Path Planning with Drones at CSP plants

Adrián Gutiérrez Camacho

Path Planning with Drones at CSP plants

Adrián Gutiérrez Camacho

Trabajo de fin de máster que forma parte de los requisitos para la obtención del título de Máster Universitario en Matemáticas por la Universidad de Sevilla.

Dirigido por

José Miguel Díaz Báñez

Agradecimientos:

En primer lugar, me gustaría agradecer la inestimable ayuda de mi tutor, José Miguel Díaz-Báñez, que ha sabido encauzar el trabajo y darme los hilos de los cuáles ir tirando a lo largo de su desarrollo. También a mi jefe, el profesor Juan Sebastián Valverde por ofrecernos un problema tan interesante y sobre todo realista. La aplicación de las matemáticas es algo que siempre me ha fascinado. También querría agradecerle a mi compañero José Morales, experto ya en el arte de las inspecciones en plantas CSP, por darme toda la información de la que disponía para poder hacer el trabajo de forma más realista. De igual manera, le agradezco a mi compañero Miguel Ángel Pérez-Cutiño las aportaciones de ideas y correcciones que ha hecho del trabajo.

Por último, pero no menos importante, me gustaría agradecerle a todas las personas que forman parte de mi vida personal y que son las que hacen posible mi dedicación al trabajo de investigación: a mi pareja Miriam, que es el motor que me impulsa a seguir siempre adelante, a mis amigos y amigas, que son capaces de sacarme de la cabeza todas las cuentas y ecuaciones y dejarme solo una sonrisa, y a mi familia, sin la cual nunca me hubiera propuesto estos objetivos y por supuesto nunca los habría alcanzado.

Abstract

The goal of this work is to apply mathematics knowledge and skills to efficiently solve a practical problem posed by the industry. We study an actual problem related to the inspection of Concentrated Solar Power (CSP) plants. Due to the big extension of solar fields, Unmanned Aerial Vehicles (UAV), commonly called drones, are used to inspect all the tubes of the CSP plant. We introduce a new problem, named the drone CSP inspection problem, that aims the computation of the tours to be performed by the drone in order to cover the CSP plant so that some penalization function is minimized. Specifically, we take into account two objective functions: the total time or the number of refills. First, we model the energy consumption of the UAV and the individual time inspection costs in a realistic fashion and use them as inputs for the procedures described. We also propose several formulations adapting classical optimization problems. In addition, we prove that this particular problem is NP-complete and develop some heuristics. An extensive comparison against the current approach adopted by the industry shows best performance of our algorithms, saving a considerable amount of time for inspection.

Resumen

El objetivo de este trabajo es aplicar conocimiento y habilidades matemáticas para resolver eficientemente un problema práctico propuesto por la industria. Estudiaremos un problema real relacionado con la inspección de plantas de concentración solar de potencia (CSP). Debido a la gran extensión de los campos solares se utilizan vehículos aéreos no pilotados (UAV), comúnmente llamados drones, para inspeccionar todos los tubos de la planta CSP. Introduciremos un nuevo problema, el problema de inspección CSP con drones, donde se propone calcular las trayectorias a realizar por el dron de manera que se cubra la planta CSP mientras se minimiza una cierta función de penalización. Concretamente, tendremos en cuenta dos funciones objetivo: el tiempo total de inspección y el número de recargas que el dron necesita. Primero, modelaremos el consumo de energía del UAV y los tiempos individuales de inspección de forma realista y los usaremos como entrada de los procedimientos descritos. Propondremos varias formulaciones adaptando problemas de optimización clásicos. Además, probaremos que este problema particular es NP-completo y desarrollaremos algunos heurísticos. Comparando éstos con procedimiento actual adoptado por la industria, probamos que nuestros algoritmos tienen un mayor rendimiento, ahorrando una considerable cantidad de tiempo total de inspección.

Contents

1 Introducing the problem. State of the art

It is known that the sunlight strikes the earth with more energy per hour that all of the energy consumed by humans per year [1]. The solar energy is bigger than all other renewable and fossil-based energy resources combined. Energy is nowadays an indispensable good but in most cases the energy demand doesn't match its availability. Thus, to provide a durable and widespread primary energy source, solar energy must be captured, but also stored and used in a cost-effective fashion.

Solar energy has an unsteady nature, it varies during the days, which can be, for example, sunnier or more cloudy, and along the year, because seasons impact deeply in sun rays. In Figure 1.1 we can see the world solar energy map.

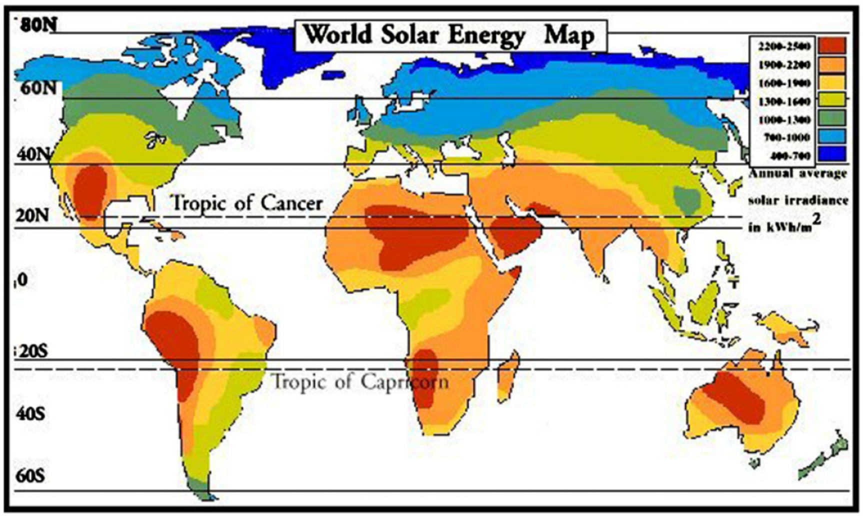

Figure 1.1: World solar energy map (borrowed from [1])

In this context, Concentrated Solar Power (CSP) plants are gaining increasing interest, mostly by using the Parabolic Trough Collector system (PTC), although Solar Power Towers (SPT) progressively occupy a significant marker position due to their advantages in terms of higher efficiency, lower operating costs and good scale-up potential.

The main problem that CSP plants face is the varying solar radiation flux throughout the day/year. To enhance the CSP process, and improve the overall yield in comparison with older systems, plants usually incorporate technologies like Thermal Energy Storage (TES) and Backup Systems (BS), which facilitate a continues and year round operation and provide a stable energy supply in response to electricity demand. In Figure 1.4 there is a representation of a TES in a PTC, which summarizes the performance of a typical CSP plant of this type. In order to determine the optimum design and operation of a CSP plant along the year, an accurate estimation of the daily solar irradiation is needed. Also, CSP plants will only accept Direct Normal Irradiance (DNI) to operate. In [1] the authors outline a procedure to calculate the hourly beam irradiation flux.

Concentrated solar power (CSP) is an electricity generation technology that uses heat provided by solar irradiation concentrated on a small area. Using mirrors, sunlight is reflected to a receiver where heat is collected by a thermal energy carrier (primary circuit), and subsequently used directly (in the case of water/steam)) or via a secondary circuit to power a turbine and generate electricity. CSP is particularly promising in regions with high DNI. There are mainly four available CSP technologies (see Figure 1.2): parabolic trough collector (PTC), solar power tower (SPT), linear Fresnel reflector (LFR) and parabolic dish systems (PDS). These CSP technologies are currently in medium to large-scale operation and mostly located in Spain and in the USA, and also there are some projects for designing CSP plants at North Africa. In this document we will focus on PTC plants, since it is the type of plant which originated the problem addressed in this thesis, but we will describe the other plants because they could lead to problems that may be presented, or solved, in a similar way.

Solar power towers (SPT), also known as central receiver systems (CRS), use a heliostat field collector (HFC), i.e, a field of sun tracking reflectors, called heliostats, that reflect and concentrate the sun rays onto a central receiver placed in the top of a fixed tower. Heliostats are flat or slightly concave mirrors that follow the sun in a two axis tracking. In the central receiver, heat is absorbed by a heat transfer fluid (HTF), which then transfers heat to heat exchangers that power a steam Rankine power cycle. Some tower plants use direct steam generation, others use fluids like molten salts as

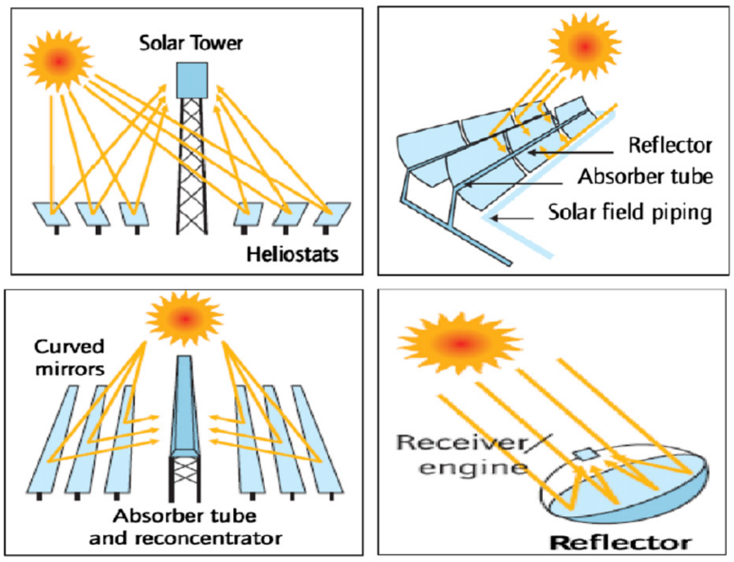

Figure 1.2: Mainly available CSP technologies, in lecture ordering: STP, PTC, LFR and PDS (borrowed from [1])

HTF and storage medium. The concentrating power of the tower concept achieves very high temperatures, thereby increasing the efficiency at which heat is converted into electricity and reducing the cost of storage. Furthermore, the concept is highly flexible, where designers can choose from a wide variety of heliostats, receivers and transfer fluids. Some plants can have several towers to feed one power block.

Parabolic trough collector (PTC) plants consist of a group of reflectors (usually silvered acrylic) that are curved in one dimension in a parabolic shape to focus sun rays onto an absorber tube that is mounted in the focal line of the parabola. The reflectors and the absorber tubes move in tandem with the sun as it daily crosses the sky from the sunrise to sunset. The group of parallel connected reflectors is called the solar field. Typically, thermal fluids are used as primary HTF, thereafter powering a secondary steam circuit and Rankine power cycle. Other configurations use molten salts and others use a direct steam generation system. The absorber tube (Figure 1.3), also called heat collector element (HCE), is a metal tube and a glass envelope covering it, with either air or vacuum between these two to reduce the convective heat losses and allow for thermal expansion. The metal tube is coated with a selective material that has high solar irradiation absorbance and low thermal remittance. The glass-metal seal is crucial in reducing heat losses.

Linear Fresnel reflectors (LFR) approximate the parabolic shape of the trough sys-

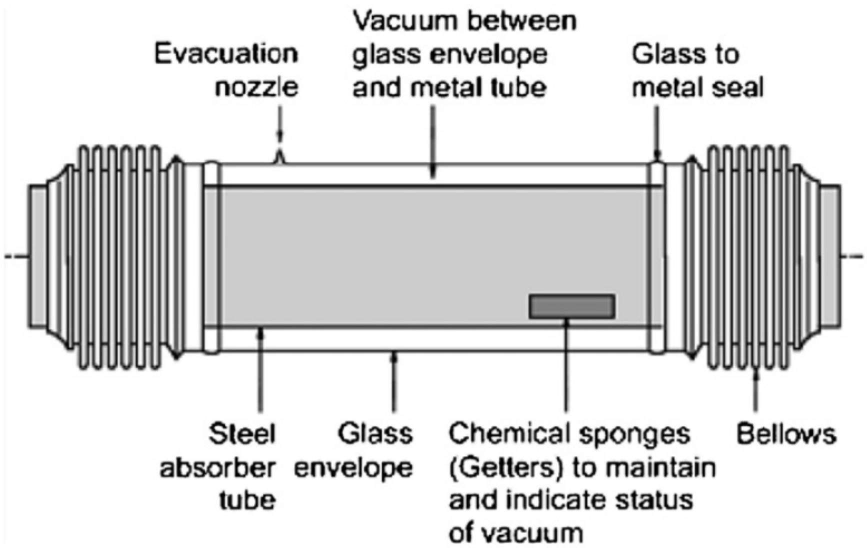
Vacuum between glass envelope and metal tube
Evacuation nozzle
Glass to metal seal
Steel absorber tube
Glass envelope
Chemical sponges (Getters) to maintain and indicate status of vacuum
Bellows

Figure 1.3: Absorber element of a parabolic trough collector (borrowed from [1])

tems by using long rows of flat or slightly curved mirrors to reflect the sun rays onto a downward facing linear receiver. The receiver is a fixed structure mounted over a tower above and along the linear reflectors. The reflectors are mirrors that can follow the sun on a single or dual axis regime. The main advantage of LFR systems is that their simple design of flexibly bent mirrros and fixed receivers requires lower investment costs and facilitates direct steam generation, thereby eliminating the need of heat transfer fluids and heat exchangers. LFR plants are however less efficient than PTC and SPT in converting solar energy to electricity. Moreover, it is more difficult to incorporate storage energy into their design.

Parabolic dish collectors (PDC) concentrate the sun rays at a focal point supported above the center of the dish. The entire system tracks the sun, with the dish and receiver moving in tandem. This design eliminates the need for a HTF and for cooling water. PDCs offer the highest transformation efficiency of any CSP system. PDCs are expensive and have a low compatibility with respect of thermal storage and hybridization. Promoters claim that mass production will allow dishes to compete with larger solar thermal systems. Each parabolic dish has a low power capacity (typically tens of kW or smaller), and each dish produces electricity independently, which means that hundreds or thousands of them are required to install a large scale plant like built with other CSP technologies.

Within the commercial CSP technologies, parabolic trough collector (PTC) plants are the most developed of all commercially operating plants. In terms of land occupancy, PTC requires more land than SPT and LFR to produce a given output. PDC has the smallest land requirement among CSP technologies. Water requirements are of high importance for those locations with water scarcity, for example, in most of the desserts. CSP requires water for cooling and condensing processes. Dry cooling (using air instead of water) can be an effective alternative.

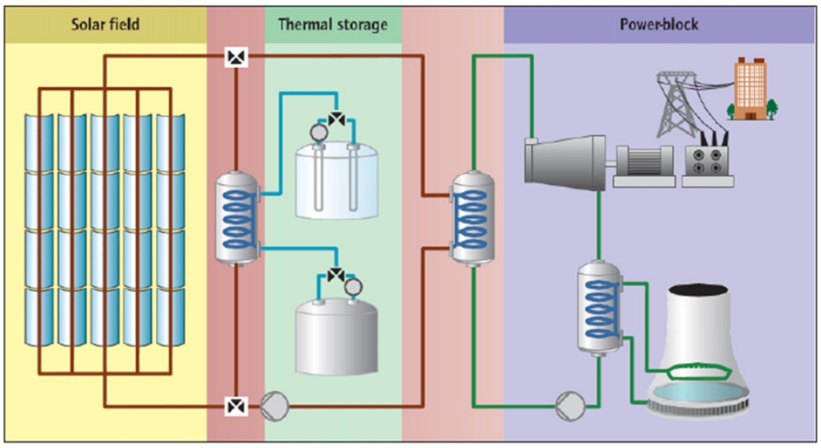

Figure 1.4: Thermal energy storage system in parabolic trough collector plant (borrowed from [1])

1.1 The drone CSP inspection problem

In this work, the goal is to study a drone path planning problem in CSP plants. We will focus on PTC ones, but this problem can be, at least partially, extended to other CSP plant types or even to another optimization problems. CSP plants are composed of regions or solar fields with absorber tubes or HCE (heat collector element) which warm up some fluid to very high temperatures. The HCEs are covered with glasses which are usually broken due to vibrations and other phenomena, and this leads to heat losses (a pipe whose glass is broken will have leak of temperature in the fluid inside) which, at the end, provoke a reduction in electrical availability.

Due to big extension of solar fields, it can be very hard to identify broken pipes in order to fix them and recover the good performance of the plant. To overcome this issue, a drone is used to inspect all the absorber tubes of the CSP plant, one-by-one. This drone flies over the plant and takes thermal pics by means of an special camera. Then, the drone CSP inspection problem or CSP problem, for short, asks for the computation of the trajectories followed by the drone along the CSP plant so that some penalization function is minimized: the needed time or the number of refills. The constraint of the problem is the autonomy of the drone that can be set to 30 min. Therefore, our goal is to study the complexity of the CSP problem and propose optimal or sub-optimal solutions. To accomplish that, we explore various related optimization problems in the literature, adapt them to our CSP problem and develop heuristic algorithms based on the known heuristics for these problems. We use the Python programming language to develop tools for solving, representing and computing the solutions associated to each approach.

Moreover, we are specially interested in the practical value it might have for companies engaged in drone inspections. In fact, this work has been developed in the framework of Virtualmechanics[1], a competitive company in the field of plants inspections. They put a real problem on the table and also have been reporting to us all the information of particularities of the problem so that we could work with a realistic model. One of the purposes of this thesis is to improve the performance of the current drone inspections that this company is conducting in CSP plants.

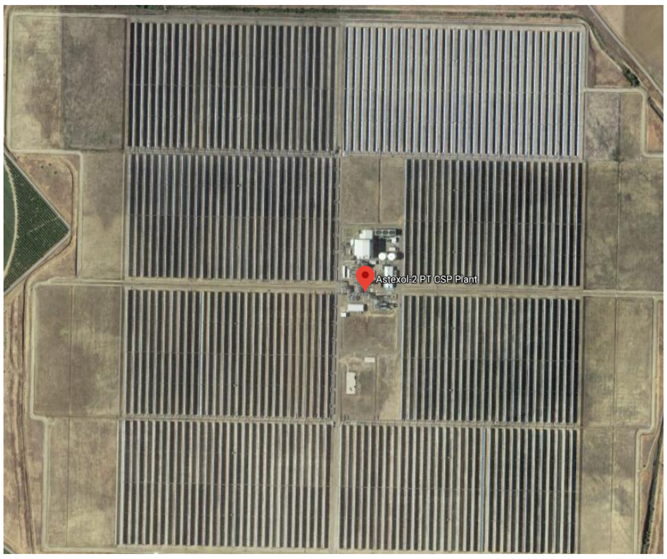

Figure 1.5: Typical CSP plant top view

Firstly, in order to state the model, we need to introduce the targets or points to be visited by the drone. In a CSP plant the absorber elements are lied in couples:

[1]https://virtualmech.com/

the upstream tube and the downstream tube, one aside the other. Another constraint of our problem is that these couples have to be analyzed consecutively by the drone (although they can be inspected in any order). Because of this fact, we will model the upstream and downstream pipes as one point to visit, instead of two. Then the input of the problem is just a cloud of n points S and we want to describe a set of tours starting and ending at a given base station so that all the points are visited. In addition, this couples are placed making big extensions called batches. Figure 1.6 shows two faced batches with 41 elements to be covered. So, actually, we can identify our CSP plant with a cloud of n points in $\mathbb{R}^2$ which are also separated in b batches. Unfortunately, due to the limited battery life of the used small drone, the size of the problem becomes very large if we consider the entire CSP plant as an instance. Moreover, the CSP plants are delicate environments where accuracy is required and also laws usually do not allow to fly drones far away from the pilot. There are similar limitations associated with the RTK system and the drone antenna, which give the drone special accuracy in those environments. Thus, we will focus in how to efficiently cover a pair of faced CSP batches.

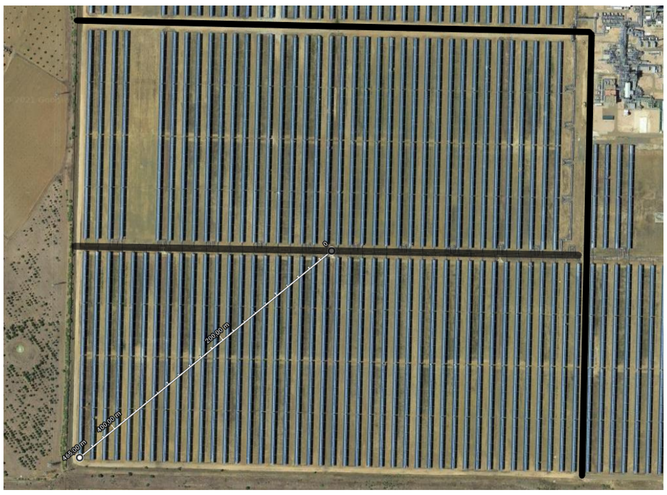

Figure 1.6: Picture with two faced CSP batches. The upper batch has 20 couples of pipes and the lower batch has 21. We identify each couple with an inspection node

The problem is how specifically the drone can visit all the inspection points so that the total time is minimized. More formally, given the complete digraph whose nodes are the n points and the edges cost is given by the time to travel between the nodes, which is the set of tours or cycles to visit all nodes of the graph in the minimum time?. The constraints for each tour are the following:

- All node which is different from the depot node (base station) must appear exactly in one tour. On the other hand, all the tours must start and end at the depot node. A set of tours verifying this and such that its union visits all the nodes in the given graph is called a *covering set.*
- The total energy consumption effectuated by the drone during a tour can not be grater than the total battery level of the drone. In fact, we will use a 75% to 90% of the maximum battery level because of security reasons. A tour with this characteristic is called *feasible tour.*

The CSP instances are defined by: the complete digraph G generated from the n inspection points and the depot node, the energy costs w_{ij}, the inspection time costs t_{ij} and the maximum level of battery W. In this scenario we consider two criteria giving rise to the following CSP problems:

1. *CSP problem 1:* How could we design a covering set of feasible tours for a given CSP instance minimizing the total inspection time?
2. *CSP problem 2:* How could we design a covering set of feasible tours for a given CSP instance minimizing the total number of tours or refills the drone makes?

In this work, we usually use the term CSP problem for the two problems defined above. However, the problems are different, as we will see in future sections, because a solution for one could not be a solution for the other. However, due to the symmetry of the instances, we will see that the two solutions can be quite similar. Moreover, we could approximate the minimum time cost covering set by the covering set with the lesser number of tours. We could also wonder about where to place the battery station but, in fact, the places where it can be put are limited to the roads around the CSP plant, with some of them crossing it. This is an interesting facility location problem that is out of the scope in this work. We will suppose that the battery station is placed in the road below or above any of the two faces batches, in a position whose x coordinate is the mean of the x coordinates of the inspection nodes in the nearest batch.

1.2 Related classical problems

In this section we will refer to some classical path planning problems that appear in the literature that come to mind to relate with the CSP problem. We also will see some recent related work. The notations and concepts will be useful in the CSP modelling that we are going to develop in the next chapters but, we do not use these problems to solve the CSP instances. Suppose we have a directed graph $G = (V, E)$ where V contains the n nodes or points to be visited by the drone and E is the set of all the possible arcs or paths between two different nodes in V, i.e, G is a complete digraph. The costs are asymmetric because of the wind. Let us define the decision variables $x_{ij} \in \{0, 1\}$ so that $x_{ij} = 1$ if and only if we use the arc (i, j) in our solution. For sake of simplicity, a tour in G may be given as an ordered subset of V, $\{i_1, i_2, \ldots, i_n\}$ which means we travel from i_j to i_{j+1}, $j = 1, \ldots, n-1$ and then from i_n again to i_1. The first known optimization problem to be related with the CSP problem is the *Travelling Salesman Problem (TSP)* whose linear formulation can be given ([2]) as:

$$(TSP) \quad \min \quad \sum_{i,j=1}^{n} c_{ij}x_{ij} \tag{1.1a}$$

$$\text{s.t.:} \quad \sum_{i=1, i\neq j} x_{ij} = 1, \quad j = 1, \ldots, n \tag{1.1b}$$

$$\sum_{j=1, j\neq i} x_{ij} = 1, \quad i = 1, \ldots, n \tag{1.1c}$$

$$u_i - u_j + nx_{ij} \leq n - 1, \quad 2 \leq i \neq j \leq n \tag{1.1d}$$

$$u_i \in \{1, 2, \ldots, n-1\}, \quad i = 2, \ldots, n \tag{1.1e}$$

$$x_{ij} \in \{0, 1\} \tag{1.1f}$$

This is known as the MTZ formulation of TSP. The objective function (1.1a) minimizes the total travel cost. The constraints (1.1b) and (1.1c) ensure that each node of G is reached by an arc exactly once and that only another one leaves from it. (1.1d) are the tour elimination constraints: using the dummy variables $u_2, \ldots, u_n$ (excluding 1 as it is a depot node) which are forced to grow up in value with the order that the tour has. The last constraints (1.1e) and (1.1f) are the domains of the decision variables. However, if we want to be more realistic, we should deal with the fact that the drone usually can not inspect all the nodes in graph without returning to the battery station (BS). In fact, the limit of duration in past real CSP inspections has been usually about 9 of these points (around 30 minutes of battery). We have a certain kind of *capacity limitation* in our problem. Because of that, the duration of the battery will be an additional parameter to consider in the model. Also, we must include the BS in our graph as the depot or 0-node. Capacity and depot are key words that remind to the *Vehicle Routing Problem (VRP)*, which we have formulated here in a Dantzig, Fulkerson and

Johnson way:

$$
\begin{aligned}
(VRP) \quad \min \quad & \sum_{i,j=1}^{n} c_{ij} x_{ij} && \text{(1.2a)}\\
\text{s.t.:} \quad & \sum_{i \in V, i \neq j} x_{ij} = 1, \quad j \in V \backslash \{0\} && \text{(1.2b)}\\
& \sum_{j \in V, j \neq i} x_{ij} = 1, \quad i \in V \backslash \{0\} && \text{(1.2c)}\\
& \sum_{i \in V} x_{i0} = K && \text{(1.2d)}\\
& \sum_{j \in V} x_{0j} = K && \text{(1.2e)}\\
& \sum_{i \notin S} \sum_{j \in S} x_{ij} \geq r(S), \quad \forall S \subseteq V \backslash \{0\}, S \neq \emptyset && \text{(1.2f)}\\
& x_{ij} \in \{0,1\}, \forall i,j \in V && \text{(1.2g)}
\end{aligned}
$$

In this formulation, K is the number of available vehicles (the number of tours or charges needed by the drone to visit all the inspection points) and $r(S)$ the minimum number of vehicles needed to serve a set S. The VRP optimal solution is the minimal low-cost set of tours, which start and end in a depot node, that reach all the nodes of the graph while ensuring enough capacity in each of them. In our problem we have a vehicle (drone) which has to visit n nodes, starting from the BS depot, while watching its battery duration. When the drone reaches a minimum amount of battery it has to come back to the base station and start a new one tour. The main goal is to plan the tours in order to minimize the time we spent in the CSP inspection. Although this model has similarities with our problem, this formulation is still too simple.

