Comparing different formulations to solve the Traveling Salesman Problem on a Quantum Computer

Bachelorarbeit im Fach Physik

von

Nikolaos Tsakalias
(Matrikelnummer: 408431)

vorgelegt der

Fakultät für Mathematik, Informatik und Naturwissenschaften
der Rheinisch-Westfälischen Technischen Hochschule Aachen

Erstgutachter: Prof. Kristel Michielsen

Zweitgutachter: Prof. Achim Stahl

Aachen, 08.10.2023

Comparing different formulations to solve the Traveling Salesman Problem on a Quantum Computer

Nikolaos Tsakalias

October 9, 2023

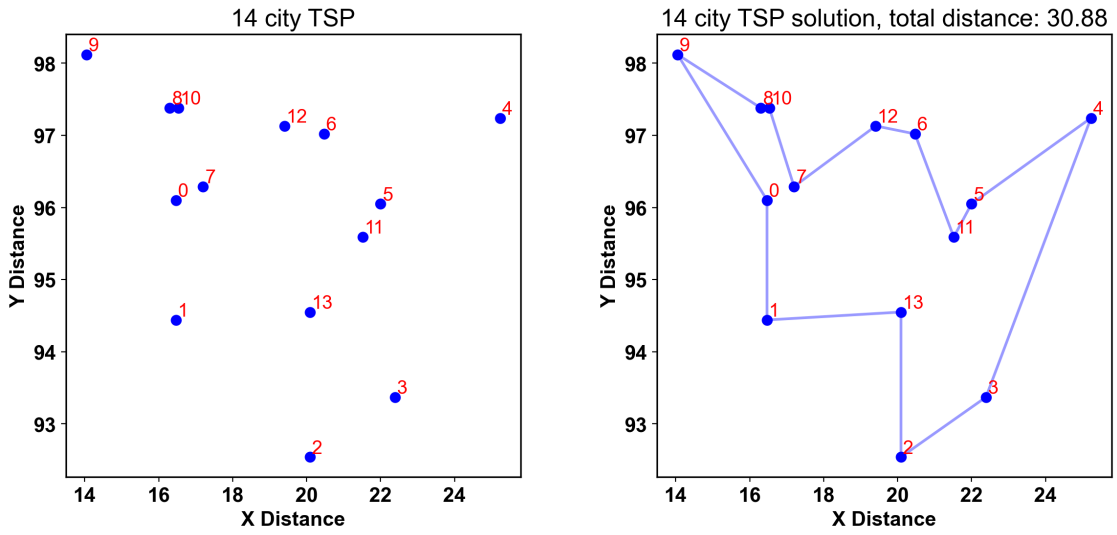
Abstract

Quantum Computing is an emerging technology. This means that many of the details of this field are still unexplored. For example, when trying to solve a problem on a classical computer, there often exists a well-established, preferable method of doing it. This is often not the case when using a quantum computer. This Bachelor's Thesis will focus on the Traveling Salesman Problem, a famous optimization problem with a lot of real applications, ranging from efficient package delivery to genome map assembly and guiding industrial machines. Different ways of encoding and solving the problem on a quantum computer will be compared. This is done in hope of determining whether a preferable encoding exists for the TSP. The result could also shed some light on the encoding of other optimization problems.

Contents

1 Introduction

The Traveling Salesman Problem (TSP) is a mathematical optimisation problem which can often be difficult to solve. It is typically solved using computers. In this problem, we are given a set of cities and the distances between them, and we are tasked with finding the shortest path that passes through all cities, returning to the start. The Traveling Salesman Problem has several real applications, such as planning the routes of vehicles, pointing telescopes and satellites, or even manufacturing microchips in an efficient manner [1]. In Fig. (1), we can see a typical illustration of a TSP, where the cities are represented as numbered points on a Cartesian coordinate system, with the goal being to connect them, using straight lines, to form the shortest possible tour.



(a) An example of an unsolved TSP with 14 cities, labeled in red.

(b) A TSP solution

Figure (1): Two plots showing an example of a TSP, as well as its solution. The city coordinates correspond to those given by the burma14 problem [2], which belongs to a set of standard TSP problems used for benchmarking purposes. The coordinates in this problem are originally meant to correspond to latitudes and longitudes, however here they were simply treated as Cartesian coordinates.

In order to demonstrate how the TSP becomes more difficult to solve when dealing with larger problem sizes, we decided to see how a human performs when faced with TSP instances of 5 to 25 cities. For each problem size, we let a human solve the problem intuitively (rather than using some mathematical algorithm), within a time frame of one minute. The human was shown an unsolved problem, like the one shown in Fig. (1). He was then tasked with ordering the cities in such a way, as to achieve an optimal tour. He solved 5 randomly generated problems for each number of cities. We can see the result, showing the human's solution's total cost, in Fig. (2). We notice that for problem sizes up to roughly 10 cities, a human is consistently able to find an optimal, or very close to optimal,

solution. After that point, the difference from the optimal cost begins to rise. For problems of 15 or more cities, a 5-10% difference can be expected. We thought it was interesting to see how well a human performs when solving relatively large TSP instances, staying within just a few percentage points from the optimal solution. This is surprising, because the number of possible solutions to a TSP is really large (there are $n!$ solutions to a TSP of size n), and yet humans can very easily narrow this large set down to a small fraction of all possible solutions, that are quite close to the optimum. In fact, it has been shown that humans can even outperform some heuristic algorithms for solving the TSP [3][4].

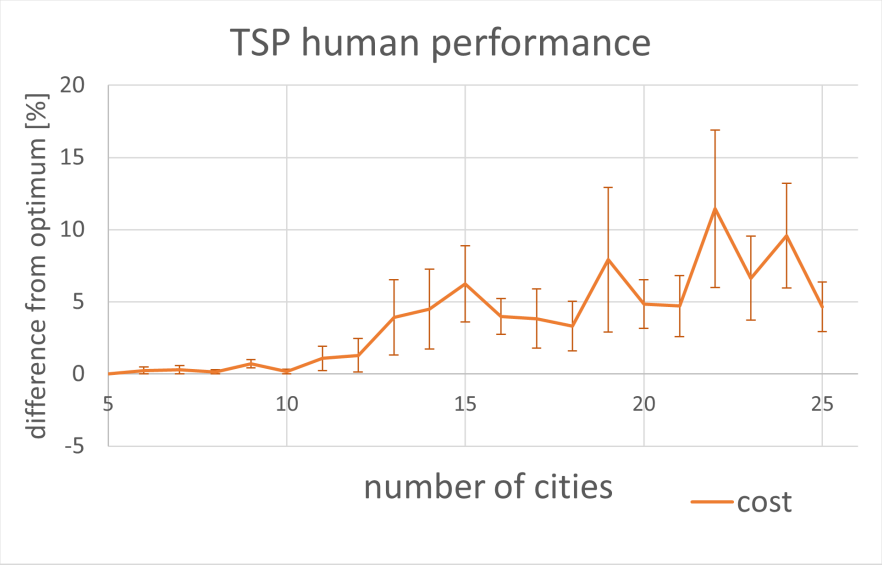


Figure (2): The solution quality given by a human when solving TSP instances of 5 to 25 cities. It is shown as a percentage difference from the optimal solution cost. The error bars represent the standard deviation of the mean from 5 attempts.

There are countless algorithms, as well as heuristic approximation methods of solving the TSP on a classical computer. Some examples might include brute force methods [5], greedy algorithms such as the nearest neighbor algorithm [6], or randomized improvement methods such as the ant colony optimization [7] and genetic algorithms [8]. In this thesis, we compare some different formulations for solving the TSP on a quantum computer.

Quantum computers are different to classical computers, in that their smallest unit of information is the qubit instead of the classical computer's bit. A bit, usually represented as a transistor being on or off, can take two different states, called 0 and 1. A qubit, most often realised by some property of a subatomic particle or atom, can exist in a superposition of the two states. When measured, it has a certain probability of taking either one of the two values, which will also set the qubit's state, according to the postulates of quantum mechanics.

While there are not any problems quantum computers can solve that classical computers can not, they are known to theoretically be able to solve certain problems much faster [9]. In this Bachelor's Thesis, we try two different formulations of solving the Traveling Salesman Problem on a quantum computer, and we will be seeing how these two methods compare against each other, and which of them works better on the classical vs. the quantum computer.

Specifically, we will be using a particular type of quantum computer, called quantum annealer. Quantum annealers are a type of quantum computer that is specialised in solving optimization problems. Starting with a superposition of all possible states, the system is then allowed to evolve, and it tries to reach its ground state. We can take advantage of this behavior by encoding the problem we are trying to solve in such a way, that a lower energy corresponds to a "better" solution. In this way, quantum annealers can be used to solve optimization problems.

In order for a problem to be solved by a quantum annealer, it first needs to be encoded into a mathematical expression called Quantum Unconstrained Binary Optimization (QUBO), the value of which the quantum annealer will then try to minimize.

The QUBO [10] is a general optimisation problem that can be solved by a quantum annealer (after

being converted to an Ising Problem [11][12], which is relatively simple, as it simply requires a change of variables.). Thus, if we find a way to represent the TSP as a QUBO, we can then submit it to the quantum annealer to find a solution.

To create this QUBO, we need to find a way to encode our problem's objective function, i.e. what we are trying to minimize, as well as our problem's constraints, i.e. the restrictions that have to be followed to ensure a valid solution. This is possible by structuring our QUBO expression in a clever way, so that a decrease in our objective function causes the QUBO's value to decrease, whereas a constraint not being met causes a "penalty", increasing the value of the QUBO.

In order, however, to encode the problem into a QUBO as we just described, there is a very important step that needs to be addressed first: formulating our problem mathematically. Namely, we must decide how many (and what kind of) variables are needed in order to describe our problem, as well as what these variables represent. There are often many ways to do this, which correspond to different ways of thinking about the problem.

A common way of solving the TSP on classical computers is called the DFJ Formulation (Dantzig-Fulkerson-Johnson), in which we think of TSP solutions in terms of what connections are made between cities. We assign a binary variable to every possible path between two cities, and its value, 1 or 0 determines whether the two cities are connected. There are $\frac{n(n-1)}{2}$ such variables in the case of a symmetric problem. However, a complication arises: as we will see, the optimal solutions will often consist of several disconnected loops, which makes them invalid. To combat this, we try solving the problem again, after adding restrictions that forbid these specific loops from forming. We will repeat this process until a valid solution is reached.

The other formulation we will be testing is called the Time formulation. In this formulation, we think of our problem as having to arrange n cities into n time slots, to figure out in which order the cities are visited. To do this, we will need to create binary variables corresponding to each city being visited at each time. There will be n^2 such variables. Unlike the DFJ formulation, in the Time formulation it is not possible for a solution to include disconnected loops, and thus we can get a valid solution after only solving the problem once.

If we compare two formulations we just discussed, we can easily see that the DFJ formulation has less variables than the Time formulation (about half). This makes the problem significantly easier to solve. However, this comes at a cost, as the DFJ formulation implies that we may need to solve the problem multiple times in case subtours appear, which is not the case when using the Time formulation. Additionally, in operations research terms, the DFJ formulation is a "linear programming problem", as opposed to a "quadratic programming" one (such as the Time formulation), which means classical computers can solve it more efficiently. The reason for this is that the total length of the tour can be mathematically described in a far more simple, linear mathematical expression in the DFJ formulation, whereas in the Time formulation, a longer, more complex expression is required.

On a classical computer, it is well known that the DFJ formulation works quite well, being able to solve large TSPs in relatively short time. On a quantum computer, however, it is not really known which of the two methods can provide a better solution in a given TSP problem, which is the question that this Bachelor's Thesis aims to answer.

2 Theoretical Background

In this section we will introduce some concepts and mathematical relations that we will be using in the later sections.

2.1 Quantum Annealing

A quantum annealer is a specialized type of quantum computer. It is designed to utilize the principles of quantum mechanics in order to solve optimization problems (such as the TSP). The basic idea behind quantum annealing is inspired by simulated annealing, a classical optimization technique. Simulated annealing mimics the process of slowly cooling a material to reach its lowest energy state, which corresponds to the optimal solution of an optimization problem. In quantum annealing, this concept is taken a step further using the principles of quantum superposition and entanglement. Quantum annealers use qubits, rather than bits, as their basic building blocks. Qubits can exist in a superposition of both binary states (0 and 1) simultaneously. The quantum annealers that will be used in this

Bachelor's Thesis, which are made by D-Wave[1], use the magnetic field generated by small circulating superconducting current loops as their qubits [13]. By applying an external magnetic field, referred to as bias, the probability of a qubit ending up in state 0 or 1 can be adjusted. This is what allows us to implement the specifics of our problem, i.e. creating a landscape in which solutions we consider 'good' tend to have the lowest energy, and are thus more likely to be reached. Another crucial part of quantum annealing is that qubits can be coupled, meaning that the state of one of them will affect the state of the other. This is done through quantum entanglement. This allows for the problem's variables to be dependant on each other, which, as we will see, is necessary for implementing constraints in our problem.

In order to solve any given optimization problem, we need to define the problem's objective function, whose purpose is to assign an energy value to each possible configuration of the problem in such a way, that better solutions will correspond to lower energies. The ultimate goal is to find the configuration with the lowest energy, which should correspond to the optimal solution. In quantum annealing, the problem's objective function is mapped to a mathematical expression called the Hamiltonian, which can be seen in Eq. (1):

$$H_{ising} = -\frac{A(s)}{2}\left(\sum_i \hat{\sigma}_x^{(i)}\right) + \frac{B(s)}{2}\left(\sum_i h_i \hat{\sigma}_z^{(i)} + \sum_{i>j} J_{i,j}\hat{\sigma}_z^{(i)}\hat{\sigma}_z^{(j)}\right), \tag{1}$$

where the symbols $\hat{\sigma}_{z/x}^{(i)}$ denote the Pauli matrices, h_i denotes the bias on qubit i, and $J_{i,j}$ denotes the coupling strength between qubits i and j. We can see that the Hamiltonian consists of two terms added together. The first of these terms corresponds to the initial (or driver) Hamiltonian, and the second to the final (or problem) Hamiltonian. The values $A(s)$ and $B(s)$ control the amplitudes of both Hamiltonian terms. The parameter s is an abstract value that increases from 0 to 1 as the time of the annealing process progresses. Over time, the value of $A(s)$ will go from really large to near-zero, whereas $B(s)$ will start at a really small value and get quite large by the end of the annealing process. The annealing process starts with the qubits prepared in a simple initial state that can be easily manipulated. The driver Hamiltonian guides the qubits' evolution towards the ground state of the problem Hamiltonian. As time progresses, the driver Hamiltonian's influence is gradually decreased, and the problem Hamiltonian's influence is increased. This slow transition from the initial state to the ground state is known as the annealing schedule. An example of an annealing schedule can be seen in Fig. (3)[13], where the values were provided by D-Wave [14].

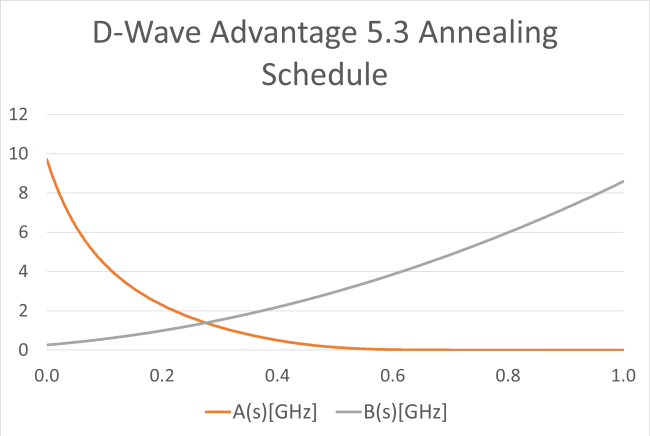


Figure (3): The annealing schedule, which describes the evolution of the factors A and B, shown as a function of s.

One of the crucial quantum phenomena exploited during the annealing process is quantum tunneling. Quantum tunneling allows qubits to overcome energy barriers and explore different configurations even if those configurations have higher energy initially, as can be seen in Fig. (4). This enables quan-

[1]See Section (3)

tum annealers to search a wider solution space efficiently. A simple illustration of this concept can be seen in Fig. (4).

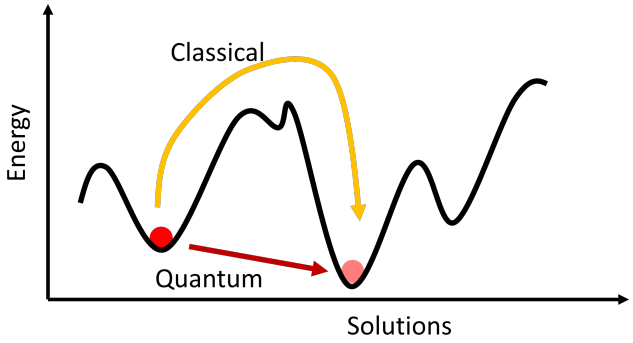


Figure (4): A visualisation of the advantage of quantum annealers in comparison to classical annealing processes.

At the end of the annealing process, the qubits are measured. By repeating the annealing process multiple times and measuring the qubits, the quantum annealer aims to find the configuration with the lowest energy, which corresponds to the optimal solution of the original optimization problem. It is important to note that quantum annealers are specifically designed to solve certain types of optimization problems. While they can solve problems that involve finding the global minimum of an energy landscape, such as the TSP or other combinatorial optimization problems, they are not designed to solve run any quantum algorithm, like Shor's algorithm for factoring large numbers or Grover's algorithm for searching unsorted databases, which are suited for universal gate-based quantum computers.

