of Ants**
   The number of Ants consists in the number of agents searching for a solution each iteration. Each ant builds a solution and consumes part of the pheromone trail by moving around the graph.
2. **Number of Iterations**
   The number of iterations controls how many times the search is done by each ant. At the start of each iteration, all ants are released and build a solution in a parallel manner. As mentioned, during their search ants consume some pheromone. At the end of each iteration, a global pheromone update is issued; the best solution found by any of the ants is used to enhance the pheromone trail of its components.
3. **Value of Alpha**
   The value of alpha (α) represents the relative importance of the pheromone trail value when considering an edge for its desirability. Pheromone trail values are elevated to this constant during quality evaluation.
4. **Value of Beta**
   The value of beta (β) represents the relative importance of the heuristic value when considering an edge for its desirability; in this case, the heuristic value of an edge is proportional to the reciprocal of the distance between the holes it connects. Heuristic values are elevated to this constant during quality evaluation.
5. **Value of Rho**
   The value of rho (ρ) represents the local evaporation rate of pheromone trails. Whenever ants traverse an edge during their search, they consume part of its pheromone (proportional to this constant) in order to avoid the convergence of all ants to a common solution and favor exploration of different edges.
6. **Value of Omega**
   The value of omega (ω) represents the global evaporation rate of pheromone trails. Whenever an iteration is completed, a global update is issued; such update gradually decays all pheromone trails by a factor proportional to this constant, while also increasing the trail for edges that belong to the optimal solution found so far.
7. **Greediness value**
   Lastly, the greediness value is a factor that determines how likely each ant is to decide between its exploitation (greedy) strategy over its exploration (stochastic) strategy.

5.2.2 - Role and Tuning of parameters

The ACS parameters have been tuned on a generated instance - in particular, a board made of **80 holes** arranged in shapes of **regular polygons.** Each parameter has been calibrated with respect to the initial configuration show in *Table 2*, which is a baseline based on parameters used by the original authors of the heuristic.

Initial Configuration						
Ants	**Iterations**	**Alpha**	**Beta**	**Rho**	**Omega**	**Greediness**
20	100	1	2.5	0.1	0.1	0.9

Table 2. Initial configuration for calibration of the ACS parameters.

Below is the exact solution found by CPLEX, as well as its execution time.

Performance of CPLEX-Exact method [1]	
Objective value	Execution time
1514.15	74.187s

Table 3. Exact method's performance on.

The following tables summarize the tests run for each parameter for different values. Each table is followed by a brief description of how influential the parameter has been proven to be during tests on running times and solution quality, and a likely explanation on why.

The rows highlighted in yellow represent the initial configuration, while the rows highlighted with **bold text represent the parameter chosen as best**. A parameter is chosen not only based on its solution quality, but also on its relative impact on running times - since the point of an approximate method is ultimately to have short running times.

5.2.2.1 - Number of Ants

Number of Ants (mean over 50 runs) [1]			
Number of ants	Objective value	Error	Execution time
10	1661.32	9.269%	0.1934s
20 (Initial)	1651.6	8.684%	0.3073s
40	**1626.67**	**7.165%**	**0.6281s**
50	1630	7.37%	0.7353s
75	1633.49	7.583%	1.235s

100	1632.19	7.503%	1.61s
200	1634.96	7.672%	3.012s
400	1635.06	7.679%	6.471s

Table 4. Calibration of the number of ants.

The role of the **number of ants** is to generate individual solutions to the problem by exploring the graph in a pseudo-random manner, communicating indirectly through the pheromone trail matrix.

Empirically, we see an increase in solution quality up to a certain point, after which the improvement stops. In fact, solutions tend to be slightly worse after the threshold of **40**, which was the chosen value. A possible explanation for this downwards trend might be that too many ants consume too much of the pheromone trail. As a consequence, some of the ants will have no information left to work on, and will not contribute (or worse, be detrimental) to the final result. One might consider a higher number of ants in bigger instances, though increasing the number of iterations is likely a better choice.

Number of Ants - Overall:

- **Impact on solution quality**: *medium/high*
- **Impact on time performance**: *high*

5.2.2.2 - Number of Iterations

Number of Iterations (mean over 50 runs) [1]			
Number of Iterations	Objective value	Error	Execution time
1	1811.1	17.86%	0.00314s
50	1664.53	9.462%	0.1533s
100 (initial)	1651.62	8.685%	0.3138s
200	1635.09	7.681%	0.6698s
400	1623.13	6.947%	1.445s
800	**1611.07**	**6.202%**	**2.951s**
1000	1610.73	6.182%	3.714s
1200	1605.41	5.851%	4.306s
1600	1603.50	5.732%	5.848s
2400	1597.25	5.342%	8.643s

Table 5. Calibration of the number of iterations.

The **number of iterations** controls how many times each ant performs its search, as well as how many times a global update of the pheromone values is issued.
Empirically, we observe a steady improvement in solution quality by increasing more and more the number of iterations. This is to be expected; as more paths are explored and global updates are issued, the best solution components become more desirable over time as their pheromone value increases.

Note that the number of iterations chosen is not necessarily the best, and a different choice might be entirely justifiable based on time or quality constraints. The chosen value was that of **800**, which roughly corresponds to the empirical best quality-to-time improvement (though the choice was arbitrary).

Number of Iterations - Overall:

- **Impact on solution quality**: *high*
- **Impact on time performance**: *high*

5.2.2.3 - Value of Alpha

Value of Alpha, importance of pheromone trail (mean over 50 runs) [1]			
α	Objective value	Error	Execution time
0.1	1692.82	11.14%	0.5619s
0.5	1660.17	9.2%	0.5349s
0.8	1649.84	8.577%	0.5049s
1.0 (initial)	**1642.79**	**8.149%**	**0.3071s**
1.5	1647.97	8.464%	0.526s
2.5	1649.01	8.527%	0.5902s
4.0	1655.88	8.942%	0.6025s
8.0	1650.79	8.635%	0.5565s

Table 6. Calibration of the value of α.

