UNIVERSITÀ DI PISA

Information Engineering Department

Master's Degree in Robotics and Automation Engineering

Ant Colony Optimization for the Dubins Traveling Salesman Problem

Supervisor:

Prof. Lucia Pallottino

Candidate:

Ilaria Martelli

Academic Year 2021/2022

Contents

1 Introduction

Usually, when one thinks of a robot or any robotic task, the first thing that comes to mind is a single operator with an autonomous processing unit that allows it to perform the assigned job. There is however a less intuitive branch of robotics that offers a different approach: distributed robotic systems [44]. Within these systems, it's studied how multiple robots work and interact with each other, and how working together improves effectiveness and can lead to results that would not be possible for an isolated individual.

Contrary to centralized intelligence, where there is a leader that decides and others that follow (more computationally expensive and less resilient to failure and damage), in *distributed systems* all the individuals (or *agents*) are simpler and work together to organize, plan, and solve problems. This method allows for the agents to be cheaper and easily replaced in case of breakdowns, improving the robustness and reliability of the system. Examples of decentralized, multi-agent, self-organized systems are common in nature, such as schools of fish, flocks of birds, colonies of insects like ants, termites, bees, etc.

Figure 1: Drone show in Sydney: each individual drone works in unison with all the others to remain in formation.

This same reasoning can be applied to abstract concepts instead of physical robots, and when it's referred to algorithms it's called *distributed intelligence* [6]. This field concerns the study of heuristics to solve complex learning, planning, and decision making problems.

One of the possible types of behavior is the *collective behavior*. It can be decomposed in three fundamental classifications:

1. the agents of the system work together to achieve a common goal;
2. the agents are not "*aware*", so they retain minimum information from the environment, can't formulate complex strategies and follow simple predetermined rules, but in turn their simplicity leads to less computationally-heavy individual input;
3. the agents are not competitive and will (indirectly) help each other using the same directives.

This kind of approach is also called *swarm intelligence* [4], since it mimics the behavior typically observed in colonies of eusocial insects.

The strength of the collective behavior is found when the system is evaluated in its entirety, and is called *emergent behavior*: homogeneous agents (with simple identical control laws) that have limited capabilities if taken on their own, can generate intelligent behavior if combined together, and will produce the desired result as a byproduct of their spontaneous interaction in a wider whole.

Figure 2: Army ants as they make a bridge with their own bodies, a display of collective behavior.

1.1 Content analysis

In the world of robotics we can find a wide variety of autonomous robots, of any shape or form and for any locomotion medium; terrestrial self-driving cars, flying drones, and submarine vehicles are some of the most widespread types. But as different as they may seem at first glance, they all share a common objective in path planning and routing: they all need a computing structure that enables them to find a feasible (and possibly *optimal*) path to follow that will bring them from their starting point to their target point.
Also, apart from particular cases of iper-mobility, these kinds of motorized robots usually have a specified turning radius based on speed, wheels dimension, wingspan or general shape of the robot, so the routes they can pursue are limited by the kinematics of the system.

One specific branch of path planning problems focuses on searching a continuous path that can touch a specified number of points of interest, without stopping to reposition, and trying to achieve the shortest tour: this particular problem is called the "Traveling Salesman Problem", and for physical, constrained systems as those listed above, specifically the "Dubins" variation. As we'll see in Chapter 1.2, this problem is useful in many applications like patrolling, mapping and delivery.

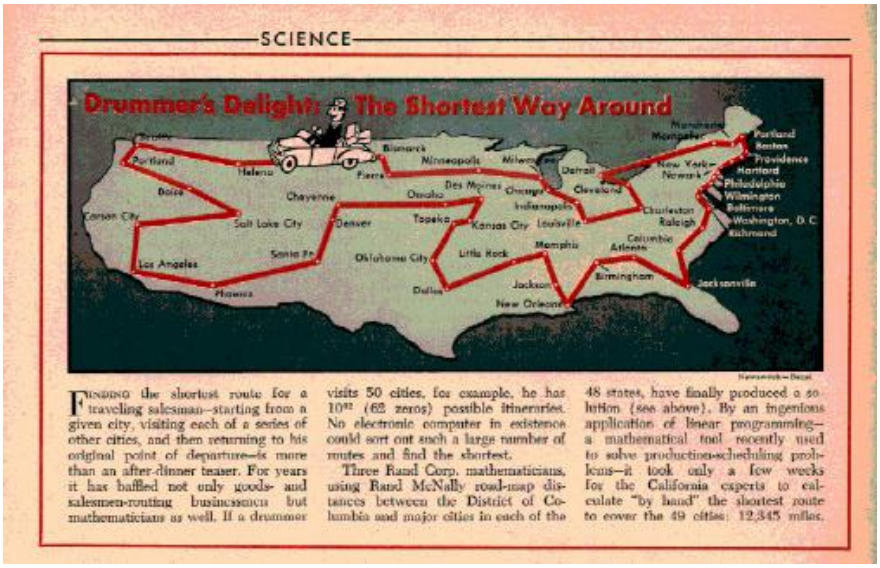


Figure 3: The 48 states problem, initially formulated by Hassler Whitney in 1954, was an earlier name for the Traveling Salesman Problem, which refers in particular to the search of an optimal tour of 48 cities, one for each USA state (Alaska and Hawaii became states only in 1959) [48].

The general TSP The *Traveling Salesman Problem* (TSP) is a combinatorial optimization problem which belongs to the class of routing and scheduling problems. It is widely studied in various areas, like theoretical computer science, operation research and computational complexity theory.

The problem has the following typical formulation: given a set of cities and a cost to travel from one city to another, we want to identify the tour that will allow a salesman to visit each city only once, starting and ending in the same city, at the minimum cost. It can be easily translated in path planning terms, where the cities become *nodes* or *vertices* in a complete *graph* with weighted *arcs*, and it's asked to seek a minimum weight Hamiltonian cycle (a cycle that touches all nodes exactly once). The only generic variable in the TSP (excluding the nodes' position) is the number of nodes, n.

Dubins' version of the TSP Being a problem of interest in various fields, in the years the TSP branched in a number of different variations, which maintain the same base assumptions while introducing more complex variables. One of those versions is the Dubins TSP (DTSP), derived from the Dubins' car originally described by L. E. Dubins [16], an important tool for demonstrating concepts in robotics and control theory including path planning subject to non-holonomic constraints, and as a way to model wheeled robots, airplanes and underwater vehicles.

In this version, in addition to the usual TSP specifications, the tour that connects every node has to follow the Dubins unicycle rules of motion, instead of being composed of simple straight lines between the vertices. This single modification leads to important differences in the final solution, as it makes all paths necessarily longer and more convoluted to account for the curves produced by a specific *direction* or *heading* kept as a node is visited, but since these changes are not necessarily uniform on every arc, consequently the best tour will often be different from the one found for the corresponding TSP.

Since many robots from various robotics application fields follow Dubins' constraints, or can generally be modeled with sufficient accuracy with a Dubins system, the DTSP is the natural next step to resolve the same routing problem while keeping track of the kinematic constraints of the system.

1.2 Applications

The TSP has many applications, including planning, logistics, and microchips manufacture; slightly modified, it appears as a sub-problem in many areas, such as DNA sequencing. As the application varies, the concept cities may represent, for example, customers, soldering points, or DNA fragments, and the concept distance between them represents traveling times or cost, or a similarity measure between DNA fragments. The TSP also appears in astronomy, as astronomers observing many sources will want to minimize the time spent moving the telescope between the sources; in such problems, the TSP can be embedded inside an optimal control problem. In general, limited resources or time windows may be added, or can be modified imposing various constraints.
For the case of the DTSP, that is the TSP modification at issue, there is a number of possible applications for robotics systems.

Figure 4: In Singapore, wheeled robots are employed as security officers or in hospitals, with tasks from cleaning to patient check-up.

1.2.1 Patrolling

The most intuitive application is robot-operated surveillance of a certain area: the DTSP is initialized by building a graph of all the possible points of interest that the robot has to pass by, and by solving the problem, a minimum cost tour is found. This allows to minimize the length of the path and in turn the fuel/battery usage, in addition to shortening the time it takes to travel the entire tour. In this category we can find land vehicles, drones or AUVs/UUVs (Autonomous/Unmanned Underwater Vehicles).

Figure 5: Drone patrol for security and surveillance of a solar farm or a wildlife park.

One example is the field of security, as seen in Singapore's police force deployed in 2018 at Suntec Singapore Convention Centre during the ASEAN summit [30]. The four-wheeled robots are programmed to move independently along a pre-defined route, and are equipped with integrated cameras that can get 360° views of its surroundings and send footage to human officers (Fig. 4.a).

Still in Singapore, at the Changi General Hospital, a large number of robots are employed, many of which are roaming wheeled ones [8]. Their routine mansions include cleaning, delivering linen or food, helping with hospital maintenance, or aiding patient rehabilitation (Fig. 4.b).

Flying robots are especially qualified for patrolling large areas like solar farms [34] or wildlife parks [40]. They can be equipped with optical and thermal infrared or radio-tracking sensors to correctly report even with adverse visibility conditions (Fig. 5).

At the 2015 Paris Air Show, car-like robots were proposed as airport security units [18]. This new equipment was designed to speed up passage through airports, to aid immigration officers and be much faster at identifying criminals through their biometric data (Fig. 6).

Figure 6: Airport security patrol in Paris.

Patrolling work is also key to improve efficiency, as seen for the robots used to inspect equipment at an electrical substation in Chuzhou, China [46]. The robots possess both an infrared thermal camera and a visual light camera, thereby giving them the ability to replace 24-hour manual inspection (Fig. 7).

Figure 7: Patrolling robot for safety evaluation at a power supply in China.

Even underwater there's need for monitoring: the underwater glider Storm Petrel is designed to patrol and report to predict the ice shelf collapse in Antarctica [38]. The information will help scientists understand the melting rate of ice on the Nansen Ice Shelf .

Other patrol robots like the Swiss ROVéo are specifically designed to work in high-risk environments, and to replace human workers in dangerous situations [7]. ROVéo can independently patrol facilities choosing its own routes and reporting instantly to security staff if an irregularity is detected; it can identify a single package missing in a warehouse that has ten thousand packages by merely passing by (Fig. 9).

Figure 8: Underwater robot monitors the Antarctic ice shelf to help predict when the next massive chunk of ice will break off.

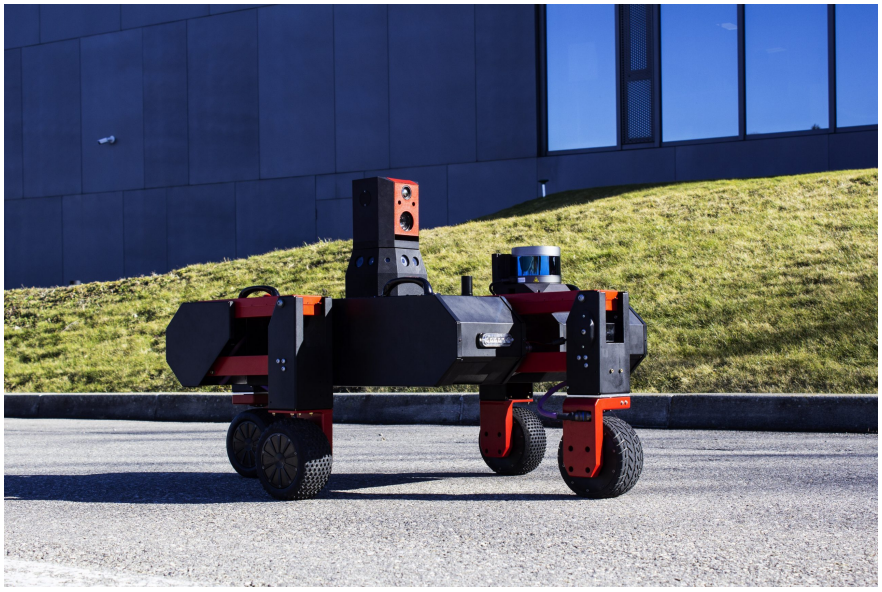


Figure 9: A robot patrol designed by a Swiss company to work in risky environments.

1.2.2 Exploration and mapping

Another scope with traits overlapping with patrolling is exploration. Consequently, it shares the same premises: once a set of hot-spots is identified, the exploring robot would be expected to return to base after reaching all of them.
Again, the robots used could be of any of the kinds already described, but in this field underwater vehicles are more common, since nowadays sea and ocean floors are probably the only real uncharted places left on Earth; moreover, they can't be observed by satellites and can't be easily reached by humans, making them the perfect candidate for robot operations.

AUVs like the SeaExplorer can gather data and transmit it periodically by satellite telemetry to a ground station [32]. In 2013 the glider completed a two-month record mission, performing several round trips between France and Corsica Island (Fig. 10.a).

The SeaBED AUV instead managed to map a previously-inaccessible area under the Antarctic sea ice, using multi-beam sonar that was compiled and converted into 3D maps [31]. This kind of underwater robots are also used to monitor the ice shelf collapse (Fig. 10.b).

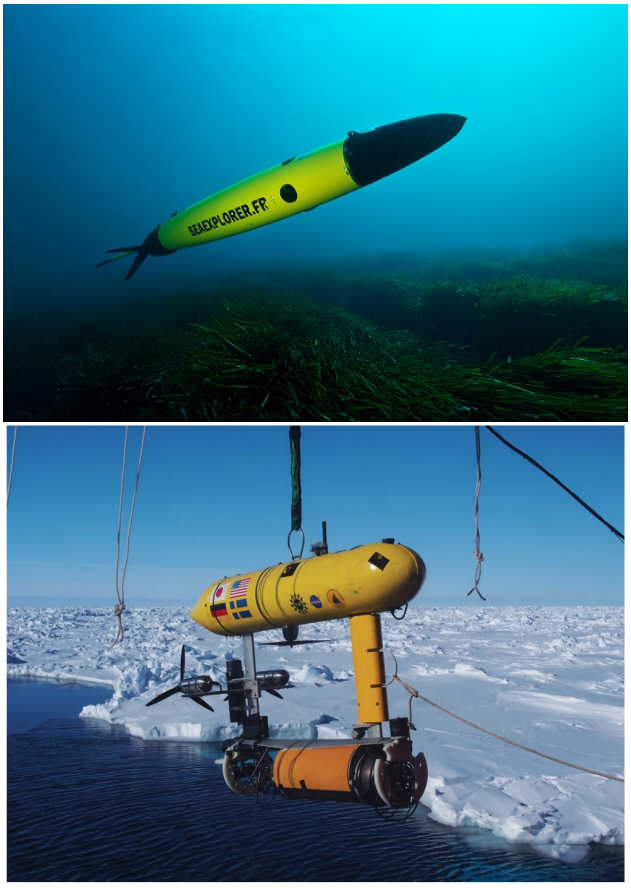


Figure 10: AUVs employed for underwater data survey or to map Antarctic sea ice.