2.2 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is an optimisation problem which aims to find the shortest path that travels through a set of cities in a loop. As a given, it includes the distances between each pair of cities, which is often represented as a matrix. The distance between cities is often referred to as cost, as it can also represent other things such as travel cost or travel time. It is denoted as c_{ij}, which corresponds to the travel cost from city i to city j.

We usually deal with the symmetric version of the TSP. This is a special case of the problem where the cost c_{ij} from city i to city j is the same as the cost c_{ji} from city j to city i, with this being true for all cities i, j. Thus, in this case, our cost matrix will be symmetric, and we only need the upper triangular part to sufficiently describe the problem. While the symmetric case is a simpler special case of the problem, it is accurate enough for most real applications. In this Bachelor's Thesis, we will always be dealing with the symmetric version of the TSP.

The TSP can be described geometrically. Most commonly, the cities are represented as points on a two-dimensional coordinate system, with the straight-line distances between each pair of cities corresponding to their distance. This representation is, of course, necessarily a symmetric case of the TSP. When a solution is found, it can be shown visually by 'connecting the dots'. The total distance of the tour is of course the sum of the individual distances that are traveled within the route. In this thesis we will also show a way to represent a more 'realistic' case of the TSP which makes use of the geographical coordinates of real cities, with the cost corresponding to the driving time between cities, as estimated by Google Maps.

2.2.1 DFJ Method

One method of representing our problem is the DFJ Formulation [15], which is named after the initials of its creators. In the DFJ Formulation, we represent our problem with a number of binary variables x_{ij}, which correspond to the paths between cities i and j. If $x_{ij} = 1$, then the corresponding path is 'active', which means that we travel through it at some point in our final tour. If $x_{ij} = 0$, this

means that the path is 'inactive', meaning it is not being used in our tour. As we are dealing with the symmetric case of the TSP, we have $\frac{n(n-1)}{2}$ such variables, with n being the total number of cities. If we have found a solution to our problem, it can be described by the value, 0 or 1, of each of these variables. The total cost of our tour is simply given by the expression:

$$C = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} c_{ij} x_{ij} \,. \tag{2}$$

This is called the cost function of our problem, and it is what we are trying to minimize. It is easy to realise that, in any given problem of n cities, a tour will have exactly n active paths. More specifically, each city must be connected to exactly two other cities. A solution where this does not hold is not a valid tour. This can be expressed mathematically as a set of equations, called 'constraints':

$$\sum_{j=0}^{i-1} x_{ji} + \sum_{j=i+1}^{n-1} x_{ij} = 2, \forall i \in [0, n) \,. \tag{3}$$

This type of constraint is called an equality constraint, as it contains an equality which must hold in our problem. There are n such constraints, which must necessarily be fulfilled for a solution to be valid. However, these constraints are not sufficient to ensure a valid solution, for the simple reason that a solution that contains multiple closed loops, and not just one, also fulfills these constraints Therefore, we have to create additional constraints to rule out this type of solution. Mathematically, we can define a constraint that forbids a set of cities Q from forming a loop by

$$\sum_{i \in Q} \sum_{j > i, j \in Q} x_{ij} \leq |Q| - 1 \,, \tag{4}$$

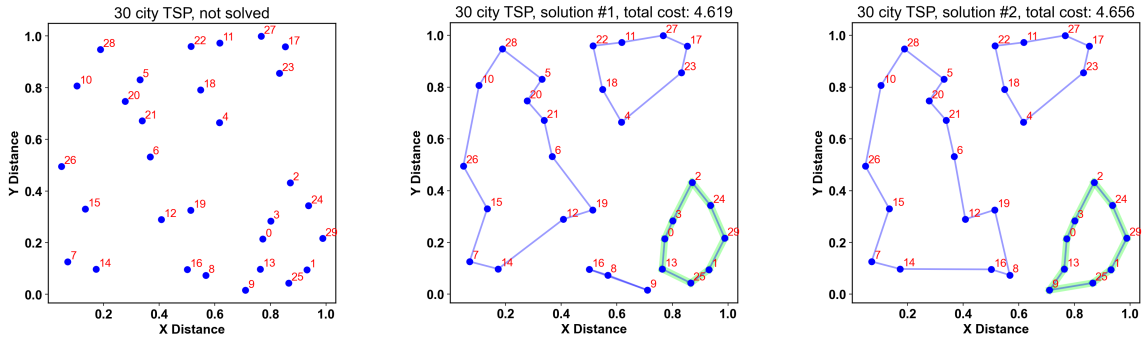
where $|Q|$ indicates the size of this set of cities. This constraint can simply be understood as follows: If the cities in this set were to form a loop, that loop would have to contain exactly $|Q|$ active paths, connecting cities within Q. This is exactly what constraint (4) forbids from happening, as it states that the total number of active paths within the set must be at most $|Q| - 1$. This type of constraint is called an inequality constraint.

The simplest way to forbid subtours from forming would be to add inequality constraints such as (4) for all possible subsets Q of cities in our problem. However, this would not be practical. As the total number of cities increases, the number of such subsets will increase extremely quickly. Specifically, for a problem with n cities, we will have to add for $2^{n/2} - 2n - 2$ subtour elimination constraints [16], which would quickly become infeasible, as this number grows exponentially.

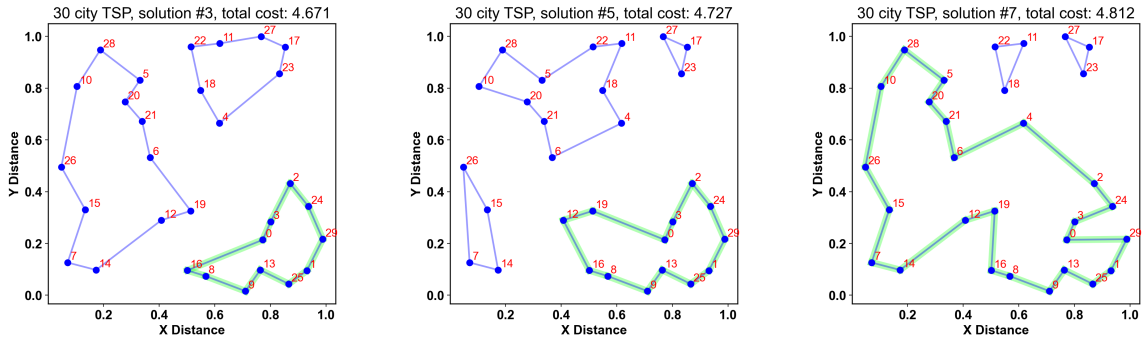
We can, however, add subtour constraints far more efficiently than described above. This is due to the fact that the vast majority of possible subsets Q of cities that could form a subtour, in practice never will, because it would give a bad solution.

Therefore, we can deal with the appearance of subtours by solving the TSP with the so-called 'finger-in-the-dike' [16] (or branch-and-cut) method. This method consists of repeatedly solving the problem and adding constraints each time, until a valid solution is reached. We first solve our problem with zero subtour constraints. If the solution found contains no subtours, we are done. However, if it does contain subtours, we can add a constraint forbidding (at least) one of these subtours and try solving the problem again. We will then get a different solution. The process continues in the same way, checking for subtours in the solution, adding the relevant constraints and solving again until a valid solution is reached.

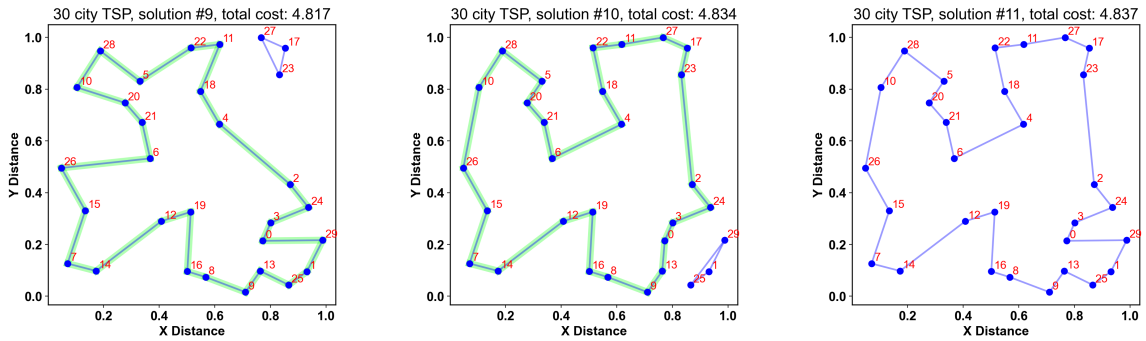
An example of the subtour elimination process followed in the DFJ formulation can be seen in Fig. (27). In this example, a 30 city TSP is being solved. Each time a solution is found, it is checked for subtours, and if subtours are present, one of them is selected (highlighted in green) and eliminated in the next run. We can see that the total cost of the solutions reached increases during the process. This happens because at the start, the lowest-cost solution is found, regardless of subtours. After the first subtour is eliminated, the second-lowest cost solution is found, and so forth, until a solution with no subtours emerges. In the example in Fig. (27), the problem needed to be solved 11 times until a solution with no subtours was reached.



(a) Unsolved 30 city TSP (b) The first solution (c) The second solution



(d) The third solution (e) The fifth solution (f) The seventh solution



(g) The ninth solution (h) The tenth solution (i) The eleventh solution

Figure (5): A 30 city TSP, solved with the DFJ formulation on DOcplex. In each iteration, the subtour that is chosen to be eliminated is highlighted in green.

2.2.2 Time Method

A different way the TSP can be mathematically described is the Time formulation, which is named this way because it utilizes variables that describe whether a certain city was visited at a certain time. Specifically, in this formulation, we use a set of binary variables x_{ij} to describe a solution to our problem, which refer to visiting city i at time j. If $x_{ij} = 1$, it means that city i was visited at time j. If $x_{ij} = 0$, it was not. In a problem with n cities,there will be n different times. Thus, since we will have one variable for each combination of city/time, there will be n^2 variables in total. We can slightly simplify the problem, if we assume without loss of generality that city 0 is visited at time 0. Therefore, we will have $x_{00} = 1$ and $x_{0j} = x_{j0} = 0$ for all $j < n$. Thus, we only have $(n-1)^2$ variables that need to be determined.

In this formulation of our problem, the cost function is a little bit more difficult to define, as we have no way of referring directly to the paths between cities. Nevertheless, we can define the cost

function in the following way:

$$C = \sum_{t=0}^{n-1} \sum_{i=0}^{n-1} \sum_{j \neq i} c_{ij} x_{it} x_{j,t+1} \,. \tag{5}$$

This expression works because the expression $x_{it}x_{j,t+1}$ within the sum is always 0 unless city i is visited at time t and city j is visited at time $t+1$, in which case it is equal to 1, and only then the travel cost between cities i and j is added to our total cost. It's important to note a small detail about what happens that when t reaches the value $n-1$, as our expression also contains a term $t+1$. In this case, we assume that t rolls back to 0. This results in finally adding the cost from the last city to the first.

To ensure that we find valid solutions, we have to add some constraints to our problem. The first type of constraint states that each city i has to be visited exactly once:

$$\sum_{t=0}^{n-1} x_{it} = 1, \forall i < n \,. \tag{6}$$

There will be n such constraints, one for each time. By themselves, however, these constraints are not enough to guarantee a valid solution. We also need to add a constraint that states that exactly one city is visited at a given time t:

$$\sum_{i=0}^{n-1} x_{it} = 1, \forall t < n \,. \tag{7}$$

There will also be n constraints of this type. We will therefore have $2n$ constraints in total. Of course, if we make the assumption that decreases the number of variables by 1, as described above, we will have $2n-2$ constraints in total.

Fortunately, with this formulation, there is no need to worry about subtours, like we did in the DFJ formulation. It is easy to understand why if we consider what the Time formulation represents. If we solve the TSP with the Time Formulation, our solution will simply correspond to assigning a time to each of our cities. This is equivalent to 'making a schedule', i.e. arranging all of our cities in a specific order in which they will be visited. Therefore, it will be impossible to end up at a solution that contains subtours, because such a solution would in fact be impossible to describe in this formulation.

2.3 QUBO

Quadratic unconstrained binary optimization, usually shortened as QUBO, is a type of optimization problem with binary variables. If we define a vector $\vec{x}$ that contains all of our binary variables,

$$\vec{x} = (x_0, x_1, ..., x_{n-1}) \,, \tag{8}$$

then with some matrix Q that is specific to our problem, we can define our QUBO function f_Q:

$$f_Q(\vec{x}) = \vec{x}^T Q \vec{x} = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} Q_{ij} x_i x_j \,. \tag{9}$$

This function is what we are trying to minimize. We have to find for what vector $\vec{x}$ (or respectively what values of x_i) the value of f_Q becomes minimal. The form of this function explains the *quadratic* part of the name QUBO, as each of our terms includes two (and no more) of our variables multiplied together.

2.3.1 How to create a QUBO

To create a QUBO for the TSP in either the DFJ or Time formulations, we have to encode the whole problem, including the cost function and the constraints, into a single mathematical expression of the form described in (9). Doing this can involve several steps, which we will describe below.

First of all, we need to define our cost function, and, if necessary, rearrange it into a form that is compatible with (9). For the Time Formulation, our cost function already has a form that is

compatible with QUBO, as described in (5). For the cost function in the DFJ case, we have to make a small modification to (2), which seemingly does not have the form of Eq. (9). However, this can simply be solved if we observe that $x_{ij} = x_{ij}^2$, as we are dealing with binary variables. Therefore we can write the cost function as:

$$C = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} c_{ij} x_{ij} = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} c_{ij} x_{ij} x_{ij} \,. \tag{10}$$

Next, we need to implement our equality constraints. Equality constraints, which are employed in both formulations, have, in both cases, the form:

$$\sum_{i=0}^{n-1} c_i x_i = k \,. \tag{11}$$

However, we need to somehow incorporate these constraints into our QUBO function, in such a way that the QUBO function takes a smaller value if the constraint is fulfilled. This can be done if we bring the constant k to the other side of the equation and square both sides. The constraint then takes the form:

$$\lambda_0 \left(\sum_{i=0}^{n-1} c_i x_i - k \right)^2 \,, \tag{12}$$

where λ_0 is called a Lagrange Factor. Eq. (12) will reach a minimum of 0 when the constraint is fulfilled. If we expand the square, also using $x_i = x_i x_i$, we can see that this expression has the form required to be added to our QUBO.

The Lagrange factor λ_0 is essentially a weighing factor, which, simply put, means that it describes how 'important' it is to us that this constraint is fulfilled, in comparison to the orther requirements of our problem. The larger λ_0 is, the more penalization we get if the term within the square in Eq. (12) is not 0. It should also be noted that this λ_0 will be the same for all of the equality constraints in our problem.

For very large λ_0, high priority will be given to the constraint being fulfilled. The value of the cost function will, in comparison, be relatively unimportant, and we will thus reach solutions that are valid (because all of the constraints will be fulfilled), but unoptimal (because the cost function will not be minimal). For values of λ_0 that are too small, however, we will observe the opposite: the satisfaction of our constraints becomes unimportant compared to the minimization of the cost function, which means that we will find low-cost solutions, which are, however, invalid.

Next, we also need a way to incorporate inequality constraints, such as those that appear in the DFJ formulation, into our QUBO. The process of doing this is quite similar to how we dealt with the equality constraints, though a bit more nuanced. Inequality constraints have the form:

$$\sum_{i=0}^{n-1} l_i x_i \leq k \,. \tag{13}$$

At this point, we will need to introduce a slack variable S, such that:

$$\sum_{i=0}^{n-1} l_i x_i + S - k = 0 \,. \tag{14}$$

The purpose of this variable is to ensure that the above expression will be able to reach a value of 0 when the constraint is fulfilled. Due to QUBO requiring binary variables, this slack variable needs to be decomposed into a number of binary variables s_i:

$$S = \sum_{m=0}^{N} 2^m s_m \tag{15}$$

For a value N that is sufficiently large. The value we used for N was:

$$N = \lceil log_2(k) \rceil + 1 \,, \tag{16}$$

with k as defined in Eq. (13). This value of N allows for the slack variable to be able to take any value that could be necessary to ensure the fulfilment of the constraint. From this point, we can proceed like we did for the case of the equality constraints, squaring and multiplying by a Lagrange factor λ_1, and we can express our constraint as:

$$\lambda_1 \left(\sum_{i=0}^{n-1} l_i x_i + \sum_{m=0}^{N} 2^m s_m - k \right)^2 . \tag{17}$$

A suitable value will have to be found for the weighting factor λ_1, so that the fulfillment of this constraint is not overvalued, dominating all of our problem's other requirements, or undervalued, resulting in invalid solutions.

For any case of the Traveling Salesman Problem, we can construct the corresponding QUBO in the way we have described up to this point. Besides the number of cities, n, the only thing that can vary between different problems are the costs c_{ij} to travel between cities. For different problems, the cost can have completely arbitrary units, such as meters, kilometers, hours of travel, etc.. This will also affect our choice of Lagrange factors. It therefore makes sense to normalize the costs in our problem in such a way, that they all lie between 0 and 1. A simple way of doing this would be to divide each cost by the maximal value of our cost matrix:

$$c'_{ij} = \frac{c_{ij}}{\max_{i,j}(c_{ij})} . \tag{18}$$

Another way of normalizing is to first subtract the smallest cost value, then dividing by the difference between maximum and the minimum of our cost function:

$$c'_{ij} = \frac{c_{ij} - \min_{i,j}(c_{ij})}{\max_{i,j}(c_{ij}) - \min_{i,j}(c_{ij})} . \tag{19}$$

This second way, although a bit less simple, will result in the normalized costs being distributed in the whole range of $[0, 1]$.

