A Fixed-Parameter Tractable Algorithm for Spatio-Temporal Calendar Management

Bernhard Nebel
Institut für Informatik
Albert-Ludwigs Universität Freiburg
79110 Freiburg, Germany

Jochen Renz
Research School of Information Sciences and Engineering
The Australian National University
Canberra ACT 0200, Australia

Abstract

Calendar management tools assist users with coordinating their daily life. Different tasks have to be scheduled according to the user preferences. In many cases, tasks are at different locations and travel times have to be considered. Therefore, these kinds of calendar management problems can be regarded as spatio-temporal optimisation problems and are often variants of traveling salesman problems (TSP) or vehicle routing problems. While standard TSPs require a solution to include all tasks, prize-collecting TSPs are more suited for calendar management problems as they require a solution that optimises the total sum of "prizes" we assigned to tasks at different locations. If we now add time windows that limit when tasks can occur, these prize-collecting TSPs with time windows (TW-TSP) are excellent abstractions of spatio-temporal optimisation problems such as calendar management. Due to the inherent complexity of TW-TSPs, the existing literature considers mainly approximation algorithms or special cases. We present a novel algorithm for TW-TSPs that enables us to find the optimal solution to TW-TSP problems occurring in real-world calendar management applications efficiently. Our algorithm is a fixed-parameter tractable algorithm that depends on the maximal number of tasks that can be revisited from some other task, a parameter which is small in the application scenario we consider.

1 Introduction

The motivation for the research described here comes from the problem of calendar management [Jennings and Jackson, 1995; Modi *et al.*, 2004; Sen and Durfee, 1994], whereby we consider the situation that we have an over-subscription problem, i.e., we have more tasks on our agenda than we can possibly handle. Furthermore, we also take space into account, which is usually ignored in calendar management. Space becomes important when

- tasks have to take place at specific locations,
- there is significant travel time between locations, and
- tasks can be done while travelling, e.g., on the train.

Interestingly, space can in this context be reduced to travel time between two tasks. If there are different modes of transportation, then costs may also factor in.

A typical scenario, we envisage, is the following. You plan to find a new apartment. For this purpose, you find out from the ad section of the local news paper, which apartments are available, where they are located, and when are the visiting times. Now, you want to visit as many apartments as possible. Clearly, there are too many and because of the restricted visiting times, you will not be able to visit them all. However, by scheduling your visits carefully, e.g., by not wasting time going back and forth between different neighbourhoods, you may be able to visit a large number.

From a formal point, this calendar management problem can be seen as the *prize-collecting travelling salesman problem with time-windows* [Bar-Yehuda *et al.*, 2005], a scheduling problem that takes also space into a account. Since the problem is a generalisation of the TSP, it is, of course, NP-hard. While there have been a number of attempts to design approximation algorithms (e.g. [Zhang and Tang, 2007; Shi *et al.*, 2008]), exact algorithms that are efficient in real-world cases have not been considered so far. We provide the first exact algorithm for the unit-prize case that is *fixed-parameter tractable* [Niedermeier, 2006], i.e. polynomial with degree independent of the parameter, when the parameter is fixed. In our case, the parameter is the maximal number of jobs that can be revisited from some other job. We further show that when generalising to non-unit-prizes one can use a pseudo-polynomial algorithm.

The rest of the paper is structured as follows. The following section introduces terminology and notation around the prize-collecting travelling salesman problem with time-windows. After that we discuss approaches to this problem that have been attempted so far. Then we introduce the notion of fixed-parameter tractability. Based on that, we specify a dynamic-programming algorithm that solves the problem exactly and that is fixed parameter tractable. Finally, we discuss the applicability of our algorithm and conclude.

2 Problem description

In this section, we formally define the Prize-Collecting Travelling Salesman Problem with Time-Windows (TW-TSP)