1.2.3 Delivery services

Lastly, another typical application is delivery. Various companies teased the possible use of robots for delivery services, and some of them already started to test or adopt them.
While the online path searching program would have to keep in mind specific pathways and natural obstacles in a changing environment, an higher level tour search has to be done before departing on general neighborhoods and recipient addresses: solving the DTSP would provide the fastest "rough" path, that would later be refined using exact algorithms on pre-existing urban routes, and refined furthermore online with data from sensors on the robot itself.

Figure 11: Autonomous drones used respectively for low and high payloads delivery.

The Finnish postal service is the first company in Europe to experiment with drones to fly parcels and deliver goods [5]. Around the world, many companies are starting testing for drone and robot autonomous delivery of groceries, fast food, or other products that can be bought via online shopping (Fig. 11.a).

The Canadian heavy cargo drone Condor is surely one of the most ambitious robots designed for delivery tasks [3]. It features a payload capacity of 180 kilograms, a potential travel distance of up to 200 kilometres, and is powered by a next generation gas propulsion engine (Fig. 11.b).

Finally, delivery robots traveling on sidewalks saw a surge in demands during the COVID-19 pandemic lockdown [29]. The service was effective since it was devoid of the need for human operators, which may inadvertently contribute to the spread of the virus, and instead offered contactless delivery under mandates of social distancing (Fig. 12).

Figure 12: Six-wheeled delivery robots roaming deserted streets during the COVID-19 lockdown.

1.3 Proposed approach and objectives

Solving the TSP/DTSP The TSP is a NP-hard (*Nondeterministic Polynomial-time hard*) problem, which means that it takes polynomial running time (i.e. the time a CPU would take to run the program would be polynomial on the input size) to verify a specific solution, but it's unknown whether finding a general solution can be done in polynomial time (unless P = NP is proven [24]); as such, to date it possesses no trivial and efficient way for being solved, and it's possible that the worst-case running time for any deterministic and exact algorithm attempting to solve it would increase exponentially; and, as a matter of fact, $\Theta(2^n)$ is actually the time complexity lower bound [17].

Since it's an offshoot of the TSP, the DTSP is also NP-hard; besides, it adds a continue search space due to the headings optimization sub-problem, thus making the solution more complex and expensive, both in terms of time and resources.

Exact algorithms For instance, the brute-force recursive method used in Heap's algorithm, that merely checks every possible path, has a running time proportional with $\frac{(n-1)!}{2}$, where $(n-1)!$ is simply the number of possible permutations in a set of n with a fixed start, which is then halved because a path and its reverse are accounted as the same one; namely, time complexity of $\Theta(n!)$.
This means that by the time we're considering 20 towns or nodes ($n = 20$), there's already more than 60 quadrillion solutions (60×10^{15}).

Num of nodes $[n]$	Num of solutions $\left[\frac{(n-1)!}{2}\right]$
3	1
4	3
5	12
6	60
7	360
8	2.520
9	20.160
10	181.440
11	1.814.400
12	19.958.400
13	239.500.800
14	3.113.510.400
15	43.589.145.600
16	653.837.184.000
17	10.461.394.944.000
18	177.843.714.048.000
19	3.201.186.852.864.000
20	60.822.550.204.416.000

Even with more sophisticated algorithms, it's very expensive and often just completely unfeasible to find the optimal solution for greater n's, as it rapidly gets to a point where a standard computer would need computing times on the order of hundreds of years [25].

The Dynamic programming approach [11] is more efficient, as it yields a solution in $\Theta(n^2 2^n)$ time, which is considerably better but still too slow to use.
The Held–Karp algorithm [22] again has exponential time complexity $\Theta(n^2 2^n)$, but requires $\Theta(n 2^n)$ space to hold all computed values, while the brute force approach needs only $\Theta(n^2)$ space to store the graph itself.

Heuristic algorithms For the reasons listed above, a lot of different heuristic techniques have been developed to quickly find sub-optimal solutions for these kinds of NP-hard problems. Heuristics are methods for problem solving that employ a practical approach that is not guaranteed to be optimal, but is nevertheless sufficient for reaching an immediate, adequate approximation.

In particular, we'll focus on the heuristic approach for the TSP called Ant Colony Optimization (ACO), initially proposed by M. Dorigo [12] and later developed by Dorigo, Maniezzo, Colorni and Gambardella [10], [15], [19], [13]; when compared with classical heuristic algorithms, the ACO showed encouraging results or even dominance over the others [33]. A supplementing algorithm will also be concurrently used to support the more complex Dubins version.

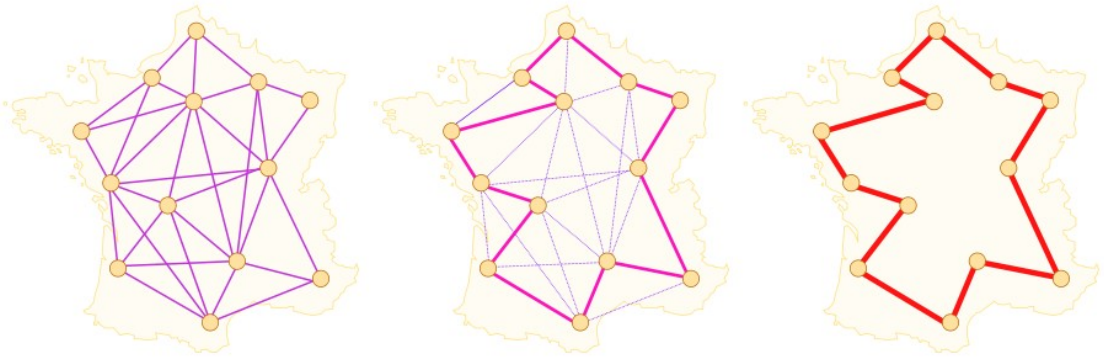


Figure 13: Example shows how the algorithm filters out paths to select one possible tour.

Objectives This work thus proposes as its goal to find a feasible solution for the DTSP through the use of the ACO algorithm, that is both fast and as close to optimal as possible. It will also address the balance between cost and efficiency due to the iterative nature of the ACO-DTSP algorithm. Then, a statistical analysis will be applied to showcase the results.

2 Description of the DTSP

2.1 The optimal Dubins path between two points

Dubins car (or *unicycle*) is a kinematic two-dimensional model of a car that can only drive forward, or turn with a bounded turning radius. It's motion rules are:

$$\begin{cases} \dot{x} &= V_0 \cos(\theta), \\ \dot{y} &= V_0 \sin(\theta), \\ \dot{\theta} &= \frac{V_0}{r} u, \quad \text{with } u \in [-1, 1] \end{cases} \tag{1}$$

where (x, y) are the position coordinates, θ is the heading, $V_0 > 0$ is the speed, $r > 0$ is the minimum turning radius and u is the control action; conventionally, $V_0 = 1$ to simplify the equations.
While u could be any number between -1 and 1, codifying every steering intensity that is less or equal to $1/r$, every time-optimal Dubins path which requires turning will only use minimum curvature steering (according to the following Theorem), therefore the control action can be reduced to $u \in \{-1, 0, 1\}$.

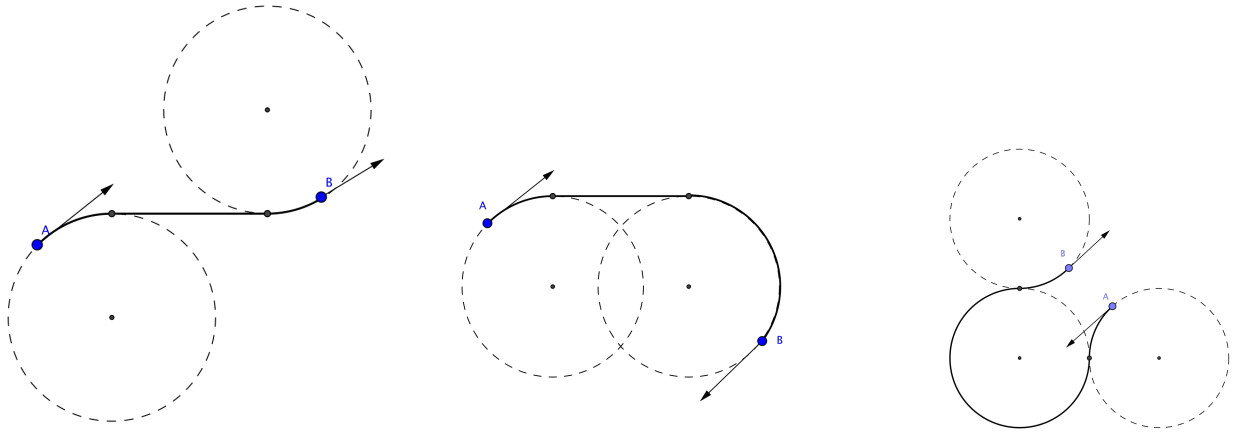


Figure 14: Examples of RSL, RSR and LRL Dubins paths.

Theorem [Dubins] Being C a general circular segment of radius r (L/R if it's a left/right turn) and S a straight segment, the Dubins Path Theorem states that the shortest path between any two configurations of a Dubins vehicle in an environment without obstacles is of type CCC or CSC, or a subpath of either of these two types [16], [27]; this reduces the number of possible paths down to six:

$$\{LSL, RSR, RSL, LSR, RLR, LRL\}$$

which constitute the sufficient family of paths [1]. In other words, the shortest path is constructed by joining circular arcs of maximum curvature and straight lines.

The method used in this work to find optimal Dubins paths is based on the algebraic solutions by Shkel *et al.* [42]. However, rather than using angular symmetries to improve performance, the simpler approach of testing all candidate solutions is used [47], [26]; this can be done efficiently by any machine, precisely because the number of eligible candidates in the sufficient family of paths is negligible.
The Dubins Path Algorithm pseudocode is showed below in Algorithm 1.

Algorithm 1 Dubins Path Algorithm

procedure
  $c_i \leftarrow (x_i, y_i, \theta_i)$
  $c_f \leftarrow (x_f, y_f, \theta_f)$
  $r \leftarrow$ turning radius
  $L_{best} \leftarrow +\infty$
  for all 6 families of canonical paths **do**
    $M_{current} \leftarrow$ mode of the canonical path
    $L_{current} \leftarrow \textsc{CheckCanonicalPath}(c_i, c_f, r, M_{current})$
    if $L_{current} < L_{best}$ **then**
      $M_{best} \leftarrow M_{current}$
      $L_{best} \leftarrow L_{current}$
    end if
  end for
  $path \leftarrow \textsc{GenerateBestDubinsPath}(c_i, c_f, r, M_{best})$
return $(path, L_{best})$
end procedure

2.2 The TSP as a series of Dubins paths - the DTSP

The *Dubins Traveling Salesman Problem* (DTSP), as previously discussed, encapsulates the motion rules of the Dubins unicycle with the path-finding rules of the TSP.
Most notably, we can envision applying DTSP algorithms to the setting of a UAV (Unmanned Aerial Vehicle) monitoring a collection of spatially distributed points of interest, whose location is known and static; the Dubins vehicle is indeed commonly accepted as a reasonably accurate kinematic model for aircraft motion planning problems.
Solving the DTSP reduces then to choosing a permutation of the points specifying in which order to visit them, as well as choosing a *heading* for the vehicle at each of these points.

The fact that the Dubins path between two nodes is strongly dependent on the two headings of the vehicle when it starts and then when it and arrives at the target, makes it so that, for the whole n-tour, the choice of those headings becomes as important a variable as the number of nodes n. We can for instance imagine that given a pre-determined set of n headings that optimizes a DTSP tour, adding another node might, with very high probability, make the $(n+1)$-tour non-optimal for any heading of the last added node.

An example of an algorithm that produces a naive, complete solution for the DTSP, including a waypoint ordering and a heading for each point, is the *Nearest Neighbor Algorithm* [27]: the heuristic starts by arbitrarily choosing the first node and its heading, fixing an initial configuration, and then computes the best heading for every subsequent node that minimizes each inter-node path; in other words, it chooses as its next configuration one that is closest to the last added one according to the Dubins metric. Then, when all nodes have been added to the path, it adds a Dubins path connecting the last and the initial configurations.

One of the first algorithms proposed that distances itself from the "greedy" approach is the *Alternating Algoritm* [41]: it essentially consists in replacing the even-numbered edges of a simple TSP tour with Dubins paths. This simple method became the standard benchmark or reference for many new following DTSP algorithms.

Together with the Alternating, the *Recursive Bead-Tiling Algorithm* [41] was proposed; a more complex approach that instead tessellates the space in a specific way and then follows a set of rules to visit nodes inside those tiles.

The *Randomized Headings Algorithm* [27] instead just randomizes the headings at each waypoint for the Dubins Path. As bleak of a method as it may seem, it's actually shown to perform better than the Alternating as the number of nodes increases.

The *Look-Ahead Algorithm* [28] (also called Three Point Algorithm) picks a random heading for the first node, then considers the first 3 nodes and gets the optimal heading for the middle one, to then repeat, shifting one node over, until the tour is closed. It's a direct improvement to the greedy solution or Two Point Algorithm.

The *Heading Discretization Algorithm* [27] considers k possible headings for each node, and then solves the ATSP (*Asymmetric Traveling Salesman Problem*) on $k \cdot n$ nodes with direct algorithms like the LKH (Lin-Kernighan Heuristic) [23]. In a problem where the waypoint order is given, it's possible to consider only the headings as variables.
However, it was shown that not only the average computation time increases exponentially with the discretization level, but even small changes of the discretization may completely change the course of the tour and significantly increase or decrease the tour length, and consequentially it's advised to choose as fine a discretization as possible [2]. For this reason, the algorithm can get to high levels of optimization, but requires a lot of time to complete.