As we will see, it makes sense to express the Lagrange factors as a multiple of the average cost. This, in combination with the normalization of the cost matrix, will later allow us to find a relatively consistent way to estimate the necessary Lagrange factors for any given TSP.

2.3.2 QUBO Matrix

After converting our problem to a QUBO expression, we can then piece together the QUBO Matrix Q, first mentioned in Eq. (9). This is a necessary step when trying to solve the problem on a quantum annealer, as it is this matrix that we submit as our input. However, this step can also be useful in order to obtain a visual overview of the entirety of our problem.

This matrix will have dimensions $N \times N$, where N corresponds to the total number of variables in our problem, including any slack variables. Each entry can be filled by the coefficient of the product of the corresponding variables in our QUBO expression. For example, the coefficient Q_{ij} of the product $x_i x_j$ in our QUBO, where x_i and x_j represent our problem's ith and jth variable, will be placed in column i, row j. This matrix will be symmetric, due to the commutative property of multiplication. Therefore, the upper-right triangular half of the matrix will suffice to describe the problem.

From the size of this matrix, we can deduce the number of variables of our problem, while from its shape, it is possible to infer several things, such as the form of the applied constraints, the number of slack variables, roughly the size of the Lagrange factors chosen, and more.

Below, in Figures (6) and (7), we can see the form of this QUBO matrix in an example problem in each formulation.

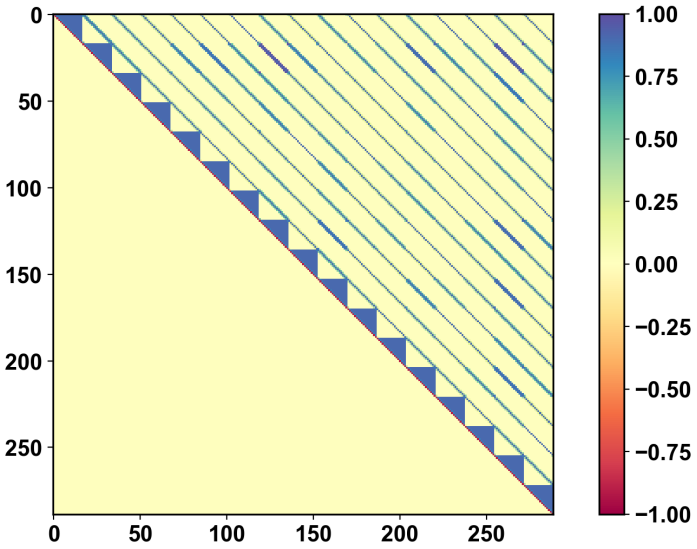


Figure (6): An example of a QUBO matrix for a TSP problem of 17 cities, encoded with the Time Formulation. The colorbar to the right aims to illustrate the rough value of the matrix's entries. The triangles along the diagonal, as well as the many parallel lines, are created due to the equality constraints added for each city. The 'smearing' that we see flat against the parallel lines is caused by the cost function.

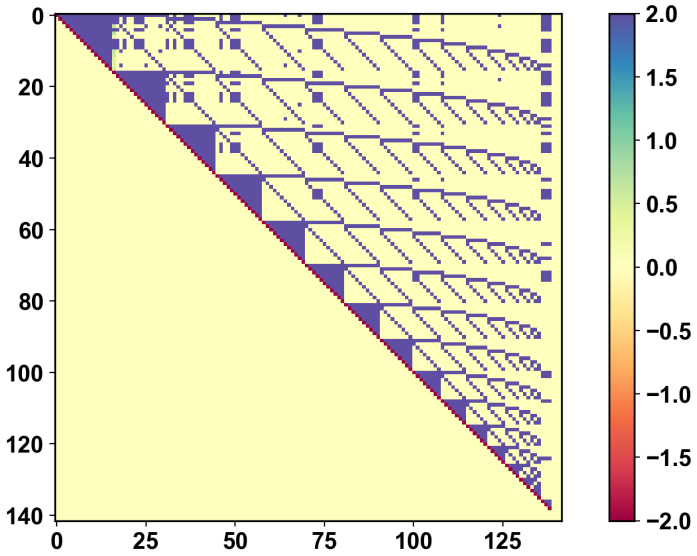


Figure (7): An example of a QUBO matrix for a TSP problem of 17 cities, encoded with the DFJ Formulation. The colorbar to the right aims to illustrate the rough value of the matrix's entries. The triangles along the diagonal correspond to the equality constraints added for each city. The triangles get smaller because of the way the variables are defined, namely x_{ij}, where $j > i$. The 'smearing' in some areas of our pattern is caused by the addition of subtour (inequality) constraints. The width of the mostly blank area to the right of the main pattern corresponds to the number of slack variables.

2.4 Other concepts

In this subsection, we will introduce some miscellaneous theoretical concepts that will be used later in this Bachelor's Thesis.

2.4.1 Hamming Distance

The Hamming distance is defined as a way to measure how different two binary data strings are to each other [17]. This will be useful later, as we will be looking at the Hamming distance between different TSP solutions as a measure of the level of similarity between them.

The Hamming Distance between two binary strings of equal length can be defined as the number of positions in those strings with different entries. By this definition, the Hamming distance between the strings 1101 and 1011 would be 2, as the two strings being compared have two differences, namely in the second and third place. The Hamming distance can also be normalized by dividing by our string's total length. This means that the normalized Hamming distance will always assume values between 0 and 1. In the above example, the normalized Hamming distance would be equal to $\frac{1}{2}$.

2.4.2 t-SNE visualization

T-distributed stochastic neighbor embedding (t-SNE) is a method used to visualize multi-dimensional data on a 2 or 3-dimensional coordinate system [18]. t-SNE arranges the data points in such a way, that similar data points will tend to be closer to each other, whereas data points that are more different will be further apart. This similarity between points can be determined by a chosen metric, such as Euclidean distance or Hamming distance. In this thesis, we will not go into the specifics of how t-SNE works, as it will not be used in any sort of quantitative analysis. It will only be used to visually represent some of our data, and to qualitatively discern some relations that may be present.

3 Setup

In this section, we will discuss the practical side of implementing TSP solution techniques on a quantum annealer. We will go into the programming required to implement the TSP on a classical computer, what modifications are needed to be able to submit the problems on quantum annealers, as well as what hardware we used for quantum annealing in the project.

3.1 Python Implementation

In order to perform the necessary testing of the methods discussed in section 2, we have to implement them in a programming language. This was done in Python, which is the programming language that is typically used when dealing with quantum computing.

We can easily generate a TSP problem with a given number of cities. The cities were defined as sets of two random numbers, x and y, between 0 and 1. This means that we could, if we wanted to, easily visualize our problem by drawing these cities as points on a Cartesian coordinate system. A cost matrix was created by calculating the straight-line distance between each pair of cities.

3.1.1 DOcplex

Our first task was to implement our models in a way for them to be solvable on a classical computer. This will not only allow for testing and refining our model to facilitate the implementation of the quantum annealer solution method later on, but it will also be useful to make comparisons between the solutions generated by the classical and quantum methods respectively.

For this, we used the DOcplex Python Modeling API [19]. DOcplex is a library that allows for the creation and solution of optimization models, such as the TSP, in Python. It allows for everything described in section 2, such as the creation of the necessary variables, the definition of the cost function and the addition of constraints. Note that it is not at all necessary to define our problem as a QUBO in order for it to be solved by DOcplex. That will, however, be needed for solving it on the quantum annealer.

3.1.2 D-Wave

To solve TSP problems on a quantum computer, we will be relying on the services offered by D-Wave [20]. D-Wave is a quantum computing company based in Canada, which, apart from designing

and selling quantum annealers, also offers users the ability to submit problems remotely for the quantum annealer to solve. Problems are submitted in the form of a QUBO, and the solutions we receive are lists of 1's and 0's that correspond to the values of our problem's binary variables. In this thesis, we will be using the D-Wave quantum annealer "JUPSI" located in Jülich, Germany. A photograph of the exterior of a quantum annealer from D-Wave can be seen in Fig. (8).

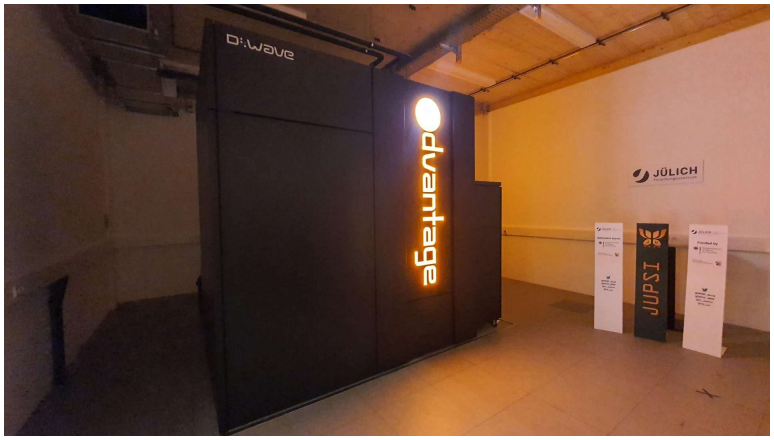


Figure (8): The D-Wave Advantage quantum annealer 'JUPSI', located in Forschungszentrum Jülich. The photograph was provided by Dennis Willsch.

3.1.2.1 D-Wave QPU

D-Wave's QPU (which stands for Quantum Processing Unit) is a quantum annealer by D-Wave, on which optimization problems can be solved. There exists a tool named D-Wave Embedding Composite, which handles the embedding of these problems to a set of qubits, in order for the quantum annealer to then solve them by letting the qubits relax to the ground state. The solutions are then read. The user is provided with a large number of solution samples. This way of solving also involves additional parameters, such as the chain strength and annealing time, which have to be adjusted in order to reach better solutions. When solving a problem with this method, it is important to note that only the QPU is being used, with no external assistance or corrections.

3.1.2.2 D-Wave Hybrid

D-Wave Hybrid is one of the ways offered by D-Wave to solve optimization problems. As the name suggests, it uses a hybrid approach, meaning it does not rely solely on quantum processes to generate the result. Instead, it also utilizes classical methods in order to split the problem into smaller parts, which it then solves and refines the solution. In the end, only one solution is chosen and returned. While this system is not purely a quantum annealer, it exhibits similar behavior. Additionally, it has the ability to solve larger problems than what is possible with purely the quantum method. In fact, the solutions provided by D-Wave Hybrid give an indication about how larger quantum annealers could solve the problem in the future. When comparing TSP solutions between our two different formulations, we decided to utilize D-Wave Hybrid, rather than the QPU, as this would greatly increase the range of problem sizes for which we can gather data.

3.1.2.3 Quantum Annealer Implementation of the TSP

In order to be able to solve TSP instances on D-Wave's quantum annealers, we needed to implement our problem accordingly.

For the Time formulation, this was as simple as generating our QUBO with the chosen Lagrange factors and then submitting it to D-Wave QPU or D-Wave Hybrid. We then get a number of solution samples (in the case of the QPU) or a single solution (in the case of Hybrid), which we can further analyze. A solution is considered as invalid if it violates the equality constraints described in Eq. (6) or in Eq. (7).

For the case of the DFJ formulation, the implementation was slightly less straightforward. We did it by initially creating a QUBO, assuming no inequality constraints. After submitting it to the quantum annealer (and, in the case of the QPU, identifying the best solution sample that was returned), we check our solution for subtours. If any are found, a new QUBO is created, which includes the new constraint that was just found. The problem is then repeatedly solved in this way. An important detail is that a solution is considered invalid not only when the equality constraints given in Eq. (3) are violated, but also if the same loop is identified for the second time, after having already been found in a previous run. This is an indication that the Lagrange factor responsible for the observance of the equality constraints is too low, as it means that the constraint that was added to avoid that particular subtour has not been effective. The solution is thus declared invalid, to prevent the problem from being repeatedly solved again and again, with the same subtour being found every time.

A different implementation, for example, would be to first solve the problem classically with DOcplex, keeping track of the subtour elimination constraints that are added in the process. Then, we would create a QUBO that includes these inequality constraints and submit that to the quantum annealer. With this method, there would be no need to solve the problem repeatedly on the quantum annealer. Instead, we would only solve it once, and then any solutions that violate the equality constraints or include any subtours would be considered invalid. We decided not to go for this method due to the consideration that the solutions provided by the quantum annealer would not be optimal, as opposed to those found by DOcplex. Thus, there is no guarantee that the subtours we would run into would necessarily correspond to the ones that we have found using DOcplex and added elimination constraints for. This would cause a large number of potential solutions found by the quantum annealer to be considered as invalid, when might in fact be simply suboptimal.

3.1.3 Google Maps

As part of this study, we also tried to create more realistic problem instances for the TSP, that could even be of practical use. We wanted to use real geographical data, and get a realistic forecast of the optimal route within a set of cities or addresses. To this end, we decided to utilize the Google Maps Python API [21], which allows, among other things, for the location of city coordinates and the calculation of travel time, in minutes, between two cities or addresses. We will then be using the travel time as our cost. We noticed that the travel time between two cities, as estimated by Google Maps, can differ slightly depending on the travel direction. Note that we made our cost matrix symmetric to ensure a symmetric TSP.

Google Maps offers the ability to find the travel time between two places by several modes of transportation, such as by car, by train or by bus. For our model, we made the assumption that our traveller is travelling by car. The reason that led us to this choice, instead of also including other modes of transportation, was to keep the model more accurate: the use of other transport modes will involve waiting for the departure of the bus, train or flight. The amount of time that the traveler needs to wait to travel, let's say, from city A to city B, will massively depend on the time of arrival in city A. The time of arrival at city A, of course, cannot be known, as it depends on the TSP solution. Therefore, any time prediction using these modes of transportation would be very inaccurate. A model that includes these methods would be possible, but more complicated than the regular TSP, and would in fact be more similar to the TDTSP (time-dependent TSP) [22]. The car, despite having some disadvantages compared to the other two, is much more reliable, as the traveler can start and stop at any time, which leads to a time prediction that is much closer to reality.

We also decided to include a way to plot our TSP solutions on a map, which was done by using the Basemap Matplotlib Toolkit Python Library [23]. This library can generate geographical maps within any given boundaries, and allows further data to be plotted on the map. On the maps, we drew our cities as points and our paths as squiggly lines that correspond to the real driving routes that the car would follow. The routes were determined by Google Maps.

Throughout this Bachelor's thesis, we will be using a specific set of cities when solving instances of the TSP. That set of cities consists of the 25 largest cities in the European Union by population within city limits. The cost matrix of this TSP can be seen in Tables (5) and (6). This is the cost matrix that will always be used unless we specify otherwise. This set of cities will not only be used for solving 25-city problems, but also problems of smaller size, by trimming the cost matrix from the top left. For example, when solving a 14-city TSP, we will be using a new cost matrix, consisting of rows and columns 0 to 13 of the original cost matrix. A solution of the full 25 city problem, plotted with

the real coordinates on a map, can be seen in Fig. (9). Some more solutions to realistic TSP instances can be seen in Fig. (10) and Fig. (11), as well as in the appendix, in Fig. (33) - Fig. (38).

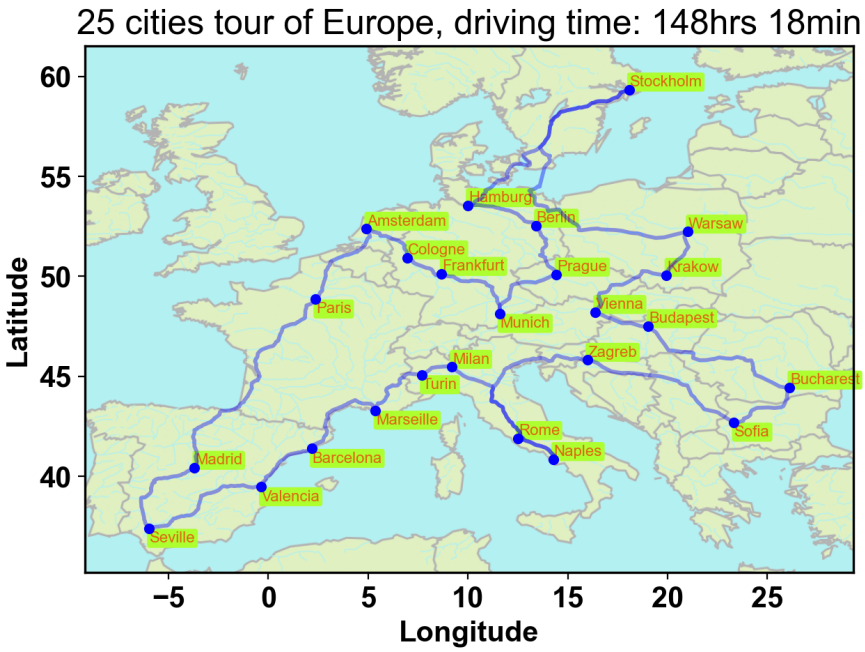


Figure (9): An example of a TSP solution with geographically accurate data. In this case, the cities used are the 25 largest in the European Union by population within city limits. In this example, the total driving time amounted to about 148 hours.