A more advanced version of VRP is the *Capacitated Vehicle Route Problem (CVRP)* [3]. Its formulation is pretty similar to the previous one, but we must add some new components. In this case, let $G = (V, H, c)$ a complete directed graph with $V = \{0, 1, 2, \ldots, n\}$ as the nodes and $H = \{(i,j) \colon i, j \in V, i \neq j\}$ as the arcs, where node 0 is the depot for a fleet of p vehicles with the same capacity Q. Each node $i \in V \backslash \{0\}$ has a certain positive demand $d_i \leq Q$. We also consider a travel cost c_{ij} associated with each arc $(i,j) \in H$. The minimum number of vehicles to needed to serve all customers is $\lceil \frac{\sum_{i=1}^{n} d_i}{Q} \rceil$ (note we could also add a battery efficiency coefficient to consider uncertainty). The binary decision variables $x_{rij} \in \{0,1\}$ are defined to indicate if vehicle r, $r \in \{1, 2, \ldots, p\}$, traverses an arc $(i, j) \in H$ in an optimal solution. Then, the linear programming model of the CVRP can be written as:

$$(CVRP) \quad \min \quad \sum_{r=1}^{p} \sum_{i=0}^{n} \sum_{j=0, i \neq j}^{n} c_{ij} x_{rij} \tag{1.3a}$$

$$\text{s.t.:} \quad \sum_{r=1}^{p} \sum_{i=0, i \neq j}^{n} x_{rij} = 1, \quad j \in V \backslash \{0\} \tag{1.3b}$$

$$\sum_{j=1}^{n} x_{r0j} = 1, \quad r \in \{1, \ldots, p\} \tag{1.3c}$$

$$\sum_{i=0, i \neq j}^{n} x_{rij} = \sum_{i=0}^{n} x_{rji}, \quad j \in V, \ r \in \{1, \ldots, p\} \tag{1.3d}$$

$$\sum_{i=0}^{n} \sum_{j=1, i \neq j}^{n} d_j x_{rij} \leq Q, \quad r \in \{1, \ldots, p\} \tag{1.3e}$$

$$\sum_{r=1}^{p} \sum_{i \in S} \sum_{j \in S, i \neq j} x_{rij} \leq |S| - 1, \quad \forall S \subseteq V \backslash \{0\}, S \neq \emptyset \tag{1.3f}$$

$$x_{rij} \in \{0, 1\}, \forall r \in \{1, \ldots, p\}, \ i, j \in V, i \neq j \tag{1.3g}$$

The objective function (1.3a) minimizes the total travel cost of all the vehicles in the fleet. The degree constraints (1.3b) ensure that each node is visited by exactly one vehicle. The flow constraints (1.3c) and (1.3d) guarantee that each vehicle can leave the depot only once, and the number of the vehicles arriving at each customer and entering the depot is equal to the number of the vehicles leaving. The capacity constraints (1.3e) make sure that the sum of the demands of the customers visited in a route is less than or equal to the capacity of the vehicle performing the service. The tour elimination constraints (1.3f) ensure that the solutions contains no cycles disconnected from the depot. The last constraints (1.3g) specify the definition domains of the variables. This model is known as a *three-index vehicle flow formulation* and has the drawback that the number of inequalities in (1.3f) grows exponentially with the number of nodes.

A more related problem is studied in [4]. Although the costs and topologies in their graphs are different to the ones in a CSP problem, the goal is the same: a drone has to visit a set of points S, while recharging its battery in some locations of a discrete set C, minimizing the total trip time. They studied how to estimate, through a regression model, the battery consumption of determined drone flight actions. The regression uses the basic kinematics of the drone as prediction variables. We want to take advantage of this work using their model to compute realistic energy consumption weights. The authors also develop some heuristics to give approximations to the problem. Mainly, their contributions are:

1. A model the energy consumption of drones, considering various flight scenarios.
2. A study on the joint problem of flight tour planning with recharging optimization for drones with the goal of completing a tour mission for a set of locations of interest in the shortest time.
3. A real implementation of their algorithms in an intelligent drone management system, which supports real-time flight path tracking and re-computation in dynamic environments.

Since we borrow some notation of that paper, we include here more details on their model and frormulation. Consider a set of cities S that a drone has to visit, and a set of charging stations C where a drone can charge its battery. The base location or depot of drones is denoted by $\{v_0\}$. The problem is to find a sequence of locations or *flight plan* in $S \cup C$, such that the drone can start its tour at v_0, visit all sites in S and then returns to the depot, with the objective of minimizing the total trip time, while keeping the *SoC (state-of-charge)* within the operational range.

Given a pair of locations (u, v), the *flight path* between them is noted by $l(u, v)$, and the *flight time* by $\tau(u, v)$. Let $f(l(u, v), \tau(u, v))$ be the energy consumption related to a drone travelling along $l(u, v)$ within flight time $\tau(u, v)$. Also, let $V = S \cup C \cup \{v_0\}$. For a drone flying between two sites $u, v \in V$, there is a battery consumption by an amount of $n_d f(u, v)$. If the drone returns to a charging station $u \in C$, it can recharge its battery by an amount of energy $n_c b(u)$. The coefficients n_d and n_c stand for discharging and charging efficiency performance. In addition, when recharging its battery there will be a incurred charging time denoted as $\tau_c(b(u))$. Let $T \subseteq S \cup C \cup \{v_0\}$ be a flight plan and denote its k-th location by T_k. In order to find a flight plan which minimizes the total trip time (travel and charging time) it is needed to impose that $T_1 = T_{|T|} = v_0$.

Let x_k be the SoC when arriving at T_k, assuming that $x_k \in [B_0, B_1]$, that is, the SoC lies within a feasible range. The lower bound B_0 ensures sufficient energy for the drone to return to the depot. Then, the drone flight plan optimization problem (DFP) is formulated as follows:

$$
\begin{aligned}
(DFP) \quad & \min_{T,b(\cdot),x} \sum_{k=1}^{|T|-1} \tau(T_k, T_{k+1}) + \sum_{k=1:\, T_k \in C}^{|T|} \tau_c(b(T_k)) && \text{(1.4a)} \\
& \text{s.t.:} \quad T_1 = T_{|T|} = v_0 && \text{(1.4b)} \\
& S \subseteq T \subseteq S \cup C \cup \{v_0\} && \text{(1.4c)} \\
& x_k = \begin{cases} x_{k-1} - n_d f(l(T_k, T_{k+1}), \tau(T_k, T_{k+1})) \\ \text{if } T_k \in S \\ x_{k-1} + n_c b(T_{k+1}) - n_d f(l(T_k, T_{k+1}), \tau(T_k, T_{k+1})) \\ \text{if } T_k \in C \end{cases} && \text{(1.4d)} \\
& B_0 \leq x_k \leq B_1 && \text{(1.4e)}
\end{aligned}
$$

This formulation accounts for the situation of recharging the drone without filling up the entire battery. Our CSP inspection problem is not as complex in that aspect: at C the drone replaces its battery obtaining $x_k = B_1$. Also, for us, $C = \{v_0\}$, that is, our depot is the only charging station. The authors then developed some heuristic algorithms to apply its DFP problem and found lemmas to support its work with some theory.

In [5], the authors study the Drone Arc Routing Problems (DARP) and its relation with the Postman Arc Routing Problems. The main difference is that drones can travel directly between any two points in the plane without following the edges of the network. Thus the DARP is a continuous optimization problem with infinite feasible solutions. However, they solve it as a discrete optimization problem by approximating curves as polygonal chains. In their research, they consider a set of drones that must jointly service a set of edges of a network. Drone start and end at a specified vertex (depot) and have a route length limit that applies to the total route (capacity). Arc routing problems consist of, given a network, and given a set of lines that are required to be covered, each one with an associated cost, finding a tour covering all the required lines with total minimum cost. Typical examples are the Chinese Postman Problem , the Rural Postman Problem. They give same results relating optimal drone tours with optimal postman tours. Specifically, the Drone Arc Routing Problem is: Given a set of lines or curves on the plane, each one with an associated service cost, and a point called the depot, and assuming that the cost of deadheading between any two points of the plane is the Euclidean distance, find the minimum cost tour starting and ending at the depot that services all the given lines and curves. They develop an algorithm which uses upper bounds and branch-and-cut procedures for solving instances of their problems. After that, they present a procedure for generating random DARP instances and show the results obtained. In addition, they explore how the problem can generalize to have several vehicles instead of one. In our work, the components of the graph to be covered are not the edges but vertices. Also, our graphs present a simple topology that we want to use to design the algorithms.

In [6] a system onboard an UAV to monitor CSP plants using open source hardware and software was developed with customization capabilities depending on use. The thermal inspection of absorber tubes at a CSP plant was performed. The proposed methodology is more efficient than traditional methods based on walking with a thermal gun through the solar plant. They also try to reduce the inspection time although their inspections seem less intensive than ours. In [7] heuristic methods for the task allocation and collision-free path planning for three robots working in a industrial plant inspection is developed. There were ninety fixed locations in a plant, which were to be inspected by three robots after traveling through the minimum distance. Moreover, overall task completion time was to be as minimum as possible. A genetic algorithm (GA) was used for the task allocation, and A* algorithm was utilized for path planning. A* algorithm is a graph search technique used to find a path from a given initial node to the pre-specified goal node. On the other hand, Genetic Algorithm (GA) is a population-based probabilistic search and optimization technique based on Darwin's principle of natural selection. The matematical formulation of their problem made the following assumptions: each robot executes only one task at a time, only one robot is required for each task, each task is executed only once, all task must be executed and all robots start from the depots at the same time. Another interesting problem that we could state in our work is doing CSP inspections using more than one drone, and then distribute the inspection nodes among the drones.

A review of path planning, routing algorithm and routing protocols is presented in [8]. They compare algorithms and methods in order to find the best ones for each a application. Among the introduced UAV path planning algorithms are: conventional ones, cell based, model based (linear, for example), and learning based (neural networks, evolutionary). The work is a detailed state of the art of methods to solve these type of problems. In [9] the authors present the Energy Efficient Coverage Path Planning problem, which resembles a bit to our CSP problem. They perform measurements to understand the energy consumption of a drone. The authors studied the consumption of the drone after travel a straight line distance, the effect of velocity and the effect of turning. They prove that turning most increases energy consumption, as we had suppose to happen. The EECPP problem goal is to cover an arbitrary area containing obstacles using multiple drones minimizing the maximum energy required for any individual drone to traverse its assigned path. Their problem seemed NP-hard so they developed some heuristic approaches, improving previous proposed algorithms and results.

In the next sections our purpose is to model the CSP problem, propose some MIP formulations and some heuristics, and also evaluate them using realistic instances. In addition, given that we know the current path planning strategy which pilots perform during inspections, we can compare this *current solution* with those obtained by the proposed methods.

2 | Some formulations for the CSP problem

In this chapter our aim is to find some formulations which may seemed suitable, and also worthy, for the CSP problem. It is pretty obvious that the graphs made from CSP plant layouts present a lot of symmetry, and we must take advantage of that. However, if we want to obtain realistic exact solutions we have to develop formulations and exact approaches. After solving exact instances of the CSP problems we might be able to discover certain patterns. These patterns could guide us to develop heuristics for the problem.

Now, we are going to widely explain how CSP inspections work. To accomplish the inspection, CSP plant owners subcontract a maintainer which carries out the task of collect the needed data, process it and then send an evaluation report to the client. This work focuses in how the data is collected so we do not give many details of the processing and evaluation parts. The data used by the maintainers is composed by mainly consecutive pics from the pipes in a CSP plant. These pics are taken with a thermal camera which is integrated into a drone and they show the potential heat leak points in pipes. Also, the maintainer must, at the same time, subcontract a pilot in order to use a drone. Due to delicacy of the environment and the potential appearance of errors in the drone flight, it is not possible to only rent a drone without its pilot. Generally, drone pilots use KML files to record the tours that their drones will travel in given inspections. KML files are used, among others, by Google Earth software. Each tour is saved as a collection of coordinates or waypoints to be visited by the drone during the inspection. In each experiment, the CSP plant is divided in subsets of pipes called missions. The sets are usually composed of 3 pipe couples, although sometimes they can only have two of them or have the third couple separated from the others (and halved). In Figure 2.1 we have plotted a cloud of points, corresponding to an actual full CSP plant, partitioned in 54 missions which have been extracted from 54 different KML files. They were used in a real CSP inspection and the numbers reflect the order in which the inspection was performed, which is the current path planning solution adopted by the company.

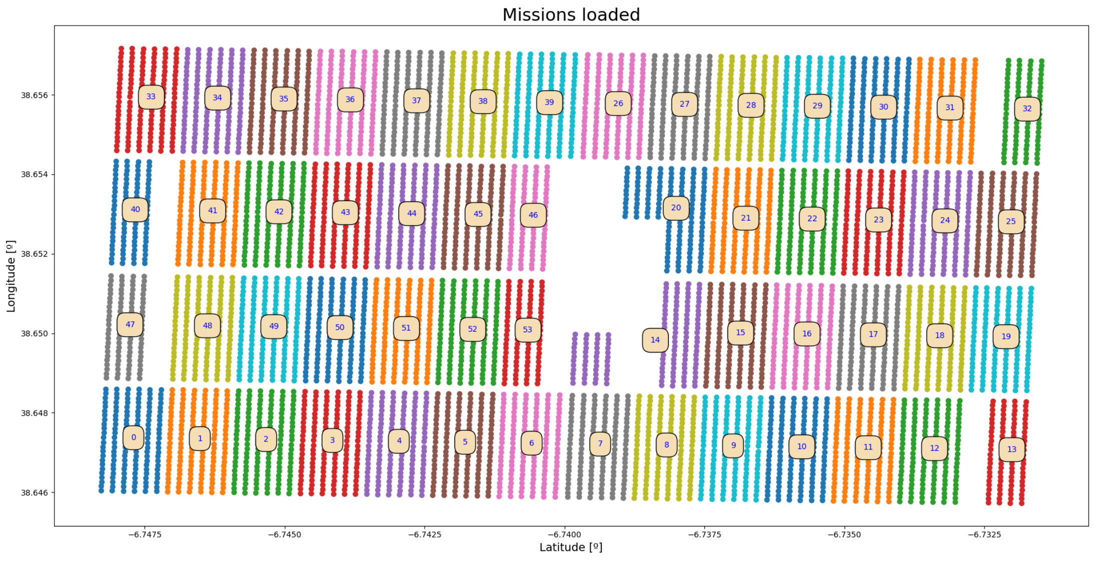

Figure 2.1: Cloud of points from KML files partitioned in missions in an actual CSP inspection.

The path planning strategy during the inspection related to Figure 2.1 consisted of sequentially visiting all the couples in each mission, and recharging the battery when needed (this usually happens after doing 3 or 4 missions of 3 points each one). This current solution (we will refer it with this name in future comparisons) might not be the best because it implies many changes of drone flight direction, which have been proved to increase the time and battery costs.

Along this section we will present different approaches to find exact solutions for CSP instances. To do that, we will use some classical problems as the *Knapsack Problem* (specifically in its generalization, the *Multiple Knapsack Problem*), the *Bin Packing Problem* and the *Exact Cover Problem*. Before that, we must define the weights and costs for travelling between the nodes of each instance graph.

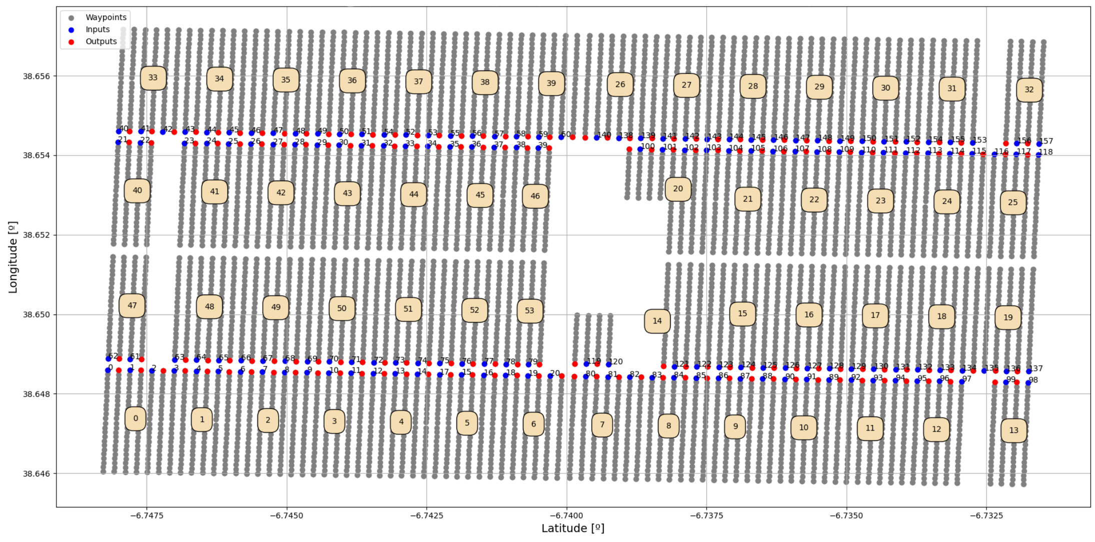

Figure 2.2: Cloud of points extracted from KML files used by drone pilot in actual CSP inspections. Every blue and red point pair stands for a couple of upstream and downstream pipes

2.1 Modelling the energy consumption

In order to control the SoC (state-of-charge) we will need to compute the energy costs that battery suffers due to displacements effectuated by the drone. We will model 4 different trajectory types:

- Takeoff: when doing a tour, the drone starts at the base or charging station point, at ground level, and it has to take off from it in order to visit the interest points. We will assume that takeoffs and landings are done in vertical. We will use the notation A_t for this action.
- Horizontal displacement: the drone moves in XY plane at some altitude which we will assume as constant. This is the trajectory made between points of interest and from/to the base station (the drone first takes off in vertical until reaches some altitude and then does an horizontal displacement to the first node in tour or comes from the last node doing an horizontal displacement and then lands in vertical). Its notation will be A_h.
- Landing: when finishing a tour, the drone has to return to base station and land in it. This vertical displacement, noted A_l, is the counterpart of takeoff.

- Turning: optionally, but very often, a drone effectuates a turn to head its objective (in general, a node to inspect) and then, at the arrival, to head parallel to the pipe. In addition, during the inspection phase, inside a node, the drone needs to turn at least 3 times, in U shape, to visit the entire couple defining a point of interest. It is important to remark that one of the most expensive displacements a drone can do are the ones which imply more turnings, in the sense of increasing the time and energy costs. We will refer to turns as A_g.

The notations for each type of action will help us to differentiate states of the energy consumption parameters, as their values change with the action performed. Let $[0, T] = [0, t_1] \cup [t_1, t_2] \cup [t_2, T]$ be the *time interval* during the performance of a given action. Also, each displacement type (except turnings) will be decomposed in three parts:

1. Acceleration part: the drone increases its speed in the direction of movement (horizontal, vertical or negative vertical) by using propellers. This is an uniformly accelerated rectilinear motion (linear acceleration is negative). This part lasts from $t = 0$ to $t = t_1$.
2. Uniform part: the drone moves with constant speed in the desired direction. This is an uniformly rectilinear motion (linear acceleration is zero). It lasts from $t = t_1$ to $t = t_2$.
3. Braking part: the drone decreases its speed in the direction of movement, until it is 0. This is an uniformly decelerated rectilinear motion (linear acceleration is positive). It lasts from $t = t_2$ to $t = T$.

Turning is considered to be uniform motion, so it has neither acceleration nor braking parts. So now, we can distinguish an action or displacement by its type, $A \in \mathcal{A} = \{A_t, A_h, A_l, A_g\}$, and the time interval when it occurs, $t \in [0, t_1]$, $[t_1, t_2]$ or $[t_2, T]$. In addition, note that the total acceleration of an object, $\vec{a_T}$, can be decomposed as linear acceleration, $\vec{a}$, plus the gravity acceleration, $\vec{g}$. We will define the gravity vector[1] as $\vec{g} = [0, 0, -9.81] m/s^2$. For example, in order to positively accelerate while ascending, the drone has to generate a positive vertical linear acceleration Z component.

Moreover, we must use some formula to compute the energy consumption from drone kinematics. For this we will use [4] and also [10]. In the articles, authors de-

[1]We will assume that Z axis is perpendicular to ground and positive while ascending.

fine a standard regression model which allows to compute the instantaneously power consumption, $\hat{P}$, as function of $\boldsymbol{s} = [\vec{v}_{xy}, \vec{a}_{xy}, \vec{v}_z, \vec{a}_z, \vec{\omega}_{xy}, m]$ where:

- $\vec{v}_{xy}$ and $\vec{a}_{xy}$ are the horizontal speed and acceleration vectors.
- $\vec{v}_z$ and $\vec{a}_z$ are the vertical speed and acceleration vectors.
- $\vec{\omega}_{xy}$ is the horizontal wind speed vector.
- m is the constant mass of the drone payload.

Then, $\hat{P}$ can be computed[2] as follows:

$$\hat{P}(\boldsymbol{s}) = \begin{bmatrix}\beta_1\\ \beta_2\\ \beta_3\end{bmatrix}^T \begin{bmatrix}||\vec{v}_{xy}||\\ ||\vec{a}_{xy}||\\ ||\vec{v}_{xy}||||\vec{a}_{xy}||\end{bmatrix} + \begin{bmatrix}\beta_4\\ \beta_5\\ \beta_6\end{bmatrix}^T \begin{bmatrix}||\vec{v}_z||\\ ||\vec{a}_z||\\ ||\vec{v}_z||||\vec{a}_z||\end{bmatrix} + \begin{bmatrix}\beta_7\\ \beta_8\\ \beta_9\end{bmatrix}^T \begin{bmatrix}m\\ \vec{v}_{xy}\cdot\vec{\omega}_{xy}\\ 1\end{bmatrix} = \beta^T\hat{\boldsymbol{s}}$$

where $\hat{\boldsymbol{s}} = [||\vec{v}_{xy}||, ||\vec{a}_{xy}||, ||\vec{v}_{xy}||||\vec{a}_{xy}||, \ldots, 1]$. If during a period T the parameters needed for computation remain unaltered, the energy consumption can be computed as $\hat{E} = \hat{P} \cdot T$. However, the reader would likely observe that 2/3 of each displacement type which we have defined is not constant in speed. To tackle this, we will generalize the energy consumption computation in a more analytic fashion. Let $\boldsymbol{s}$ be now a time dependent function modelling the kinematics state of the drone, which also depends on the action type performed. Fixed $A \in \mathcal{A}$, we could define the instantaneous energy consumption during an infinitesimal period dt (where the state $\boldsymbol{s}$ remains constant) as:

$$d\hat{E}(\boldsymbol{s}(t)) = \hat{P}(\boldsymbol{s}(t)) \cdot dt = \beta^T\hat{\boldsymbol{s}} \cdot dt$$

and then compute $\hat{E}(\boldsymbol{s}(t))$ integrating along $[0, T]$. In the Appendix A there is a more detailed explanation of how we can compute kinematics states for each action in order to obtain $\hat{E}$.

Now, we have all the ingredients needed to model the energy costs for visiting a point of interest in our CSP plant. We will distinguish between trips where the drone is coming from the base station node (*beginnings*), trips where the drone is going to the base station node (*endings*) and trips where the drone is travelling between points of interest (*normal trips*). In beginnings, the drone starts with a takeoff, heads (turning) to the next node to visit and then does an horizontal displacement. The endings start heading the depot node, horizontal movement is performed and then the landing happens. The normal trips are just turnings to the following node and

[2]Obviously we will need the values of $\beta = [\beta_1, \ldots, \beta_9]$ but we will discuss this question later.

horizontal displacements between nodes. There is an important detail in how the energy costs are modelled, they depend on current and previous visited node: let w^t_{ij} be the energy consumption of the drone when it is travelling from node i to node j, $i, j \in \{0, 1, \ldots, n\}$, $i \neq j$, that is,

$$w^t_{ij} = \hat{E}(\boldsymbol{s}(i, j))$$

where, here, $\boldsymbol{s}(i, j)$ models the parameter state the drone will present if it travels from i to j. However, we usually don't just visit the nodes but we also inspect them (except for the depot). The inspection cost associated to the node $i \neq 0$, w^e_i, can be computed creating fictitious nodes along pipes and performing normal trips between them. Recall that when we reach a point of interest and the inspection begins, the drone flies until the end of pipe, jumps to the other part of the couple, and finishes at the beginning of it. So, actually, the start position at node i and the end position after inspection doesn't match. But we can just create fictitious nodes i_1, i_2, i_3, i_4 and compute the energy costs by parts, $w^e_i = \hat{E}(\boldsymbol{s}(i_1, i_2)) + \hat{E}(\boldsymbol{s}(i_2, i_3)) + \hat{E}(\boldsymbol{s}(i_3, i_4))$, where i_1 stands for the start position coordinates at i, i_2 represents the coordinates at the end of the first pipes, i_3 plays as the coordinates at the end of the other part of the loop and i_4 defines the end position coordinates (see Figure 2.3). Clearly, we will use different i and j nodes coordinates when computing w^t_{ij} (output coordinates for i and input coordinates for j) that when getting w^t_{ji} (output coordinates for j and input coordinates for i). The depot node is the only one with the same input and output coordinates.