[Bar-Yehuda *et al.*, 2005]. We first introduce all the components we need. For a travelling salesman problem (TSP) we have given a set V of *cities* (called *jobs* in scheduling terminology and in this paper) at given locations in a metric space (V,ℓ) where ℓ is a metric. This could be a planar map or the surface of the earth. What is interesting to us is the non-symmetric *travel time* $\ell(u,v)$ it takes to travel from job u to job v. Note that a metric space satisfies the triangle inequality, i.e., $\ell(u,v) \leq \ell(u,w)+\ell(w,v)$ for all $u,v,w \in V$. In other papers in the literature, the location of jobs is specified in terms of coordinates and the distance between jobs is calculated as the shortest distance according to the given metric. In this paper we consider a TSP simply as a directed graph $G=(V,E)$ where the jobs are nodes and the edges correspond to the travel time between nodes. In the prize-collecting TSP, each job $v \in V$ has a *prize* $p(v) \in \mathbb{N}$ assigned to it that specifies the profit (a natural number) we get when performing job v. In general, this allows us to give different profits to different jobs or to specify preferences. In this paper, however, we are mostly dealing with unit prizes, i.e., $p(v)=1$ for all $v \in V$. In addition, each job has a *processing time* $h(v)$ which is the minimum time we have to spend for job v. For the TSP with time windows, each job is also assigned a *time window* $I_v=[r(v),d(v)]$, which is a time interval with *release time* $r(v)$ and *deadline* $d(v)$ during which a job v has to be started and finished. We can assume that $h(v) \leq |d(v)-r(v)|$ and that $r(v) \leq d(v)$. A TW-TSP instance over the metric space (V,ℓ) consists of:

- A subset $S \subseteq V$ of jobs.
- Each element $s \in S$ is assigned $p(s), h(s), r(s)$, and $d(s)$.
- *Origin* $v_0 \in S$ with $p(v_0)=h(v_0)=r(v_0)=d(v_0)=0$.

A *tour* is a sequence of points (v_i,t_i), where t_i is the *arrival time* at point v_i and v_0 is the origin. A feasible tour $\{(v_i,t_i)\}_{i=0}^{k}$ must satisfy the following conditions: $t_0=0$ and $t_{i+1} \geq t_i+h(v_i)+\ell(v_i,v_{i+1})$.

A solution to a TW-TSP instance (a *TW-tour*) is a tour $\{(v_i,t_i)\}_{i=0}^{k}$ such that each $v_i \in S$, every job occurs at most once, and for every $0 \leq i \leq k$, we have that $t_i \geq r(v_i)$ and $t_i+h(v_i) \leq d(v_i)$. The *profit* of a TW-tour $\mathcal{T}=\{(v_i,t_i)\}_{i=0}^{k}$ is the sum of the profits of all participating jobs, i.e., $p(\mathcal{T})=\sum_{i=0}^{k} p(v_i)$. The *optimal solution* to a TW-TSP instance is a TW-tour with maximum profit.

3 Background and previous work

There are many variants of the TSP which are similar to TW-TSPs and which have different optimality criteria. For standard TSPs, a feasible tour has to visit all cities and the task is to find the shortest tour that visits all cities. With additional time-windows, it is clear that some instances cannot be solved if all cities have to be visited. Given that an instance of a TSP with time windows has a solution, the most common optimisation criteria is again to minimise the arc-traversal cost, which corresponds to the shortest path [Tsitsiklis, 1992]. Another optimality criterion is to minimise the makespan, i.e., to minimise the time to return to the origin, which is often called the *depot* for vehicle routing problems (VRP). VRPs are another variant where there are sometimes multiple vehicles available which together have to visit all cities during the given time windows [Danzing and Ramster, 1959]. Variants of TSPs with time-windows where not all cities have to be visited are often called *deadline TSPs*, or *Orienteering TSPs* if all deadlines are the same. In the context of VRPs, it is common to refer to these TSPs as prize-collecting TSPs [Balas, 1989], which we are considering in this paper.

All these variants of the TSP are optimisation problems that are at least NP-hard. It is therefore common to consider approximation algorithms that provide a fast approximation to the optimal solution and heuristics. The quality of an approximation algorithm is usually measured by the approximation ratio, i.e., the factor by which the approximate solution differs from the optimal solution. Many approximation algorithms are based on dynamic programming, which gives solutions very fast, but doesn't allow to store path information about how the intermediate results during the run of the algorithm are achieved. The major disadvantage of not being able to consider history of paths is that it is impossible to test if a job occurs more than once in a path. Bar-Yehuda et al (2005), introduce a density measure of a TW-TSP problem instance that limits the number of zig-zags in a tour, i.e., how often it is possible to go from one job to another and vice-versa within the time windows of both jobs. Bar-Yehuda et al define *density* of a TW-TSP instance Π:

$$\sigma(\Pi)=\max_{u,v}\frac{|I_u|}{\ell(u,v)+\ell(v,u)+h(u)+h(v)}.$$

If the density of an instance is below 1, i.e., if it is not possible to visit a job twice within the same time-window, then the dynamic programming algorithm they present computes the optimal solution of Π. In general, the approximation ratio of their algorithm is $\lfloor \sigma(\Pi) \rfloor + 1$ for the TW-TSP problem without processing times.

