FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Implementing Variations of the Traveling Salesperson Problem in a Declarative Dynamic Programming Environment

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Marius Liviu Moldovan
Matrikelnummer 0725855

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Mitwirkung: Projektass.(FWF) Dipl.-Ing. Michael Abseher

Wien, 04.03.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Implementing Variations of the Traveling Salesperson Problem in a Declarative Dynamic Programming Environment

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computational Intelligence

by

Marius Liviu Moldovan
Registration Number 0725855

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Univ.Prof. Dipl.-Ing. Dr.techn. Stefan Woltran
Assistance: Projektass.(FWF) Dipl.-Ing. Michael Abseher

Vienna, 04.03.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Marius Liviu Moldovan
Spengergasse 27/1611, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Acknowledgements

Hereby, I would first like to show my gratitude to my advisors, Prof. Stefan Woltran and Michael Abseher for their close coordination and the extensive implication in this work. They have always had valuable input for me and took time for me when I needed. I also thank Bernhard Bliem and Günther Charwat, for providing help with D-FLAT and with the generation of the instances, respectively.

Further, I sincerely thank my parents and grandparents for the precious advice they have been giving me, and for their support in my decisions, whatever they have been. Without their care and the foundations they laid in my education I could not have achieved this accomplishment.

Also, I am thankful to girlfriend, Alexandra, who has been there for me in both cheerful and stressful times. Last but not least, I would like to thank my friends and colleagues Bianca, Florin, Martin, Michael and Rareş, who have been good companions throughout my studenthood and have helped me to unlock this achievement.

Marius

Abstract

The *D-FLAT* System is a free framework that combines the advantages of *dynamic programming* with those of *answer set programming* (ASP). Hereby, the input instance is first decomposed into a so-called tree decomposition, then ASP is used to declaratively specify the materialization of the tables for the dynamic programming, which is done along the tree decomposition in a bottom-up manner. D-FLAT proved to perform well for simple graph problems when applied on instances with small treewidth. A more complex and prominent combinatorial problem is the *traveling salesperson problem* (TSP), which is an NP-hard optimization problem. Solving the TSP on large instances still represents a big challenge. There exist both exact algorithms and heuristics which deal with the TSP, but to the best of our knowledge neither of the two methods is at the same time practically efficient and allows for rapid prototyping.

In this master's thesis we use the D-FLAT System to propose a versatile solution for the TSP, which is implemented in a declarative, flexible environment and offers a competitive alternative to monolithic ASP implementations on instances with small treewidth. We present the dynamic programming concept we developed for the TSP and introduce the implementations of two variations of the TSP for D-FLAT. To prove the aforementioned claim, we conduct a series of experiments on generated graphs, as well as on real world instances based on the Viennese public transportation system. The significance of our work derives not only from the results of these experiments but also from the fact that the concept we proposed can be adapted for other NP-hard combinatorial optimization problems that are related to the TSP.

Kurzfassung

Das *D-FLAT* System ist ein Framework das die Vorteile von *dynamischer Programmierung* (DP) und *Answer Set Programming* (ASP) vereint. Hierbei wird die Eingabeinstanz zuerst in einen Baum zerlegt und anschließend wird ASP für das Befüllen der DP-Tabellen, basierend auf der deklarativen Spezifikation, verwendet. Dies wird entlang der Baumzerlegung von unten nach oben durchgeführt. Für einfache Graphenprobleme hat sich D-FLAT als durchaus effizient erwiesen, solange die Instanzen niedrige Baumweite aufweisen. Komplexere Probleme, wie zum Beispiel das NP-schwierige Optimierungsproblem des Handlungsreisenden, stellen weiterhin eine Herausforderung dar. Obwohl zahlreiche exakte sowie auch heuristische Ansätze für dieses Problem bekannt sind, gibt es bis dato noch keine Methode die sowohl durch praktische Effizienz, als auch durch eine einfache Handhabe und Adaptierbarkeit besticht.

In dieser Diplomarbeit verwenden wir D-FLAT um eine vielseitige Lösung für das Handlungsreisendenproblem vorzuschlagen, welche in einem deklarativen, flexiblen Umfeld implementiert ist und eine kompetitive Alternative zu monolythischen ASP-Implementierungen auf Instanzen mit niedriger Baumweite darstellt. Wir stellen ein Konzept nach den Prinzipien der dynamischen Programmierung für das Handlungsreisendenproblem vor und implementieren zwei konkrete Versionen dieses Problems in D-FLAT. Unsere Aussagen untermauern wir mit den Ergebnissen unserer Experimente, die wir sowohl auf generierten Graphen, als auch auf Instanzen die auf dem öffentlichen Wiener Verkehrsnetz basieren, durchgeführt haben. Neben den vielversprechenden Resultaten für Probleminstanzen mit niedriger Baumweite bekommt unsere Arbeit zusätzliche Bedeutung durch die Tatsache, dass das neu entwickelte, deklarative Konzept auch für zahlreiche weitere Probleme, die mit dem Handlungsreisendenproblem verwandt sind, angewandt werden kann.

Contents

CHAPTER 1

Introduction

The *traveling salesperson problem* (TSP) is a combinatorial optimization problem, which was tackled by many scientists throughout the past decades. According to [5] the TSP[1] was spread in the 1920's by the mathematician and economist Karl Menger as *Botenproblem* among his colleagues in Vienna, although the problem itself was probably formulated much earlier. Reappearing at Princeton in the 1930's, it was studied in the 1940's by statisticians and got popularized at the RAND Corporation, to finally gain notoriety as being a hard problem in combinatorial optimization in the 1950's. The TSP is actually an NP-hard combinatorial optimization problem [4], which however becomes tractable when applied on instances with small treewidth, the latter denoting a limited cyclicity of a graph.

The TSP finds application in several fields. The most obvious one is logistics, with implementations for vehicle routing, tourism, post delivery, etc. Another use is mapping the human genome, where the TSP provides a tool for building genetic sequences. Further, it found use in the production and scanning of circuit boards, guiding lasers for crystal art, aiming telescopes or even data clustering. In psychology it is used to understand the native problem-solving abilities of humans.

Throughout the time, several approaches have been used to solve the TSP: linear programming using the Simplex algorithm [19], the branch-and-bound method (introduced whilst searching an algorithm for the TSP [48]), the cutting-plane method [15], local search heuristics such as the Lin-Kernighan algorithm [47] or ant colony optimization [23] and various combinations thereof.

Dynamic programming (DP) is a method used in mathematics and computer science mainly for solving discrete optimization problems by first breaking down the problem into subproblems, solving the latter and storing their solutions, and then combining the latter into a complete solution. The term was first introduced in 1950 by the mathematician Richard Bellman and stood for multistage decision processes. When he was required to find a name for it, he came up with 'programming' for the notion of planning and 'dynamic' for something that was multi-stage and

[1] For the first time it was referenced under the name *Traveling Salesman Problem* in [54].

time-varying. In fact, the name had no connection to computer science. According to Bellman *„it also has a very interesting property as an adjective, and that is it's impossible to use the word, dynamic, in a pejorative sense"*. This was important also because the Secretary of Defense from that time, who was accountable also for the RAND Corporation, *„actually had a pathological fear and hatred of the word research"* or *„the term mathematical"* [24]. Later in the 1950s dynamic programming received its current meaning. Richard Bellman already proposed solving the TSP by means of dynamic programming in 1962 [7]. However, while solving the TSP for 17 cities required a highly performant computer, for 21 it was already unfeasible, due to memory issues.

1.1 Aim of the Work

Reasoning problems of high complexity represent an important challenge in the fields of Artificial Intelligence and Knowledge Representation when applied over large amounts of data. While offering high maintainability, exhaustive approaches face problems when applied over large data. The *D-FLAT* System is a free software framework for rapid development of answer-set-programming-encoded DP algorithms that are based on tree-decompositions [8]. *Answer set programming* (ASP) [13] is a declarative programming language that is suitable for writing programs that follow the *Guess/Check/Optimize* scheme. D-FLAT offers on one hand the easy problem modeling and flexibility of answer set programming and on the other hand the performance advantages that dynamic programming brings with itself. It was developed at the Database and Artificial Intelligence Group at the Vienna University of Technology as part of the project "Extending the Answer-Set Programming Paradigm to Decomposed Problem Solving".

The goal of this master's thesis is to

- offer versatile and easily maintainable implementations of two TSP variations,
- defined in a declarative programming environment,
- which are efficient and competitive against monolithic ASP encodings when applied on instances with small treewidth.

We will accomplish this goal by developing a dynamic programming concept for the TSP, and based on the latter, ASP-encodings which shall be suited for D-FLAT. We will show that, while maintaining the flexibility of ASP, our TSP implementations for D-FLAT work more efficiently than state-of-the-art ASP encodings on instances with small treewidth. To this end, we will evaluate the implementations both on generated and on real world instances, namely the tramway, and the metro and urban train systems of Vienna, that have treewidths 6 and 5, respectively.

1.2 Structure of the Master's Thesis

In Chapter 2 we will introduce the TSP and give basic insight into dynamic programming, ASP and finally D-FLAT. For the latter we will also show a simpler example of an encoding for the

shortest path problem. Next, in Chapter 3 we present several monolithic ASP encodings, for both variations of the TSP, and show on some example graphs how the TSP works hands-on. Chapter 4 contains our main contribution, namely various encodings of the TSP variations for D-FLAT with different configurations. Further, in Chapter 5 we present the experiments and their results and finally discuss the latter. To draw the line, Chapter 6 presents the main outcomes of this master's thesis and suggests possible directions for future work.

CHAPTER 2

Background

2.1 The Traveling Salesperson Problem

The traveling salesperson problem (TSP) can be defined as follows: Given the distances between n cities, return the cheapest tour that visits each of them once, or: Given a complete graph, find a Hamiltonian cycle with minimum cost. We will also refer to it as the *traveling salesperson problem without repetitions* (TSP-NR) as each city must be visited exactly once. Yet will we work on graphs that are not complete. Next, we will introduce a formal definition for the TSP [38,41]:

Definition 2.1.1:
Given a simple undirected graph $G = (V, E)$, with V = {1,2,...,n}, and a weight function w: $E \to \mathbb{R}^+$, we seek a cyclic permutation $\sigma = (1, \sigma(1), \sigma^2(1), ..., \sigma^{n-1}(1))$ of V, $\sigma^i(1)$ denoting the ith successor of vertex 1 (with $\sigma^0(1) = \sigma^n(1) = 1$), such that

$$w(\sigma) = \sum_{i=0}^{n-1} w(\{\sigma^i(1), \sigma^{i+1}(1)\})$$

is minimal.

We call any cyclic permutation σ of {1,2,...,n} as well as the corresponding Hamiltonian cycle 1 - $\sigma(1)$ - ... - $\sigma^{n-1}(1)$ - 1 in V a *tour* and $w(\sigma)$ its *cost*. If $w(\sigma)$ is minimal among all tours, σ is called an *optimal tour*. $\Diamond$

Further, when we will speak about a TSP variation in which some cities can be omitted or visited several times without using any edge more than once, according to an additional specification for each of these cities on a minimum and/or maximum number of visits, we will refer to the *traveling salesperson problem with repetitions* (TSP-R), a generalization of the TSP-NR. Formally it can be defined in the following way:

Definition 2.1.2:
Given a simple undirected graph $G = (V, E)$, with V = {1,2,...,n}, an optional minimum amount of visits $min(i)$ and maximum amount $max(i)$ for each node $i \in V$, both with default value 1, and a weight function w: $E \rightarrow \mathbb{R}^+$, we seek a cyclic permutation with repetition $\sigma = (1, \sigma(1), \sigma^2(1), ..., \sigma^{m-1}(1))$ of V, $\sigma^i(1)$ denoting the i$^{\text{th}}$ successor of vertex 1, such that

$$w(\sigma) = \sum_{i=1}^{m} w(\{\sigma^i(1), \sigma^{i+1}(1)\}) \text{ is minimal and}$$

$$\forall j \in V \; min(j) \leq \mid visits(\sigma, j) \mid \leq max(j), \text{ where}$$

$$\sum_{i=1}^{n} min(i) \leq m \leq \sum_{i=1}^{n} max(i) \text{ and}$$

$$visits(\sigma, j) = \{i \mid \sigma^i(1) = j, 0 \leq i \leq m-1\}. \qquad \Diamond$$

Next, we will present alternative yet equivalent definitions for the TSP-R and the TSP-NR, respectively, which further stand as starting points for our work:

Definition 2.1.3:
Given a simple undirected graph $G = (V, E)$, with V = {1,2,...,n}, and a weight function w: $E \rightarrow \mathbb{R}^+$, we seek a subset E' of E, such that there exists a path that consists only of edges in E' between any two vertices in V, further

$$\forall j \in V \mid adj(j) \mid = 2, \text{ where}$$

$$adj(j) = \{i \in V \mid (j, i) \in E'\}, \text{ and}$$

$$\sum_{e \in E'} w(e) \text{ is minimal.} \qquad \Diamond$$

Definition 2.1.4:
Given a simple undirected graph $G = (V, E)$, with V = {1,2,...,n}, an optional minimum amount of visits $min(i)$ and maximum amount $max(i)$ for each node $i \in V$, both with default value 1, and a weight function w: $E \rightarrow \mathbb{R}^+$, we seek a subset E' of E, such that there exists a path that consists only of edges in E' between any two vertices j_1 and j_2 in V for which

$$adj(j_1) > 0 \text{ and } adj(j_2) > 0, \text{ where}$$

$$adj(j) = \{i \in V \mid (j, i) \in E'\}, \text{ further}$$

$$\forall j \in V 2 * min(j) \leq \mid adj(j) \mid \leq 2 * max(j),$$

$$\forall j \in V adj(j) = 0 \bmod 2 \text{ and}$$

$$\sum_{e \in E'} w(e) \text{ is minimal.} \qquad \Diamond$$

2.2 Answer Set Programming

„*Answer set programming (ASP) is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems. As an outgrowth of research on the use of nonmonotonic reasoning in knowledge representation, it is particularly useful in knowledge-intensive applications.*“ These are the words used by Vladimir Lifschitz to describe this programming paradigm. Compared to Prolog [17], a declarative logic programming language that is goal-driven, ASP is fact-driven. This means that instead of starting with a goal and using backward chaining in the search of facts and rules to sustain the given goal, ASP starts from the given facts and rules and by forward chaining it deduces all viable results.

Stable Models and Answer Sets

ASP is based on the semantics of *stable models* [46]. These were defined by Vladimir Lifschitz and Michael Gelfond [33] and reexamined by Victor V. Marek and Mirosław Truszczyński [49]. As stable models derive from research on nonmonotonic reasoning, they also imply the concept of negation as failure [16]. This means that the default negation $not\ a$ of an atom a is true under an interpretation M if a cannot be derived in the program. In [33] Gelfond and Lifschitz consider programs that are sets of rules of the form:

$$h \longleftarrow l_1, ..., l_m$$

where the atom h comprises the head and the literals $l_1, ..., l_m$, that can be atoms or default negated atoms, with $m \geq 0$, form the body of a rule. In order to obtain the stable models of a program P one must first ground it to obtain a variable-free (propositional) equivalent of P. The actual computing of the stable models happens in the second step. An interpretation M, which is a set of atoms a_i, such that a_i is $true$ under M if $a_i \in M$ and $false$ otherwise, is a classical model of the program if it is a classical model of all rules, meaning that whenever the body of a rule is true under M, also the head must be true under M. A *Gelfond-Lifschitz reduct* P^M with regard to an interpretation M is constructed as follows:

- Every rule containing a literal $not\ a$ in its body with $a \in M$ is eliminated.
- All default negated literals in the bodies of the remaining rules are eliminated.

If M is a minimal model, with regard to subset inclusion, of P^M it is also a stable model of P. In fact, the set of all stable models of P is comprised by all minimal models of P^M.

Subsequently, we will present some examples meant to make some important remarks. Example 2.2.1 will illustrate the fact that a program can have several stable models, while Example 2.2.2 will illustrate the fact that a program can have no stable model at all. Examples 2.2.3 and 2.2.4 introduce constraints, and the necessity of strong negation, respectively.

i	M_i	P^{M_i}	Model	Stable Model
1	$\{\}$	$\{a.b.\}$		
2	$\{a\}$	$\{a.\}$	✓	✓
3	$\{b\}$	$\{b.\}$	✓	✓
4	$\{a, b\}$	$\{\}$	✓	

Table 2.1: Interpretations, reducts and models for Example 2.2.1.

i	M_i	P^{M_i}	Model	Stable Model
1	$\{\}$	$\{a.\}$		
2	$\{a\}$	$\{\}$	✓	

Table 2.2: Interpretations, reducts and models for Example 2.2.2.

Example 2.2.1:
Let the program P consist of the rules:

$$a \longleftarrow \text{not} \quad b$$

$$b \longleftarrow \text{not} \quad a$$

Table 2.1 shows all possible interpretations M_i for Example 2.2.1, with $1 \leq i \leq 4$, , their corresponding Gelfond-Lifschitz reducts P^{M_i} and whether they are models or stable models of P. M_2, M_3 and M_4 are all models of P^{M_2}, P^{M_3} and P^{M_4}, respectively but M_4 is not a minimal model. Thus, the program P has two stable models: $M_2 = \{a\}$ and $M_3 = \{b\}$. $\Diamond$

Example 2.2.2:
Let the program P consist of the rule:

$$a \longleftarrow \text{not} \quad a$$

Table 2.2 shows all possible interpretations M_1 and M_2 for Example 2.2.2, their corresponding Gelfond-Lifschitz reducts P^{M_1} and P^{M_2}, respectively, and whether they are models or stable models of P. M_2 is a model of P^{M_2}, but not a minimal one. $\{\}$ would be a smaller model of P^{M_2} but $\{\}$ is actually M_1 which is not a model to its Gelfond-Lifschitz reduct. Thus, the program P has no stable models. $\Diamond$

Example 2.2.3:
By using a rule such as the following one, we can avoid certain models. The rule

$$aux \longleftarrow a, \quad \text{not} \quad b, \quad \text{not} \quad aux$$

avoids models where a holds and b does not. $\Diamond$

The auxiliary variable from Example 2.2.3 can be omitted. Such a rule is called a *constraint* and is usually written:

$$\longleftarrow l_1, ..., l_m$$

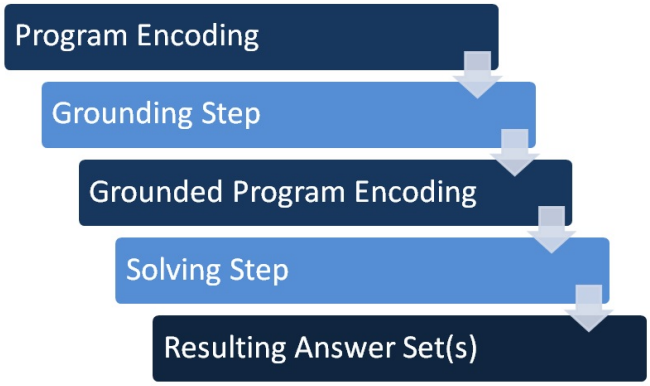


Figure 2.1: Problem solving in the paradigm of ASP.

with $m \geq 0$, and with the meaning that any answer set candidate in which the literals $l_1, ..., l_m$ hold is thrown away.

Example 2.2.4:
Sometimes default negation does not suffice. The situation modeled by the rule

$$cross \longleftarrow \text{not} \quad train$$

might be risky as it expresses that crossing is allowed if and only if there is no knowledge about a train approaching. The strong negation $\neg a$ of an atom a is true only if it can be specifically derived by a rule of the program and is more appropriate in this situation

$$cross \longleftarrow \neg train$$

as this rule expresses that crossing is allowed only when there is knowledge about the fact that there is no train approaching. $\Diamond$

Thus, ASP allows also strong negation, answer sets being stable models of programs that contain also strong negation. In ASP, the *closed-world assumption* (CWA) [46] holds, a term first introduced by Raymond Reiter [52] meaning that if an atom cannot be derived, it is allowed to derive its negation, as opposed to the *open-world assumption* (OWA) in which an atom that cannot be derived has the truth value *undefined* and which holds in the *well-founded semantics*, a three-valued version of the stable model semantics [32].

Answer Set Solving

Figure 2.1 shows the process of obtaining desired results from a given problem instance using the paradigm of ASP. First, the programmer has to define the so-called encoding, which is basically a set of rules like those defined before. Similarly to the process of finding stable models in logic, in logic programming the grounder first takes the given program and returns an equivalent variable-free program instance in which each variable is replaced by all its possible instantiations. In a second step, the solver takes the grounded program instance and returns the sought answer

```
1 0 { selected(X,Y) } 1 :- edge(X,Y,W).
2 path(X,Y) :- selected(X,Y).
3 path(X,Z) :- path(X,Y), path(Y,Z).
4 :- start(X), end(Y), not path(X,Y).
5 cost(C) :- C = #sum { W,X,Y : edge(X,Y,W), selected(X,Y) }.
6 #minimize { C : cost(C) }.
7 #show selected/2.
```

Listing 2.1: Shortest path implementation in ASP.

sets. For more details about the process we refer the reader to [42] where it is explained more thoroughly.

Programming Environment

There are several answer set solving collections. The first grounder and solver were *lparse* and *smodels* [51], respectively. The *DLV* system [44] is a deductive database system developed at the University of Calabria and the Vienna University of Technology and has a wide applicability, from hard search and optimization problems to deductive database applications. For this purpose, it offers several front ends, such as the K planning system, a front end for abductive diagnosis and Reiter's diagnosis and an SQL front end. The Potsdam Answer Set Solving Collection [28] is a set of ASP tools developed at the University of Potsdam, including tools for ASP and for constraint logic programming. In this thesis we use *clingo* [27], that uses lparse syntax and comprises the grounder *gringo* [31] and the solver *clasp* [30] from the Potsdam Answer Set Solving Collection and offers more control over the grounding and solving process than the latter two offer individually, for example by grounding and solving incrementally. The answer set solver clasp combines ASP modeling capacities with techniques from Boolean constraint solving and satisfiability checking (SAT) and relies on conflict-driven nogood learning [29]. Further, WASP is an ASP solver that uses SAT-based techniques, such as conflict-driven constraint learning, restarts and backjumping [3].

Methodology

The best practice methodology for ASP is the so-called *Guess/Check/Optimize* (GCO) scheme [44] which is an extension and refinement of the *Guess&Check* methodology, introduced in [26] and to which the *Generate/Define/Test* methodology [45] is closely related:

Guess: This part of the scheme comprises the rules and facts of the program, particularly those which build up the search space by guessing certain predicate arguments.

Check: The second, optional, part consists of the constraints that cut off the search space by eliminating answer set candidates which represent interpretations that would be inconsistent with the given problem. As shown in the example above, these constraints' heads are empty while their bodies are conjunctions of facts that would lead to invalid results.

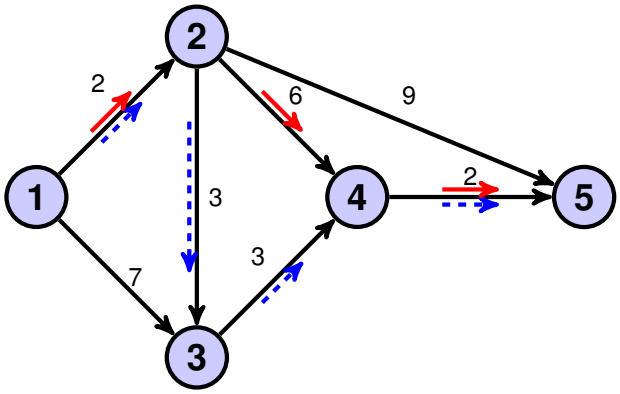


Figure 2.2: Shortest path input instance graph.

```
1 edge(1,2,2).
2 edge(1,3,7).
3 edge(2,3,3).
4 edge(2,4,6).
5 edge(2,5,9).
6 edge(3,4,3).
7 edge(4,5,2).
8 start(1).
9 end(5).
```

Listing 2.2: Shortest path input instance encoding.

Optimize: This last and optional part consists of rules that allow to grade the program's answer sets by using an objective function that implicitly maps answer sets to natural numbers, and filtering the answer sets with the minimum or maximum value, respectively.

ASP in Practice

Now we will use the *shortest path problem* on directed graphs to show an example of ASP encoding, its grounding when given a problem instance, and the resulting answer sets. The used environment will be the Potsdam Answer Set Solving Collection [28]. The goal of the shortest path problem is to find the path with minimum weight cost from a start node to an end vertex in a graph that has only positive edge weights.

Listing 2.1 shows a possible encoding for the shortest path problem on directed graphs using the paradigm of ASP. First, one can observe that according to the lparse syntax, used also by the Potsdam Answer Set Collection, :- is used instead of $\longleftarrow$. Lines 1 to 3 comprise the *Guess* section of the program as for each edge supplied with the problem instance, the solver will guess whether the edge is selected or not. Further, lines 2 and 3 build up the search space by deducing the existence of a path between two vertices if it is the case. Line 4 comprises the *Check* section as it is being checked whether there exists a path between the start and the end vertex, also supplied in the program instance, and if there is none, the answer set candidate is abandoned.

```
1 path(1,2) :- selected(1,2).
2 path(1,3) :- selected(1,3).
3 path(2,3) :- selected(2,3).
4 path(2,4) :- selected(2,4).
5 path(2,5) :- selected(2,5).
6 path(3,4) :- selected(3,4).
7 path(4,5) :- selected(4,5).
```

Listing 2.3: Excerpt from grounded shortest path encoding.

```
1 Answer: 1
2 selected(1,2) selected(2,4) selected(4,5)
3 Optimization: 10
4 Answer: 2
5 selected(1,2) selected(2,3) selected(3,4) selected(4,5)
6 Optimization: 10
```

Listing 2.4: Resulting optimal answer sets.

Finally, the *Optimize* section is comprised by lines 5 and 6. The objective function is defined in line 5 by using the predicate `cost/1`. Its argument represents the sum of the weights of all selected edges. Line 6 is a directive for grading answer sets by the arguments of their `cost/1` predicates, the lower the better. The last line is a directive for showing only the `selected/2` predicate instances of the answer set(s) as only these are relevant for the given problem.

Figure 2.2 shows a possible problem instance in which we are looking for the path with minimum cost from vertex 1 to vertex 5.

Listing 2.2 shows the encoding for the input instance shown in Figure 2.2, with vertex 1 as start node and vertex 5 as end node, an enumeration of facts which are nothing else than rules that lack a body, meaning that they must hold at any time.

In Listing 2.3 we can see an excerpt from clingo's grounded encoding for the shortest path problem encoding presented in Listing 2.1 corresponding to line 2 in the latter. We observe that the rule is instantiated with all possible values according to the given facts and to the rule in line 1, Listing 2.1, such that a `path/2` predicate will be deduced by the solver from a `selected/2` predicate with the same arguments only if the latter has been guessed positively.