The α parameter represents the **importance of the pheromone trail value** when calculating an edge's desirability. Though results show that exaggerating this value does not improve solution quality, values smaller than 1 show quite clearly that the pheromone trail factor is important in the search for better solutions.

Some authors omit this parameter altogether (in other words, fix this at 1), and this seems to be the empirical best solution in this case as well. The faster run time with $\alpha =$ **1.0** is likely to

be attributed to optimization procedures from the compiler that eliminate the power operation, as it has no effect.

Value of Alpha - Overall:

- **Impact on solution quality**: *low/medium*
- **Impact on time performance**: *none/low*

5.2.2.4 - Value of Beta

Value of Beta, importance of heuristic value (mean over 50 runs) [1]			
β	Objective value	Error	Execution time
0.1	1929.58	24.13%	0.3056s
0.5	1760.21	15.03%	0.3145s
1.0	1697.56	11.42%	0.1119s
2.5 (initial)	1647.92	8.461%	0.3109s
3.0	1643.13	8.17%	0.3052s
4.0	**1636.21**	**7.749%**	**0.3074s**
5.0	1635.06	7.679%	0.3086s3
6.0	1640.1	7.986%	0.3291s
8.0	1638.15	7.868%	0.3162s

Table 7. Calibration of the value of β**.**

The β parameter represents the **importance of the heuristic value** when calculating an edge's desirability. The heuristic value of an edge is based on the reciprocal of the distance between the two holes it connects. As it turns out, enhancing the importance of this factor can improve results in a significant manner. The chosen value was that of **4.0**, both because it provided one of the best results and because it was the smaller value that did so.

While some higher values provided similar improvements, enhancing this factor too much runs into the risk of nullifying the importance of the other components of the algorithm (i.e. the pheromone trail). Therefore, this is an additional reason why this value was preferred.

Value of Beta - Overall:

- **Impact on solution quality**: *high*
- **Impact on time performance**: *low*

5.2.2.5 - Value of Rho

Value of Rho, local evaporation rate (mean over 500 runs) [1]			
ρ	Objective value	Error	Execution time
0.05	1655.8	8.937%	0.393s
0.1 (initial)	1646.15	8.354%	0.3826s
0.2	1637.49	7.827%	0.3769s
0.4	1628.3	7.265%	0.3905s
0.6	1627.71	7.229%	0.3848s
0.8	**1624.98**	**7.061%**	**0.3771s**
0.95	1625.35	7.084%	0.3843s

Table 8. Calibration of the value of ρ.

The value of ρ represents the **local evaporation rate of pheromone trails**. Local update of pheromones is applied in order to dissuade following solutions from using the same components in the same order; as such, it comes to no surprise that a higher value obtains better performance. The role of the local update rule is to "shuffle the tours", so that early holes in one ant's tour may be explored later in other ants' tours. Since using a particular edge makes it less desirable (since pheromone is "consumed"), this guarantees ants never converge to a common path.

Note that this parameter was tuned over a higher number of trials, since results were erratic and a mean over 50 values could not find an appropriate candidate. The chosen value was therefore that of **0.8**, such as to favor exploration of new edges while not nullifying the value of pheromone trails altogether.

Value of Rho - Overall:

- **Impact on solution quality**: *medium*
- **Impact on time performance**: *none*

5.2.2.6 - Value of Omega

Value of Omega, global evaporation rate (mean over 50 runs) [1]			
ω	Objective value	Error	Execution time
0.001	1698.94	11.5%	0.322s
0.01	1689.94	10.97%	0.3042s

0.05	1660.6	9.226%	0.3164s
0.1 (initial)	**1646.89**	**8.399%**	**0.3248s**
0.2	1650.86	8.639%	0.3012s
0.4	1651.9	8.702%	0.3138s
0.6	1655.98	8.948%	0.3209s
0.8	1673.93	10.02%	0.3017s
0.95	1727.86	13.18%	0.3062s

Table 9. Calibration of the value of ω**.**

The value of ω represents the **global evaporation rate of pheromone trails**. Such value is used during the global update phase of pheromone trails, and controls how much of the pheromone trail decays globally. Empirical results show that a smaller decay factor favors slightly better solution components, and as such a value of **0.1** was chosen.

Tests seem to point out that having all pheromone evaporate gives much worse results (as is to be expected), but no evaporation altogether also lowers the quality of the solutions - therefore suggesting that global evaporation is an important factor in helping towards better solutions.

Value of Omega - Overall:

- **Impact on solution quality**: *medium*
- **Impact on time performance**: *none*

5.2.2.7 - Greediness value

Greediness value - exploitation over exploration (mean over 50 runs) [1]			
Greediness	Objective value	Error	Execution time
0	2157.79	35.06%	2.904s
0.5	1819.2	18.3%	1.388s
0.75	1702.47	11.71%	0.6723s
0.9 (initial)	1647.2	8.418%	0.3198s
0.95*	***1638.36***	***7.88%***	***0.2471s***
*0.99**	*1633.12*	*7.56%*	*0.1438s*
*1**	*1629.67*	*7.349%*	*0.115s*

Table 8. Calibration of the greediness value**.**

The **greediness factor value** (sometimes referred to as *Q_0*) determines whether an ant is more likely to exploit the best edges available (high pheromone and lower edge weight) or decide probabilistically between all available remaining edges (exploration strategy). A value closer to 1 means the exploitation strategy is chosen more often.