A variation of the Look-Ahead, the *Discretized Look-Ahead Algorithm* [9] uses the same principles but adds heading discretization. The discretization is proposed to strike a balance between the execution time and the length of the resulting admissible tour.

The *Genetic Algorithm* [49] exploits a well known approach in the artificial intelligence field: the variables that need to be optimized (usually called *genes*) are selected, crossed-over and mutated in a population of individuals; the process is iterative, and as it goes on the individuals with genes that satisfy the fitness function the most "survive", while the others are eliminated, in a cycle that resembles natural selection and evolution.
In the context of the DTSP, this algorithm has the ability to search in continuous variable domain, discrete variable domain or mix type variable domain, so it can be set to search for the best DTSP tour from scratch or just to pick the nodes' headings as genes.

Lastly, the *Pulley Algorithm* [20] is a geometric method based on the behavior of the physical pulley mechanism, initially proposed as an improvement to the Look-Ahead, and proved to outperform many classical heuristics.

3 Solving the DTSP using the ACO algorithm

3.1 Description of the ACO and solving the TSP

The *Ant Colony Optimization* (ACO) algorithm, as its name suggests, draws inspiration from the emergent behavior of ants, more specifically their path seeking abilities as they move between the colony and a source of food.
Like other systems studied in swarm robotics, even if a single ant has only simple capabilities, interesting aspects arise from the observation of a whole ant colony, which, as a result of coordinated interactions, is instead highly structured.
This distributed problem solving environment is thus proposed, and it's used to search for a solution to the TSP.

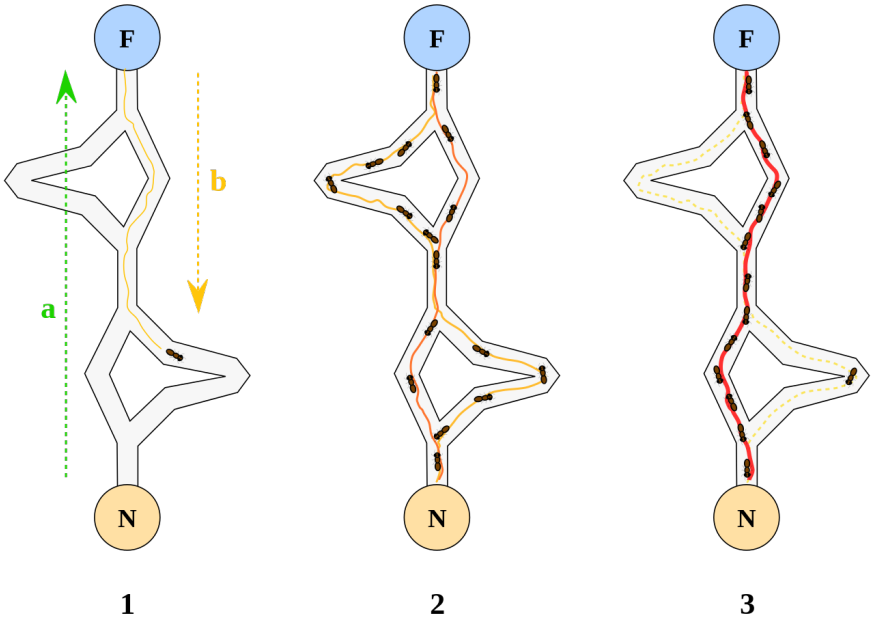


Figure 15: Ant colony emergent behavior.

In Fig. 15 we can see the natural ant colony behavior in action:

1. the first ant finds food (F) using a random path (a), then returns to its nest (N) while leaving a pheromone trail behind (b);

2. other ants also randomly choose every possible path to reach (F) from (N), but *reinforcement* of the pheromone trail makes the shortest path more desirable (this is obviously not a choice made by the ants, but an emergent behavior, since shortest paths are faster to cross and therefore the pheromone will accumulate at a higher rate);

3. after a while, all the ants will use the shortest path, as other paths lose their pheromone due to evaporation.

For the formal description of the ACO, we define:

$\tau \rightarrow$ the "pheromone trail level", that simulates the stimuli ants experience with their peculiar communication method;
$\eta \rightarrow$ the "attractiveness" of a possible path (or equally, of the next possible edge to anoher node), usually set as the inverse of the distance among the nodes;
$M_k \rightarrow$ the "working memory" of each ant, a table of every node that was already visited by that particular ant, used to avoid the creation of non-hamiltonian tours.
Finally, the "desirability" is the trail level multiplied by the attractiveness (that is, $\tau \cdot \eta$) [13], and is the core of the algorithm, as those two are the main metrics which the optimization is based on [14].

The ACO algorithm for the TSP is composed by three fundamental rules.

State Transition Rule (Pseudo-Random Proportional Rule)

$$s = \begin{cases} \arg\max\limits_{u \notin M_k} \{[\tau(r,u)]^\alpha \, [\eta(r,u)]^\beta\}, & \text{if } q \leq q_0 \quad \text{(exploitation)} \\ S, & \text{otherwise} \quad \text{(biased exploration)} \end{cases} \tag{2}$$

Here, α is the pheromone power parameter, and controls how likely each ant will be to follow the same paths as the ants before it (too high and the algorithm will keep searching the same path, too low and it will search too many paths, virtually ignoring the pheromone on them); β is the distance power parameter, and controls the extent to which ants will prefer nearby points (too high and algorithm will essentially be a greedy search, too low and the search will likely stagnate); q is a random number uniformly distributed in [0,1]; q_0 is the exploitation threshold parameter ($0 \leq q_0 \leq 1$); and S is a random node selected according to the following probability distribution:

$$p_k(r,s) = \begin{cases} \frac{[\tau(r,s)]^\alpha \, [\eta(r,s)]^\beta}{\sum_{u \notin M_k} [\tau(r,u)]^\alpha \, [\eta(r,u)]^\beta}, & \text{if } s \notin M_k \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

We can see that $p_k(r,s)$ is the probability distribution function that shows the probability for which the (r,s) edge would be chosen, normalizing the desirability of each remaining candidate and assigning a probabilistic weight proportional to it.
This is done as a parallel to genetic algorithms [45], where the *mutation* term makes the solution research more flexible and less subject to stagnation in a local minimum.

Local Updating Rule

$$\tau(r_k, s_k) \leftarrow (1-\rho)\,\tau(r_k, s_k) + \rho\,\Delta\tau(r,s) \tag{4}$$

Here, ρ is the evaporation rate (or pheromone decay parameter), and is the proportion of pheromone that evaporates on each step; $\Delta\tau(r,s)$ can be set to τ_0, the initial pheromone level, because it's simple to implement and produces better or similar performance with

respect to other tested options (Q-learning formula or 0) [14]; another given option is to use $\frac{1}{n\,L^{-}_{gb}}$, where L^{-}_{gb} is the best tour from the last iteration, to implement the correction rule for the Reinforcing Ant Colony System (RACS) [36] [35].
The variable τ_0 can in turn be set as an editable variable (`initialPheromoneIntensity`, explained in Chapter 4.1) or $(n\,L_{nn})^{-1}$ [39], where n is the number of nodes and L_{nn} is the length of the naive tour found with the Nearest Neighbor algorithm (referenced in Section 2.2); this last option is used as a very rough approximation of the optimal tour length.
This update is made every *step*, which means each time a new node is chosen for all m ants whose tours are still in-progress; this is done to simulate the pheromone deposited by an ant in its wake as it moves.

Global Updating Rule

$$\tau(r_k, s_k) \leftarrow (1-\rho)\,\tau(r_k, s_k) + Q\,\Delta\tau(r, s) \tag{5}$$

$$\Delta\tau(r,s) = \begin{cases} (L_{gb})^{-1}, & \text{if } (r,s) \in \text{global-best-tour} \\ 0, & \text{otherwise} \end{cases}$$

In this last control rule, Q can be chosen between an editable variable (`pheromoneIntensity`) and the pheromone evaporation rate ρ. There is again the option to use the RACS, and in that case the Global Updating Rule also re-initializes trails that go over the upper bound $\tau_{max} = \frac{1}{(1-\rho)\,L_{gb}}$ [36] [43].
This update is made every *iteration*, which means each time a group of m ants has finished their search and a global best tour is being found; this is done to simulate a predilection for the shortest path by the colony, as with time more ants would walk on it and make the pheromone trail more intense and less prone to vanishing through evaporation.

3.2 From TSP to DTSP - the DSPP

As seen in Chapter 2.2, the state of the art for DTSP solvers offers various competing algorithms, that can be arranged in two different categories:

- ones that solve the TSP and modify the already best Hamiltonian tour;
- ones that directly solve the DTSP without solving the TSP.

The ACO algorithm already creates ordered sequences of waypoints for the TSP; therefore, to solve the DTSP, it needs an underlying function to get sensible node headings. Those headings then, together with the known node positions, would be converted in Dubins paths to get the tour length, and enabling the algorithm to evaluate the best Dubins tour for every iteration.
Finding a Dubins path through a pre-existing ordered set of targets is a simplified version of the DTSP, called the *Dubins Shortest Path Problem* (DSPP). For this reason, only the second category of DTSP solvers was considered as possible candidates.

What is needed is a fast algorithm, which can be used in conjunction with an iterative algorithm as is the ACO. Many possibilities were examined, and the available algorithms were skimmed to discard those that weren't relevant to the case in exam: any algorithm that inherently include choosing the order of nodes, which can't be separated from the choice of the headings (such as the Bead-Tiling); any algorithm that is iterative itself (such as the Genetic); or any algorithm that needs long computation times to perform (such as the Heading Discretization).

In the end, the Alternating Algorithm was implemented in the initial version, that later was improved with the Pulley Algorithm as a new, fast, performing DSPP algorithm.

3.2.1 Alternating Algorithm

The *Alternating Algorithm* (AA) was initially chosen as a way to easily transform a TSP tour in a DTSP tour [41], in order to redirect the objectives of the ACO algorithm to the optimization of a different version of the TSP, without modifying its core dynamics.
The fundamental idea of the AA is to alternate straight line edges, as you'd have for classic TSP tours, with minimum-length Dubins paths, preserving the point ordering.
The AA pseudocode is showed below in Algorithm 2.

However, this approach has a downside: if the curvature radius of the Dubins system is comparable or even bigger than the minimum distance between any two nodes, the resulting tour is bound to have "loops" that make the best tour obtainable less optimal.

In Fig. 16 is shown that if a tour which has been optimized for the TSP is then converted into a Dubins path, it will almost inevitably become non-optimal, since the TSP as it's elaborated doesn't have headings taken into account. On the other hand, Fig. 17 displays the optimizing possibilities of running the AA online.

It can be proved that the AA performs well when the points to be visited by the tour are chosen in an adversarial manner [41]. However, it is not a constant-factor approximation algorithm in the general case. Moreover, the AA might not perform very well when dealing with a random distribution of the target points.

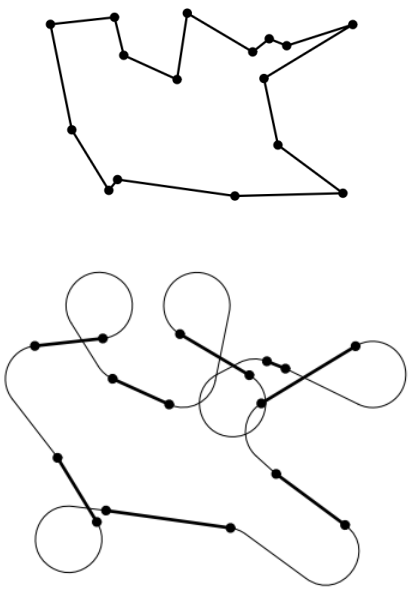


Figure 16: Example of application of the AA after a TSP optimization.

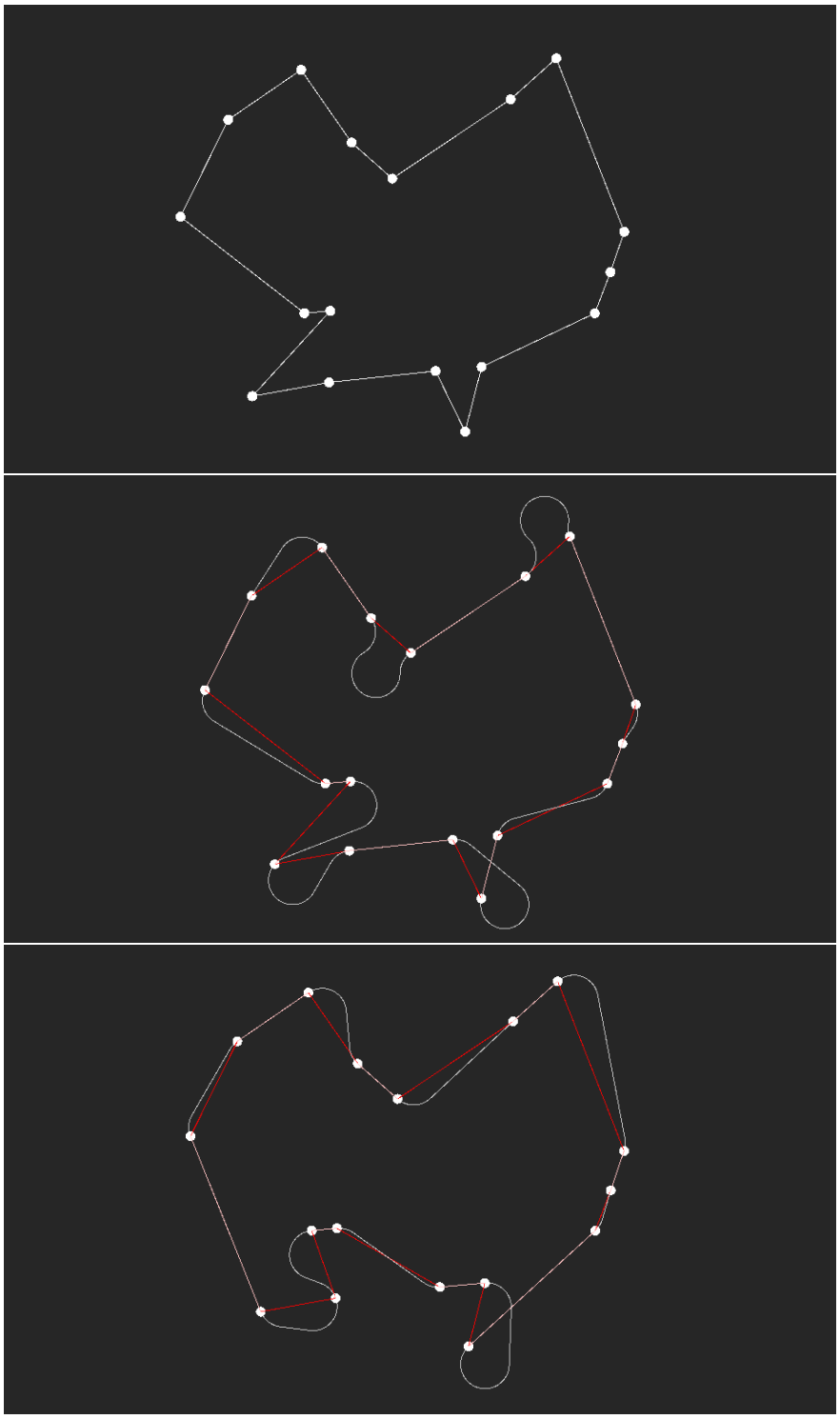


Figure 17: Example taken from the newly-proposed ACO-DTSP program using the AA. TSP tour length = 313.555; AA applied on the TSP tour length = 413.123 (same waypoint order as TSP tour visible in red); complete DTSP with AA tour length = 373.102 (different waypoint order, optimized for the DTSP).