Finally, Listing 2.4 shows the answer sets supplied as output by clingo. They were chosen as such, as among the three answer set candidates deduced by the solver, which are not abandoned because of a constraint, they are those with the lowest total cost, minimizing the objective function.

2.3 The D-FLAT System

The D-FLAT System, developed at the Vienna University of Technology [8], promises the flexibility offered by declarative programming and also handles large amounts of data by generating a tree from the initial problem instance and by running a declarative programming algorithm

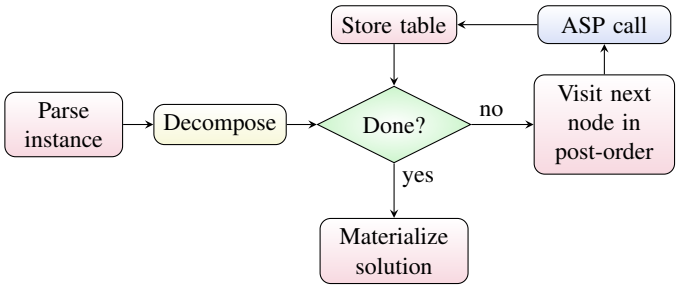


Figure 2.3: Control flow in D-FLAT, adapted from Figure 4 in [1].

on each of this tree's nodes which represent subproblems of the original instance. The system itself written in C++, lets the user specify the algorithm using ASP and delegates the burden of computation and optimization to a library for finding a good tree decomposition and to an ASP solver.

Figure 2.3 shows the basic control flow in the D-FLAT system during the execution of an algorithm. First, the given problem instance is parsed and a tree decomposition is built out of it, by decomposing the instance into several smaller subinstances. A loop follows, which handles every node of the tree decomposition in post-order. D-FLAT solves each subproblem corresponding to a node by calling the ASP encoding provided by the programmer, once for each node. The partial results are stored into D-FLAT's own data structures, either in so-called *item trees* or in *tables*. Finally, after visiting all the nodes of the tree decomposition, it combines all the partial solutions using the data stored in the provided data structures and prints all complete solutions. Alternatively, it can return an optimal solution or the result for a decision or counting problem.

According to [9], any problem that is expressible in monadic second-order logic, which includes many problems from NP and some from PSPACE, can be solved using D-FLAT in fixed-parameter time (FPT).

In this section we will further describe the D-FLAT System, offering an introduction into tree decompositions and dynamic programming, and illustrating D-FLAT's interface, by using information from the D-FLAT Progress Report [1] and by presenting a similar example to the one introduced in the previous section. However, note that we will show an implementation for the shortest path problem on directed graphs only for illustration purposes as it can be easily solved with greedy polynomial-time heuristics, such as Dijkstra's algorithm [22].

Tree Decompositions

A tree decomposition is a mapping of a graph to a tree that fulfills certain conditions and is used in general for solving NP-hard problems more easily by removing the cyclicity of the initial graph. For disambiguation, from now on we will use the term „node“ for referring to a node in the tree decomposition and „vertex“ for referring to a vertex in the initial graph. A tree decomposition and a graph's treewidth, which can be understood as its cyclicity degree, are formally defined as follows [12, 53]:

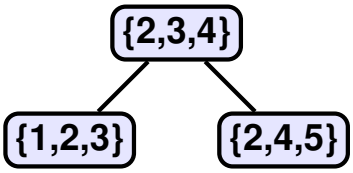


Figure 2.4: An optimal tree decomposition for the graph in Figure 2.2.

Definition 2.3.1:
Given a graph $G = (V, E)$, a *tree decomposition* of G is a pair (T, χ) where $T = (N, F)$ is a tree and $\chi : N \to 2^V$ assigns to each node a set of vertices (called the node's *bag*), such that the following conditions hold:

1. For each vertex $v \in V$, there exists a node $n \in N$ such that $v \in \chi(n)$.
2. For each edge $(v_1, v_2) \in E$, there exists an $n \in N$ with $v_1 \in \chi(n)$ and $v_2 \in \chi(n)$.
3. For each $n_i, n_j, n_k \in N$: If n_j lies on the path between n_i and n_k then $\chi(n_i) \cap \chi(n_k) \subseteq \chi(n_j)$.

The *width* of a tree decomposition is defined as $\max_{n \in N} |\chi(n)| - 1$ and the *treewidth* of a graph is the minimum width over all its tree decompositions. ♢

To sum up, each vertex and each edge must belong to a node and if a vertex belongs to two different nodes, it must belong to all nodes on the path between the two as well. Note that a graph always has a tree decomposition and that the tree decomposition of a graph is in general not unique. A tree has treewidth 1 and a cycle has treewidth 2. Figure 2.4 shows an optimal tree decomposition of the graph in Figure 2.2, which has treewidth 2. The notion of treewidth is important, as many problems that are intractable in general become tractable when the treewidth of the input instance is bounded by a constant [11, 35, 36]. Moreover, research shows that in many practical situations instances have a small treewidth [2,37,40,43,50,56]. When building a tree decomposition, the directedness of as graph is not taken into account. However, there exist so-called arboreal decompositions, which also define a directed tree-width [25], but D-FLAT does not work with these due to the lack of libraries to implement them.

Determining a graph's treewidth and constructing an optimal tree decomposition are intractable, yet fixed-parameter tractable. This means that, given a graph and a non-negative integer k, deciding whether the graph's treewidth is at most k is NP-complete [6], yet having k fixed in advance as part of the problem statement, and not the instance, one can decide in linear time whether the graph's treewidth is at most k and, if it is the case, construct an optimal tree decomposition [10]. Furthermore, there are efficient heuristics available for producing reasonably good tree decompositions, for the situation that k is not known in advance [12, 21, 34].

As mentioned above, the D-FLAT system works on tree decompositions. In order to construct them, it first creates a hypergraph representation of the input, according to the user's specification of edges and vertices in the program call command. D-FLAT constructs a hypergraph that has binary hyperedges corresponding to the graph's edges and unary hyperedges that correspond to the graph's vertices. In order to build a tree decomposition of small width, it uses

SHARP, an external C++ library, which internally makes use of a software called *htdecomp* [21], and relies on a bucket elimination algorithm [20]. One way to build a tree decomposition is to successively extract a vertex from the hypergraph, according to a predefined *elimination order*, make it simplicial (i.e. connect all its neighbors such that they form a clique) and create a new decomposition node that contains the vertex and all its neighbors.

The order in which the tree decomposition is built by the external library is given by the order in which the vertices are progressively eliminated from the hypergraph. The framework supports the following heuristics for finding such elimination orders:

Min-degree: The vertex with the least number of not selected neighbors is selected.

Min-fill: The vertex whose elimination adds the least number of edges to the hypergraph.

Maximum Cardinality Search: After selecting the first vertex randomly, it is always the vertex with the greatest number of already selected neighbors, that gets selected.

In order to easen the process of implementing in ASP, D-FLAT offers some optional *normalizations* of tree decompositions, even if it linearly increases their size:

Weak Normalization: Each node that has more than one child, called a *join node*, must have the same vertices in its bag as its children. Nodes that have only one child are called *exchange nodes*.

Semi-normalization: The condition for weakly normalized tree decompositions must hold and a join node must have exactly two children.

Normalization: The conditions for semi-normalized tree decompositions must hold and each exchange node can either consist of all but one vertex of its child's bag, in which case it is a *remove node*, or of all vertices of its child's bag plus another vertex, in which situation it is called an *introduce node*.

The root and the leaves of the tree decomposition are by default empty in D-FLAT. Additionally, the user has the possibility to impose having above each join node an extra identical node which can be beneficial for the simplicity of the encoding when using the `--default-join` option, which we will also introduce later on.

Dynamic Programming

Dynamic programming (DP) is a mathematical and computer engineering method used mainly for solving discrete optimization problems by first solving subproblems and storing their solutions, and then combining the latter into a complete solution. A property that all problems which can be efficiently solved by dynamic programming show is called *overlapping subproblems*. A problem having this property can be divided into several subproblems whose solutions, stored for example in tables using a technique called *memoization* or *result caching*, can be then reused multiple times for constructing a solution to the initial problem. This property is also the reason for the inefficiency of recursive solutions to these problems, as such a solution would

i	(2,3)	(2,4)	(3,4)	j	cost
0			✓	{(0,0)}	12
1				{(2,1)}	11
2		✓		{(2,3)}	10
3		✓	✓	{(2,3)}	13
4	✓	p	✓	{(3,0)}	10
5	✓			{(3,1)}	14
6	✓	✓		{(3,3)}	13
7	✓	✓	✓	{(3,3)}	16

i	(1,2)	(1,3)	(2,3)	cost
0		✓		7
1		✓	✓	10
2	✓			2
3	✓	p	✓	5
4	✓	✓		9
5	✓	✓	✓	12

i	(2,4)	(2,5)	(4,5)	cost
0			✓	2
1		✓		9
2		✓	✓	11
3	✓	p	✓	8
4	✓	✓		15
5	✓	✓	✓	17

Figure 2.5: Dynamic programming tables for the shortest path problem on the tree decomposition in Figure 2.4.

lead to many identical recursive calls [57]. Dynamic programming is related to the *divide-and-conquer* technique which also solves problems by combining solutions to their subproblems. Both techniques work on problems with *optimal substructure*. A problem displays this property if an optimal solution to it, holds optimal solutions to its subproblems within itself. However, opposed to dynamic programming, a divide-and-conquer algorithm does not take advantage of overlapping subproblems [18].

Figure 2.5 shows the dynamic programming tables for the shortest path problem on the tree decomposition shown in Figure 2.4. The goal is to find a path from vertex 1 to vertex 5 in the graph from Figure 2.2. Every node of the tree decomposition has a corresponding table. The tables are computed in a bottom-up manner, starting from the *leaf* nodes. The tables corresponding to the latter are generated by listing all possible partial solutions. Each row stands for a possible partial solution and each column for the identifier of the partial solution, for the edges between the node's vertices and for the cost of the partial solution, respectively. A cell is checkmarked if the corresponding edge has been selected in the specific partial solution, while p stands for the fact that there exists a directed path between the two vertices. The latter is used for example to determine whether a path exists from the start to the end vertex. For instance, the table for the left leaf node in Figure 2.5 shows all possible combinations, for selecting the edges between the so-called *current* vertices, that can possibly lead to a solution, current vertices being those that are contained in the node's bag, in this case the vertices 1, 2 and 3. Please note that the partial solution candidates in which no outgoing edges of vertex 1 were selected are left out, as they cannot lead to a solution. In non-leaf nodes, new partial solutions are

computed based on the current bag vertices and on partial solutions of their children. Their tables are computed by *extending* their children's partial solutions and posses an extra column which contains the identifiers of the latter. In our example, the *root* node is at the same time a *join* node, as it combines and extends partial solutions of its two child nodes, but also removes those combinations that would not lead to a solution. Its bag contains vertices from both child nodes, while some of the child nodes' bag elements have been eliminated, which is why vertices 1 and 5 are called *removed* vertices. At the same time, a vertex contained by the current bag but not by any of its children's bags would be called an *introduced* vertex. Looking at the join node's table we can see possible combinations of the partial solutions in the child nodes (at the same time extensions to the latter) which lead to a viable solution to the problem, i.e. which contain the necessary edges to form a path from the start vertex 1 to the end vertex 5. For each distinct row in the join node we keep and display the pair of extended rows that leads to the lowest accumulated cost. For example, row 1 could be obtained by combining child rows 0 and 0, 0 and 1, 2 and 1, 2 and 2, 4 and 0, 4 and 1, or 4 and 2, whilst selecting also the edge (3,4). However, combining rows 0 and 0 leads to a minimum cost of 12 so this is the one stored in the table. Row 1 could be obtained also by combining child rows 0 and 1, which would lead to a lower cost of 9, but it would not contain enough edges for a solution to the given problem. Finally, among the partial solutions listed in the join node, the solutions with the lowest cost are chosen as optimal solutions, corresponding to rows 2 and 4. Now, the total solutions can be computed by tracing back the partial solutions down to the leaf nodes by following the identifiers stored in the *j* column. We observe that by combining row 2 from the join node's table with row 2 from the left leaf node and row 3 from the right one we obtain the edges (1,2), (2,4), (4,5), which put in the right order, form an optimal path from vertex 1 to vertex 5. Similarly, combining rows 4, 3 and 0, respectively, we obtain the other optimal solution.

In D-FLAT, each of the tree decomposition nodes is connected either to an item tree or to a table which works in a similar way to the one explained above. In this master's thesis we will use only the table as data structure for storing partial solutions, so please refer to [1] for further information about item trees. In D-FLAT, each partial solution is stored as a so-called *item set*, consisting of *items* which are arbitrary ground ASP terms. In our example, if there is a checkmark for a certain edge, say (1,2) in a node's table, the item set to the specific partial solution would contain an item `selected(1,2)`. Furthermore, the D-FLAT system uses *extension pointers* for creating the link between the nodes' partial solutions. From each row of a non-leaf node's table there are one or more extension pointers indicating the extended rows.

D-FLAT's Interface for ASP

During a bottom-up traversal of the tree decomposition, D-FLAT invokes the ASP solver for every node. The solver is fed with the user-specified encoding of the problem, the facts describing the input instance and facts about both the current bag and items, and those of the current node's children. Table 2.3 explains the predicates we will further use in this work, for deriving these facts about the tree decomposition and its contents. The first twelve predicates are used as input for the ASP solver while the others are output predicates. Note that we make use of D-FLAT's simplified, less general interface for problems in NP, that usually suffices for problems in NP, and which also makes use of tables instead of item trees in the process of dynamic programming.

Input Predicate	Meaning
`initial`	The current tree decomposition node is a leaf.
`final`	The current tree decomposition node is the root.
`currentNode(N)`	N is the identifier of the current decomposition node.
`childNode(N)`	N is a child of the current decomposition node.
`bag(N,V)`	V is contained in the bag of the decomposition node N.
`current(V)`	V is an element of the current bag.
`introduced(V)`	V is a current vertex but was in no child node's bag.
`removed(V)`	V was in a child node's bag but is not in the current one.
`childRow(R,N)`	R is a table row belonging to decomposition node N.
`childItem(R,I)`	The item set of table row R contains I.
`childAuxItem(R,I)`	The auxiliary item set (for the default join) of table row R contains I.
`childCost(R,C)`	C is the cost value corresponding to the table row R.
Output Predicate	**Meaning**
`item(I)`	The item set of the current table row shall contain the item I.
`auxItem(I)`	The auxiliary item set (for the default join) of the current table row shall contain the item I.
`extend(R)`	The current table row shall extend the child table row R.
`cost(C)`	The current table row shall have a cost value of C.
`currentCost(C)`	The current table row shall have a current cost value of C.

Table 2.3: Predicates describing the tree decomposition and its contents as presented in [1].

In order to exemplify the way the D-FLAT framework functions and how the predicates in Table 2.3 are used, we implemented the shortest path problem on directed graphs also in D-FLAT, as shown in Listing 2.5. The encoding follows the GCO-scheme [44]. We first guess a child node's row to be extended and whether to select incoming or outgoing edges of an introduced vertex for the pool of edges that will later represent a path. Next, we deduce which vertices are connected by a path, particularly with the start and the end vertex, and also what we can deduce from items of the extended child node's row. The checking part consists of a constraint that applies only for join nodes, one that throws away all solution candidates that do not contain a path from the start to the end vertex, and two which are fired when the start or end vertex is eliminated and it is not connected by a path to any current vertex, respectively. Finally we define the cost function for nodes with one and two children, respectively.

The first line is a so-called *modeline*: if the first line of the encoding, specified in the command line with `-p`, starts with `%dflat:` (followed by a space), the rest of the line is considered to be part of the command line. In our example encoding we specify the binary hyperedge predicate `edge/2` with `-e` (we do not need a unary hyperedge predicate `vertex/1` as we work solely with edges and a path could not contain an isolated vertex anyway), the fact that we work with tables rather than item trees with the option `--tables` and the fact that our encoding is constructed for working on semi-normalized tree decompositions with `-n semi`, such that we

```
1  %dflat: -e vertex -e edge --tables -n semi

2  %Guess row to be extended.
3  1 { extend(R) : childRow(R,N) } 1 :- childNode(N).

4  %Guess whether to select introduced nodes' adjacent edges.
5  0 { item(selected(X,Y)) } 1 :- edge(X,Y), introduced(X), current(Y).
6  0 { item(selected(Y,X)) } 1 :- edge(Y,X), introduced(X), current(Y).

7  %Define the path/2 predicate.
8  item(path(X,Y)) :- item(selected(X,Y)), current(X;Y).
9  item(path(X,Z)) :- item(path(X,Y)), item(path(Y,Z)), current(X;Y;Z).

10 % Fix constant arguments if start, end or a link vertex is in the bag.
11 item(path(s,Y)) :- start(X), item(path(X,Y)), current(X;Y).
12 item(path(X,e)) :- end(Y), item(path(X,Y)), current(X;Y).
13 item(path(s,e)) :- item(path(s,X)), item(path(X,e)), current(X).

14 % Define transitivity for path/2 with constant argument.
15 item(path(s,Z)) :- item(path(s,Y)), item(path(Y,Z)), current(Y;Z).
16 item(path(X,e)) :- item(path(X,Y)), item(path(Y,e)), current(X;Y).

17 % Pass on items from child nodes.
18 item(selected(X,Y)) :- extend(R), childItem(R,selected(X,Y)), current(X;Y).
19 item(path(X,Y)) :- extend(R), childItem(R,path(X,Y)), current(X;Y).
20 item(path(s,Y)) :- extend(R), childItem(R,path(s,Y)), current(Y).
21 item(path(X,e)) :- extend(R), childItem(R,path(X,e)), current(X).
22 item(path(s,e)) :- extend(R), childItem(R,path(s,e)).

23 % Constraints for join nodes, exchange nodes and for the root node, respectively.
24 :- extend(R;S), R != S,
       childItem(R,selected(X,Y)), not childItem(S,selected(X,Y)).
25 :- removed(X), start(X), not item(path(s,_)).
26 :- removed(X), end(X), not item(path(_,e)).
27 :- final, not item(path(s,e)).

28 % Cost function for join and exchange nodes, respectively.
29 cost(CC - LC) :- 2 #count{ X : childNode(X) } 2,
       CC = #sum { CCI,R : extend(R), childCost(R,CCI) },
       LC = #sum { W,X,Y : weight(X,Y,W), item(selected(X,Y)) }.
30 cost(CC + NC) :- 1 #count { X : childNode(X) } 1,
       extend(R), childCost(R,CC), NC = #sum {
       W,X,Y : weight(X,Y,W), item(selected(X,Y)), introduced(X),
       W,X,Y : weight(X,Y,W), item(selected(X,Y)), introduced(Y) }.
```

Listing 2.5: Shortest path implementation for D-FLAT.

only need to specify the problem and the instance encodings when running D-FLAT.

Line 3 guesses a row to extend for each child of the current node and deduces a fact `extend(R)`, where `R` is the child node's extended row. The `extend/1` predicate and its argument will then be used throughout the code to refer to the items of a specific row of a child node's table. Usually a rule like this one is part of every table-mode encoding. Lines 5 and 6 guess for all introduced vertices whether the edges ingoing from or outgoing to other current vertices shall be selected. If affirmative, a fact `selected/2` is deduced. Line 8 establishes that, if an edge between two vertices is selected, then there exists also a (directed) path between them. Note that such rules are fired only if the vertices in cause are in the current node's bag.

Otherwise we would deviate from the dynamic programming paradigm. Please also note that `current(X;Y)` is the condensed notation of `current(X), current(Y)`. Line 9 represents the transitivity of the `path/2` predicate. Lines 11, 12 and 13 are meant to fix a constant argument `s` or `e` for the `path/2` items if the start or the end vertex is contained in the current bag, respectively. If the linking vertex between a path from the start vertex and to the end vertex is in the current bag, a fact `path(s,e)` is deduced. Note that, even if we actually use the start and the end vertex at other nodes than those which they are contained in, we do not deviate from the principle of dynamic programming as we do this only for a constant number of vertices. Lines 15 and 16 deal with the transitivity of the `path/2` predicate specifically when one of the arguments is not a current vertex but a constant. In lines 18 to 22 we deduce items from those of child nodes, namely if in a child node an edge was selected, the corresponding `selected/2` fact must also be deduced in the current node yet only if the two implied vertices are still current. The same holds for the `path/2` predicate.

When joining partial solutions from two children, they must match, meaning that if and only if an edge is selected in one child node it must be selected also in the other one. This is the reason for introducing the constraint in line 24, which throws away all solution candidates where an edge is selected in one child node and in the other one not. As we are working on a semi-normalized tree we do not have to mind about edges between vertices that are in one child node's bag and in the other one's not. Also, one could worry about an edge between two vertices that are not in the join node's bag, being selected somewhere in one tree branch, while in the other one not. However, this case is excluded due to the construction of the tree decomposition: if one vertex appeared in both tree branches, it would be contained by all nodes on the path inbetween, including the join node, which we had assumed otherwise. Next, we discard all answer set candidates in which there does not exist any vertex connected to the start or end vertex through a path of selected edges after the former have been removed, in lines 25 and 26, respectively. Once the start and the end vertex have been removed and they are not connected to any other vertex it is impossible to obtain a path from one to another. Finally, in line 27 we get rid of all solution candidates that do not contain a path between the start and the end node. At this point we are left only with solution candidates that contain the desired path but might also contain additional edges.

The latter are handled by the cost function, which is responsible also for choosing the optimal solution, as it will always rather take a solution candidate that contains in its edge pool only edges to form the desired path rather than one that contains also additional edges, given that the edge weights are positive. Lines 29 and 30 define the `cost/1` predicate for exchange and join nodes, respectively, and implicitly the cost function. In both rules we use the `#count`-*aggregate* for counting the current node's number of children and the `#sum`-*aggregate* for summing up the current cost. They count and sum up, respectively, the values standing before the colons, which apply to the facts listed after them. As leaf nodes are by default empty in D-FLAT, it is not necessary to deduce a cost for those, in which case D-FLAT automatically assumes the cost is 0. The rule for join nodes, in line 29, counts the number of child nodes and is applicable to those having exactly two of them. Further, it sums up all child nodes' costs. In fact, the aggregate sums up only the first value listed before the colon while the second one is only for differentiation. Without the second one, D-FLAT would add the child node's cost only once if both children had

```
1  edge(1,2).
2  edge(1,3).
3  edge(2,3).
4  edge(2,4).
5  edge(3,4).
6  edge(4,5).
7  edge(2,5).
8  weight(1,2,2).
9  weight(1,3,7).
10 weight(2,3,3).
11 weight(2,4,6).
12 weight(3,4,3).
13 weight(4,5,2).
14 weight(2,5,9).
15 start(1).
16 end(5).
```

Listing 2.6: Shortest path input instance encoding.

equal costs. The last part of the rule sums up the weights of all selected edges in the current bag. Finally, the cost of a join node is deduced as being the sum of the children's costs which we subtract the weights of the edges between current vertices from. Otherwise, the latter would be counted twice as a join node's current edges are contained by both children's bags, and their weights are included in both children's costs. The last rule, for calculating the cost for exchange nodes, applicable only for nodes with one child, adds the latter's cost to the weights of newly introduced nodes' selected edges. In more detail, we add the weights for selected edges whose start vertices are newly introduced, to the weights of the selected edges whose end vertices are newly introduced. We do not have to mind selected edges whose both adjacent vertices are newly introduced as the `#sum`-aggregate adds a value only once if the instantiated variables in front of the colons are the same, no matter their order, and we do not have an additional differentiator besides the value and the two vertices, such that these edges' weights are always counted only once.

Further, in Listing 2.6 we present the D-FLAT encoding for the input instance shown in Figure 2.2, with vertices 1 and 5 as start and end vertices, respectively, also supplied when running D-FLAT. Note that for D-FLAT we need to specify the weights separately because by supplying a ternary predicate `edge` that would include also an edge's weight, D-FLAT would construct its graph with ternary, instead of binary, hyperedges. Also note that in the input instance, the vertices should be designated by numbers rather than letters in order not to interfere with the constants `s` and `e` used in the encoding for designating the start and the end vertex. For better understanding the way in which D-FLAT works, we show an excerpt of the input the ASP solver receives at the decomposition node 6 in Listing 2.7. The first part contains the literals used for referencing to table rows 1 and 2 in the table of the decomposition node 7 (see Figure 2.6). Besides the items and the cost of the child node we can see a reference to each row of the child node's table, reference which then appears also in the references to the child items and the child cost. In this way and by using the rule in line 3 of the encoding in Listing 2.5 and the predicate

```
% Child item tree facts.
...
childRow(n7_1,7).
childItem(n7_1,path(2,4)).
childItem(n7_1,path(2,5)).
childItem(n7_1,path(2,e)).
childItem(n7_1,path(4,5)).
childItem(n7_1,path(4,e)).
childItem(n7_1,selected(2,4)).
childItem(n7_1,selected(2,5)).
childItem(n7_1,selected(4,5)).
childCost(n7_1,17).
childRow(n7_2,7).
childItem(n7_2,path(2,4)).
childItem(n7_2,path(2,5)).
childItem(n7_2,path(2,e)).
childItem(n7_2,path(4,5)).
childItem(n7_2,path(4,e)).
childItem(n7_2,selected(2,4)).
childItem(n7_2,selected(4,5)).
childCost(n7_2,8).
...
% Decomposition facts.
currentNode(6).
bag(6,2). current(2).
bag(6,3). current(3).
bag(6,4). current(4).
#const numChildNodes=1.
childNode(7).
bag(7,2). -introduced(2).
bag(7,4). -introduced(4).
bag(7,5). -introduced(5).
postJoin.
introduced(X) :- current(X), not -introduced(X).
removed(X) :- childNode(N), bag(N,X), not current(X).
```

Listing 2.7: Excerpt from the input for the solver at decomposition at node 6.

```
Answer: 1
path(1,2) path(1,3) path(2,3) path(2,4) path(2,e) path(3,4) path(3,e)
     path(4,5) path(4,e) path(s,2) path(s,3) path(s,4) path(s,e)
     selected(1,2) selected(2,3) selected(3,4) selected(4,5)
(cost: 10)
Answer: 2
path(1,2) path(2,4) path(2,5) path(2,e) path(4,5) path(4,e) path(s,2)
     path(s,4) path(s,e) selected(1,2) selected(2,4) selected(4,5)
(cost: 10)
```

Listing 2.8: D-FLAT result for the shortest path problem.