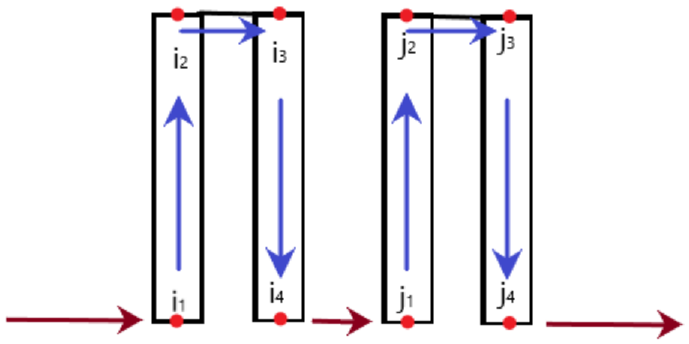

Figure 2.3: Fictitious nodes scheme for inspection cost computation at nodes i, j with trajectories of inspection (in blue) and between nodes (in red).

Besides the energy consumption costs, the time costs t^t_{ij} model the spent time of going from i to j. To compute them, we could use basic kinematics formulas and the distance between the nodes, $d(i,j)$ (if the speed is constant, the spent time equals to the distance divided by the speed). This is deepened in the Appendix A. There is also an inspection time cost, t_i^e, which can be computed creating fictitious nodes as w_i^e. To sum up, we define $t_{ij} = t_{ij}^t + t_i^e$ and $w_{ij} = w_{ij}^t + w_i^e$ as travel-inspection time costs and energy weights between nodes i and j, respectively.

In order to model the costs of coming from the base station and returning to it, we create a fictitious node $0'$ which is at the same XY coordinates as 0 but at some altitude above it, z_0. Then, the costs w_{0j}^t can decompose in a takeoff from 0 to $0'$ and an horizontal displacement from $0'$ to j (similar for w_{i0}^t, which implies a landing from $0'$ to 0). Evidently, $w_{i0} = w_{i0}^t$ because in this node there is no inspection. However, to be more realistic, the time costs at depot node will be $t_{i0} = t_{i0}^t + t^r$, where t^r is the recharging or battery replacement time cost.

2.2 Multiple Knapsack Problem approach

In this section we look at the CSP problem as an instance of the MKP. Suppose that now the n points to visit are items from a collection. We want to pack these items in k bins (tours or cycles) with k in $\{1,\dots,n\}$. Each of the items has a weight (associated energy consumption) and a profit, which is related with the time cost. At the same time, each bin has a maximum weight capacity (total battery energy when it is full of charge), which we denote by W. Then, we would like to choose the items that will be in each bin in the most efficient way, i.e, maximizing the profits while ensuring the weight of formed bins is less than W. The Multiple Knapsack Problem is known in the literature as an immediate extension of Knapsack Problem, and which is NP-hard in the strong sense [11].

The basic formulation for Multiple Knapsack problems can also be found in [11], and it is as follows: let n the number of items, m the number of knapsacks ($k \leq n$) and let $p_j =$ the profit of item j, w_j the weight of item j and c_i the capacity of knapsack i. We want to select m disjoint subsets of items so that the total profit of selected items is a maximum, and each subset can be assigned to a different knapsack whose capacity is no less than the total weight of items in the subset:

$$\begin{aligned}
(MKP) \quad \max \quad & z = \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij} && \text{(2.1a)} \\
\text{s.t.:} \quad & \sum_{j=1}^{n} w_j x_{ij} \leq c_i, \quad i \in M = \{1, \ldots, m\} && \text{(2.1b)} \\
& \sum_{i=1}^{m} x_{ij} \leq 1, \quad j \in N = \{1, \ldots, n\} && \text{(2.1c)} \\
& x_{ij} = 0 \text{ or } 1, \quad i \in M, \ j \in N && \text{(2.1d)}
\end{aligned}$$

Where $x_{ij} = 1$ if item j is assigned to knapsack i and $x_{ij} = 0$ otherwise. When $m = 1$ the MKP reduces to the 0-1 (single) knapsack problem.

However, we cannot use this formulation directly. Firstly, we want to pack all the items in any of the bins, but the constraint (2.1c) admits to not assign a item to any bin. In fact, another problem which is related to MKP and CSP problem is the *Generalized Assignment Problem*, which ensures that all the tasks or items are assigned to exactly one task processor or bin. Also, the profits and weights that model our problem depend on the previous visited point or, in knapsack terms, the last added item. These complications will be tackle in the following.

The main idea is to consider the nodes to visit as items and the tours as knapsacks with some capacity, i.e, the energy battery level when the drone is full of charge. In the following, we will develop a Mixed Integer Programming formulation which generalizes the MKP so that we can use it to solve instances of our particular problem.

2.2.1 Generalized MKP

Firstly, let $V = \{0, 1, \ldots, n\}$, $\overline{V} = \{1, \ldots, n\}$ and $K = \{1, \ldots, m\}$ be the items and knapsacks sets, respectively, where $m \leq n$ is fixed. If we recalled MKP formulation from Section 2, we would like to have just one w_j and p_j by node. However, the profits and the weights usually depend on previous visited node. To overcome this, let:

$$y_{ij} = \begin{cases} 1 & \text{if item } i \in V \text{ goes before item } j \in V \\ 0 & \text{otherwise} \end{cases}$$

Then, we can compute the weight of node j as $w_j = \sum\limits_{i \in V} w_{ij} y_{ij}$. To compute the profit of node j p_j, suppose that we have p_{ij} as the profit for going from node i to j, then $p_j = \sum\limits_{i \in V} p_{ij} y_{ij}$. On the other hand, let:

$$x_{jk} = \begin{cases} 1 & \text{if item } j \in V \text{ is assigned to knapsack } k \in K \\ 0 & \text{otherwise} \end{cases}$$

Actually, x_{jk} are the main decision variables in our problem that assign nodes to knapsacks. The extra y_{ij} variables are needed for our CSP problem. Then, the MIP formulation for the GMKP could state as follows.

$$
\begin{aligned}
(GMKP) \quad \max \quad & z = \sum_{i \neq j \in V} \sum_{k \in K} p_{ij} y_{ij} x_{ik} x_{jk} && \text{(2.2a)} \\
\text{s.t.:} \quad & \sum_{i \neq j \in V} w_{ij} y_{ij} x_{ik} x_{jk} \leq W, \quad \forall k \in K && \text{(2.2b)} \\
& \sum_{k \in K} x_{jk} = 1, \quad \forall j \in \overline{V} && \text{(2.2c)} \\
& \sum_{j \in V} x_{jk} = \sum_{i \neq j \in V} y_{ij} x_{ik} x_{jk}, \quad \forall k \in K && \text{(2.2d)} \\
& 2 y_{ij} \leq \max_{k \in K} x_{ik} + x_{jk}, \quad \forall i \neq j \in V && \text{(2.2e)} \\
& \sum_{k \in K} x_{0k} = m, \sum_{i \in \overline{V}} y_{i0} = m, \sum_{j \in \overline{V}} y_{0j} = m && \text{(2.2f)} \\
& \sum_{i \in V} y_{ij} = 1, \quad \forall j \in \overline{V}, \quad \sum_{j \in V} y_{ij} = 1, \quad \forall i \in \overline{V} && \text{(2.2g)} \\
& L_i + w_{ij} y_{ij} - W(1 - y_{ij}) \leq L_j && \text{(2.2h)} \\
& y_{ij} w_{ij} \leq L_j \leq W, \quad \forall i \in V, \forall j \in \overline{V}, \ i \neq j, && \text{(2.2i)} \\
& L_j \in \mathbb{R}, \ \forall j \in \overline{V}, \ x_{jk}, y_{ij} \in \{0,1\} \ \forall i \neq j \in V, \ k \in K && \text{(2.2j)}
\end{aligned}
$$

The objective function (2.2a) maximizes the total profit in all the knapsacks. The terms of the sum, $p_{ij} y_{ij} x_{ik} x_{jk}$, mean that we are only adding a profit when the items i and j are assigned, in this order, to the same bin, i.e, we visit node i and then node j in the same tour k. The first two constraints, (2.2b) and (2.2c), correspond to the ones at MKP formulation, with slight changes: for example, we impose that every item which is not the depot must be assigned to some bin. Also, the capacities between knapsacks remain constant and equal to W. The capacity constraints are formulated using the same idea as objective function, we are only adding a given weight w_{ij} when item j is assigned to bin k preceded by item i, which is also assigned to bin k. Constraint (2.2d) ensures that the number of items assigned to a given bin k matches the sum of item pairs (i, j) that are assigned to bin k being j preceded by i. Constraint (2.2e) ensures that if j is preceded by i, then they are assigned to the same bin: $x_{ik}, x_{jk} \in \{0, 1\}$ so $\max_{k \in K} x_{ik}, x_{jk} \in \{1, 2\}$ (it can be zero because we have to assign i and j to some bin). This maximum will be 2 if and only if exists $k \in K$ with $x_{ik} = x_{jk} = 1$. So, if $y_{ij} = 1$ (i is assigned before j) the inequality implies that i, j are assigned to the same bin. Constraints at (2.2f) mean that the depot node is assigned to all the m initialized bins and also that it goes before and goes after other items the same number of times. This means that we start and end tours at this node. Constraints at (2.2g) force all the items $j \in \overline{V}$ to go before and after another item, that is, all the nodes are reached exactly once in a tour. Constraints (2.2h) are useful to ensure the continuity of the route and to eliminate tours that doesn't include the depot node. The continuous variables $L_j,\ j \in \overline{V}$ show the drone battery consumption after visiting node j. We use the constraints at (2.2i) to define lower and upper bounds for L_j. The load or accumulated battery consumption L_j must be greater or equal than the weight w_{ij} only if i goes before j in the same bin. On the other hand, the load is not allowed to surpass the battery capacity W. Finally, constraints at (2.2j) define the domain of load and decision variables and indexes.

2.2.2 Solver for GMKP

In order to obtain exact solutions to our CSP problem instances via the GMKP formulation we will need some sort of solver. In this work we have decided to use *Gurobi*. The Gurobi Optimizer is a commercial optimization solver for linear programming (LP), quadratic programming (QP), quadratically constrained programming (QCP), mixed integer linear programming (MILP), mixed-integer quadratic programming (MIQP), and mixed-integer quadratically constrained programming (MIQCP). The Gurobi Optimizer supports a variety of programming and modeling languages including: *C++, Java, MATLAB, R and Python*, and also links to standard modeling languages as *AMPL*. The selected programming language in this work was Python, so this solver fits enough our needs. Also, we acquired an academic license in order to access a bigger computation power. The Gurobi Optimizer also includes a number of features to support the building of optimization models including support for:

- A powerful Python modeling API[3] that integrates with vector and matrix objects from NumPy and SciPy, two of the main libraries in Python.
- Multiple objective functions with flexibility in how they are prioritized.
- General constraints as MIN, MAX, ABS, AND, OR, and indicator constraints help avoid having to turn commonly occurring constraints into linear constraints.
- Client-server computing, to offload optimization or tuning runs to a compute server.
- Cloud computing, to perform optimization or tuning jobs in the cloud.
- Distributed computation, to harness the power of multiple machines working together on a single optimization or tuning job.

The use of Gurobi in Python is very simple, as we can see in the next example. Suppose we want to optimize the next model:

$$\max \quad x + y + 2z \tag{2.3a}$$
$$\text{s.t.:} \quad x + 2y + 3z \leq 4 \tag{2.3b}$$
$$x + y \geq 1 \tag{2.3c}$$
$$x, y, z \in \{0, 1\} \tag{2.3d}$$

We can program this model with library Gurobipy in Python as follows:

```python
import gurobipy as gp
from gurobipy import GRB
# Create a new model
m = gp.Model("mip1")
# Create variables
x = m.addVar(vtype=GRB.BINARY, name="x")
y = m.addVar(vtype=GRB.BINARY, name="y")
z = m.addVar(vtype=GRB.BINARY, name="z")
# Set objective
m.setObjective(x + y + 2 * z, GRB.MAXIMIZE)
# Add constraint: x + 2 y + 3 z <= 4
m.addConstr(x + 2 * y + 3 * z <= 4, "c0")
# Add constraint: x + y >= 1
m.addConstr(x + y >= 1, "c1")
# Optimize model
m.optimize()
for v in m.getVars():
```

[3]Application Programming Interface is a software intermediary that allows two applications to talk to each other

```
18     print('%s %g' % (v.varName, v.x))
19 print('Obj: %g' % m.objVal)
```

The two first lines are imports: we import *gurobipy* library to have access to its functions and classes through the prefix *gp* and also import class *GRB* which contains useful constants we use to define the variables domain. The next line is for creating the model. Lines 6 to 8 define variables x, y, z as binary ones. Line 10 set the objective function of the model. After that we add the constraints of our model. Line 16 starts the process of optimization of the model. The last lines print the optimal solution and value. This was a simple model that also had a simple code implementation. However, Gurobi is flexible enough to allow us implement the GMKP formulation, although that implementation won't be as simple.

2.3 Bin Packing Problem approach

Now we put an instance of the CSP problem as an instance of another well-known problem in the literature, the Bin Packing Problem (BPP). It is similar to the Multiple Knapsack Problem but, in this case, we would like to minimize the number of bins rather than maximize the total profit among them. The BPP is NP-hard in the strong sense [11]. The BPP can be described, using the terminology of knapsack problems, as follows. Given n items and n knapsacks (or bins), with

$$w_j = \text{ weight of item } j,$$

$$c = \text{ capacity of each bin},$$

assign each item to one bin so that the total weight of the items in each bin does not exceed c and the number of bins used is a minimum. One possible mathematical formulation of the problem is:

$$(BPP) \quad \min \quad z = \sum_{i=1}^{n} y_i \tag{2.4a}$$

$$\text{s.t.:} \quad \sum_{j=1}^{n} w_j x_{ij} \leq c y_i, \quad \forall i \in N = \{1, \dots, n\} \tag{2.4b}$$

$$\sum_{i=1}^{n} x_{ij} = 1, \quad \forall j \in N \tag{2.4c}$$

$$x_{ij} \in \{0, 1\} \ni y_i, \quad i, j \in N \tag{2.4d}$$

where

$$y_i = \begin{cases} 1 & \text{if bin } i \text{ is used} \\ 0 & \text{otherwise} \end{cases}$$

$$x_{ij} = \begin{cases} 1 & \text{if item } j \text{ is assigned to bin } i \\ 0 & \text{otherwise} \end{cases}$$

A formulation that generalizes BPP one could give us the minimum number of tours (or returns to the battery station) that the drone can do in a given inspection, m^*. We denote this formulation by GBPP. If we combine GBPP with the GMKP formulation, we could find the best m^* tours to perform the inspection, in the sense of minimizing the total time cost. However, we must highlight that finding the minimum number of tours needed to properly[4] cover the inspection using GBPP and then solve the GMKP may lead us to a solution worse than if we use more tours, as it is shown in the Figure 2.4. If we consider the instance which is described in the figure, is obvious that we have enough battery to visit all the nodes and then return to base. If we do this, i.e, if we use one tour, the minimum total time cost we can achieve is 8, because to cover both sides we need to spend 3 units of time costs, to travel to the other side of the graph costs 4 units and return to base costs 1. Now, consider we use two tours, one for each side of the graph, that is, for example, $\{0, 1, 2\}$ and $\{0, 3, 4\}$, then the total profit is $3 + 3 = 6$, which is less than using one tour. To overcome this, we can solve multiple GMKP instances for $m = m^*, m^* + 1, \ldots$ and observe the behaviour of the obtained solutions. Nevertheless, solutions for GMKP using fewer number of tours usually give the optimal in total time cost.

In addition, in order to use the BPP for solving the CSP problem, we must adapt this formulation to one which accounts for the restrictions of our situation. The main differences are related to weights dependence on previous assigned items. Another difference, as we have commented in GMKP, is that we have a depot node acting as a required item to assign to each bin, the 0-item. So, let us now consider the *Generalized Bin Packing Problem (GBPP)*, based on BPP and GMKP, where k varies in $K = \{1, \ldots, n\}$ and we are supposing that we have enough capacity to, at least, visit any of the nodes (adding it to some bin) and returning to base station (adding the 0-item to the same bin), that is, $W \geq w_{0j}x_{0jk} + w_{j0}x_{j0k},\ \forall j, k \in \{1, \ldots, n\}$[5]. Also,

[4]Not violating the problem constraints.

[5]This supposition will be done in all the instances because if they do not verify this, they are infeasible instances.

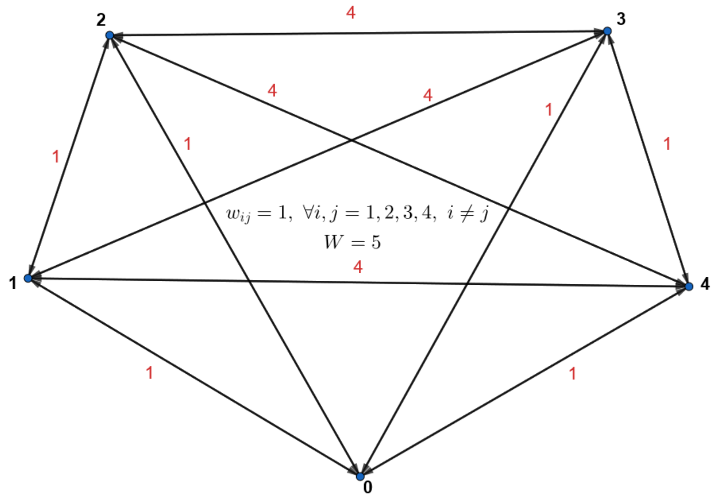

Figure 2.4: Counterexample: There are instances for which using a non-optimal number of tours can lead to less total time cost. Time costs are the red values above edges and they are the same in both directions

the let w_{ij} be the same weights as in GMKP. In addition, let:

$$y_k = \begin{cases} 1 & \text{if bin } k \in K \text{ is used} \\ 0 & \text{otherwise} \end{cases}$$

$$x_{jk} = \begin{cases} 1 & \text{if item } j \in V \text{ is assigned to knapsack } k \in K \\ 0 & \text{otherwise} \end{cases}$$

$$a_{ij} = \begin{cases} 1 & \text{if item } i \in V \text{ goes before item } j \in V \\ 0 & \text{otherwise} \end{cases}$$

The formulation would be as follows:

$$
\begin{aligned}
(GBPP) \quad \min \quad & z = \sum_{k=1}^{n} y_k && (2.5a)\\
\text{s.t.:} \quad & \sum_{i \neq j \in V} w_{ij} a_{ij} x_{ik} x_{jk} \leq W y_k, \quad \forall k \in K && (2.5b)\\
& \sum_{k \in K} x_{jk} = 1, \quad \forall j \in \overline{V} && (2.5c)\\
& \sum_{j \in V} x_{jk} = \sum_{i \neq j \in V} a_{ij} x_{ik} x_{jk}, \quad \forall k \in K && (2.5d)\\
& 2a_{ij} \leq \max_{k \in K} x_{ik} + x_{jk}, \quad \forall i \neq j \in V && (2.5e)\\
& \sum_{k \in K} x_{0k} = z, \sum_{i \in \overline{V}} a_{i0} = z, \sum_{j \in \overline{V}} a_{0j} = z && (2.5f)\\
& \sum_{i \in V} a_{ij} = 1, \quad \forall j \in \overline{V}, \quad \sum_{j \in V} a_{ij} = 1, \quad \forall i \in \overline{V} && (2.5g)\\
& L_i + w_{ij} a_{ij} - W(1 - a_{ij}) \leq L_j && (2.5h)\\
& a_{ij} w_{ij} \leq L_j \leq W, \quad \forall i \in V, \forall j \in \overline{V}, \ i \neq j, && (2.5i)\\
& L_j \in \mathbb{R}, \ \forall j \in \overline{V}, \ y_k, x_{jk}, a_{ij} \in \{0,1\} \ \forall i \neq j \in V, k \in K && (2.5j)
\end{aligned}
$$

This formulation has also several constraints. These are very similar to the ones in GMKP formulation, with slight changes: in the capacity constraints (2.5b), we have $W y_k$ in the right side, which means that we must initialize bin k if we want to assign items to it. Also, in the depot node constraints (2.5f) we use $z = \sum_{k=1}^{n} y_k$ instead of m. It is because it does not make sense to define a constant number of initialized bins: The depot node appears in tours as many times as tours are initialized.

As in GMKP, we can implement this GBPP formulation in Gurobi Optimizer and obtain exact solutions to our CSP instances, but for the problem of finding the smallest number of feasible tours to cover the inspection graph. Then we can use this solution as seed for initializing the GMKP iterations.

2.3.1 Some algorithms for BPP

Now we mention some heuristic algorithms for solving BPP that can be useful to find good approximations for large instances. We use the work ([11], Chapter

8). The simplest approximate approach to the bin packing problem is the Next-Fit (NF) algorithm: the first item is assigned to bin 1. Items 2 to n are then considered by increasing indices: each item is assigned to the current bin, if it fits, otherwise a new bin is initialized, which becomes the new current one. It can be proven that $NF(I) \leq 2z(I)$, being $NF(I)$ the solution value provided by the NF algorithm for an instance I and $z(I)$ the optimal value. The current solution adopted by the company at the CSP inspections is a sort of Next-Fit procedure. A basic Python implementation of Next Fit algorithm follows ([15]):

```python
def nextfit(weight, c):
    res = 0 #Number of initialized bins
    rem = c #Remaining capacity in current bin
    for _ in range(len(weight)): #Loop in each item
        if rem >= weight[_]: #Checking if can assign next item to current bin
            rem = rem - weight[_]
        else: #If cannot...
            res += 1 #Initialize new bin
            rem = c - weight[_] #Update remaining capacity
    return res
```

Evidently, we would have to adapt this a bit more to record the items which are assigned to each bin, to have a full solution. But essentially, this is its Python implementation.

A better algorithm, the First-Fit (FF), considers the items according to increasing indices and assigns each item to the lowest indexed initialized bin into which it fits. Only when the current item cannot fit into any initialized bin, is a new bin introduced. This algorithm has also related bounds: $FF(I) \leq 1.7z(I) + 2$. Its Python implementation, also borrowed from [15], is:

```python
def firstFit(weight, n, c):
    res = 0 # Initialize result (Count of bins)
    # Create an array to store remaining space in bins
    # there can be at most n bins
    bin_rem = [0]*n
    # Place items one by one
    for i in range(n):
        # Find the first bin that can accommodate
        # weight[i]
        j = 0
        while( j < res):
            if (bin_rem[j] >= weight[i]):
                bin_rem[j] = bin_rem[j] - weight[i]
                break
            j+=1
        # If no bin could accommodate weight[i]
        if (j == res):
            bin_rem[res] = c - weight[i]
            res= res+1
    return res
```

The next algorithm, Best-Fit (BF), is obtained from FF by assigning the current item to the feasible bin (if any) having the smallest residual capacity. Also, there is another known algorithm in the literature which is the Worst-Fit (WF). One could say that it is the inverse of BF: it assigns items to the least tightest bin. If the item does not fit in any bin, then it will be assigned to a new one. Their Python implementations are[6]:

```
def bestFit(weight, n, c):
    res = 0;# Initialize result (Count of bins)
    # Create an array to store
    # remaining space in bins
    # there can be at most n bins
    bin_rem = [0]*n;
    # Place items one by one
    for i in range(n):
        # Find the first bin that
        # can accommodate
        # weight[i]
        j = 0;
        # Initialize minimum space
        # left and index
        # of best bin
        mini = c + 1;
        bi = 0;
        for j in range(res):
            c1 = bin_rem[j] >= weight[i]
            c2 = bin_rem[j] - weight[i] < mini
            if (c1 and c2):
                bi = j;
                mini = bin_rem[j] - weight[i];
        # If no bin could accommodate weight[i],
        # create a new bin
        if (mini == c + 1):
            bin_rem[res] = c - weight[i];
```

[6]The Python implementation of Worst-Fit have been easily adapted from C++ implementation at the reference.

```
                res += 1;
            else: # Assign the item to best bin
                bin_rem[bi] -= weight[i];
    return res;
```

```
def worstFit(weight, n, c):
    res = 0;# Initialize result (Count of bins)
    # Create an array to store
    # remaining space in bins
    # there can be at most n bins
    bin_rem = [0]*n;
    # Place items one by one
    for i in range(n):
        # Find the first bin that
        # can accommodate
        # weight[i]
        j = 0;
        # Initialize maximum space
        # left and index
        # of worst bin
        mx = -1;
        wi = 0;
        for j in range(res):
            c1 = bin_rem[j] >= weight[i]
            c2 = bin_rem[j] - weight[i] > mx
            if (c1 and c2):
                wi = j;
                mx = bin_rem[j] - weight[i];
        # If no bin could accommodate weight[i],
        # create a new bin
        if (mx == -1):
            bin_rem[res] = c - weight[i];
            res += 1;
        else: # Assign the item to best bin
            bin_rem[wi] -= weight[i];
    return res;
```

Assume now that the items are sorted so that $w_1 \geq w_2 \geq \cdots \geq w_n$, and then NF, FF, BF or WF is applied. The resulting algorithms are called Next-Fit Decreasing (NFD), First-Fit Decreasing (FFD), Best-Fit Decreasing (BFD) and Worst-Fit Decreasing (WFD). It has been proven (Johnson, Demers, Ullman, Garey and Graham, 1974) that $FFD(I) \leq \frac{11}{9}z(I) + 4$. However, during the resolution of our CSP realistic instances, we have found that using these techniques leads to worse solutions than if we apply an opposite procedure for ordering: instead of decreasing ordering we will consider an increasing ordering of weights, that is, select the next node which is the closest. The adaptation of this four basic bin packing heuristics to solve our CSP problem is done easily. First, we must rearrange the weights w_{ij} in a matrix where row i indicates the energy costs of going from node i to node j (indicated by column) and inspecting it. Each bin will be initialized with 0-item inside. When we introduce a new item into the bin, we look at the row associated with the last item assigned to the bin. We must save the last added item of each bin in order to know which row look at in each iteration. Also, we can sort the values in the row by an ordering strategy in each iteration, omitting assigned items.