Results show that a higher greediness value obtains better results overall. However, exploitation strategies are inherently deterministic, which can be detrimental to the final result - since the algorithm then relies on the nondeterministic initial positioning of ants.
The values of 0.95, 0.99 and 1.0 were *evaluated over a pool of 1000 runs**, rather than 50. While a greediness value of 1 seems to provide better results, in order to support exploration of different solutions the value of **0.95** was chosen. This somewhat arbitrary choice was taken in hopes this might provide better solutions on different instances (i.e. this choice encourages diversification).

It's also worth noting that exploration strategies are not only non deterministic but also more computationally expensive. Purely exploratory strategies display wild swings in their results, and as such their mean results are substantially worse.

Greediness value - Overall:

- **Impact on solution quality**: *very high*
- **Impact on time performance**: very *high*

5.2.2.8 - Summary

Table 9 summarises the best parameters found during the calibration phase of the algorithm.

Best configuration found [1]						
Ants	**Iterations**	**Alpha**	**Beta**	**Rho**	**Omega**	**Greediness**
40	800	1.0	4.0	0.8	0.1	0.95

Table 9. Best configuration found.

A few final remarks:

- It is likely that some instances could obtain better result by further tweaking some of the parameters;
- A better approach would be to evaluate parameters not only individually but also based on their mutual influence;
- Values were not only chosen based on their objective value performance, but also their time performance. As such, different configurations might be used if different time constraints were in place.
- If any value were to be tweaked first, the safer choice would be to increase the number of iterations.

6 - Tests results

The following section presents test results obtained by the two methods. Instances are compared based on their solution quality performance (objective value found) and time performance. The minimum, maximum, mean and standard deviation value of these metrics are provided. The ACS heuristic was run using the optimal configuration shown in the previous section.

Note that, since the objective function value has marginal significance by itself when considering generated instances, solution quality is gaged with relative error percentages (i.e. how far the solution found was from the optimal one).

6.1 - TSP instances

6.1.1 - TSP12

Table 10 reports the comparison of objective function values for the *tsp12.dat* instance. As this instance is quite small, the ACS algorithm performs quite quell, often finding the global optimum as a solution. In any case, even when the algorithm doesn't find the best solution, it is not too far from it (on average, the error rate obtained is roughly 0.5%). The low standard deviation value suggests that the algorithm is quite stable in its results.

TSP12 - Objective function results (10 runs) [1]				
Method	Mean obj.	Obj. stdev	Min obj.	Max obj.
CPLEX	66.4	0	66.4	66.4
ACS	66.76	0.5713	66.4	67.9

Table 10. Solution quality comparison for TSP12.

Table 11 reports the comparison of time performance between the two methods. The average time is pretty much identical, with a slight advantage towards the heuristic solver. Given the marginal difference, it is clear that one should prefer the exact method in this case, as it guarantees the best solution.

TSP12 - Time performance results (10 runs) [1]				
Method	Mean time (s)	Time stdev (s)	Min time (s)	Max time (s)
CPLEX	0.0821	0.01925	0.062	0.118
ACS	0.0735	0.001432	0.071	0.076

Table 11. Time performance comparison for TSP12.

6.1.2 - TSP60

Table 12 reports the comparison of objective function values for the *tsp60.dat* instance. This instance is considerably larger than the previous, and therefore the two methods begin to differ in their performances. The *ACS* algorithm is slightly inaccurate, with a mean relative error of roughly 2.9%. The standard deviation on this error is fairly low, amounting to approximately 0.7%.

TSP60 - Objective function results (10 runs) [1]				
Method	Mean obj.	Obj. stdev	Min obj.	Max obj.
CPLEX	629.8	0	629.8	629.8
ACS	648.9	5.019	639.6	652.9

Table 12. Solution quality comparison for TSP60.

Table 13, however, showcases a much better time performance by the heuristic method, which is, on average, up to six times faster. Furthermore, this method proves much more stable than the exact method, which has a considerable variance in its time performance. The *ACS* method, on the other hand, has a negligible standard deviation.

TSP60 - Time performance results (10 runs) [1]				
Method	Mean time (s)	Time stdev (s)	Min time (s)	Max time (s)
CPLEX	11.1761	1.469	9.279	14.725
ACS	1.769	0.03934	1.711	1.827

Table 13. Time performance comparison for TSP60.

6.2 - Generated instances

The following tests were performed with the following criteria:

- Each number of holes (instance size) had 10 different configurations;
- Each configuration was tested 10 times, and the average of results was taken.

6.2.1 - Coin Toss grid

The following graphs showcase a comparison between the exact and heuristic method while solving instances generated by the *Coin Toss grid* method.

Time Performance

Figures 4 and 5 display the relative performances of CPLEX and ACS. Two things are immediately evident upon close examination. First off, upon surpassing instance sizes of 60 and above, the time performance for CPLEX becomes substantially worse in a much faster

fashion than the heuristic method. Notably, some particularly complex configurations for the largest instances can cause the exact method to take up to more than a minute.