We are not interested in approximation algorithms, but in algorithms that compute the optimal solution to the TW-TSP problem as efficient as possible. Since it is an NP-hard problem, our algorithm cannot be tractable (given P $\neq$ NP), but we will present a fixed-parameter tractable algorithm instead.

For this purpose, we use the standard definition of *fixed-parameter-tractable problem* [**?**]. A *parameterised problem* (over the alphabet Σ) is a pair (Q,κ) consisting of a set $Q \in \Sigma^*$ of strings over Σ and an arbitrary function $\kappa : \Sigma^* \to \mathbb{N}$, the parameterisation. $x \in \Sigma^*$ is called an *instance* of Q and $\kappa(x)$ is the corresponding parameter. This means, a parameterised problem consists of a problem in the usual complexity-theoretic sense together with a parameterisation.

As an example, κ may be the function on strings that selects the number of different propositional atoms of a propositional formula, provided the string represents a propositional formula. Otherwise κ returns 0. Then the parameterised problem p-SAT is the problem (Q,κ), where Q is the set of satisfiable propositional formulae and $\kappa : \Sigma^* \to \mathbb{N}$.

A parameterised problem (Q,κ) is called *fixed-parameter tractable*, if there exists an algorithm deciding Q for each instance x in runtime $O(f(\kappa(x)) \times |x|^c)|$ for some function κ, and constant c.

The usefulness of an FPT algorithm depends on the value of $f(\kappa(x))$. If $f(\kappa(x))$ is small for all interesting problem instances, then the FPT algorithm is as good as a tractable algorithm. For instances where it gets large, the algorithm is most likely not better than an exponential algorithm. An example for a parameter that depends on the problem instance would be the density as defined by Bar-Yehuda et al. However, the runtime of their dynamic programming algorithm does not depend on this parameter, only the quality of their solution does. Another example is by Tsitsiklis (1992) who sketches a dynamic programming algorithm for TSPs with time-windows that is exponential in the maximal number of simultaneous time-windows. Since Tsitsiklis considers only the problem where *all* jobs must be part of a feasible tour, these algorithms are not applicable to our case of prize-collecting problems where only a subset of all jobs can be part of a tour.

In the next section we present a new algorithm that identifies an optimal solution of a given TW-TSP instance. We will then show that our algorithm is fixed-parameter tractable in a parameter which is very small in many real-world instances.

4 A fixed-parameter tractable algorithm

For a given set of jobs S, the number of different tours is very large. Provided that all jobs participate in a tour, all permutations of S are possible, which gives a number of $|S|!$ different tours. If not all jobs have to participate in a tour, then we have $n!$ different tours for each n-element subset of S, of which there are $\binom{|S|}{n}$ many. So the total number of possible tours is

$$\sum_{i=1}^{|S|} \binom{|S|}{i} \times i! = \sum_{i=1}^{|S|} \prod_{j=i}^{|S|} j,$$

which lies between $|S|!$ and $(|S|+1)!$. The number of TW-tours is much smaller, since the time-windows assigned to each job make some permutations impossible. The number of TW-tours depends on the time-windows and the distances of a given instance and cannot be specified in general.

Using a dynamic programming algorithm, it is possible to systematically extend partial tours and to get the optimal tour. Following an idea by Bar-Yehuda et al (2005), one could set up a table, where the columns are indexed by nodes and the rows are indexed by the number of already visited nodes. Row one is initialised with the release times of all nodes. Now one proceeds, starting at row 2 and fills in the earliest possible arrival time in each row i (iterating from 2 to $|S|$) for each node v. This can be easily computed by iterating over all nodes u in level $i-1$ adding the duration time of u and the travel time from u to v. Now one has to minimise over all nodes we can come from and take the maximum of the release time of v and the earliest arrival time. If it is not possible to arrive at a job in time, no matter from which other job we come from, then this job can be deleted at level i. The highest level we reach is equal to the longest tour that can be obtained respecting all time windows.

The problem with this simple algorithm is that we do not record the partial tour that leads us to each level and hence it is not possible to guarantee that every job is visited only once in our tour. Therefore, the result of this algorithm is not necessarily a proper TW-tour. A simple example that demonstrates this are two close-by jobs that have very long overlapping time-windows. The basic algorithm would alternate between these two jobs during the whole period of overlapping time-windows, leading to a very long but useless tour. One could of course delete double nodes, however, one would loose quality. For this reason, the algorithm can only give an approximation [Bar-Yehuda *et al.*, 2005].