2.4 Exact Cover Problem approach

The Exact Cover Problem (ECP) reminds us to children puzzles or maybe the times we spent hours solving a Sudoku, problems where we have to construct a matching of pieces or numbers in a certain way. The statement is as follows: given a collection S of subsets of a set X, an *exact cover* is a subcollection S^* of S such that each element in X is contained in *exactly one* subset in S^*. The decision problem of decide whether such S^* exists is known as the ECP. It is also NP-Complete (in fact, it is one of *Karp's 21 NP-complete problems*), like the CVRP, MKP and BPP. This problem can be represented as a binary matrix A where we would like to find a subset of rows such that if we sum them by components we obtain a row full of ones (that is, each column in the subset of rows has exactly one non-zero value). Each column is an element of X and each row is a subset of S.

As we have commented, Sudoku problem can be formulated with ECP. Another example of Exact Cover Problem is the *Pentomino*, which is a puzzle where pieces are made of five unit squares connected edge-to-edge and the aim is to form a rectangular or squared shape using the given pieces (the *Tetris* game is based on this). Figure 2.5 shows a pentomino puzzle. Obviously, there are also tetramino's puzzles and other extensions. In addition, the *n queens problem* is a slightly generalized exact cover problem. The n queens problem is a generalization of the eight queens puzzle of placing 8 non-attacking queens on an 8×8 chessboard. For this generalization there are solutions for all natural numbers n with the exception of $n = 2, 3$.

Now, we are going to explain further ECP through the next detailed example: let $S = A, B, C, D, E, F$ be a collection of subsets of a set $X = \{1, 2, 3, 4, 5, 6, 7\}$ such

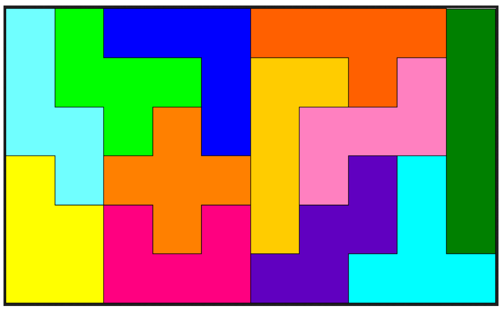
Figure 2.5: A 6x10 pentomino puzzle example

that:

$A = \{1, 4, 7\}$, $B = \{1, 4\}$, $C = \{4, 5, 7\}$, $D = \{3, 5, 6\}$, $E = \{2, 3, 6, 7\}$, $F = \{2, 7\}$.

Then the subcollection $S^* = \{B, D, F\}$ is an exact cover of X using S, since each element in X is contained in exactly one of the subsets. Moreover, the given S^* is the only exact cover, as the following argument demonstrates: because A and B are the only subsets containing 1, an exact cover must contain A or B, but not both. If an exact cover contains A, then it doesn't contain B, C, E or F, as each of these subsets overlap with A. Then D is the only remaining subset, but the collection $\{A, D\}$ doesn't cover the element 2. So, there is no exact cover containing A. On the other hand, if an exact cover contains B, then it doesn't contain A or C, as each of these subsets has an element on common with B. Because D is the only remaining subset containing 5, D must be part of the exact cover. If an exact cover contains D, then it doesn't contain E, as E overlaps D. Then F is the only remaining subset, and the collection $\{B, D, F\}$ is indeed an exact cover. Clearly, the hardness of this problem grows when there are a lot of subsets in S and X is big.

Now, we want to explain how we can relate the Exact Cover Problem with the CSP one. Suppose we have a directed graph $G = (V, E)$ where V is the set of nodes where inspections take place and E are the possible paths between two of the nodes. What we are looking for is a collection of cycles, C, such that it covers the entire set V and we don't repeat a node v in two of the cycles, that is, a ECP where pieces are cycles. Obviously, there might be more than one solution. Since we have a drone battery restriction, many of the cycles that may appear in some solutions could be infeasible, so we will remove them from potential solutions to the CSP problem. Finally, when we have found the feasible solutions, we will keep that with the minimum time consumption or size, in the sense of less tours performed.

In order to solve CSP problem by ECP, we must associate weights to cycles: a cycle will be feasible if it correspond to a feasible tour, that is, the total weight of the cycle doesn't exceed a given battery consumption bound, W. Also, we can associate profits or costs to each cycle and maximize or minimize them, respectively. Note that each cycle will start and end in the depot node (because the drone has to recharge its battery), so S is the collection of all the cycles (subsets of V) which visit the depot node, and whose total weight is less or equal than W. Actually, we are not interested in all the cycles, but in the elementary ones (see [12]), which are those where no vertex appears twice (except the first one which matches the last one), and also two elementary cycles are different if they are not cyclic permutations of each other. There are exactly:

$$\sum_{i=1}^{n-1} \binom{n}{n-i+1}(n-i)!$$

elementary cycles in a complete digraph with n vertices.

Computing the total weight of a cycle can be done as follows: let $c = \{u, v, v', u\}$ the visited nodes in a given one (u is the depot node), then we will compute $w(c) = w(u,v)+w(v,v')+w(v',u)$, where $w(i,j)$ are computed like in the previous sections, given a pair of nodes i, j.

The ECP has been studied for a long time as it is derived from the famous *Set Cover Problem*, but nowadays there are few efficient algorithms for solving it. In our research we have found two main references that will help us with the task of solving CSP problem instances. First of all, we need to obtain the collection S, and this is not current because, at the beginning, the feasible cycles in G are unknown, although we know that there will be a large number of them if there are many vertices and edges. For this first task we will use the *Johnson's Algorithm for finding all elementary cycles in a given digraph*, deeply explained in [12]. We must keep only the elementary cycles that visit the depot node.

2.4.1 The Johnson's algorithm

The Johnson's algorithm finds all the elementary circuits of a digraph in time bounded by $O((n+e)(c+1))$ and space bounded by $O(n+e)$, where there are n vertices, e edges and c elementary circuits in the graph. The algorithm is faster than previous ones because it considers each edge at most twice between any one circuit and the next in the output sequence. We can also consider elementary paths, which are paths where no vertex appears twice.

In this document, we will be using a nonrecursive, iterator/generator version of Johnson's algorithm which is implemented in the Python library *networkx*. In fact, we have found that for regular instances of CSP problem, where $N \approx 40$, it is very difficult to found all elementary circuits with arbitrary length. To overcome this disadvantage, we use an adapted version of networkx's implementation which only consider paths whose length is dominated by a given limit L. Also, we have modified it to only search for paths starting at 0 node, which are actually the ones we want to obtain. Another possible relaxation we can assume is to only use edges between neighboring nodes, instead of considering a complete digraph.

This algorithm allows us to obtain all the elementary circuits with maximal length $L \leq 13$ of particular graphs with $N \approx 40$ nodes and $|E|$ edges (those related to neighboring nodes in two faced batches associated graphs) relatively quickly. Since the drone usually inspects 9 to 10 interests points before returning for charging, it seems like this adapted Johnson's algorithm is good enough for our purposes.

2.4.2 The Knuth's Algorithm X

To solve ECP in Python, we have used some libraries which implement the dancing links technique of Donald Knuth for his Algorithm X, which solves ECP instances. In the following we will briefly explain how the algorithm X works and also detail the dancing links technique which leads to DLX algorithm and give a small example in order to understand better how it operates.

Once he have obtained all elementary circuits S in graph G we can transform the instance of the CSP problem in one for Exact Cover Problem. S is a collection of subsets of $V = \{0, 1, \ldots, N\}$ all of them starting at 0 (or at least containing it, we can reorder them). To reduce the computational costs of solving the ECP, we also can discard every cycle in S such that its total associated weight exceeds W, that is, infeasible cycles, obtaining S^*. We will solve the ECP instance (S^*, V) using a Python implementation of the Donald Knuth's X algorithm, described in [13]. We probably will obtain several possible exact covers C for our instance, due to typical big size of S^*. In order to solve the CSP problems 1 and 2 associated with the instance, we'll select the best cover solution C^* in terms of total inspection time cost and number of cycles used, respectively.

The Knuth's Algorithm X is an algorithm that finds all solutions to an exact cover problem. DLX is the name given to Algorithm X when it is implemented efficiently using Donald Knuth's Dancing Links technique on a computer. It is a straightforward recursive, nondeterministic, depth-first[7], backtracking algorithm used by Donald Knuth to demonstrate an efficient implementation called DLX, which uses the dancing links technique. Dancing links is a technique for reverting the operation of deleting a node from a circular doubly linked list[8]. The idea of DLX is based on the observation that in circular doubly linked list of nodes, if we let $L[x]$ and $R[x]$ point to the predecessor and successor of that element,

$$L[R[x]] \leftarrow L[x], \quad R[L[x]] \leftarrow R[x]$$

remove x from the list, while

$$L[R[x]] \leftarrow x, \quad R[L[x]] \leftarrow x$$

will put x back into the list again, assuming that $L[x]$ and $R[x]$ haven't been modified. Knuth observed that a naive implementation of his Algorithm X would spend an inordinate amount of time searching for 1's. To improve this search time from complexity $O(n)$ to $O(1)$, Knuth implemented a sparse matrix where only 1's are stored. At all times, each node in the matrix will point to the adjacent nodes to the left and right (1's in the same row), above and below (1's in the same column), and the header for its column (a special node each column will have, they form the control row consisting of all the columns which still exist in the matrix[9]). Each row and column in the matrix will consist of a circular doubly-linked list of nodes.

The reciprocal procedures for deleting and re-adding an element x to a doubly linked list are extremely useful when updates to a data structure are not indented to be permanent. For example, this occurs in backtrack programs, which enumerate all solutions to a given set of constraints. Backtracking is also called depth-first search. These programs can require to maintain a very large stack and might take too much time, if we use a naive implementation. The Dancing Links technique improve the performance in this algorithms. For example, the exact cover problem, which we can represent by a matrix A consisting of 0's and 1's such that we want to select a subset of the rows where the digit 1 appears in each column exactly once, is a natural candidate

[7]Algorithm which searches in tree or graph data structures by starting at a root node and exploring as far as possible along each branch before backtracking.

[8]A linked data structure that consists of a set of sequentially linked records called nodes. Each node contains three fields: references to the previous and to the next node in the sequence and one data field.

[9]Also, each column header may optionally track the number of nodes in its column, so that locating a column with the lowest number of nodes is of complexity $O(n)$ rather than $O(mn)$. Selecting a column with a low node count is a heuristic which improves performance in some cases.

for backtracking. The algorithm X, that we will use for solving it, is detailed in the following:

Algorithm 1: Algorithm X

```
input  : $A$
output: Solutions to exact cover problem represented by $A$
1. If $A$ is empty, the problem is solved; terminate successfully.
2. Otherwise, choose a column, c (deterministically, we can select it by the
   lowest node count, for example).
3. Chose a row, r, such that $A[r, c] = 1$ (non-deterministically).
4. Include $r$ in the partial solution.
5. For each $j$ such that $A[r, j] = 1$,
    6. delete column $j$ from matrix $A$;
    7. for each $i$ such that $A[i, j] = 1$,
        8. delete row $i$ from matrix $A$.
9. Repeat this algorithm recursively on the reduced matrix $A$.
```

The non-deterministic choice of r means that the algorithm generates independent subalgorithms. Each subalgorithm inherits the current matrix A, but reduces it with respect to a different row r. If a column c is entirely zero, there are no subalgorithms and the process terminates unsuccessfully. The subalgorithms form a search tree in a natural way, with the original problem at the root and with level k containing each subalgorithm that corresponds to k chosen rows. Any systematic rule for choosing column c in this procedure will find all solutions, but some rules work much better than others. Golomb and Baumert suggested choosing, at each stage of a backtrack procedure, a subproblem that leads to the fewest branches, whenever this can be done efficiently. In the case of an exact cover problem, this means that we want to choose at each stage a column with fewest 1's in the current matrix A. We will see that dancing links allow us to do this quite nicely.

A way to implement algorithm X is to represent each 1 in the matrix A as a data object x with five fields $L[x]$, $R[x]$, $U[x]$, $D[x]$, $C[x]$. Rows of the matrix are doubly linked as circular lists via the L and R fields ("left" and "right") and columns are doubly linked as circular lists via the U and D fields ("up" and "down"). Each column list also includes a special data object called its *list header.* The list headers are part of a larger object called a *column object.* Each column object y contains the fields $L[y]$, $R[y]$, $U[y]$, $D[y]$ and $C[y]$ of a data object and two additional fields, $S[y]$ ("size") and $N[y]$ ("name"). The size is the number of 1's in the column, and the name is a symbolic identifier for printing the answers. The C field of each object points to the column object at the head of the relevant column. L and R fields of the list headers link together all columns that still need to be covered. This circular list also includes a special column object called the root, h, which serves as a master header for all the active headers. Fields $U[h]$, $D[h]$, $C[h]$, $S[h]$ and $N[h]$ aren't used. To deepen this with more detail, consider the following 0-1 matrix:

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

It represents an exact cover problem with an exact cover formed by rows $\{1, 4, 5\}$. This matrix would be represented by the objects shown in Figure 2.6, if we name the columns A, B, C, D, E, F and G. The diagram wraps around toroidally at the top, bottom, left, and right. The C links are not shown because they would mess up the pic. Every C field points to the topmost element in its column.

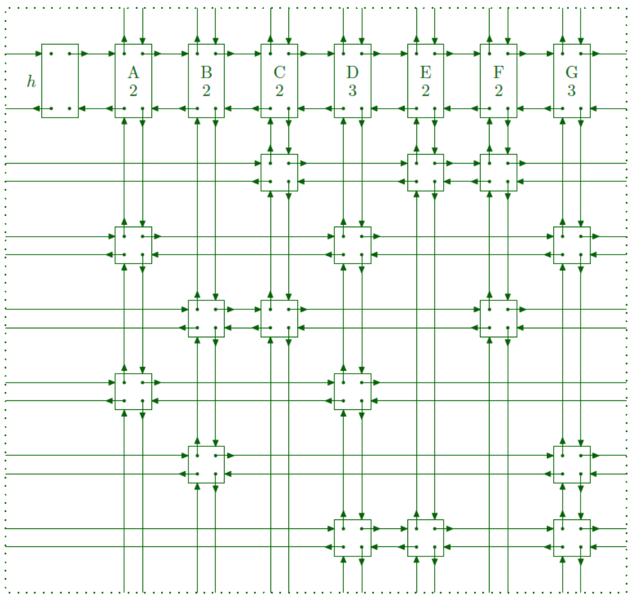

Figure 2.6: Four-way-linked representation of the exact cover problem (borrowed from [13])

The non-deterministic algorithm to find all exact covers can now be cast in the following explicit, deterministic form as a recursive procedure $search(k)$, which is invoked initially with $k = 0$. But, before to see the pseudo-code of the procedure, let's define some concepts:

- Operation of printing the current solution. We successively print the rows containing $O_0, O_1, \ldots, O_{k-1}$, where the row containing data object O is printed by printing $N[C[O]]$, $N[C[R[O]]]$, $N[C[R[R[O]]]]$, etc.
- Operation of choose a column object c. We could simply set $c \leftarrow R[h]$ (leftmost uncovered column). Or if we want to minimize the branching factor, we could select the column with the smallest number of 1's (we can obtain it by using S fields and iterating j in $R[h]$, $R[R[H]]$,..., while $j \neq h$).
- Operation of covering column c. Removes c from the header list and removes all rows in c own list from the other column lists they are in:

Algorithm 2: $COVER(c)$

```
1. Set $L[R[c]] \leftarrow L[c]$ and $R[L[c]] \leftarrow R[c]$.
2. **For** each $i \leftarrow D[c], D[D[c]], \ldots,$ while $i \neq c$,
    3. **for** each $j \leftarrow R[i], R[R[i]], \ldots,$ while $j \neq i$,
        4. set $U[D[j]] \leftarrow U[j]$ and $D[U[j]] \leftarrow D[j]$,
        5. set $S[C[j]] \leftarrow S[C[j]] - 1$.
```

- Operation of uncovering column c. Notice that uncovering takes place in precisely the reverse order of covering operation, using the fact that re-adding an element undoes deleting it from the list. We must be careful to unremove the rows from bottom to top, because we removed them from top to bottom. Similarly, it is important to uncover the columns of row r from right to left, because we covered them from left to right.

Algorithm 3: $UNCOVER(c)$

```
1. **For** each $i \leftarrow U[c], U[U[c]], \ldots,$ while $i \neq c$,
    2. **for** each $j \leftarrow L[i], L[L[i]], \ldots,$ while $j \neq i$,
        3. set $S[C[j]] \leftarrow S[C[j]] + 1$.
        4. set $U[D[j]] \leftarrow j$ and $D[U[j]] \leftarrow j$,
5. Set $L[R[c]] \leftarrow c$ and $R[L[c]] \leftarrow c$.
```

We can note this operations as PRINT, CHOOSE, COVER and UNCOVER. Once we have defined them, we can state the $search(k)$ algorithm:

Algorithm 4: $search(k)$

```
1. If $R[h] = h$, **PRINT** the current solution.
2. Otherwise, **CHOOSE** a column object $c$.
3. **COVER** column $c$.
4. **For** each $r \leftarrow D[c], D[D[c]], \ldots,$ while $r \neq c$,
    5. set $O_k \leftarrow r$;
    6. **for** each $j \leftarrow R[r], R[R[r]], \ldots,$ while $j \neq r$,
        7. **COVER** column $j$;
    8. $search(k+1)$;
    9. **for** each $j \leftarrow L[r], L[L[r]], \ldots,$ while $j \neq r$,
        10. **UNCOVER** column $j$.
11. **UNCOVER** column $c$ and return.
```

The reader may observe that this pseudo-code is pretty similar to Algorithm X's one, although it explore all the possible reductions of matrix respect to rows r in a recursive and more efficient way. To understand the printed solutions, we will explain another possible representation for matrix A. Recall that we named its seven columns A to G, so what if we just name rows by the sequence of letters associated with their non-zero columns? If we do this we can get:

$$A = \begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix} \Longleftrightarrow \begin{pmatrix} CEF \\ ADG \\ BCF \\ AD \\ BG \\ DEG \end{pmatrix}.$$

Consider, for example, what happens when $search(0)$ is applied to the data of the matrix A which we defined before, as it is represented by the four-way-linked objects in Figure 2.6. Firstly, column A is covered by removing both of its rows from their other columns, the structure now takes the form of Figure 2.7. Continuing $search(0)$, when r points to the A element of row (A,D,G), we also cover columns D and G. Figure 2.8 shows the status as we enter $search(1)$, this data structure represents the reduced matrix

$$\begin{pmatrix} B & C & E & F \\ 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 \end{pmatrix} \Longleftrightarrow \begin{pmatrix} CEF \\ BCF \end{pmatrix}.$$

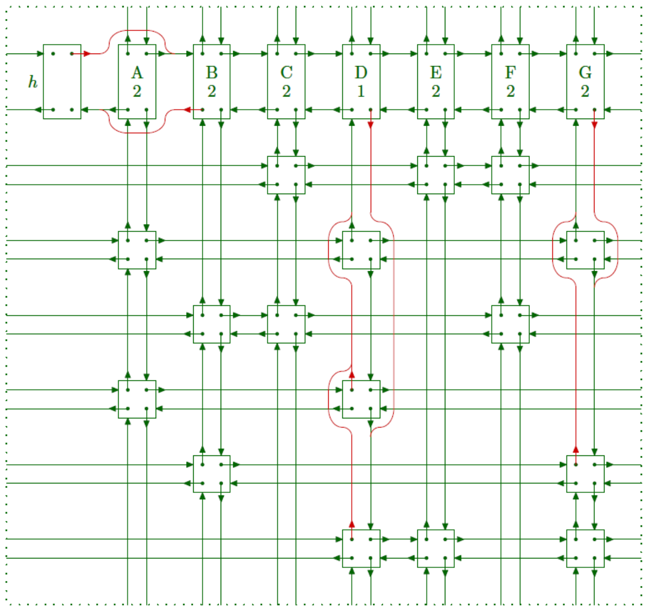

Figure 2.7: The links after column A in Figure 2.6 has been covered (borrowed from [13])

Now $search(1)$ will cover column B, and there will be no 1's left in column E. So $search(2)$ will find nothing. Then $search(1)$ will return, having found no solutions, and the state of Figure 2.8 will be restored. The outer level routine, $search(0)$, will proceed to convert Figure 2.8 back to Figure 2.7, and it will advance r to the A element of row (A,D). Now the reduced matrix would be:

$$\begin{pmatrix} B & C & E & F & G \\ 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \end{pmatrix} \Longleftrightarrow \begin{pmatrix} CEF \\ BCF \\ BG \end{pmatrix}.$$

After some few steps more, the solution will be found. It will be printed as

$$\begin{pmatrix} AD \\ BG \\ CEF \end{pmatrix}.$$

The solution obtained matches with the subset of rows $\{1, 4, 5\}$ that we gave before explaining the $search(k)$ procedure. The main problem we will face when using this approach is that even being the DLX algorithm such an efficient one, our CSP graphs have a lot of nodes and then the collection S is too large. To reduce at most the size of subcollection S, we will only consider some certain edges in G, those which are drawn in Figure 2.9. That is, the edges will be those between neighboring nodes, in the sense of being placed immediately next to another node (vertical, horizontal or diagonally). This doesn't apply to depot node: It will have arcs leaving to and arriving from every other node. Doing this we are relaxing the processing of the ECP approach. However, since $n \approx 40$ in the most faced batches pairs, we have find that usually the algorithm can't solve the ECP in a satisfactory amount of time, because there are many ($> 10^{10}$) different solution covers and it is computationally hard to sort them by its number of tours or total inspection time. Even $n \approx 20$ is a high size for solving a CSP instance using ECP, but we should be able to, at least, develop a divide and conquer simple algorithm to chop the faced batches pairs in computationally feasible instances for ECP method. Then we can merge the sub-solutions in order to approximate the global one.

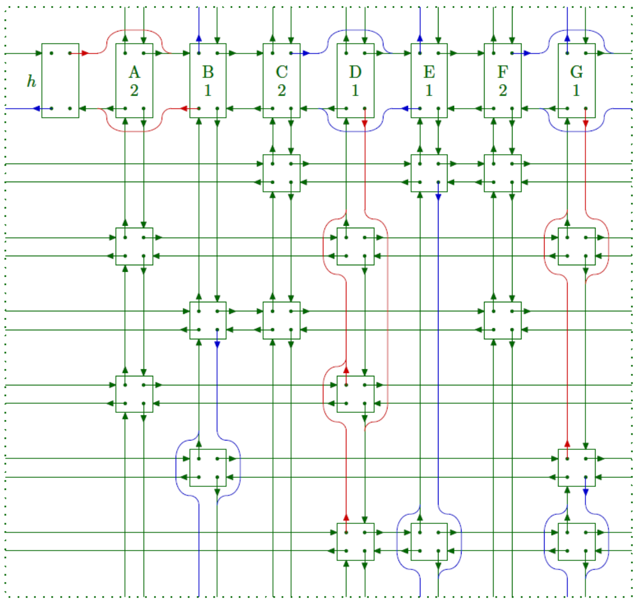

Figure 2.8: The links after columns D and G in Figure 2.7 have been covered (borrowed from [13])

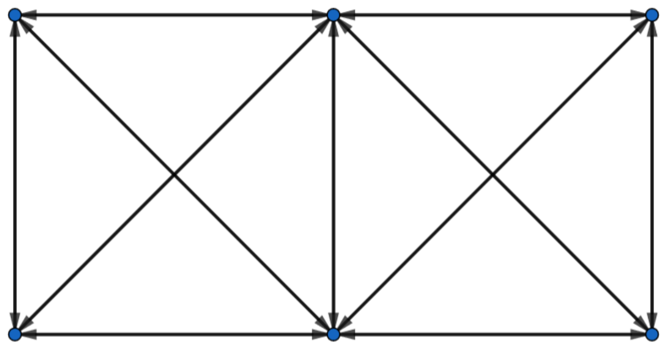

Figure 2.9: Neighboring nodes edges in graph when using the ECP approach (depot node and associated edges are omitted for clarity)

3 | Complexity and heuristics for the CSP problem

In the previous chapters we have related the CSP problem with several classical problems in the operational research field, many of them are related to situations that the mankind faces every day. We find typical examples in companies as Amazon, and also the post offices, delivery and logistic enterprises. Every day, they solve TSP generalizations, schedule and assignment problems, bin-packing problems, etc. In this chapter we present some heuristics based on approaches proposed for these classical problems to give sub-optimal solutions for the CSP problem. The mentioned problems have one thing in common, they are NP-complete, in a nutshell, they are very hard to solve for instances with a relative small size. We will proof in this chapter that the CSP problem 2 is also NP-Complete and we conjecture the same result for CSP problem 1.

The classical problems that we have presented in this document are mainly the following:

- Capacitated Vehicle Route Problem (CVRP): a natural generalization of classical *Travelling Salesman Problem (TSP)*. It models the problem of find a set of tours (the tours do not overlap except for the depot) that visits all the nodes in a given weighted graph while keeping the total accumulated demand of each tour less or equal than a given limit Q. This clearly remind us to the drone and its battery duration: our drone also has to do many tours while keeping its battery state under the limit. We add this problem here because it helped modelling CSP problem with concepts and ideas, although we have not implemented it to solve CSP instances.
- Multiple 0-1 Knapsack Problem (MKP): this generalizes one of the most known problems in operation research, the *0-1 Knapsack Problem (KP)*. Basically, the

MKP consists in how to fill a certain number of bins, each of them with a given capacity limit (which the sum of assigned items' weights can not surpass), while maximizing the associated profits between all the bins or total profit. We can identify bins with tours and items with nodes to visit and adapt the philosophy of MKP to the CSP problem.