3.2.2 Pulley Algorithm

Successively, it was chosen another more complex algorithm to convert the TSP optimal solution to a kinematically feasible optimal Dubins path, called the *Pulley Algorithm* (PA) [20].
This new method returns angles based on a natural physical system, namely the pulley, to get the tightest path given an ordered list of waypoints.
The algorithm can be described by the following two steps:

- place a circle of radius r_{min} at each waypoint, such that it intersects the arriving and departing straight paths in two equal chords, with the waypoint in the middle of the arc that starts with the first intersection and ends with the second one;
- connect every pair of consecutive circles with a line tangent to both.

The PA pseudocode is showed below in Algorithm 3.

The PA not only finds Dubins-constrained paths that better fit any arbitrarily ordered set of nodes, but unlocks faster search for better tours if used in addition to an optimization algorithm like the ACO-DTSP, since it makes tours that otherwise would be discarded feasible. Some examples of this behavior are shown in Fig. 20, 21, 22.

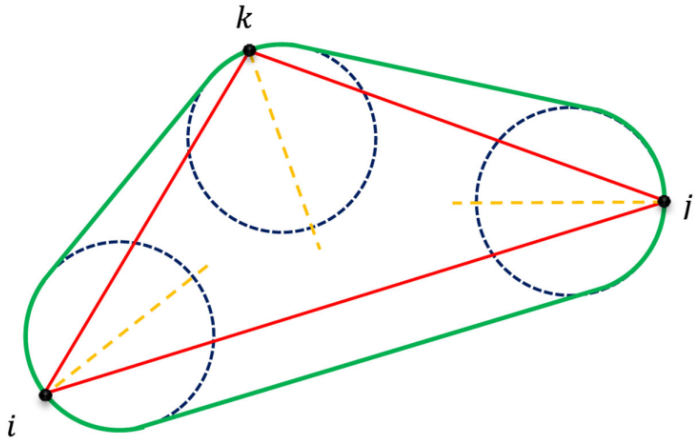


Figure 18: Example of application of the PA on three nodes.

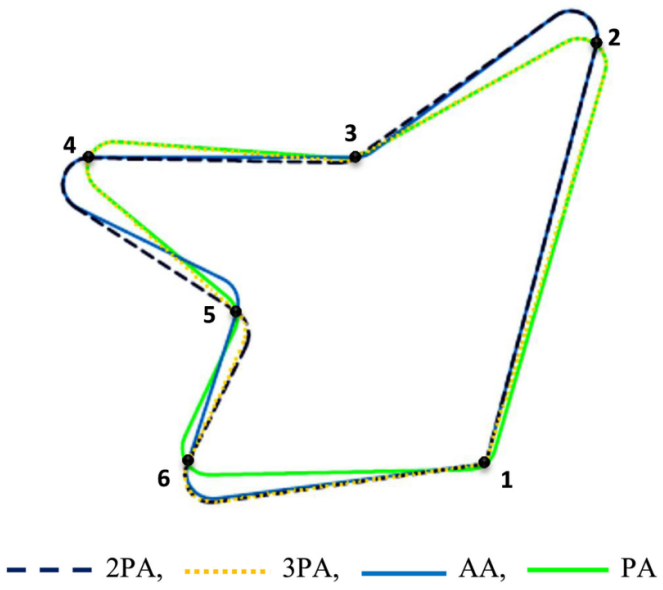


Figure 19: Graphical comparison of various headings-optimization algorithms, including the PA, for the same set of points.

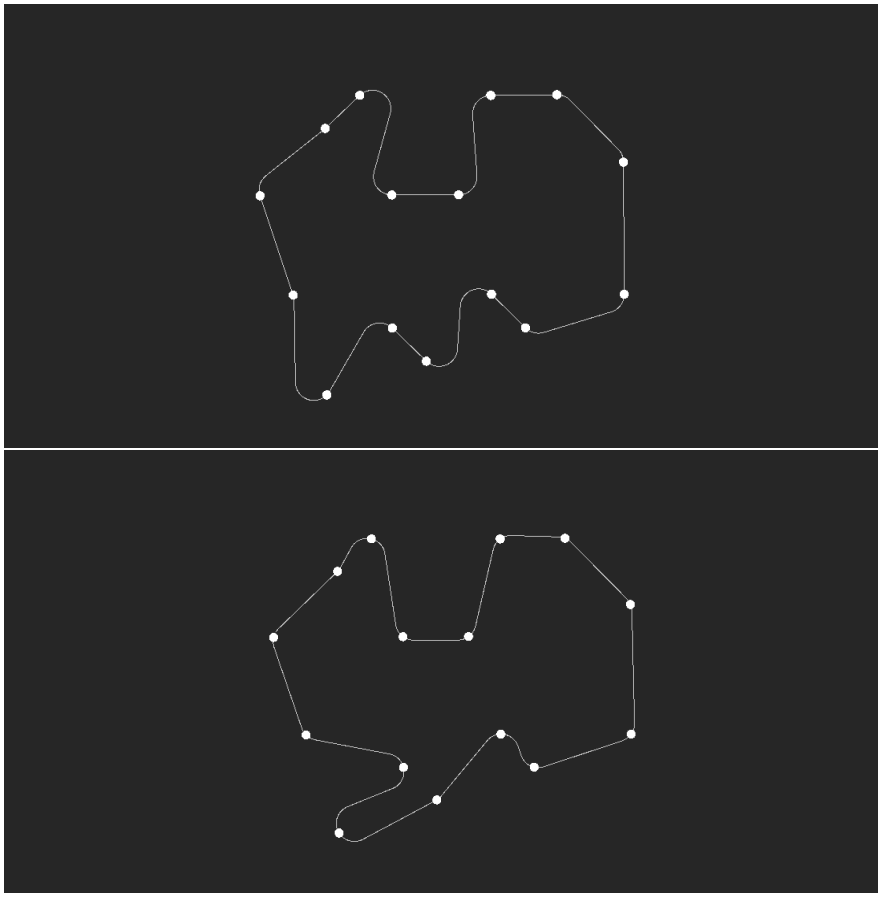


Figure 20: P01 set: AA = 315.152; PA = 307.358.

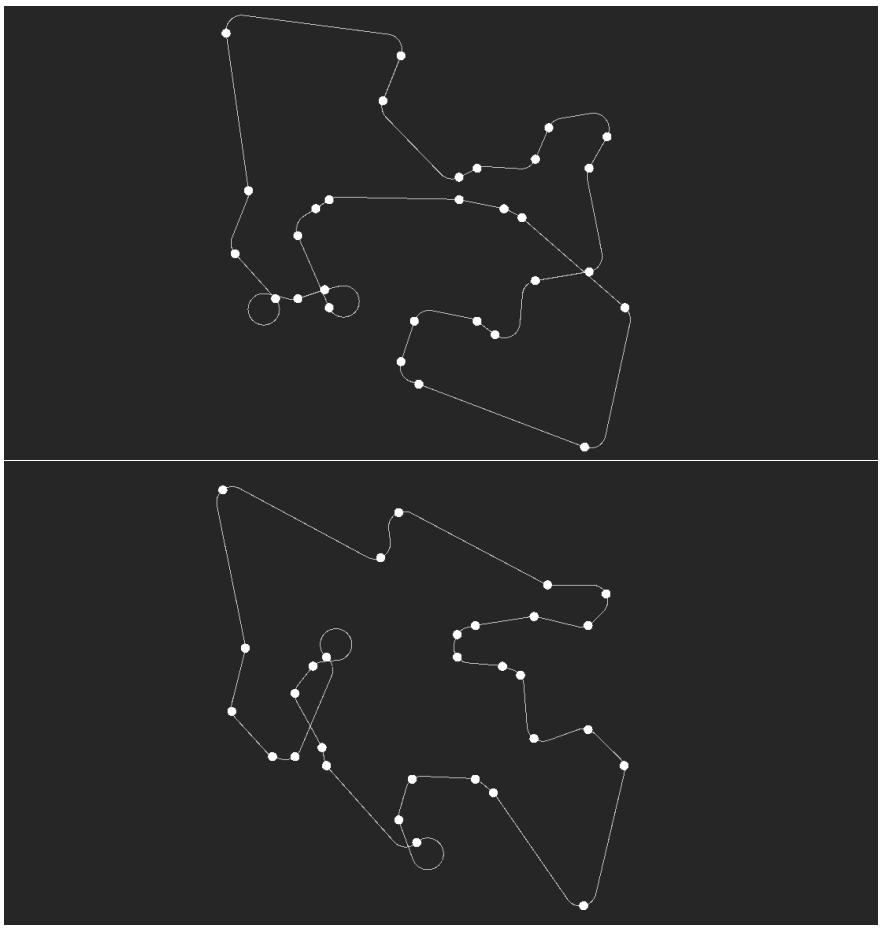


Figure 21: Oliver30 set: AA = 521.857; PA = 493.648.

Algorithm 2 Alternating Algorithm

procedure
choose an ordering of nodes $(a_1, ..., a_n)$
$\psi_1 \leftarrow$ orientation of segment from a_1 to a_2
for $i := 2 \rightarrow n - 1$ **do**
if i is even **then**
$\psi_i \leftarrow \psi_{i-1}$
else
$\psi_i \leftarrow$ orientation of segment from a_i to a_{i+1}
end if
end for
if n is even **then**
$\psi_n \leftarrow \psi_{n-1}$
else
$\psi_n \leftarrow$ orientation of segment from a_n to a_1
end if
return configurations sequence $\{(a_i, \psi_i)\}_{i \in \{1,...,n\}}$
end procedure

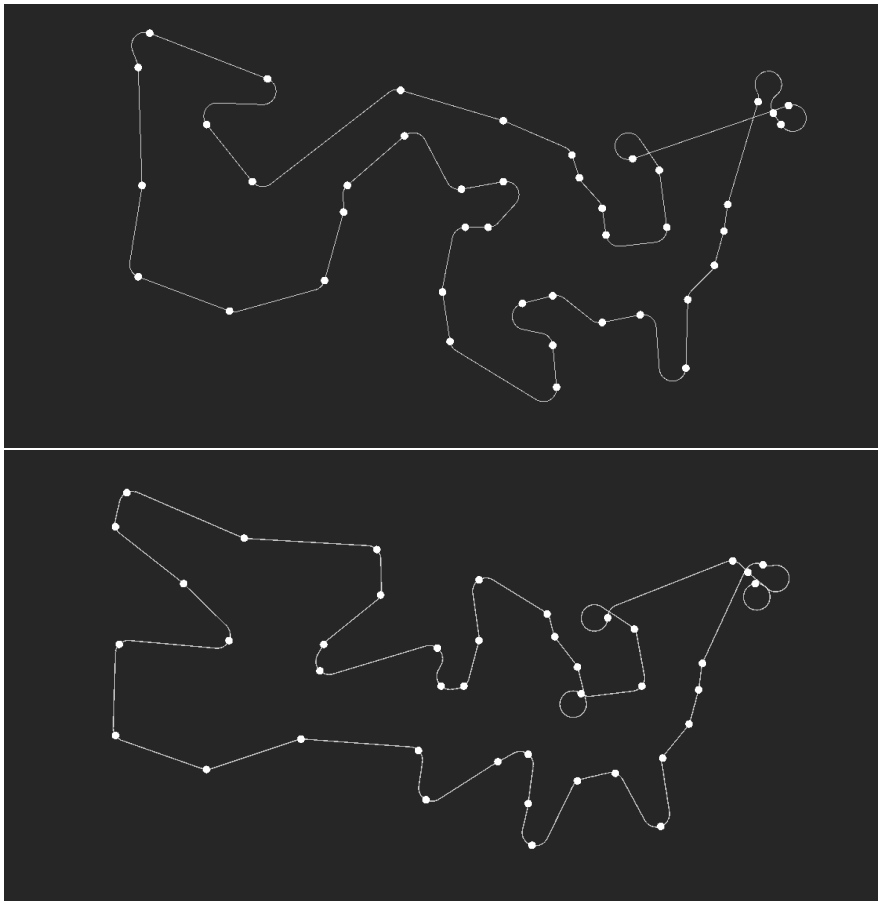


Figure 22: Dantzig42 set: AA = 809.949; PA = 797.652.