`extend/1` in the rules in which we reference to child items (e.g. line 18), we can make sure that we always extend only one table row of the child node at a time, such that each row of the current node's table relies on only one row of the child node's table, and also extend all of the child node's table rows with a single call of the ASP solver for the current node. The second part of the input the ASP solver receives at the decomposition node 6 in Listing 2.7 consists of literals related to the belongingness of vertices to a node. Finally, Listing 2.8 shows D-FLAT's result when supplied with the rules and facts in Listings 2.5 and 2.6.

Next, we will explain how D-FLAT actually obtained these results. After traversing the tree decomposition bottom-up, calling the ASP solver on the encoding and computing the dynamic programming tables at each node, the tree decomposition and its tables look as shown in Figure 2.6, which displays a subset of each node's rows for the problem defined above. The two optimal solutions and the partial solutions they extend, are marked throughout the tree decomposition. Finally, D-FLAT extends for every node, starting with the root, each of the rows recursively by following extension pointers, in absence of a cost function. When dealing with an optimization problem rather than an enumeration problem (a cost function exists), D-FLAT extends only those solutions having a minimal cost and when extending some node's partial solution that can be obtained by extending two different child rows it always picks the one with lower cost for materializing the solution. In our example, it starts at node 1 and extends the solution at node 2 only with those rows having a minimal cost, of 10. Further, the item set of the solution displayed in the second row of node 2 in Figure 2.6 is unified with the items in the second row at node 3 and those in the second row at node 4, on the left branch. On the right branch, the second row at node 2 is extended by the fifth row at node 6 which in turn is extended by the second row at node 7, finally also extended by the empty row at the leaf, thereafter unifying the items accumulated on the left branch with those from the right branch. The other optimal solution is extended, except at the leaf nodes, by the other, remaining rows whose item sets get unified separately from the first solution.

D-FLAT offers an additional option for joining several branches in a join node, which is called *default join* and can be invoked with the option `--default-join` in order to improve the performance. In this mode, D-FLAT automatically matches rows that have identical item sets, from each child node, and does not call the ASP encoding. If costs are specified, it also calculates the cost of each row by summing up all children's costs and subtracting the cost that is due to current bag elements n-1 times (where n is the number of child nodes), that must be defined by the programmer using the predicate `currentCost/1`, by the inclusion-exclusion principle. If it is desired to join the different branches based only on some common items, the programmer can specify also *auxiliary items* which work like regular items but do not need to be identical when matching different rows of child nodes, such that the latter are matched only based on the item sets of the child nodes. The *auxiliary item set* in the join node will be the union of all matched children's auxiliary item sets. The default join is implemented directly in D-FLAT using C++ and makes use of a certain order of rows in tables, similarly to the sort-merge join algorithm in relational database management systems, thus improving the performance at join nodes significantly. In case the programmer needs to do some post-processing on the resulted joined rows, which cannot be done on the join node when using the default join as the ASP encoding is not called, the option `--post-join` can be invoked and above every join node

an identical node is inserted, which can be used for this purpose and which we will call *post-processing node*.

Figure 2.6: Screenshot excerpt from D-FLAT debugger on the shortest path problem.

CHAPTER 3

Monolithic ASP Implementations for the Traveling Salesperson Problem

In this chapter we will present different monolithic ASP encodings for the TSP and prove ASP's high level of maintainability and flexibility. However, the price for these is a high running-time when applied on large problem instances. First, we will present an ASP encoding for the TSP-NR that relies on an implementation from [39], initially implemented for working only on directed graphs, and then, some ASP implementations for the TSP-NR and the TSP-R that we propose.

3.1 Monolithic ASP Encodings for the TSP-NR

The encodings presented in this section are all designed for input instances specifying an undirected graph. They expect the instances to be encoded by using the `vertex/1`, the `edge/2` and the `weight/3` predicates to specify their vertices, edges and the weights of the latter, respectively. The `edge/2` predicate is expected to be used twice for each edge, by specifying two facts that are symmetric with regard to their argument vertices. The same holds for the `weight/3` predicate, without loss of generality. Here we always consider the lower weight among the two specified for each edge. The encoding presented in Listing 3.2 expects also a start vertex 0 to be among the specified vertices. Without loss of generality, we will use instances in which the vertices are denoted by numbers between 0 and the input graph's number of vertices. Figure 3.1 shows a possible input instance in graph representation, while Listing 3.1 shows an ASP encoding extract of such an input instance. As we always consider the lower weight of an edge, the optimal tour (0-1-2-3-5-6-4-0) in the presented example will have a cost of 18.

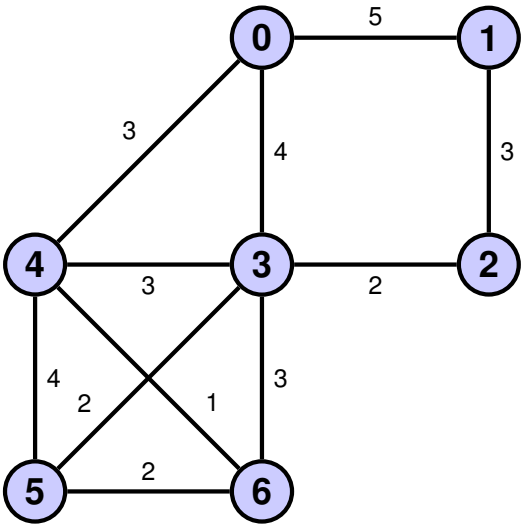


Figure 3.1: TSP input instance graph.

```
1  vertex(0).
2  vertex(1).
3  vertex(2).
4  ...
5  edge(0,1).
6  edge(1,0).
7  edge(1,2).
8  edge(2,1).
9  ...
10 weight(0,1,5).
11 weight(1,0,7).
12 weight(1,2,3).
13 weight(2,1,3).
14 ...
```

Listing 3.1: Excerpt from sample input instance for the TSP-NR.

ASP Encoding for the TSP-NR Relying on Reachability

The encoding presented in Listing 3.2 first finds Hamiltonian cycles in a graph in order to then find the one with the lowest cost. The basic idea for finding Hamiltonian cycles is, following the GCO-scheme [44], to guess for each vertex an incoming and an outgoing edge and then remove all solution candidates in which not all vertices are reachable by selected edges from a start vertex. Finally we keep only the solutions with the minimum cost.

First, please note that the identifiers `s` and `r` stand for *selected* and *reachable*, respectively. Lines 1 and 2 specify that a vertex should have exactly one outgoing and one incoming edge in the tour, using the cardinality restriction. These lines represent the actual guessing part as we guess which of the outgoing and incoming edges of a vertex, respectively, should be the ones

```
1 1 { s(X,Y) : edge(X,Y) } 1 :- vertex(X).
2 1 { s(X,Y) : edge(X,Y) } 1 :- vertex(Y).
3 r(Y) :- s(0,Y).
4 r(Y) :- s(X,Y), r(X).
5 :- vertex(X), not r(X).
6 minWeight(X,Y,MW) :- s(X,Y),
      MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
7 cost(C) :- C = #sum { W,X,Y : minWeight(X,Y,W) }.
8 #minimize { C : cost(C) }.
9 #show s/2.
```

Listing 3.2: TSP-NR encoding that relies on the reachability of all vertices [39].

```
1 Answer: 1
2 s(0,4) s(1,0) s(3,2) s(4,6) s(2,1) s(6,5) s(5,3)
3 Optimization: 18
4 Answer: 2
5 s(0,1) s(1,2) s(3,5) s(4,0) s(2,3) s(6,4) s(5,6)
6 Optimization: 18
```

Listing 3.3: Resulting optimal answer sets when applying the encoding from Listing 3.2 on the given instance.

selected to be part of the tour. In fact, each of the these two rules instantiates for each vertex exactly one `s(X,Y)` literal among the edges defined in the input instance. The first one does it for outgoing edges, whereas the second one for incoming ones. However, by merely guessing edges we obtain also solution candidates in which the selected ones do not constitute a tour so we will have to throw away the latter. In lines 3 and 4 we deduce that a vertex is reachable from the start vertex 0 if there exists a selected edge from 0 to the specific vertex, and that knowing that a vertex is reachable, and having selected an edge from this reachable vertex to another one, the latter is also reachable. Line 5 finally excludes all answer set candidates that contain at least one vertex not reachable from the starting vertex 0. In line 6 we handle the situation in which an edge has two different weights for its two reverse edges. As the encoding is meant for undirected graphs, we just take the smaller weight into consideration for edges that have been selected. Line 7 defines the objective function, using the `cost/1` predicate which uses an aggregate to sum up the lower weights for all selected edges, and which is then specified to be minimized in line 8, such that we obtain only the optimal tour(s) in the graph. Line 9 contains a directive for showing only the `s/2` predicate instances of the solution(s).

In Figure 3.2 we can see the optimal solution for the TSP-NR on the graph form Figure 3.1, while Listing 3.3 shows clingo's output when supplied with the encoding in Listing 3.2 and the instance in Figure 3.1 and Listing 3.1. Please note that the two answer sets actually represent the same tour.

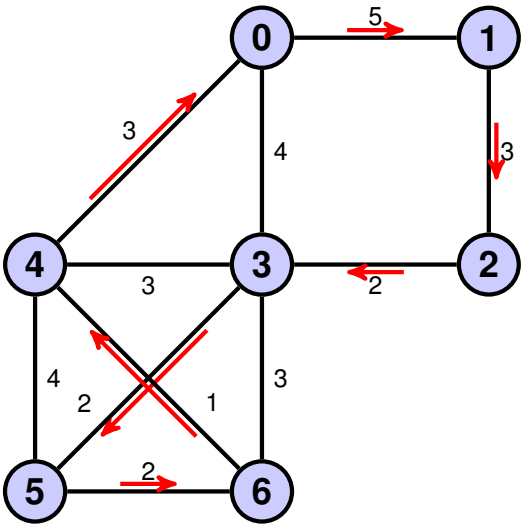


Figure 3.2: Input instance graph with optimal solution for TSP-NR.

ASP Encodings for the TSP-NR Relying on Connectedness

In Listing 3.4 we present a simple encoding that we developed, which also first finds Hamiltonian cycles in a graph in order to then find the one with the lowest cost, and uses the GCO-scheme [44] as well. However, the guessing is done here for each edge whether it shall be selected to be part of the tour, and not for each vertex, to then eliminate those candidates in which there exists at least one vertex not having exactly two selected adjacent edges, and those in which not all vertices are connected via the chosen tour.

In this encoding, and in the following two as well, the identifiers `s` and `c` stand for *selected* and *connected*, respectively. The first line guesses for each edge in the graph whether it shall be selected as part of the tour or not. This way, however, we receive answer set candidates in which some vertices might have more or less than two selected adjacent edges and some in which we obtain two or more tours rather than one connected tour. To rule out the first, in line 2 we count for each vertex how many selected edges are adjacent to the specific vertex using an aggregate, and if the result is different than 2, the solution is thrown away. Next, in lines 3, 4 and 5 we define the `c/2` predicate that states that there exists a path of selected edges between its two argument vertices. First, we say that if an edge between two vertices has been selected, they are also connected, and next we encode the predicate's reflexivity and transitivity properties. While it is trivial that the connectedness of two vertices in a graph is a transitive relation, we need the reflexivity for the situation when the order of the vertices in the `edge/2` predicate is arbitrary and hinders the rule in line 5 to fire. Finally, in line 6 we exclude also those solution candidates in which there exist at least two vertices which are not connected by a path of selected edges, in order not to have the graph's vertices distributed among several tours. The last four lines deduce the lower weight of each selected edge, define the objective function and contain directives for minimizing the latter and for showing the edges selected to comprise the optimal tour(s), as explained also on the encoding in Listing 3.2.

```
1  0 { s(X,Y) } 1 :- edge(X,Y).
2  :- N = #count { Y : s(X,Y); Y : s(Y,X) }, vertex(X), N != 2.
3  c(X,Y) :- s(X,Y).
4  c(Y,X) :- c(X,Y).
5  c(X,Z) :- c(X,Y), c(Y,Z).
6  :- not c(X,Y), vertex(X;Y).
7  minWeight(X,Y,MW) :- s(X,Y),
       MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
8  cost(C) :- C = #sum { W,X,Y : minWeight(X,Y,W) }.
9  #minimize { C : cost(C) }.
10 #show s/2.
```

Listing 3.4: Simple TSP-NR encoding that relies on the connectedness of a guessed tour.

```
1  Answer: 1
2  s(0,4) s(1,0) s(3,2) s(4,6) s(2,1) s(6,5) s(5,3)
3  Optimization: 18
4  Answer: 2
5  s(0,1) s(1,2) s(3,5) s(4,0) s(2,3) s(6,4) s(5,6)
6  Optimization: 18
```

Listing 3.5: Resulting optimal answer sets when applying the encoding from Listing 3.4 on the given instance.

Listing 3.5 shows clingo's output when supplied with the encoding in Listing 3.4 and the instance in Figure 3.1 and Listing 3.1, assuming that the edge (1,2) is the only one that has the same weight specified for both its `weight/3` facts. Normally, this encoding generates only one answer set for each optimal tour, unless there are edges in the tour that have the same weight specified by both its `weight/3` facts, such as edge (1,2), with `weight(1,2,3)` and `weight(2,1,3)`.

Next, we present the ASP encoding for the TSP-NR, which we will later compare in Chapter 5 to the D-FLAT encoding that will be shown in Chapter 4. Listing 3.6 shows a refined version of the ASP encoding relying on connectedness presented in Listing 3.4, the latter being faster than the one relying on reachability, presented in Listing 3.2. We will use this refined encoding for comparison in Chapter 5 as it is constructed following the same principles as the D-FLAT encodings we will show in Chapter 4. The main difference to the simple version is that we now take care that the order of the argument vertices in instantiations of the predicates `s/2` and `c/2` will always be ascending, supporting symmetry breaking (e.g. an edge selected between vertices 0 and 4 will always be denoted by `s(0,4)`, and not `s(4,0)`). This way, we have to implement more rules, but the program works faster and on less memory, as it creates less instantiations of the `c/2` predicate and less answer set candidates are generated.

In line 1 we guess for each edge whether it shall be selected for the tour or not. As we work on symmetric graphs we consider each edge exactly once by firing the rule for every instantiation

```
1  0 { s(X,Y) } 1 :- edge(X,Y), X < Y.
2  :- N = #count { Y : s(X,Y); Y : s(Y,X) }, vertex(X), N != 2.
3  c(X,Y) :- s(X,Y).
4  c(X,Z) :- c(X,Y), c(Y,Z).
5  c(X,Z) :- c(X,Y), c(Z,Y), X < Z.
6  c(Z,X) :- c(X,Y), c(Z,Y), X > Z.
7  c(X,Z) :- c(Y,X), c(Y,Z), X < Z.
8  c(Z,X) :- c(Y,X), c(Y,Z), X > Z.
9  :- not c(X,Y), vertex(X), vertex(Y), X < Y.
10 minWeight(X,Y,MW) :- s(X,Y),
       MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
11 cost(C) :- C = #sum { W,X,Y : minWeight(X,Y,W) }.
12 #minimize { C : cost(C) }.
13 #show s/2.
```

Listing 3.6: Refined TSP-NR encoding that relies on the connectedness of a guessed tour.

```
1  Answer: 1
2  s(0,1) s(0,4) s(1,2) s(2,3) s(3,5) s(4,6) s(5,6)
3  Optimization: 18
```

Listing 3.7: Resulting optimal answer set when applying the encoding from Listing 3.6 on the instance in Figure 3.1 and Listing 3.1.

of the `edge/2` predicate that has its first argument lower than the second one, and take care that for each `s/2` predicate instantiation, the arguments shall be ordered ascendantly. Line 2 again throws away those solution candidates in which at least one vertex has a different number of adjacent selected edges than 2. Next, in lines 3 to 8, we specify the `c/2` predicate. First, we say that if an edge between two vertices has been selected, they must also be connected. As we now work only with argument vertices ordered ascendantly we do not have the reflexivity relation defined anymore, but we need five rules to define the transitivity relation, one for each possible combination for the order of the argument vertices of the predicate instantiations of `c/2`. Line 9 again excludes those solution candidates which contain vertices not connected by a path of selected edges. The rules in lines 10 and 11 calculate the lower value for each selected edge and the total cost. Lines 12 and 13 contain directives for minimizing the objective function and for showing the selected edges, working as in the previous two encodings.

Finally, Listing 3.7 shows clingo's output when supplied with the encoding in Listing 3.6 and the instance in Figure 3.1 and Listing 3.1. One can observe that for each `s/2` predicate instantiation, its arguments are ordered in an ascendant manner and that for each tour clingo provides exactly one answer set.

```
1  vertex(0).
2  vertex(1).
3  vertex(2).
4  ...
5  edge(0,1).
6  edge(1,0).
7  edge(1,2).
8  edge(2,1).
9  ...
10 weight(0,1,5).
11 weight(1,0,7).
12 weight(1,2,3).
13 weight(2,1,3).
14 ...
15 minVisits(3,2).
16 maxVisits(3,2).
```

Listing 3.8: Excerpt from sample input instance for the TSP-R.

3.2 Monolithic ASP Encoding for the TSP-R

In this section we will present an ASP encoding for the TSP-R which is based on the encoding shown in the previous section in Listing 3.6. The encoding is also designed for input instances specifying an undirected graph by making use of the `vertex/1`, the `edge/2` and the `weight/3` predicates in the same manner. Besides these, the predicates `minVisits/2` and `maxVisits/2` specify the minimal and maximal number of visits for a vertex, respectively. If such predicates are not specified for a vertex, a default value of 1 is taken. Listing 3.8 shows an ASP encoding extract of such an input instance based on the graph in Figure 3.1, in which vertex 3 must be visited exactly twice. This leads to having a slightly more expensive optimal tour (0-1-2-3-4-6-5-3-0) with cost 22.

Additionally to the encoding in Listing 3.6, the one we present in Listing 3.9 takes care also of the vertices which have a different number of allowed minimal and maximal visits than the default value 1. One difference relies in the fact that, instead of discarding all answer set candidates that are not visited exactly once, we now throw away all those which are visited more times than the maximum or less than the minimum allowed. The other difference is that we now ensure connectedness only between those vertices that are actually selected to be part of the tour. The first is done by replacing the responsible constraint in Listing 3.6 with three constraints and six rules needed by the latter. The rules in lines 2 and 3 deduce for each vertex whether a minimum and maximum number of visits, respectively, was specified in the given instance. In lines 4 and 6 we instantiate the auxiliary predicate `minV/2` for each vertex. The arguments of the deduced instantiations are the respective vertex and 1 in case a minimum number of visits was not specified (line 4), or the vertex and the specified number of visits otherwise (line 6). The rules in lines 5 and 7 work in the same way for the maximum number of visits by instantiating the `maxV/2` predicate for each vertex. These auxiliary predicates are necessary for the constraints in lines 8 to 10, that eliminate all answer set candidates in which the number of visits does not correspond for at least one vertex. If we just instantiated `minVisits/2`,

```
0 { s(X,Y) } 1 :- edge(X,Y), X < Y.

minStated(X) :- vertex(X), minVisits(X,V).
maxStated(X) :- vertex(X), maxVisits(X,V).
minV(X,1) :- vertex(X), not minStated(X).
maxV(X,1) :- vertex(X), not maxStated(X).
minV(X,V) :- vertex(X), minVisits(X,V).
maxV(X,V) :- vertex(X), maxVisits(X,V).

:- N = #count { Y : s(X,Y); Y : s(Y,X) }, vertex(X), minVisits(X,V), N < 2*V.
:- N = #count { Y : s(X,Y); Y : s(Y,X) }, vertex(X), maxVisits(X,V), N > 2*V.
:- N = #count { Y : s(X,Y); Y : s(Y,X) }, vertex(X), N/2*2 != N.

c(X,Y) :- s(X,Y).
c(X,Z) :- c(X,Y), c(Y,Z).
c(X,Z) :- c(X,Y), c(Z,Y), X < Z.
c(Z,X) :- c(X,Y), c(Z,Y), X > Z.
c(X,Z) :- c(Y,X), c(Y,Z), X < Z.
c(Z,X) :- c(Y,X), c(Y,Z), X > Z.

:- not c(X,Y), s(X,_), s(Y,_), X < Y.
:- not c(X,Y), s(X,_), s(_,Y), X < Y.
:- not c(X,Y), s(_,X), s(Y,_), X < Y.
:- not c(X,Y), s(_,X), s(_,Y), X < Y.

minWeight(X,Y,MW) :- s(X,Y),
     MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
cost(C) :- C = #sum { W,X,Y : minWeight(X,Y,W) }.

#minimize { C : cost(C) }.
#show s/2.
```

Listing 3.9: TSP-R as a variation of the encoding in Listing 3.6.

```
Answer: 1
s(0,1) s(0,3) s(1,2) s(2,3) s(4,6) s(3,4) s(5,6) s(3,5)
Optimization: 22
```

Listing 3.10: Resulting optimal answer sets when applying the encoding from Listing 3.9 on the instance in Figure 3.1 and Listing 3.8.

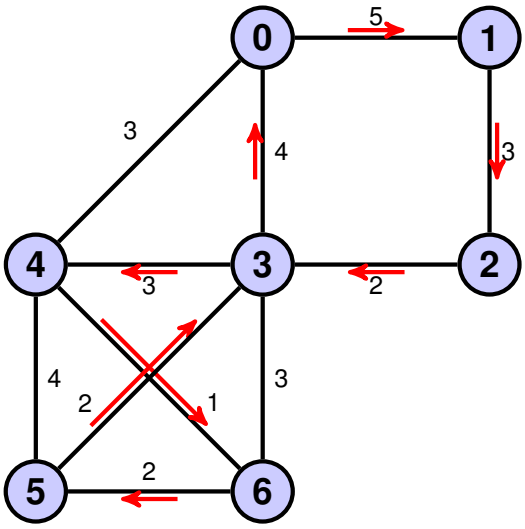


Figure 3.3: Input instance graph with optimal solution for TSP-R, where vertex 3 must be visited twice and the others only once.

or `maxVisits/2`, with the relevant vertex and 1, respectively, in case there was no lower or upper limit specified, we would obtain a cycle of dependency between `minStated/1` and `minVisits/2`, or `maxStated/1` and `maxVisits/2`, and the solver would eliminate all answer sets. Line 8 disposes all solution candidates in which there exists a vertex that has less selected adjacent edges than twice the number of minimum allowed visits, each visit being denoted by two selected adjacent edges. Line 9 again works in the same way for the maximum number of visits. Finally, line 10 throws away all solutions in which there exist vertices that have an odd number of selected adjacent edges, as this would not lead to a permutation but to an invalid result. The parity of the number of selected adjacent edges of a vertex is verified by checking whether dividing this number by 2 and again multiplying the result by 2 yields the same number, as ASP uses integer division. The second modification compared to the encoding in Listing 3.6 occurs at the elimination of unconnected answer set candidates. Instead of discarding those which have two vertices not connected to each other we now remove only those solution candidates where both of these vertices were also selected to be part of the tour (as it might not be necessary for some of the vertices to be visited). Essentially, this is one condition blown up to four constraints in lines 17 to 20 to cover all possible combinations for the positions of the vertex arguments in the `s/2` instantiations, for the sake of symmetry breaking. Please refer to the previous section for a detailed explanation for the rest of the code.

Figure 3.3 now displays the optimal solution for the TSP-R on the graph form Figure 3.1, where vertex 3 must be visited twice and the others only once, while in Listing 3.10 we can see clingo's resulting answer set when supplied with the encoding from Listing 3.9 and the instance in Figure 3.1 and Listing 3.8. It can be observed that vertex 3 has exactly four selected adjacent edges, meaning that it is visited twice.

CHAPTER 4

D-FLAT Implementations for the Traveling Salesperson Problem

In this chapter we will introduce our D-FLAT encodings for the TSP-NR and the TSP-R. They keep the structure of the encodings relying on connectedness in Listing 3.6 and Listing 3.9, respectively. Their complexity grows, the flexibility and maintainability representative for ASP are yet preserved. Furthermore, the running-time decreases considerably due to the dynamic programming approach as we will show in Chapter 5.

4.1 D-FLAT Encodings for the TSP-NR

The D-FLAT encodings for the TSP-NR (traveling salesperson problem without repetitions) work on the same instances as the monolithic encoding from Listing 3.6, expecting the input graph's vertices, edges and the weights of the latter to be specified by using the `vertex/1`, the `edge/2` and `weight/3` predicates, respectively. The `edge/2` predicate is expected to be given twice for each edge, by specifying two facts that are symmetric with regard to their argument vertices. The same holds for the `weight/3` predicate, without loss of generality. Listing 3.1 shows an extract of a possible input instance encoding using ASP.

The Dynamic Programming Concept

Although our D-FLAT encodings for the TSP-NR rely on the encoding from Listing 3.6, now we will be able to work only on the vertices that belong to the node (of the tree decomposition) that is currently processed, in order to comply with the principles of dynamic programming. This brings more complexity to the development and the understanding of the encodings, yet it is what drastically improves the performance, as we are now working on far smaller parts of the instance. The latter is accomplished by guessing and memorizing only the selection of edges between vertices of the current node, by using the item `s/2`, for *selected*. However, this will hinder us from checking whether a vertex has exactly two selected adjacent edges in the way

we did it in the proposed monolithic encoding, as it may happen that these two vertices that are adjacent to the vertex under focus are never in the same bag. The latter makes it useless to count the number of selected adjacent edges only on a node-level. Thus, we will use another item `ct/2`, to keep a *counter* for the selected adjacent edges of a vertex. This counter is kept from the bag or the bags where it is introduced to the last one before it is removed, including join nodes where we combine the vertices' counters from different branches by summing them up for each vertex and subtracting the value corresponding to common selected edges, by the inclusion-exclusion principle. Also, in join nodes we must check whether the edge selection between current vertices, which belonged also to the bags of the nodes right below the join node, is identical in the latter nodes. In the contrary case, the answer set candidate must be eliminated, as the distinct branches below the join node represent different solution candidates. Finally, if a vertex does not have exactly two selected adjacent edges at the time of removal, or exceeds this number already before it is removed, the group of solution candidates currently investigated cannot lead to a valid permutation without repetition, and the row is discarded.

Furthermore, we only deduce and memorize connectedness between two vertices if they are in the same bag, by using the item `c/2`, for *connected*. This again, makes it impossible to check the global connectedness on a node-level. We rather rely on the fact that, when a vertex that is not connected to all the vertices in its bag gets removed, the vertices which it was connected to last and which are still present in the next node above, will act like representatives for the removed vertex to eventually get connected also to the vertices it was not connected to at the time of removal. When such a vertex is removed, it can again pass on this responsibility to vertices it was last connected to. This process can continue until reaching the root's child node, in which all vertices must be connected to each other by selected edges. In case this condition is not fulfilled, the dynamic programming table row is discarded. However, passing the responsibility for fulfilling connectedness does not work if a removed vertex is not connected by selected edges to any other vertex in its bag at the time of removal, as it would be impossible to fulfill its connectedness to the rest of the vertices in the nodes above, for which reason we eliminate rows containing such situations.