In the following we add some modifications to the basic dynamic programming algorithm that allow us to record which jobs are not allowed to be visited in the next step since they have been visited already. We will not keep track of all jobs of partial paths, but only of jobs that have been visited already and that can be visited again without violating any temporal constraints. These jobs form a list of *prohibited jobs*.

Definition 1 (prohibited jobs) *Given a tour* $\mathcal{T} = \{(v_i,t_i)\}_{i=0}^{k}$. *The* prohibited jobs of $\mathcal{T}$, *denoted as* $P(\mathcal{T})$, *is the set of all jobs of* $\mathcal{T}$ *that satisfy the following property:*

$$P(\mathcal{T}) = \{v_i \in S \mid 1 \le i \le k, \; t_k + h(v_k) + \ell(v_k,v_i) + h(v_i) \le d(v_i)\}.$$

It is clear that there are possibly many different tours of length k that lead to a job v and that these tours have different arrival times and different prohibited jobs. In order to keep track of these different possibilities, we introduce *tour constraints* for a job v at a certain level k.

Definition 2 (tour constraints) *Given a job* v *at a level* k. *The* tour constraints C_v^k *of a job* v *at level* k *consist of the following set:*

$$C_v^k = \{(t_1,P_1),(t_2,P_2),\ldots\},$$

where t_i *is the arrival time of the* i*-th tour* $\mathcal{T}_i$ *of length* k *to job* v *and* P_i *the set of prohibited jobs of* $\mathcal{T}_i$.

Under certain conditions it is possible to remove tuples from the tour constraints. This can be done if a tour is not necessary because another tour with similar conditions has an earlier arrival time.

Lemma 1 *Given two tours* $\mathcal{T} = \{(v_i,t_i)\}_{i=0}^{k}$ *and* $\mathcal{T}' = \{(v'_i,t'_i)\}_{i=0}^{k}$ *with the same last job* $v_k = v'_k$ *and prohibited jobs* P *and* P', *respectively. If* $P \subseteq P'$ *and* $t_k \le t'_k$, *then* $\mathcal{T}'$ *cannot lead to a better overall tour than* $\mathcal{T}$.

Proof. Let $\mathcal{D}$ be the set of jobs that are part of $\mathcal{T} \cup \mathcal{T}'$ but not part of P'. No job of $\mathcal{D}$ can be used in an extension of $\mathcal{T}'$, since their deadlines cannot be reached from t'_k anymore. This is a consequence of the definition of prohibited jobs. Therefore, the jobs that can still be used to extend $\mathcal{T}'$ are a subset of the jobs that can still be used to extend $\mathcal{T}$. Since $t_k \le t'_k$, extending $\mathcal{T}'$ cannot lead to a better tour than extending $\mathcal{T}$. ■

Since we are only interested in finding the optimal TW-tour, we can delete any tour which cannot lead to a better tour than another one we have already. This leads to the following corollary:

Corollary 1 *Let* Π *be a TW-TSP instance. Given a job* v *of* Π *at a level* k *and its tour constraints* C_v^k. *If* $(t,P),(t',P') \in C_v^k$, $P \subseteq P'$ *and* $t \le t'$, *then* (t',P') *can be removed from* C_v^k *without losing the optimal solution of* Π.

Algorithm: FPT-TW-TSP(Π)
Input: A TW-TSP instance Π consisting of a set S of n jobs with time-windows $I_v = [r(v), d(v)]$, processing times $h(v)$, unit prizes $p(v) = 1$ and travel times $\ell(u,v)$ for all $u, v \in S$; v_0 is the origin.
Output: The maximal prize that can be collected from Π.

1. done = false; prize = 1;
2. *allocate* $C_v^1 = \{(r(v), \{v\})\}$ *for all* $v \in S$;
3. *for* $k = 1$ to n *do* % (different levels)
4. *if* done==true *then break*;
5. *allocate* C_v^{k+1} *for all* $v \in S$;
6. *for* $i = 1$ to n *do* % (jobs at current level)
7. done = true;
8. *for all* $(t, P) \in C_{v_i}^k$ *do* % (tour constraints)
9. *for* $j = 1$ to n *do* % (jobs at next level)
10. *if* $j \in P$ *then continue;*
11. $t_j = t + h(v_i) + \ell(v_i, v_j) + h(v_j)$;
12. $newP = \{v_j\}$;
13. *if* $t_j > d(v_j)$ *then continue*
14. *for all* $p \in P$ *do* % (prohibited jobs)
15. *if* $t_j + \ell(v_j, p) + h(p) < d(p)$
16. *then* $newP = newP \cup \{p\}$;
17. done=false; prize=k+1;
18. newt = $max(t + h(v_i) + \ell(v_i, v_j), r(v_j))$;
19. *if there is a* $(t', P') \in C_{v_j}^{k+1}$
20. with $newP \subseteq P'$ and $newt \leq t'$
21. *then remove* (t', P') from $C_{v_j}^{k+1}$;
22. *add* $(newt, newP)$ to $C_{v_j}^{k+1}$;
23. *return* prize;