Algorithm 3 Pulley Algorithm

procedure
choose an ordering of nodes $(a_1, ..., a_n)$
$\psi_1 \leftarrow$ orientation tangent to circle given a_n, a_1 and a_2
for $i := 2 \rightarrow n - 1$ **do**
$\psi_i \leftarrow$ orientation tangent to circle given a_{i-1}, a_i and a_{i+1}
end for
$\psi_n \leftarrow$ orientation tangent to circle given a_{n-1}, a_n and a_1
return configurations sequence $\{(a_i, \psi_i)\}_{i \in \{1,...,n\}}$
end procedure

3.3 The ACO-DTSP algorithm

The last step for the implementation of the ACO for the DTSP consists in the integration of the chosen DSPP method with the standard ACO algorithm, in a way that shifts the focus from the TSP, the main problem which the ACO was elaborated for, and directly addresses the DTSP.

In the original formulation of the ACO algorithm, the length of any TSP tour found is used to compare them to each other and search for the shortest one, enabling for the optimization to take place. This behavior was slightly modified for the ACO-DTSP.

Each time an ant creates a new tour, through the repeated application of the State Transition Rule, the ordered waypoints are saved, and on every *iteration* they're converted into Dubins tours using the Pulley Algorithm; the resulting path length is then used to check which one is the shortest, so the best Dubins tour can be computed and visualized online as the algorithm goes on. In this way the loops issue indicated in Section 3.2 is more easily avoided, since the loops would make the paths longer and therefore are filtered out during the optimization process; however, they persist if the nodes that have a small distance between them with respect to the curvature radius are in a configuration that doesn't allow a smooth motion.
Because of this, it's always advisable to choose a curvature radius that's smaller than the minimum distance between any two nodes of the system. The complete ACO-DTSP algorithm is reported as pseudocode in Agorithm 4.

To roughly check the performance of the ACO-DTSP algorithm, given the lack of available datasets for the Dubins TSP, a simple test was carried out using geometry axioms.
It was programmed for the nodes to be positioned in a certain way, in order for the resulting set to assume the shape of a circumference. Since for such a set the shortest tour will always be the one that draws the implied circumference, we easily have not only the desired waypoint order, but the approximation of the desired path length too: in fact, knowing the radius r and using the circumference perimeter formula we have the path length when the number of nodes approaches infinity:

$$\lim_{n \to \infty} L = 2\,\pi\,r$$

Since the number of nodes is finite, we can expect the length L to be even smaller. With this knowledge, we have an upper bound for the best tour length found, that becomes more precise as the number of nodes n increases. Additionally, since a GUI (Graphical User Interface) is present, for a single execution it's trivial to visually check the optimization results. The program successfully recognises the circular pattern as the best tour very quickly, usually merely within 5 iterations or less.

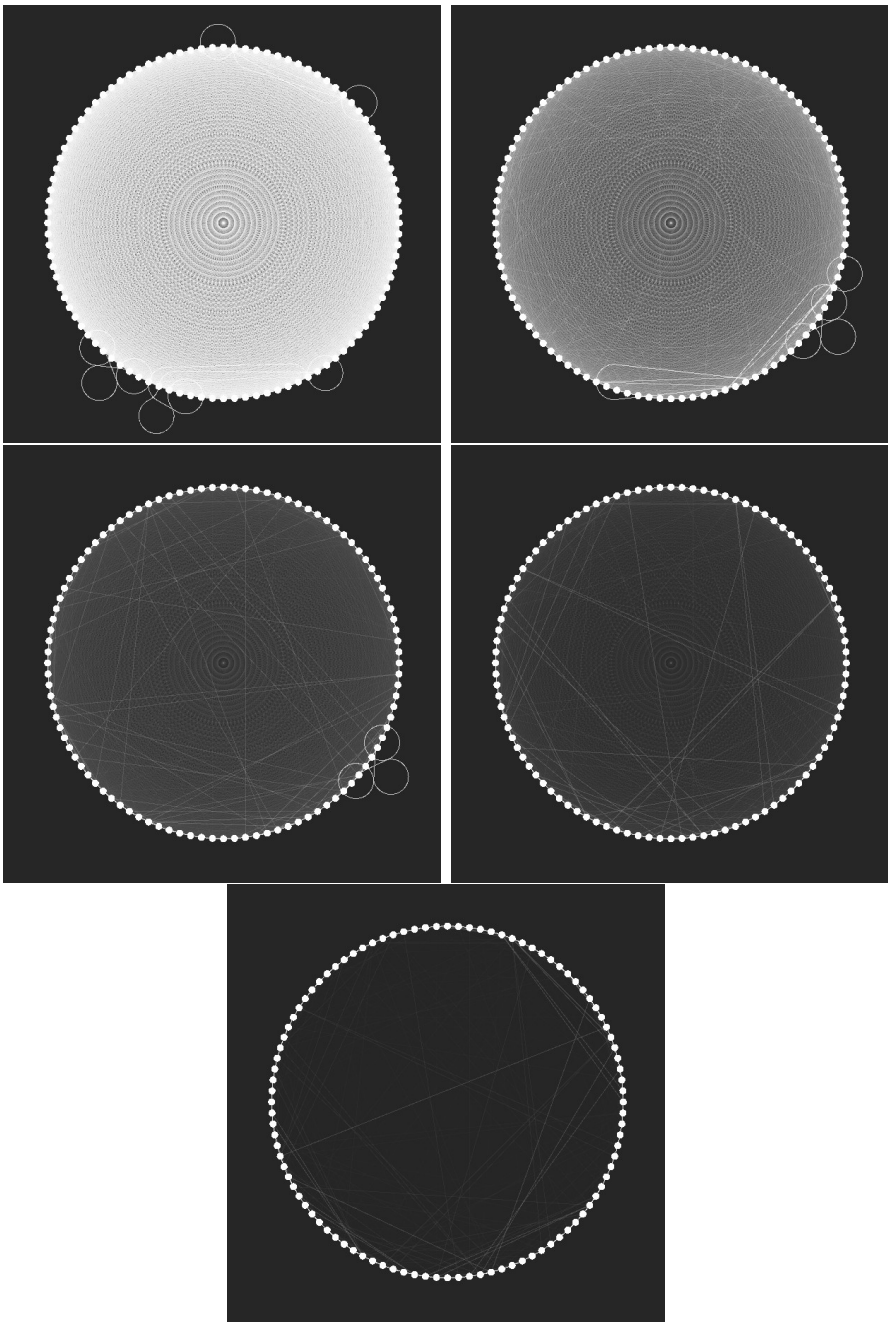


Figure 23: Circumference test on 100 nodes. The circumference pattern is correctly found in 4 iterations. A visual representation of pheromone trails can be seen as white graph lines, getting paler on every new iteration as the pheromone "evaporates" from bad paths and converges to the optimum.

Algorithm 4 ACO-DTSP Algorithm

procedure
Initialization:
  for all (r, s) **do**
    $\tau(r, s) \leftarrow \tau_0$
  end for
Main Loop:
  while stop_condition != false **do**
    Iteration:
    for $k := 1 \rightarrow m$ **do**
      r_{k_1} starting node
      $r_k \leftarrow r_{k_1}$
      reset M_k
    end for
    for $i := 1 \rightarrow n$ **do**
      Step:
      for $k := 1 \rightarrow m$ **do**
        $M_k \leftarrow M_k \cup r_k$
        if $i < n$ **then**
          Pseudo-Random Proportional Rule:
          $s_k \leftarrow \text{ChooseNextNode}(r_k, M_k)$
        else
          $s_k \leftarrow r_{k_1}$
        end if
        $Tour_k(i) \leftarrow r_k$
      end for
      Local Updating Rule:
      for $k := 1 \rightarrow m$ **do**
        $\tau(r_k, s_k) \leftarrow (1 - \rho)\,\tau(r_k, s_k) + \rho\,\Delta\tau$
        $r_k \leftarrow s_k$
      end for
    end for
    for $k := 1 \rightarrow m$ **do**
      $(L, Tour) \leftarrow \text{TSPtoDTSPalgorithm}(Tour_k)$
      search for $(L_{gb}, Tour_{gb})$
    end for
    Global Updating Rule:
    for all (r, s) **do**
      $\tau(r, s) \leftarrow (1 - \rho)\,\tau(r, s)$
      if (r, s) in $Tour_{gb}$ **then**
        $\tau(r, s) \leftarrow \tau(r, s) + Q/L_{gb}$
      end if
    end for
  end while
end procedure

4 Software overview

For this work, two programs where used: "Ant Colony Optimization DTSP" and "ACO-DTSP Statistical Analysis". Both programs were written in C# programming language, and run on the Unity engine.

4.1 Main program: Ant Colony Optimization DTSP

Unity is a real-time development platform that provides a graphical-driven workspace. The simulation takes place in the Scene window, where the nodes and edges of the dynamic graph are drawn and updated in real-time once the algorithm has started. In the Game window, another camera shows a text object with useful data on the current simulation:

- number of nodes for the current execution;
- duration of the current algorithm session, in seconds;
- best TSP tour length found;
- best DTSP tour length found;
- turning curvature ("None" if the search is set on TSP).

Moreover, in the same window the interactive Start and Stop button are located.
Finally, the Inspector tab contains all the parameters for the execution.

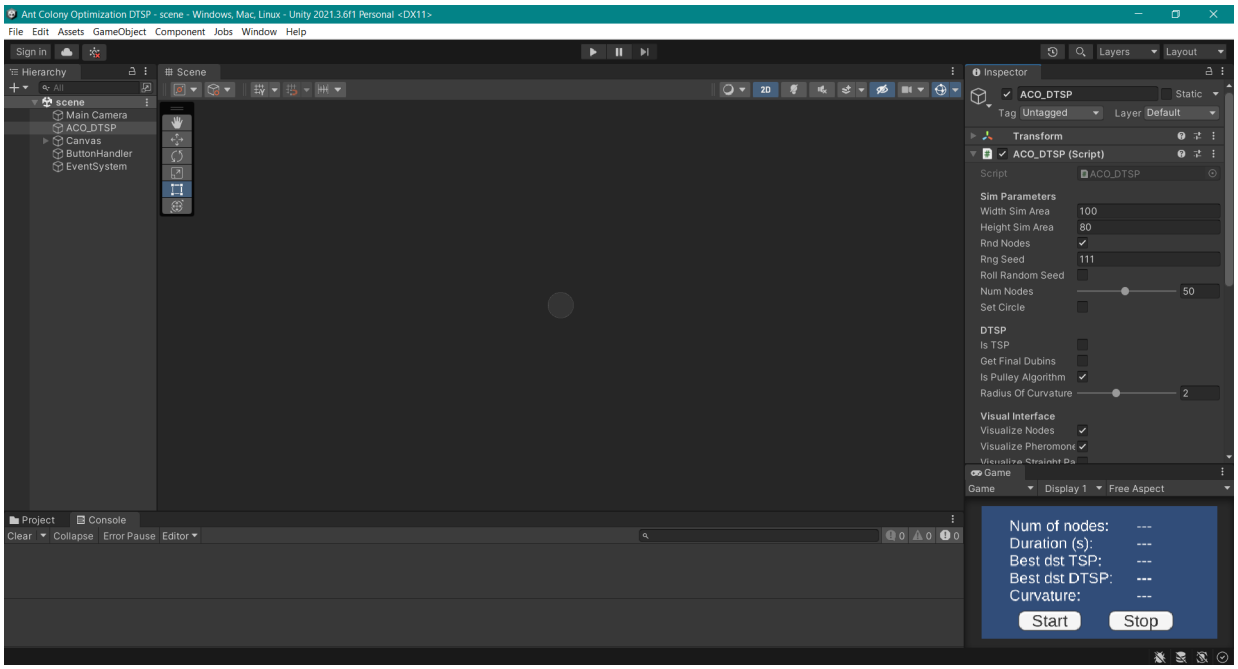


Figure 24: Unity GUI for the "Ant Colony Optimization DTSP" project.

The program gives the option to manually choose the position of each node by clicking with the cursor on different parts of the Scene window (multiple nodes on the same location are not permitted), or to get random positions for a set number of nodes. If the former option is chosen, the presence of a Start button is especially useful since it allows for a time window between the start of the program and the start of the algorithm to give the required input.

The whole project is composed of five scripts.

- ACO-DTSP: the main algorithm, includes the method that draws the simulation.
- Dubins: contains all the methods to get the best Dubins path given starting/ending position and orientation.
- TSPLib: contains some more general purpose methods and the positions for the chosen TSPLIB sets (see below).
- ButtonHandler: monitors the buttons and sends a event-driven signal when they are pressed.
- TextManager: manages the text in the Game window as the variables are updated real-time.

Below, a rundown of the customizable parameters available in the Inspector, and how they change the program execution.

Simulation parameters $\downarrow$

General attributes for the simulation.

`widthSimArea` and `heightSimArea` specify the boundaries of the simulation when random nodes are chosen (i.e. the range for the random choice of positions).

`rndNodes` is a boolean to get random nodes. If checked, it will bypass any previously set nodes; on the other hand, even if unchecked the program will automatically choose random nodes if the number of set nodes is < 3.

`rngSeed` sets the seed for the random number generator. If it's kept unchanged, it'll make the program return the same set of random nodes; useful when dealing with statistics on specific sets on multiple sessions.

`rollRandomSeed` instead bypasses `rngSeed` to always get a different set on every execution. It uses the time since program startup and a hash code to always return different random seeds.

`numNodes` ("n") is the number of nodes. It can be set *a priori* and then get randomly generated node positions; alternatively, the user can choose to set the nodes by hand by clicking on the Scene window before starting the simulation, and in this case it will be equal to the number of click inputs.

`setCircle` is a boolean that draws a circular perimeter using the amount of nodes set in `numNodes`. As the number increases, the approximation gets more accurate.

DTSP ↓

To set the specific type of Traveling Salesman Problem needed for the current execution, and the relative parameters for the Dubins TSP.

`isTSP` is a boolean that switches between TSP and DTSP solving ACO algorithm.

`getFinalDubins` is a boolean that, when the program is stopped, takes whatever final order of nodes the algorithm has produced and outputs a Dubins path. It can be useful while `isTSP` is set to true to see the length difference, and how directly solving for the DTSP or solving for the TSP and then switching to a Dubins tour in the end leads to considerably different results.

`isPulleyAlgorithm` is a boolean that switches between Pulley Algorithm and Alternating Algorithm, useful for potential comparisons of the implemented DSPP algorithms.