The cost of a solution is calculated gradually, starting with the leaf nodes in which the cost is 0, adding the cost for each further selected edge in each node, and combining the costs from distinct branches in join nodes by the inclusion-exclusion principle, in order to finally receive the total cost of the resulting tour. Please note that the concept's explanations were specifically for the TSP-NR implementation for D-FLAT without default join on semi-normalized tree decompositions, from Listing A.3, the other implementations being slightly different but fundamentally complying with the concept .

The TSP-NR Implementation for D-FLAT without Default Join on Semi-Normalized Tree Decompositions

This implementation for the TSP-NR for the D-FLAT framework is called in each node of the tree decomposition and can be found in full version in the Appendix, in Listing A.3. It also follows the GCO-scheme [44], as it first guesses for each edge adjacent to an introduced vertex whether it should be selected for the tour or not, and then eliminates those answer set candidates

```
1 %dflat: -e vertex -e edge --tables -n semi
```

Listing 4.1: Modeline of TSP-NR encoding for D-FLAT without default join on semi-normalized tree decompositions.

```
1 %Guess row to be extended.
2 1 { extend(R): childRow(R,CH) } 1 :- childNode(CH).
3 %Guess introduced vertices' adjacent edges' selection for the tour.
4 0 { item(s(X,Y)) } 1 :- edge(X,Y), introduced(X), current(Y), X < Y.
5 0 { item(s(Y,X)) } 1 :- edge(X,Y), introduced(X), current(Y), X > Y.
6 item(s(X,Y)) :- childItem(R,s(X,Y)), extend(R), current(X;Y).
7 %Remove join nodes' table rows with different edge selection in distinct child nodes.
8 :- extend(R), extend(S), R!=S, childItem(R,s(X,Y)), not childItem(S,s(X,Y)).
```

Listing 4.2: TSP-NR encoding for D-FLAT with default join on semi-normalized tree decompositions: edge selection and propagation and partial solution matching at join nodes.

which cannot lead to a valid solution due to having a wrong number of selected adjacent edges or to the candidates' impossibility to fulfill the connectedness condition. The optimization is done gradually as we update the current cost with every node and as D-FLAT prefers always the cheaper solution candidate among the ones with the same item set and different costs.

Next, we will start explaining our encoding for the TSP-NR line by line. Listing 4.1 shows the modeline of this encoding, which instructs D-FLAT to create semi-normalized tree decompositions and use tables instead of item trees as our encoding is designed using D-FLAT's simplified interface. Further, we specify the unary hyperedge predicate `vertex/1` and binary hyperedge predicate `edge/2` for D-FLAT's internal hypergraph used for creating the tree decomposition.

Listing 4.2 discloses the actual guessing, the code passing up the information on the edge selection and the removal of solution candidates with different edge selections in distinct child nodes of a join node. In line 2, which is part of every table-mode encoding, we guess which row to extend for each child of the current node and deduce a fact `extend(R)`, where `R` is the child node's extended row. Will further use `extend(R)` to refer to a child node's specific answer set candidate. When a vertex is introduced into a node's bag, we guess for its outgoing and ingoing edges, which connect vertices that are in the same bag, whether to select them for the tour or not. Whether the introduced vertices' adjacent edges are ingoing or outgoing is determined by comparing the numbers which represent the names of the two vertices each of them connects. This is done in lines 4 and 5, where we introduce the predicate `s/2`, that states that the edge between the argument vertices is selected for the tour and whose arguments are always ordered in an ascending manner. The latter is necessary for symmetry breaking, by which means we do not create table rows with symmetric arguments of the `s/2` literal and shorten the running time by not having to handle redundant rows. More specifically, line 4 matches each edge having an introduced vertex as its first argument and another vertex of the current bag as its second argument, while the first argument must be lower than the second one, and guesses if the edge

```
1 %Count number of selected adjacent edges.
2 item(ct(X,N0)) :-  1 #count { CH : childNode(CH) } 1, introduced(X),
      N0 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
3 item(ct(X,N1+N0)) :- 1 #count { CH : childNode(CH) } 1,
      childItem(R,ct(X,N1)), extend(R), current(X),
      N0 = #count { Y : item(s(X,Y)), introduced(Y);
                    Y : item(s(Y,X)), introduced(Y) }.
4 item(ct(X,N1+N2-N12)) :- extend(R), extend(S), R!=S,
      childItem(R,ct(X,N1)), childItem(S,ct(X,N2)), current(X),
      N12 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
5 %Eliminate answer set candidates which do not lead to a permutation without repetition.
6 :- childItem(R,ct(X,N)), extend(R), removed(X), N != 2.
7 :- item(ct(X,N)), N > 2.
```

Listing 4.3: TSP-NR encoding for D-FLAT without default join on semi-normalized tree decompositions: Determining and eliminating table rows which would not lead to valid permutations.

between these two vertices should be selected or not. For the case it is, we specify the `s/2` literals to be in the item set of the node in order to be able to access the information from parent nodes. Line 5 works in a similar way. In line 6 we learn from child nodes between which current vertices there has already been decided an edge to be selected. We placed `extends(R)` into the rule body as we need the reference for retrieving the items from the extended rows of the child node. Finally, the rule in line 7 eliminates join nodes' answer set candidates which have distinct child nodes with a different edge selection. More particularly, we throw away those table rows where in one child node an edge between two current vertices, which belong also to the child nodes in semi-normalized tree decompositions, has been selected and in another one not, as this would mean propagating an answer set candidate that would actually incorporate two different solution candidates. The condition whether we are in a join node is checked by matching two different instantiations of the `extend/1` predicate, as we always instantiate exactly one for each child node.

In Listing 4.3 we instantiate and keep the counter, denoted by the predicate `ct/2`, whose arguments are the current vertex whose counter we are keeping and the number of selected adjacent edges. The rule in line 2 instantiates the counter for vertices newly introduced into bags of exchange nodes. For determining if we are dealing with an exchange node, we check if it has exactly one child node. The counter is set to the number of edges that go into and out of the introduced vertex. Also the rule in line 3 fires only in exchange nodes. However, here we update the counter of current vertices that are not newly introduced, which is implied by the reference to the child node's `ct/2` item. Thus, we have to add the value of the vertex' counter in the child node, retrieved from the `ct/2` reference, to the number of selected edges, which connect the vertex under focus with newly introduced vertices. If we counted all vertices, not only the newly introduced ones, we would do it multiple times, as they have already been counted in the child node. Line 4 updates the counter for vertices in a join node, the latter condition being checked again by matching two different instantiations of the `extend/1` predicate. As our join node's bag is always identical to the bags of its child nodes, we do not have to mind about newly

```
1  %Deduce connectedness.
2  item(c(X,Y)) :- item(s(X,Y)).
3  item(c(X,Z)) :- item(c(X,Y)), item(c(Y,Z)).
4  item(c(X,Z)) :- item(c(X,Y)), item(c(Z,Y)), X < Z.
5  item(c(Z,X)) :- item(c(X,Y)), item(c(Z,Y)), X > Z.
6  item(c(X,Z)) :- item(c(Y,X)), item(c(Y,Z)), X < Z.
7  item(c(Z,X)) :- item(c(Y,X)), item(c(Y,Z)), X > Z.
8  item(c(X,Y)) :- childItem(R,c(X,Y)), extend(R), current(X;Y).
9  %Eliminate unconnected answer set candidates.
10 :- 1 #count{ U : current(U) }, removed(X), extend(R),
       not childItem(R,c(X,Y)) : current(Y);
       not childItem(R,c(Y,X)) : current(Y).
11 :- final, not childItem(R,c(X,Y)), X < Y, extend(R), removed(X;Y).
12 %Optional code for the case that the input graph's connectedness was not verified.
13 :- 1 #count { X : bag(N,X), childNode(N) }, not final, not oldVertex.
14 oldVertex :- current(X), not introduced(X).
```

Listing 4.4: TSP-NR encoding for D-FLAT without default join on semi-normalized tree decompositions: Ensuring connectedness.

introduced vertices, but only to sum up each vertex' counters from the child nodes, and subtract the number of selected edges through which the vertex under focus is adjacent to other current vertices, by the inclusion-exclusion principle. The latter would be counted twice, once for each child node, if they were not subtracted.

Finally, we remove all solution candidates which cannot lead to a permutation. In line 6 from Listing 4.3 we eliminate those rows in which a removed vertex had a counter with a value different than 2 in the child node. The latter would mean that we would not be able to obtain a permutation without repetition as the removed vertex would not have exactly one predecessor and one successor. Finally, the constraint in line 7 is not necessary for obtaining the correct result but it lowers the running time by eliminating rows in which a vertex has its counter set to a value higher than 2, already before it is removed.

Listing 4.4 displays the part of our encoding which is responsible for ensuring the connectedness of the output tour. Line 2 states that in the case in which an edge between two vertices of the current node's bag is selected for the tour, the former vertices are also connected, and introduces the `c/2` predicate, whose instantiations are also items, as they must be retrieved from child nodes sometimes. Similarly to the encoding in Listing 3.6, lines 3 to 7 describe the transitivity property of the connectedness, by matching any possible combination of having three vertices among which one (`Y`) is connected to both others (`X` and `Z`) and deducing that the two other vertices must also be connected. Please note that the `c/2` predicate also has its arguments stated in an ascending manner, supporting symmetry breaking and avoiding the creation of redundant table rows. In line 8 we pass the item `c/2` from a child node in case the two argument vertices have not been removed. This is important for keeping the `c/2` predicate sound, as a link vertex between two other vertices could be removed while keeping the two connected vertices in the bag when moving on to the next node above.

In order to guarantee the connectedness of the output we have to first ensure connectedness

on a node-level, i.e. at the removal of a vertex from a bag it must be connected to at least another vertex in the same bag, and at two vertices' removal from the bag of an empty node's child, they must be connected to each other. In this manner, a removed vertex is either already connected to all other vertices in the bag or it will be connected after being removed by means of other link vertices it is connected to at the time of removal. Assuming our input graph is connected, when a vertex is connected to all the others in the bags it is part of, it is connected also to all other vertices of the given input instance, as all vertices connected by an edge must appear in the same bag at least once, according to the second condition of Definition 2.3.1, the definition of tree decomposition. Connectedness on a node-level is ensured in lines 10 and 11. In line 10 we eliminate all solution candidates in which a vertex is removed from the bag and was not connected to any other vertex in the same child bag. It is important to set the condition to have a non-empty current bag as the solver would always consider the last two conditions true in an empty bag. However, the situation when an isolated vertex is removed and the current bag is empty, is covered by the constraint in line 11, where we remove all solution candidates that have two unconnected vertices in the root's child node. Secondly, for guaranteeing connectedness, we must ensure this property also between the different nodes of the tree decomposition. In a connected graph this would not be an issue, as connectedness on a node-level would also imply connectedness between nodes, due to the second condition of Definition 2.3.1.

We can assume that the input graphs are connected, as this can be checked by a depth-first search in linear time [55]. However, we specified some rules also for dealing with an unconnected graph. In the latter case it can happen that a current node's bag and its child node's bag are disjoint, which would lead to a solution consisting of two or more tours instead of one, as there would be a rupture between those two nodes. To prevent this situation we discard all solution candidates in which there exists a node, which is not a root or a parent of a leaf and that has completely different vertices than its child node. We exclude the root and the parent nodes of the leaves, as the root and the leaves are always empty and the rule would otherwise fire even if there was no real rupture between these nodes and their child or parent nodes, respectively. In line 14 we deduce if there exists a vertex that was already in the child node's bag, while the rule in line 13 is fired when there is no such vertex and neither is the child node a leaf node, nor is the current node a root node. It is important to exclude the last two situations as in these cases `oldVertex` is always deduced, as root and leaf nodes are empty, but we can still obtain a valid solution.

In Listing 4.5 we present the final part of our code, the cost calculation. In line 2 we set the cost of the leaf nodes to 0. Next, we handle all other nodes in which the ASP solver is called, namely exchange and join nodes. The rule in line 3 deals with exchange nodes and adds the child node's cost to the cost of the edges between newly introduced vertices. The latter is calculated by aggregating over the weights of all these edges with the help of the `relevantWeight/3` predicate. Literals of the latter are deduced in line 5 and represent the minimum weight of a selected edge, among its two specified weights. Please note that in line 2 an edge between two newly introduced vertices is not counted twice as there is no differentiator among the variables before the semicolon inside the aggregator function. Finally, in line 4, we calculate the cost for join nodes by adding up both child nodes' costs and subtracting the sum of all costs of selected edges between current vertices, by the inclusion-exclusion principle.

```
1 %Calculate costs.
2 cost(0) :- initial.
3 cost(CC+NC) :- childCost(R,CC), extend(R),
      1 #count { CH : childNode(CH) } 1,
      NC = #sum { W,X,Y : relevantWeight(X,Y,W), introduced(X);
                  W,X,Y : relevantWeight(X,Y,W), introduced(Y) }.
4 cost(CC1+CC2-LC) :- extend(R), extend(S), R < S,
      childCost(R,CC1), childCost(S,CC2),
      LC = #sum { W,X,Y : relevantWeight(X,Y,W) }.
5 relevantWeight(X,Y,MW) :- item(s(X,Y)),
      MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
```

Listing 4.5: TSP-NR encoding for D-FLAT without default join on semi-normalized tree decompositions: Calculating the costs.

```
1 Answer: 1
2 c(0,1) c(0,2) c(0,3) c(0,4) c(1,2) c(2,3) c(3,4) c(3,5) c(3,6) c(4,5) c(4,6)
      c(5,6) ct(0,1) ct(0,2) ct(1,2) ct(2,1) ct(2,2) ct(3,1) ct(3,2) ct(4,1)
      ct(4,2) ct(5,2) ct(6,2) s(0,1) s(0,4) s(1,2) s(2,3) s(3,5) s(4,6) s(5,6)
3 (cost: 18)
```

Listing 4.6: Possible resulting optimal answer set when supplied with the encoding presented in Listing A.3 and the instance in Figure 3.1 and Listing 3.1.

Finally, Listing 4.6 displays D-FLAT's possible output when applying the encoding in Listing A.3 from the Appendix on the instance in Figure 3.1 and Listing 3.1. While the instantiations of the `s/2` predicate are always the same, those of `c/2` and `ct/2` may vary depending on the tree decomposition created by D-FLAT. One can observe that all items deduced while traversing the tree decomposition are displayed and that the `s/2` facts coincide with those in the output of the monolithic encoding for the TSP-NR, in Listing 3.7.

The TSP-NR Implementation for D-FLAT with Default Join on Semi-Normalized Tree Decompositions

The D-FLAT framework offers the `--default-join` option on whose activation it does not call the ASP solver in join nodes, but matches their child nodes' table rows on their items. If the item sets are identical, the solution candidate is kept. Furthermore, it calculates the cost of the join node automatically by following the inclusion-exclusion principle. The use of the default join shows great running time improvements as compared to calling D-FLAT without this option and also an enhancement of the memory consumption. This can be put down to the fact that the joining of two branches with the `--default-join` option is implemented in C++ and makes use of a certain order of rows in tables, similarly to the sort-merge join algorithm in relational database management systems [1], as compared to D-FLAT's classical joining in which the grounding is immense. The encoding we present next is an adaptation of the previously disclosed encoding for D-FLAT to take over the join node handling by using the `--default-join` option and can be entirely found in the Appendix, in Listing A.4. With

```
1 %dflat: -e vertex -e edge --tables -n semi --default-join --post-join
```

Listing 4.7: Modeline of TSP-NR encoding for D-FLAT with default join on semi-normalized tree decompositions.

```
1 %Guess row to be extended.
2 1 { extend(R): childRow(R,CH) } 1 :- childNode(CH).
3 %Guess introduced vertices' adjacent edges' selection for the tour.
4 0 { item(s(X,Y)) } 1 :- edge(X,Y), introduced(X), current(Y), X < Y.
5 0 { item(s(Y,X)) } 1 :- edge(X,Y), introduced(X), current(Y), X > Y.
6 item(s(X,Y)) :- childItem(R,s(X,Y)), extend(R), current(X;Y).
```

Listing 4.8: TSP-NR encoding for D-FLAT with default join on semi-normalized tree decompositions: edge selection and propagation.

the use of the default join, we will discriminate between `items` and `auxItems`, as D-FLAT matches items at join nodes and we still need to refer to child node instantiations which should not be matched upon, such as those of the `c/2` predicate. Rules for join nodes are not necessary any more because the ASP solver is not called for these. Yet, we will need to do some post-processing, for setting the counter for the number of selected adjacent edges of a vertex right, for which purpose we will use the `--post-join` option, making sure that there is a node with identical bag contents above every join node which the post-processing is for. In Listing 4.7 we can also see the specification of these two options we are using.

In Listing 4.8 we again present the edge selection in a large sense: from the actual edge selection to passing up the information on the latter. The rule that handles matching on `s/2` instantiations at join nodes from the code in Listing 4.2 is not necessary any more as this is taken care of by D-FLAT's default join. In this encoding, instantiations of the `s/2` predicate must again be part of the item set, not only for being able to access them from parent nodes, but also because the matching of items at join nodes is done specifically on these.

The code in Listing 4.9 works similarly to the one in Listing 4.3. However, here we work with a counter predicate `ct/3` with three arguments. Besides the current vertex whose counter we are keeping, and the number of selected adjacent edges, we now memorize also the node it stems from, which is necessary when adding up the counters from two different branches in the post-processing node above a join node. The third argument of `ct/3` is always set to be the current node, except for the case of join nodes, to which the items and auxiliary items are passed upwards just as they are, such that we can determine in the post-processing node above the join node which of the former's children the literal stems from. Lines 2, 3 and 4 handle newly introduced vertices into exchange nodes that are not post-processing nodes, vertices that were already part of the child node, also in such exchange nodes, and post-processing nodes, respectively. To determine if we are at an exchange node which is not a post-processing node, we use the auxiliary item `n/1` that states which is the current node (as deduced in line 5), by counting the number of its different instantiations in the bag of the current node's child, in lines 2 and 3. If there were more than one, it would mean the child node was a join node to which

```
1 %Count number of selected adjacent edges.
2 auxItem(ct(X,N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
      extend(R), introduced(X), currentNode(CR),
      N0 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
3 auxItem(ct(X,N1+N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
      extend(R), current(X), currentNode(CR),
      childAuxItem(R,ct(X,N1,CH1)),
      N0 = #count { Y : item(s(X,Y)), introduced(Y);
                    Y : item(s(Y,X)), introduced(Y) }.
4 auxItem(ct(X,N1+N2-N12,CR)) :-
      childAuxItem(R,ct(X,N1,CH1)), childAuxItem(R,ct(X,N2,CH2)), CH1 != CH2,
      extend(R), currentNode(CR),
      N12 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
5 auxItem(n(CR)) :- currentNode(CR).
6 %Eliminate answer set candidates which do not lead to a permutation without repetition.
7 :- childAuxItem(R,ct(X,N,CH)), extend(R), removed(X), N != 2.
8 :- auxItem(ct(X,N,_)), N > 2.
```

Listing 4.9: TSP-NR encoding for D-FLAT with default join: Determining and eliminating table rows which would not lead to valid permutations.

all items and auxiliary items, including the instantiations of `n/1`, were passed unchanged from its child nodes. Thus, if there is only one instantiation, we know for sure that we are not in a post-processing node. The counter is again set to the number of edges that go into and out of the introduced vertex. The rule in line 3 updates the counter of current vertices that are not newly introduced, which is implied by the presence of the auxiliary item `ct/3` of the child node. Line 4 updates the counter for vertices in a post-processing node, and not for join nodes as it did in the previous encoding. We verify if we are dealing with a post-processing node by matching two instantiations of the `ct/3` item in the child node, which must be a join node, with different arguments for the current node. As our post-processing node is always identical to its child node, we do not have to mind about newly introduced vertices but only to sum up each vertex' counter from the child join node, received in turn from the latter's child nodes, and subtract the number of selected edges through which the vertex under focus is adjacent to other current nodes, by the inclusion-exclusion principle. Here, the condition `current(X)` would be superfluous, as we know that `X` was already part of the bags of the join node's children and that the current post-processing node is identical to those.

The elimination of answer set candidates which would not lead to a permutation without repetition is done in the same way as in Listing 4.3.

In Listing 4.10, we present the part of our encoding which is responsible for ensuring connectedness and functions in the same manner as the one in Listing 4.4. However, now the `c/2` literals are just auxiliary items as we do not join on them, but rather need to know in post-processing nodes which additional connectedness relation we can deduce from the relations in the join node's children.

Next, in Listing 4.11, we disclose the final part of the encoding: the cost calculation. The only modifications to the code in Listing 4.5 are the fact that we now have to distinguish only between exchange and leaf nodes and the fact that we do not need to calculate the cost of join nodes any more as this is taken over by D-FLAT due to the use of the `--default-join`

```
1  %Deduce connectedness.
2  auxItem(c(X,Y)) :- item(s(X,Y)).
3  auxItem(c(X,Z)) :- auxItem(c(X,Y)), auxItem(c(Y,Z)).
4  auxItem(c(X,Z)) :- auxItem(c(X,Y)), auxItem(c(Z,Y)), X < Z.
5  auxItem(c(Z,X)) :- auxItem(c(X,Y)), auxItem(c(Z,Y)), X > Z.
6  auxItem(c(X,Z)) :- auxItem(c(Y,X)), auxItem(c(Y,Z)), X < Z.
7  auxItem(c(Z,X)) :- auxItem(c(Y,X)), auxItem(c(Y,Z)), X > Z.
8  auxItem(c(X,Y)) :- childAuxItem(R,c(X,Y)), extend(R), current(X;Y).
9  %Eliminate unconnected answer set candidates.
10 :- 1 #count{ U : current(U) }, removed(X), extend(R),
       not childAuxItem(R,c(X,Y)) : current(Y);
       not childAuxItem(R,c(Y,X)) : current(Y).
11 :- final, not childAuxItem(R,c(X,Y)), X < Y, extend(R), removed(X;Y).
12 %Optional code for the case that the input graph's connectedness was not verified.
13 :- 1 #count { X : bag(N,X), childNode(N) }, not final, not oldVertex.
14 oldVertex :- current(X), not introduced(X).
```

Listing 4.10: TSP-NR encoding for D-FLAT with default join: Ensuring connectedness.

```
1  %Calculate costs.
2  cost(0) :- initial.
3  cost(CC+NC) :- not initial, childCost(R,CC), extend(R),
       NC = #sum { W,X,Y : relevantWeight(X,Y,W), introduced(X);
                   W,X,Y : relevantWeight(X,Y,W), introduced(Y) }.
4  currentCost(C) :- C = #sum { W,X,Y : relevantWeight(X,Y,W) }.
5  relevantWeight(X,Y,MW) :- item(s(X,Y)),
       MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
```

Listing 4.11: TSP-NR encoding for D-FLAT with default join: Calculating the costs.

```
1  Answer: 1
2  c(0,1) c(0,2) c(0,3) c(0,4) c(1,2) c(2,3) c(3,4) c(3,5) c(3,6) c(4,5) c(4,6)
       c(5,6) ct(0,1,4) ct(0,1,5) ct(0,2,3) ct(1,2,5) ct(2,1,5) ct(2,2,4)
       ct(3,1,3) ct(3,1,4) ct(3,2,2) ct(4,1,3) ct(4,2,2) ct(5,2,2) ct(6,2,2)
       n(1) n(2) n(3) n(4) n(5) n(6) s(0,1) s(0,4) s(1,2) s(2,3) s(3,5) s(4,6)
       s(5,6)
3  (cost: 18)
```

Listing 4.12: Possible resulting optimal answer set when supplied with the encoding presented in Listing A.4 and the instance in Figure 3.1 and Listing 3.1.

option. Instead we must specify the `currentCost/1` predicate (line 4) which represents the sum of all weights of selected edges between current vertices and is used by D-FLAT's default join to subtract the common weight from the sums of the join nodes' children's costs, by the inclusion-exclusion principle.

Finally, in Listing 4.12 we show D-FLAT's possible output when applying the encoding in Listing A.4 on the instance in Figure 3.1 and Listing 3.1. We can notice that it contains the same

```
1 %dflat: -e vertex -e edge --tables -n weak --default-join --post-join
```

Listing 4.13: Modeline of TSP-NR encoding for D-FLAT with default join on weakly normalized tree decompositions.

```
1 %Count number of selected adjacent edges.
2 auxItem(ct(X,N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
      extend(R), introduced(X), currentNode(CR),
      N0 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
3 auxItem(ct(X,N1+N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
      extend(R), current(X), currentNode(CR),
      childAuxItem(R,ct(X,N1,CH1)),
      N0 = #count { Y : item(s(X,Y)), introduced(Y);
                    Y : item(s(Y,X)), introduced(Y) }.
4 auxItem(ct(X,NC-((NCH-1)*N12),CR)) :- current(X), currentNode(CR),
      NCH = #count { CH: childAuxItem(R,n(CH)) }, extend(R), NCH > 1,
      NC = #sum { N,CH: childAuxItem(R,ct(X,N,CH))},
      N12 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
5 auxItem(n(CR)) :- currentNode(CR).
6 %Eliminate answer set candidates which do not lead to a permutation without repetition.
7 :- childAuxItem(R,ct(X,N,CH)), extend(R), removed(X), N != 2.
8 :- auxItem(ct(X,N,_)), N > 2.
```

Listing 4.14: TSP-NR encoding for D-FLAT with default join on weakly normalized tree decompositions: Determining and eliminating table rows which would not lead to valid permutations.

`s/2` items as the one from the previous encoding.

The TSP-NR Implementation for D-FLAT with Default Join on Weakly Normalized Tree Decompositions

Next, we will present a D-FLAT encoding for the TSP-NR, which works not only on semi-normalized tree decompositions but also on weakly normalized ones and also makes use of the default join. The whole encoding can be found in the Appendix, in Listing A.5. It works in exactly the same way as the encoding in Listing A.4, except when it updates the counter in post-processing nodes. Ensuring the connectedness is not affected by having auxiliary items from more children of the join nodes in the post-processing nodes as we do not operate directly on join nodes due to the use of the `--default-join` option, and the cost calculation also stays unchanged as the default join automatically calculates a join node's cost. More specifically, the modeline in Listing 4.7 and the rule in line 4 of Listing 4.9, the latter being meant for two children for each join node, must be adapted.

In the modeline, solely the option stating the normalization type changes from `semi` to `weak`, as shown in Listing 4.13.