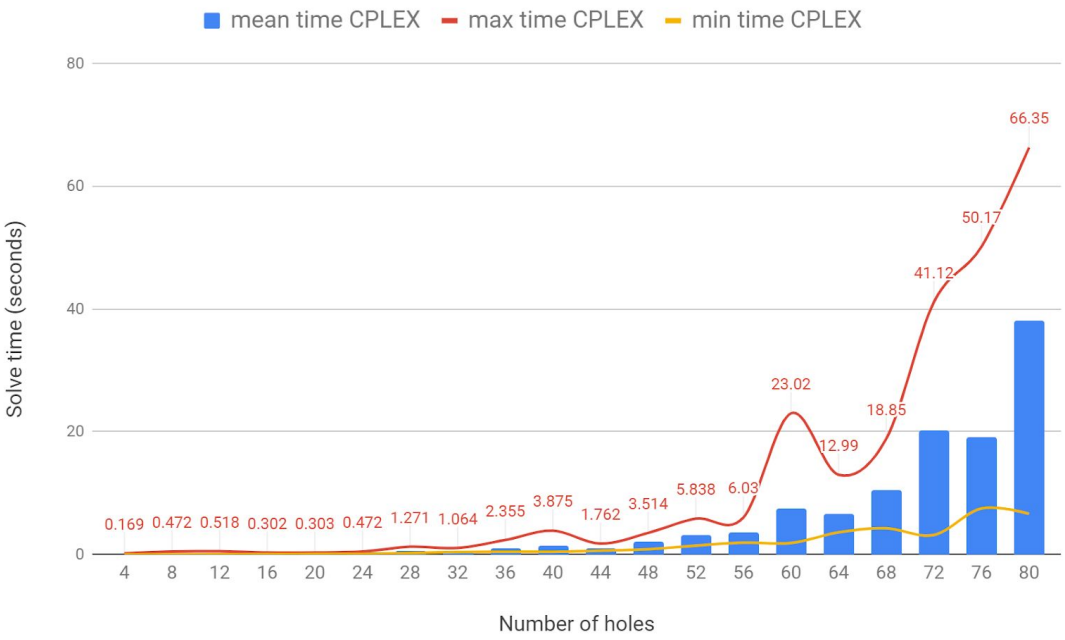

Fig 4. [2] Coin Toss time performance curves for CPLEX, smoothed.

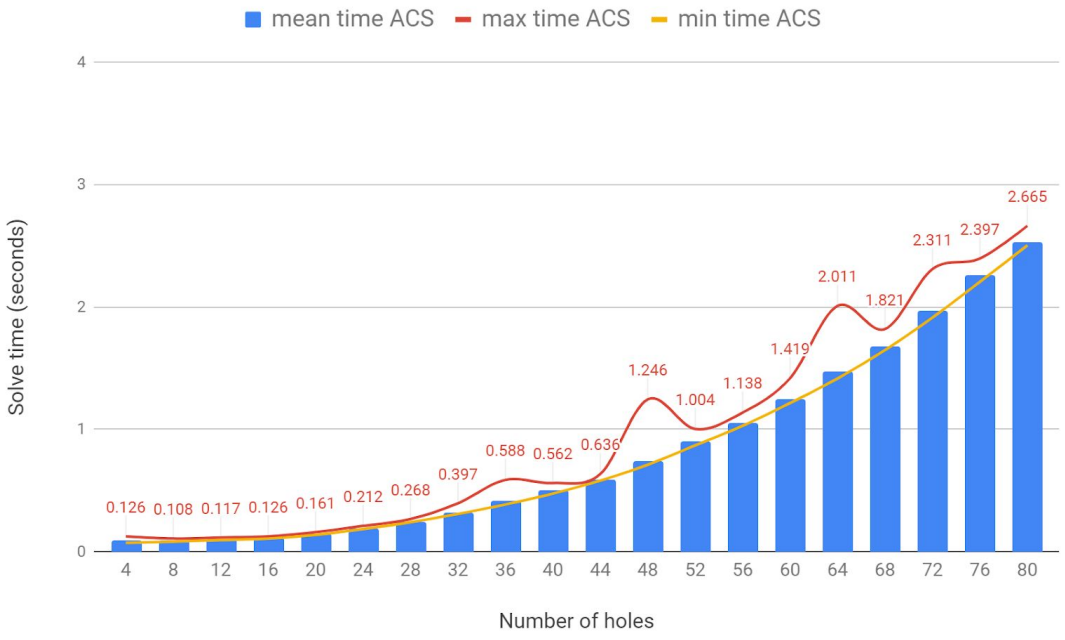

Fig 5. [2] Coin Toss time performance for ACS, smoothed.

This is also related to the second observation: since different instances with the same number of holes require vastly dissimilar times to optimize, the standard deviation for the exact method becomes quite considerable. Figure 6 (logarithmic in scale) showcases how the variance in time performance grows significantly faster when compared to the heuristic method. In other words, the ACS solver is far more stable in its solve times, as well as increasing in mean value much more slowly (Figure 5).

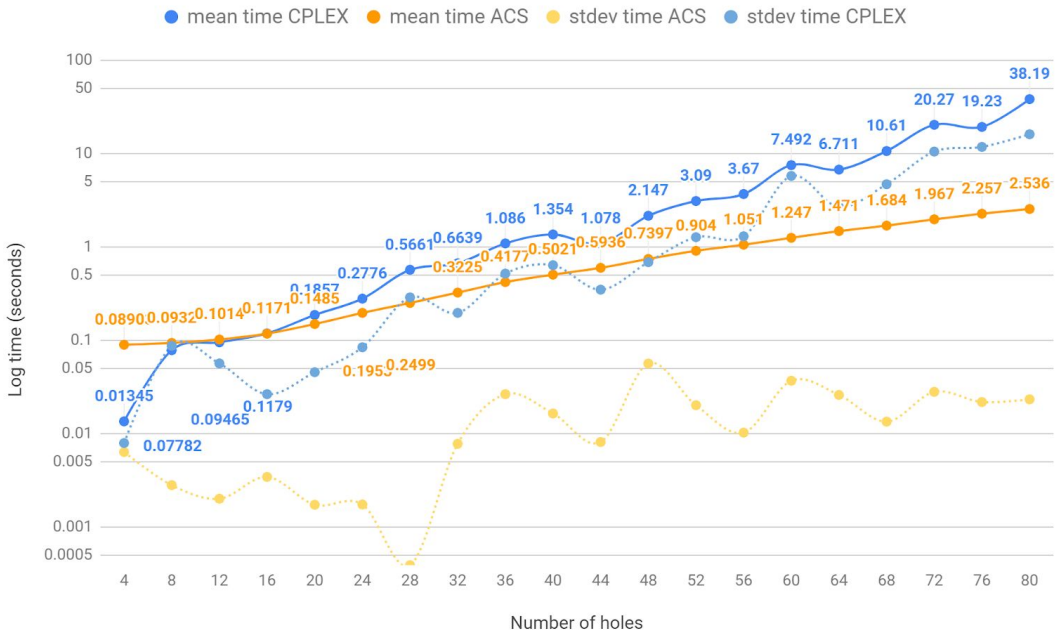

Fig 6. [2] Coin Toss stability comparison of time performance between ACS and CPLEX.