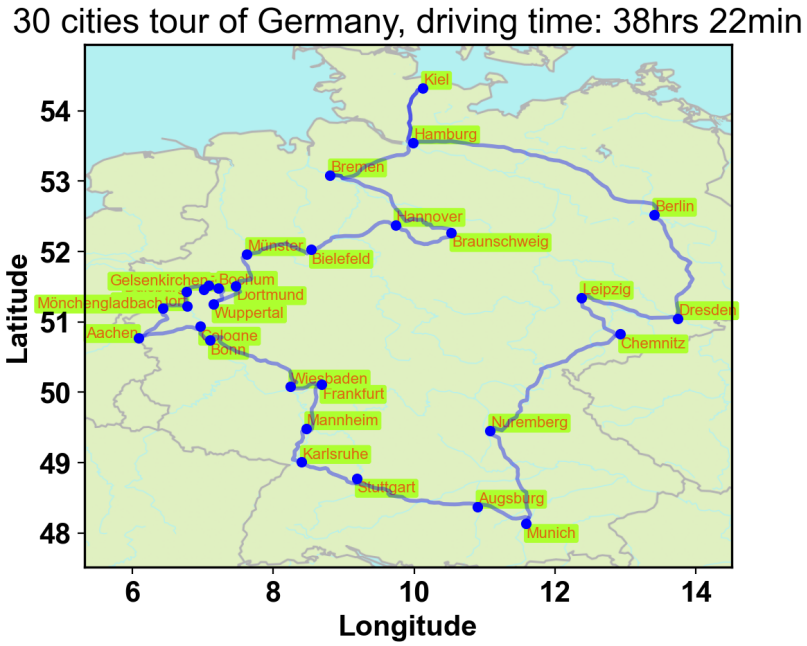


Figure (10): An example of a TSP solution with geographically accurate data. In this case, the cities used are the 30 largest in Germany by population. In this example, the total driving time amounted to about 38 hours.

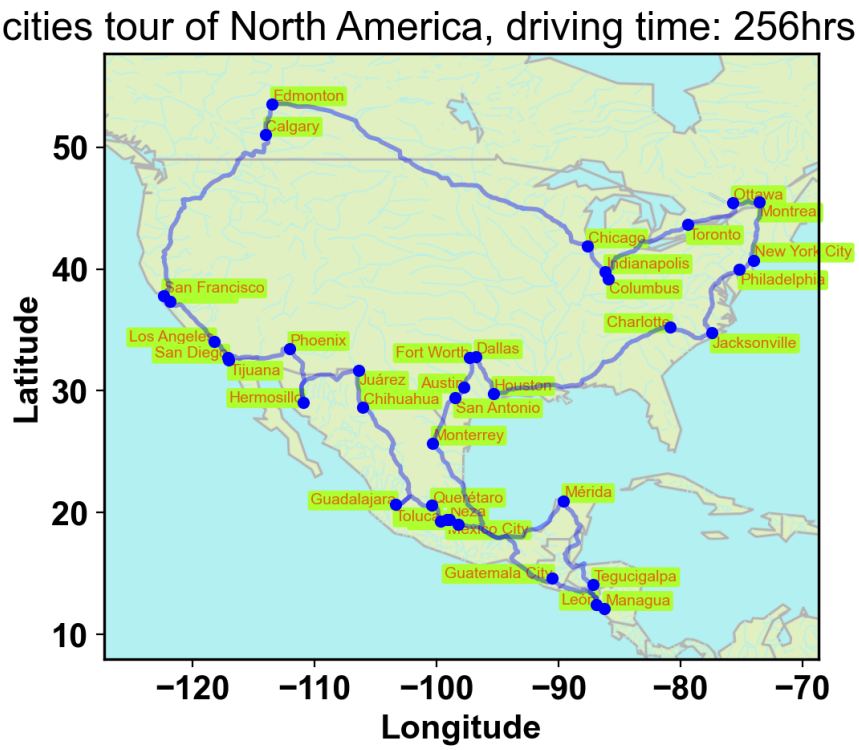


Figure (11): An example of a TSP solution with geographically accurate data. In this case, we are using a set of 38 of the largest cities in North America. In this example, the total driving time amounted to about 256 and a half hours.

3.1.4 Test: Time vs. DFJ on a Classical Computer

As a first test of our Python implementation of the TSP, we set out to compare the DFJ and Time Formulations when solved by classical computing, i.e. using DOcplex. The goal was to see if either method can have any advantage compared to the other, and if so, whether that still holds when solved with quantum, rather than classical, methods.

We began by testing different TSP instances solved using the DFJ method. While measuring the solution time, we also made sure to count how many times the solution needed to run again due to having found a subtour. We wanted to see to what extent this correlates with the total computation time.

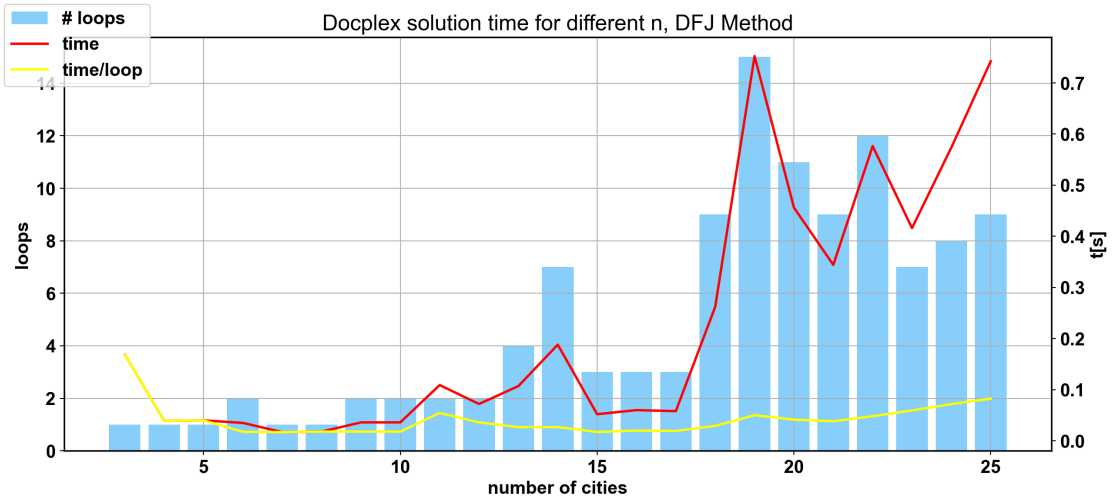


Figure (12): The solution time of TSPs with different numbers of cities, shown by the red line. The problems were solved using the DFJ formulation. The blue bar chart shows the number of repetitions that were required to reach the solution for each example. The yellow line shows the calculation time divided by the number of loops. The red line has an upward trajectory, although it bounces up and down a lot because some city numbers happen to require more repetitions. We can see a clear correlation between the number of repetitions and the total solution time. The yellow line fluctuates much less than the red, although it still has a slow increase over time.

Our next task was to repeat each measurement 20 times in order to get statistically more significant data of the computation time. We did this using the DFJ and Time formulations on DOcplex.

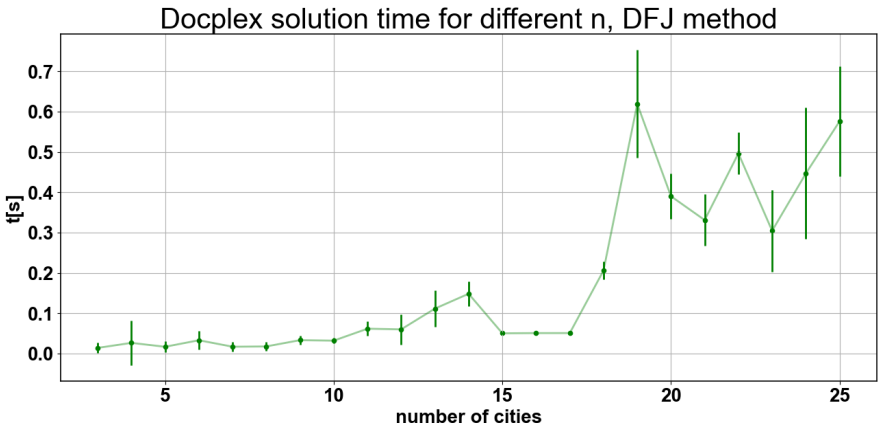


Figure (13): A statistical measurement of the DOcplex calculation times for the DFJ formulation, where we made 20 measurements for each number of cities, and used the standard deviation of the result as our uncertainty, shown by the error bars.

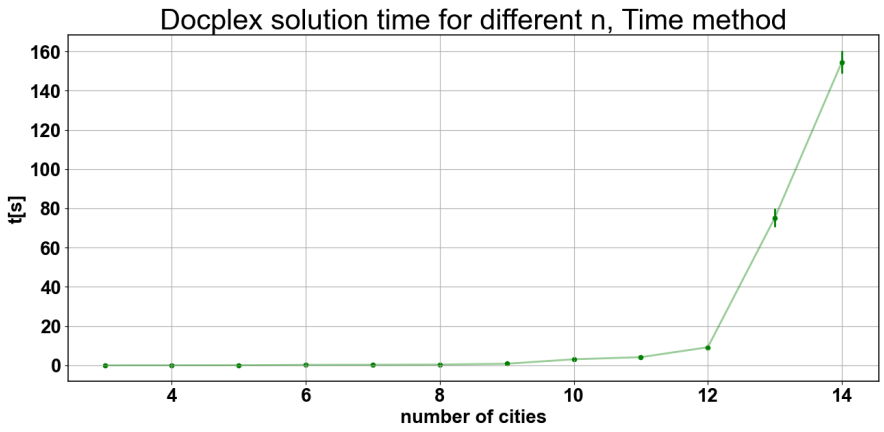


Figure (14): A plot of the solution time of TSPs with different numbers of cities, solved with the Time formulation, shown by the green line. We see a sharp increase in time needed as the city number grows beyond 12. We did not include any data beyond 14 cities as the solution times became extremely large. The curve resembles an exponential increase, as we see a quick rise in the order of magnitude of the solution time.

Comparing the two plots shown in Fig. (13) and Fig. (14), we first observe that the scale of the vertical axis is quite different: When using the DFJ formulation, solution times do not exceed 1 second, whereas with the Time formulation they can range from a few seconds to several minutes for cases of large number of cities. Secondly, we can see that with the DFJ method, larger numbers of cities can be solved, whereas with the Time method, we had to stop after 14 cities, as solution times grew very rapidly.

We will now try to model the data for both cases with an exponentially increasing function. To this end, we will take the natural logarithm of our data and then perform a linear regression. This will provide us with a coefficient to see how quickly the solution time increases.

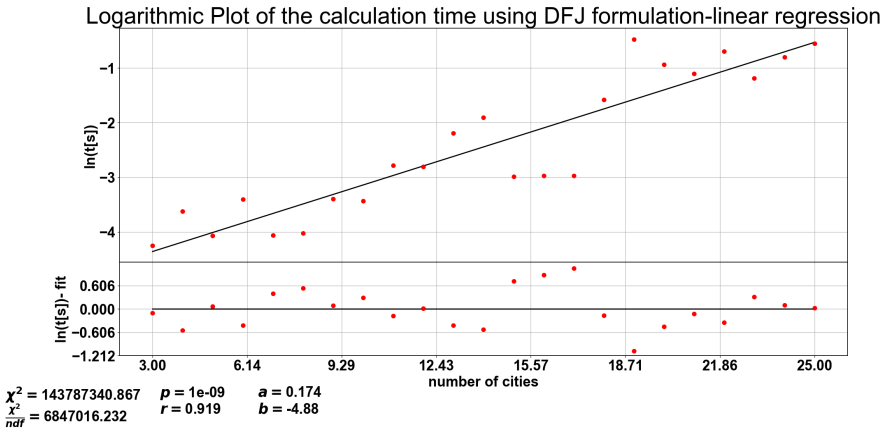


Figure (15): A linear regression plot of the natural logarithm of the solution times for the DFJ formulation.

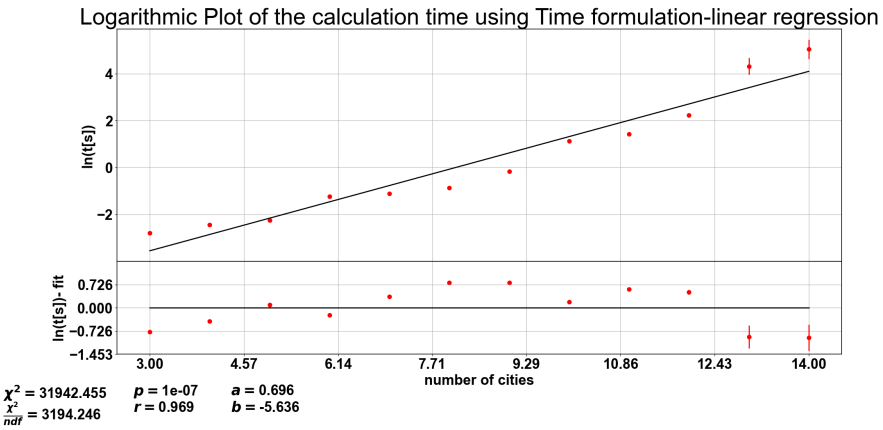


Figure (16): Linear regression of the natural logarithm of the solution times for the Time formulation.

From the linear regressions, we obtain the results

$$t_{\text{DFJ}} \approx e^{-4.88} e^{0.174n} = 0.0076 \cdot 2^{0.251n} = 0.0076 \cdot 1.19^n \,, \tag{20}$$

and

$$t_{\text{Time}} \approx e^{-5.636} e^{0.696n} = 0.0037 \cdot 2^{1.004n} = 0.0037 \cdot 2.005^n \,. \tag{21}$$

If we look at the exponential approximations shown in Eq.(20) and Eq.(21), we can easily observe that the DFJ formulation causes a much slower increase in computation time, as it has a significantly smaller base in the exponential. These results might also be surprising, as the best known classical TSP algorithms run in time $O(2^n)$ [24], and it is generally unknown if classical algorithms that run in time $O(1.9999^n)$ exist, much less in time $O(1.19^n)$. So, what is going on? The answer lies in the fact that the statement we just gave describes the solution time of the worst possible case, meaning acts as an upper bound of the computation time. The values we have calculated through linear regression, however, instead refer to the average computation time, which can be significantly smaller. Therefore, there is no contradiction.

4 Lagrange Factor Calibration

In order to make a comparison between the two methods, we first need to find the optimal Lagrange factors to be used with each formulation. This is important to ensure that the solutions we get from using both formulations are the best each formulation can deliver, and thus comparable to each other.

From to the way that the QUBO function is formulated in both cases, we can see that it consists of two parts: the cost function and the constraints. The constraints consist of similarly structured terms: Lagrange factors multiplied by a binary sum (such as in Eq. (12)). The cost function, on the other hand, consists of binary sums weighted by the cost c_{ij}. Since we want our constraints to not be underweighted or overweighted compared to the cost function, as we have described above, a good ansatz for the Lagrange factors might be to test different multiples of our problem's average cost.

4.1 Hamming distance analysis

Before going forward with the rest of our experiments, we first set out to demonstrate how the choice of Lagrange factor can influence the solution we receive to our problem.

In order to do that, we will be utilizing the QPU. We will be using the DFJ formulation of the TSP, where we will be testing different values for the equality constraint Lagrange factor (λ_0), while holding the inequality constraint Lagrange factor (λ_1) constant. We will get 1000 samples from the QPU and calculating the Hamming distance between each pair of samples. As we will see, this will give us an insight into how a suitable Lagrange factor can affect the way the quantum annealer searches for solutions.

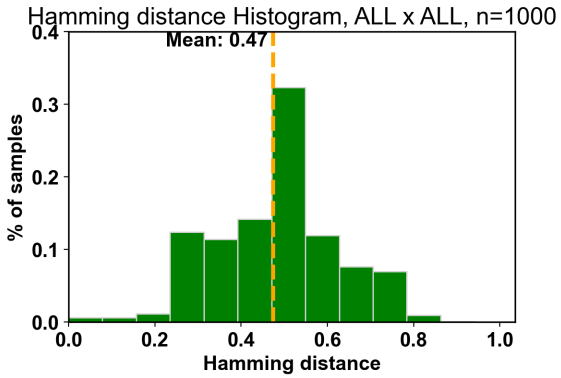


Figure (17): Hamming distance histogram for a 6 city TSP with λ_0 set to 6 times the average distance. The mean hamming distance, in this case, is 0.47.

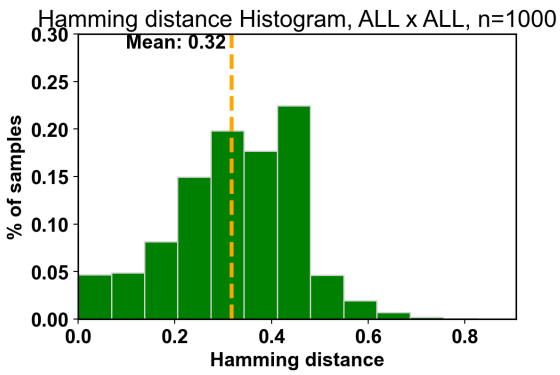


Figure (18): Hamming distance histogram for a 6 city TSP with λ_0 set to 0.8 times the average distance. The mean hamming distance, in this case, is 0.32.

