University of Utrecht
Faculty of Humanities

Utrecht University

Bachelor Thesis

submitted for the degree of

Bachelor of Science

The k-Cycling Dinner problem

and its placement in the computational complexity zoo

by

Niels Scholten
Student Nr.: 5967406

Submission Date:	February 22, 2021
Supervisors:	Natasha Alechina

1 Abstract

In this paper, I introduce a new family of scheduling and routing problems. These problems are called the k-cycling dinner problems. Given a graph, the problem is to find a set of tuples of nodes for which the distance between these nodes is minimized under the following three constraints. Each node is present in exactly k tuples. Each pair of tuples is not allowed to share more than one node, and each node should have the same position within the tuple in all the tuples in which that node in present. I provide a constraint satisfaction program which is able to solve these problem, however this solution is not sufficiently fast. After explaining the necessary terms on Np-Completeness, I provide an NP-Completeness reduction proof for the 2-Cycling Dinner problem, which proves that the 2-Cycling Dinner problem is in the complexity class NP-Complete. This suggests that finding a solution to this problem would be very hard to do in polynomial time. This paper also contains an attempt to provide an NP-Completeness reduction proof for the 3-CD problem. These proofs offer strong arguments for researchers trying to solve these problems, to utilise approximation algorithms.

Contents

2 Introduction

Scheduling and routing problems have been a large topic in the field of AI since the 1950s, with McCarthy's Advice Taker [1] being one of the first. With the complexity of many scheduling and routing tasks on large scale, the help of an intelligent system is often needed to find an appropriate solution. However, in the field of computational complexity, certain problems have been proven to be impossible to solve in reasonable time, even with the help of intelligent systems. Within this field the choice has been made to try to categorize these problems into classes based on their complexity. In 2000, one of the seven Millennium Prize Problems concerned two of these complexity classes [2]. The task of the Millennium Prize Problem was to provide a proof or a counterproof on whether P = NP, which roughly translates to, is every problem for which the solution can be verified reasonably fast, also solvable reasonably fast? This question has not yet been answered, but the general consensus between researchers is that P $\neq$ NP [3]. In pursuit of an answer to this question, the class NP-Complete was created. This class consists of problems which are proven to be as difficult as the most difficult problems in NP. In this paper, I will introduce a family of scheduling and routing problems, and will prove every member of this family is part of NP and that the seemingly least difficult problem in this family is NP-Complete. This suggests, but does not prove, that the entire family of these problems is NP-Complete. As a reasonably fast solution to a NP-Complete problem has never been found, the proof in this paper functions as an argument to convince researchers trying to solve a problem of this family to pursue other means of solving it, such as approximation algorithms.

3 Initial problem

In this section, I will describe how I came across the inspiration for the k-cycling dinner problem; the scheduling and routing problem which is the topic of this paper.

I was asked to organize an event, a cycling dinner, with the purpose of getting a large amount of people acquainted. For a cycling dinner, each participant is asked to prepare either an appetizer, main dish or a dessert. During the cycling dinner, each participant will have a meal at three different locations, one of which is their own home. Each meal will be shared with 2 other participants and you will never share a meal with the same person twice. In between meals, the participants will be cycling from one location to another. In order to organize this event efficiently, it is preferred that the distance that the participants have to travel is as small as possible. Finding an optimal path for each participant appeared to be a very challenging problem.

4 Problem definition

4.1 Optimization problem

The problem found in the last section is one of a family of problems. These problems will be called k-Cycling Dinner problems The example in the last section would be the 3-Cycling Dinner problem. The k-Cycling Dinner problem can be written as following.