Figure 1: A fixed-parameter tractable algorithm for TW-TSP

We can now modify the basic dynamic programming algorithm in a way that for every job at every level we keep track of the tour constraints for this job. In the basic algorithm we extend the tours from level k to level $k+1$ by going from each job at level k to every other job that can be performed next. In the modified version, we have to do this for every tuple in the tour constraints, since they can all lead to the optimal tour. When doing so, we have to make sure that we do not extend a tour by any of its prohibited jobs. The tour constraints at level k can be propagated to level $k+1$ by updating the jobs that remain prohibited at the next level, i.e., keeping those jobs that can still be reached and removing those that cannot be reached anymore. The modified algorithm is given in Figure 1. Note that the algorithm does not assume a given start time, and therefore the time we arrive at the first job is always the release time of the first job. This can be easily changed by modifying line 2 of the algorithm.

By adding information about the previous job to the tour constraints, it is easily possible to reconstruct the optimal TW-tour and the start time and duration of the optimal TW-tour. We leave this as an exercise to the reader and will now focus on analysing the worst-case complexity of our algorithm. This will demonstrate that the algorithm is indeed a fixed-parameter tractable algorithm.

The complexity of our algorithm depends on the maximum number or tour constraints we can have for a job and also on the maximum number of prohibited jobs we can have in a single tour constraint. In order to quantify them, we define some parameters of specific TW-TSP instances.

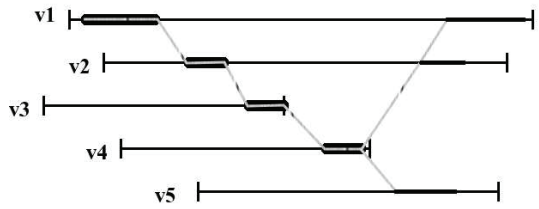

Figure 2: The TW-tour $\mathcal{T} = \{(v_1,t_1),(v_2,t_2),(v_3,t_3),(v_4,t_4\}$ has an in-between sequence length of $sl(v_4) = 2$, since it can revisit the two jobs v_1 and v_2. These two jobs are in the prohibited jobs list $P(\mathcal{T})$ and, therefore, the only feasible extension of $\mathcal{T}$ is job v_5.

Definition 3 (revisitable job, in-between job) *Given a TW-TSP instance* Π. *A* revisitable job *u of* Π *is a job for which there exists a job v in* Π *such that it is possible to visit u before v and then visit u again. Formally, this means that the following conditions are satisfied for* $t_v = \max(r(u) + h(u) + \ell(u,v), r(v)) + h(v)$*:*

1. $t_v \leq d(v)$,
2. $t_v + \ell(v,u) + h(u) \leq d(u)$.

A job v that can be used in this way is called an in-between job *of u. We write* $u \nearrow v$ *to specify that u is a revisitable job with v as its in-between job.*

A revisitable job u must have a time-window which is at least as long as the minimum duration $2 \times h(u) + \ell(u,v) + \ell(v,u) + h(v)$ for all in-between jobs v of u. A job v can be in-between job of multiple other jobs. It is possible that $u \nearrow v$ and $v \nearrow u$ both hold.

Definition 4 (in-between number) *Given a TW-TSP instance* Π. *The* in-between number $ib(v)$ of a job v *of* Π *is equivalent to the number of revisitable jobs u in* Π *that can use v as an in-between job. The* in-between number $IB(\Pi)$ of a TW-TSP instance Π *is the maximal in-between number of any job of* Π*, i.e.,* $IB(\Pi) = \max_{v \in \Pi}(ib(v))$.

The in-between number of a job is independent of how the different revisitable jobs relate to each other and doesn't specify whether multiple jobs can be revisited. For the analysis of our algorithm, it is important whether it is possible that a job v can be an in-between job of multiple revisitable jobs u and w where u is an in-between job of w. This is specified in the following definition (see Figure 2).