`radiusOfCurvature` ("r") lets you decide the minimum turning radius for the Dubins unicycle. It can also be changed while the algorithm is running, but doing so may degrade the performances of the search, especially for complex tours; this is because, if the algorithm runs on predetermined conditions for a long time, it will converge towards a result based on them, and if those conditions are later changed the obtained results could be far from optimal.

Visual Interface ↓

Parameters used to get different level of visual information from the simulation.

`visualizeNodes` draws white discs over the nodes' position to better locate them on the GUI.

`visualizePheromone` draws straight white lines on edges, whose brightness is directly proportional to the pheromone level of each edge.

`visualizeStraightPath` draws straight red lines to highlight the waypoint order in case of a DTSP search.

TSP library $\downarrow$

The program also offers the possibility to set the nodes to well-known configurations, present in the data catalogue TSPLIB [21] [37]. The implemented configurations are `p01` (15 nodes), `oliver30` (30 nodes), and `dantzig42` (42 nodes). For each of them it was checked empirically that the tour length found by the ACO algorithm is indeed consistent with the given optimal solution. Unfortunately the library doesn't account for DTSP tours, but it's still useful to check the expected convergence for the general TSP.

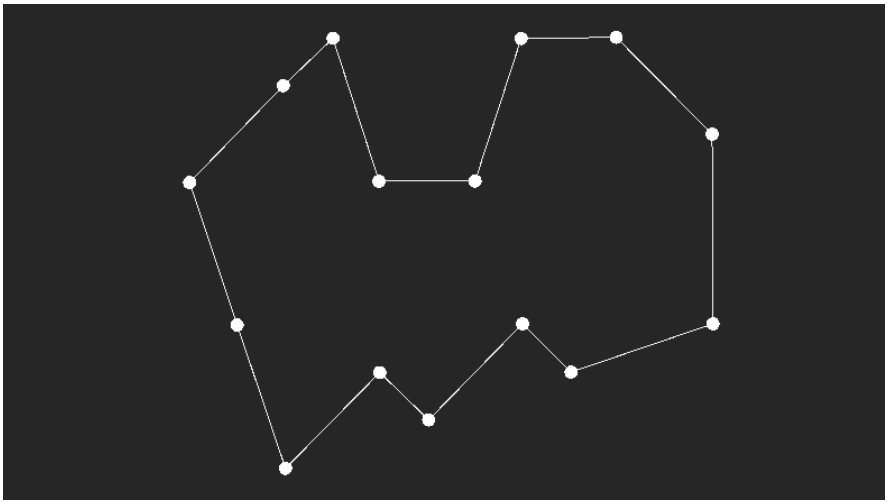


Figure 25: Set "p01". Best tour length found = 284.381. From TSPLIB: shortest cycle length (if each distance is rounded to the nearest integer) = 291.

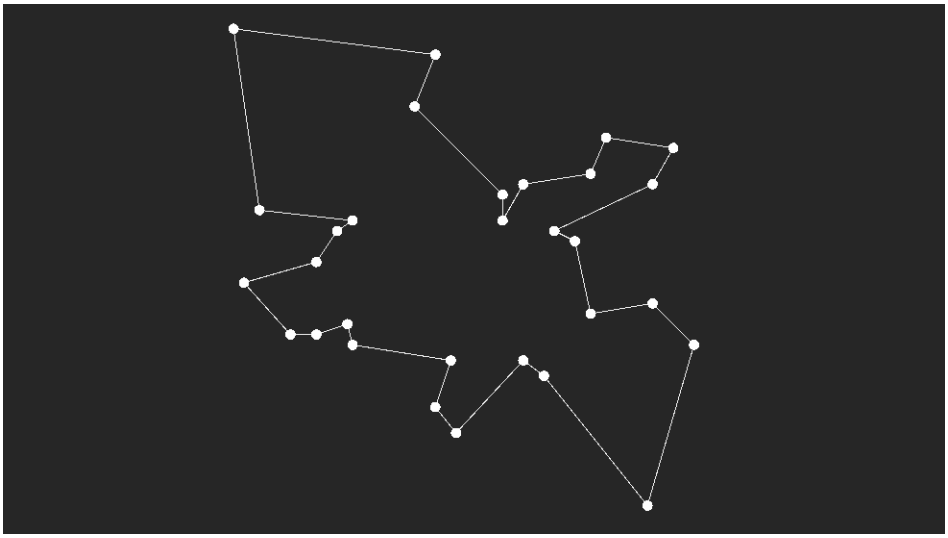


Figure 26: Set "oliver30". Best tour length found = 423.741. From TSPLIB: shortest cycle length (if each distance is rounded to the nearest integer) = 420.

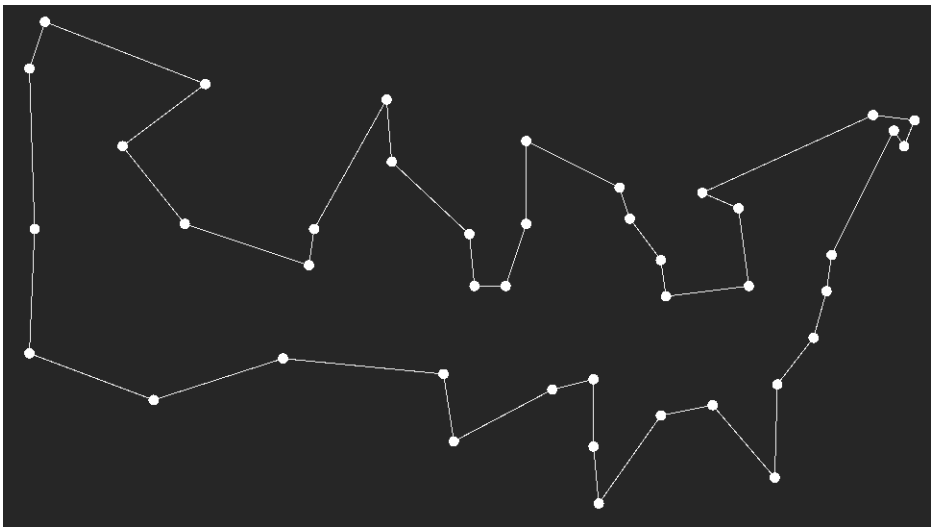


Figure 27: Set "dantzig42". Best tour length found = 682.992. From TSPLIB: shortest cycle length (if each distance is rounded to the nearest integer) = 699.

Ant Colony Optimization settings ↓

All the parameters used to set up the ACO algorithm.

`numAnts` ("m") is the number of ants in each group.

`dstPower` ("β") controls the extent to which ants will prefer nearby points: too high and the algorithm will essentially be a greedy search, too low and the search will likely stagnate.

`pheromonePower` ("α") controls how likely each ant will be to follow the same paths as the ants before it: too high and algorithm will keep searching the same path, too low and it will search too many paths.

`evaporationRate` ("ρ") is the proportion of pheromone that evaporates each step (also called pheromone decay parameter).

`exploitationThreshold` ("q_0") is the probability that the best desiderability will be chosen: represents the rate of exploitation versus exploration.

`pheromoneIntensity` ("Q") is the parameter that substitutes ρ in Ant Optimization techniques.

`initialPheromoneIntensity` sets the initial pheromone strength along all paths, since otherwise initial probabilities will all be 0. Later in development it became obsolete with the introduction of L_{nn} (see Chapter 3.1), but as other options it's still kept for completeness.

`choosePheromoneIntensity` is a boolean that basically switches between the Ant Op-

timization (a primitive version of the model [10]) and Ant Colony Optimization techniques; affects the use of `pheromoneIntensity` and `initialPheromoneIntensity`.

`reinforcingACS` is a boolean that implements the Reinforcing ACS [36] [35], changing the correction rule in the Local update and re-initializing trails over an upper bound in the Global update. Unfortunately, even with re-initialization, empirical tests proved this mode to converge too quickly, which makes it prone to getting stuck in a local minimum.

Debug ↓

Used during debugging or to gain additional info on the program execution.

`dstSetInfo` is a boolean that, if set, displays the smallest and average distance between nodes of the current node set in the Console log as soon as the algorithm is started.

`timeBetwIterations` specifies the time in seconds between each ACO algorithm iteration. The parameter makes it easier to debug and also to take screenshots of an evolving graph. If set to 0 there will be no artificial pause, but the algorithm execution will be momentarily interrupted anyway to get the current time for the Duration timer parameter, check for any pressed button or any settings edit in the Inspector, and to update the graph drawing: in this way, the algorithm behaves like a thread, and waits for other tasks to be updated before resuming.

4.2 Wrapper: ACO-DTSP Statistical Analysis

To assess the quality of the model and compare the algorithm variations used, a statistical study was implemented. The main program "Ant Colony Optimization DTSP", explained in the previous chapter, was wrapped in a higher-level interface: the resulting program is the "ACO-DTSP Statistical Analysis".
For this reason, the scripts used are the same as before, with the addition of the Wrapper script, that, as mentioned, wraps the fist program to allow for multiple subsequent executions.

The wrapper program lets the user choose all the parameters previously listed, plus some newer ones:

- number of iterations;
- number of repetitions.

The number of iterations specifies how many iteration loops the ACO-DTSP algorithm keeps running before stopping, with each iteration ending when the Global Updating Rule occurs (see Algorithm 4).
The number of repetitions specifies how many times the whole ACO-DTSP algorithm is being initialized from scratch, started and consequently stopped.

The Stop button was removed since with the new iteration feature the algorithm stops on its own when the last repetition terminates its last iteration.

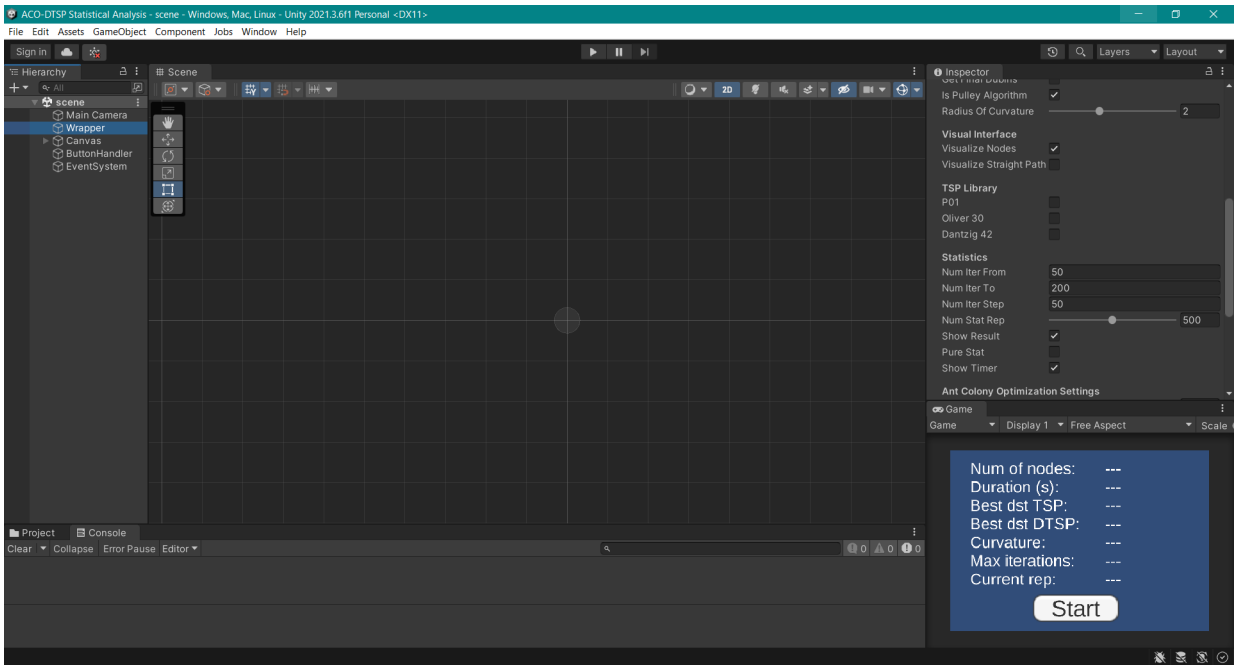


Figure 28: Unity GUI for the "ACO-DTSP Statistical Analysis" project.

The program was specifically created to automatize the report of statistically significant data: to get the most out of a single run and minimize the time spent modifying parameters,

the program runs on *batches*, or arrays of successive repetitions with the same number of maximum iterations. After each batch, the best tour length average is computed and then displayed in the Console log.

To define the new described features, a new module of parameters was added in the Inspector.

Statistics ↓

`numIterFrom` is the starting number of iteration. The program will perform its first batch of repetitions using this number of iterations before stopping.

`numIterTo` is the ending number of iterations. The last repetition batch in the overall execution will use this number of iterations.

`numIterStep` is the iteration resolution between batches. For each new batch the number of maximum iterations will increase of a step specified by this parameter.

`numStatRep` is the number of repetitions for each batch: the higher, the more accurate the resulting statistics will be, but it also obviously increases the total execution time.

`showResult` is a boolean that lets you choose if you want the program to display the final tour for every repetition before getting to the next one. If checked, the algorithm stops for 1 second at the end of each repetition to show the resulting best tour.

`pureStat` is a boolean that allows the program to block any other task except for the ACO-DTSP algorithm. It makes the execution considerably faster since it doesn't have to check for changes in the Inspector and doesn't have to draw anything on every repetition; it just gives the resulting statistics at the end of each batch.

`showTimer` is a boolean that lets the user choose if the duration is displayed or not.