Given a weighted undirected graph $G = (V, E)$ and an amount k, find an unordered collection $F = \{m_1, m_2, \ldots, m_n\}$ of tuples of elements from V such that $\forall_i |m_i| = k$ under the following restrictions:

1. Each vertex $v \in V$ is present in exactly k elements of F.

$$\forall_v(\exists_A((|A| = k) \wedge \forall_m(v \in m \rightarrow m \in A)))$$

2. Each pair of tuples m_a, m_b is not allowed to share more than one vertex.

$$\forall_{m \in F}\forall_{n \in F}\forall_{i \in V}\forall_{j \in V}((i \in m \wedge j \in m \wedge i \neq j \wedge n \neq m) \rightarrow \neg(i \in n \wedge j \in n))$$

3. each vertex $v \in V$ within each tuple should occur at the same index at all tuples in which it is present.

$$\forall_v((\exists_m m[i] = v) \rightarrow (\forall_m v \in m \rightarrow m[i] = v))$$

which minimizes $\sum_{m \in F} \sum_{i<k} w(m[i], m[i+1])$

4.2 Decision problem

This optimization problem also has an accompanying decision problem for the following form. The k-Cycling Dinner decision problem is to, given a graph $G = (V, E)$ and a value d, determine whether an unordered collection $F = \{m_1, m_2, \ldots, m_n\}$ of tuples of elements from V for which $\sum_{m \in F} \sum_{i<k} w(m[i], m[i+1]) < d$ exists, following the constraints mentioned above.

5 Constraint Satisfaction Problem

In my pursuit for a solution to the k-Cycling Dinner problem, I chose to formulate it as an constraint satisfaction problem, as this suits the description of the problem well. [4] A constraint satisfaction problem consists of three components, X, D and C:

1. X is a set of variables $\{X_1, \ldots X_n\}$

2. D is a set of domains $\{D_1, \ldots, D_n\}$

3. C is a set of constraints that specify allowable combinations of values

In my k-Cycling Dinner problem, the set of variables C would be the set of tuples $F = \{m_1, m_2, \ldots, m_n\}$. The domains would be the different nodes V from the graph $G = (V, E)$. The set of constraint would be the three constraints mentioned in the problem definition, in combination with the minimization term. I have implemented this constraint satisfaction problem in a constraint programming language. This code can be seen in appendix A. When running this code on small examples such as a 3-CD problem with 9 nodes, I noticed that the solution was only found after several hours. This made me suspect that this problem might not be solvable in polynomial time and I continued to try to prove this problem to be NP-Complete.

6 NP-Completeness

To understand the content of this paper, a basic understanding of NP-Completeness and the field of computational complexity is needed. Therefore, I will explain the necessary terms in this section. The field of computational complexity is concerned with the analysis of the complexity of an algorithm, and especially with the amount of resources, such as for example memory or time, needed to run it. When analysing the amount of time needed to run an algorithm, this amount is often expressed in rate of growth. This represents the speed with which the time needed to run the algorithm grows, in relation to the size of the problem. Often this is expressed using the big O( ) notation. Suppose we were to express the time needed to solve a problem of size n in a function $f(n)$. If the form of the function f were that of a polynomial, we would conclude that this algorithm would be able to run in polynomial time.

In computation complexity, the class which we call P consists of all problems which can be solved by an algorithm in polynomial time. This class is a subset of a bigger class, called NP. NP consists of all problems for which, given a solution, this solution can be verified in polynomial time. As of this date, we have no formal proof whether these two groups are identical or not [3].

In 1971, Stephen Cook proved in his paper "The complexity of theorem-proving procedures" [5] that one certain problem, known as the 3-Satisfiability problem (abbreviated as 3-SAT), is as least as difficult as all other problems in NP. He provided this problem with its own group called NP-Complete. If an algorithm would be found that could solve this problem in polynomial time, that would proof that P = NP.

A subsequent paper by Richard Karp [6], shows that 3-SAT can be rewritten as 21 different known problems. This proved that these problems were as least as difficult as 3-SAT and are part of the group NP-Complete. Since the introduction of NP-Completeness, researchers have been highly motivated to search for an algorithm which would be able to solve one of these NP-Complete problems in polynomial time, however none have been successful. Therefore it is widely believed that these problems can not be solved in polynomial time.

7 Reduction proofs

In the book Computers and Intractability [7], Garey and Johnson laid out a roadmap on how to prove that a problem Π belongs to the group NP-Complete. This roadmap contained the following steps.