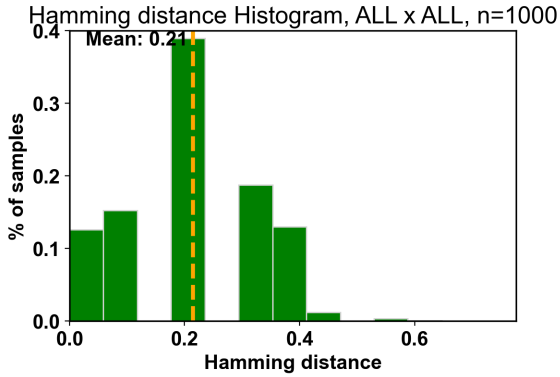


Figure (19): Hamming distance histogram for a 6 city TSP with λ_0 set to 0.5 times the average distance. The mean hamming distance, in this case, is 0.21.

As we can see from Fig. (17) - Fig. (19), as we decrease the Lagrange factor λ_0, the distribution of Hamming distances within our set of 1000 samples becomes increasingly skewed to the right. The mean Hamming distance becomes smaller for small values of λ_0, whereas for large λ_0, it approaches a value of 0.5, which is the expected Hamming distance of two binary strings that are chosen at random. In other words, for larger values of the Lagrange factor, the quantum annealer searches for solutions almost randomly. As λ_0 gets smaller, however, the search becomes more concentrated around certain areas of the solution space. This is also noticeable in the t-SNE representations of our data shown in Fig. (20) - Fig. (22), where we have set the inequality constraint Lagrange factor (λ_1) to 0.1 times the average cost, and the equality constraint Lagrange factor (λ_0) to 6, 0.8 and 0.5 times the average cost respectively. Comparing Fig. (20) to Fig. (21), we can see that in the former, the solutions are much more uniformly distributed, with no large clusters, or clumps, visible within the diagram. In the latter, however, we can observe several clusters around which many of the solutions are concentrated. This behavior is even more apparent in Fig. (22). We can also see that the percentage of valid solutions quickly drops as the Lagrange factor decreases. Valid solutions comprise the majority in Fig. (20), whereas few valid solutions are present in Fig. (22).

We can thus deduce that, to optimally solve a TSP or similar optimization problem on a quantum annealer, we have to select Lagrange factors that are low enough to allow the annealer to be able to more easily reach good solutions with an efficient search, but not too low, as to make finding a valid solution an extremely rare occurence.

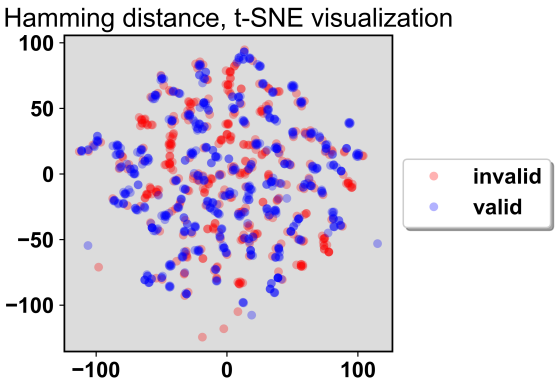


Figure (20): t-SNE visualization of the solutions samples of a 6-city TSP with λ_0 set to 6 and λ_1 to 0.1 times the average cost, using Hamming distance as a metric. Valid solutions are shown in blue, and invalid solutions are shown in red. The valid solutions constitute 62.4% of the total.

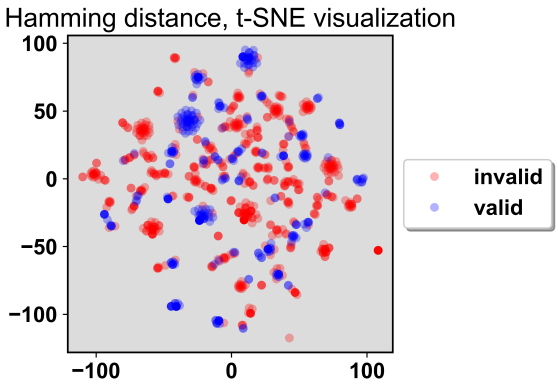


Figure (21): t-SNE visualization of the solutions samples of a 6-city TSP with λ_0 set to 0.8 and λ_1 to 0.1 times the average cost, using Hamming distance as a metric. Valid solutions are shown in blue, and invalid solutions are shown in red. The valid solutions constitute 37.6% of the total.

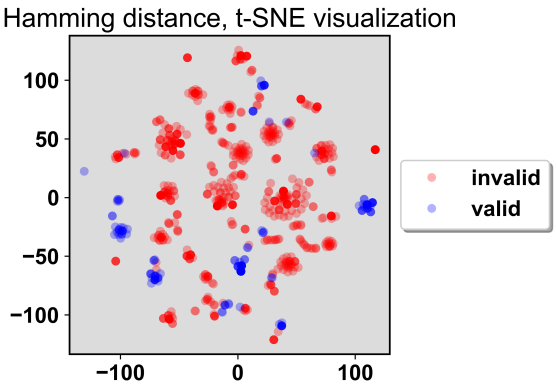


Figure (22): t-SNE visualization of the solutions samples of a 6-city TSP, with λ_0 set to 0.5 and λ_1 to 0.1 times the average cost, using Hamming distance as a metric. Valid solutions are shown in blue, and invalid solutions are shown in red. The valid solutions constitute 16.8% of the total.

Additionally, we decided to measure the mean Hamming distance between solution samples for several Lagrange factor values to obtain a clearer result. As we can see in Fig. (23), for smaller λ_0 values, the mean Hamming distance decreases, whereas for larger values of λ_0, it approaches one-half.

We performed a similar test, this time adjusting the 'annealing time', which is a parameter that can be customized when submitting a problem to the QPU. The resulting plot can be seen in Fig. (24). We observe that the data is all located around a constant value, with only small fluctuations. From this, we can discern that the precise value of the annealing time is unlikely to have any significant effect on the quality of the solutions we will find, or the probability of finding a valid solution. This might be the case because we are dealing with a relatively small problem, which the annealer can solve well, even with a small annealing time.

Another numerical parameter that can be adjusted when solving problems on the QPU is called 'chain strength'. We performed the same test as with the Lagrange factors and annealing time. This can be seen in Fig. (25). We can observe that the value of this parameter can affect the mean Hamming

distance. While for most chain strength values, the mean Hamming distance stays nearly constant, there is a valley that forms in the center of the plot with a minimum at 0.4. We also observe that the number of valid samples drops as the chain strength increases. This might be an indication that the chain strength functions similar to the Lagrange parameters: if it receives a high value, priority is given to fulfilling its purpose when solving the problem, overpowering any other Lagrange factors that are involved in our problem, meaning the constraints are not upheld and a valid solution cannot be easily reached.

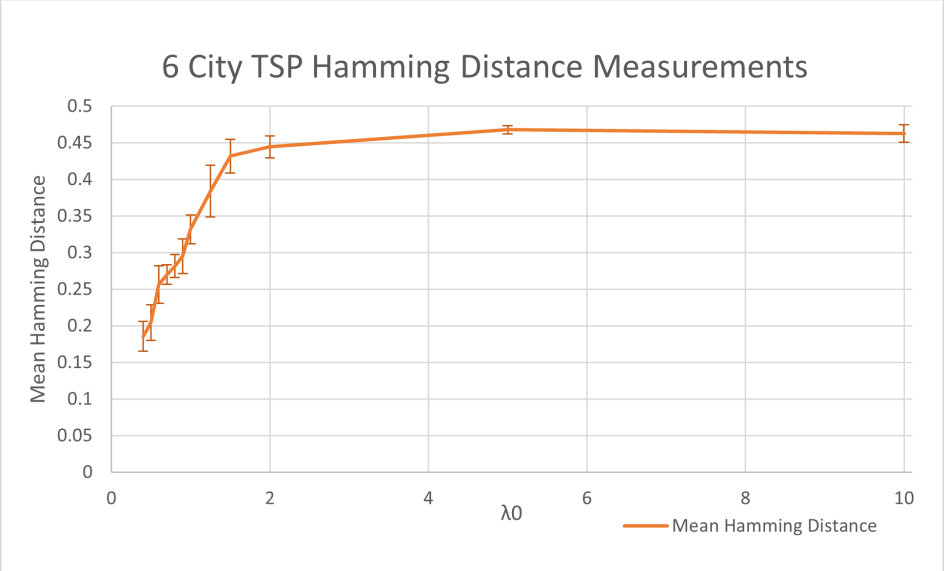


Figure (23): The resulting mean Hamming distance between each pair of 1000 samples, for varying values of the Lagrange factor λ_0. These results were from the 6 city TSP, encoded with the DFJ formulation.

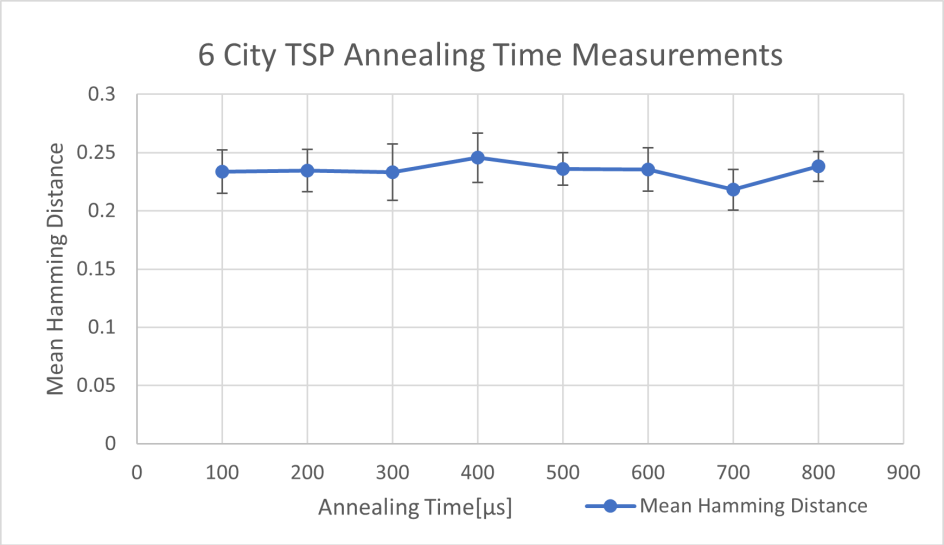


Figure (24): The resulting mean Hamming distance between each pair of 1000 samples, for varying values of the Annealing Time. These results were from the 6 city TSP, encoded with the DFJ formulation.

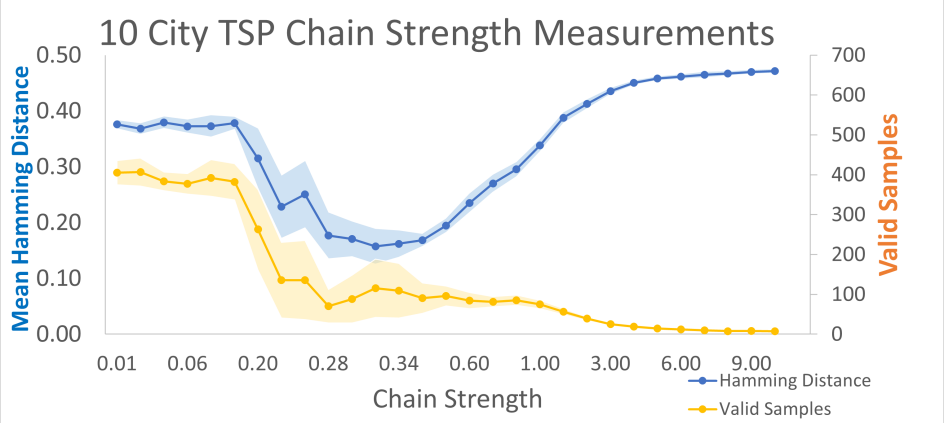


Figure (25): The resulting mean Hamming distance between each pair of 1000 samples, as well as the number of samples that are valid, for varying values of the Chain Strength parameter. These results were from the 10 city TSP, encoded with the DFJ formulation. The x-axis is shown on a logarithmic scale. The mean Hamming distance is rendered in blue, whereas the number of valid samples is rendered in orange. The confidence bands around each line show the standard deviation around each value.

4.2 Time Formulation λ scan

The Time formulation only includes one Lagrange parameter, which makes it easier to calibrate than the DFJ formulation, which includes two. We tested many different λ values in a range between 0.9 and 10, which can be seen in Table (1). This test was done for various numbers of cities, namely 5, 10, 15, 20 and 25. 5 tests were performed for each city number-Lagrange factor combination. The results can be seen in Fig. (26). We notice that smaller lambda factors reliably lead to solutions of smaller cost, up to a point, after which no valid solutions can be found. For smaller problem sizes, such as the 10 city problem and especially the 5 city problem, we notice that when a valid solution is reached, it will almost always be the optimal solution, as there don't exist many other low-cost valid solutions in this relatively simple problem.

λ values tested												
$\lambda/average\ cost$	0.9	1	1.1	1.2	1.3	1.4	1.5	1.6	1.7	2	5	10
num. of tests	5	5	5	5	5	5	5	5	5	5	5	5

Table (1): The λ values that were tested for several numbers of cities.

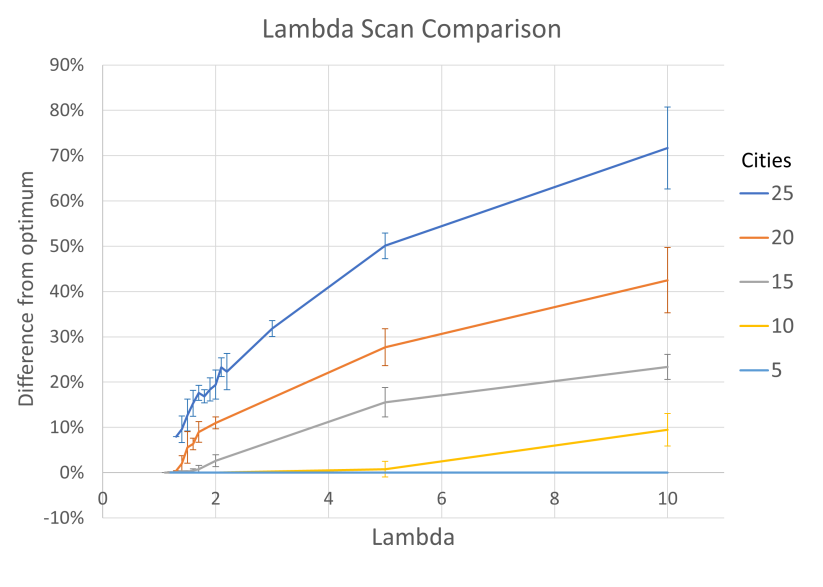


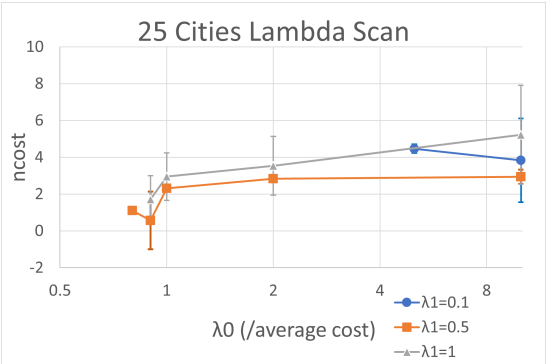
Figure (26): Lambda scan results for different numbers of cities, all shown on the same plot. Measurement results for testing different λ for the Time formulation, in a 5 to 25 city TSP, showing the solution's cost as a percentage difference from the optimum. The error bars denote the standard deviation from 5 runs.

4.3 DFJ Formulation λ scan

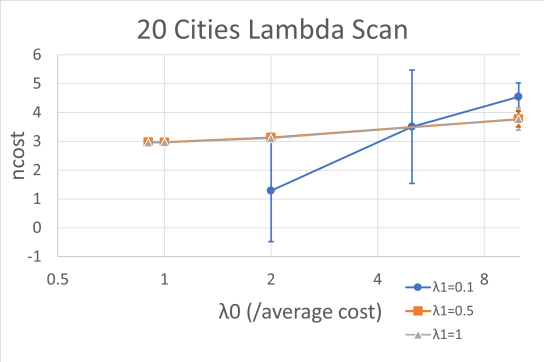
To calibrate the λ factors for the case of the DFJ formulation was slightly less straightforward as there exist two of them, one for the equality constraints and one for the inequality constraints. We decided to simply approach this by trying all combinations from a set of λ_0 values and a set of λ_1 values. We had to limit the number of different Lagrange factor values to be tested, in order for the λ scan to be feasible in a realistic time frame. This is the case due to the nature of our testing, where we are trying all combinations between two parameters, rather than just one parameter, and also due to the fact that problems in the DFJ formulation often require multiple solution cycles. However, the results we obtained are enough to settle on a set of Lagrange factors that works well for each case.

The resulting plots can be seen in Fig. (27). As when using the Time formulation, we can see that larger Lagrange factors generally lead to a larger solution cost. In comparison to the Time formulation case, we can see that the results fluctuate much more, also showing much larger relative uncertainties. This might relate to the way problems are tackled in the DFJ formulation: because each problem has to be solved multiple times, the 'path' taken during the process of a solution can be vastly different, leading to large deviations between results.