Definition 5 (in-between sequence length) *Given a TW-TSP instance* Π. *The jobs* $u_1, \ldots, u_k, v$ *of* Π *form an* in-between sequence *of length k, if there is a TW-tour using the nodes* $\{u_1, \ldots, u_k, v\}$ *that can be extended by any of the k jobs* $u_1, \ldots, u_k$. *The* in-between sequence length $sl(v)$ *of a job v in* Π *is the maximal length of all in-between sequences with v as the last job. The* in-between sequence length $SL(\Pi)$ *of a TW-TSP instance* Π *is the maximal in-between sequence length of all of its jobs.*

Note that the conditions for k jobs $u_1, \ldots, u_k, v$ to form an in-between sequence of length k are stronger than just assuming

$u_i \nearrow u_j$ and $u_i \nearrow v$ for all $i < j$. The above defined concepts are independent from the density parameter used by Bar-Yehuda et al. (2005). They are also independent from the maximal number of simultaneous time windows, a parameter used by Tsitsiklis (1992). Both parameters could be large while at the same time parameters based on our concepts are small. We can now prove some important properties of the algorithm.

Lemma 2 *Given a TW-TSP instance* Π*. Let* $P(v)$ *be the maximal number of prohibited jobs of a TW-tour* $\mathcal{T}$ *of* Π *that ends in a job* v*. Then* $P(v) \leq sl(v)$*.*

Proof. A prohibited job of a TW-tour $\mathcal{T}$ is defined as a job that is part of $\mathcal{T}$ and which can be performed again after visiting v. If $P(v)$ were larger than $sl(v)$, then the jobs of $P(v)$ form an in-between sequence of v which is longer than $sl(v)$. This contradicts the definition of $sl(v)$ as being the longest such tour. ∎

Lemma 3 *Given a TW-TSP instance* Π*. Let* $maxC_v^k$ *be the maximal number of tour constraints of a job* v *of* Π *at level* k*. Then*

$$maxC_v^k \leq \binom{ib(v)}{sl(v)}.$$

Proof. From lemma 1 it follows that every tour constraint $(t,P) \in C_v^k$ must have a different P. Therefore, $maxC_v^k$ is restricted by the number of different sets of prohibited jobs P. From Lemma 2 it follows that the length of P is at most $sl(v)$. The number of jobs that can occur in any P is limited to those jobs u for which v can be an in-between job of u. This number is restricted by $ib(v)$, and the number of different sets of prohibited jobs is consequently restricted by the number of possibilities of choosing $sl(v)$ jobs of $ib(v)$ available ones. ∎

We call $maxC(\Pi) = \max_{v,k} maxC_v^k$ the maximal number of tour constraints of any job $v \in \Pi$ at any level k. We can now estimate the worst case complexity of our algorithm.

Lemma 4 *The worst-case complexity of FPT-TW-TSP(*Π*) is*

$$O(SL(\Pi) \cdot maxC(\Pi)^2 \cdot n^3).$$

Proof. The algorithm has three nested loops from 1 to n (lines 3,6,9). The loop in line 14 is over the number of prohibited jobs, which is equal to $sl(v)$ for each job v (see Lemma 2), and smaller than $SL(\Pi)$. There is one loop (line 8) over the number of tour constraints. By Lemma 3, this is restricted to $\binom{ib(v)}{sl(v)}$ for each job v and is at most $maxC(\Pi)$. The test in line 19 is in the worst-case also applied to the maximal number of tour constraints. Therefore, we need $maxC(\Pi)$ again as a factor. ∎

This leads us to the main result of our paper.

Theorem 1 *FPT-TW-TSP(*Π*) is a fixed-parameter tractable algorithm that finds the optimal solution of a TW-TSP instance* Π*.*

Proof. $sl(v)$ and $ib(v)$ for any node $v \in \Pi$ and therefore $maxC(\Pi)$, $IB(\Pi)$ and $SL(\Pi)$ are parameters of Π that are independent of the size n of Π. $SL(\Pi)$ is smaller than $IB(\Pi)$ and $maxC(\Pi)$ is smaller than $\binom{IB(\Pi}{\lfloor IB(\Pi)/2 \rfloor}$, which is smaller than $(2e)^{IB(\Pi)/2}$ (e is Euler's number). So if we set our parameter κ to IB, then the worst-case complexity of our algorithm is $O(\kappa(\Pi) \cdot (2e)^{\kappa(\Pi)} \cdot n^3)$, which is of the form $O(f(\kappa(\Pi)) \cdot n^c)$, and therefore our algorithm is fixed-parameter tractable.