1. Show that Π is in NP
2. Select a known NP-Complete problem Π'
3. Construct a transformation f from Π' to Π
4. Proof that f is a polynomial transformation

These steps provide a problem with an upper bound an a lower bound on the complexity. In this section, I will further explain these steps, and how successfully performing theses steps results in a proof that problem Π belongs to the group NP-Complete. I will do this by going over every step and explaining what has to be done in each step, and how this contributes to the overall proof.
To proof that Π belongs in NP, which is the first step, is to proof that there exists an algorithm which is able to verify whether a solution S to problem Π is correct. This can be done by providing an algorithm which can do this verification. If one were to find an algorithm which could verify the accompanying decision problem in polynomial time, this would also suffice. This is because, this algorithm could first be run with a value of 1 and iterate up, until the lowest possible value is found. As this is a chain of polynomial time algorithms, this can be done in polynomial time. Hereby proving that the problem is in NP. By proving that the problem is in NP, you provide it with an upper bound.
The second step involves selecting a known NP-Complete problem Π'. the 3-Satisfiability problem is such a problem, but one can also choose to select one of Karps' 21 problems [6] or any other problem which is proven to be in NP-Complete by the method described in this section.
The third step of the reduction is to provide a transformation f from Π' to Π. A transformation is a method of translating a problem Π' to a problem Π and, after Π has been solved, translating that solution of Π to a solution for Π'. In this step, it is important to note that every particular instance of problem Π' should be able to be translated via the transformation f. Secondly, it is important that the answer to the translated problem $f(\Pi')$ corresponds to the answer for the original problem Π'.
The final step, is to prove that transformation f can be performed in polynomial time. The function of this step, in combination with the third step, is to provide the lower bound. If one were to find a solution to problem Π. This solution, in combination with the polynomial transformation

f, would proof that the known NP-Complete problem Π' would be able to be solved in polynomial time. The problem Π' would be able to be solved by

1. translating the problem Π' to Π (in polynomial time)
2. solving problem Π (in polynomial time)
3. translating the solution to a solution for the original problem (in polynomial time).

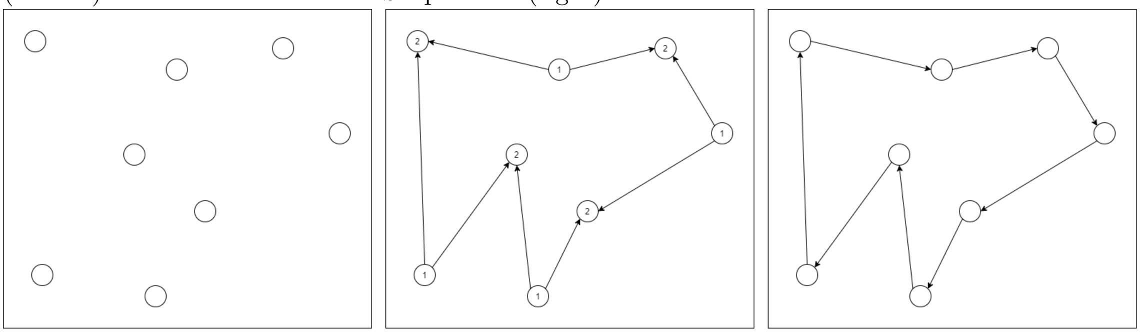
This would prove that $P = NP$.

8 2-CD Reduction

8.1 Intuition

In this paragraph, I will show the intuition behind the polynomial time reduction proof from the Euclidean Traveling Salesman Problem to the 2-Cycling Dinner problem. In the Euclidean Traveling Salesman Problem, you are given a roadmap. The task is to find a path which is as short as possible, while crossing every city. Say we would create a 2-CD problem where every node in the graph corresponds to a city and all the edges correspond to the distances between the two cities. We know that, if we have a solution for the 2-CD problem, for every node, there is are exactly 2 paths leading either to it or from it. Therefore, If we were to connect all of these paths together, we would get a path which visits every node.