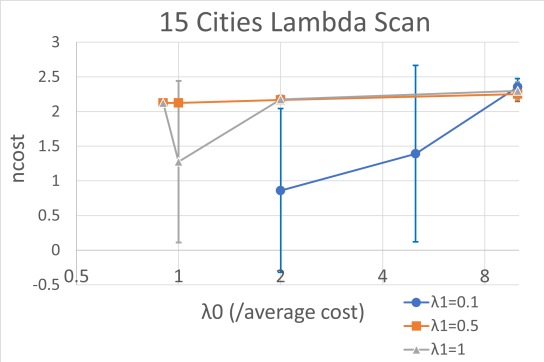
In Fig. (28), we have plotted the smallest value of the equality constraint Lagrange factor λ_0 that still yielded a valid solution for a given value of the inequality constraint Lagrange factor λ_1. We notice that small λ_1 values tend to require larger λ_0 values in order to return a valid solution. This, however, does not appear to be the case for larger values of λ_1. This may be explained by our constraints 'competing against each other', if both λ are low: If a city does not fulfill the equality constraint, i.e. does not have exactly 2 active connections, then perhaps having 3 or more connections to other cities may end up forming a loop. Conversely, in an attempt to eliminate a loop, the equality constraints may be violated by leaving a city with only one connection, or more than two.



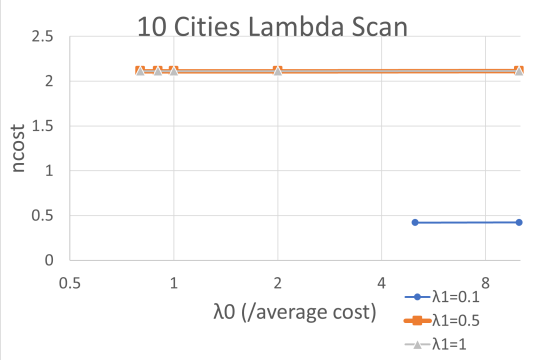
(a) 25 city DFJ λ scan



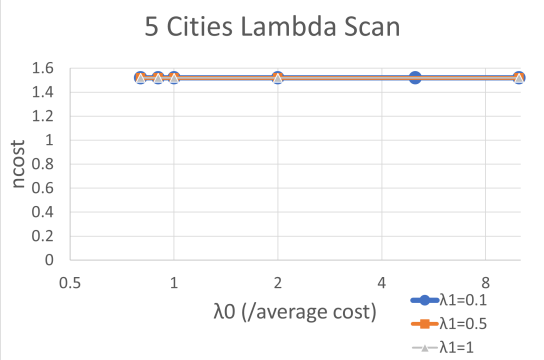
(b) 20 city DFJ λ scan



(c) 15 city DFJ λ scan



(d) 10 city DFJ λ scan



(e) 5 city DFJ λ scan

Figure (27): Measurement results for testing different λ for the DFJ formulation, in a 25, 20, 15, 10 and 5 city TSP, showing the solution's cost as a percentage difference from the optimum. The vertical axis shows the normalized cost obtained for each set of Lagrange factors for that specific problem. The horizontal axis shows the value of the equality constraint Lagrange factor, measured as a multiple of the specific problem's average cost.

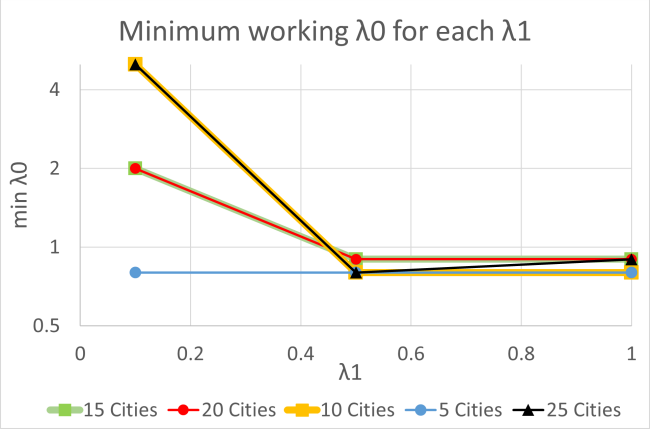


Figure (28): The minimal λ_0 value that still returns a result, as a function of the value of λ_1.

4.4 Results

In the appendix, in tables (7) to (11) we include the data obtained from the lambda scans of TSPs with different city numbers.

For each number of cities, we choose the smallest λ that still returned at least one successful solution from the five trials, as that solution had, in all cases, the lowest cost. This leads to the following result, shown in Table (2) and Fig. (29):

Table (2): Resulting λ that was chosen for each n using the Time formulation. The Lagrange factor is written as a multiple of the average cost in our problem. Although we can see some variation, the value stays around 1.3. We have included a statistical uncertainty of 0.05 because of the interval between the values at which we measured.

n	5	10	15	20	25
$\lambda/average$	1.2 ± 0.05	1.3 ± 0.05	1.1 ± 0.05	1.3 ± 0.05	1.3 ± 0.05

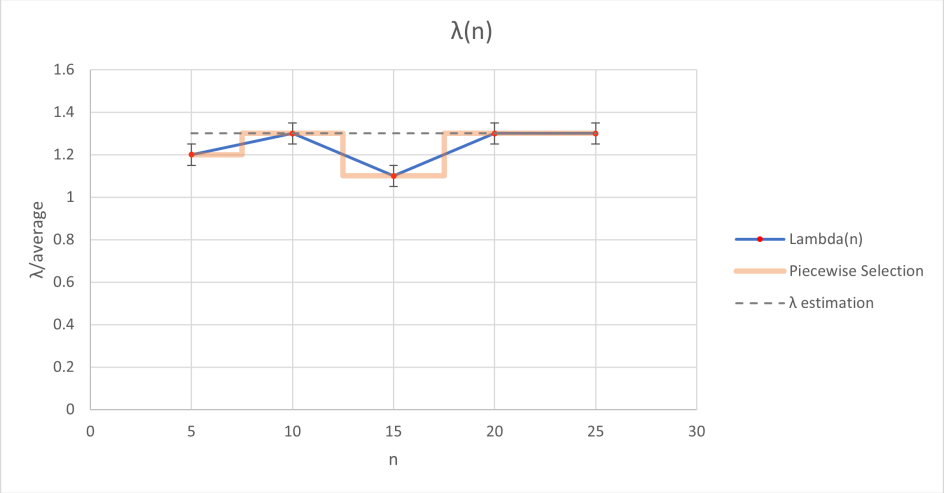


Figure (29): Estimated λ as a function of the city number n. The value of λ is shown as a factor of the average cost. The orange and gray lines show different ways of approximating λ for intermediate values of n.

Along with the best measured Lagrange factor values, Fig. (29) also includes a couple of lines that can be used to estimate a good factor of λ_0 for problem sizes where we did not conduct this test. The orange line is the way we employed to estimate the λ locally, for values of n that lie close to problem

sizes where the Lagrange factor value was experimentally determined. It is a piecewise function, where for each n where we did obtain a λ value, we also assigned that value to the points nearest to it. We also observe that the λ value seems to mostly be independent of the problem size n, as it fluctuates around an average value of 1.24. Also, at no point did the best Lagrange factor exceed a value of 1.3 times the average distance. This means that we are able to make an extrapolation (indicated by the gray dashed line) and say that for any given problem solved by the Time formulation, a safe guess of a relatively effective value for the Lagrange factor might be approximately 1.3 times the average distance between cities.

The full data of the λ scan for the DFJ formulation can be found in the appendix in tables (12) to (16). The 'optimal' Lagrange factor combinations, i.e. the ones that returned the best result for each problem size during testing, can be found in Table (3). We also determined the 'second best' Lagrange factor combinations, which can be seen in Table (4). This was needed for when we would do the full comparison with all problem sizes: If the 'optimal' Lagrange factor combination for the problem size we were testing returned no valid solutions at all, we would then switch to this 'second best' combination and perform the test again for this specific problem size.

Table (3): The resulting 'best' λ_0 / λ_1 combination that was chosen for each n when using the DFJ formulation. The Lagrange factors are written as a multiple of the average cost in our problem. To select these pairs of Lagrange factors, we looked at the data shown in Fig. (**??**) - Fig. ((27)), and for each problem size, selected the pair of λ factors that led to the lowest total cost in the solution, i.e. the lowest point in the plots.

n	5	10	15	20	25
$\lambda_0/average$	0.8	5	2	2	0.9
$\lambda_1/average$	0.1	0.1	0.1	0.1	0.5

Table (4): The resulting 'second best' λ_0 / λ_1 combination that was chosen for each n when using the DFJ formulation. The Lagrange factors are written as a multiple of the average cost in our problem. To select these pairs of Lagrange factors, we looked at the data shown in Fig. (**??**) - Fig. ((27)), and for each problem size, selected the pair of λ factors that led to the second lowest total cost in the solution, i.e. the second lowest point in the plots. This set of Lagrange factors was created to be used in case the 'best' sets of Lagrange factors from Table ((3)) did not provide any valid solution during the testing.

n	5	10	15	20	25
$\lambda_0/average$	0.8	10	5	1	0.8
$\lambda_1/average$	0.5	0.1	0.1	0.5	0.5

5 Comparison between Formulations

Having found a way to obtain some good Lagrange factors for TSP instances with different city numbers in Section (4), our next step was to perform a detailed test of our two formulations, to see how they perform with different problem sizes.

5.1 Setup

To compare our methods, we determine the best Lagrange factors we could use from Section (4), shown in tables (2), (3) and (4). We perform tests on TSP instances of 5 to 25 cities. Each test is repeated 20 times for each formulation. In case no valid result is generated from any of the 20 tries, we run the test again, this time using a slightly increased, 'second-best' set of Lagrange factors, again as determined in part (4). We also find the optimal cost value for each TSP instance, in order to compare it to the results by both formulations on D-Wave Hybrid. In order to do this, we solve each TSP instance with DOcplex. For this, we use the DFJ formulation, as with the Time formulation, considerable time is required to reach a solution for problems of size larger than 15, as shown in section (3).

5.2 Results

The results of our tests can be seen in Fig. (30) and Fig. (31). From this data, we can observe that all three methods follow a similar course, with the total cost growing as the number of cities increases. We observe that for particularly small problem sizes, in this case for 5 to 10 cities, both formulations settle on the optimal solution, with the DFJ formulation first starting to deviate from the optimum at 11 and the Time formulation at 14 cities. From there, both start to exhibit an increasingly large difference from the optimal value.

The solution costs obtained by using the Time formulation mostly appear to be slightly lower than the equivalent costs of the DFJ formulation, even though the last 3 TSP instances tested seem to be an exception to this, with the DFJ method obtaining better TSP solutions. This behavior that is observed in the last 3 problem sizes has another interesting connection: the large drop in solution cost (or rise in solution quality) that we can observe in the DFJ formulation coincides with a sharp increase in the (average) amount of inequality constraints that have been added to the problem. This can be seen in Fig. (32). A possible interpretation could be that for these numbers of cities, a solution with no subtours is less straightforward to come across, as it has a relatively increased cost compared to solutions that feature subtours. Therefore, more attempts have to be made before a valid solution is reached, with more constraints being added in the process. It may be due to the larger number of restrictions imposed by these constraints, that our solver is more often guided to a good solution. In any case, the fact that the DFJ formulation starts to outperform the Time formulation after 22 cities can be attributed to the specific TSP instances we are conducting our testing with. An interesting question to ask might be whether we can prove if this is guaranteed to happen for any TSP instances past a certain problem size.

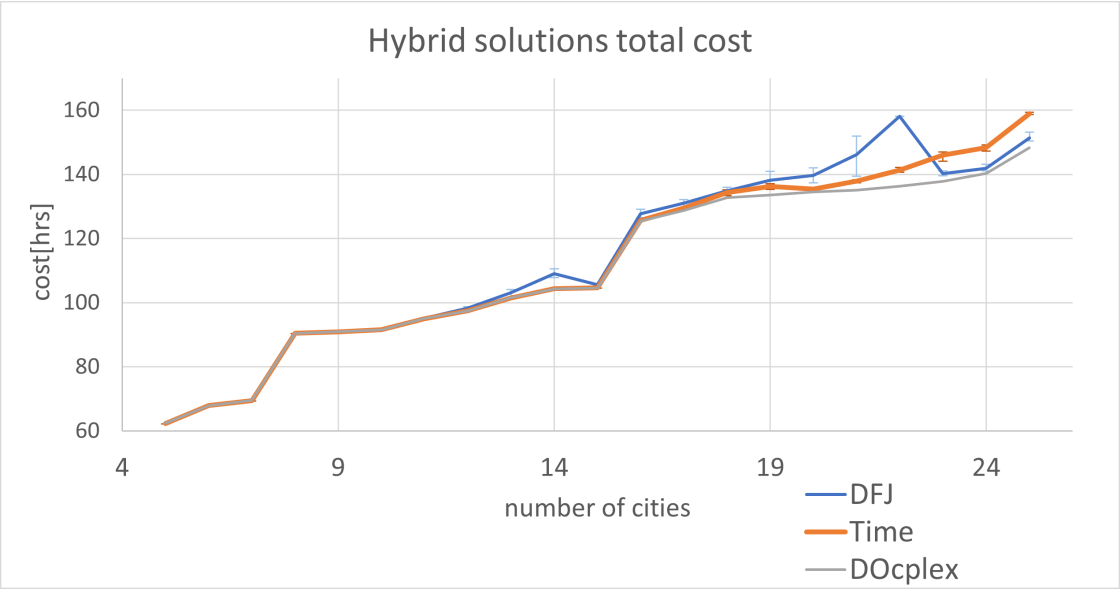


Figure (30): The total cost of the solutions obtained for different numbers of cities, by using the DFJ and Time formulation on D-Wave Hybrid, rendered in blue and orange respectively, as well as the optimal solution, rendered in grey. The error bars display the 20th and 80th percentile of the found solution costs.

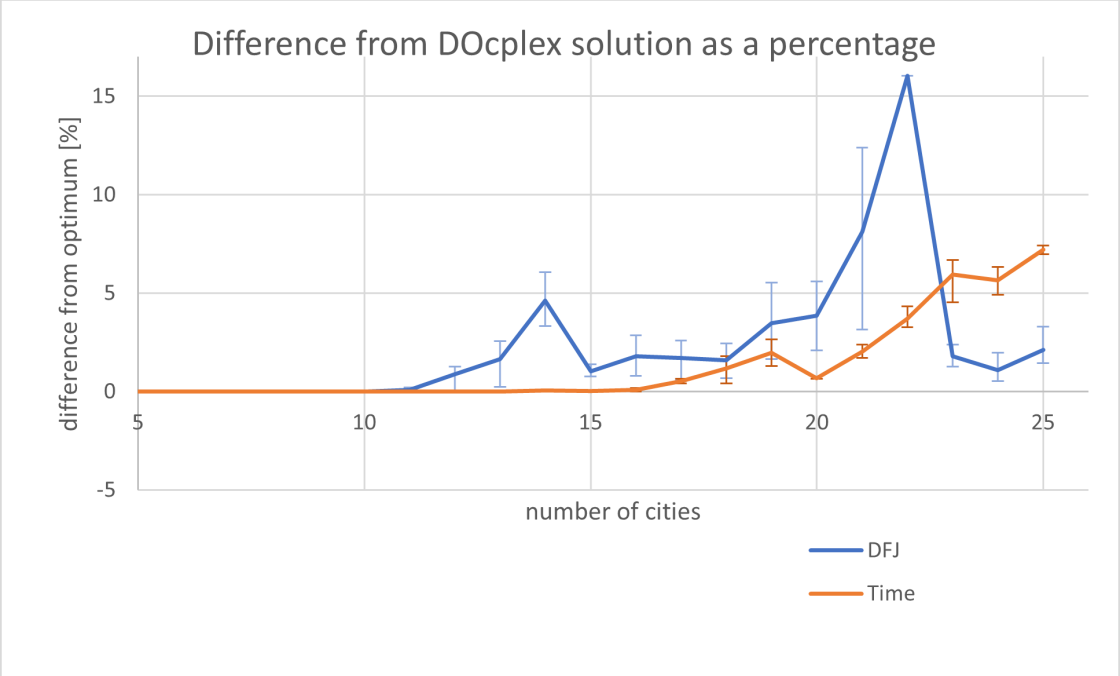


Figure (31): Difference from the optimum of the TSP solutions obtained for different numbers of cities, by using the DFJ formulation (rendered in blue) and Time formulation (rendered in orange) on D-Wave Hybrid, displayed as a percentage. The error bars display the 20th and 80th percentile of the percentage differences.

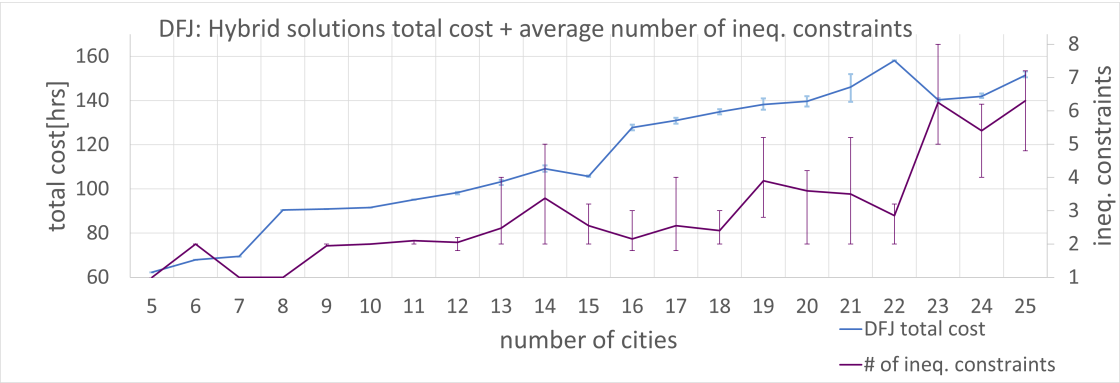


Figure (32): The total cost for the DFJ formulation, shown together with the average number of inequality constraints that were added in each problem. The error bars display the 20th and 80th percentile of the found solution costs/number of inequality constraints.