Quality of results

Figure 7 showcases the minimum, maximum and mean error rate of the approximate method when related to the exact solution. The results are fairly decent: the heuristic can often times find the global optima up to instances of size 44. Even on larger instances, the mean value surpasses the optimal objective value by at most 5%. In some unfortunate cases the heuristic can reach an error rate of almost 10%, though it is quite rare.

Figure 8 displays the standard deviation of such error rates. The approximate algorithm proves to be mostly stable, with a standard deviation that never surpasses the value of 2%.

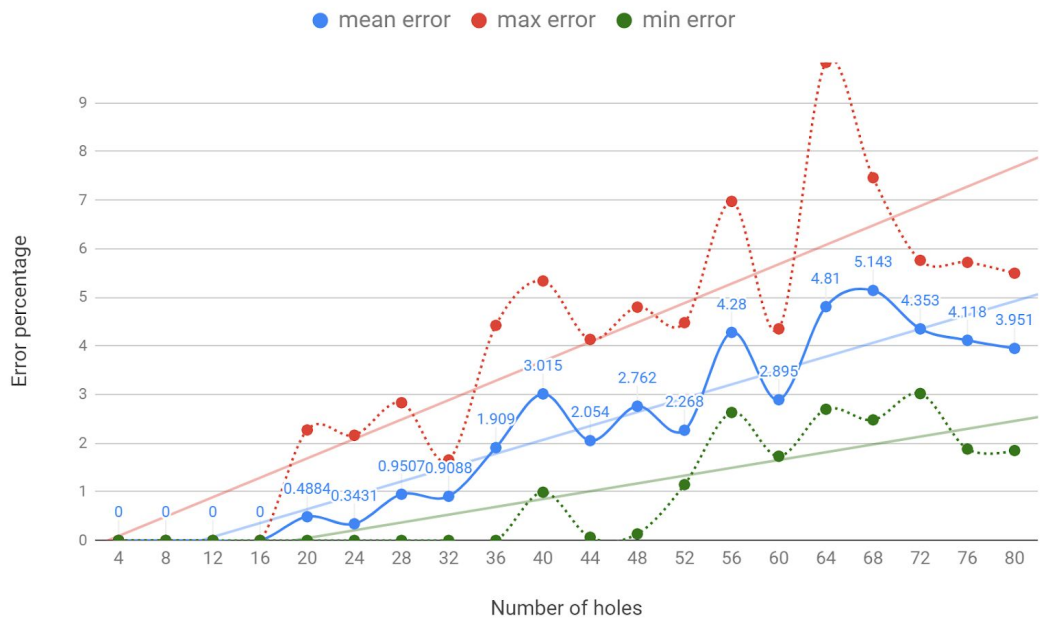

Fig 7. [2] Coin Toss solution error for ACS, smoothed.

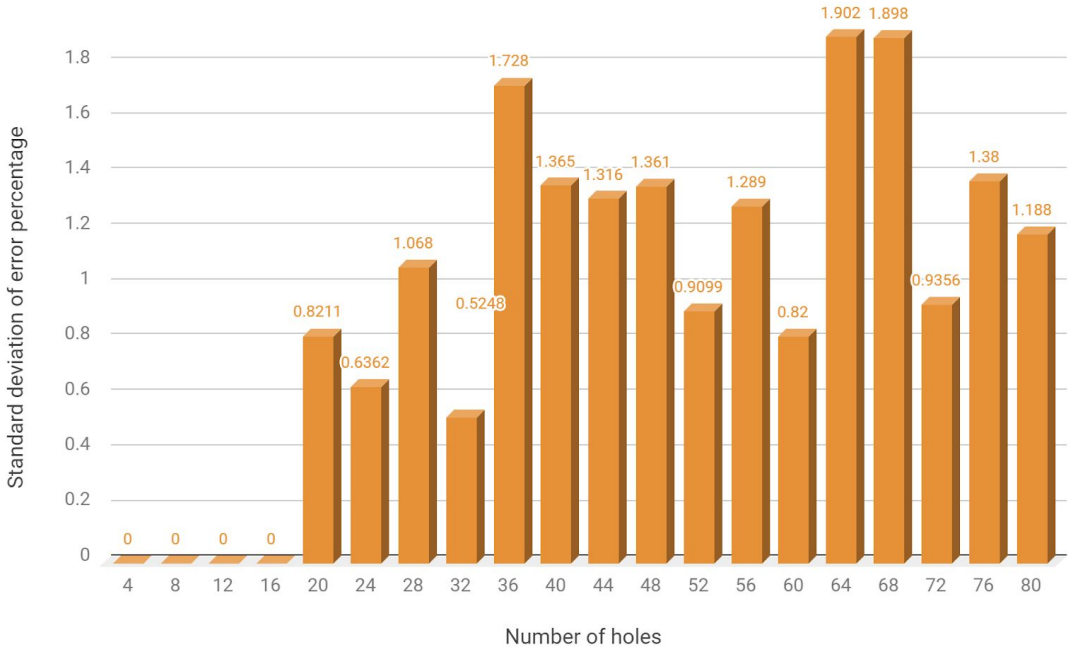

Fig 8. [2] Coin Toss stability in error percentage for the ACS algorithm.