- Bin Packing Problem (BPP): this problem is twinned with KP and MKP because in these problems we maximize the total profit, in the BPP we minimize the number of used bins (which remains fixed in MKP). The problem is to find an assignment of the items to bins such that we use the minimum needed number of bins while ensuring that the total weight limit in each of them is not surpassed. As well as for the MKP, we can identify bins with tours and items with nodes, while the weights are the battery consumption costs.
- Exact Cover Problem (ECP): this is culturally associated to the resolution of Sudokus, puzzles like pentominos, or the *n-queens problem*. Briefly, the problem consists in select a subset of pieces that, in union, complete a given universe set while non-overlapping between them. In this case, we can identify feasible cycles to be performed by the drone along the nodes and try to complete the inspection to all the nodes by selecting a subset from the feasible cycles set.

3.1 NP-completeness of CSP problem

We prove that the decision problem associated to the CSP problem 2 is NP-complete. We review here some concepts needed in the proof. A decision problem X is a (possible infinite) set of binary strings where an instance is a finite binary string s, of length $|s|$ and algorithm A solves X if $A(s) = YES \iff s \in X$. We say that algorithm A runs in polynomial time if for every instance s, it terminates in at most $p(s)$ steps, where p is some polynomial. [14] The class P consists of those (decision) problems that are solvable in polynomial time. Specifically, they are the problems that can be solved in time $O(n^k)$ for some constant k, where n is the size of the input to the problem.

Another important concept is certification algorithm: it checks whether a proposed solution is a YES instance. Algorithm C is an efficient certifier for X if C is a polynomial time algorithm that takes two inputs (s, t) and there exists a polynomial $p()$ so that for every s, $s \in X \iff$ there exists a string t such that $|t| \leq p(|s|)$ and $C(s,t) = YES$. For example, consider the problem if given integer s, is s composite? An efficient certifier for this problem is just the euclidean division algorithm: if

$s = 12$, $t = 4$ we divide s by t and look at the remainder, if it is 0 then s is a YES instance for this problem, else we have no conclusion.

The set NP is the set of all decision problems for which there exists an efficient certifier. It is well known that $P \subseteq NP$, because if $X \in P$ we can use any of the possible polynomial time algorithms for solving instance s in X as a current certifier for the algorithm. The reciprocal, $NP \subseteq P$, it has not been proved or refuted yet. However, for practical purposes, great part of the community accepts the conjecture of $NP = P$ as false. In addition, it can be defined the set EXP, which is the set of all decision problems solvable in exponential time on a deterministic Turing machine. It can be proven that $NP \subseteq EXP$: if $X \in NP$ there is a certifier C for X and then we can use it on all possible solutions t (performing a brute force procedure), and return YES if $C(s,t)$ returns YES for any of these. A relation between sets P, NP and EXP can be seen in Figure 3.1.

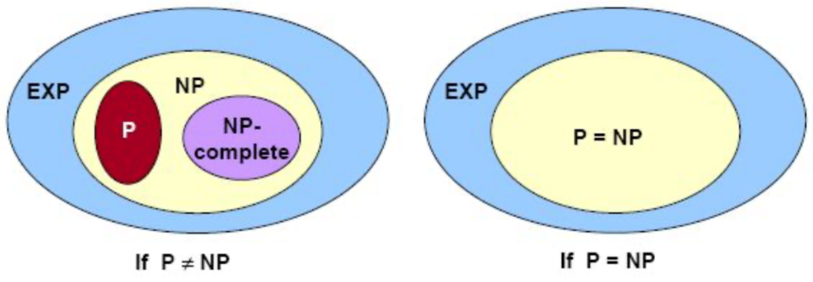

Figure 3.1: Diagram of the relation between sets P, NP, NP-complete and EXP

The notion of showing that one problem is no harder/easier than another is very useful in NP-completeness theory. We take advantage of this idea in almost every proof, as follows. Let us consider a decision problem A, which we would like to solve in polynomial time. We call the input to a particular problem an instance of that problem. For example, in BPP, an instance would be a particular tuple $(N, k, (w_j), c)$, where N is the number of items, k is a particular integer, (w_j) is the weights array and c is the bins capacity. Now suppose that we already know how to solve a different decision problem B in polynomial time. Finally, suppose that we have a procedure that transforms any instance α of A into some instance β of B with the following characteristics:

- The transformation takes polynomial time.

- The answers are the same. That is, the answer for α is "yes" if and only if the answer for β is also "yes".

We call such a procedure a polynomial-time reduction algorithm. It provides us a way to solve problem A in polynomial time:

1. Given an instance α of problem A, use a polynomial-time reduction algorithm to transform it to an instance β of problem B.
2. Run the polynomial-time decision algorithm for B on the instance β.
3. Use the answer for β as the answer for α.

There are two principal polynomial reductions:

- Problem X polynomial reduces (Cook-Turing) to problem Y, $X \leq_P Y$ if arbitrary instances of problem X can be solved using polynomial number of standard computational steps plus polynomial number of calls to "oracle"[1] that solves problem Y.
- Problem X polynomial transforms (Karp) to problem Y if given any input x to X, we can construct an input y such that x is a YES instance of X if and only if y is a YES instance of Y (we require $|y|$ to be of size polynomial in $|x|$). Polynomial transformation is polynomial reduction with just one call to oracle for Y, exactly at the end of the algorithm for X.

The technique of reduction relies on having a problem already known to be NP-complete, so we need a first NP-complete problem. The first natural problem which was shown to be NP-complete was the *boolean satisfiability problem* or B-SAT, which asks whether the variables of a given Boolean formula can be consistently replaced by the values TRUE or FALSE in such a way that the formula evaluates to TRUE.

A problem Y in NP (1) with the property that for every problem X in NP, X polynomial transforms to Y (2) is said to be a NP-complete problem. If a problem Y satisfies (2) but not necessarily (1) is called NP-hard. These are the "hardest computational problems" in NP. What the definition means is that if any NP-complete problem can be solved in polynomial time, then every problem in NP has a polynomial-time solution, that is, P = NP. Despite years of study, though, no polynomial-time algorithm has ever been discovered for any NP-complete problem, and they are called "intractable"

[1]Suppose that we have a hypothetical algorithm for solving problem Y in polynomial time

(these are unlikely to be solvable given limited computing resources). Most natural problems in NP are either in P or NP-complete. Packing problems (Set-Packing, Independent-Set), covering problems (Set-Cover, Vertex-Cover), sequencing problems (Hamiltonian-Cycle, TSP), partitioning problems (3-Color, Clique), constraint satisfaction problems (SAT, 3-SAT) and numerical problems (Subset-Sum, Partition, Knapsack) are six basic genres and paradigmatic examples of NP-complete problems. But, we would like to prove that problems are NP-complete without directly reducing every problem in NP to the given one. For this, we will use the next recipe to establish NP-completeness of a given problem Y:

1. Show that $Y \in NP$. In general this step consists in to *find an efficient certifier.*
2. Select a known NP-complete problem X.
3. Describe an algorithm that computes a function f mapping every instance x of X to an instance $f(x)$ of Y.
4. Prove that the function f satisfies x in X if and only if $f(x)$ in Y for all instance x.
5. Prove that the algorithm computing f runs in polynomial time.

In this work, we have proven that the CSP problem 2 (that is, the CSP problem in its tours minimization[2] version) is a NP-complete problem. Before the theorem, we have to reformulate the CSP problem 2 as a decision one:

CSP decision problem: let fix $N \in \mathbb{Z}_{\geq 1}, k \in \{1, \ldots, N\}$. Consider a CSP instance, i.e, a set of N inspection nodes, which form a weighted complete digraph $G = (V, E, (w_{ij}))$ with associated energy consumption weights $w_{ij}, i \neq j \in V = \{0, 1, \ldots, N\}$ as edge costs, and a total weight limit W. Is there a *covering set of feasible tours* for the given instance such that $|S| \leq k$?

Theorem 3.1. *CSP decision problem is NP-complete.*

Proof. First of all, we will define an efficient certifier for CSP decision problem. Consider an instance $I = (N, k, (w_{ij}), W)$ and suppose we have a solution candidate S, which is a set of cycles.

1. We check that $|S| \leq k$, otherwise, S is not a solution for the decision problem.
2. Then we must check that every node different from the depot is allocated in exactly one of the tours, and that the depot node appears in all tours exactly

[2]Recall we distinguish between minimize the total inspection time and minimize the number of needed tours for inspection. They are two related problems, but not the same.

once. Otherwise, S is not a solution. This is clearly a simple task which can be done in polynomial time.

3. In addition, we must compute the total weight in each $s \in S$ and check that is less or equal than W. Otherwise, S is not a solution. This task is also performed in polynomial time. We define

$$w(s) = w(\{0, j_1, j_2, \dots, j_r\}) = w_{0j_1} + w_{j_1 j_2} + \cdots + w_{j_r 0}$$

as the total energy weight of $s \in S$.

If S passes all the checkings, then the instance is a YES instance for CSP decision problem. It is easy to see that the asymptotic running time of this certifier algorithm is polynomial. Namely, CSP problem is a NP one. Let's now see which of the NP-complete problems is easier to polynomial transform to the CSP problem. The classical NP-complete problem that we have chosen here is the Bin Packing decision problem: deciding if N items can fit into a specified number k of bins, given an array w_j of weights associated to each item $j \in \{1, \dots, N\}$ and capacity value c for the bins. We are following the steps of the recipe now:

1. We have already shown that $CSP \in NP$.
2. The NP-complete problem that we use in the proof will be BPP.
3. Now we are going to describe an algorithm to define the function f which transforms an instance of BPP into an instance of CSPP. Consider x of BPP, $x = (N, k, (w_j), c)$. Let f be the function which maps $x \to f(x)$, which is an instance for CSP decision problem, then

$$f(x) = f(N, k, (w_1, \dots, w_N), c) = \left(N+1, k, \begin{pmatrix} 0 & w_1 & w_2 & \cdots & w_N \\ w_0 & 0 & w_2 & \cdots & w_N \\ w_0 & w_1 & 0 & \cdots & w_N \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_0 & w_1 & w_2 & \cdots & 0 \end{pmatrix}, c + w_0 \right),$$

where w_0 can be chosen as $\frac{1}{N} \sum_{j=1}^{N} w_j$, for example. Note that the matrix (w_{ij}) is formed using w_0 and the (w_j) weights array. This matrix represents the arc costs in a complete digraph $G = (V, E, C)$ where $V = \{0, 1, \dots, N\}$ is the node set, $E = \{(i, j) \colon u \neq v \in V\}$ is the edge set, and $C = \{c_{ij} = w_j \colon (i, j) \in E\}$ is the edge costs set. The computation of the algorithm is almost direct, and is clear that it runs in polynomial time (the most complicated operations are computing a mean and reordering $(w_0, w_1, \ldots, w_N)$ in a specific way in a $(N+1) \times (N+1)$ matrix). In Figure 3.2 it is shown how this function works.

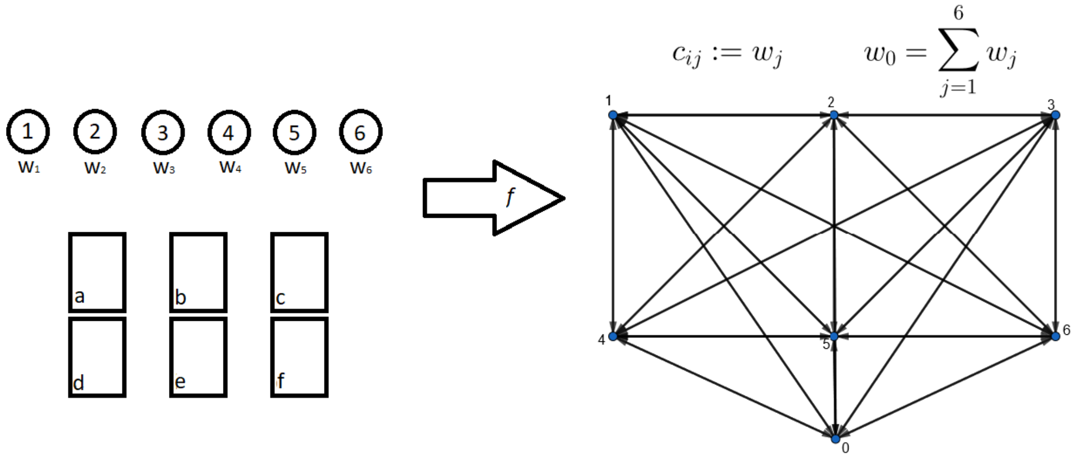

Figure 3.2: Transforming an example of BPP instance into a CSP one by f

4. The fourth step is about prove that x is a YES instance for BPP if and only if $f(x)$ is a YES instance for CSPP.
   Suppose, first, that x is a YES instance for BPP, so there exists a collection B of bins $b_l = \{i_1, \ldots, i_r\}$ such that $|B| \leq k$ and $w(b_l) := w_{i_1} + \cdots + w_{i_r} \leq c$ for each $b_l \in B$. Now consider $f(x) = (N+1, k, (w_{ij}), c + w_0)$. For each b_l we can construct $C_l = \{0, i_1, \ldots, i_r\}$, which is a cycle in G that starts from zero node and don't repeat intermediate nodes (as all items in b_l are different). It's pretty evident, by Figure 3.3 that $w(C_l) = w_{i_1} + \cdots + w_{i_r} + w_0 \leq c + w_0$. Then, each feasible bin produces a feasible cycle. Besides, there the same number of bins that cycles, as we have defined a one-to-one correspondence here, $|\{C_l\}| \leq k$. So $f(x)$ is a YES instance for CSPP.
   On the other hand, suppose that $f(x) = (N+1, k, (w_ij), c + w_0)$ is a YES instance for CSPP. Then there exists a collection of cycles C such that $|C| \leq k$ and for each $C_l = \{0, i_1, \ldots, i_r\} \in C$, $w(C_l) = w_{i_1} + \cdots + w_{i_r} + w_0 \leq c + w_0$. Let's define $b_l = \{i_1, \ldots, i_r\}$. Since $w_0 + w(b_l) = w(C_l) \leq c + w_0$, then $w(b_l) \leq c$ and then b_l is a feasible bin. As before, we have the same number of bins that cycles, hence $|\{b_l\}| \leq k$. Thus, x is a YES instance for BPP. Further explanation can be found in Figure 3.3.

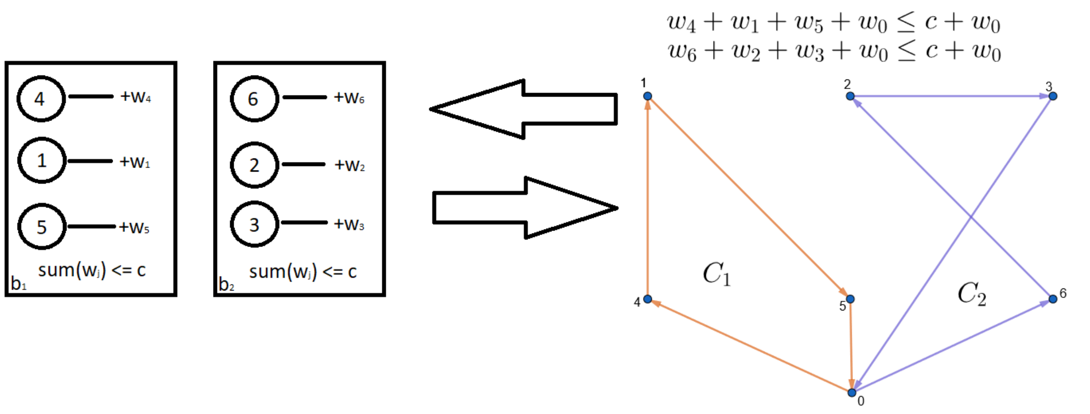
$w_4 + w_1 + w_5 + w_0 \leq c + w_0$
$w_6 + w_2 + w_3 + w_0 \leq c + w_0$
4
+w₄
1
+w₁
5
+w₅
sum(wᵢ) <= c
b₁
6
+w₆
2
+w₂
3
+w₃
sum(wᵢ) <= c
b₂
C_1
C_2
0
1
2
3
4
5
6

Figure 3.3: We have that x is a YES instance for BPP if and only if $f(x)$ is a YES instance for CSP. Where x is the example instance from Figure 3.2

5. We have noted yet that the algorithm which defines f runs in polynomial time. This ends the proof.

∎

In the next section we will explain how transform classical algorithms and procedures for solving known NP problems to solve CSP instances.

3.2 Algorithms for solving CSP problem

In this section we will show the modifications we made to classical algorithms for solving the CSP problems 1 and 2.

3.2.1 Adapting BPP heuristics

In the Bin Packing Problem section we described four heuristics that could be adapted to the CSP problem 2. Although our problem is similar to BPP, there is a main difference: CSP weights depend on previous visited node. This affects in two ways: we have to record our last location and also we did not know at first how to sort the items if we would want to apply the sorting versions of these heuristics. For example, let us analyze the following adaptation of Best-Fit algorithm:

```python
def bestfitCSP(N,w,W):
    items = list(range(1,N+1))
    bins = [[]]*N
    last_item = 0
    res = 0# Initialize result (Count of bins)
    # Create an array to store remaining space in bins
    # there can be at most n bins
    bin_rem = [W]*N
    # Place items one by one
    while items:
        # Find the first bin that can accommodate weight[i]
        j = 0
        # Initialize minimum space left and index of best bin
        mini = W + 1
        bi = 0
        item = items.pop(0)
        for j in range(res):
            last_item = bins[j][-1]
            travel = w[last_item][item]+w[item][0]
            c1 = bin_rem[j] >= travel
            c2 = bin_rem[j] - travel < mini
            if c1 and c2:
                bi = j
                mini = bin_rem[j] - travel
                last_item_bi = last_item
        # If no bin could accommodate weight[i], create a new bin
        if (mini == W + 1):
            bin_rem[res] -= w[0][item]
            bins[res] = [0,item]
            res += 1
            last_item = item
        else: # Assign the item to best bin
            bin_rem[bi] -= w[last_item_bi][item]
            bins[bi].append(item)
    solution = [b for b in bins if len(b)>0]
    return solution,res
```

We first initialize some lists for the $N+1$ items (inspection and depot nodes), the N possible bins, and the remaining space or energy in each bin. We also initialize the last item variable at 0 (we start from the depot node). Then we enter a loop until items list is empty. At each iteration, we get the index of the next item to assign from the remaining unassigned items. Once we have the next item, we search for the bin with minimum space left and its index. Note that weights depend on last item which is assigned to each bin. If the item can fit in any of the bins, then a new one is filled. Otherwise, we assign it to the best bin, those with minimum space left. Also we have added some variables to save and return the solution assignment of items. The adaptations are pretty similar for all the mentioned heuristics and their sorting versions, then we omit here the other codes. In the sorting versions, the main difference is that we select the next item by a given ordering procedure. In this work, we select the next item which returns the smaller weight, given the last added item to each bin. We denote these versions by NFS, FFS, BFS and WFS.

3.2.2 Implementing GBPP with Gurobi

In this section we will analyze the implementation of the Generalized Bin Packing Problem formulation with Python and Gurobi. To accomplish this task, we have defined a Python function, *SolverBPP_MIP*, which takes an input instance and returns the exact solution for GBPP as a MIP problem.

```python
def SolverBPP_MIP(instance):
    #Getting instance relevant parameters
    _,n,w,W,_,_ = instance
    points = [i for i in range(1,n+1)] #points to inspect
    nodes = [0]+points #nodes set

    # Creating Gurobi model
    model = Model("Generalized_BPP")

    # Defining variables
    ind_y_vars = [i for i in nodes]
    y = model.addVars(ind_y_vars,vtype=GRB.BINARY,name='y')
    ind_a_vars = [(i,j) for i in nodes for j in nodes if i!=j]
    a = model.addVars(ind_a_vars,vtype=GRB.BINARY,name='a')
    ind_x_vars = [(j,k) for j in nodes for k in points]
    x = model.addVars(ind_x_vars,vtype=GRB.BINARY,name='x')
    ind_s_vars = [(i,j,k) for i in nodes for j in nodes for k in
   points if j!=i]
    s = model.addVars(ind_s_vars,vtype=GRB.INTEGER,lb=0,ub=2,name
   ='sum')
    z = model.addVars(ind_a_vars,vtype=GRB.INTEGER,lb=1,ub=2,name
   ='summax')
    b = model.addVars(ind_s_vars,vtype=GRB.BINARY,name='bmult')
    ind_L_vars = [i for i in points]
    L = model.addVars(ind_L_vars,vtype=GRB.CONTINUOUS,lb=0,ub=W,
   name='L')

```

```
    # Setting objective function
    model.setObjective(quicksum(y[i] for i in nodes),GRB.MINIMIZE)

    # Defining Constraints
    model.addConstrs(quicksum(a[i,j]*w[i,j]*b[i,j,k] for i in nodes for j in nodes if j!=i)<=W*y[k] for k in points)
    model.addConstrs(b[i,j,k] == x[j,k]*x[i,k] for i in nodes for j in nodes for k in points if j!=i)
    model.addConstrs(quicksum(x[j,k] for k in points)==1 for j in points)
    model.addConstrs(quicksum(x[j,k] for j in nodes)==quicksum(a[i,j]*b[i,j,k] for i in nodes for j in nodes if j!=i) for k in points)
    model.addConstrs(quicksum(a[i,j] for i in nodes if j!=i)==1 for j in points)
    model.addConstrs(quicksum(a[i,j] for j in nodes if j!=i)==1 for i in points)
    model.addConstrs(2*a[i,j]<=z[i,j] for i in nodes for j in nodes if j!=i)
    model.addConstr(quicksum(x[0,k] for k in points)==quicksum(y[i] for i in points))
    model.addConstr(quicksum(a[i,0] for i in points)==quicksum(y[i] for i in points))
    model.addConstr(quicksum(a[0,j] for j in points)==quicksum(y[i] for i in points))
    model.addConstrs(w[i,j]*a[i,j] <= L[j] for i in nodes for j in points if j!=i)
    model.addConstrs(L[i]+w[i,j]*a[i,j]-W*(1-a[i,j]) <= L[j] for i in points for j in points if j!=i)
    model.addConstrs(s[i,j,k] == x[i,k]+x[j,k] for i in nodes for j in nodes for k in points if j!=i)
    model.addConstrs(z[i,j] == max_(s[i,j,k] for k in points) for i in nodes for j in nodes if j!=i)

    # Optimization
    model.params.outputflag = 0 #Silent mode: 0
    model.optimize()

    # Extracting solution in a more fancy structure
    b = model.ObjVal
    x_sol = np.zeros((n+1,n))
    x_list = []
    a_sol = np.zeros((n+1,n+1))
    a_list = []
    for v in model.getVars():
        if v.varName[0] in ['s','L','y','b']:
            continue
        if v.xn > 0.9:
            row,col = [int(i) for i in v.VarName[2:-1].split(',')]
            if v.VarName[0] == 'x':
                x_sol[row,col-1] = float(v.xn)
                x_list.append((row,col,float(v.xn)))
            else:
                a_sol[row,col] = float(v.xn)
                a_list.append((row,col))
    a_list_copy = a_list.copy()
    sequences = []
    while a_list_copy:
        seq = []
        ai = a_list_copy.pop(0)
        last = ai[1]
        seq.append(ai)
        while last != 0:
            al = [aj for aj in a_list_copy if aj[0]==last]
            if len(al) == 0:
                print("FAILED")
                break
            ai = al[0]
            last = ai[1]
            seq.append(ai)
            a_list_copy = [a for a in a_list_copy if a!=ai]
        sequences.append(seq)
    solution = []
    for seq in sequences:
        sol = [t[0] for t in seq]
        solution.append(sol)

    return solution
```

We first extract the weights w_{ij}, total weight limit W and the number of inspection nodes n. We use n to define points and nodes sets, which correspond to $\overline{V}$ or K and V in GBPP formulation. Then we generate the Gurobi model and define the variables x_{jk}, a_{ij}, y_k, L_j and two auxiliar ones, $s_{ijk} = x_{ik} + x_{jk} \in \{0, 1, 2\}$ and $z_{ij} = \max_{k \in K} x_{ik} + x_{jk} \in \{1, 2\}$, which we use to implement the constraints $2y_{ij} \leq \max_{k \in K} x_{ik} + x_{jk}$ for $i \neq j = 0, \ldots, n$. We also define variables b_{ijk} which will stand for products $x_{ik}x_{jk}$. After that, we add the objective function and the constraints to our problem. The objective function minimizes the number of initialized bins. The constraints are those described in Chapter 2, in the GBPP section. The only difference is the definition of auxiliary variables to compute the products and the maximum sum of x_{jk} pairs. Finally, we launch the optimization and then extract the solution in a more convenient format: we would like to return a covering of feasible tours with minimum size. We express it as a list of tours, for example $solution = [[0, 1, 2, 3], [0, 4, 5, 6]]$ means that we cover the inspection with two tours which are walked in that order.

As long as we have seen, Gurobi Optimizer can not solve GBPP instances for $N = 40$, at least with an academic license. However, we managed to solve $N = 20$ instances, which is half a faced batches pair. Therefore, we can halve the pair and solve each half as a CSP instance, joining the solutions as a global one.

3.2.3 Implementing GMKP with Gurobi

The implementation of the Generalized Multiple Knapsack Problem formulation with Python and Gurobi is pretty similar to the given for GBPP. The main differences with GBPP implementation are the following:

- It solves CSP problem 1. That is, it maximizes total inspection profit. We define the profit of inspections as negative inspection time costs so that maximizing profits is minimizing time costs. Total profits closer to 0 mean that we spend little total inspection time.

```
model.setObjective(quicksum(y[i,j]*p[i,j]*b[i,j,k] for i in
    nodes for j in nodes for k in K if j!=i),GRB.MAXIMIZE)
```

- GMKP takes the number of initialized bins m as a constant. For example:

```
model.addConstr(quicksum(x[0,k] for k in K)==m)
```

  Since we do not know which m is the best for solving CSP problem 1, we can find the minimum number of feasible tours that is needed for covering the inspection, m^*, using GBPP. Then, we will have that GMKP is feasible for $m = m^*, m^* + 1, \ldots, n$. Finally, we can perform a search of best the covering in the multiple defined GMKP problems. We usually will find that is enough to only solve instances for $m = m^*, m^* + 1, m^* + 2$ in order to obtain satisfactory solutions.