Figure 1: Representation of a TSP problem (left), a solution to the corresponding 2-CD problem (middle) and a solution to the TSP problem (right)



8.2 Show that Π is in NP

To prove the 2-CD problem (further denoted as Π) is in NP, I will provide an algorithm which, given a solution of the form $S = \{m_1, m_2, \dots m_n\}$ for which $\forall_i |m_i| = 2$ checks whether $\sum_{(x,y) \in S} w(x, y)$ is minimized under the constraints specified in the section on the formal problem definition. In order to do this, i will provide an algorithm which is able to verify whether a solution for the decision problem of 2-CD is correct in polynomial time. This algorithm can then be used multiple times, in the way described in the section on reduction proofs, to construct an algorithm for verifying the optimization problem. The pseudo-code for the algorithm can be found in appendix B. As you can see, this algorithm has a polynomial runtime. Therefore, I have proven the 2-CD problem to be in NP.

8.3 Select a known NP-Complete problem Π'

The NP-Complete problem that I will use is the Euclidean Traveling Salesman Problem (abbreviated with Euc TSP) [8]. In this problem, a set of cities $C = \{c_1, c_2, \ldots, c_n\}$ is given, with a distance function $d : C \times C \to \mathbb{R}$. The objective is to find a tour, that is, a simple path visiting all cities, so that the total distance traveled is the least possible.

8.4 Construct a transformation f from Π' to Π

Given an instance of an Euc TSP problem with a finite set $C = \{c_1, c_2, \ldots, c_m\}$ and a distance function $d : C \times C \to \mathbb{R}$. First I will describe how to transform an Euc TSP problem to a 2-CD problem. To transform an Euc TSP problem to a 2-CD problem, Create a graph $G = (V, E)$ where $V = C$ and $E = C \times C$ and $w : V \times V \to \mathbb{R} = d : C \times C \to \mathbb{R}$. I have to prove that the resulting 2-CD problem is solvable.

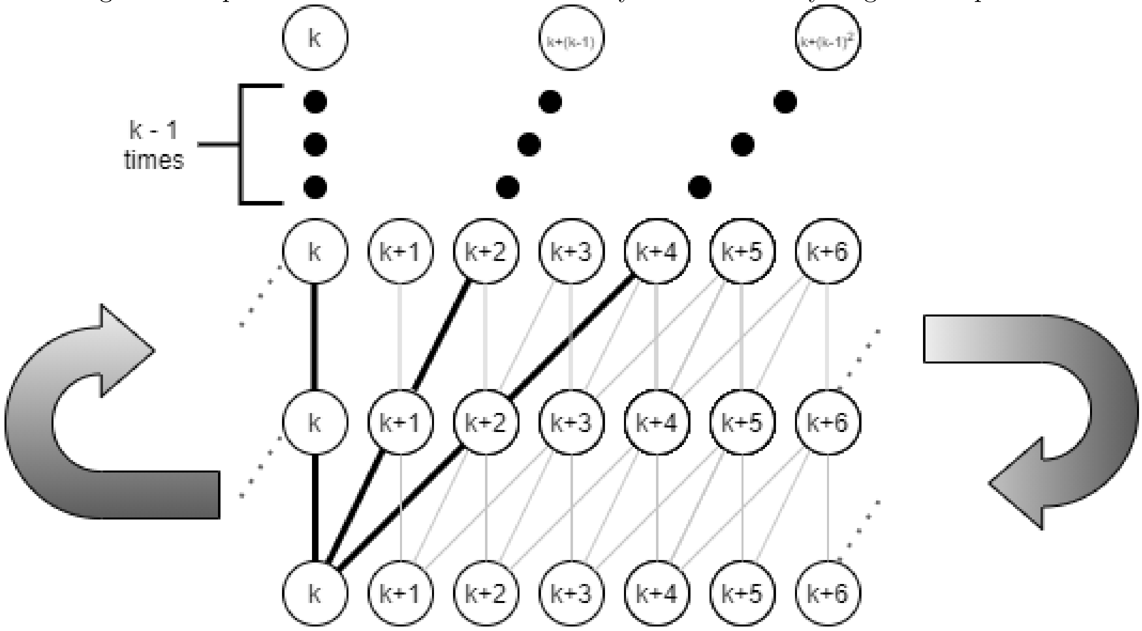
Theorem: For every k-Cycling Dinner problem where the graph $G = (V, E)$ is fully connected $E = V \times V$. If $|V| \geq k \times k \times (k-1)$ and $|V|$ is divisible by k, Then there is a solution for the problem.