6.2.2 - Regular polygons

The following graphs showcase the performances of the two methods when applied to boards that contain holes arranged in various polygonal shapes.

Time Performance

Figures 9 and 10 display the solve time for CPLEX and ACS for various instance sizes. As a reminder, the performance on each instance is averaged over 10 different configurations with the same number of holes. The results can be considered similar to the ones obtained in the evaluation of the Coin Toss configuration; upon reaching instance sizes of 50 and above, the exact method starts to lose footing and is much slower than the heuristic method, as is to be expected.

Regular polygons are supposed to give a semblance of regularity to the board, but in a much less predictable fashion when compared to the previously considered grid. As such, it comes to no surprise that CPLEX finds these instances on average slightly more difficult to solve. The solve time arguably grows faster than it did in the previously considered examples, with some particularly difficult instances taking more than 72 seconds to solve.

In contrast, the ACS heuristic performs much in the same way as it did previously, only ever surpassing 3 seconds of computation in unfortunate cases.

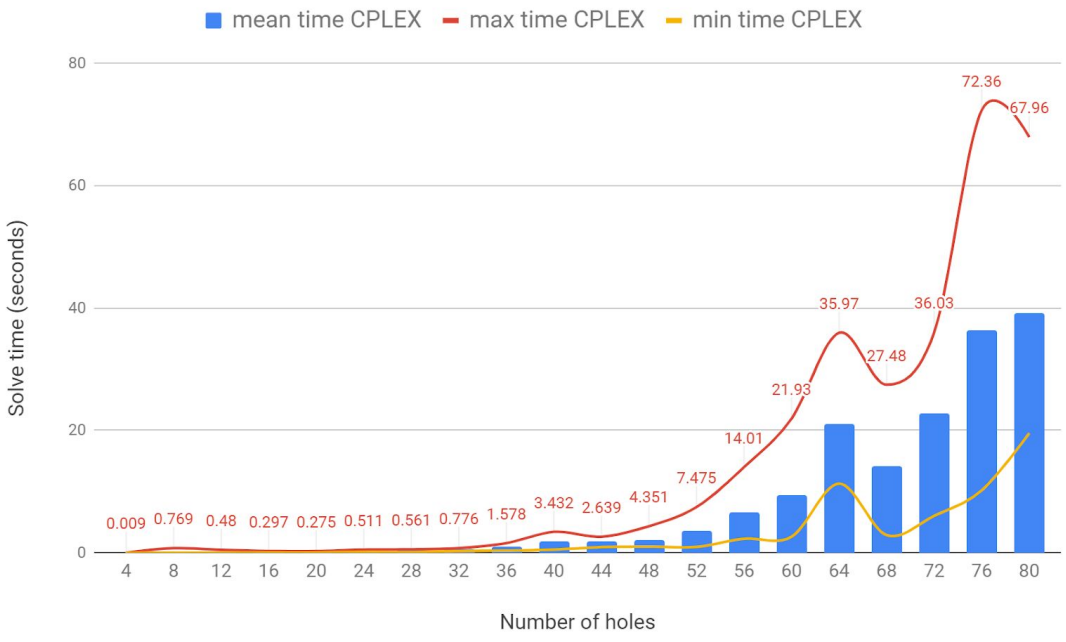

Fig 9. [2] Polygons time performance curves for CPLEX, smoothed.

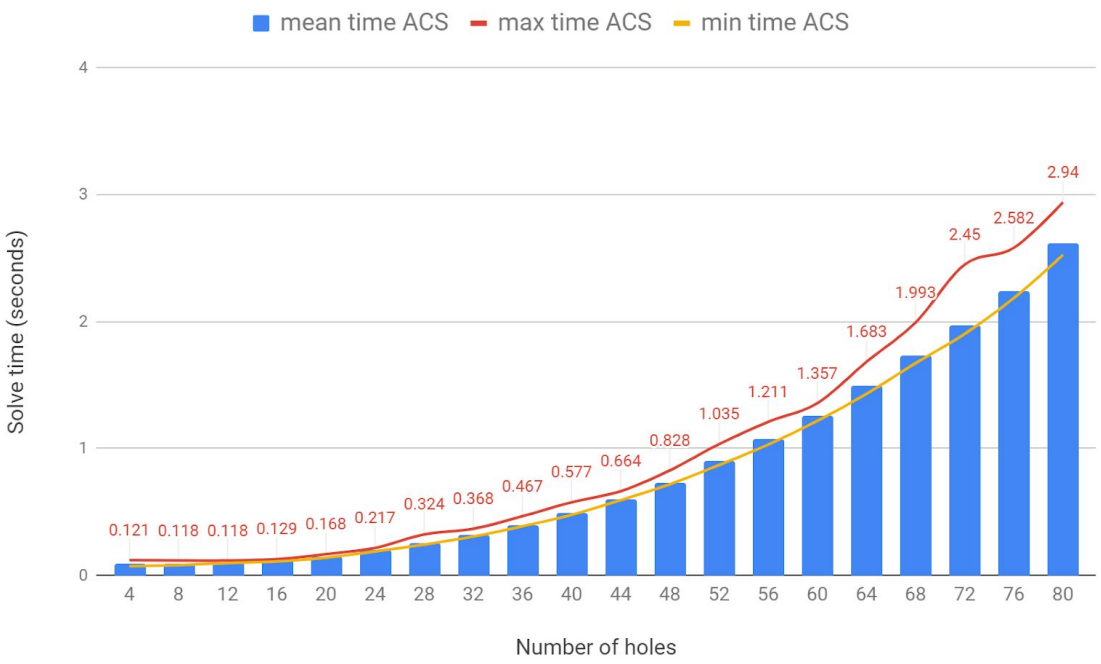

Fig 10. [2] Polygons time performance for ACS, smoothed.