Since we cannot find solutions for GBPP with $N = 40$ in a satisfactory amount of time, we are not able to solve GMKP for these large instances. However, we could solve the problem for $N = 20$, which is half faced batches pair. In addition, we could consider relaxations for the GMKP, as the neighboring nodes one, which states that the only allowed edges are the associated with neighboring nodes and the associated with the depot node. This relaxation allows us to solve instances until $N = 40$ nodes.

3.2.4 Adapting ECP algorithms

In this section we will explain how we combine Johnson's algorithm and Algorithm X to solve the CSP problems (versions 1 and 2). The code is the following:

```python
import numpy as np
from algorithm_x import AlgorithmX
import graph_utils as gu
from graph_utils import tic,toc
def SolverECP(instance,cycle_limit_length=13,
             maxit = -1,timeLimit=-1,stop_when_imp=0,plot=True):
    # Getting instance
    G,N,w,W,p,pos2 = instance
    print('\n'+'-'*30+' ECP SOLVER '+'-'*30)

    # Getting cycles. Applying Johnson's algorithm
    limit = cycle_limit_length
    C = list(gu.limit_simple_cycles(G,limit,start=0))
    Cw = [c for c in C if gu.GetCycleWeight(c,w)<=W]

    # Solving ECP. Applying AlgorithmX with DLX implementation
    C0 = [c[1:] for c in Cw]
    C0.sort(key=lambda x:-len(x))
    solver = AlgorithmX(N)
    for i,c0 in enumerate(C0):
        solver.appendRow(np.array(c0)-1, i+1)
    solutions = solver.solve()

    # Getting optimal covers
    tStart = tic()
    i0 = 0
    C0_arr = np.array(C0,dtype='object')
    min_len_sols = []
    min_len = N
    max_profit_sols = []
    max_profit = -1e10
    n_covers = 0
    tsol = gu.GetcurrentSolution(N, W, w)
    tsol_profit = gu.GetCoverProfit(tsol, p)
    tStartGetSols = tic()
    print("Starting getting covers process... (feasible cycles: {})"\
        .format(len(Cw)))
    for solution in solutions:
        cover_rows = [s-1 for s in solution]
        cover = C0_arr[cover_rows]
        current_profit = gu.GetCoverProfit(cover,p)
        current_len = len(cover)
        n_covers += 1
        if current_len < min_len:
            min_len = current_len
            min_len_sols = [cover]
        elif current_len == min_len:
            min_len_sols.append(cover)
        if current_profit > max_profit:
            max_profit = current_profit
            max_profit_sols = [cover]
        elif current_profit == max_profit:
            max_profit_sols.append(cover)
        # Terminate because of max iteration exceeded
        i0 += 1
        if maxit > 0:
            if i0 >= maxit:
                print("\nMAXIT_Warning: there are more than {} covers!"\
                    .format(maxit))
                break
        # Terminate because of time limit exceeded
        if timeLimit>0:
            if toc(tStartGetSols)>timeLimit:
                print("\nTIMELIM_Warning: processing has exceeded {}s of processing"\
                    .format(timeLimit))
                break
        # Terminate because of max iteration exceeded
        if stop_when_imp > 0:
            current_imp = (tsol_profit-current_profit)/tsol_profit
            if current_imp >= stop_when_imp:
                print("\nReached an improvement of {:.2f}% over current solution"\
                    .format(current_imp*100))
                break
        # Stats
```

```python
        if i0%10000==0:
            print("\rECP (current,best) - iter: {}, profit: ({:.2f},{:.2f}), len: ({},{})"\
                    .format(i0,current_profit,max_profit,current_len,min_len),end='')
    # Optimizing criteria
    ## Max profit criteria
    F = max_profit_sols
    nmax_len = len(F)
    ## Min length criteria
    F1 = min_len_sols
    nmin_len = len(F1)
    F1.sort(key = lambda x: gu.GetCoverProfit(x,p))
    print("Ellapsed (global): {}, CycleLengthLimit: {}, FeasibleCycles: {}, Covers: {}".format(toc(tStart),limit,len(Cw),n_covers))

    # Print sol
    print("RESULTS SUMMARY (min length criteria, nsols: {}):".format(nmin_len))
    print("\tItems in each bin:")
    for i,kn in enumerate(F1[-1]):
        s = [0]+[s for s in F1[-1] if set(s)==set(kn)][0]
        print("\tBin {}: {}\tOrdering: {}".format(i+1,[0]+kn,s))
    print()
    print("RESULTS SUMMARY (max profit criteria, nsols: {}):".format(nmax_len))
    print("\tItems in each bin:")
    for i,kn in enumerate(F[-1]):
        s = [0]+[s for s in F[-1] if set(s)==set(kn)][0]
        print("\tBin {}: {}\tOrdering: {}".format(i+1,[0]+kn,s))
    print()
    print("current solution: Profit: {:.2f},\tN.bins: {}".format(gu.GetCoverProfit(tsol,p),len(tsol)))
    print("{}: Profit: {:.2f},\tImprovement: {:.2f}%,\tN.bins: {}".format(
        "ECP (min length criteria)",
        gu.GetCoverProfit(F1[-1],p),
        (gu.GetCoverProfit(tsol, p)-gu.GetCoverProfit(F1[-1],p))/gu.GetCoverProfit(tsol, p)*100,
        len(F1[0])))
    print("{}: Profit: {:.2f},\tImprovement: {:.2f}%,\tN.bins: {}".format(
        "ECP (max profit criteria)",
        gu.GetCoverProfit(F[-1],p),
        (gu.GetCoverProfit(tsol, p)-gu.GetCoverProfit(F[-1],p))/gu.GetCoverProfit(tsol, p)*100,
        len(F[-1])))
    print('\n'+'-'*25+' END OF SOLVER '+'-'*25)

    # Plot sol
    if plot: gu.PlotSolution(F1[-1],pos2,sufix=' ECP (min length) N={}'.format(N))
    if plot: gu.PlotSolution(F[-1],pos2,sufix=' ECP (max profit) N={}'.format(N))

    return F[-1],F1[-1]
```

We first get the instance parameters and define a warning message. We will talk about the warning later. The procedure consists in:

1. Firstly, we use Johnson's algorithm with limited cycles length to find all the elementary cycles of length $\leq L$, starting at the depot node, in the given instance.
2. Then, we keep only the feasible cycles. This define an input for Exact Cover Problem.
3. We solve the Exact Cover Problem for $X = \{1, \dots, n\}$ and S the obtained feasible cycles starting at depot node.
4. Once we have collect all the possible exact covering sets, we start a search of the optimal solutions for both CSP problem 1 and CSP problem 2. Solutions for CSP problem 2 are sorted by total inspection cost in order to return that with minimal needed time.

This last part can be very time expensive. To overcome this, we have added some terminations parameter to stop the search before be completed. If we had to do this, then a warning text would be displayed. In the algorithm, profits of coverings are computed using function $GetCoverProfit$, which belongs to an auxiliary library.

We have mentioned that this procedure is not very efficient when we have to visit many inspection nodes, i.e, $N \geq 10$. For example, in a complete digraph with 11 nodes (10 inspection nodes plus the depot one) related to a realistic CSP instance, we have that the subcollection of feasible elementary cycles, S, has a size greater than $6.6 \cdot 10^6$. For this reason, in ECP approach we only solve CSP instances with the neighboring nodes relaxation: only edges associated with neighboring nodes (and all possible edges $(0, j), (j, 0)$ for $j = 1, \dots, N$) are considered. Doing this, a $N = 10$ realistic instance has $|S| \approx 1.6 \cdot 10^4$ feasible cycles which gives almost $3 \cdot 10^5$ possible exact coverings. However, even this relaxation is not enough for a $N = 20$ instance. Because of that we came up with the termination criteria. An alternative approach could be divide a the CSP instance into overlapped smaller sub-instances and then merge its solutions by a local search or another procedure in the overlapping zones. Both procedures might not return the optimal solution but may return a satisfactory one which improves the current solution adopted by the company.

4 | Results

In this chapter we show the results obtained for some particular CSP instances. We highlight that we want these instances to be as realistic as possible such that our results can guide the maintainer companies to improve their drone inspections performance. As we have commented in previous chapters, a CSP instance with $N = 40$ is computationally infeasible by the exact procedures described in this document. Also, a real CSP plant counts with many more than 40 nodes. However, we can follow different relaxation procedures to obtain good enough solutions, in the sense of improving the current company solution:

- We can solve exact CSPP1 and CSPP2 instances with GBPP and GMKP, respectively, for $N \leq 20$. This is enough to ensure that the current solution can be improved by other path planning strategies. An approach could be divide horizontally (that is, using the mean x-coordinate as the limit between halves) a pair of batches with $N \approx 40$ nodes into instances of $N \leq 20$ nodes. After solving the two instances we would define the global solution as the union of the solutions. For the depot node we consider two different locations (one in each sub-solution) because it is possible to change its position while the drone is working and the cost of this action is almost negligible.
- As we will see, there are certain patterns that appear in exact solutions for GMKP in the solved instances. These patterns make evident that solutions include mostly edges associated with neighboring nodes. This information may guide us to develop new heuristics based on the shape of the CSP instance graph. By the moment, we could state the following hypothesis: Solutions only composed of edges associated with neighboring nodes might be good solutions.
- The adapted BPP heuristics actually can solve big instances in few seconds, and we have found that the improvement over current company solution is usually very high (depending on the heuristic). When instances size grows sorting versions of heuristics still return good feasible solutions. The sorting function used

at each step to choose the next item performs an increasing sort of weights for remaining items.

- Recall that the target of this work is to improve path planning strategies currently used in drone inspections of entire CSP plants. Since we only can solve the problem for few nodes (and CSP plants might have many of them), we must divide our inspection task in small zones (solvable CSP instances). The global proposed solution will be the union of the solutions in the different zones or partitions of the plant. We can estimate the total inspection time of the full CSP inspection multiplying the total time in a solvable instance by the size of the plant partition. In this section we show the comparison using the actual CSP plant from Figure 2.1. This is a small size CSP plant with 156 pairs of upstream and downstream tubes, i.e, 156 inspection nodes.

Also, note that the CSP instances graphs present a lot of symmetry. This symmetry makes us think that the improvement margin is small, comparing to graphs with a different organization. Anyway, even if we only saved some minutes by instance, it could lead to a significant improvement when considering all the instances by plant. Suppose that we had a large CSP plant as the Noor Power Station at Morocco, which is a 510 MW plant. The CSP plant has approximately 500 inspection nodes (pairs of pipes to inspect). In that plant, saving minutes for 10 to 40 sized sub-instances may give an overall saving of hours.

During this section we will use some particular CSP instances. They are instances for $N = 10$, 20 and 40 inspection nodes. Their energy consumption weights and inspection time costs are closer to those experienced during the last CSP inspections performed by the company Virtualmechanics, based on conversations we had with some of the workers ([16]). They told us that the drone usually performs 9 or 10 inspections before returning to base station, and we also agreed that most expensive edges are those who force the drone to turn the most. In Figure A.4 (in the Appendix A) we can observe this realistic weights for the $N = 10$ CSP instance. We only have added edges associated with neighboring nodes and with the depot node for better clarity. In all the instances we consider a West to East wind direction. Due to this, horizontal and diagonal edges have $w_{ij} \neq w_{ji}$ while vertical ones have equal weights in both directions (recall that wind component of standard regression model for weights computation is $\vec{v}_{xy} \cdot \vec{\omega}_{xy}$). We observe a lot of symmetry in the edge weights, in fact, because the distances between neighboring nodes are constant, each node has the same weights for travelling to another neighboring node (different from the depot). We will see more evidences about this symmetry in the solutions obtained.

In addition, battery total limit W is set as $0.75 \cdot 274$ Wh, which is the maximum energy of battery models of the drone used in inspections. The 0.75 coefficient is for security: The drone must not run out of battery when it is flying over the critical zones of CSP plants. The current company solution for the 10 nodes instance is shown in Figure 4.1, which gives a covering with two feasible tours. We plot the tours as paths in figures to facilitate their comprehension and clarity. As we commented before, the drone usually returns after 9 to 10 inspections. Figures 4.2 and 4.3 do the same for the $N = 20, 40$ instances, which return a covering with 3 and 5 tours. In Figure A.5 we can see the profits p_{ij} for the neighboring edges and depot node of the 10 nodes CSP instance. The total profits for the instances $N = 10, 20, 40$ are 1920 s, 4004 s and 7978 s, respectively. During this section, our purpose will be to show results obtained with proposed algorithms and compare them with these current company solutions. We will see that, although we usually can not obtain exact solutions for big sized instances, the improvement over current company solution is good enough from an industrial point of view. The first part of the section is divided in sub-sections, each one related to a certain approach.

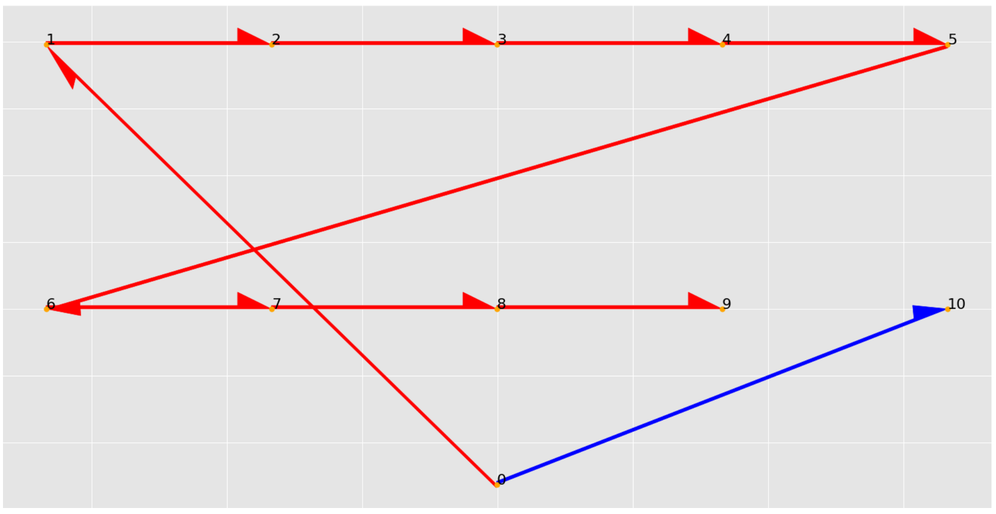

Figure 4.1: Total profit of current company solution with $N = 10$: 1920 $s = 32$ min

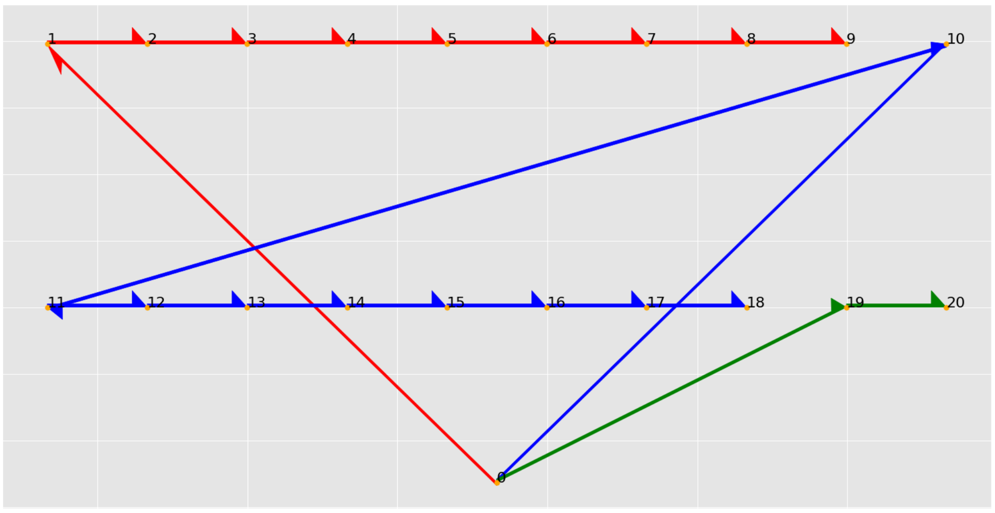

Figure 4.2: Total profit of current company solution with $N = 20$: $4004s\ s \approx 67\ min$

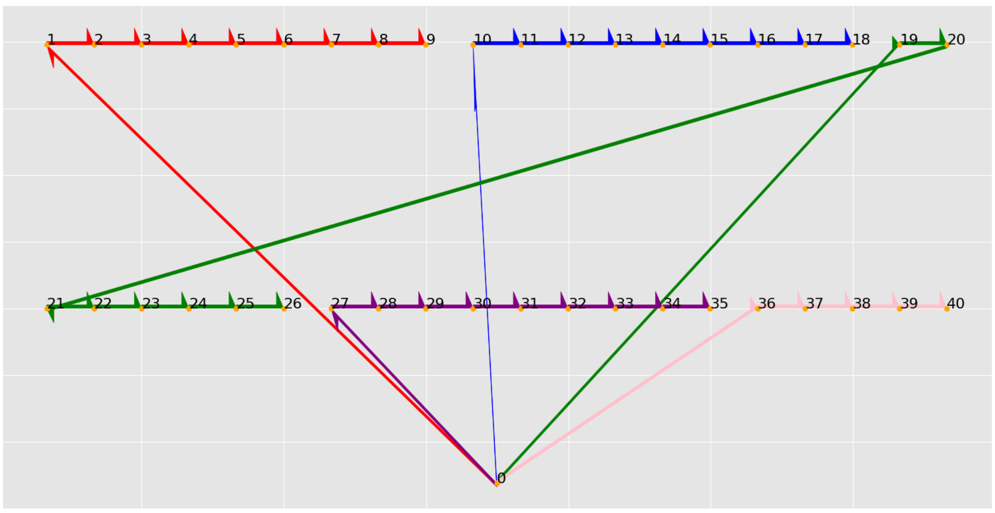

Figure 4.3: Total profit of current company solution with $N = 40$: $7978\ s \approx 132\ min$

4.1 BPP heuristics results

The results of the BPP-based heuristics are shown in this sub-section. This is the fastest approach for solving CSP instances due to its low computational cost. For example, an instance with $N = 40$ can be resolved in $0.01\ s$ with the heuristic BPP solver that we developed, which applies the four heuristics presented in this work and the sorting versions. However, we must keep in mind that this procedure solves CSP problem 2, although it may return good feasible solutions for CSP problem 1. For obtaining this results, we have implemented the adapted algorithms in Python and also have included them into a more fancy style. A typical output for the $N = 10$ instance presented before is the following:

```
1  -------------- BPP (Heuristic) SOLVER -------------
2  Ellapsed (global): 0.0
3  Current solution: Profit: -2007.00, N.bins: 2
4  NF: Profit: -2007.00,   Improvement: -0.00%,    N.bins: 2
5  NFS: Profit: -1565.00,  Improvement: 22.02%,    N.bins: 1
6  FF: Profit: -2007.00,   Improvement: -0.00%,    N.bins: 2
7  FFS: Profit: -1565.00,  Improvement: 22.02%,    N.bins: 1
8  BF: Profit: -2007.00,   Improvement: -0.00%,    N.bins: 2
9  BFS: Profit: -1565.00,  Improvement: 22.02%,    N.bins: 1
10 WF: Profit: -2007.00,   Improvement: -0.00%,    N.bins: 2
11 WFS: Profit: -1565.00,  Improvement: 22.02%,    N.bins: 1
12
13 ---------------- END OF SOLVER -----------------
```

The total elapsed time during the processing of the 4 heuristics in its two versions (normal and sorting) is displayed in line 1. Line 2 shows the profit and number of used bins (tours) for the current company solution. Next lines show the total profits (negative total inspection times), the improvement over the current company solution and number of used bins obtained, by each heuristic: Next-Fit (sorting), First-Fit (sorting), Best-Fit (sorting) and Worst-Fit (sorting). As we have highlighted, BPP heuristics solve CSP problem 2, that is, the minimization total tours problem. For that, we expected to rarely find good enough solutions for CSP problem 1 (minimizing total inspection time cost). However, as we can observe in the output for the $N = 10$ instance, we managed to improve the current solution in a 22.02% with the sorting versions of these heuristics. This saves 442 seconds, which is more than 7 minutes! CSP inspections are developed in all pipes of the plant ($\approx 160 - 180$ inspection nodes, in small CSP plants). So, if we divided the CSP plant in groups of ten nodes and apply the two strategies, we would obtain an overall improvement of approximately two hours! In Figures 4.4 and 4.5 we represent the two solutions obtained by these approaches (the FFS, BFS and WFS heuristics return the same covering). Also, we have surprisingly found that good enough solutions for the $N = 40$ instance are given by these procedures. Figures 4.7 and 4.6 show the solutions obtained (in this case the BFS, FFS and WFS also concur).

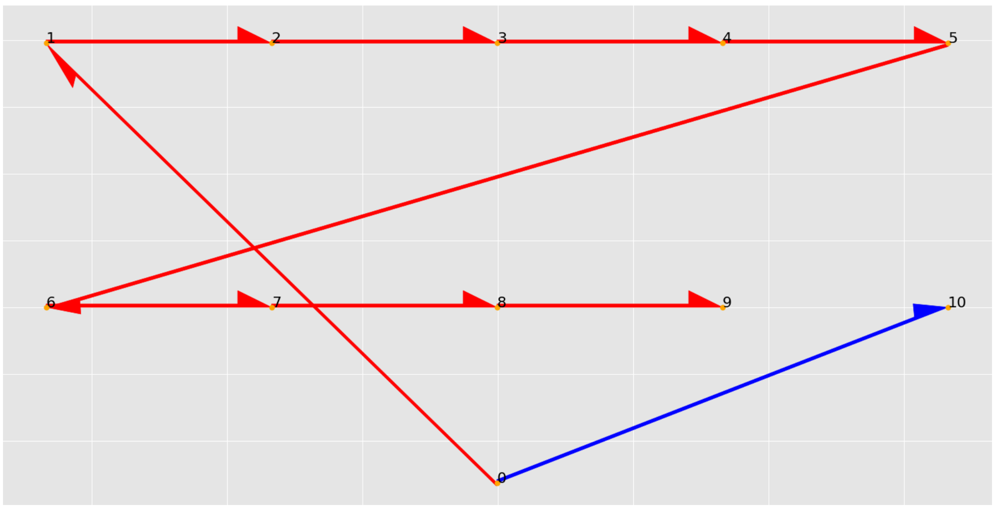

Figure 4.4: Total profit of NF, FF, BF and WF solutions with $N = 10$: 2007 s

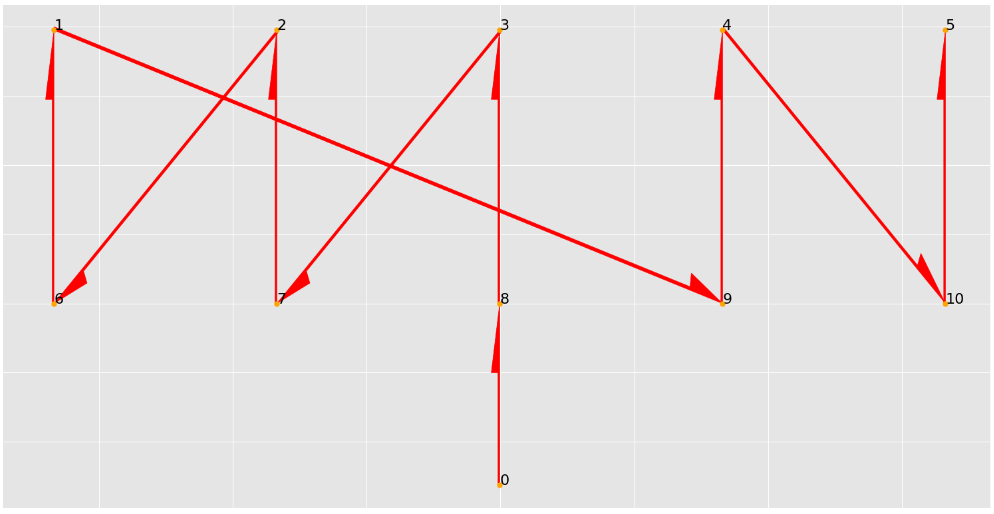

Figure 4.5: Total profit of FFS, BFS and WFS solutions with $N = 10$: 1565 s

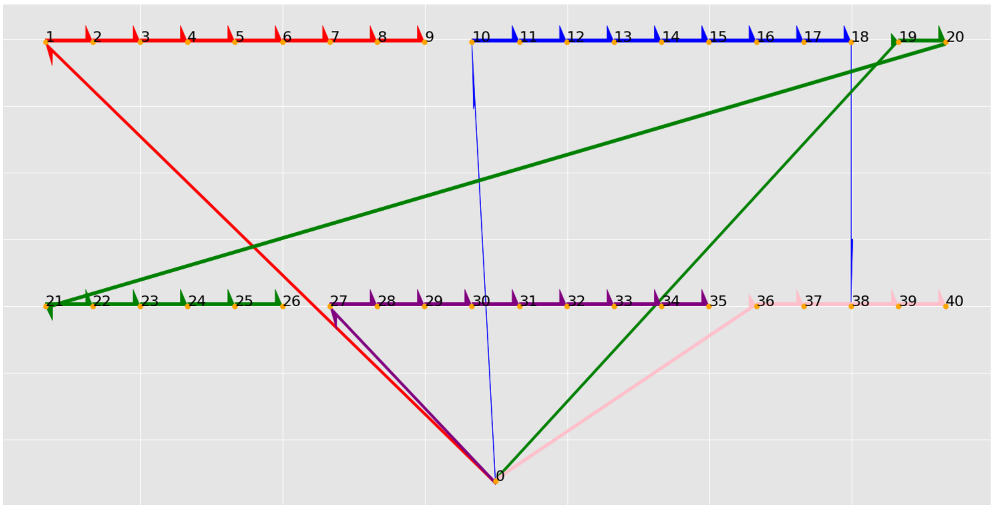

Figure 4.6: Total profit of FF and BF solutions with $N = 40$: $7944\ s$

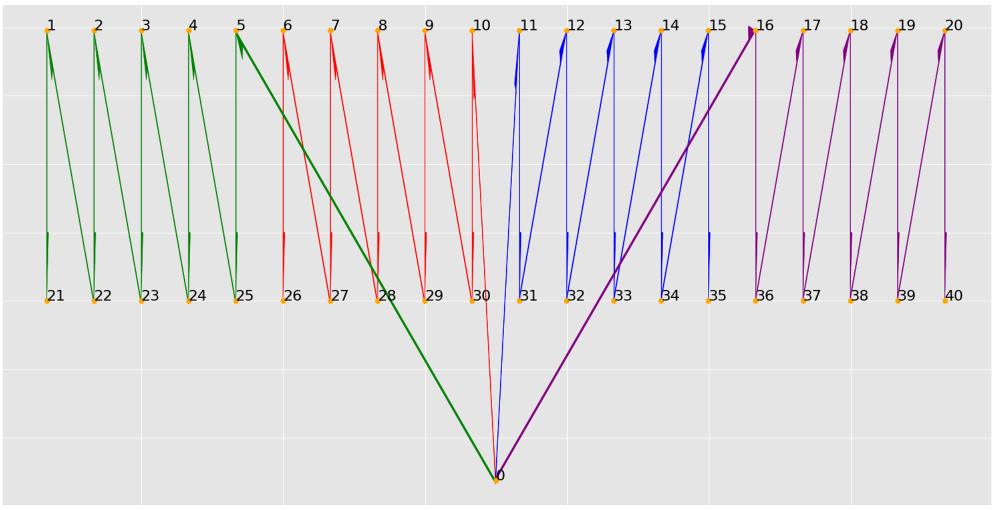

Figure 4.7: Total profit of NFS, FFS, BFS and WFS solutions with $N = 40$: $6466\ s$