6 Summary and Outlook

In this Bachelor's Thesis, we have discussed two different formulations for solving the TSP on a quantum annealer: the DFJ and Time formulations. We implemented each of these methods in Python, so as to be able to solve these problems on D-Wave Hybrid. We also found a way to use Google Earth's navigation data to create more realistic TSP instances. We then investigated the effect a different choice of Lagrange factors can have on the solutions reached by the D-Wave QPU by charting Hamming distance measurement data in histograms and t-SNE visualizations. We were then able to determine that a smaller Lagrange factor value results in a more clustered search, although a Lagrange factor that is too small leads to invalid solutions. Having established this, our next step was to perform a calibration of the Lagrange factors in both formulations, in order to determine which

values we should use in order to reach good results in a given TSP. After we found the Lagrange parameters we would use, we were finally able to conduct a test to see how the two formulations compare to each other, as well as to a classical TSP solver, in terms of the quality of the solutions reached.

From the results shown in Section (5.2), we can observe that the DFJ and Time formulations provide solutions of similar quality, with the Time formulation performing slightly better on most instances, although the DFJ formulation appears to begin outperforming the Time formulation past a certain problem size. We could not verify whether this behavior will persist when presented with much larger problems, as we restricted our range to 25 cities. This was mainly due to time considerations, as the time required to generate a QUBO and, in the case of the DFJ formulation, the number of attempts required to solve a given problem, increase with the problem size. However, what we can take away from our results is that, as far as we have been able to test, the Time formulation performs similarly to, if not slightly better than, the DFJ formulation on a quantum annealer, at least in the TSP instances that we were able to test within our existing limitations. This is in contrast to the case on classical computers, where the Time formulation can only solve TSP instances of very limited size in a realistic time frame.

It will certainly be interesting to see in which direction the trend continues as devices become larger in the future.

6.1 Acknowledgements

I would like to thank Prof. Kristel Michielsen for giving me the opportunity to work on this thesis, as well as the whole team in Forschungszentrum Jülich for being tremendously welcoming and helpful during my work at FZJ. Special thanks to Alejandro Montañez-Barrera and Dennis Willsch for their helpful advice and valuable insights, which helped me during the whole process.

A Appendix: Setup

In this section, we will include some additional resources related to the 'Setup' section of this thesis, including the cost matrix for the 5-to-25 city TSP used in most of our testing, as well as the solutions to some realistic TSPs.

A.1 25 city cost matrix

25 City TSP Cost Matrix																									
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
0	0.00	23.25	15.68	11.01	7.28	5.95	3.10	17.93	8.93	18.63	6.09	11.39	16.67	3.91	6.09	12.48	6.89	17.47	15.40	12.32	6.50	21.82	11.04	5.89	27.51
1	23.25	0.00	19.97	12.56	24.33	28.34	21.53	33.89	24.96	6.31	19.43	16.09	30.03	22.35	17.41	32.68	17.65	21.63	10.83	14.83	27.50	3.75	22.09	17.91	5.16
2	15.68	19.97	0.00	14.37	11.50	18.05	17.57	21.00	12.03	14.29	9.75	6.12	17.09	13.67	14.62	27.67	17.12	2.42	9.73	7.27	16.10	17.40	9.14	13.19	23.54
3	11.01	12.56	14.37	0.00	12.67	15.87	9.49	23.83	14.86	10.33	8.54	8.87	22.08	10.58	5.24	20.51	5.59	15.98	7.57	8.16	15.42	13.45	14.14	5.84	16.97
4	7.28	24.33	11.50	12.67	0.00	6.91	9.92	11.50	2.53	18.04	4.53	8.86	10.35	3.52	9.36	19.17	11.91	12.98	14.29	10.15	5.03	21.24	4.09	7.65	27.34
5	5.95	28.34	18.05	15.87	6.91	0.00	8.46	17.14	8.50	23.20	10.65	15.45	16.32	7.22	11.16	17.05	11.96	19.57	19.96	16.79	3.51	26.38	10.68	10.84	32.48
6	3.10	21.53	17.57	9.49	9.92	8.46	0.00	20.61	11.64	17.99	8.07	12.27	19.48	6.59	4.57	11.52	4.95	19.41	14.76	13.20	9.19	21.18	13.21	5.41	25.94
7	17.93	33.89	21.00	23.83	11.50	17.14	20.61	0.00	9.40	28.63	15.96	18.54	5.29	14.21	20.87	30.21	23.47	22.92	23.95	19.95	13.93	31.70	12.00	19.16	37.67
8	8.93	24.96	12.03	14.86	2.53	8.50	11.64	9.40	0.00	19.30	6.76	9.47	8.17	5.13	11.59	20.77	13.97	13.60	14.91	10.87	5.98	22.48	3.48	9.88	28.62
9	18.63	6.31	14.29	10.33	18.04	23.20	17.99	28.63	19.30	0.00	13.82	10.57	24.53	17.29	13.91	29.80	15.18	16.02	5.26	9.21	22.15	3.70	16.51	13.38	9.80
10	6.09	19.43	9.75	8.54	4.53	10.65	8.07	15.96	6.76	13.82	0.00	6.07	14.59	4.12	5.96	18.12	8.51	11.53	10.54	7.00	9.09	16.93	5.88	4.24	23.12
11	11.39	16.09	6.12	8.87	8.86	15.45	12.27	18.54	9.47	10.57	6.07	0.00	14.64	9.71	8.97	23.55	11.57	7.71	6.10	2.01	13.70	13.61	6.68	7.46	19.71
12	16.67	30.03	17.09	22.08	10.35	16.32	19.48	5.29	8.17	24.53	14.59	14.64	0.00	13.13	19.59	28.77	21.97	18.71	20.04	16.01	13.96	27.65	8.44	17.88	33.75
13	3.91	22.35	13.67	10.58	3.52	7.22	6.59	14.21	5.13	17.29	4.12	9.71	13.13	0.00	7.26	15.73	8.93	15.34	13.92	10.55	5.40	20.34	7.30	5.54	26.44
14	6.09	17.41	14.62	5.24	9.36	11.16	4.57	20.87	11.59	13.91	5.96	8.97	19.59	7.26	0.00	16.06	2.95	16.29	11.55	9.92	11.20	17.97	11.36	2.18	21.83
15	12.48	32.68	27.67	20.51	19.17	17.05	11.52	30.21	20.77	29.80	18.12	23.55	28.77	15.73	16.06	0.00	16.02	29.51	26.05	24.44	17.88	32.47	22.97	16.52	37.08
16	6.89	17.65	17.12	5.59	11.91	11.96	4.95	23.47	13.97	15.18	8.51	11.57	21.97	8.93	2.95	16.02	0.00	18.81	12.25	12.46	12.44	18.50	13.93	4.76	22.03
17	17.47	21.63	2.42	15.98	12.98	19.57	19.41	22.92	13.60	16.02	11.53	7.71	18.71	15.34	16.29	29.51	18.81	0.00	11.38	8.93	17.85	19.05	10.84	14.69	25.16
18	15.40	10.83	9.73	7.57	14.29	19.96	14.76	23.95	14.91	5.26	10.54	6.10	20.04	13.92	11.55	26.05	12.25	11.38	0.00	4.98	18.96	8.38	11.98	10.00	14.48
19	12.32	14.83	7.27	8.16	10.15	16.79	13.20	19.95	10.87	9.21	7.00	2.01	16.01	10.55	9.92	24.44	12.46	8.93	4.98	0.00	14.79	12.23	7.85	8.26	18.33
20	6.50	27.50	16.10	15.42	5.03	3.51	9.19	13.93	5.98	22.15	9.09	13.70	13.96	5.40	11.20	17.88	12.44	17.85	18.96	14.79	0.00	25.39	8.77	9.97	31.49
21	21.82	3.75	17.40	13.45	21.24	26.38	21.18	31.70	22.48	3.70	16.93	13.61	27.65	20.34	17.97	32.47	18.50	19.05	8.38	12.23	25.39	0.00	19.83	16.48	6.51
22	11.04	22.09	9.14	14.14	4.09	10.68	13.21	12.00	3.48	16.51	5.88	6.68	8.44	7.30	11.36	22.97	13.93	10.84	11.98	7.85	8.77	19.83	0.00	9.60	25.70
23	5.89	17.91	13.19	5.84	7.65	10.84	5.41	19.16	9.88	13.38	4.24	7.46	17.88	5.54	2.18	16.52	4.76	14.69	10.00	8.26	9.97	16.48	9.60	0.00	22.49
24	27.51	5.16	23.54	16.97	27.34	32.48	25.94	37.67	28.62	9.80	23.12	19.71	33.75	26.44	21.83	37.08	22.03	25.16	14.48	18.33	31.49	6.51	25.70	22.49	0.00

Table (5): The cost matrix we primarily used when solving TSP instances in this thesis. The row and column indices, rendered in light blue, denote the index of the city travelled from/to. The matrix is symmetric. The costs are shown in hours of driving time.

Names of Cities by Index													
City Index	0	1	2	3	4	5	6	7	8	9	10	11	12
City Name	Berlin	Madrid	Rome	Paris	Vienna	Warsaw	Hamburg	Bucharest	Budapest	Barcelona	Munich	Milan	Sofia
City Index	13	14	15	16	17	18	19	20	21	22	23	24	-
City Name	Prague	Cologne	Stockholm	Amsterdam	Naples	Marseille	Turin	Krakow	Valencia	Zagreb	Frankfurt	Seville	-

Table (6): The name of each city used in the (up to) 25-city TSP, shown by index

A.2 Some more maps

In this section, we decided to include some more of the geographically realistic TSP solutions we generated during the writing of this thesis, which we did not include in the main text.

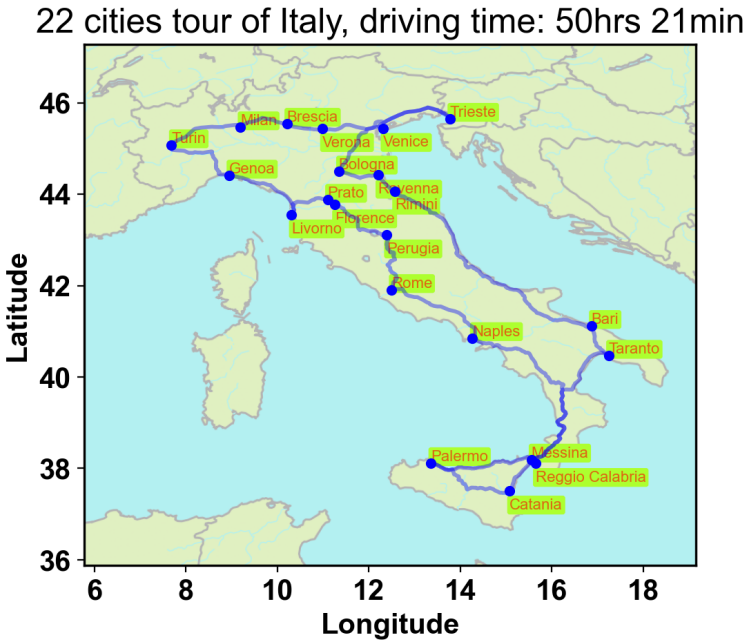


Figure (33): An example of a TSP solution with geographically accurate data. In this case, we included 22 of the largest cities of Italy. In this example, the total driving time amounted to about 50 hours.

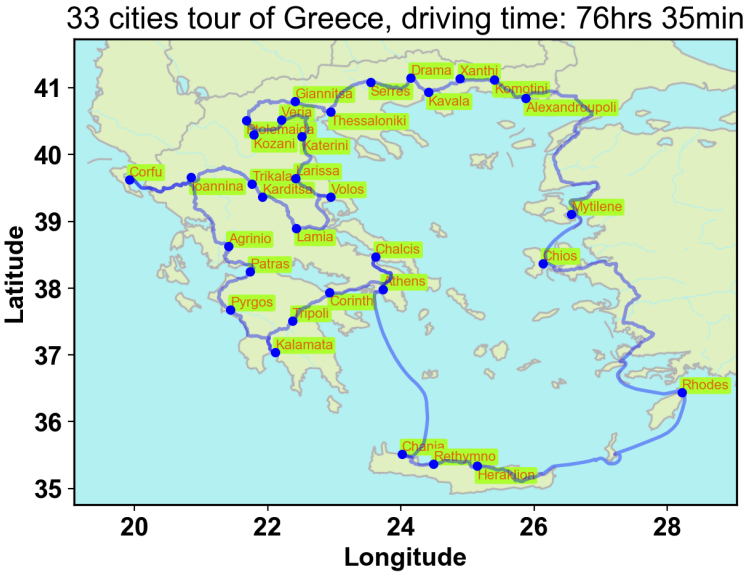


Figure (34): An example of a TSP solution with geographically accurate data. In this case, we included the 33 largest cities of Greece. In this example, the total driving time amounted to about 76 and a half hours. The large travel time primarily comes from the need to use a ferry at some stages in the tour. Due to this fact, the total travel time could also vary a lot from the prediction.

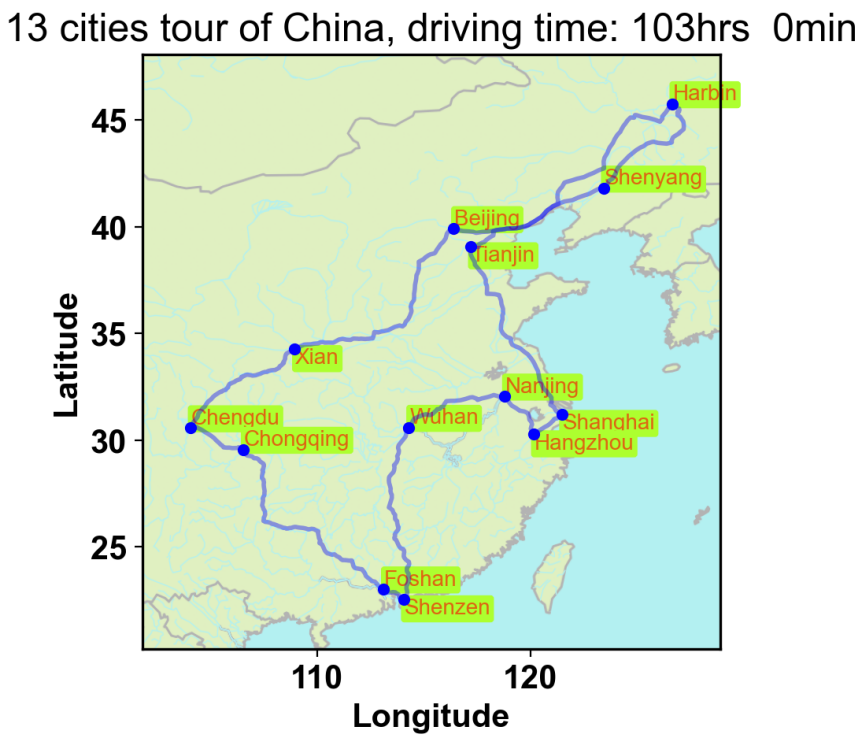


Figure (35): An example of a TSP solution with geographically accurate data. In this case, we included 13 of the largest cities of China. In this example, the total driving time amounted to 103 hours.

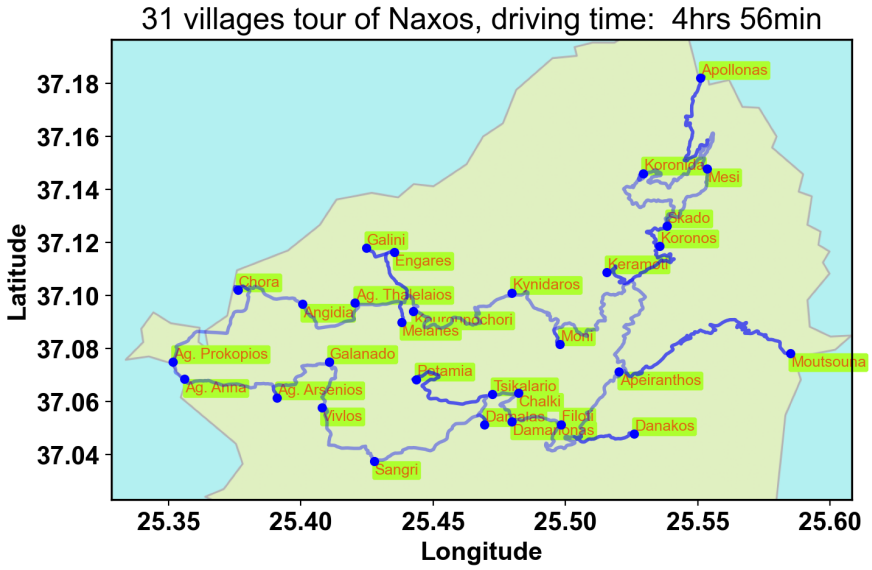


Figure (36): An example of a TSP solution with geographically accurate data. In this case, we included 31 villages on the island of Naxos, in Cyclades, Greece. In this example, the total driving time amounted to just below 5 hours.