Proof:First, divide the nodes V evenly in k rows and as much columns as needed. Note that you will have at least $k-1$ columns. I will now refer to a node in row i column j as $n_{(i,j)}$. Denote all tuples in which node $n_{(i,j)}$ is present as $m_0, m_1, \ldots m_{k-1}$. In order to construct all these tuples, follow the following steps. $\forall_j \forall_{z \in [0,k]} m_z = ((1, j+0z), (2, j+1z), (3, j+2z), \ldots, (k, j+(z-1)z))$. If an column index is larger than the amount of columns, this will loop around back to $j = 1$. Notice that, the path in this set of paths that reaches the furthest to the left is m_0. This path reaches column j. The path in this set of paths which reaches the furthest to the right is m_{k-1}. This path reaches column $j + (k-1)(k-1)$. We know that there are at least $(k-1)^2$ columns, therefore can no path with n in it can every cross another path with n in it.

Now that we know that each 2-CD problem where $|V|$ is divisible by 2 is solvable, we need to make sure that the transformed problem has an even amount of nodes. In order to translate an Euc TSP problem where the amount of cities is not a multitude of 2, one can introduce an extra node. This node n has the following properties. Take an arbitrary node n_0 from the problem and add the extra node n such that $((n, n_0) \in E) \wedge (d(n, n_0) = 0) \wedge (\forall_x ((n_0, x) \in E) \to ((n, x) \in E))$

Secondly, I will describe how to transform a 2-CD solution to an Euc TSP solution. When given a solution to the 2-CD problem of the from $F = \{m_1.m_2, \ldots, m_n\}$, were each tuple has size two.

Figure 2: Representation of a solution to a fully connected k-Cycling Dinner problem



For the sake of clarity, I will denote the first element of a tuple m as $m^{(1)}$ and the second element of a tuple m as $m^{(2)}$.

Create an ordering $< c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)} >$ where $c_{\pi(1)} = m_1^{(1)}$ and

$$c_{\pi(k)} = \begin{cases} m_x^{(1)} & \text{if } c_{\pi(k-1)} = m_x^{(2)} \wedge m_x^{(1)} \notin < c_{\pi(1)}, \ldots, c_{\pi(k)} > \\ m_x^{(2)} & \text{if } c_{\pi(k-1)} = m_x^{(1)} \wedge m_x^{(2)} \notin < c_{\pi(1)}, \ldots, c_{\pi(k)} > \end{cases}$$

Lastly, I will provide a proof that the solution found by the application of this transformation yields an optimal solution. This will be a proof by contradiction.

Suppose that there is a solution $S^* =< c_{\pi^*(1)}, c_{\pi^*(2)}, \ldots, c_{\pi^*(n)} >$ for the Euc TSP which is better than the solution $S =< c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)} >$ found using the transformation. This means that $\sum_{i=2}^{n} d(c_{\pi^*(i-1)}, c_{\pi^*(i)}) < \sum_{i=2}^{n} d(c_{\pi(i-1)}, c_{\pi(i)})$ This solution S^* could be translated to a solution for the corresponding 2-CD problem by performing the inverse of the transformation explained above. This translated solution $f^{-1}(S^*)$ would yield a better solution than the solution to the 2-CD problem which was found. As the 2-CD solution found is by definition the optimal solution, we find a contradiction. Therefore, there is no solution $S*$ which is better than the solution S found by the algorithm, and the solution S is optimal.

8.5 Proof that f is a polynomial transformation

For this transformation, all node should be copied. This takes $O(n)$ time. furthermore, for each combination between nodes an edge should be created, which takes $O(n^2)$ time. Lastly, a single additional variable might have to be created. This would take $O(1)$ time.

The transformation from the solution from the 2-CD problem to the solution from the Euc TSP would take $O(n)$ time. Therefore, the transformation f can clearly be performed in polynomial time.