4.2 GBPP results

The Generalized Bin Packing Problem that we have developed in this document aims to solve the CSP problem 2. As its purpose is not minimizing the total inspection time, the obtained solutions include many edge crossings that likely delay the total inspection time costs. In fact, we use this algorithm to feed the Generalized Multiple

Knapsack Problem, which needs to specify the number of used bins. If we combine the two procedures, we can obtain very good solutions for CSPP1, even exact ones for $N \leq 20$ instances. However, when N is bigger, for example 30 or 40, the problem becomes computationally infeasible, spending lot of time without finding solutions, even if we apply relaxations like the neighboring nodes ones or if we configure the Gurobi Optimizer to find at least approximate solutions. A typical output for a $N = 10$ instance is the following:

```
--------------- BPP (MIP) SOLVER ------------
Ellapsed (global): 2.6068475246429443
RESULTS SUMMARY (min length criteria, nsols: 2):
(Retrieving solution with maximum associated profit)
    Items in each bin:
    Bin 1: [0, 3, 5, 9, 10, 6, 1, 7, 4, 8, 2]

Current company solution: Profit: -2007.00, N.bins: 2
BPP-MIP: Profit: -1761.00,  Improvement: 12.26%,    N.bins: 1

--------------- END OF SOLVER ---------------
```

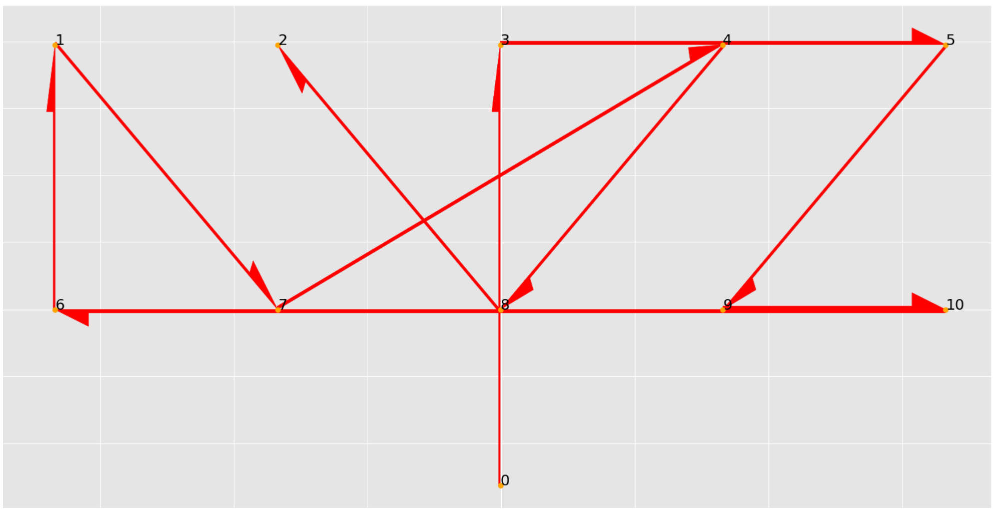

Figure 4.8: Total profit of GBPP exact solution with $N = 10$: 1761 s

Line 2 shows elapsed time solving the CSP instance. The next lines summary the results: the solver finds 2 solutions with minimum number of tours, and it selects that with minimum inspection time cost (maximum profit). After that it comes the solution and the summary of total profit, improvement over current company solution and number of bins. We observe that this instance returns a solution worse than the given ones by the previous heuristics. Figures 4.8 and 4.9 represent the exact solution of GBPP for the $N = 10$ and $N = 20$ instances.

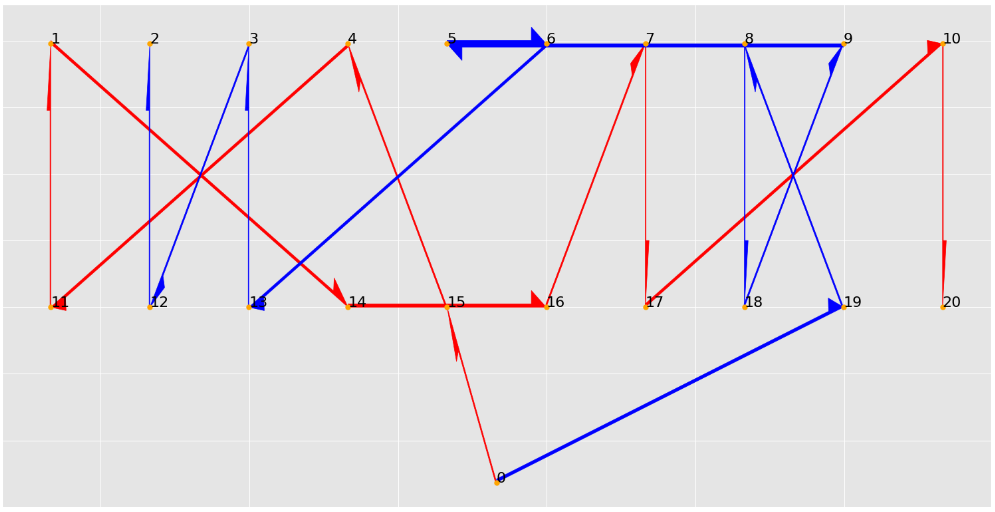

Figure 4.9: Total profit of GBPP exact solution. with $N = 20$: 3517 s

4.3 GMKP results

The GMKP problem is responsible for solving the CSP problem 1, that is, the total inspection time minimization problem. In order to relax this formulation when the size of the instance is big ($N = 30$ or $N = 40$), we can consider the hypothesis on the neighboring nodes and reduce the number of variables and constraints. Until $N \approx 20$ we can obtain exact solutions by this procedure. Recall that this formulation requires as input the number of used bins m. Our idea is to solve the CSP problem 2 by GBPP, obtaining the minimum number of feasible tours m^* needed for cover the graph and use it as input. Although technically (we saw an example) the best solution in terms of CSP problem 1 can use more tours than best solution for CSP problem 2, we have observed that this rarely occurs, probably due to the symmetry of the graphs. A typical output of this procedure for the $N = 10$ CSP instance is shown below.

```
------------------ MKP SOLVER -----------------------
Ellapsed (global): 0.1405172348022461
RESULTS SUMMARY (max profit criteria, nsols: 9):
    Items in each bin:
    Bin 1: [0, 8, 3, 7, 2, 6, 1, 10, 5, 9, 4]

```

```
Current company solution: Profit: -2007.00, N.bins: 2
GMKP: Profit: -1564.00, Improvement: 22.07%,    N.bins: 1

------------------- END OF SOLVER ------------------
```

Line 2 shows the elapsed time in the processing. The next lines show the results summary: the solver has found 9 optimal solutions, then it is shown the selected solution and the comparison between company solution profits and GMKP ones. The improvement ascend to 22.07%, the inspection of ten nodes lasts 1564 seconds. Figures 4.10 and 4.11 represent the optimal solutions of CSP problem 1 for instances with $N = 10$ and $N = 20$. We see that the solutions present a lot of symmetry and that mainly use edges associated with neighboring nodes. We also observe that the solutions avoid horizontal edges, which imply a longer turning phase than other ones, and focuses in use vertical and diagonal arcs. These patterns show that maybe a possible heuristic to develop could be one which smartly chooses between diagonal and vertical edges composing a good covering of feasible tours. As we commented before, the GBPP approach can not solve instances for $N > 20$ and therefore we do not know exactly the value of m^*, in order to use it as input for GMKP. For large instances we use the BPP heuristics, which usually give one or two more bins that optimal solutions, at least for the used instances. That information gives an estimation of a good m for feeding the GMKP, and we can also try use one or two less bins that the returned by the heuristics to see if it is feasible. Another interesting point in the GMKP is the neighboring relaxation: as commented above, if we only consider edges associated with neighboring nodes, we can relax the formulation and solve instances for $N \geq 30$. We call this the relaxed GMKP. This relaxation can also apply for the GBPP but we have found that it does not increase the solver performance very much. The exact solution for the relaxed GMKP with $N = 30$ is shown in Figure 4.12. We also have even found an exact solution for a $N = 40$ CSP instance using relaxed GMKP displayed in Figure 4.13. The gap between the best lower bound found and the objective value was of $< 2\%$ after 49 minutes of processing. After approximately 24 hours and 30 minutes, the gap was of 0.18%, with a best lower bound of $6448.5\ s$. The processing stopped with an elapsed time of 25 hours and 46 minutes. So, the final exact solution of $6460\ s$ is the optimal for the relaxed GMKP in the $N = 40$ instance and it would return a saving over the current company solution of 25 minutes and 20 seconds.

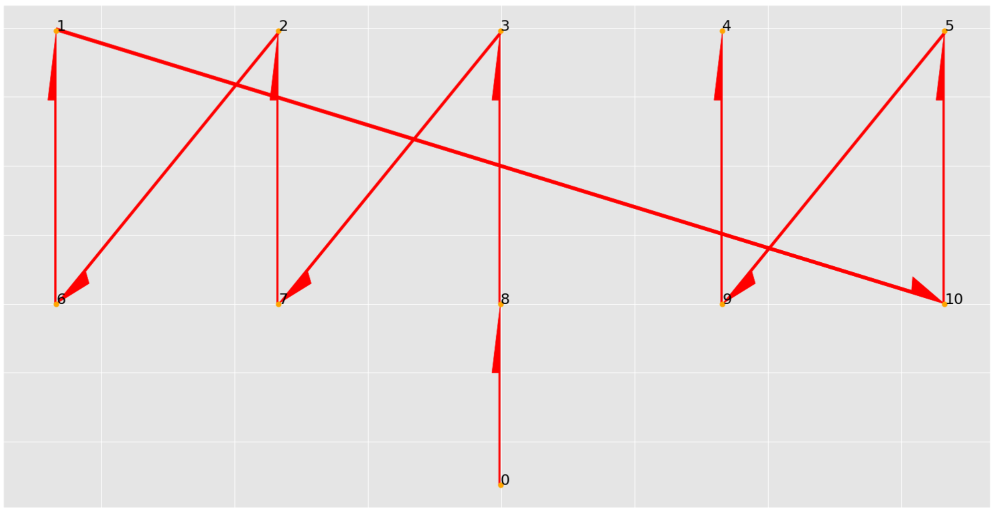

Figure 4.10: Total profit of GMKP exact solution with $N = 10$: $1564\ s$

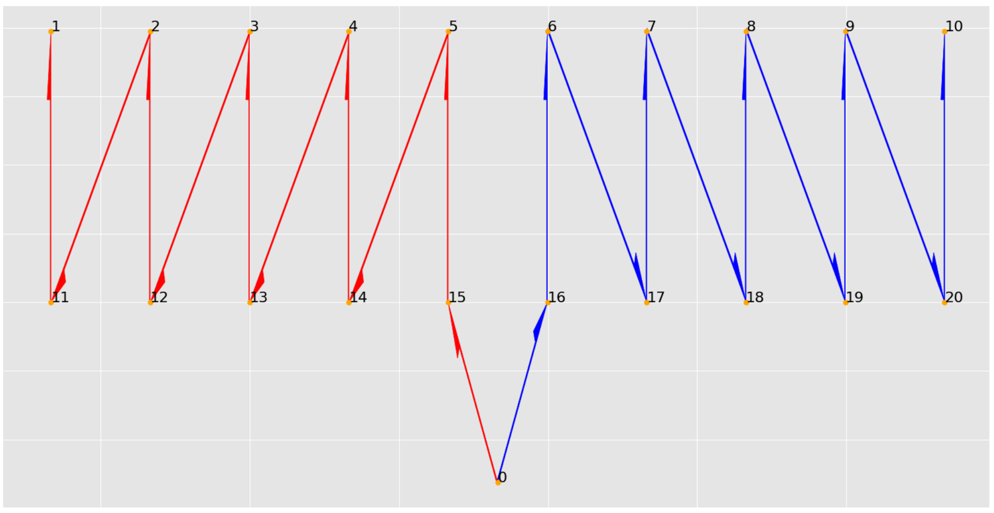

Figure 4.11: Total profit of GMKP exact solution with $N = 20$: $3158\ s$

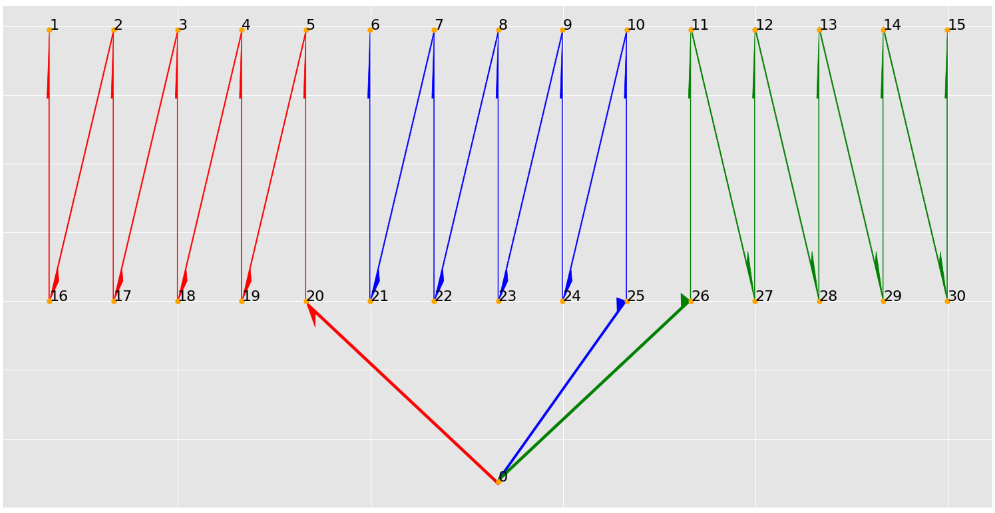

Figure 4.12: Total profit of relaxed GMKP exact solution with $N = 30$: $4837\ s$

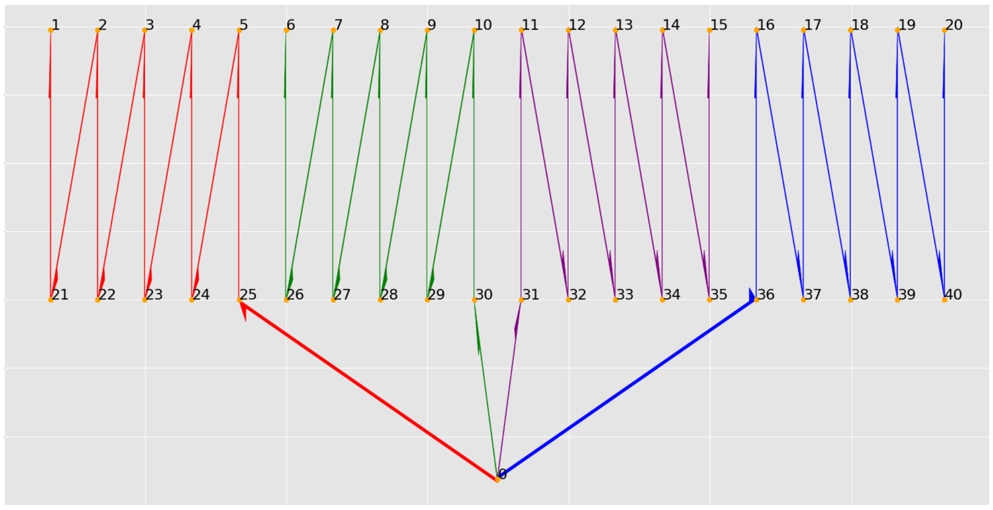

Figure 4.13: Total profit of relaxed GMKP approximate solution with $N = 40$: $6460\ s$

4.4 ECP results

Among all the procedures described in this document, Exact Cover Problem one is the most expensive from the computational point of view. We even can not compute the exact solution for the $N = 10$ instance. However, we have managed to obtain good enough solutions aggregating different relaxations:

- Instead of considering the complete digraph, we only keep the edges associated with neighboring edges and the ones who joint the depot node with the inspection nodes. This obviously leads to sub-optimal solutions but they are usually good enough solutions.
- We add some termination criteria to the algorithm, in the part that searches for optimal solutions for CSP problems 1 and 2. These criteria are three: maximum number of iterations of the search, time spent in the search and improvement over the current company solution.

Combining the relaxation and the termination criteria we could obtain solutions even for a $N = 40$ CSP instance. The output for the $N = 10$ instance is shown below:

```
------------------- ECP SOLVER ---------------------
Starting getting covers process...
ECP (current,best) - iter:
    290000, profit: (-1942.00,-1593.00), len: (4,1)
Ellapsed (global): 155.21034479141235, CycleLengthLimit: 13,
   FeasibleCycles: 15798, Covers: 299613
RESULTS SUMMARY (min length criteria, nsols: 1506):
    Items in each bin:
    Bin 1: [0, 6, 1, 7, 2, 8, 3, 9, 4, 10, 5]   Ordering: [0, 6,
   1, 7, 2, 8, 3, 9, 4, 10, 5]

RESULTS SUMMARY (max profit criteria, nsols: 2):
    Items in each bin:
    Bin 1: [0, 6, 1, 7, 2, 8, 3, 9, 4, 10, 5]   Ordering: [0, 6,
   1, 7, 2, 8, 3, 9, 4, 10, 5]

Current company solution: Profit: -2007.00, N.bins: 2
ECP (min length criteria): Profit: -1593.00,    Improvement:
   20.63%,    N.bins: 1
ECP (max profit criteria): Profit: -1593.00,    Improvement:
   20.63%,    N.bins: 1

----------------- END OF SOLVER -----------------
```

First line shows a message that helps to know when the searching part of the algorithm starts. After that we added a dynamic printing that shows the best solution found and the current solution at the iteration (this message updates after some iterations). The next lines show the counting of feasible cycles and covers found, and also the selected cycle length limit. As we can see, the cycle limit length is loose enough because there are no feasible cycles with that length. During the searching process the algorithm tries to solve both CSP problems, because we could find a counterexample where the minimum number of tours doesn't give the best total inspection time. Therefore, the results summary is divided in two sections, one for each problem. We usually find many coverings with minimal number of tours (1506 for the $N = 10$ instance). We note that the neighboring nodes relaxation makes the total profit to be sub-optimal, as we know that GMKP exact total profit is lower. However, it is only 29 seconds slower. Figures 4.14, 4.15, 4.16 illustrate approximate solutions for the instances with $N = 10, 20, 40$. In order to obtain the solutions for the two big sized instances, we used a time limit of 1 and 2 hours, respectively, as termination criteria. We also note that solutions abuse of diagonal and vertical edges and mainly avoid to use horizontal ones. This makes sense because horizontal edges imply most expensive turnings than vertical or diagonal ones. We highlight that we also can use the ECP approximate solutions to obtain a good m for feeding the GMKP and obtain a good solution.

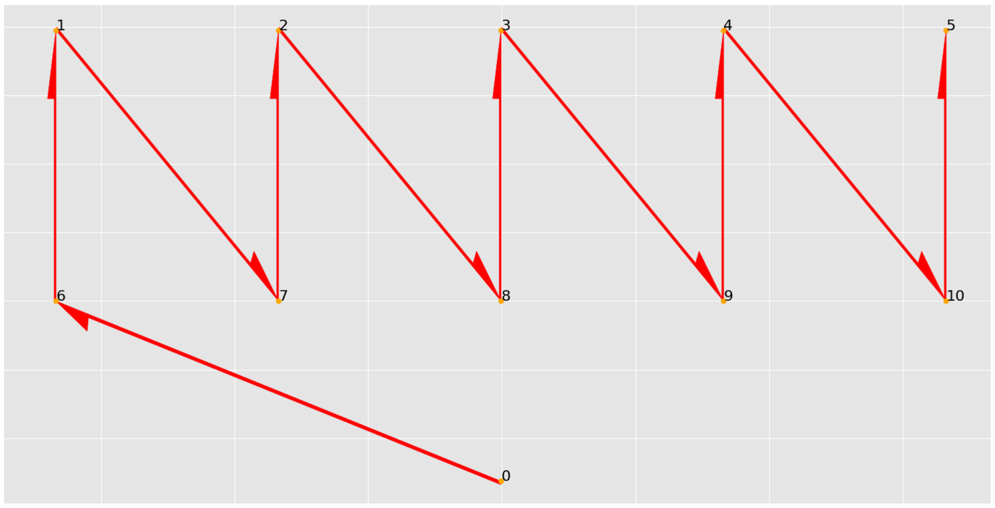

Figure 4.14: Total profit of relaxed ECP exact solution with $N = 10$: 1593 s

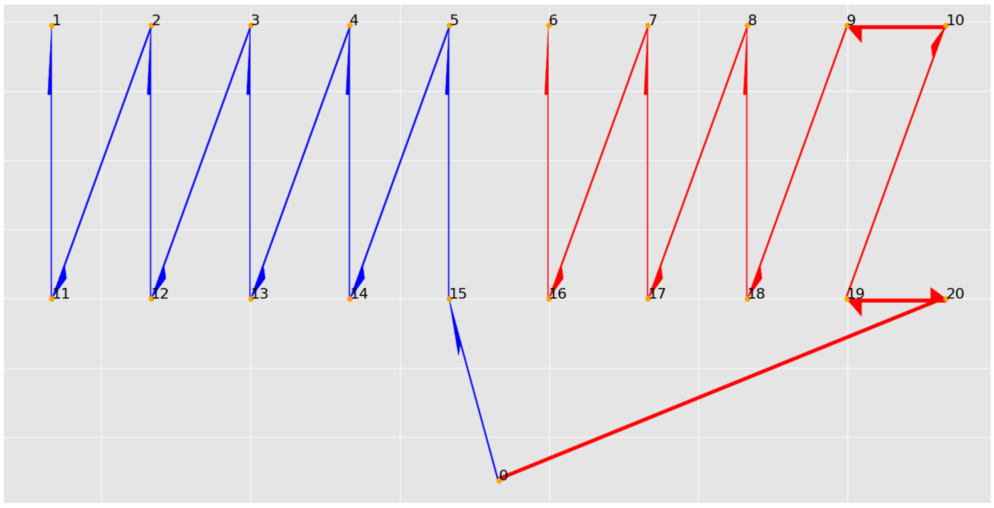

Figure 4.15: Total profit of relaxed ECP solution with $N = 20$ after 1 h of time processing: $3272\ s$

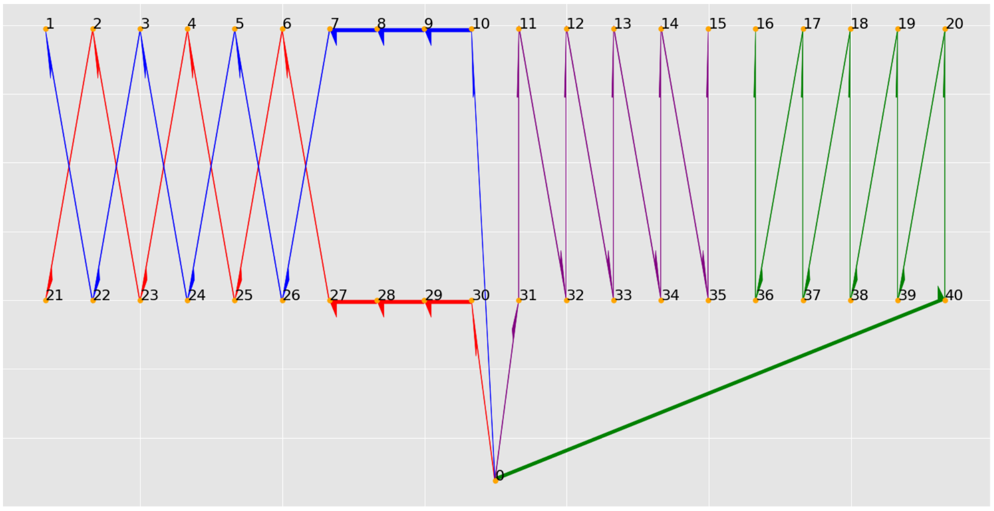

Figure 4.16: Total profit of relaxed ECP solution with $N = 40$ after 2 h of time processing: $6863\ s$

4.5 Comparisons

After analyzing the behaviour of the different approaches one by one, in this section we conclude with a direct comparison of them. We have been applying these procedures to a certain configuration of weights and profits which stands for realistic ones, for instances sizes of $N = 10, 20, 30, 40$. We can observe the patterns that the solutions manifest and improve the current company solutions from the beginning of the last section. The reader may probably think that solutions obtained are logical with the values of the graphs in Figures A.4 and A.5: The weights and profits present lot of symmetry. However, we study instances with graphs based on realistic CSP plants layouts, which present lot of symmetry too. Solutions are merely a consequence of this fact. We now elaborate on the comparison between the obtained solutions in instances with $N = 20, 40$. We focus in these medium and big sized instances because $N = 10$ instance was used as an example of individual studies.