Figure 11 showcases the stability of time performances by the two methods. Again, CPLEX proves to be much less stable, as a particularly difficult instance might prove very difficult or very easy depending on the values of the distances matrix. ACS, on the other hand, still proves to be quite reliable.

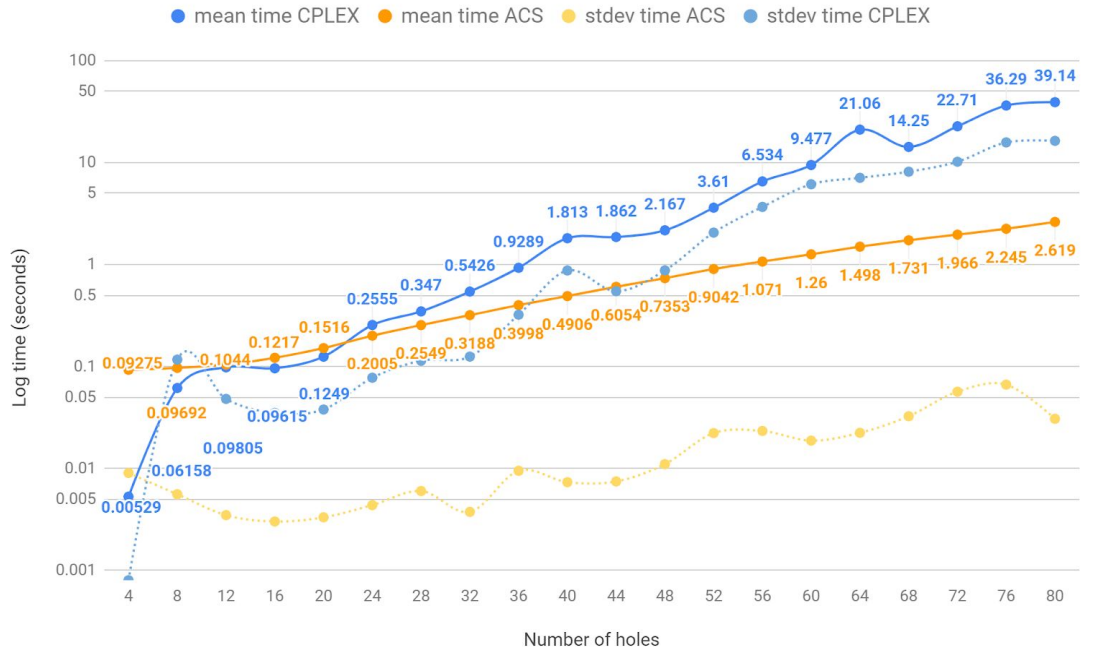

Fig 11. [2] Polygons stability comparison of time performance between ACS and CPLEX.

Quality of results

Figure 12 shows error trends for the approximate method. The results are similar to the ones obtained for the Coin Toss grid, if not slightly better. Error rates are quite stable (as shown by figure 13) and are often considerably lower than 2%. Even in the worst cases, the maximum error on record was just above 6%. One might worry about the increasing error rates of bigger and bigger instances; that issue will be discussed in the final part of the report.

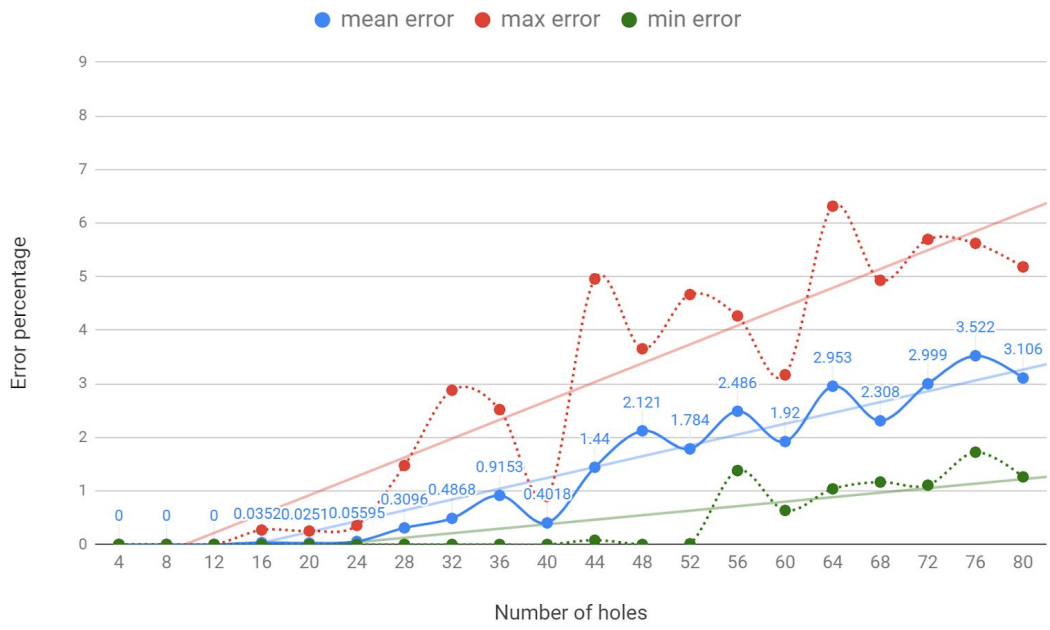

Fig 12. [2] Polygons solution error for ACS, smoothed.