Assume that the TW-tour found by our algorithm is worse than the optimal TW-tour $\mathcal{T} = \{(v_i,t_i)\}_{i=0}^{k}$, where all times t_i are the earliest possible times of starting the job. Then there must be a layer k' where our algorithm arrives at $v_{k'}$ at time $t_{k'}$ but does not extend the TW-tour $\mathcal{T}' = \{(v_i,t_i)\}_{i=0}^{k'}$ to $v_{k'+1}$ on layer $k'+1$. According to the algorithm, this can happen for the following reasons:

1. $v_{k'+1}$ is a prohibited job (line 10),
2. $v_{k'+1}$ cannot be reached in time from $v_{k'}$ (line 13),
3. there is another tour arriving at $v_{k'+1}$ whose set of prohibited jobs is a subset of that of $\mathcal{T}'$ and whose arrival time is earlier or equal (line 21).

The first and second possibilities are a contradiction to $\mathcal{T}$ being a TW-tour. If the third possibility is responsible, then by Corollary 1 there must be a TW-tour going through $v_{k'+1}$ that is identified by our algorithm and that is at least as good as $\mathcal{T}$. This contradicts the assumption that $\mathcal{T}$ is better than the TW-tour identified by our algorithm. ∎

5 Applicability of the algorithm

In the previous section we have presented a fixed-parameter tractable algorithm that computes the optimal solution of a TW-TSP instance. In this section we discuss the usefulness of our algorithm for real-world problems. The main motivation of our work is the management of spatially distributed jobs, i.e., jobs where the travel time plays a considerable role. This can be jobs that are distributed over different parts of a large campus, a city or in different cities, but typically not jobs in the same office building. Another assumption we make is that jobs have a non-negligible processing time. Appointments for calendar management are typically at least 10 minutes to half an hour and not only one minute. We can also assume that time-windows are usually not huge, but typically thirty minutes to maybe a few hours, and often proportional to the processing times of jobs. Under these assumptions we can now estimate the size of the parameters of realistic TW-TSP instances. We give some examples in the following table.

process time	travel time	minimum length of time-window for sl=1	sl=2	sl=3	sl=5	sl=10
10	10	50	70	90	130	230
5	10	35	50	65	95	170
10	5	40	55	70	100	175
5	20	55	80	105	155	280
20	5	70	95	120	170	295

If we have an in-between sequence length of, say, $sl = 5$ then the time windows of the 5 participating jobs must have at least the value for $sl = 5$ for one job, the value for $sl = 4$ for the second job, the value for $sl = 3$ for the third job and so on. Therefore the time-windows of all participating jobs must be quite long. This demonstrates that for many real-world applications that fit our intended scenario, sl is usually around 2 or 3, often only 1 and very rarely larger.

For the motivating example in the Introduction it is usually the case – looking into newspapers or real-estate websites – that the visiting time for apartments is a fixed 15-30 minutes interval if a real-estate agency is involved, and a similar interval by appointment if it is a private inspection. When we assume that it takes at least 5 minutes to inspect an apartment and at least 5 minutes to travel between apartments, one would expect an *sl*-value of 1 on average. If the travel time is smaller, *sl* could become 2.

In some rare cases, such as new developments with many available apartments or private inspections where the owner has less time constraints, inspection time-windows can be longer. In these cases, a set of prohibited jobs can become larger, but it will always include some of the rare cases if larger than 2. Therefore, the actual number of different sets of prohibited jobs will not become much larger.

The worst case complexity of our algorithm is $O(SL(\Pi) \cdot maxC(\Pi)^2 \cdot n^3)$. This is smaller than $O(IB(\Pi)^{2 \cdot SL(\Pi)} \cdot n^3)$. So if $SL(\Pi)$ is small and since $IB(\Pi)$ is much smaller than n in real-world settings, we effectively get an efficient $O(n^3)$ algorithm for identifying the optimal solution to many real-world TW-TSP instances. This is the same complexity as the approximation algorithm given by Bar-Yehuda et al [2005] but is guaranteed to find the optimal solution.

6 Extensions and future work

So far, we have only considered unit prizes. However, one can extend our framework. One possible application of having non-unit prizes is to be able to specify preferences. In order to say that some jobs are preferred over other jobs, we can assign a higher prize, say 2, to the more preferred jobs. However, in order to deal with non-unit prizes, the algorithm has to be modified substantially.