Listing 4.14 displays the part of the encoding for weakly normalized trees which instantiates and keeps each edge's counter for selected adjacent edges. The rule in line 4 was adapted as

it applied the inclusion-exclusion principle only for two unified sets and now it works for an unlimited number of the latter. Now we also have to match the current vertex as we do not refer any more to the counter of the same vertex in the child node. We check whether we are in a post-processing node by counting the number of instantiations of predicate `n/1` in the child node, which coincides with the number of children the child node has, the latter being a join node. At the same time we remember this number. We also calculate the sum of all counters' values of the desired vertex in the child node and the number of selected adjacent edges in the current node. The new value of the counter is set to the sum of all counters' values in the child node, corresponding to the counter's value in each of the join node's children, minus the number of common edges, that corresponds to the number of selected adjacent edges in the current node, times the number of the join node's children minus one, in order not to count any edge multiple times, by the inclusion-exclusion principle.

Please note that this encoding also works on semi-normalized trees and that the output might not be identical when working on different tree decompositions, but the `s/2` items remain the same.

4.2 D-FLAT Encoding for the TSP-R

In this section we will prove the high flexibility and maintainability when working with D-FLAT, by proposing an encoding for the TSP that allows to specify for any city how often it may be visited, i.e. an encoding for the TSP-R (traveling salesperson with repetitions). Our implementation keeps the structure and the largest part of the encoding for the TSP-NR presented in Listing A.4, and can also be found entirely in the Appendix, in Listing A.6. The only adaptations are done at eliminating those answer set candidates that would not lead to valid solutions, as we are now looking for a permutation with repetition that complies with the specifications in the input instance, and at eliminating unconnected solution candidates, where we want to remove only those in which only an already selected vertex is not connected to all other vertices of the tour .

Our D-FLAT encoding for the TSP-R also works on the same instances as the monolithic encoding from Listing 3.9, expecting the undirected input graph's vertices, edges and the weights of the latter to be specified in the same manner by using the `vertex/1`, the `edge/2` and `weight/3` predicates, respectively, and the `minVisits/2` and `maxVisits/2` predicates to specify the minimal and maximal number of visits for a vertex, respectively, in case the default value 1 is not wanted. Listing 3.1 shows an ASP encoding extract of such an input instance based on the graph in Figure 3.1, in which vertex 3 must be visited exactly twice. Next, we will present the two excerpts of the TSP-R encoding which contain modifications to the TSP-NR encoding in Listing A.4.

In Listing 4.15 we present the excerpt of the TSP-R encoding that counts the number of selected adjacent edges and then eliminates those answer set candidates which do not comply in this regard. The first part (lines 2 to 5) is identical to the one in Listing 4.9. In fact, the code which is relevant for the TSP-R consists of the constraints responsible for eliminating answer set candidates that would not lead to a permutation with repetition and their auxiliary rules. The former replace similar constraints to be seen in Listing 4.9. Instead of eliminating those solution

```
1  %Count number of selected adjacent edges.
2  auxItem(ct(X,N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
       extend(R), introduced(X), currentNode(CR),
       N0 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
3  auxItem(ct(X,N1+N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
       extend(R), current(X), currentNode(CR),
       childAuxItem(R,ct(X,N1,CH1)),
       N0 = #count { Y : item(s(X,Y)), introduced(Y);
                     Y : item(s(Y,X)), introduced(Y) }.
4  item(ct(X,N1+N2-N12)) :- extend(R), extend(S), R!=S,
       childItem(R,ct(X,N1)), childItem(S,ct(X,N2)), current(X),
       N12 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
5  auxItem(n(CR)) :- currentNode(CR).
6  %Eliminate solution candidates that do not lead to a valid permutation with repetition.
7  :- childAuxItem(R,ct(X,N,CH)), extend(R), minV(X,V), removed(X), N < 2 * V.
8  :- childAuxItem(R,ct(X,N,CH)), extend(R), maxV(X,V), removed(X), N > 2 * V.
9  :- childAuxItem(R,ct(X,N,CH)), extend(R), removed(X), N / 2 * 2 != N.
10 :- maxV(X,V), auxItem(ct(X,N,_)), N > 2 * V.
11 minV(X,1) :- currem(X), not minStated(X).
12 maxV(X,1) :- currem(X), not maxStated(X).
13 minStated(X) :- currem(X), minVisits(X,V).
14 maxStated(X) :- currem(X), maxVisits(X,V).
15 minV(X,V) :- currem(X), minVisits(X,V).
16 maxV(X,V) :- currem(X), maxVisits(X,V).
17 currem(X) :- current(X).
18 currem(X) :- removed(X).
```

Listing 4.15: TSP-R encoding for D-FLAT with default join on semi-normalized tree decompositions: Eliminating table rows which would not lead to valid permutations.

candidates where the number of the selected adjacent edges of a vertex is different than 2 when they are removed, or exceeds 2 before being removed, we now eliminate those where this number is lower or higher than the value allowed. Further, we eliminate those solution candidates where this number is odd at removal, or if it is higher already before removal. For not having to differentiate between the vertices for which `minVisits/2` or `maxVisits/2` is specified, and those for which it is not, and in order not to obtain cycles of dependency, we introduce the predicates `minV/2` and `maxV/2`, similarly to the way we do it in the monolithic encoding in Listing 3.6. These are set for a vertex to 1, on the second position, if a minimum (line 11) or maximum (line 12) number of visits was not specified for the relevant vertex, and to the number of minimum (line 15) or maximum (line 16) specified visits otherwise. Whether the latter were specified is checked by means of the `minStated/1` and `maxStated/1` predicates, derived in lines 13 and 14 only for those vertices for which a minimum, or maximum, number of visits was specified in the input, respectively. All rules in lines 11 to 16 are fired only when dealing with a vertex that is in the current bag or one that has just been removed, as defined in lines 17 and 18, as these are the only vertices for which a constraint can be applied. Please note that all these rules in lines 11 to 18 actually handle the default case for the minimum and maximum number of allowed visits, and could be omitted if the input instances defined a minimum and

```
1  %Deduce connectedness.
2  auxItem(c(X,Y)) :- item(s(X,Y)).
3  auxItem(c(X,Z)) :- auxItem(c(X,Y)), auxItem(c(Y,Z)).
4  auxItem(c(X,Z)) :- auxItem(c(X,Y)), auxItem(c(Z,Y)), X < Z.
5  auxItem(c(Z,X)) :- auxItem(c(X,Y)), auxItem(c(Z,Y)), X > Z.
6  auxItem(c(X,Z)) :- auxItem(c(Y,X)), auxItem(c(Y,Z)), X < Z.
7  auxItem(c(Z,X)) :- auxItem(c(Y,X)), auxItem(c(Y,Z)), X > Z.
8  auxItem(c(X,Y)) :- childAuxItem(R,c(X,Y)), extend(R), current(X;Y).
9  %Eliminate unconnected answer set candidates.
10 :- 1 #count{ U : current(U) }, removed(X),
       not childAuxItem(R,ct(X,0,_)), extend(R),
       not childAuxItem(R,c(X,Y)) : current(Y);
       not childAuxItem(R,c(Y,X)) : current(Y).
11 :- final, removed(X;Y), X < Y, not childAuxItem(R,c(X,Y)), extend(R),
       not childAuxItem(R,ct(X,0,_)), not childAuxItem(R,ct(Y,0,_)).
12 %Optional code for the case that the input graph's connectedness was not verified.
13 auxItem(noselection(CR)) :- 1 #count { X : bag(N,X), childNode(N) },
       not final, not oldVertex, auxItem(selection), currentNode(CR).
14 oldVertex :- current(X), not introduced(X).
15 auxItem(selection) :- item(s(X,Y)).
16 auxItem(noselection(CR)) :- childAuxItem(R,noselection(CH)),
       extend(R), currentNode(CR).
17 auxItem(selection) :- childAuxItem(R,selection),
       not auxItem(noselection(CR)), extend(R).
18 :- auxItem(selection), auxItem(noselection(CR)).
19 :- auxItem(noselection(CH1)), auxItem(noselection(CH2)), CH1 < CH2.
```

Listing 4.16: TSP-R encoding for D-FLAT with default join: Ensuring connectedness.

a maximum number of allowed visits for each vertex. However, by providing these extra rules we simplify the formulation of input instances. In lines 7 and 8 we eliminate those table rows which contain at least one vertex that has less, or more, selected adjacent edges, than twice the minimum, or maximum, number of allowed visits, respectively. As we check only the minimum and maximum number of the vertices' selected adjacent edges we have to eliminate also those cases in which a vertex has an odd number of the latter at removal, in which situation we would not even obtain a permutation at all. This is done in line 9 by checking whether dividing this number by 2 and again multiplying the result by 2 yields the same number, as ASP uses integer division. Finally, in line 10, we remove those solution candidates which contain a vertex that already has more selected adjacent edges than twice the specified number of allowed visits before removal.

Listing 4.16 shows the other excerpt that had to be complemented in order to comply with the specifications of the TSP-R. Again, it is the constraints which needed to be adapted. The constraints in lines 10 and 11 are now fired only if the removed vertices, which were not connected to other vertices they were supposed to, were actually selected to be part of the tour, as in the TSP-R it is possible, or even mandatory for certain vertices not to be visited, depending on the input instance. We verify this condition by checking whether the removed vertices' counters were different from 0.

```
1 Answer: 1
2 c(0,1) c(0,2) c(0,3) c(0,4) c(1,2) c(2,3) c(3,4) c(3,5) c(3,6) c(4,5) c(4,6)
      c(5,6) ct(0,1,5) ct(0,2,3) ct(0,2,4) ct(1,2,5) ct(2,1,5) ct(2,2,4)
      ct(3,2,4) ct(3,3,3) ct(3,4,2) ct(4,1,3) ct(4,2,2) ct(5,2,2) ct(6,2,2)
      n(1) n(2) n(3) n(4) n(5) n(6) s(0,1) s(0,3) s(1,2) s(2,3) s(3,4) s(3,5)
      s(4,6) s(5,6) selection
3 (cost: 22)
```

Listing 4.17: Possible resulting optimal answer set when supplied with the encoding presented in Listing A.6 and the instance in Figure 3.1 and Listing 3.8.

Lines 13 to 19 ensure connectedness between nodes and are only optional as there are efficient algorithms for determining a graph's connectivity beforehand. While, in Listing 4.4 the situation in which a parent node had a completely different bag than its child node, meant that the graph was disconnected and the solution candidate should be discarded, now we keep it if all nodes below the rupture, or all of those above it (including nodes below join nodes which are situated above the rupture), contain only vertices that were not selected for the tour. This corresponds to the situation in which the input graph is not connected and all vertices selected for the tour belong to the same component. In such a case, connectedness between nodes is not mandatory. To handle such situations we introduce the auxiliary items `selection/0` and `noselection/1`. The first one marks that at least one edge between vertices from the same node has been selected for the tour (line 15), while the second marks that from the node, where it was first deduced upwards, no more edges must be selected, or the solution candidate must be discarded. Both are passed upwards from node to node once they were deduced, in lines 16 and 17, with the remark that `selection/0` stops being propagated from the point where `noselection/1` was deduced. In case such a rupture, where the child node's bag is disjoint to the current node's bag, is encountered, and `selection/0` has already been deduced, implying that an edge has already been selected, it follows that we are not allowed to select any edges above the rupture, i.e. we deduce `noselection/1`, which is done in line 13. Deducing again `selection/0` after having deduced the former, implies a gap between two nodes whose vertices selected for the tour are not connected to each other, such that the answer set candidate is discarded (line 18). The other situation in which the solution candidate is discarded, occurs when there exist ruptures on two different branches of the tree decomposition and below each of them at least on edge was selected. Naturally, `noselection/1` is deduced and propagated upwards on each branch and when they meet at the post-processing node, the solution is discarded (line 19). This is also the reason for storing the current node as an argument for `selection/1`, as otherwise we could not differentiate in the post-processing node between the two different instantiations coming from different children of the join node.

In Listing 4.17 we can see again that we obtain all items and auxiliary items deduced throughout the traversal of the tree decomposition, but those determining the tour correspond to the literals from in Listing 3.10.

Further, we propose an implementation for the TSP-R which works also on weakly normalized tree decompositions and is identical to the previous encoding, except for the modeline and the rule which updates the counter of a vertex in post-processing nodes, the latter being identical

to the one in Listing 4.15. It can be found in the Appendix, in Listing A.7. The reader can notice that we concentrated on weakly and semi-normalized decompositions, because normalized ones lead to higher memory consumption due to the inherent bigger size of the former, while for unnormalized ones the encodings would have become much more complicated to write and understand, defeating our purpose to keep them maintainable and flexible.

CHAPTER 5

Evaluations

In this chapter we present the results of the empirical tests performed to evaluate the efficiency of the encodings we proposed for the TSP-NR and TSP-R. First we present the test data, consisting from both generated and real world test instances. Next, we give the specifications of the hardware and software environment used for our computations. Finally, we disclose the results for each data set in terms of runtime, memory consumption and number of conflicts, comparing D-FLAT's results with clingo's, and also the results obtained with D-FLAT with each other, on different data sets, implementations and configurations.

5.1 Problem Instances

Now, we will present the data used for our experiments[1], divided into six different data sets, by the purpose they serve. The first five were generated by us, while the last one contains real world instances. The generated grid instances in each data set are of a certain graph type: *8-connected full grids*, *8-connected grid-like graphs* or *4-connected grid-like graphs*. An 8-connected grid is a graph with matrix structure in which each vertex is connected by an edge to other vertices that are positioned next to it horizontally, vertically or diagonally, while in a 4-connected grid the vertices are connected only to their horizontal and vertical neighbors. Figure 5.1 shows such an 8-connected full grid with treewidth 7. In a full grid, each vertex is connected to all its neighbors, while in a graph that we call grid-like, the connections to the neighboring vertices are determined based on a specified probability. For full grids we can control the (theoretical) treewidth by creating matrices with a certain width and height. The maximum bag size in an optimal tree decomposition of a graph is one more than the value of its treewidth. Each two vertices connected by an edge must be in the same bag at some point and once we removed a vertex we cannot introduce it back. Thus, we need the width of a grid to be two less than the number of bag elements, for the bag to be stretched out on two rows over all columns of

[1] `http://dbai.tuwien.ac.at/research/project/dflat/system/theses/moldovan/tsp.zip`

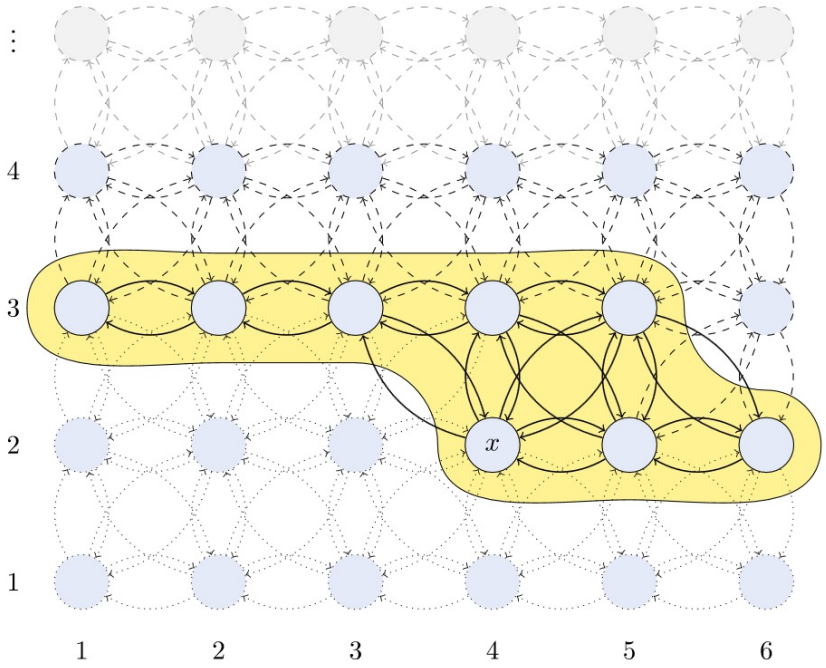


Figure 5.1: 8-Connected grid (6 x m), treewidth 7. [14]: Figure 5.1.

the grid and to contain a clique of four vertices that sequentially passes over all vertices row by row from left to right, without loss of generality. The height of the grid must be at least as large as the width, otherwise the bag could be laid out vertically. In Figure 5.1, the colored area represents the bag of the current node of an optimal tree decomposition at the moment right before vertex `x` can be removed and the clique covered by the bag under focus can move to the right by one column. The vertices below the colored area have already been removed and those above are waiting to be introduced. Nonetheless, for the instances that are not full grids, we cannot determine the (theoretical) treewidth exactly. These instances were created by giving a maximum wanted treewidth and a probability for selecting the grid's edges, and then checking the width of 10 tree decompositions created by D-FLAT. This means in their case we speak of a practical treewidth, or the width of D-FLAT's actual decompositions which usually also is roughly the same as the theoretical treewidth.

Each of the generated data sets contains various scenarios, where a scenario consists of 20 instances each, distinguished mainly by their treewidth and their number of vertices, and in the case of 8-connected grid-like graphs, also their satisfiability. For the latter we could determine the satisfiability by varying the probability to select an edge of the grid at their creation. Table 5.1 shows an overview of the generated grid instance scenarios, giving information on the type of TSP they are meant for, on their graph type, their treewidth, their number of vertices, the range of their weights, their satisfiability, and on the required minimum and maximum number of visits for 10 percent of the vertices (relevant only for TSP-R). `minVisits/2` and `maxVisits/2` are not specified at all in Data Sets 1 to 4, while in Data Set 5 they are specified only for 10

Data Set	TSP	Graph Type	TW	Vertices	Weights	SAT	min	max
Data Set 1	NR	8-con. full grid	2	20..50..5	1..20	✗	-	-
			2	50..300..50	1..20	✗	-	-
			2	1000	1..20	✗	-	-
			2	greatest	1..20	✗	-	-
			3	20..50..5	1..20	✓	-	-
			3	50..300..50	1..20	✓	-	-
			3	1000	1..20	✓	-	-
			3	greatest	1..20	✓	-	-
			4	20..50..5	1..20	✓	-	-
			4	50..300..50	1..20	✓	-	-
			4	1000	1..20	?	-	-
			4	greatest	1..20	✓	-	-
			5	20..50..5	1..20	?	-	-
			5	50..300..50	1..20	?	-	-
			5	1000	1..20	?	-	-
			5	greatest	1..20	?	-	-
Data Set 2	NR	8-con. full grid	3	100,300	1..5	✓	-	-
			4	100,300	1..5	✓	-	-
Data Set 3	NR	8-con. grid-like	3	100,300	1..20	✓	-	-
			3	100,300	1..20	✗	-	-
			4	100,300	1..20	✓	-	-
			4	100,300	1..20	✗	-	-
Data Set 4	NR	4-con. grid-like	4	100	1..20	17	-	-
Data Set 5	R	8-con. full grid	3	100,300	1..20	20,20	0	0
			3	100,300	1..20	11,7	2	2
			3	100,300	1..20	20,20	0	10
			4	100,300	1..20	20,20	0	0
			4	100,300	1..20	16,13	2	2
			4	100,300	1..20	20,20	0	10

Table 5.1: Generated grid instances.

percent of the vertices. Here we would like to mention that the instances for the TSP-NR can be solved also by the proposed TSP-R encodings, as the latter are generalizations of the former. Each row in Table 5.1 represents one scenario, if only one instance size is specified with regard to the number of vertices, or more scenarios, otherwise. In the latter case we either enumerate the instance sizes, delimited by a comma, or specify the smallest scenario, the largest one, and the step width. For example *50..300..50* means that the specific row represents six scenarios for instance sizes 50, 100, 150, 200, 250 and 300. Further, *greatest* stands for the greatest instance with the specific treewidth given in column `TW` that could be solved in less than 10 minutes for any of the 20 decompositions[2]. In the *SAT* column, a checkmark means that all instances

[2]The actual number of vertices will be disclosed in Section 5.3.

of that or those scenarios are satisfiable, an *x* that none is satisfiable, a question mark that we do not have knowledge of the satisfiability, and a sequence of numbers gives the number of satisfiable instances for each scenario the row represents. The real world instances are modeled after Vienna's public transportation system.

Main Data for TSP-NR: 8-Connected Full Grids

Data Set 1, which contains the most scenarios, is meant to compare D-FLAT's efficiency on our implementation for the TSP-NR with and without default join on semi-normalized tree decompositions to clingo's performance, with its different configurations. For this purpose we created full grids for which we can precisely control the treewidth. They range from treewidth 2 to 5 and for each treewidth from 20 to 50 vertices, with a pace of 5, and from 50 to 300 vertices, with a pace of 50. Further, we created one instance with 1000 vertices for each treewidth and instances with even more vertices to test which is the maximum number of vertices that can be processed in 10 minutes for each treewidth. In our case all instances that have a treewidth of 2 are unsatisfiable as the generated graphs are very sparse. All other instances are satisfiable.

8-Connected Full Grids with Lower Edge Weights for TSP-NR

Data Set 2 is meant for comparing D-FLAT's performance in dependence of the interval of the weights of the edges. For this purpose we took the same 8-connected full grids, but replaced the weights ranging from 1 to 20 with others ranging from 1 to 5. The selected scenarios deal with 100 and 300 vertices, with a treewidth of 3 and 4.

8-Connected Grid-Like Graphs for TSP-NR

With Data Set 3, we intend to provide insight into D-FLAT's performance on our TSP-NR implementation when operating on a different type of graph. Another reason is that we intend to check D-FLAT's performance on satisfiable instances as compared to unsatisfiable ones, which is possible because we can control the satisfiability for this type of graphs by means of the probability we set for the edge selection. For these purposes we created eight scenarios, one for each combination of treewidth (3 or 4), number of vertices (100 and 300), and positive or negative satisfiability.

4-Connected Grid-Like Graphs for Weakly Decomposed Trees for TSP-NR

In order to check how D-FLAT behaves on weakly normalized tree decompositions as opposed to semi-normalized ones we created Data Set 4, consisting of 20 instances of 4-connected grid-like graphs with 100 vertices and treewidth 4. We chose this type of graph as D-FLAT actually always decomposes 8-connected graphs into semi-normalized decompositions even when specifying in the modeline or the D-FLAT call that a weakly normalized decomposition is wanted. Here, all but three instances are satisfiable.

Instance	Satisfiable	Vertices to Be Visited Once	`maxVisits` Other Vertices
Metro1	✓	85	10
Metro2	✓	4	10
Metro3	✓	4	1
Metro4	✓	10	1

Table 5.2: Real world instances based on the Viennese metro and urban train system (138 vertices, treewidth 5).

Data for TSP-R: 8-Connected Full Grids

Data Set 5 is meant for testing D-FLAT's performance also on the TSP-R, when operating on instances in which some of the vertices must be visited more or less often than the standard case checked on the first data set, respectively, and when leaving full freedom to the algorithm whether to visit some of the vertices or not. More specifically, we adapted the 8-connected full grids with 100 and 300 vertices, treewidths 3 and 4, from the main data set, such that about 10 percent of the vertices must be visited exactly twice. Further, we adapted the instances from the former scenarios such that about 10 percent of the vertices must not be visited at all, and once more such that 10 percent of the vertices may be left out of the tour, but can also be visited up to 10 times if this leads to a lower tour cost. All other vertices must be visited exactly once. Thus, we obtained a total of 12 adapted scenarios. While the instances in which 10 percent of the vertices must not be in the tour and for those for which 10 percent of the vertices can, but do not have to be visited, remain satisfiable, about half of those with vertices that must be visited twice became unsatisfiable.

Real World Data: The Viennese Rail Transportation System

In order to present also a real world case study we ran some tests for the TSP-R on Vienna's tramway system as well as on its metro and urban train system. The purpose was to introspect D-FLAT's behavior both on semi-normalized and on weakly normalized tree decompositions, as opposed to clingo's (with different configurations). Furthermore, we wanted to observe how the performance changed depending on the number of stations and on the number of times the latter had to be visited. The TSP-NR is unsatisfiable on both the tram and the metro-suburban train systems due to the dead ends at the end of some of the lines and to the fact that the TSP-NR does not allow an edge to be used twice. However, we simulated the TSP-NR on the metro system by specifying to visit only (almost all) metro stations and the urban train stations when needed to create a tour.

The instances based on Vienna's metro and urban train system, depicted in Figure 5.2[3], have a practical treewidth of 5, contain 138 vertices and are all satisfiable. Their weights correspond to the time in minutes needed to get from one station to another. The first one is meant for simulating the TSP-NR by putting also the urban train lines and stations at the disposal of the TSP-R encoding. More specifically, the metro stations marked by a filled oval in Figure 5.2, i.e.

[3] `http://www.wienerlinien.at`

Figure 5.2: Vienna's metro and urban train system. ©Wiener Linien

all metro stations except for two at one end of the red line, *U1*, and six at the end of the brown line, *U6* (85 in total), must be visited once, while all other stations can be visited at most 10 times or not at all. The second instance specifies that only four stations, which are dispersed throughout the network, must be visited exactly once, while all others can be visited at most 10 times or not at all. The third instance is almost identical, the difference being that those instances

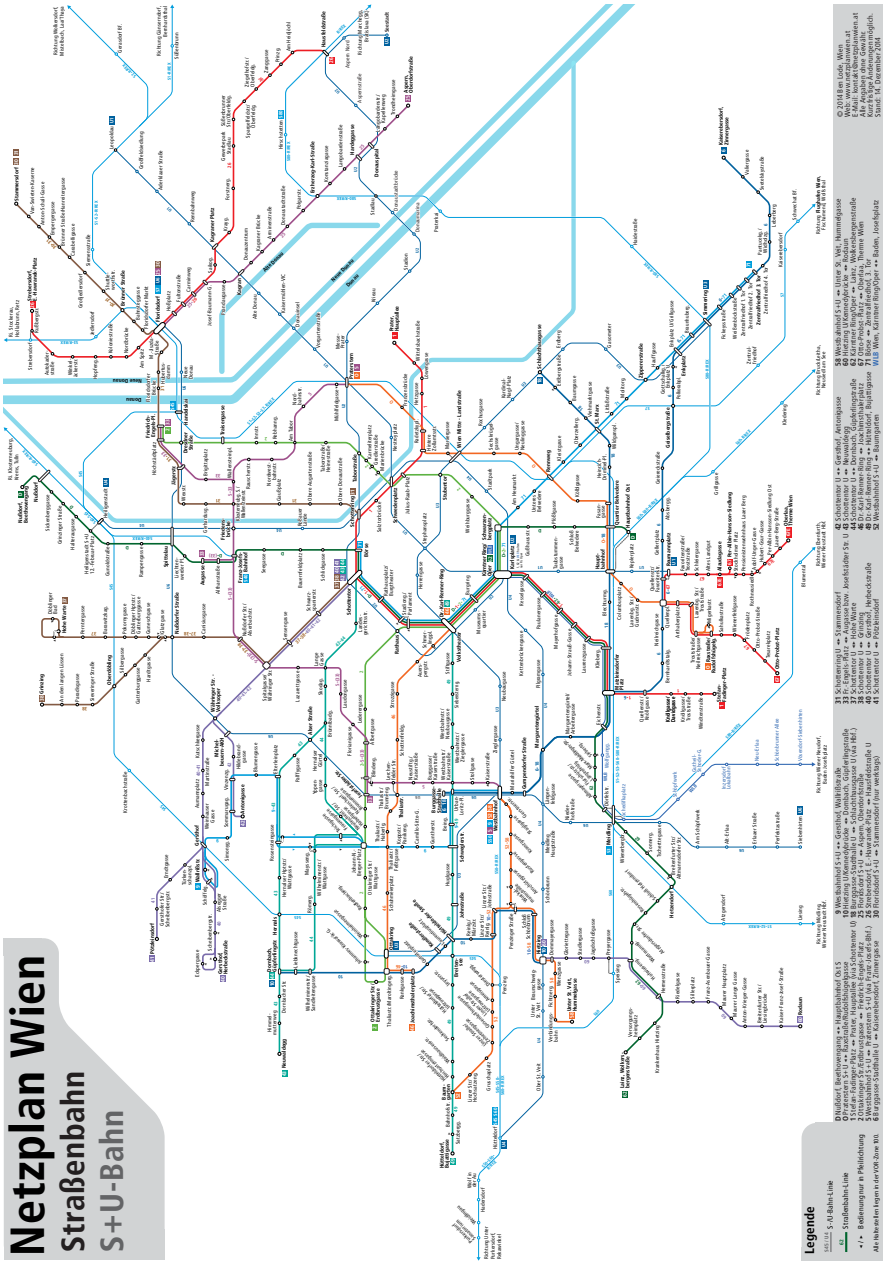


Figure 5.3: Vienna's tramway, metro and urban train system. ©Ben Lode, Vienna

which do not have to be visited, can be visited at most once. For the fourth and last instance it is necessary to visit 10 stations exactly once while the others can be visited at most once. Table 5.2 provides an overview over the variables that change from instance to instance.