9 Representational gap

It is important to recognize that the formulation of the problem as a graph leaves open the possibility of constructing graphs which would be hard to translate to the real world application of minimizing distance. This is because in most real scenarios, distance between points follows certain restrictions, such as the triangle inequality. This inequality states that, to move from a point A to a point C, one would have to travel no more distance that to move from A to B and to move from B to C. As the set of k-CD problems which holds true to the triangle inequality is a strict subset of the set of k-CD problems, the proof that a k-CD problem is NP-Complete does not exclude the possibility that the subset of k-CD which hold true to the triangle inequality is easier than the k-CD problem to an extent that it is no longer NP-Complete. Note that, the reduction provided in this paper also applies to the 2-CD problem following the euclidean restrictions. This is because the graph created from the cities from the Euc TSP adhere to the euclidean restrictions.

10 Complexity of the k-CD family

10.1 NP

The algorithm given in section 8.2 is able to verify solutions for k-CD problems of any size k in polynomial time. The existence of this algorithm proofs that each member of the k-CD family is in NP.

10.2 3-CD

During the course of this research, I have tried to proof that 3-CD is in NP-Complete. Unfortunately, I have not been successful. I suspect that this is not a consequence of 3-CD not being in NP-Complete. On the contrary, I suspect 3-CD to be complex to the extent that the similarities between 3-CD and known NP-Complete problems are difficult to recognize. This made finding a reduction proof challenging.

10.3 Method of reducing to 3-CD

In the rest of this section, I will describe the general method with which I have tried to reduce a known NP-Complete problem to 3-CD.

10.3.1 Show that Π is in NP

I have already proven that 3-CD is in NP, by providing the algorithm from section 8.2.

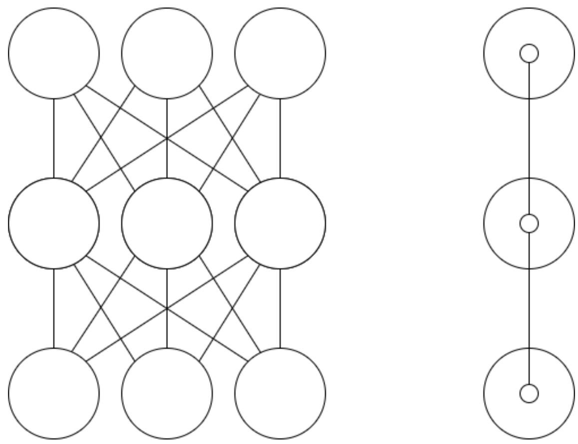
10.3.2 Select a known NP-Complete problem Π'

The NP-Complete problem I had selected is the Three Dimensional Matching problem (abbreviated with 3DM) [6]. This problem is described as the following:
given a set $M \subseteq W \times X \times Y$, where W, X and Y are disjointed sets having the same number of elements. Find the maximal matching for M, i.e. a subset $M' \subseteq M$ such that $|M|$ is maximized and no two elements of M' agree in any coordinate.

10.3.3 Construct a transformation f from Π' to Π

To clearly explain the transformation f, I first have to define some notation. I introduce clusters, which are sets of 3 nodes. When two clusters c_1 and c_2 are fully connected, denoted as $(c_1 \boxtimes c_2)$, this means that $\forall_{x \in c_1} \forall_{y \in c_2} ((x, y) \in E)$.

Figure 3: Representation of 3 clusters. With all nodes shown (left) and illustrated as 3 clusters (right)



Furthermore, I would like to introduce a gadget. This gadget would be a subgraph of the 3-CD problem which would correspond to a tuple in M. This gadget is shown below. All edges shown in the picture have a weight of 1. In this gadget, the three most outer clusters would correspond with the three objects in the tuple. I will refer to these as object clusters. The purpose of this gadget would have been to have two possible states. In one state, the closed state, the object clusters are assigned to the clusters inside of the gadget. In the other state, the open state, all clusters in the gadget would have been assigned except the object clusters. Unfortunately, this gadget has more states that just these two. However, I will continue the proof as if these gadgets only have these two states, to show how that would lead to an NP-Complete reduction.