Procedure	Nºbins	Total inspection time (s)	Improvement (%)	CSP Problem
Current	3	4004	—	1, 2
NF (NFD)	3 (2)	4004 (3160)	0.0 (21.08)	2
FF (FFD)	3 (2)	4004 (3160)	0.0 (21.08)	2
BF (BFD)	3 (2)	4004 (3160)	0.0 (21.08)	2
WF (WFD)	3 (2)	4004 (3160)	0.0 (21.08)	2
GBPP	**2**	3517	12.16	2
GMKP	2	**3158**	**21.13**	1
ECP	2*	3272*	18.28*	1, 2

Table 4.1: Comparison between different approaches for solving CSP problems 1 and 2 in a $N = 20$ instance. *: approximate solution for neighboring relaxation after 1 hour of searching process

The case $N = 20$ is displayed in Table 4.1. The columns of the comparison tables stand for: the used procedure, the number of used bins of the solution returned, the total inspection time costs, the improvement over current solution and the CSP problem which applies (although we could use feasible solutions in both, CSPP1 and CSPP2). For example, NF, FF, BF and WF approaches do not improve the current company solution. This is usually the behaviour of these procedures, that are very simple ones. However, their sorting versions obtain an improvement very high (21.08%), even better than obtained by ECP algorithm (18.28%). The four heuristic solutions are surpassed only by GMKP solution (21.13%). The GMKP solution, as an exact one, is the optimal for the CSP instance with $N = 20$ and the given configuration of weights and profits. GBPP is used to ensure that minimal number of tours is 2 (although is pretty obvious than graph can not be covered with one tour and heuristics approach has returned 2 as solution) and feed the GMKP exact solver. It is important to remark that bigger instances can have worse solutions than partitioning it into two sub-instances and the sum its profits: for example, $N = 10$ instance has as optimal value 1564 but $N = 20$ has $3158 > 2 \cdot 1564$. These is due to the position of the base station, which affects to edges joining the depot node with the others. However, this times are estimations and probably it is better to limit the moves of the base station. Solutions returned by GMKP and sorting BPP heuristics are very close, their objective values differ in 2 seconds. On the other hand, the elapsed processing times for the procedures are $\approx 35.23\ s$ in the GMKP and $< 0.01\ s$ in the heuristic solver, which is a large difference. The GBPP solver elapses 219.54 seconds to solve a $N = 20$ instance. The ECP solution is obtained after waiting one hour using the termination criteria.

Procedure	Nºbins	Total inspection time (s)	Improvement (%)	CSP Problem
Current	5	7978	—	1, 2
NF (NFD)	5 (4)	7978 (6466)	0.0 (18.95)	2
FF (FFD)	5 (4)	7944 (6466)	0.43 (18.95)	2
BF (BFD)	5 (4)	7944 (6466)	0.43 (18.95)	2
WF (WFD)	5 (4)	7978 (6466)	0.0 (18.95)	2
GBPP	—*	—*	—*	2
GMKP	4	**6460*****	**19.03*****	1
ECP	**4****	6863**	13.98**	1, 2

Table 4.2: Comparison between different approaches for solving CSP problems 1 and 2 in a $N = 40$ instance. *: we can not solve GBPP for $N = 40$. **: approximate solution for neighboring relaxation after 2 hour of searching process. ***: exact solution for the relaxed GMKP

Another comparison for a big size instance with 40 inspection nodes is presented in Table 4.2. This instance stands for two faced batches in a CSP plant, like in Figure 1.6. This is the more realistic instance solved by the procedures developed in this work. We ignore the optimal solution as we only managed to solve it by approximate and relaxed procedures, specifically, with neighboring relaxation and, in the case of ECP, with a termination criteria. The instance was not solvable by GBPP, at least in a satisfactory amount of time, so its solution is not displayed. As in the previous comparison, NF, FF, BF and WF solutions do not improve (or improve a little), the current solution. On the other hand, the sorting versions of heuristics perform very well (18.95%), obtaining improvements better than ECP (13.98%) and close to the one returned by GMKP (19.03%), which is the best solution of comparison. As we use a sorting function inside the algorithm for selecting the next item with the smallest weight, we indirectly restrict the procedure to neighboring nodes (although it is not impossible to choose another one). This explains why sorting BPP heuristics and

GMKP solutions are pretty similar, although they differ a lot in the processing time: GMKP spent more than one day to finding the solution while heuristics solver spends less than one second. Using the GMKP solution we can save 25 minutes every two batches pair. In small CSP plants with 8 batches, it leads to an overall saving of 1 hour and 40 minutes. In big sized CSP plants, as Noor, we can save until 5 hours and 15 minutes from the total inspection current company solution, which would spend 27 hours and 42 minutes. This could be the difference between spending 3 or 4 days in the CSP inspection. That is, this information may have positive economical implications.

5 | Conclusions

In this work, a collaborating research between university and industry, we have studied a realistic problem which affects to maintainer companies in the performance of drone inspections at CSP plants. CSP plants are essential for renewable energy and therefore, for the present and future of humanity. One main problem that arises in CSP plants is that collector tubes elements, which are very important components, present heat leakage like we would like to localize. As CSP plants are big extensions and also the points of interest are difficult to reach by humans, a drone inspection is suitable for the task of inspecting such points. Drone inspections are often used in critical environments like these in order to develop some maintenance tasks that are slow to perform by humans or by other procedures. The task developed by the drone consists in overfly the tubes of the plant while taking thermal pics. Currently, a collaborating company is performing the inspections with a path planning strategy based on intuition more than mathematical approaches, as they are more focused in the important data post-processing than in the optimization problem for collecting them. However, an improvement of the total time spent in inspections would clearly increase the economical profits of their work. Therefore, they raised the CSP problem. Our main purpose have been to study the problem of finding the best path planning strategy for performing this kind of drone inspections in CSP plants. Specifically, our aim is solving CSP problems 1 and 2, described in Chapter 1.

The tubes are grouped in couples that must be visited one after another. Because of this fact, we abstract the tubes couples to one interest point, and model the CSP plant layout as many of such points. In addition, CSP plants can be divided in zones called batches, which contain many points to inspect. A small sized CSP plant counts with 160 to 180 inspection nodes (8 batches). In larger CSP plants, this quantity ascends to ≈ 500 inspection points (12 to 14 batches). This situation could lead to very large instances to solve and we focus to smaller instances, for example two batches, and solve them. This also makes sense because the drone can not fly very far away from the pilot in critical environments.

We considered two problems related with the inspections of CSP plants, namely the CSP problem 1 and the CSP problem 2. The former asks for a minimization in the inspection total time while the latter proposes minimizing the number of tours. We have developed a series of procedures, which are based in classical optimization problems, we have adapted and implemented them in Python programming in order to solve CSP instances, and compared the results obtained with the current solution adopted by the company Virtualmech. The CSP instances where we evaluated the procedures were configured to be as realistic as possible in such a way the solutions obtained will be suitable estimations for real scenarios. The results show improvements over the current solution of almost 20% for instances with 40 inspection points. This supports the use of mathematics instead of only good intuition.

In addition, during the work we also have proven that the CSP problem 2 is NP-complete. We think that the CSP problem 1 could also be proven to be NP-complete, if we use the MKP in the reduction.

Finally, although we have made some advances in the study of path planning with drones at CSP plants, there are still some loose ends, which could give ideas for future research in this area:

- It is necessary to model the real energy consumption weights of the currently used drone. The company uses a DJI Matrice 300RTK[1] drone model. We have tried several times to extract the kinematics information from log files generated during the inspections but we have not succeed. It would be very useful for the continuity of the work to manage extract this information and then use it to compute the beta coefficients of the standard regression model explained in the Chapter 2.
- The formulations developed to solve the CSP problems 1 and 2, by exact procedures, GMKP and GBPP, respectively, are very complex formulations that use many variables and constraints. This makes difficult to solve big sized instances exactly. Developing new smaller formulations may be helpful to reduce the computational cost seems possible.
- Another research line could be to improve our control over Gurobi Optimizer. It includes several features and functionalities. A well-chosen configuration could increase the performance of the exact procedures and reduce the computational cost.

[1]https://www.dji.com/es/matrice-300/specs

- We can also experimented with the neighboring relaxations. For example, we have restricted the edges of the graph to be neighbors no further than 1 in the x coordinates, but we could increase that threshold and compare both the improvement and the elapsed processing time with the original neighboring relaxation.
- We observed that the obtained showed some particular patterns. These patterns could be divided into: using a zigzag strategy and using a diagonal strategy. That is, we start from a point and move only using vertical or diagonal edges associated with neighboring nodes. We could use this information as an idea for developing a smart heuristic algorithm which sequentially decides between using some types of edges to cover the graph of the CSP instance.
- Finally, in real situations, the edge weights are changing during a working day, for example, with wind changes. Therefore, there is a clear need in the inspection of CSP plants for online route planning algorithms that use incomplete information that is acquired in a dynamic manner.

A Kinematics for energy consumption weights and time costs

In this appendix, the aim is to compute realistic values for energy consumption weights w_{ij}. We use basic classical kinematics equations to model the kinematics state, $\boldsymbol{s}$ of a drone travelling during the CSP inspection. The given inputs are:

- v_{xy}^{max} m/s, the maximum horizontal speed (in module) reached by the drone.
- v_{z}^{max} m/s, maximum speed module while ascending.
- v_{z}^{-max} m/s, maximum speed module while descending.
- $a_{xy,h}^{a}$ m/s^2, horizontal acceleration module of the drone when it's moving horizontally, during the acceleration part.
- $a_{xy,h}^{b}$ m/s^2, horizontal acceleration module of the drone when it's moving horizontally, during the braking part.
- $a_{z,h}$ m/s^2, vertical acceleration component of the drone when it's moving horizontally. It remains constant in the three parts, keeping the drone at the same z coordinate.
- $a_{z,t}^{a}$ m/s^2, vertical acceleration component of the drone when it's taking off, during acceleration part.
- $a_{z,t}^{b}$ m/s^2, vertical acceleration component of the drone when it's taking off, during braking part.
- $a_{z,l}^{a}$ m/s^2, vertical acceleration component of the drone when it's landing, during acceleration part.
- $a_{z,l}^{b}$ m/s^2, vertical acceleration component of the drone when it's landing, during braking part.
- $d(i,j)$ m, distance between node i and node j.
- $v_{xy}^{max_r}$ m/s, maximum speed module when returning to base.
- $a_{xy}^{max_r}$ m/s^2, maximum acceleration module when returning to base.

- z_{target} m, constant vertical distance between the take-off zone and the inspection points.
- $\boldsymbol{g} = [0, 0, -9.80665]$ m/s^2, gravity acceleration vector.
- ω^{max} rad/s, angular velocity during turnings. It remains constant.
- m kg, constant payload. This value is senseless for us because we don't have a load.
- ω_{xy} m/s, horizontal proyection of wind speed.

A.1 Takeoffs

Takeoff kinematics can be modelled using equations for uniform and accelerated rectilinear motion. Suppose the drone is on the ground and we want to elevate it z_{target} meters. It firstly starts the vertical motion by accelerating at $a^a_{z,t}$, until it reaches v^{max}_z and begins to move uniformly. At the end it brakes at $a^b_{z,t}$ and arrives at z_{target} with $v_z = 0$. In Figure A.1 we can observe the behaviour of z components for acceleration, speed and position vectors of a drone during a takeoff with inputs: $z_{target} = 20\ m, a^a_{z,t} = 10\ m/s^2, a^b_{z,t} = -10\ m/s^2, v^{max}_z = 4.5\ m/s$.

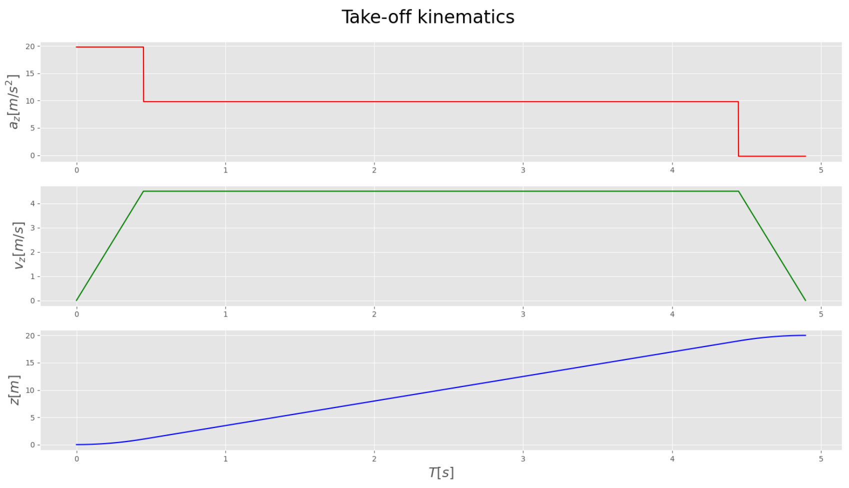

Figure A.1: Drone kinematics during a takeoff. The magnitudes that don't appear are supposed to remain constant.

A.2 Landings

Suppose the drone is on the air and we want it to land from z_{start} meters. It firstly starts the motion by descending at $a^a_{z,l}$, until it reaches v_z^{-max} and begins to move uniformly. At the end it brakes at $a^b_{z,l}$ and arrives to ground with $v_z = 0$. In Figure A.2 we can observe the behaviour of z components for acceleration, speed and position vectors of a drone during a landing with inputs: $z_{target} = 20\ m, a^a_{z,l} = 8\ m/s^2, a^b_{z,l} = -8\ m/s^2, v_z^{min} = 3.5\ m/s$.

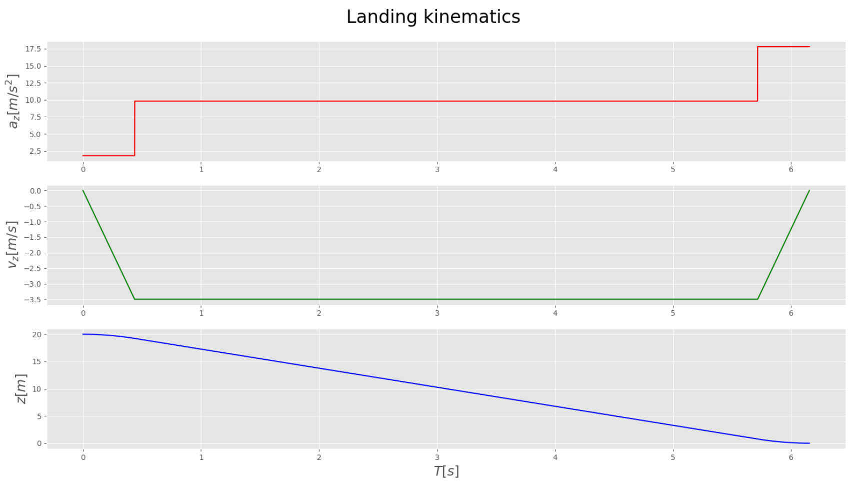

Figure A.2: Drone kinematics during a landing. The magnitudes that don't appear are supposed to remain constant.

A.3 Horizontal displacements

Suppose the drone is on the air and we want it to travel in straight line between two inspection nodes, i and j, or more generally, between two locations with coordinates $[x_0, y_0, z_0]$ and $[x_1, y_1, z_1]$. At first the drone is hovering at $(x, y, z) = (x_0, y_0, z_0)$. We suppose it is already heading its target. Then it accelerates at $a^a_{xy,h}$ until reach v_{xy}^{max} and performs a uniform motion. Before it arrives at $(x, y, z) = (x_1, y_1, z_1)$ it performs the braking part and decelerates at $a^b_{xy,h}$. At target, $v_{xy} = 0$. During this displacement, the z components of acceleration, speed and coordinate remain constant. In Figure A.3 we can observe the behaviour of x components for acceleration, speed and position vectors of a drone during a horizontal displacement between locations $(0, 0, 20)\ m$ and $(10, 10, 20)\ m$ with inputs: $v_{xy}^{max} = 4\ m/s$, $a_{xy}^{max} = 10\ m/s^2$.

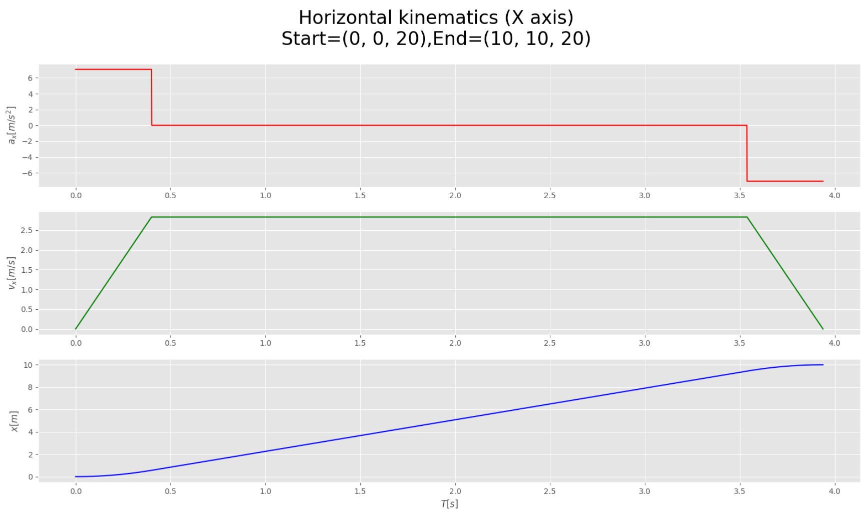

Figure A.3: Drone kinematics for x components during a horizontal displacement. The kinematics for y components is pretty similar and the z components don't vary.

A.4 Turnings

Turnings are modelled as simple uniform rotation at a given constant angular velocity ω^{max}. We suppose that the drone always turns at the same velocity, it doesn't perform neither acceleration nor braking part. We also consider that it always choose the orientation of turning with the smallest arc length, respect to the initial and target angles. The angles are defined by convention or using the director vector associated with the displacement from i to j. For example, we suppose that the drone starts at base station with an initial angle of $\pi/2\ rad$ respect the OX axis. The kinematics of this part is computed using simple equations: we know the initial and target angle and the angular velocity, so we know how much time the process lasts.

A.5 Weights and costs computation

The process is then simple, knowing the kinematics we know how much the action last and which are the components for accelerations and velocities. Then we can use the firsts to compute inspection time costs: t_{ij} is the last value of the time array associated with kinematics of going from i to j. On the other hand, the acceleration and velocities components are used in the standard regression model showed at Chapter 2 to obtain the w_{ij} values, i.e, the energy consumption weights of going from i to j. Finally, in order to evaluate the standard regression model, we need some beta coefficients values for that. In [4] and [10] the authors present 3 vectors standing for beta coefficients obtained from 3 different drone models. In this work we have been working with these values, specifically with 3DR-Solo ones, in order to evaluate the model and obtain realistic weights. This model was chosen because weights behave qualitatively pretty similar to the energy consumption behaviour of the drone used at real CSP inspections by the company.

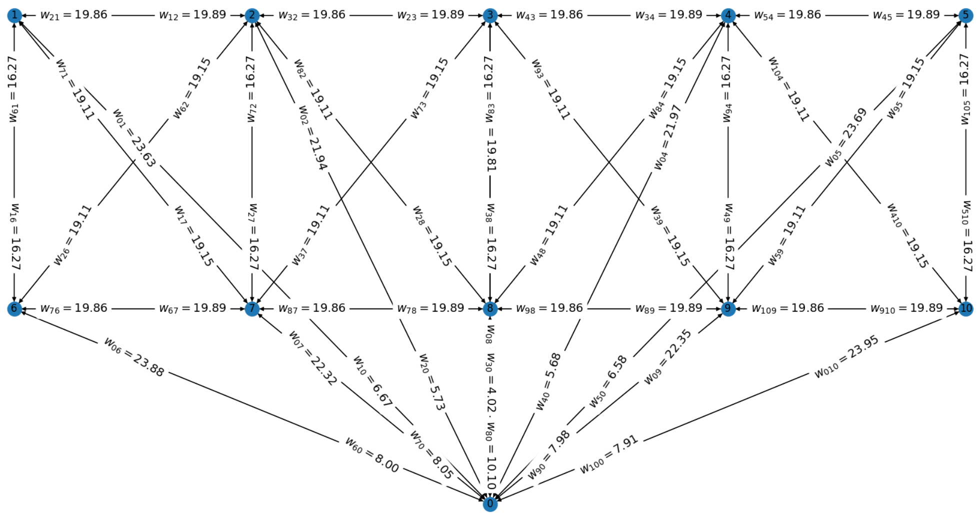

Figure A.4: *Realistic energy consumption weights* computed for an inspection graph with 10 nodes. Wind horizontal vector: $\vec{\omega}_{xy} = (1, 0)$. Beta coefficients model: 3DR-Solo

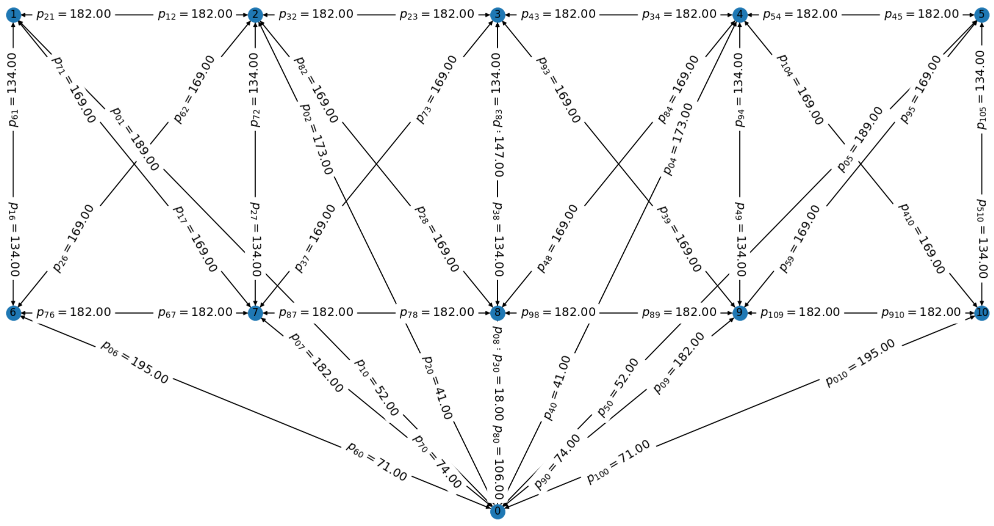

Figure A.5: *Realistic inspection time costs* computed for an inspection graph with 10 nodes

Bibliography

[1] H.L. Zhang, J. Baeyens, J. Degrève, G. Cacères. (2013). Concentrated solar power plants: Review and design methodology. Renewable and Sustainable Energy Reviews, 22, 2013, Pages 466-481

[2] C. E. Miller, A. W. Tucker, and R. A. Zemlin. 1960. Integer Programming Formulation of Traveling Salesman Problems. J. ACM 7, 4 (Oct. 1960), 326–329. DOI:https://doi.org/10.1145/321043.321046

[3] Borcinova, Zuzana. (2017). Two models of the capacitated vehicle routing problem. Croatian Operational Research Review. 8. 463-469. 10.17535/crorr.2017.0029.

[4] Tseng, C. M., Chau, C. K., Elbassioni, K. M., & Khonji, M. (2017). Flight tour planning with recharging optimization for battery-operated autonomous drones. CoRR, abs/1703.10049.

[5] Campbell, J. F., Corberán, Á., Plana, I., & Sanchis, J. M. (2018). Drone arc routing problems. Networks, 72(4), 543-559.

[6] Mesas-Carrascosa, F. J., Verdú Santano, D., Pérez Porras, F., Meroño-Larriva, J. E., & García-Ferrer, A. (2017). The development of an open hardware and software system onboard unmanned aerial vehicles to monitor concentrated solar power plants. Sensors, 17(6), 1329.

[7] Jose, K., & Pratihar, D. K. (2016). Task allocation and collision-free path planning of centralized multi-robots system for industrial plant inspection using heuristic methods. Robotics and Autonomous Systems, 80, 34-42.

[8] Amarat, S. B., & Zong, P. (2019). 3D path planning, routing algorithms and routing protocols for unmanned air vehicles: a review. Aircraft Engineering and Aerospace Technology.

[9] Modares, J., Ghanei, F., Mastronarde, N., & Dantu, K. (2017). Ub-anc planner: Energy efficient coverage path planning with multiple drones. In 2017 IEEE international conference on robotics and automation (ICRA) (pp. 6182-6189). IEEE.

[10] Tseng, C. M., Chau, C. K., Elbassioni, K., & Khonji, M. (2017). Autonomous recharging and flight mission planning for battery-operated autonomous drones. arXiv preprint arXiv:1703.10049.

[11] Martello, Silvano and Toth, Paolo. (1990) Knapsack problems: algorithms and computer implementations. New York, NY, USA: John Wiley & Sons, Inc.

[12] Johnson, D. B. (1975). Finding all the elementary circuits of a directed graph. SIAM Journal on Computing, 4(1), 77-84.

[13] KNUTH, D. E. (2000). Dancing links. Millennial Perspectives in Computer Science. Houndmills., 187.

[14] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2001). Introduction to Algorithms Second edition Cambridge.

[15] Dheeraj Gupta, (28 Jul, 2021). *GeeksforGeeks.* Bin Packing Problem (Minimize number of used Bins). Recovered on 20th August 2021 from https://www.geeksforgeeks.org/bin-packing-problem-minimize-number-of-used-bins/.

[16] Morales, J., Valverde, J.S., Virtualmechanics, S.L (2021). Drone inspection at CSP plants interviews [Personal interviews].