The tramway system can be seen in Figure 5.3[4], which contains all tram lines. The metro

[4] http://www.netzplanwien.at

Instance	Satisfiable	Vertices to Be Visited Once	`maxVisits` Other Vertices
Tram1	✗	4	10
Tram2	✓	4	10
Tram3	✓	4	1
Tram4	✓	15	1
Tram5	✓	24	1
Tram6	✓	24	10

Table 5.3: Real world instances based on the Viennese tramway system (402 vertices, treewidth 6).

and urban train lines also appear in this figure and are marked with fine blue lines and the inscriptions *U1* to *U6* and *S1* to *S80*, as opposed to the thicker tramway lines. However, the metro and urban train lines are not part of our tramway instances. All instances contain Vienna's 402 tram stations, have a treewidth of 6 and were generated based on Vienna's *Open Gov Data*[5]. The first two instances stipulate that four specific stations, dispersed throughout the network, must be visited exactly once while all the others can be visited at most 10 times or not at all. While the first one is unsatisfiable, the second one is satisfiable. The third one is similar to the second one, the difference being that the stations which do not have to be visited can be visited at most once. The fourth and fifth instance are specified similarly to the third one but now they contain 15 and 24 vertices that must be visited, respectively. The sixth and last instance also contains 24 vertices that must be visited while the others can be visited at most 10 times. Table 5.3 again provides an overview over the variables that change from instance to instance.

5.2 Experimental Setting

Table 5.4 shows the hardware and software specifications of the system where the empirical tests were performed. All of them were carried out on a single core of an AMD Opteron 6308@3.5GHz processor running Debian GNU/Linux 7 (kernel 3.2.0-4-amd64). For all tests we allowed 32 GB of memory and 10 minutes to solve the problem, concerning generated grid instances, and 60 minutes concerning real world instances. As D-FLAT's performance can vary

[5] `https://open.wien.at/site/datensatz/?id=add66f20-d033-4eee-b9a0-40019828e698`

[6] Also contained in `http://dbai.tuwien.ac.at/research/project/dflat/system/theses/moldovan/tsp.zip`

Processor:	AMD Opteron 6308@3.5GHz
Main Memory:	192 GB
Operating System:	Debian GNU/Linux 7 (kernel 3.2.0-4-amd64)
clingo version:	4.4.0
D-FLAT version:	v1.0.1-2-ge59ccca [6]

Table 5.4: Hardware and software system specifications.

clingo Configuration 1:	`clingo PROGRAMPATH INSTANCEPATH --opt-mode=optN --stats=1 --quiet --configuration=auto`
clingo Configuration 2:	`clingo PROGRAMPATH INSTANCEPATH --opt-mode=optN --stats=1 --quiet --configuration=trendy`
clingo Configuration 3:	`clingo PROGRAMPATH INSTANCEPATH --opt-mode=optN --stats=1 --quiet --configuration=handy`
D-FLAT Configuration 1:	`dflat -e edge -e vertex -p PROGRAMPATH --stats --depth 0 --output quiet --tables -n semi --seed SEED < INSTANCEPATH`
D-FLAT Configuration 2:	`dflat -e edge -e vertex -p PROGRAMPATH --stats --depth 0 --output quiet --tables -n semi --seed SEED --default-join --post-join < INSTANCEPATH`
D-FLAT Configuration 3:	`dflat -e edge -e vertex -p PROGRAMPATH --stats --depth 0 --output quiet --tables -n weak --seed SEED --default-join --post-join < INSTANCEPATH`

Table 5.5: Program calls used for benchmarking.

with different tree decompositions of the same instance, we performed each test 20 times. We used different types of tree decompositions with D-FLAT, namely semi-normalized and weakly normalized. For 8-connected instances, it turned out they are always decomposed by D-FLAT into semi-normalized tree decompositions, while the benchmarking on 4-connected graphs and on the real world instances was performed both on weakly and semi-normalized decompositions. At the same time, clingo also provides various possible configurations. *trendy* and *handy* imply the use of heuristics meant for industrial and large problems, respectively, and proved to be the most suited for our purposes. For all scenarios we used the *auto* configuration and one of the other two which proved to be more effective for that specific type of scenario in preliminary experiments.

Table 5.5 shows the exact calls used when performing the tests, where PROGRAMPATH and INSTANCEPATH have to be replaced by the actual path to the encoding or the instance file, respectively and SEED by the wanted seed. Instead of using the modelines displayed as part of the code in Chapter 4 we used the full calls for D-FLAT as they are presented here. The options `--quiet` and `--depth 0 --output quiet` were used for clingo and D-FLAT, respectively, to get the actual computation times without the time needed to output the results, which can be a significant part of the total runtime. The `--stats` option was activated for both systems in order to record also the number of conflicts necessary to get to the solution. For clingo, we specified the `--opt-mode=optN` option so it computed all optimal models. The three configurations for clingo are different only with respect to their configuration type. Those

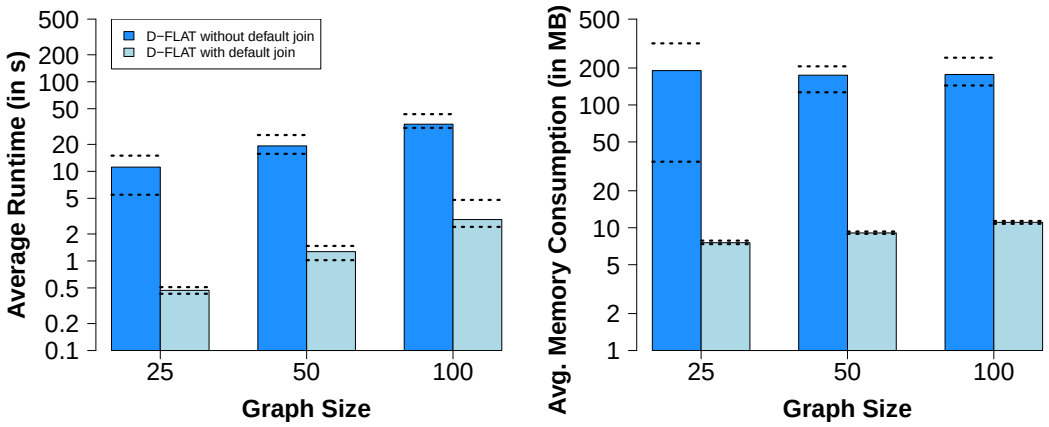


Figure 5.4: TSP-NR encoding for D-FLAT without default join, Listing A.3, versus TSP-NR encoding for D-FLAT with default join, Listing A.4, on grids of treewidth 3 (from Data Set 1).

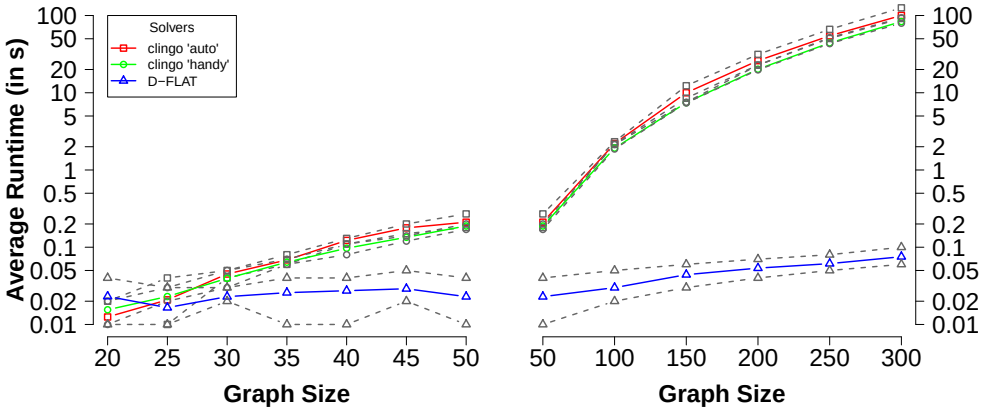
configurations which were used for testing the scenarios with D-FLAT differ with respect to the use of default join and the post-processing node, and the creation of weakly and semi-normalized tree decompositions, respectively. The used seeds were randomly generated numbers between 1 and 1000000. Further `-p` is used to supply the program, `-e` to state the edges and vertices, and `--tables` to indicate that we are using D-FLAT's simplified interface.

5.3 Results of the Empirical Tests

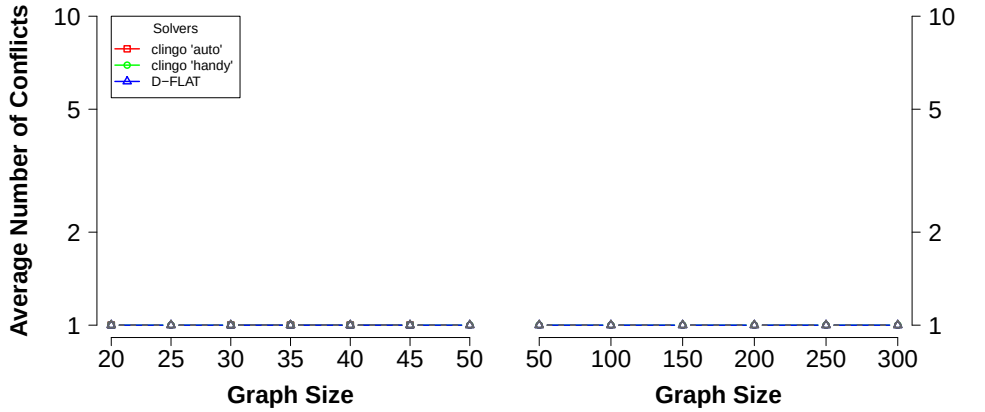
TSP-NR for Semi-Normalized Tree Decompositions on 8-Connected Full Grids

Preliminary tests have shown that the proposed TSP-NR encoding for D-FLAT with default join, Listing A.4, is much more efficient than the TSP-NR encoding for D-FLAT without default join, Listing A.3. To exemplify this we selected three scenarios with treewidth 3 from Data Set 1. Figure 5.4 shows the minimum, average and maximum runtime and memory consumption for the TSP-NR encoding for D-FLAT without default join, Listing A.3, compared to the TSP-NR encoding for D-FLAT with default join, Listing A.4, for these three scenarios from Data Set 1. D-FLAT Configurations 1 and 2 from Table 5.5 were used for the calls. In matters of runtime and memory consumption the use of default join brings 10 to 25 times improvements due to the fact that the grounding does not explode at join nodes when using the default join. For this reason we will further concentrate on encodings designed for the former's use. Nonetheless, our TSP-NR encoding for D-FLAT without default join, Listing A.3, is already faster than clingo for the scenarios with 50 and 100 vertices, treewidth 3. Still, for treewidth 4 we already obtained timeouts even for the smallest instances.

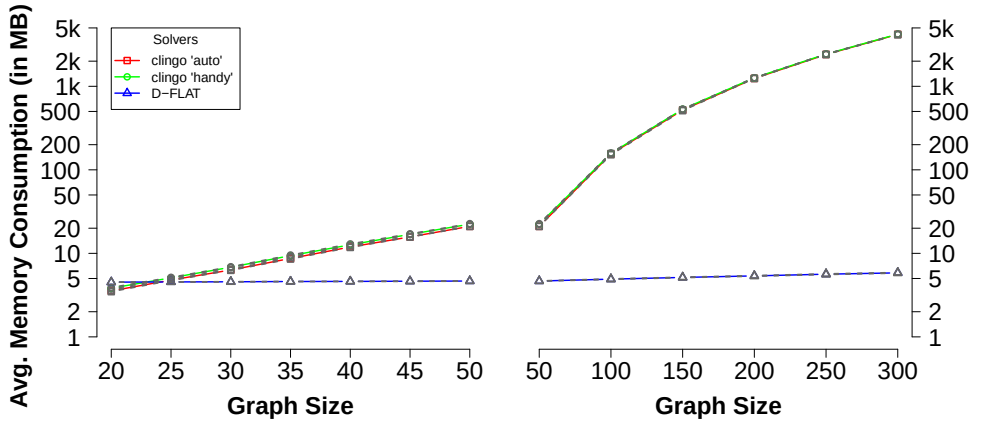
Figure 5.5 shows the performance of the TSP-NR encoding for D-FLAT with default join on semi-normalized tree decompositions, Listing A.4, compared to the ASP TSP-NR encoding, Listing A.1, with clingo, on the scenarios with treewidth 2 from Data Set 1. The mapped values are those of the minimum, average and maximum runtime, number of conflicts and memory



(a) Minimum, average and maximum runtime.

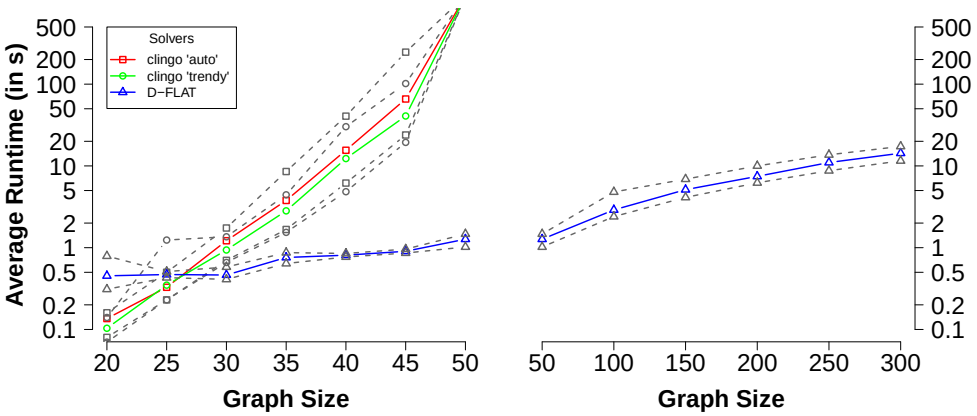


(b) Minimum, average and maximum number of conflicts.

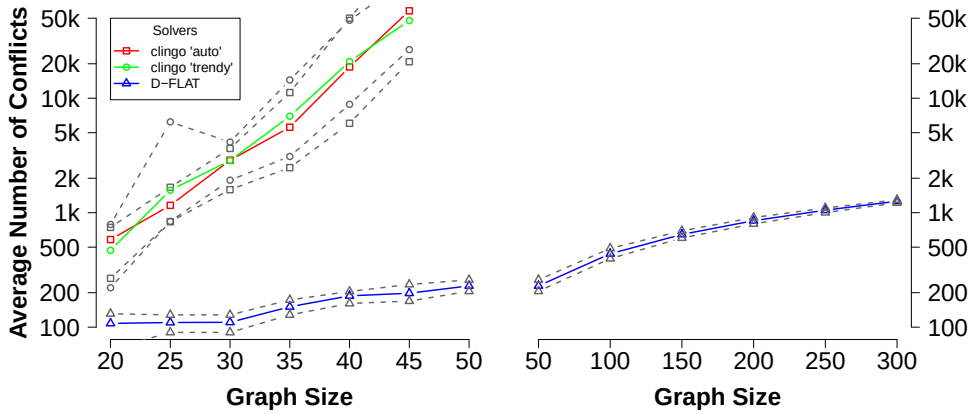


(c) Minimum, average and maximum memory consumption.

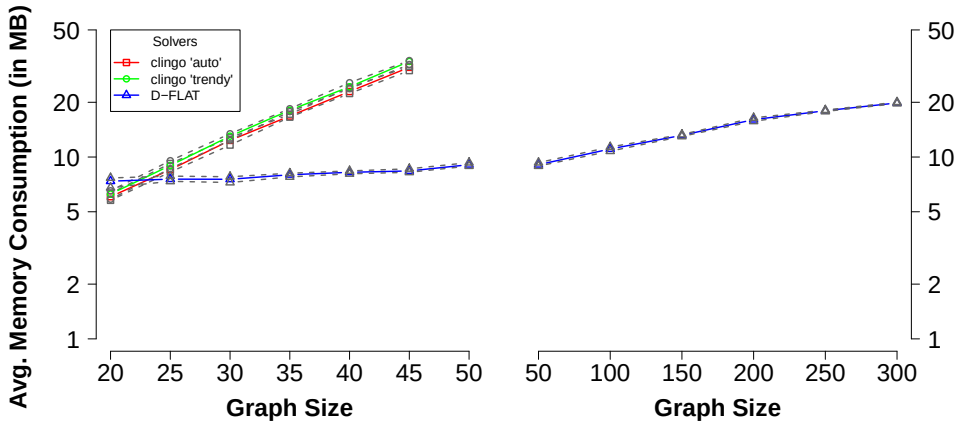
Figure 5.5: TSP-NR encoding for D-FLAT, Listing A.4, versus ASP TSP-NR encoding, Listing A.1, with clingo, on the instances with treewidth 2 from Data Set 1.



(a) Minimum, average and maximum runtime for each scenario in Data Set 1.

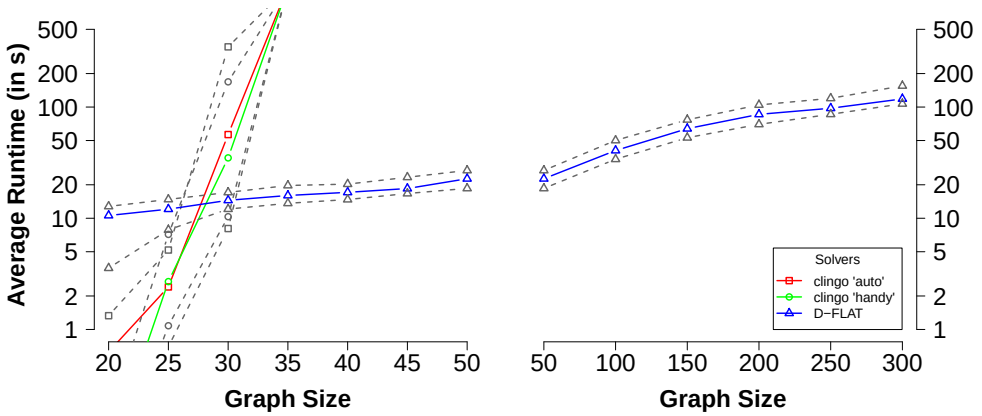


(b) Minimum, average and maximum number of conflicts for each scenario in Data Set 1.

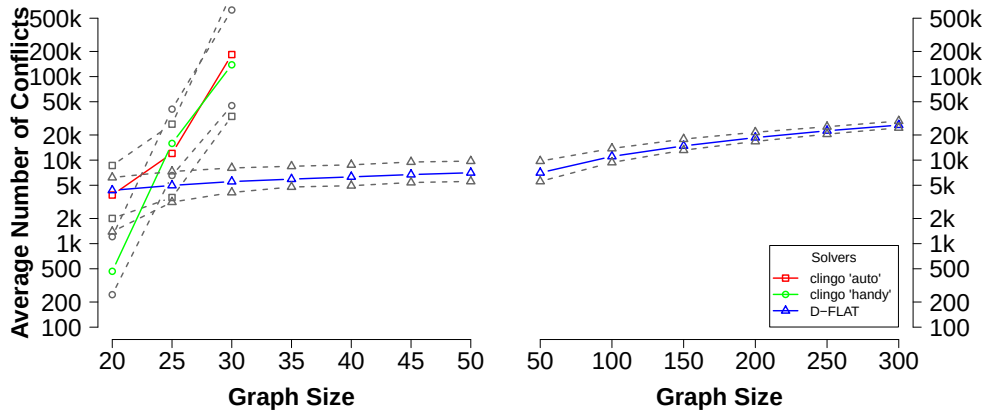


(c) Minimum, average and maximum memory consumption for each scenario in Data Set 1.

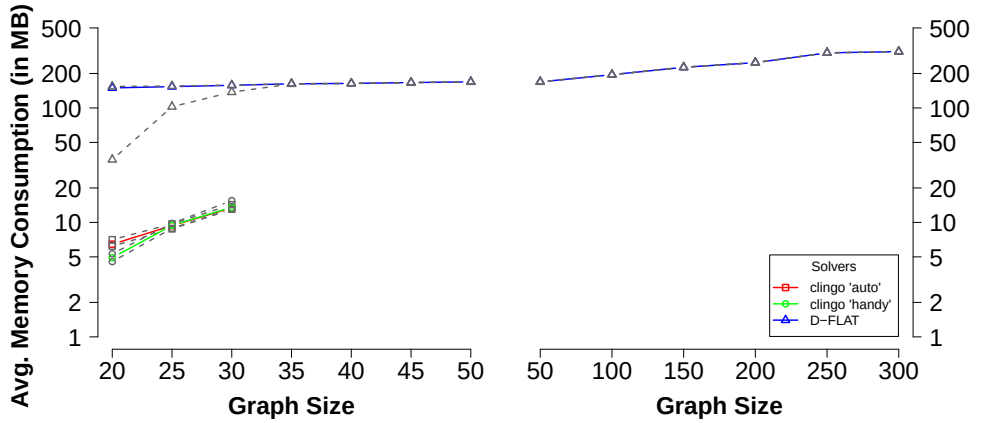
Figure 5.6: TSP-NR encoding for D-FLAT, Listing A.4, versus ASP TSP-NR encoding, Listing A.1, with clingo, on the instances with treewidth 3 from Data Set 1.



(a) Minimum, average and maximum runtime for each scenario in Data Set 1.

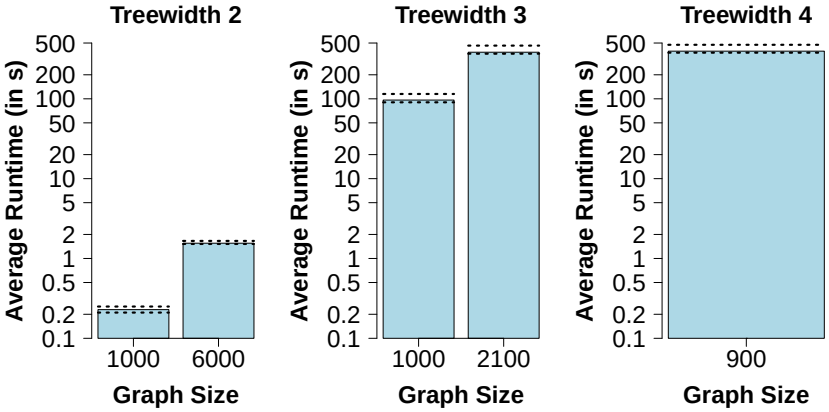


(b) Minimum, average and maximum number of conflicts for each scenario in Data Set 1.

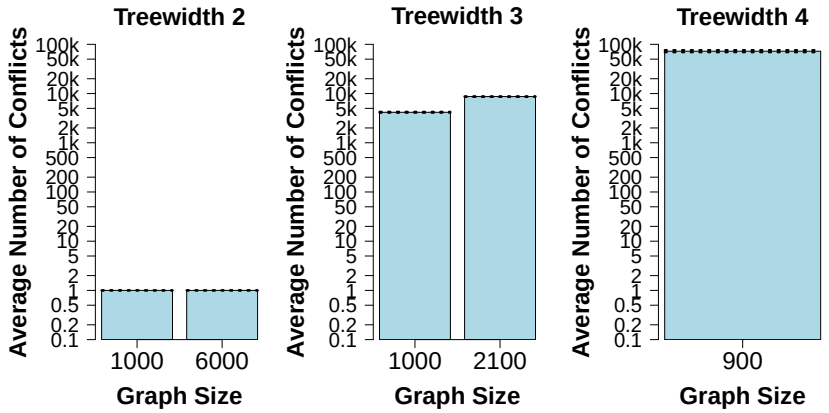


(c) Minimum, average and maximum memory consumption for each scenario in Data Set 1.

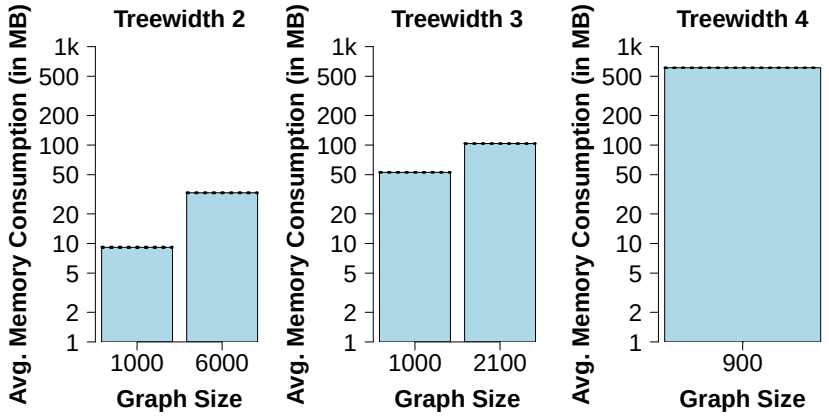
Figure 5.7: TSP-NR encoding for D-FLAT, Listing A.4, versus ASP TSP-NR encoding, Listing A.1, with clingo, on the instances with treewidth 4 from Data Set 1.