5 Statistical analysis

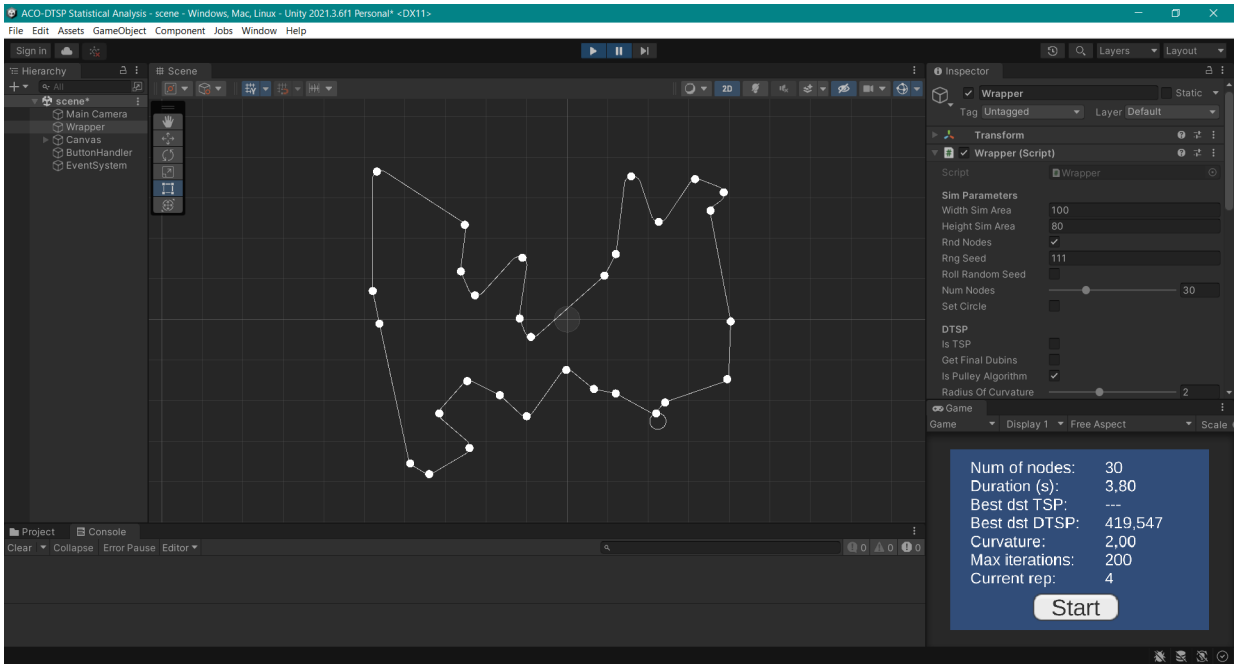


Figure 29: "ACO-DTSP Statistical Analysis" project in execution.

As seen in Chapter 3.1, the ACO algorithm by construction heavily relies on random chance to decide new paths to explore: even with the same initial parameters and the same number of iterations, two executions could return different final values.
To prevent the return values to be skewed by random bias, the wrapper stores the results from every ACO-DTSP complete repetition, and then, when all the required repetitions are finished, it displays the arithmetic average of the results.

For every statistics applied the number of repetitions was set to 500, as a compromise between statistical accuracy and feasible computation times. Also, in the following Tables the classical TSP will be denoted as ETSP, which stands for *Euclidean Traveling Salesman Problem*, since the cost of each edge is given by the euclidean distance between the nodes (a straight line in 2D space).

The ACO parameters used for every statistics implemented are:
$m = 15$, $\alpha = 1$, $\beta = 2$, $\rho = 0.7$, $q_0 = 0.7$.

5.1 ETSP and TSPtoD

Num of points	Radius	Iterations	Best ETSP average	Best TSPtoD average
20	2	50	357.433	364.368
20	2	100	357.057	363.246
20	2	150	356.433	361.773
20	2	200	356.595	361.880
20	4	50	357.387	408.878
20	4	100	356.877	410.103
20	4	150	356.692	411.376
20	4	200	356.311	411.280
30	2	50	397.536	423.114
30	2	100	397.417	423.571
30	2	150	397.242	423.264
30	2	200	397.262	423.577
30	4	50	397.466	536.125
30	4	100	397.363	537.231
30	4	150	397.290	536.327
30	4	200	396.995	535.251
50	2	50	516.176	573.813
50	2	100	511.354	568.503
50	2	150	511.135	568.435
50	2	200	509.472	567.072
50	4	50	516.060	974.457
50	4	100	511.225	972.684
50	4	150	509.999	973.945
50	4	200	508.878	973.783

First the program was made to run an optimization for the ETSP tour, and as it found the best ETSP path for every repetition, the final tour was converted in a Dubins path before ending (this procedure used the PA and is designated as "TSPtoD" in the Tables); this was done to later compare it with DTSP optimization. Also, the ETSP lengths given are the baseline both for the ACO algorithm operations and for the node set itself, as the ETSP is made of straight lines and as such always returns the shortest possible tour compared to any other DTSP-variation. For ETSP tours it was chosen to document new lengths when the turning radius was changed, even if the change only affects Dubins paths.

As expected, it can be seen that the ETSP lengths are always shorter than their TSPtoD counterpart; additionally, for both variations, the tour lengths tend to get smaller as the number of iterations gets bigger.
It's interesting to note though that for the TSPtoD the inverse proportion of number of

iterations and tour length is less noticeable or sometimes downright absent: this is due to the fact that since the optimization is done specifically on ETSP tours, it's entirely possible that a worse ETSP tour leads to a better DTSP tour found. In fact, the problem of solving first the ETSP is that in a dense area of waypoints (where the minimum turn-radius is too big compared to the smallest distance between two nodes) the modifications done to the best path is more cost-effective, producing a bad DTSP tour in most cases. This also explains why this difference is even more noticeable for a bigger turning radius.

For each set is then reported the smallest distance between nodes, to allow a comparison with the used radius, and the average distance between nodes.

Num of points	Min dst	Avg dst	Min dst > Rad 2	Min dst > Rad 4
20	5.324	47.489	✓	✓
30	3.481	45.910	✓	✗
50	0.661	47.510	✗	✗

This information holds true for the following Tables too, since the sets were kept the same while the solver was modified.
It can be seen that the 20-points set has a minimum distance that is higher than both the used radii, for the 30-points set it's higher than the second radius, and for the 50-points set is lower than both. This explains why more drastic variations in tour length are observed when the radius is increased for larger sets.

5.2 Complete DTSP with PA

Num of points	Radius	Iterations	Best TSPtoD average	Best DTSP average
20	2	50	364.368	**362.702**
20	2	100	363.246	**361.490**
20	2	150	361.773	**360.761**
20	2	200	361.880	**360.419**
20	4	50	408.878	**373.658**
20	4	100	410.103	**372.517**
20	4	150	411.376	**372.045**
20	4	200	411.280	**371.362**
30	2	50	423.114	**420.360**
30	2	100	423.571	**419.793**
30	2	150	423.264	**419.936**
30	2	200	423.577	**419.441**
30	4	50	536.125	**505.774**
30	4	100	537.231	**502.286**
30	4	150	536.327	**500.031**
30	4	200	535.251	**500.033**
50	2	50	573.813	**569.828**
50	2	100	568.503	**564.098**
50	2	150	568.435	**561.668**
50	2	200	567.072	**560.052**
50	4	50	974.457	**846.952**
50	4	100	972.684	**829.531**
50	4	150	973.945	**817.679**
50	4	200	973.783	**812.145**

Then the complete ACO-DTSP is analysed (thus with the Pulley Algorithm as its DSPP component), and compared with the previous results of the TSPtoD. For comparisons the best tour lengths are reported in bold to identify them more easily.

As it's shown, all the resulting best DTSP tours are shorter than their TSPtoD counterpart, especially when running with a bigger turning radius: this highlights the optimizing power of the program, with up to a maximum of 16.6% decrease in tour length.
The difference is so evident that even considering the highest max iterations for the TSPtoD and the lowest max iterations for the DTSP, the DTSP performance is still similar or better.

It's also worth noting how the DTSP manages to better exploit the tour when the turning radius is bigger than the minimum inter-node distance, avoiding loops or other headings configurations that may stretch the total path length.

5.3 AA-PA comparison

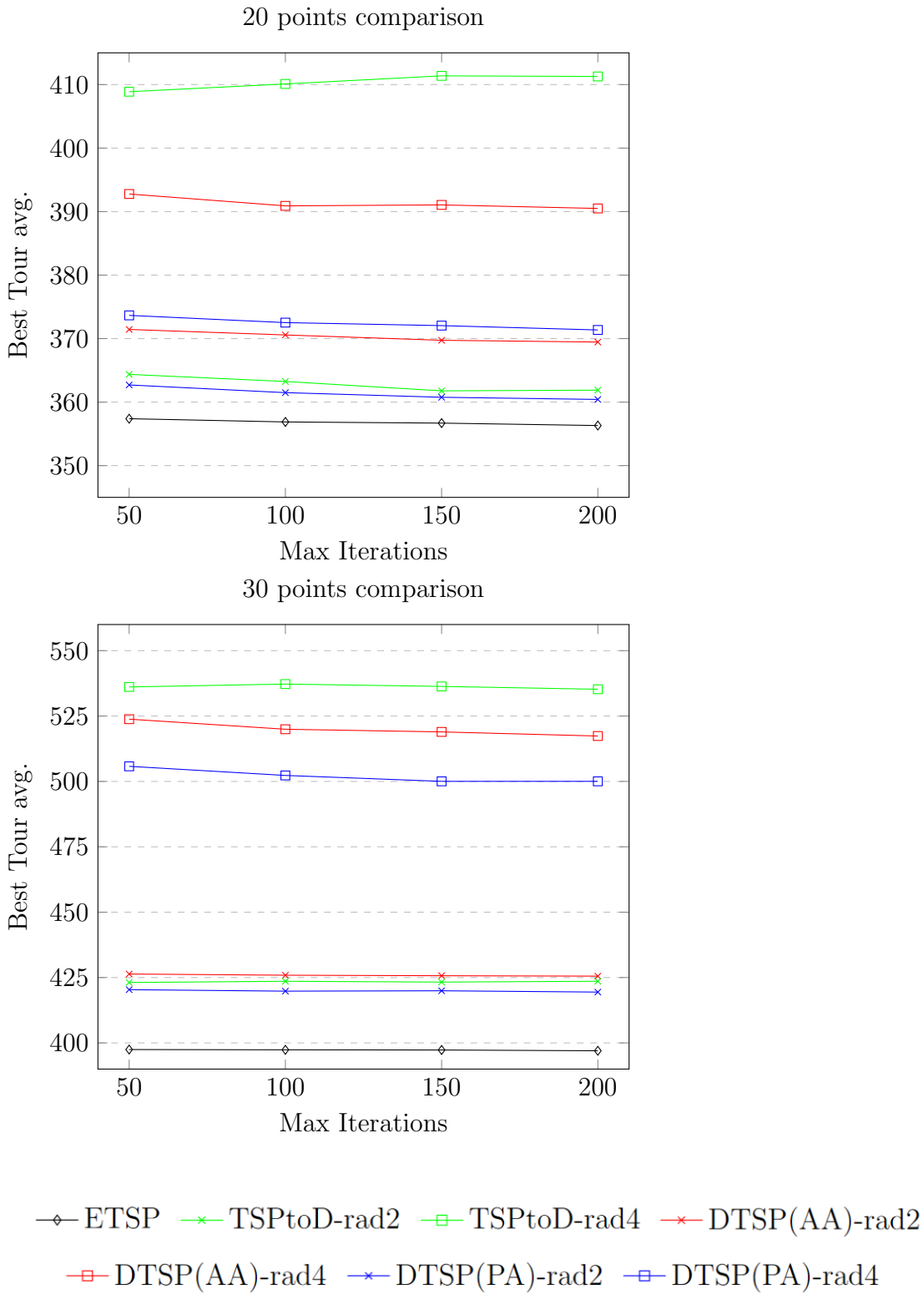
Num of points	Radius	Iterations	Best DTSP avg. (AA)	Best DTSP avg. (PA)
20	2	50	371.440	**362.702**
20	2	100	370.574	**361.490**
20	2	150	369.737	**360.761**
20	2	200	369.457	**360.419**
20	4	50	392.771	**373.658**
20	4	100	390.903	**372.517**
20	4	150	391.053	**372.045**
20	4	200	390.496	**371.362**
30	2	50	426.348	**420.360**
30	2	100	425.889	**419.793**
30	2	150	425.691	**419.936**
30	2	200	425.538	**419.441**
30	4	50	523.811	**505.774**
30	4	100	519.954	**502.286**
30	4	150	518.932	**500.031**
30	4	200	517.364	**500.033**
50	2	50	603.901	**569.828**
50	2	100	596.627	**564.098**
50	2	150	593.546	**561.668**
50	2	200	591.481	**560.052**
50	4	50	856.339	**846.952**
50	4	100	843.202	**829.531**
50	4	150	835.735	**817.679**
50	4	200	832.842	**812.145**

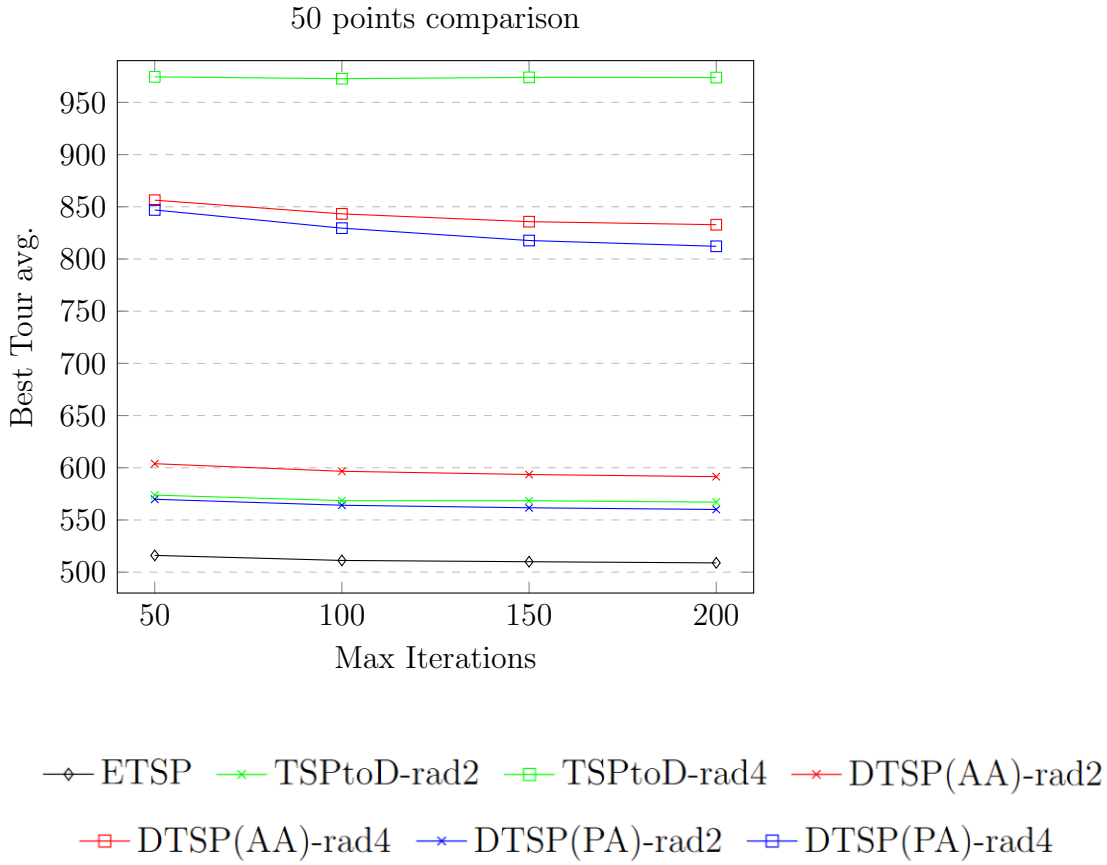
Lastly, is reported a comparison of DTSP tours using either the Alternating Algorithm or the Pulley Algorithm.