Note that the object clusters are shared between gadgets for which the corresponding tuples share

Figure 4: Representation of a gadget

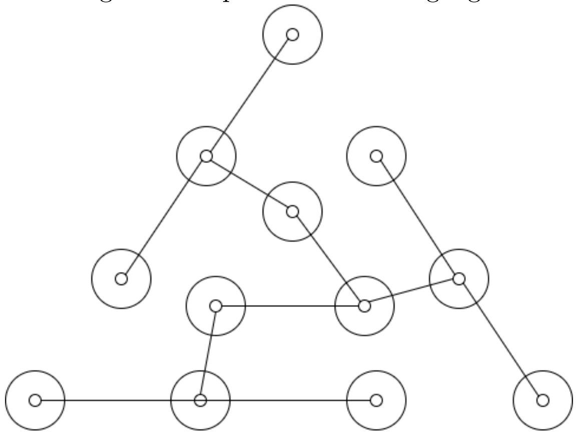
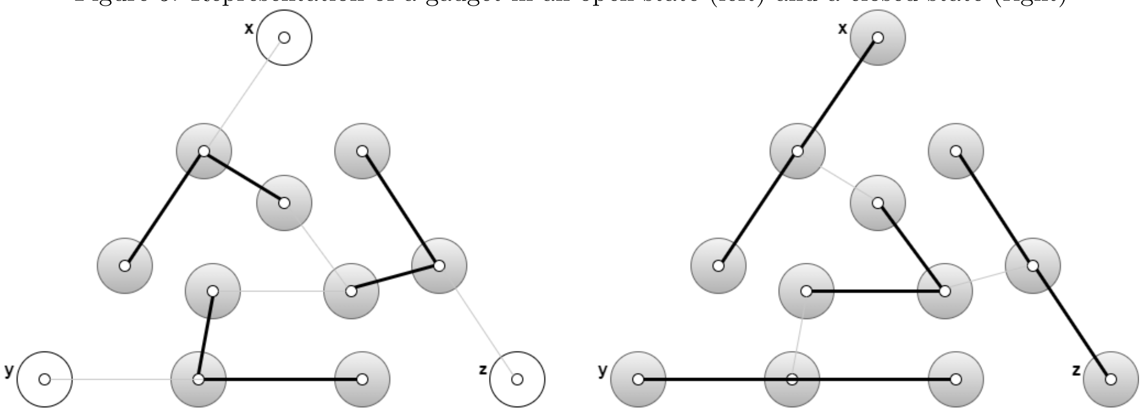


Figure 5: Representation of a gadget in an open state (left) and a closed state (right)



an object. Furthermore, all object clusters are connected via edges with weight w which is larger than the sum of all the weight in the gadgets. This is to provide a way of assigning all objects which are not part of the maximal matching, while discouraging the use of these edges.
These gadgets simulate the 3DM problem as:

1. the gadgets enforce that either all objects cluster of a gadget are assigned or none.

2. the gadget enforces that the object clusters can not be assigned to multiple gadgets.

3. In the solution, as much gadgets as possible should be in closed state.

As the 3DM problem has the constraints:

1. The solution consists of tuples which are either full in the matching or not.

2. no two elements of M' are allowed to be the same.

3. The size $|M|$ has to be maximized

If this gadget would have only had these two states, then this would prove that 3-CD is in NP-Complete. Unfortunately, that is not the case and therefore this proof does not hold up.

11 Discussion

In this paper, I have proven that 2-CD is part of the class NP-Complete. This implies that finding an algorithm which would be able to solve this problem in polynomial time would either be impossible (in the case that $P \neq NP$) or extremely difficult (in the case that $P = NP$). In both cases, I would suggest researchers trying to solve this problem to use approximation algorithms. One would intuitively think that the k-CD problem for any $K > 2$ would also be in NP-Complete as they seem to be more complex. Note that it is not proven in this paper that these problems are in NP-Complete. However, if one would accept the notion that k-CD problems with a higher k are not strictly less complex than 2-CD, than that would lead to the conclusion that these problems are also in NP-Complete. Further research for finding a reduction proof for others members of this family of problems could proof this fact. For 3-CD especially, finding a gadget which has the characteristics described in section 10.3.3. could lead to a reduction proof. For researcher trying to solve this problem for a real world application, it might be useful to use a different term to minimize. One could for example choose to try to make sure that no participant would have to travel more than d distance. At this moment, nothing can be said on whether this variation of the k-CD problem is in NP-Complete.