One way to deal with non-unit prizes is to extend the tour constraints by the prize achieved so far, i.e., a tour constraint is now a triple (t_i, P_i, pr_i). The removal rule spelled out in Corollary 1 must also be refined. A constraint (t_i, P_i, pr_i) can only be removed if there is another constraint (t_j, P_j, pr_j) if $t_j \leq t_i, P_j \subseteq P_i$ and $pr_i \leq pr_j$. Let now $Pr(\Pi) = \sum_{v \in S} p(s)$. This means that in the worst case we may have as many tour constraints for one set of prohibted jobs P as there are different prize values possible, i.e., $Pr(\Pi)$ many. Obviously, this gives us a pseudo-polynomial FPT-algorithm. For the example above, where we only have jobs with prizes 1 and 2, this seems very reasonable.

If the values of prizes are relatively high, this may result in too many tour constraints. In order to deal with this problem, one could employ a *fully polynomial time approximation scheme* (FPTAS) in order to get an approximative answer similar to what one does in order to arrive at a FPTAS for the *knapsack* problem (see e.g. [Papadimitriou, 1994, p. 306].

There are different possible extensions to our work that can make the problems we solve more realistic or might help to solve them more efficiently. One extension is to introduce "soft jobs" and "hard jobs", similar to soft and hard constraints, where hard jobs are jobs that must be processed in any feasible tour. This is another way of enforcing preferences. It would also be interesting to consider time-windows that consist of different parts, for example, between 10am and 11am or between 2pm and 4pm. However, if we allow to express constraints between time windows using Interval Algebra relations [Allen, 1983], then algorithms of the type presented in this paper might not be applicable anymore, as we have no information about the duration of time windows.

7 Conclusion

We have presented a dynamic-programming algorithm to solve the prize-collecting travelling salesman problem with time-windows exactly. This algorithm is fixed-parameter tractable in the maximal number of jobs that can be revisited from some other job if we consider only unit prizes, and also in the maximal attainable sum of prizes if we consider non-unit prizes. As we have argued, both parameters often lead to reasonably low constant factors for the calendar management applications we consider. Therefore, the algorithm can be used in practice to find optimal solutions efficiently.

References

[Allen, 1983] J. F. Allen. Maintaining knowledge about temporal intervals. *Comm. ACM*, 26(11):832–843, 1983.

[Balas, 1989] E. Balas. The prize collecting traveling salesman problem. *Networks*, 19:621–636, 1989.

[Bar-Yehuda *et al.*, 2005] R. Bar-Yehuda, G. Even, and S. Shahar. On approximating a geometric prize-collecting traveling salesman problem with time windows. *Journal of Algorithms*, 55:76–92, 2005.

[Danzing and Ramster, 1959] G. Danzing and R. Ramster. The truck dispatching problem. *Management Science*, 80–91, 1959.

[Flum and Grohe, 2006] J. Flum and M. Grohe. *Parameterized complexity theory.* Springer, 2006.

[Jennings and Jackson, 1995] N. R. Jennings and A. J. Jackson. Agent based meeting scheduling: A design and implementation. *IEE Electronics Letters*, 31(5), 1995.

[Modi *et al.*, 2004] J. Modi, M. Veloso, S. F. Smith, and J. Oh. CMRADAR: A personal assistant agent for calendar management. In *Proc. AOIS-2004*, 2004.

[Niedermeier, 2006] R. Niedermeier. *Invitation to Fixed-Parameter Algorithms*. Oxford University Press, 2006.

[Papadimitriou, 1994] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, Reading, MA, 1994.

[Sen and Durfee, 1994] S. Sen and E. H. Durfee. On the design of an adative meeting scheduler. In *Proc. 10th IEEE Conference on AI for Applications* , 40–46, 1994.

[Shi *et al.*, 2008] X. Shi, L. Wang, Y. Zhou, and Y. Liang. An ant colony optimization method for prize-collecting traveling salesman problem with time windows. In *Proc. ICNC'08*, 480–484, 2008.

[Tsitsiklis, 1992] J. N. Tsitsiklis. Special cases of traveling salesman and repairman problems with time windows. *Networks*, 22:263–282, 1992.

[Zhang and Tang, 2007] Y. Zhang and L. Tang. Solving prize-collecting traveling salesman problem with time windows by chaotic neural network. In *Proc. ISNN'07*, 63–71, 2007.