The PA outperforms the AA in every category: this is due to the nature of the PA, that organically adapts to the considered path, as the Dubins transformation is practically form-fitting to the node net. Again, is possible to observe that as the number of iterations gets higher11 the tour lengths get shorter, except for some random outliers.

The PA seems able to consistently shrink the total length, and the difference of best tour lengths is rather uniform across the board and doesn't considerably change for larger sets. However, since the tour lengths get bigger, this means that the rate of improvement actually decreases: this could be attributed to the denseness of the studied sets, since as it was recorded the minimum distance between points gets smaller and smaller as the number of points grows.

5.4 Final comparison





To help visualize the comparison between the different modes, three line graphs for the 20, 30 and 50 points sets are shown. The ETSP line is displayed as baseline for each set.

The AA is shown to perform significantly better than the TSPtoD when the turning radius is bigger, but it's worse otherwise. In general, it's trivial to assume that as the turning radius tends to 0, the ideal best DTSP tour tends to the ideal best ETSP: the simple AA forces the path to add curves haphazardly every other edge, and in the long run this approach still loses to the TSPtoD when the latter uses the PA.

Generally, the DTSP using the PA is again proven to be the best method out of all those analysed, from computing speed to overall optimizing performance.

6 Conclusions

This paper explores the potential of the ACO algorithm combined with a DSPP solver, and presents the empirical outcome of the resulting DTSP optimization; the ACO-DTSP algorithm shows positive results as a cost-effective approach to the problem.

The modular nature of the program with respect to the DSPP sub-algorithm allows for further research in this scope; future works may thus implement new methods for heading search.
Besides, the ACO algorithm itself offers many settings that can be finely tuned to get the best results. One way to explore them to the fullest could be through the use of artificial intelligence iterative algorithms, such as the generalized genetic algorithm: each setting could be seen as a gene, each ACO-DTSP algorithm running with those specific settings could be an individual in the population, and with the final best tour length set up as the fitness function.

A limit for this research was the absence of a library of sets for the DTSP, complete with the best tour length recorded for every set, similar to the concept of TSPLIB [21] for TSP tours. This would make it easier to measure the performance of the proposed algorithm, as well as any other DTSP algorithm.
Usually these kinds of libraries rely on supercomputers that use exact algorithms which check for every possible tour (or a reasonably pruned subset) to get the best result with absolute certainty, at the cost of an expensive use of resources and high computation times, and then collect them as a list of sets on a public website that can be freely consulted. The only problem for this concept is that while for the TSP this can be done in a discrete searching space, since the number of possible TSP tours for a set of points is factorial with n, but it's nonetheless *finite*, the DTSP pairs it with the search for the optimal headings, making it a mix type continue search space. One option could be to use a finely discretized heading choice to get as close of a result to the global best as possible, added to the waypoint order, to return to a unified discrete space.

References

[1] D. Anisi. "Optimal Motion Control of a Ground Vehicle". In: (Jan. 2003).

[2] L. Babel. "New heuristic algorithms for the Dubins traveling salesman problem". In: *Journal of Heuristics* 26 (Aug. 2020). DOI: 10.1007/s10732-020-09440-2.

[3] Mike Ball. *Drone Delivery Canada Showcases Long-Range Heavy Cargo Drone*. 2019. URL: https://www.unmannedsystemstechnology.com/2019/02/drone-delivery-canada-showcases-long-range-heavy-cargo-drone/.

[4] G. Beni and J. Wang. "Swarm Intelligence in Cellular Robotic Systems". In: vol. 102. Jan. 1993, pp. 703–712. ISBN: 978-3-642-63461-1. DOI: 10.1007/978-3-642-58069-7_38.

[5] James Billington. *Postal service begins testing drone delivery*. 2015. URL: https://www.ibtimes.co.uk/postal-service-begins-testing-drone-delivery-1519713.

[6] A. Bond and L. Gasser. "Readings in Distributed Artificial Intelligence". In: (Jan. 1988).

[7] Giacinto Bottone. *Patrol robot to help industrial workers avoid risky environments*. 2020. URL: https://innovationorigins.com/en/patrol-robot-to-help-industrial-workers-avoid-risky-environments/.

[8] Rebecca Cairns. *More than 50 robots are working at Singapore's high-tech hospital*. 2021. URL: https://edition.cnn.com/2021/08/25/asia/cgh-robots-healthcare-spc-intl-hnk/index.html.

[9] I. Cohen et al. "Discretization-Based and Look-Ahead Algorithms for the Dubins Traveling Salesperson Problem". In: *IEEE Transactions on Automation Science and Engineering* PP (Sept. 2016), pp. 1–8. DOI: 10.1109/TASE.2016.2602385.

[10] A. Colorni, M. Dorigo, and V. Maniezzo. "Distributed Optimization by Ant Colonies". In: Jan. 1991.

[11] Subham Datta. *Traveling Salesman Problem – Dynamic Programming Approach*. 2022. URL: https://www.baeldung.com/cs/tsp-dynamic-programming.

[12] M. Dorigo. "Optimization, Learning and Natural Algorithms". PhD thesis. Politecnico di Milano, Italy, Jan. 1992.

[13] M. Dorigo and L.M. Gambardella. "Ant Colonies for the Traveling Salesman Problem". In: *Bio Systems* 43 (Feb. 1997), pp. 73–81. DOI: 10.1016/S0303-2647(97)01708-5.

[14] M. Dorigo and L.M. Gambardella. "Ant Colony System: A cooperative learning approach to the Traveling Salesman Problem". In: *IEEE Transactions on Evolutionary Computation* 1 (May 1997), pp. 53–66. DOI: 10.1109/4235.585892.

[15] M. Dorigo, V. Maniezzo, and A. Colorni. "Ant system: Optimization by a colony of cooperating agents". In: *IEEE Trans. Syst., Man, and Bybern, Part B* 26 (Jan. 2002), pp. 29–41.

[16] L.E. Dubins. "On Curves of Minimal Length with a Constraint on Average Curvature, and with Prescribed Initial and Terminal Positions and Tangents". In: *American Journal of Mathematics* 79.3 (July 1957), pp. 497–516. ISSN: 00029327, 10806377. DOI: 10.2307/2372560.

[17] C.A. Feinstein. "The Computational Complexity of the Traveling Salesman Problem". In: (Nov. 2006). DOI: https://doi.org/10.48550/arXiv.cs/0611082.

[18] Agence France-Presse. *Robot border guards among new airport tech at Paris Air Show*. 2015. URL: https://www.defencetalk.com/robot-border-guards-among-new-airport-tech-at-paris-air-show-64592/.

[19] L.M. Gambardella and M. Dorigo. "Solving symmetric and asymmetric TSP by ant colonies". In: June 1996, pp. 622–627. ISBN: 0-7803-2902-3. DOI: 10.1109/ICEC.1996.542672.

[20] W. Ghadiry et al. "Optimal Path Tracking With Dubins' Vehicles". In: *IEEE Systems Journal* PP (July 2020), pp. 1–12. DOI: 10.1109/JSYST.2020.3006990.

[21] Ruprecht-Karls-Universität Heidelberg. *TSPLIB - A Traveling Salesman Problem Library*. 1991. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/.

[22] M. Held and R. Karp. "A Dynamic Programming Approach to Sequencing Problems". In: *Journal of the Society for Industrial and Applied Mathematics* 10 (Mar. 1962), pp. 196–210. DOI: 10.2307/2098806.

[23] K. Helsgaun. "An Effective Implementation of the Lin-Kernighan Traveling Salesman Heuristic". In: *European Journal of Operational Research* 126 (Oct. 2000), pp. 106–130. DOI: 10.1016/S0377-2217(99)00284-2.

[24] Clay Mathematics Institute. *Millennium Problems*. 2000. URL: https://www.claymath.org/millennium-problems.

[25] Sebastian Lague. *Coding Adventure: Ant and Slime Simulations*. 2021. URL: https://www.youtube.com/watch?v=X-iSQQgOd1A.

[26] Patrick Lancaster. *cse490r_18wi*. 2018. URL: https://gitlab.cs.washington.edu/cse490r_18wi/lab4.git.

[27] J. Le Ny, E. Feron, and E. Frazzoli. "On the Dubins Traveling Salesman Problem". In: *IEEE Trans. Automat. Contr.* 57 (Jan. 2012), pp. 265–270. DOI: 10.1109/TAC.2011.2166311.

[28] X. Ma and D. Castanon. "Receding Horizon Planning for Dubins Traveling Salesman Problems". In: Jan. 2007, pp. 5453–5458. DOI: 10.1109/CDC.2006.376928.

[29] Bernard Marr. *Demand For These Autonomous Delivery Robots Is Skyrocketing During This Pandemic*. 2020. URL: https://www.forbes.com/sites/bernardmarr/2020/05/29/demand-for-these-autonomous-delivery-robots-is-skyrocketing-during-this-pandemic/?sh=614019287f3c.

[30] Kayla Matthews. *Singapore's newest security officer is a robot*. 2019. URL: https://betanews.com/2018/11/19/singapores-newest-security-officer-is-a-robot/.

[31] Richard Moss. *Underwater robot provides first detailed, high-resolution 3D maps of Antarctic sea ice*. 2014. URL: https://newatlas.com/seabed-auv-high-resolution-3d-maps-antarctic-sea-ice/34924/.

[32] NavalDrones. *SeaExplorer*. URL: http://www.navaldrones.com/SeaExplorer.html.

[33] V. Ojha, A. Abraham, and V. Snasel. "ACO for Continuous Function Optimization: A Performance Analysis". In: Nov. 2014, pp. 145–150. DOI: 10.1109/ISDA.2014.7066253.

[34] Jeongsoo Park and Dongho Lee. "Precise Inspection Method of Solar Photovoltaic Panel Using Optical and Thermal Infrared Sensor Image Taken by Drones". In: *IOP Conference Series: Materials Science and Engineering* 611 (Oct. 2019), p. 012089. DOI: 10.1088/1757-899X/611/1/012089.

[35] C.M. Pintea and D. Dumitrescu. "Improving ant systems using a local updating rule". In: Oct. 2005, 4 pp. ISBN: 0-7695-2453-2. DOI: 10.1109/SYNASC.2005.38.

[36] C.M. Pintea, P. Pop, and C. Chira. "The Generalized Traveling Salesman Problem solved with Ant Algorithms". In: *J Univers Comput Sci* 13(7)rem (Aug. 2017). DOI: 10.1186/s40294-017-0048-9.

[37] Gerhard Reinelt. *Data for the Traveling Salesperson Problem*. URL: https://people.sc.fsu.edu/~jburkardt/datasets/tsp/tsp.html.

[38] Matthew Reitman. *Underwater Robot Helps Monitor Arctic Ice Shelf Collapse*. 2017. URL: https://www.insidehook.com/article/science/underwater-robot-helps-predict-arctic-ice-shelf-collapse.

[39] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis II. "An Analysis of Several Heuristics for the Traveling Salesman Problem". In: *SIAM Journal on Computing* 6 (Sept. 1977), pp. 563–581. DOI: 10.1137/0206041.

[40] Debbie Saunders et al. "Radio-tracking wildlife with drones: a viewshed analysis quantifying survey coverage across diverse landscapes". In: *Wildlife Research* 49 (Feb. 2022). DOI: 10.1071/WR21033.

[41] K. Savla, E. Frazzoli, and F. Bullo. "Traveling Salesperson Problems for the Dubins Vehicle". In: *IEEE Transactions on Automatic Control* 53 (Aug. 2008), pp. 1378–1391. DOI: 10.1109/TAC.2008.925814.

[42] A.M. Shkel and V. Lumelsky. "Classification of the Dubins set". In: *Robotics and Autonomous Systems* 34 (Mar. 2001), pp. 179–202. DOI: 10.1016/S0921-8890(00)00127-5.

[43] T. Stützle and H.H. Hoos. "Improvements on the Ant-System: Introducing the MAX-MIN Ant System". In: (Jan. 1996). DOI: 10.1007/978-3-7091-6492-1_54.

[44] A. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Vol. 3. Jan. 2002.

[45] Alan Turing. "Computing Machinery and Intelligence". In: vol. 59. Oct. 1950. DOI: 10.1093/mind/LIX.236.433.

[46] VCG. *Chuzhou Uses Robots To Guarantee Power Supply For College Entrance Exam Period*. 2016. URL: https://www.gettyimages.ch/detail/nachrichtenfoto/two-robots-inspect-the-equipment-in-a-220kv-electrical-nachrichtenfoto/538526112.

[47] A.B. Walker. "Hard real-time motion planning for autonomous vehicles". PhD thesis. Swinburne University, 2011.

[48] University of Waterloo. *TSP*. 2005. URL: https://www.math.uwaterloo.ca/tsp/index.html.

[49] X. Yu and J. Hung. "A genetic algorithm for the Dubins Traveling Salesman Problem". In: *IEEE International Symposium on Industrial Electronics* (May 2012). DOI: 10.1109/ISIE.2012.6237270.