References

[1] J. McCarthy, "Programs with common sense," 1960.

[2] J. A. Carlson, A. Jaffe, and A. Wiles, *The millennium prize problems*. American Mathematical Society Providence, RI, 2006.

[3] L. Fortnow, "The status of the P versus NP problem," *Commun. ACM*, vol. 52, no. 9, pp. 78–86, 2009.

[4] S. Russell and P. Norvig, "Artificial intelligence: a modern approach," 2002.

[5] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158, 1971.

[6] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*, pp. 85–103, Springer, 1972.

[7] M. R. Garey and D. S. Johnson, *Computers and intractability*, vol. 174. freeman San Francisco, 1979.

[8] C. H. Papadimitriou and P. CH, "The euclidean traveling salesman problem is np-complete.," 1977.

Appendices

A Constraint programming code

The code in this appendix is written for the MiniZinc constraint programming language, but could be used for many other constraint programming languages after minor changes.

```
int : n;
int : M;
array[1..n, 1..n] of 0..M: d;

array[1..n, 1..n] of var 0..3: personal_route;

constraint forall (i in 1..n) (personal_route[i,i] > 0); % Everybody eats at home

 % every location has 3 guests
constraint forall (x in 1..n) (sum(p in 1..n where personal_route[p,x] > 0) (1) = 3);
constraint forall (p in 1..n) (sum(x in 1..n) (personal_route[p,x]) = 6); % everybody has 3 dinners

% Everybody has one of each dinners
constraint forall (p in 1..n) (count(personal_route[p,..],1,1));
constraint forall (p in 1..n) (count(personal_route[p,..],2,1));
constraint forall (p in 1..n) (count(personal_route[p,..],3,1));

%All the dinners at one place are at one time
constraint forall (x in 1..n) (count(personal_route[..,x],1,3) \/ count(personal_route[..,x],1,0));
constraint forall (x in 1..n) (count(personal_route[..,x],2,3) \/ count(personal_route[..,x],2,0));
constraint forall (x in 1..n) (count(personal_route[..,x],3,3) \/ count(personal_route[..,x],3,0));

%calculating total distance
array[1..n,1..3] of var int: personal_location;
constraint forall (p,j in 1..n where personal_route[p,j] = 1) (personal_location[p,1] = j);
constraint forall (p,j in 1..n where personal_route[p,j] = 2) (personal_location[p,2] = j);
constraint forall (p,j in 1..n where personal_route[p,j] = 3) (personal_location[p,3] = j);

array[1..n] of var int: personal_distance;
constraint forall (p in 1..n) (sum(i in 2..3)
                    (d[personal_location[p,i-1],personal_location[p,i]])
                    == personal_distance[p]);
var int: total_distance = sum(personal_distance);
```

```
%Not allowed to meet each other twice
constraint forall(p1,p2 in 1..n where p1 != p2 /\ personal_route[p1,p2] > 0)
    (sum(loca in 1..n) (personal_route[p1,loca] * personal_route[p2,loca]) == 9 \/
     sum(loca in 1..n) (personal_route[p1,loca] * personal_route[p2,loca]) == 4 \/
     sum(loca in 1..n) (personal_route[p1,loca] * personal_route[p2,loca]) == 1 );

solve minimize total_distance;
```

B Verification algorithm pseudo-code

Algorithm 1: Verification algorithm for k-Cycling Dinner

```
Result: True or False
forall tuple m in F do
    if length of m ≠ k then
        return False
    end
end
Array index;
forall tuple m in F do
    for int i ← 1 to k do
        if index[m[i]] ≠ i ∧ index[m[i]] ≠ null then
            return False
        end
        index[m[i]] = i;
    end
end
Dictionary of Lists paths;
forall tuple m in F do
    for int i ← 1 to k do
        add m to paths[m[i]]
    end
end
forall tuple m in F do
    for int i ← 1 to k do
        if m[j] in paths[m[i]] then
            return False
        end
    end
end
Float distance = 0;
forall tuple m in F do
    for int i ← 1 to k − 1 do
        distance = distance + d(m[i],m[i+1]);
    end
end
if distance ¿ d then
    return False ;
end
return True ;
```