(a) Minimum, average and maximum runtime.



(b) Minimum, average and maximum number of conflicts.



(c) Minimum, average and maximum memory consumption.

Figure 5.8: TSP-NR encoding for D-FLAT with default join on semi-normalized tree decompositions, Listing A.4, on 1000-vertex instances, and largest, in 10 minutes solvable 8-connected grids (all in Data Set 1).

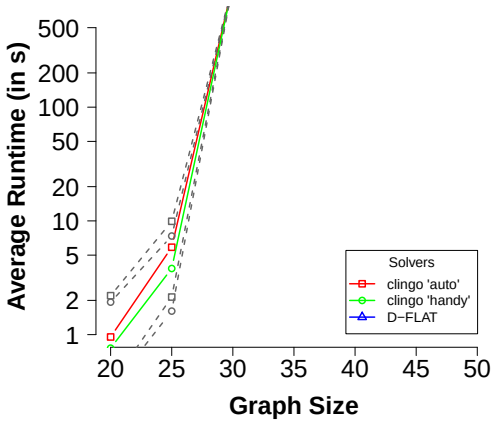


Figure 5.9: Runtime for TSP-NR encoding, Listing A.1, with clingo on and the instances with treewidth 5 from Data Set 1.

consumption. The same is shown in Figure 5.6 and Figure 5.7 on the scenarios with treewidth 3 and 4, respectively. For calling D-FLAT we used Configuration 2 from Table 5.5, while for clingo we used Configuration 1 for all instances, Configuration 2 for those having treewidth 3 and Configuration 3 for those having treewidth 2, 4 and 5. Looking first at clingo's performance, we can see that for treewidth 2 the instances in all scenarios are solvable in less than 10 minutes, for treewidth 3 and 4, it fails already at 45 and 30 vertices, respectively. At the same time D-FLAT manages all scenarios up to 300 vertices (and more). In Subfigures 5.6a and 5.7a we can see that for very small instances clingo actually performs better, up to the point where the expense of creating tree decompositions and running the ASP solver several times pays off. Further, in all three figures we can observe the resemblance between the graphics depicting the running time with those for the number of conflicts, and the memory consumption, except for Subfigure 5.5b where the instances are unsatisfiable and both clingo and D-FLAT get to the result with only one conflict. Further, the memory consumption proves to be more robust than the runtime for both clingo and D-FLAT.

Figure 5.8 discloses for treewidths 2, 3 and 4, which are the scenarios with the largest instances in terms of number of vertices, for which every instance can be solved in less than 10 minutes, for any of the 20 tree decompositions. Further, it shows the performance of the TSP-NR encoding for D-FLAT with default join on semi-normalized tree decompositions, in terms of minimum, average and maximum runtime, number of conflicts and memory consumption, for the latter scenarios and for those containing instances with 1000 vertices. While, for treewidth 2 it could solve 6000-vertex instances in less than 10 minutes, for treewidth 4 it reached the 900-vertex mark. In fact, for treewidth 2 we did not witness a timeout up to 110000-vertex instances. However, starting from 6100 vertices our code caused segmentation faults. When considering a treewidth of 2, clingo can solve all instances of the scenarios with at most 450 vertices, in less than 10 minutes.

When dealing with the grids of treewidth 5 from Data Set 1, our TSP-NR encoding for D-FLAT with default join on semi-normalized tree decompositions, Listing A.4, could not even

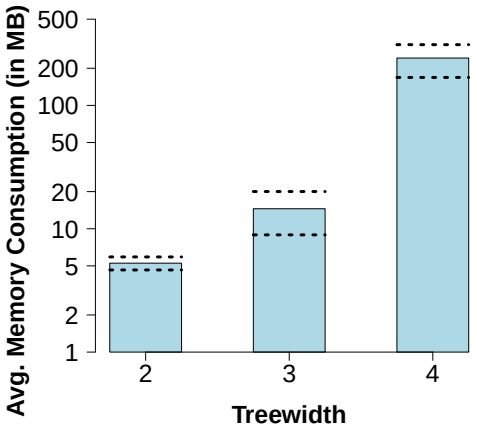


Figure 5.10: Memory consumption of TSP-NR encoding for D-FLAT, Listing A.4 on the instances with 100 to 300 vertices from Data Set 1.

solve the smallest instance, of 20 vertices, in less than 10 minutes, while clingo also failed in terms of runtime for both configurations from 30 vertices upwards, as shown in Figure 5.9.

Figure 5.10 shows the average memory consumption, with lower and upper bounds, for the TSP-NR encoding for D-FLAT with default join on semi-normalized tree decompositions, Listing A.4, over all instances with 100 to 300 vertices, and discriminates between the different treewidths. We can see that the memory consumption rises drastically with the treewidth, and with the former also the runtime, which however does not have such clear bounds over instances of the same treewidth.

TSP-NR for Semi-Normalized Tree Decompositions on 8-Connected Full Grids with Lower Edge Weights

Figure 5.11 shows the efficiency of the TSP-NR encoding for D-FLAT with default join on semi-normalized trees, Listing A.4, on instances with weights from 1 to 5 (Data Set 2), compared to the one on instances with weights from 1 to 20 (Data Set 1), in terms of minimum, average and maximum runtime and memory consumption. Here we used again Configuration 2 for calling D-FLAT (Table 5.5). Working on instances with lower weights improved the runtime by 20 to 30 percent, but not the memory consumption. This could be attributed to the fact that the aggregate functions used in our encodings are instantiated into as many grounded rules, as the number of results the aggregate can possibly deliver. Thus, when working on a wider range of weight values, the rule, in which the cost that contains an aggregate summing up weights is calculated (line 40 in Listing A.4), is instantiated more times in the grounder than when working on a more narrow range, as there are more possible results of the `#sum` function. The growth of the grounding seems not to be extensive enough to influence overall memory consumption, yet sufficient to visibly affect the runtime, which proved to be less robust than the former.

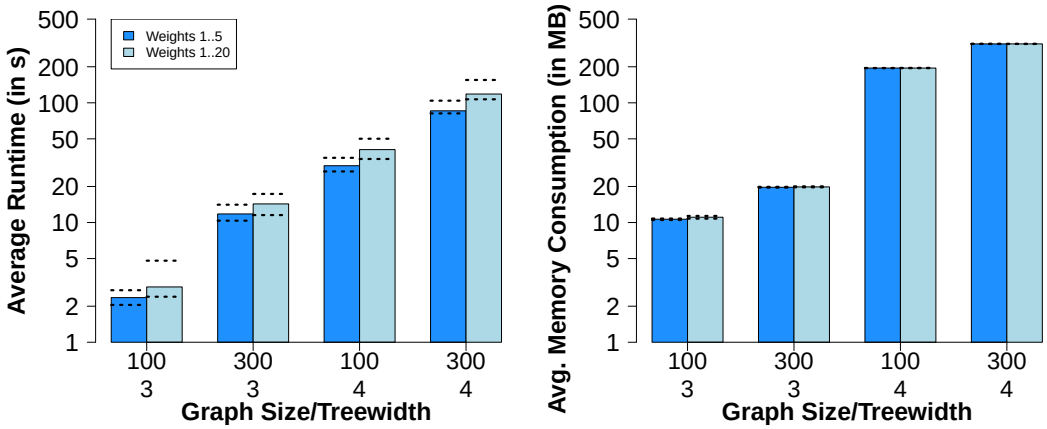


Figure 5.11: Performance of TSP-NR encoding for D-FLAT, Listing A.4, on instances with weights from 1 to 5 (Data Set 2) versus 1 to 20 (Data Set 1).

TSP-NR for Semi-Normalized Tree Decompositions on 8-Connected Grid-Like Graphs

Figure 5.12 shows the performance of the TSP-NR encoding for clingo, Listing A.1, on satisfiable and unsatisfiable 8-connected grid-like graphs (Data Set 3), and of the TSP-NR encoding for D-FLAT with default join on semi-normalized tree decompositions, Listing A.4, on the satisfiable and unsatisfiable 8-connected grid-like graphs (Data Set 3), and on (satisfiable) full grids (Data Set 1), respectively. The minimum, average and maximum runtime, number of conflicts and memory consumption are depicted. For calling D-FLAT we used Configuration 2 from Table 5.5 and for clingo we used Configurations 1 and 2. Here, the treewidth indicated in the graphics is a practical treewidth, namely the width of the tree decompositions created by D-FLAT. First we would like to mention that we did not include the performance of the TSP-NR encoding for clingo on satisfiable 8-connected grid-like graphs from Data Set 3, because for all four scenarios we witnessed timeouts, as we also did for the (satisfiable) instances with the same treewidths and sizes from Data Set 1. Further, we experienced isolated timeouts with clingo on instances with 300 vertices but also with D-FLAT on instances with 300 vertices and treewidth 4. For these we counted with the timeout value of 10 minutes, and with the maximum number of conflicts and the maximum memory consumption among the other instances from the same scenario, respectively.

Looking only at the unsatisfiable instances from Data Set 3, we can see that in average our TSP-NR encoding for D-FLAT performs better than the ASP encoding (except for the scenario with 100 vertices at treewidth 4, where it does only slightly worse). Yet, specifically for instances with 100 vertices and treewidth 4, the performance can be much worse, but also much better, depending on the tree decomposition. Overall, we can see a very wide range for the runtime on unsatisfiable instances, especially for D-FLAT, but also for the memory consumption and the number of conflicts, the latter going down even to the value 1 for all scenarios. The wide range can be traced back to the fact that in the case of a suitable tree decomposition all answer set candidates are eliminated in the nodes which are closer to the bottom while in the case of an

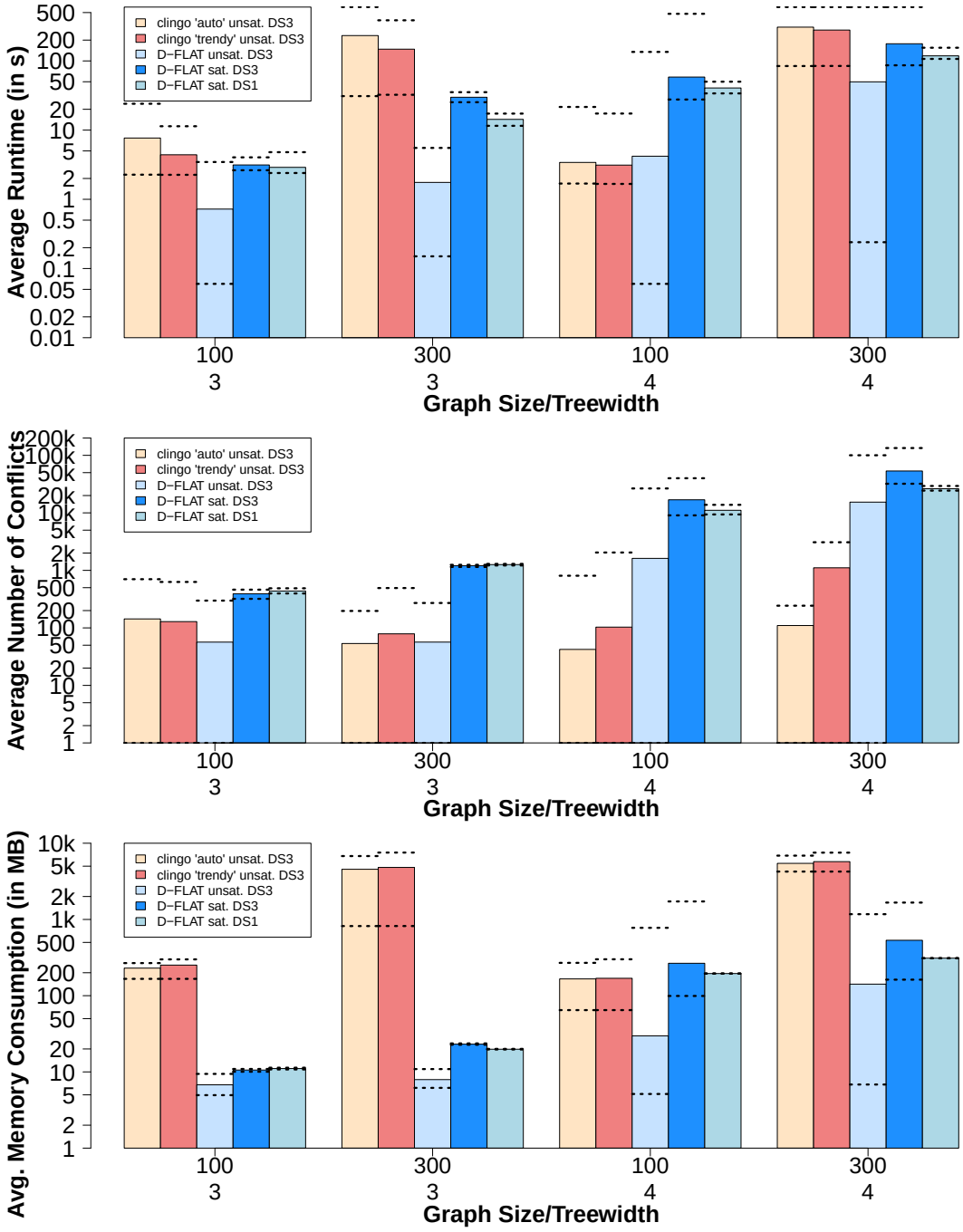
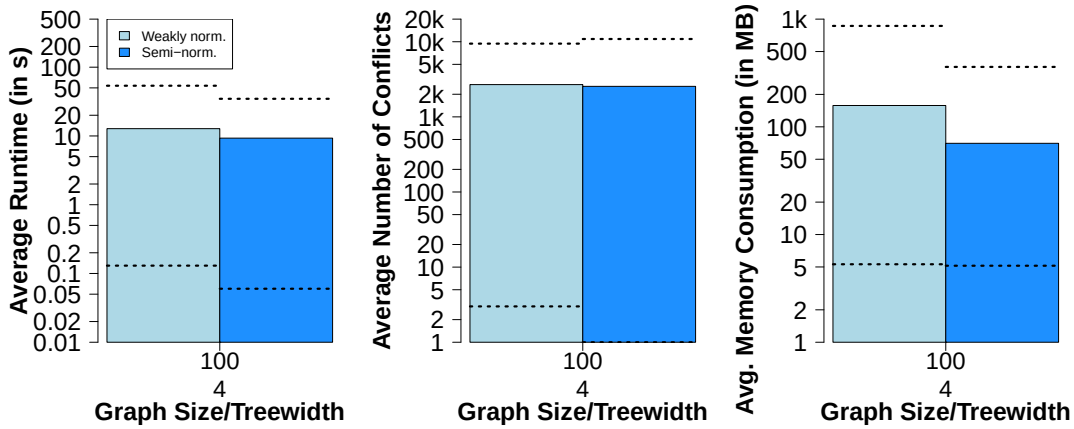
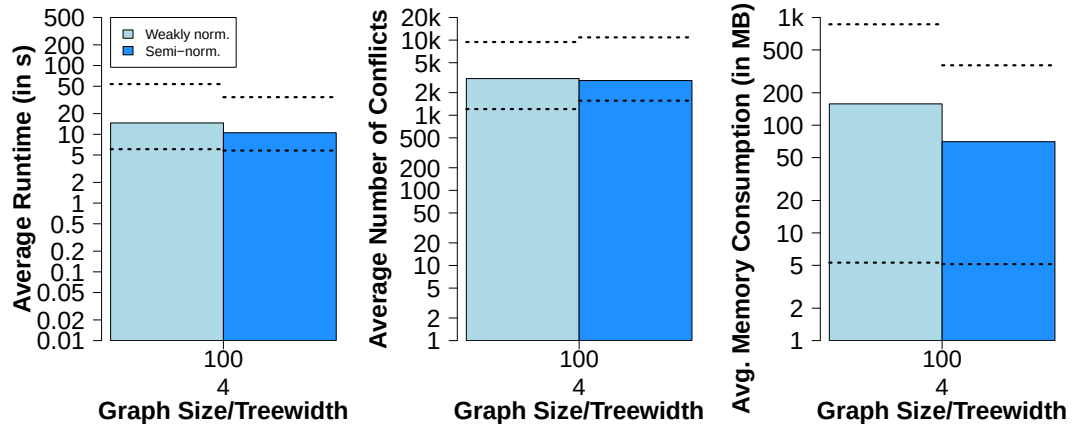


Figure 5.12: TSP-NR encoding for clingo, Listing A.1, with ’auto’ and ’trendy’ configurations, on satisfiable versus unsatisfiable grid-like graphs (Data Set 3), and of TSP-NR encoding for D-FLAT, Listing A.4, on unsatisfiable grid-like graphs (Data Set 3), compared to TSP-NR for D-FLAT, Listing A.4, on satisfiable full grids (Data Set 1).



(a) All instances from Data Set 4.



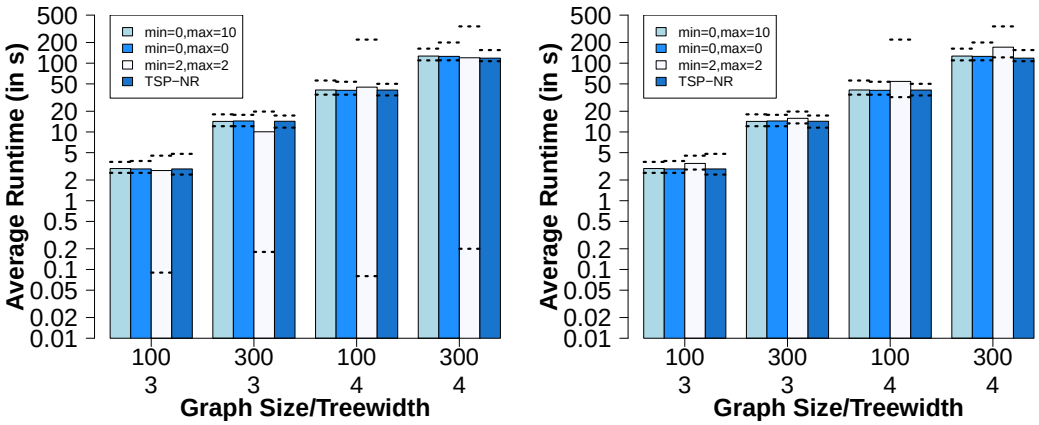
(b) Only satisfiable instances from Data Set 4.

Figure 5.13: TSP-NR encoding for D-FLAT on weakly normalized tree decompositions, Listing A.5, versus encoding for D-FLAT on semi-normalized tree decompositions, Listing A.4, on Data Set 4.

unsuitable one many answer set candidates are kept up to the root's child node.

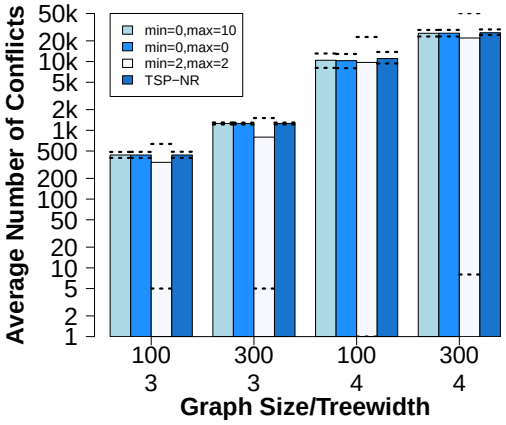
When considering only the satisfiable instances, D-FLAT performs much better than clingo also on 8-connected grid-like graphs. However, D-FLAT is about 30 percent better in terms of runtime, and 20 percent better in terms of memory consumption on the (satisfiable) full grids from Data Set 1 than on the satisfiable grid-like graphs from Data Set 3.

Further, taking into account both satisfiable and unsatisfiable instances, we see that unsatisfiable instances require less memory and runtime to be solved. Moreover, the number of conflicts is drastically lower when feeding clingo unsatisfiable instances, even when the runtime is considerably higher.

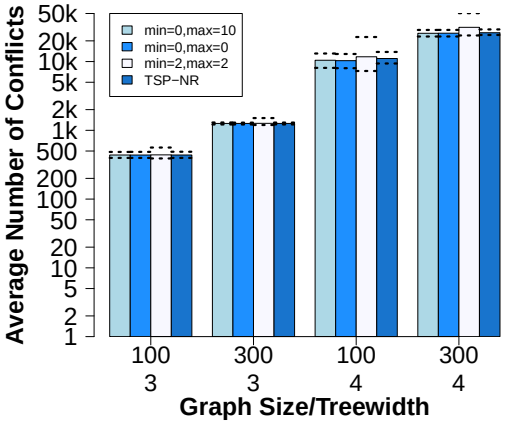


(a) Runtime over all instances.

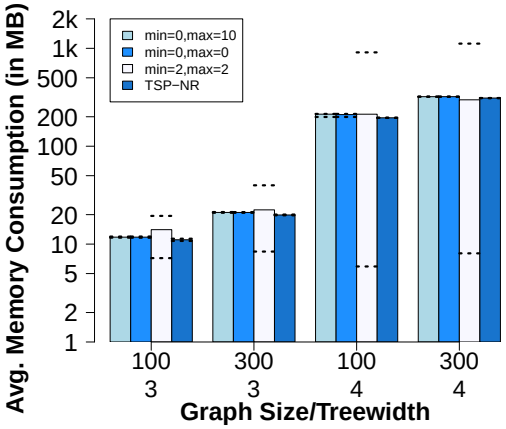
(b) Runtime over satisfiable instances.



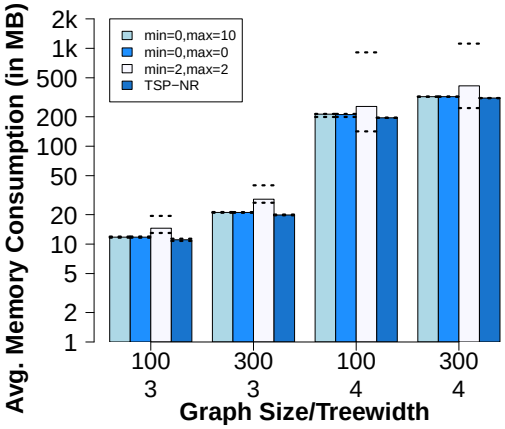
(c) Number of conflicts over all instances.



(d) Number of conflicts over sat. instances.



(e) Memory consumption over all instances.



(f) Memory consumption over sat. instances.

Figure 5.14: Performance of TSP-R encoding for D-FLAT, Listing A.6, on instances from Data Set 5, versus the TSP-NR encoding, Listing A.4, on instances from Data Set 1.

TSP-NR for Weakly Normalized Tree Decompositions on 4-Connected Grid-Like Graphs

Figure 5.13 shows the performance of the TSP-NR encoding for D-FLAT with default join on weakly normalized tree decompositions, Listing A.5, using D-FLAT Configuration 3 from Table 5.5, compared to the TSP-NR encoding for D-FLAT with default join on semi-normalized tree decompositions, Listing A.4, using D-FLAT Configuration 2 from Table 5.5, on 4-connected grid-like graphs (Data Set 4), presenting the minimum, average and maximum runtime, number of conflicts and memory consumption once for all instances from Data Set 4 and once for only the 17 satisfiable instances. First, we remark that the range between maximum and minimum values is much higher in Subfigure 5.13a than in Subfigure 5.13b due to the presence of the three unsatisfiable instances in the former. Further, we can observe that D-FLAT performed worse on weakly normalized tree decompositions than on semi-normalized ones: By 38 percent concerning the runtime and by 124 percent as for the memory consumption, when considering all instances. The fact that D-FLAT performed better on the semi-normalized tree decompositions holds for all satisfiable instances in the data set. We also checked whether the difference can be caused by the encoding itself, and not by the different tree decompositions, by calling the encoding for weakly normalized ones with D-FLAT Configuration 2 from Table 5.5, although it is only one line that makes the difference between the encodings for weakly normalized and semi-normalized tree decompositions. We obtained almost the same performance as we did when calling the TSP-NR encoding for semi-normalized tree decompositions with Configuration 2.

TSP-R for Semi-Normalized Tree Decompositions on 8-Connected Full Grids

In Figure 5.14 we see the minimum, average and maximum runtime, number of conflicts and memory consumption of the TSP-R encoding for D-FLAT with default join on semi-normalized tree decompositions, Listing A.6, fed with instances from Data Set 5, compared to the TSP-NR encoding for D-FLAT with default join on semi-normalized tree decompositions, Listing A.4, fed with instances of the same size and treewidth from Data Set 1 (D-FLAT Configuration 2 in Table 5.5). On the left side we show the measurements for all instances, while on the right side only for satisfiable ones. We remark again the higher range between maximum and minimum values when considering all instances. When creating Data Set 5 by adding instantiations of `minVisits/2` and `maxVisits/2` for 10 percent of the vertices from Data Set 1, some instances became unsatisfiable, but only for those scenarios in which the latter vertices have to be visited exactly twice, as can be seen in Table 5.1. Specifically for this group of scenarios, the average runtime increased by 27 percent and the average memory consumption by 35 percent when solved with the encoding for the TSP-R, compared to the instances from Data Set 1 solved by the encoding for the TSP-NR, but only when ruling out the unsatisfiable instances. When considering all of them, the fact that unsatisfiable instances are easier to solve outbalances the increased runtime and memory consumption. As for the instances for which 10 percent of the vertices must not be visited, and the ones for which the latter only may be visited, the runtime increases only by 1 and 2 percent, respectively, and the memory consumption by 6 percent, as compared to the TSP-NR encoding on instances from Data Set 1.