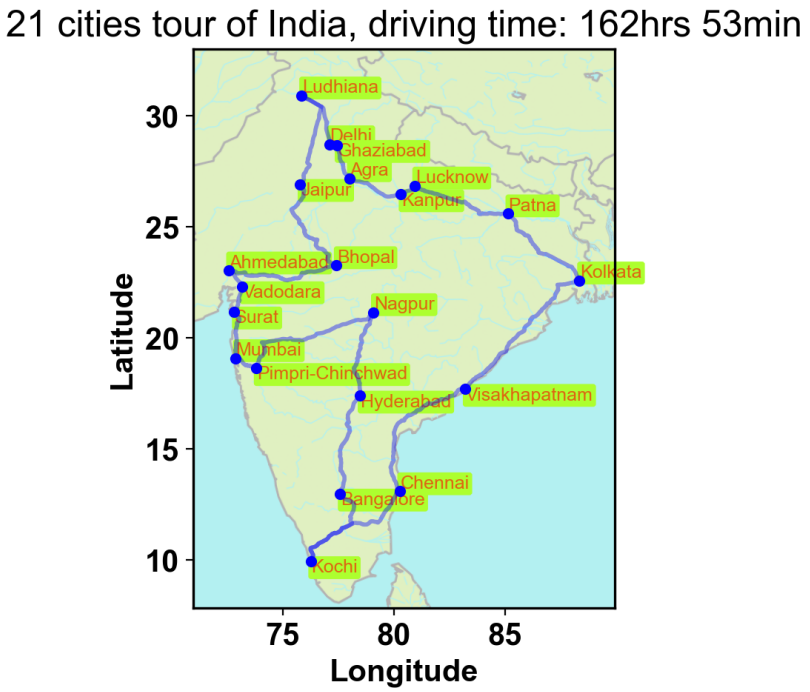


Figure (37): An example of a TSP solution with geographically accurate data. In this case, we included 21 of the largest cities in India. In this example, the total driving time amounted to about 163 hours.

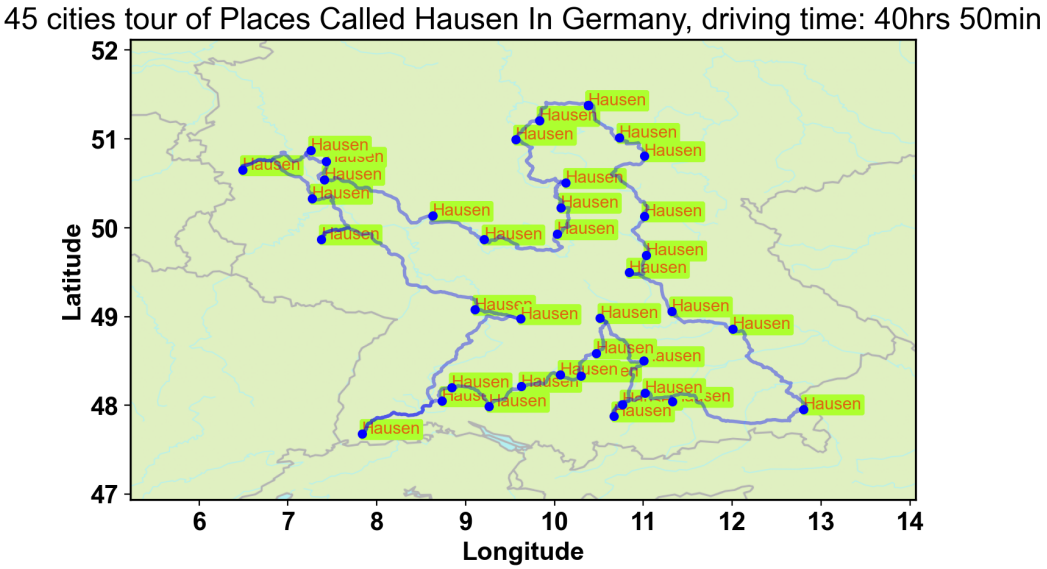


Figure (38): A tour between 45 instances of the place name 'Hausen', which is possibly the most common in Germany [25], according to most sources. In this example, the total driving time amounted to about 41 hours. Note that it is not certain if 'Hausen' is the most common place name in Germany. Depending on the source, another candidate might be 'Steinbach' [26]. It should be noted that both names return exactly 107 results on a postal code search [27].

B Appendix: Lagrange Factor Calibration

In this section, we provide some additional data and other information relating to Section 4.

B.1 Time

Lambda	Average of ncost	Success Count	StdDev of ncost
0.9	-	0	-
1	-	0	-
1.1	-	0	-
1.2	3.228	5	0.000
1.3	3.228	5	0.000
1.4	3.228	5	0.000
1.5	3.228	5	0.000
1.6	3.228	5	0.000
1.7	3.228	5	0.000
2	3.228	5	0.000
5	3.228	5	0.000
10	3.228	5	0.000

Table (7): Lambda scan data for the 5 city problem, Time formulation.

Lambda	Average of ncost	Success Count	StdDev of ncost
0.9	-	0	-
1	-	0	-
1.1	-	0	-
1.2	-	0	-
1.3	2.839	5	0.000
1.4	2.839	5	0.000
1.5	2.839	5	0.000
1.6	2.839	5	0.000
1.7	2.839	5	0.000
2	2.839	5	0.000
5	2.861	5	0.049
10	3.107	5	0.101

Table (8): Lambda scan data for the 10 city problem, Time formulation.

Lambda	Average of ncost	Success Count	StdDev of ncost
0.9	-	0	-
1	-	0	-
1.1	3.254	5	0.000
1.2	3.257	5	0.005
1.3	3.259	5	0.006
1.4	3.264	5	0.005
1.5	3.261	5	0.006
1.6	3.271	5	0.010
1.7	3.280	5	0.026
2	3.340	5	0.043
5	3.761	5	0.106
10	4.014	5	0.089

Table (9): Lambda scan data for the 15 city problem, Time formulation.

Lambda	Average of ncost	Success count	StdDev of ncost
0.9	-	0	-
1	-	0	-
1.1	-	0	-
1.2	-	0	-
1.3	4.174	1	undefined
1.4	4.239	2	0.073
1.5	4.387	3	0.144
1.6	4.420	4	0.052
1.7	4.529	4	0.095
2	4.612	5	0.054
5	5.308	5	0.170
10	5.922	5	0.299

Table (10): Lambda scan data for the 20 city problem, Time formulation.

Lambda	Average of ncost	Success Count	StdDev of ncost
0.9	-	0	-
1	-	0	-
1.1	-	0	-
1.2	-	0	-
1.3	-	0	-
1.4	4.496	5	0.121
1.5	4.625	2	0.144
1.6	4.731	5	0.117
1.7	4.825	5	0.068
1.8	4.794	5	0.059
1.9	4.856	5	0.105
2	4.900	5	0.132
2.1	5.059	5	0.085
2.2	5.019	5	0.165
3	5.408	5	0.073
5	6.158	5	0.116
10	7.043	5	0.371

Table (11): Lambda scan data for the 25 city problem, Time formulation.

B.2 DFJ

λ scan for 5 city TSP												
	λ_1=0,1				λ_1=0,5				λ_1=1			
λ_0	avg cost	σ cost	avg ncost	σ ncost	avg cost	σ cost	avg ncost	σ ncost	avg cost	σ cost	avg ncost	σ ncost
0.6	-	-	-	-	-	-	-	-	-	-	-	-
0.7	-	-	-	-	-	-	-	-	-	-	-	-
0.8	62.33	0.00	1.52	0.00	62.33	0.00	1.52	0.00	62.33	0.00	1.52	0.00
0.9	62.33	0.00	1.52	0.00	62.33	0.00	1.52	0.00	62.33	0.00	1.52	0.00
1.0	62.33	0.00	1.52	0.00	62.33	0.00	1.52	0.00	62.33	0.00	1.52	0.00
2.0	62.33	0.00	1.52	0.00	62.33	0.00	1.52	0.00	62.33	0.00	1.52	0.00
5.0	62.33	0.00	1.52	0.00	62.33	0.00	1.52	0.00	62.33	0.00	1.52	0.00
10.0	62.33	0.00	1.52	0.00	62.33	0.00	1.52	0.00	62.33	0.00	1.52	0.00

Table (12): Lagrange factor scan results for the 5 city TSP solved with the DFJ formulation using D-Wave Hybrid.

λ scan for 10 city TSP												
	λ_1=0.1				λ_1=0.5				λ_1=1			
λ_0	avg cost	σ cost	avg ncost	σ ncost	avg cost	σ cost	avg ncost	σ ncost	avg cost	σ cost	avg ncost	σ ncost
0.6	-	-	-	-	-	-	-	-	-	-	-	-
0.7	-	-	-	-	-	-	-	-	-	-	-	-
0.8	-	-	-	-	91.55	0.00	2.11	0.00	91.55	0.00	2.11	0.00
0.9	-	-	-	-	91.55	0.00	2.11	0.00	91.55	0.00	2.11	0.00
1.0	-	-	-	-	91.55	0.00	2.11	0.00	91.55	0.00	2.11	0.00
2.0	-	-	-	-	91.55	0.00	2.11	0.00	91.55	0.00	2.11	0.00
5.0	91.55	n/a	2.11	n/a	91.55	0.00	2.11	0.00	91.55	0.00	2.11	0.00
10.0	91.55	n/a	2.11	n/a	92.39	1.68	2.14	0.05	92.59	1.42	2.15	0.05

Table (13): Lagrange factor scan results for the 10 city TSP solved with the DFJ formulation using D-Wave Hybrid.

λ scan for 15 city TSP												
	λ_1=0.1				λ_1=0.5				λ_1=1			
λ_0	avg cost	σ cost	avg ncost	σ ncost	avg cost	σ cost	avg ncost	σ ncost	avg cost	σ cost	avg ncost	σ ncost
0.6	-	-	-	-	-	-	-	-	-	-	-	-
0.7	-	-	-	-	-	-	-	-	-	-	-	-
0.8	-	-	-	-	-	-	-	-	-	-	-	-
0.9	-	-	-	-	104.59	0.00	2.13	0.00	104.59	0.00	2.13	0.00
1.0	-	-	-	-	104.59	0.00	2.13	0.00	104.59	0.00	2.13	0.00
2.0	105.45	1.21	2.15	0.04	105.90	1.28	2.17	0.04	106.14	1.70	2.18	0.05
5.0	110.66	2.68	2.32	0.09	108.53	3.29	2.25	0.10	110.01	3.92	2.30	0.12
10.0	111.72	3.82	2.35	0.12	109.15	3.53	2.27	0.11	111.28	3.90	2.34	0.12

Table (14): Lagrange factor scan results for the 15 city TSP solved with the DFJ formulation using D-Wave Hybrid.

λ scan for 20 city TSP												
	λ_1=0.1				λ_1=0.5				λ_1=1			
λ_0	avg cost	σ cost	avg ncost	σ ncost	avg cost	σ cost	avg ncost	σ ncost	avg cost	σ cost	avg ncost	σ ncost
0.6	-	-	-	-	-	-	-	-	-	-	-	-
0.7	-	-	-	-	-	-	-	-	-	-	-	-
0.8	-	-	-	-	-	-	-	-	-	-	-	-
0.9	-	-	-	-	134.71	0.32	2.96	0.01	134.67	0.25	2.96	0.01
1.0	-	-	-	-	134.66	0.36	2.96	0.01	134.78	0.31	2.97	0.01
2.0	142.73	7.17	3.22	0.23	139.89	2.68	3.13	0.08	139.16	3.42	3.10	0.11
5.0	179.81	5.15	4.38	0.16	160.35	8.59	3.77	0.27	160.45	12.29	3.77	0.39
10.0	184.89	15.25	4.54	0.48	180.00	12.70	4.39	0.40	176.22	13.04	4.27	0.41

Table (15): Lagrange factor scan results for the 20 city TSP solved with the DFJ formulation using D-Wave Hybrid.

λ scan for 20 city TSP												
	λ_1=**0.1**				λ_1=**0.5**				λ_1=**1**			
λ_0	**avg cost**	σ **cost**	**avg ncost**	σ **ncost**	**avg cost**	σ **cost**	**avg ncost**	σ **ncost**	**avg cost**	σ **cost**	**avg ncost**	σ **ncost**
0.6	-	-	-	-	-	-	-	-	-	-	-	-
0.7	-	-	-	-	-	-	-	-	-	-	-	-
0.8	-	-	-	-	148.36	0.08	2.75	0.00	-	-	-	-
0.9	-	-	-	-	152.55	n/a	2.87	n/a	152.19	1.76	2.86	0.05
1.0	-	-	-	-	153.27	1.54	2.89	0.04	155.41	2.67	2.95	0.07
2.0	-	-	-	-	176.91	6.86	3.55	0.19	176.39	6.35	3.54	0.18
5.0	209.41	8.44	4.46	0.24	-	-	-	-	-	-	-	-
10.0	221.33	31.23	4.80	0.88	224.90	9.07	4.90	0.25	236.40	13.71	5.22	0.38

Table (16): Lagrange factor scan results for the 25 city TSP solved with the DFJ formulation using D-Wave Hybrid.

References

[1] William J Cook. In pursuit of the traveling salesman: mathematics at the limits of computation. Princeton University Press, 2015.

[2] Ruprecht-Karls-Universität Heidelberg. Discrete and Combinatorial Optimization. 2018. URL: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/.

[3] James N MacGregor and Tom Ormerod. "Human performance on the traveling salesman problem". In: Perception & psychophysics 58 (1996), pp. 527–539.

[4] Scott M Graham, Anupam Joshi, and Zygmunt Pizlo. "The traveling salesman problem: A hierarchical model". In: Memory & cognition 28 (2000), pp. 1191–1204.

[5] Sriyani Violina. "Analysis of brute force and branch & bound algorithms to solve the traveling salesperson problem (TSP)". In: Turkish Journal of Computer and Mathematics Education (TURCOMAT) 12.8 (2021), pp. 1226–1229.

[6] Antima Sahalot and Sapna Shrimali. "A comparative study of brute force method, nearest neighbour and greedy algorithms to solve the travelling salesman problem". In: International Journal of Research in Engineering 2.6 (2014), pp. 59–72.

[7] Marco Dorigo, Vittorio Maniezzo, and Alberto Colorni. "Ant system: optimization by a colony of cooperating agents". In: IEEE transactions on systems, man, and cybernetics, part b (cybernetics) 26.1 (1996), pp. 29–41.

[8] Noraini Mohd Razali, John Geraghty, et al. "Genetic algorithm performance with different selection strategies in solving TSP". In: Proceedings of the world congress on engineering. Vol. 2. 1. International Association of Engineers Hong Kong, China. 2011, pp. 1–6.

[9] Chris Bernhardt. Quantum computing for everyone. MIT Press, 2019.

[10] Fred Glover, Gary Kochenberger, and Yu Du. "A tutorial on formulating and using QUBO models". In: arXiv preprint arXiv:1811.11538 (2018).

[11] Naeimeh Mohseni, Peter L McMahon, and Tim Byrnes. "Ising machines as hardware solvers of combinatorial optimization problems". In: Nature Reviews Physics 4.6 (2022), pp. 363–379.

[12] Andrew Lucas. "Ising formulations of many NP problems". In: Front. Phys. 2 (2014), p. 5. ISSN: 2296-424X. DOI: 10.3389/fphy.2014.00005. URL: https://www.frontiersin.org/article/10.3389/fphy.2014.00005.

[13] D-Wave Leap. What is quantum annealing? 2023. URL: https://docs.dwavesys.com/docs/latest/c_gs_2.html.

[14] D-Wave. QPU-Specific Characteristics. 2023. URL: https://docs.dwavesys.com/docs/latest/doc_physical_properties.html.

[15] George B. Dantzig, Richard M. Fulkerson, and Selmer M. Johnson. "Solution of a Large-Scale Traveling-Salesman Problem". In: Operations Research 2.4 (1954), pp. 393–410.

[16] JA Montanez-Barrera et al. "Improving Performance in Combinatorial Optimization Problems with Inequality Constraints: An Evaluation of the Unbalanced Penalization Method on D-Wave Advantage". In: arXiv preprint arXiv:2305.18757 (2023).

[17] R. W. Hamming. “Error detecting and error correcting codes”. In: The Bell System Technical Journal 29.2 (1950), pp. 147–160. DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[18] Laurens Van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE.” In: Journal of machine learning research 9.11 (2008).

[19] IBM. DOcplex Python Modeling API. 2021. URL: https://www.ibm.com/docs/en/icos/12.9.0?topic=docplex-python-modeling-api.

[20] D-Wave. Leap. 2019. URL: https://www.dwavesys.com/take-leap.

[21] PyPI. Python Client for Google Maps Services. 2023. URL: https://pypi.org/project/googlemaps/.

[22] Luis Gouveia and Stefan Voß. “A classification of formulations for the (time-dependent) traveling salesman problem”. In: European Journal of Operational Research 83.1 (1995), pp. 69–82.

[23] Matplotlib. Basemap Matplotlib Toolkit. 2016. URL: https://matplotlib.org/basemap/.

[24] Gerhard J. Woeginger. “Exact algorithms for NP-hard problems: A survey”. In: Eureka, You Shrink! Springer. 2003, pp. 185–207.

[25] Martin Orth. Deutschlands Lieblinge. 2020. URL: https://www.deutschland.de/de/topic/leben/was-die-deutschen-am-liebsten-moegen-deutschland-im-durchschnitt.

[26] Landgeist. MOST COMMON PLACE NAMES IN EUROPE. 2023. URL: https://landgeist.com/2023/02/04/most-common-place-names-in-europe/.

[27] Deutsche Post. POSTLEITZAHLENSUCHE. 2023. URL: https://www.postdirekt.de/plzserver/PlzSearchServlet?lang=en.