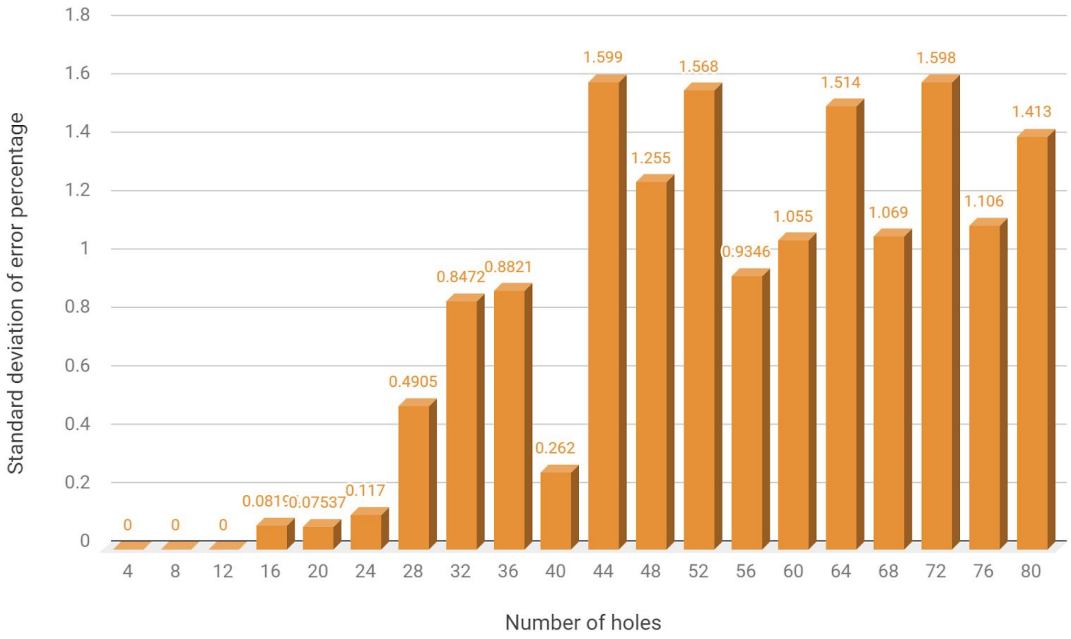

Fig 13. [2] Polygons stability in error percentage for the ACS algorithm.

7 - Conclusions

7.1 - Observations

The analysis of the results showcased previously suggests that the exact method is most certainly preferable for instances of size up to 40 holes (in particular, see figure 4 and 9). This assertion is justified by the fact that in this range CPLEX is not only fast but, more importantly, stable - as shown by low standard deviation on time performances. One might argue to extend this suggestion to instances of size up to 50; though the exact solver starts to prove less stable in these situations, this might still be an acceptable compromise in order to obtain the best solution.

When the board has strictly more than 50 holes, however, the choice becomes less obvious. As the size increases, CPLEX not only becomes much slower on average, but also displays wild swings with regards to its solve time - suggesting that some instances are quite harder than others to solve, even for the same number of holes. This is reflected by figure 6 and 11, which highlights the ever increasing standard deviation of the exact method's timing in bigger and bigger boards.

The heuristic, on the other hand, demonstrates a very low standard deviation in time performances, as well as much faster net performance. On larger instances, results can be considered decent (error rates usually remain below 5%, as shown in figure 7 and 12).
As mentioned, one might worry about the projection of error rates on larger instances. However, an easy solution to this upwards trend would be to increase the number of iterations for larger instances. Since the ACS procedure is much faster in these cases, an increase in the number of iterations is easily justifiable and would still provide preferable time performances to the exact method.

As to what should be used in the real-world scenario proposed, the answer depends on the exact situation. In any and all cases, if boards never contain more than 40 holes, one should certainly utilize the exact method, as it is fast and stable. In the range between 40 and 50 holes one might consider an heuristic approach, though it is probably still best to take some risks to find the best solution.

Finally, larger boards should prefer a heuristic solution, though this might not always be the case. If the company were to produce standardized electrical panels which always have the same configuration, then it might be worth suffering the immediate time loss of utilizing an exact method as we expect to gradually regain it once the calculation is not needed anymore. On the other hand, if configurations are changed often, and the drilling machines require an on-the-go analysis of particular boards, then an approximate solution approach is much preferable, if not strictly necessary.

7.2 - Possible improvements

Overall, the Ant Colony System heuristic achieves decent results. Its best quality is most likely its speed, which is granted by the fact that each ant is run as a parallel process. As a matter of fact, whenever the hardware does not support multithreading, the algorithm's performance suffers quite a bit. Nonetheless, it is not unfair to assume that, in real life scenarios, a company should invest on better performing machines.

On the other hand, the quality of solutions found could certainly be improved. While the error rates are not abysmal, the approximation is still not as good as the one provided by other ad-hoc algorithms for the TSP.

A first approach to improve this algorithm might be a more precise tuning of its parameters, as was mentioned in section *5.2.2.8*. While the method presented in this report has certainly improved the quality of solutions, it does not take into consideration the mutual influence that parameters have on each other. As a consequence, a different combination of parameters might have performed better than the one proposed. The field of artificial intelligence has devised many different methods of hyperparameter-tuning; an example would be the *grid search* approach, which exhaustively explores the parameter space to find good combinations.

A different kind of improvement would be to attempt some customizations of the heuristic itself. As an example, one could allow more than one ant to perform the global pheromone update (e.g. allow the k-best ants to deposit some pheromone on their solution components). The same might be done on the flip-side: the worse solutions could be used to penalize the pheromone trail values on their components.

The last proposed improvement would most likely be the most beneficial to this method, and consists in the integration of a local optimizer. ACS is a tour construction heuristic - a procedure which creates a set of feasible solutions at each iteration. The role of the local optimizer would be to move solutions towards local optima (utilizing local search methods such as 2-opt, 3-opt or lin-kernighan). The simpler approach would be to apply the local optimizer to the final solution found, though it might be interesting to apply a local search procedure at the end of each iteration. While clearly more costly, this would emphasize the role of the global update of pheromone trails - as the solution components which receive updates would be more refined.