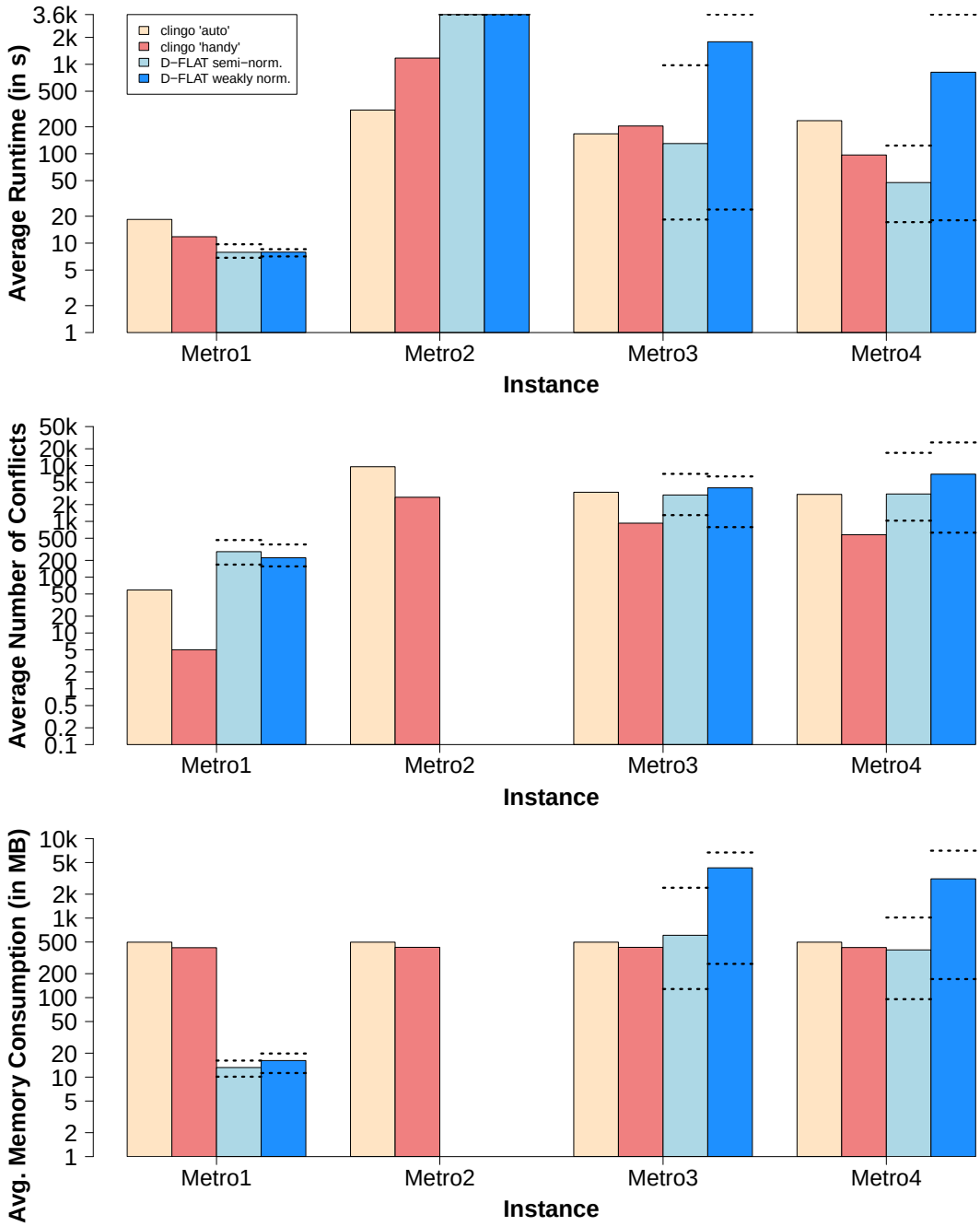


Figure 5.15: Performance of TSP-R encodings for D-FLAT, Listings A.6 and A.7, versus ASP TSP-R encoding, Listing A.2, on the instances based on the Viennese metro and urban train system.

Case Study: TSP-R on Real World Instances

Figure 5.15 displays the performance of the TSP-R encodings for D-FLAT with default join on semi-normalized and weakly normalized tree decompositions, from Listings A.6 and A.7, respectively, compared to the ASP TSP-R encoding, from Listing A.2, on the instances based on the Viennese tramway system which are presented in Table 5.2. We used Configurations 1 and 3 for clingo, as well as 2 and 3 for D-FLAT from Table 5.5, respectively. The first graphic shows the runtime for each instance and configuration, while the two other graphics show the number of conflicts and the memory consumption only for those instances and configurations for which the instance was solved in less than 60 minutes. For each instance we present the average values, whereas the minimum and maximum values are displayed only for the two D-FLAT configurations, as they were called 20 times each for every instance, with different seeds.

For the instance Metro1, which specified that almost all metro stations (but not the urban train stations, except for those in common) must be visited, both D-FLAT configurations performed better than any of the clingo configurations. The runtime for both D-FLAT configurations is 30 percent lower than the more efficient among the clingo configurations, while the memory consumption is 30 times lower. The instance Metro2, which contains only 4 vertices that must be visited, while the others can be visited as often as desired, cannot be solved with our D-FLAT encodings in less than 60 minutes, whereas clingo performs better. When restricting the maximum allowed number of visits to 1 for the vertices that do not have to be visited, D-FLAT becomes faster on semi-normalized tree decompositions than clingo with configuration 'auto' by more than 20 percent, while the memory consumption is still 1.5 times worse. Yet, when adding more vertices that must be visited, namely 10 in total in instance Metro4, the improvement becomes more visible. D-FLAT is already twice as fast as clingo and also the memory consumption is by a modest 7 percent lower. This suggests the existence of a trend in which when working on the same graph, the runtime and memory consumption decrease with the increase of the number of vertices that must be visited but also of the maximum visits restriction for the other vertices, for both the clingo and D-FLAT encodings, but more rapidly for D-FLAT. The results for instance Metro1 are also in favor of this hypothesis. Further, the rather weak performance of our TSP-R encoding for D-FLAT on weakly normalized tree decompositions of the instances Metro3 and Metro4 needs further consideration. When looking at the memory consumption we can see that clingo consumes almost exactly the same amount of memory for every instance, which means that its memory consumption does not depend on the number of vertices to be visited but only on the number of vertices in the graph, unlike D-FLAT's which varies with the number of vertices to be visited. In addition, we can say that there is no clear correlation between the number of conflicts and the runtime or the memory consumption.

The efficiency of the TSP-R encodings for D-FLAT with default join on semi-normalized and weakly normalized tree decompositions, from Listings A.6 and A.7, respectively, compared to the ASP TSP-R encoding, from Listing A.2, on the instances based on the Viennese metro and urban train system, presented in Table 5.3, is shown in Figure 5.16. The same configurations as for the tramway system were used, and the graphics are structured in the same way as those in Figure 5.15.

The instance Tram1, which is not satisfiable, can be solved almost 3.5 times faster by D-FLAT on weakly normalized trees than by clingo and consumes with this configuration 21

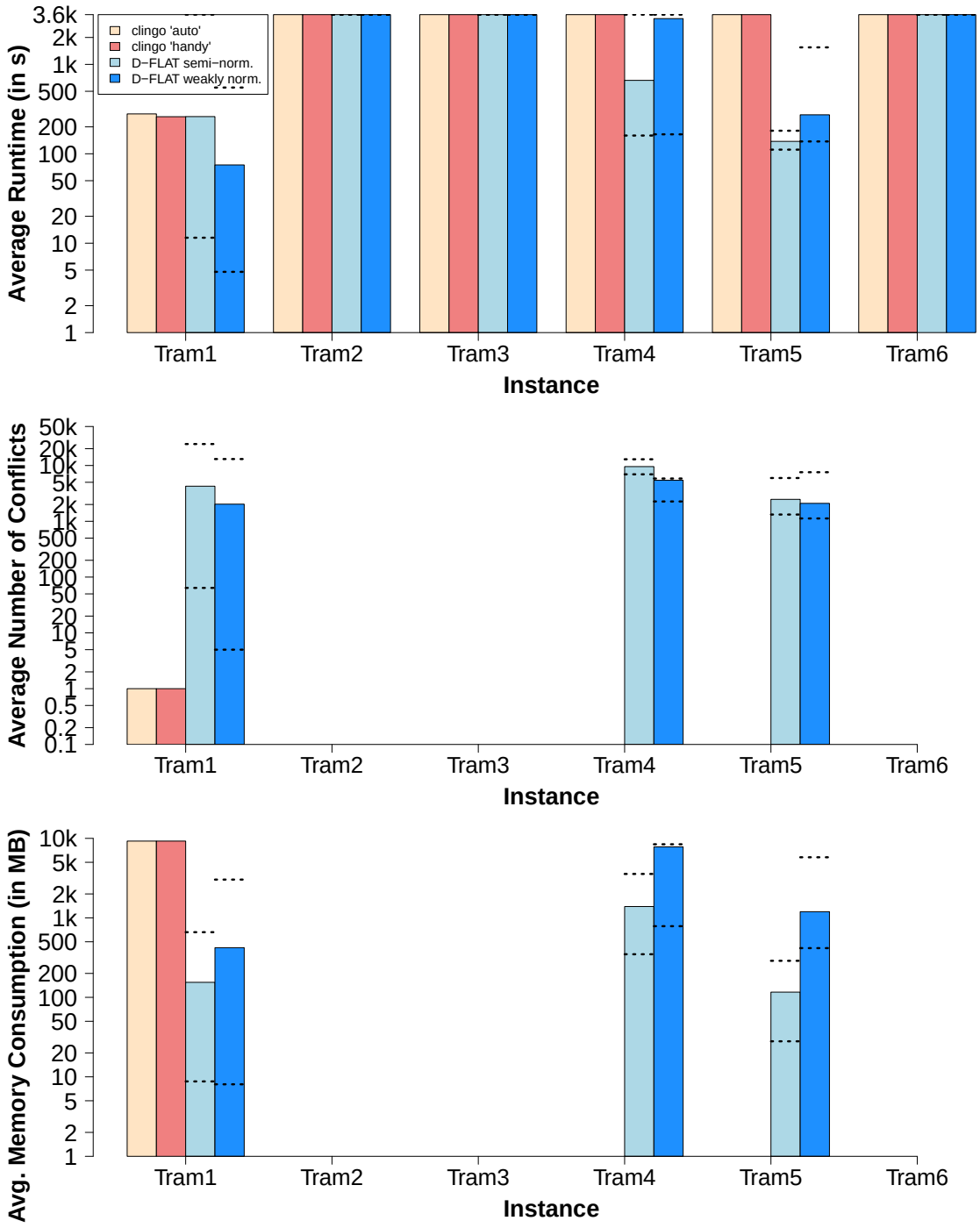


Figure 5.16: Performance of TSP-R encodings for D-FLAT, Listings A.6 and A.7, versus ASP TSP-R encoding, Listing A.2, on the instances based on the Viennese tramway system.

times less memory than clingo. While clingo cannot solve any of the other (satisfiable) instances in less than 60 minutes, D-FLAT can solve two of them and performs better on semi-normalized tree decompositions than on weakly normalized ones. Instance Tram2 is similar to Tram1 with the difference that it is satisfiable, which in this case influences the runtime negatively. Instance Tram3 is similar to Tram2, but now the vertices that do not have to be visited may be visited at most once. This does not bring the runtime under the 60-minutes mark. When we increase the number of vertices that must be visited to 15, in Tram4, the instance becomes solvable by D-FLAT on semi-normalized tree decompositions. For only one seed did D-FLAT need more than 60 minutes with Configuration 2. However, with Configuration 3, namely when working on weakly normalized tree decompositions, only for four runs with different seeds it managed to solve the instance in less than one hour. The other test runs were counted with 60 minutes, and with the maximum number of conflicts and maximum memory consumption among the successful test runs for the same instance, respectively. Now, when increasing again the number of vertices that must be visited to 24 in instance Tram5, the runtime and the memory consumption decrease once again. However, when increasing also the number of possible visits for the remaining vertices to 10 in Tram6, the instance becomes unsatisfiable. This confirms our hypothesis that the runtime and memory consumption decrease with the growth of the number of vertices that must be visited and with the restriction for the maximum number of visits for the remaining vertices. This can actually be easily explained by the fact that, the more restrictions we set for the number of visits, the smaller becomes the number of answer set candidates that are kept, improving both the runtime and the memory consumption. Further, we can once again see no correlation between the number of conflicts and the runtime or the memory consumption. We can also notice that the memory consumption for these instances with treewidth 6 and 402 vertices is higher by a multiple than those based on the metro and urban train system, that have treewidth 5 and 138 vertices.

5.4 Discussion of the Results

As the tests on Data Set 1, but also on Data Sets 3 and 5, have proven, our TSP-NR and TSP-R implementations for D-FLAT perform much better than those for clingo on full grids and grid-like graphs, in terms of runtime and memory consumption when the default join is used. More to the point, our TSP-NR for D-FLAT with default join on semi-normalized tree decompositions performs faster than clingo on 8-connected full grids up to a treewidth of 4 (inclusive). As for the real world instances, even if the ones we chose had (practical) treewidths 5 and 6, respectively, they could be solved in reasonable time if we chose to visit as many vertices as possible without making the TSP-NR unsatisfiable and restricted the visits to the remaining nodes to at most 1, such as we did in the instance Metro1. The latter is in fact not even mandatory, yet more tests would be necessary to determine more precisely based also on the treewidth and the size of an instance, how many vertices must be visited and how much the remaining ones must be restricted in their number of visits, such that it is solvable in reasonable time. For data generated by us the contrary holds. Even if the tests on Data Set 5 showed an increase in runtime when part of the vertices had to be visited twice, no substantial differences were noted when these vertices were left out of the tour or when they could be but did not have to be visited. The reason could be the

structure of the real world instances which is different from that of a full grid.

Further, we could observe both on Data Set 1 and on the real world instances that the memory consumption is more robust than the runtime, the latter fluctuating more. Moreover, while the memory consumption for clingo depends only on the graph, for D-FLAT it varies also according to the number of vertices to be visited.

Based on the outcomes of the tests on Data Sets 3, 4 and 5, we know that the range between the highest and the lowest runtime or memory consumption for a scenario is much wider for unsatisfiable instances, when working with D-FLAT, due to different configurations of the tree decompositions. Further, we have seen that the range between minimum and maximum values is wider also for grid-like graphs than for full grids, based on evidence from Data Set 3. At the same time it was wider also for the real world instances we covered. We assume that this holds in general for real world instances as also the grid-like graphs are closer to the latter than full grids.

Another insight we achieved was that for both systems the used configuration can play an important role. While for the full grids, the use of special clingo configurations brought rather moderate improvements, when handling real world instances they sometimes made a major difference. We also observed that D-FLAT on weakly normalized is more suited for unsatisfiable instances, while semi-normalized tree decompositions are more suitable for satisfiable instances. This could be clearly seen on Data Set 4 and on the instances based on the Viennese metro and urban train system.

Last but not least, a wider range of weights leads to a moderate increase in runtime due to the higher number of instantiations of aggregate functions which appears to be high enough to affect the runtime but not the memory consumption.

CHAPTER 6

Conclusion

6.1 Summary

The traveling salesperson problem is an NP-hard combinatorial optimization problem, that finds application in domains spanning from logistics to even psychology. There are various methods to solve this important problem, both exact algorithms and heuristic approaches. However, these approaches either have a rather low flexibility and maintainability, or need long running times.

In our work we developed a concept for solving two variations of the TSP, namely the traveling salesperson problem without repetitions and the traveling salesperson problem with repetitions. For this, we used the D-FLAT framework, that takes advantage of the fact that both problem variations are fixed-parameter tractable. D-FLAT combines the advantages of declarative programming, namely high maintainability and flexibility, with the reduced running time due to the use of dynamic programming. At the same time, our concept represents an exact method, always delivering an optimal solution in case one exists. We proposed several encodings for both variations of the TSP, for different configurations of D-FLAT and compared them to classical ASP algorithms. Furthermore, by presenting encodings for solving the TSP-R, we offered a solution to a problem which proved to be more suitable for real world instances, and exemplified the high maintainability and flexibility of our approach.

Our encodings for D-FLAT considerably outperformed the classical ASP encodings on our generated instances with treewidths 2, 3 and 4. For instances based on the tram, and the metro and urban train systems of Vienna's public transportation system, that have treewidths 6 and 5, respectively, the performance of the TSP-R depended very much on the number of vertices that were supposed to be visited, the more the better, and on the restrictions for the remaining vertices. Furthermore, we observed wide ranges between maximum and minimum running times and memory consumption depending on the configuration of the tree decompositions.

Thus, by proposing these implementations we provided a viable concept, with respect to running time and flexibility, which is applicable also on real world instances and delivers an optimal solution. Moreover, it is a concept that can be adapted also to encode other problems

which are related to the TSP and can be solved by D-FLAT in fixed-parameter tractable time as well.

6.2 Future Work

Even considering the aforementioned results, it still remains an issue at stake to determine which are the features of a real world graph that our method is suitable for. More to the point, it remains to find out more precisely starting from which percentage of vertices that must be visited, and up to which treewidth, our method becomes viable. Further, it would be interesting to know for which types of real world instances our method is suitable, for example if it is rather suitable for rail networks, or also for street networks, and which kinds thereof.

Another challenge will be to extract features of tree decompositions in order to be able to determine in advance for which of them we achieve better performances. The wide range between the best and the worst performance of an instance in terms of running time and memory consumption, especially for unsatisfiable instances, is still an issue, which on the other hand gives us the opportunity to improve the efficiency towards the minimum bound.

Further, a continuation of our work could consist in solving a variation of the TSP in which the tour must not be completed by one vehicle but two or more. The greatest challenge consists in ensuring connectedness. While in our implementations we had to guarantee that the output tour is connected, here it must be made sure that each of the multiple tours is connected and that there is no connection between the tours.

APPENDIX A

Collection of All Proposed Encodings

```
1  0 { s(X,Y) } 1 :- edge(X,Y), X < Y.
2  :- N = #count { Y : s(X,Y); Y : s(Y,X) }, vertex(X), N != 2.
3  c(X,Y) :- s(X,Y).
4  c(X,Z) :- c(X,Y), c(Y,Z).
5  c(X,Z) :- c(X,Y), c(Z,Y), X < Z.
6  c(Z,X) :- c(X,Y), c(Z,Y), X > Z.
7  c(X,Z) :- c(Y,X), c(Y,Z), X < Z.
8  c(Z,X) :- c(Y,X), c(Y,Z), X > Z.
9  :- not c(X,Y), vertex(X), vertex(Y), X < Y.
10 minWeight(X,Y,MW) :- s(X,Y),
       MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
11 cost(C) :- C = #sum { W,X,Y : minWeight(X,Y,W) }.
12 #minimize { C : cost(C) }.
13 #show s/2.
```

Listing A.1: ASP TSP-NR encoding that relies on the connectedness of a guessed tour.

```
0 { s(X,Y) } 1 :- edge(X,Y), X < Y.

minStated(X) :- vertex(X), minVisits(X,V).
maxStated(X) :- vertex(X), maxVisits(X,V).
minV(X,1) :- vertex(X), not minStated(X).
maxV(X,1) :- vertex(X), not maxStated(X).
minV(X,V) :- vertex(X), minVisits(X,V).
maxV(X,V) :- vertex(X), maxVisits(X,V).

:- N = #count { Y : s(X,Y); Y : s(Y,X) }, vertex(X), minVisits(X,V), N < 2*V.
:- N = #count { Y : s(X,Y); Y : s(Y,X) }, vertex(X), maxVisits(X,V), N > 2*V.
:- N = #count { Y : s(X,Y); Y : s(Y,X) }, vertex(X), N/2*2 != N.

c(X,Y) :- s(X,Y).
c(X,Z) :- c(X,Y), c(Y,Z).
c(X,Z) :- c(X,Y), c(Z,Y), X < Z.
c(Z,X) :- c(X,Y), c(Z,Y), X > Z.
c(X,Z) :- c(Y,X), c(Y,Z), X < Z.
c(Z,X) :- c(Y,X), c(Y,Z), X > Z.

:- not c(X,Y), s(X,_), s(Y,_), X < Y.
:- not c(X,Y), s(X,_), s(_,Y), X < Y.
:- not c(X,Y), s(_,X), s(Y,_), X < Y.
:- not c(X,Y), s(_,X), s(_,Y), X < Y.

minWeight(X,Y,MW) :- s(X,Y),
     MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
cost(C) :- C = #sum { W,X,Y : minWeight(X,Y,W) }.

#minimize { C : cost(C) }.
#show s/2.
```

Listing A.2: ASP TSP-R encoding that relies on the connectedness of a guessed tour.

```
1  %dflat: -e vertex -e edge --tables -n semi

2  %Guess row to be extended.
3  1 { extend(R): childRow(R,CH) } 1 :- childNode(CH).

4  %Guess introduced vertices' adjacent edges' selection for the tour.
5  0 { item(s(X,Y)) } 1 :- edge(X,Y), introduced(X), current(Y), X < Y.
6  0 { item(s(Y,X)) } 1 :- edge(X,Y), introduced(X), current(Y), X > Y.
7  item(s(X,Y)) :- childItem(R,s(X,Y)), extend(R), current(X;Y).

8  %Remove join nodes' table rows with different edge selection in distinct child nodes.
9  :- extend(R), extend(S), R!=S, childItem(R,s(X,Y)), not childItem(S,s(X,Y)).

10 %Count number of selected adjacent edges.
11 item(ct(X,N0)) :-  1 #count { CH : childNode(CH) } 1, introduced(X),
       N0 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
12 item(ct(X,N1+N0)) :- 1 #count { CH : childNode(CH) } 1,
       childItem(R,ct(X,N1)), extend(R), current(X),
       N0 = #count { Y : item(s(X,Y)), introduced(Y);
                     Y : item(s(Y,X)), introduced(Y) }.
13 item(ct(X,N1+N2-N12)) :- extend(R), extend(S), R!=S,
       childItem(R,ct(X,N1)), childItem(S,ct(X,N2)), current(X),
       N12 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.

14 %Eliminate answer set candidates which do not lead to a permutation without repetition.
15 :- childItem(R,ct(X,N)), extend(R), removed(X), N != 2.
16 :- item(ct(X,N)), N > 2.

17 %Deduce connectedness.
18 item(c(X,Y)) :- item(s(X,Y)).
19 item(c(X,Z)) :- item(c(X,Y)), item(c(Y,Z)).
20 item(c(X,Z)) :- item(c(X,Y)), item(c(Z,Y)), X < Z.
21 item(c(Z,X)) :- item(c(X,Y)), item(c(Z,Y)), X > Z.
22 item(c(X,Z)) :- item(c(Y,X)), item(c(Y,Z)), X < Z.
23 item(c(Z,X)) :- item(c(Y,X)), item(c(Y,Z)), X > Z.
24 item(c(X,Y)) :- childItem(R,c(X,Y)), extend(R), current(X;Y).

25 %Eliminate unconnected answer set candidates.
26 :- 1 #count{ U : current(U) }, removed(X), extend(R),
       not childItem(R,c(X,Y)) : current(Y);
       not childItem(R,c(Y,X)) : current(Y).
27 :- final, not childItem(R,c(X,Y)), X < Y, extend(R), removed(X;Y).

28 %Optional code for the case that the input graph's connectedness was not verified.
29 :- 1 #count { X : bag(N,X), childNode(N) }, not final, not oldVertex.
30 oldVertex :- current(X), not introduced(X).

31 %Calculate costs.
32 cost(0) :- initial.
33 cost(CC+NC) :- childCost(R,CC), extend(R),
       1 #count { CH : childNode(CH) } 1,
       NC = #sum { W,X,Y : relevantWeight(X,Y,W), introduced(X);
                   W,X,Y : relevantWeight(X,Y,W), introduced(Y) }.
34 cost(CC1+CC2-LC) :- extend(R), extend(S), R < S,
       childCost(R,CC1), childCost(S,CC2),
       LC = #sum { W,X,Y : relevantWeight(X,Y,W) }.
35 relevantWeight(X,Y,MW) :- item(s(X,Y)),
       MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
```

Listing A.3: TSP-NR encoding for D-FLAT without default join on semi-normalized tree decompositions.

```
%dflat: -e vertex -e edge --tables -n semi --default-join --post-join

%Guess row to be extended.
1 { extend(R): childRow(R,CH) } 1 :- childNode(CH).

%Guess introduced vertices' adjacent edges' selection for the tour.
0 { item(s(X,Y)) } 1 :- edge(X,Y), introduced(X), current(Y), X < Y.
0 { item(s(Y,X)) } 1 :- edge(X,Y), introduced(X), current(Y), X > Y.
item(s(X,Y)) :- childItem(R,s(X,Y)), extend(R), current(X;Y).

%Count number of selected adjacent edges.
auxItem(ct(X,N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
     extend(R), introduced(X), currentNode(CR),
     N0 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
auxItem(ct(X,N1+N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
     extend(R), current(X), currentNode(CR),
     childAuxItem(R,ct(X,N1,CH1)),
     N0 = #count { Y : item(s(X,Y)), introduced(Y);
                   Y : item(s(Y,X)), introduced(Y) }.
auxItem(ct(X,N1+N2-N12,CR)) :-
     childAuxItem(R,ct(X,N1,CH1)), childAuxItem(R,ct(X,N2,CH2)), CH1 != CH2,
     extend(R), currentNode(CR),
     N12 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
auxItem(n(CR)) :- currentNode(CR).

%Eliminate answer set candidates which do not lead to a permutation without repetition.
:- childAuxItem(R,ct(X,N,CH)), extend(R), removed(X), N != 2.
:- auxItem(ct(X,N,_)), N > 2.

%Deduce connectedness.
auxItem(c(X,Y)) :- item(s(X,Y)).
auxItem(c(X,Z)) :- auxItem(c(X,Y)), auxItem(c(Y,Z)).
auxItem(c(X,Z)) :- auxItem(c(X,Y)), auxItem(c(Z,Y)), X < Z.
auxItem(c(Z,X)) :- auxItem(c(X,Y)), auxItem(c(Z,Y)), X > Z.
auxItem(c(X,Z)) :- auxItem(c(Y,X)), auxItem(c(Y,Z)), X < Z.
auxItem(c(Z,X)) :- auxItem(c(Y,X)), auxItem(c(Y,Z)), X > Z.
auxItem(c(X,Y)) :- childAuxItem(R,c(X,Y)), extend(R), current(X;Y).

%Eliminate unconnected answer set candidates.
:- 1 #count{ U : current(U) }, removed(X), extend(R),
     not childAuxItem(R,c(X,Y)) : current(Y);
     not childAuxItem(R,c(Y,X)) : current(Y).
:- final, not childAuxItem(R,c(X,Y)), X < Y, extend(R), removed(X;Y).

%Optional code for the case that the input graph's connectedness was not verified.
:- 1 #count { X : bag(N,X), childNode(N) }, not final, not oldVertex.
oldVertex :- current(X), not introduced(X).

%Calculate costs.
cost(0) :- initial.
cost(CC+NC) :- not initial, childCost(R,CC), extend(R),
     NC = #sum { W,X,Y : relevantWeight(X,Y,W), introduced(X);
                 W,X,Y : relevantWeight(X,Y,W), introduced(Y) }.
currentCost(C) :- C = #sum { W,X,Y : relevantWeight(X,Y,W) }.
relevantWeight(X,Y,MW) :- item(s(X,Y)),
     MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
```

Listing A.4: TSP-NR encoding for D-FLAT with default join on semi-normalized tree decompositions.

```
1  %dflat: -e vertex -e edge --tables -n semi --default-join --post-join
2  %Guess row to be extended.
3  1 { extend(R): childRow(R,CH) } 1 :- childNode(CH).
4  %Guess introduced vertices' adjacent edges' selection for the tour.
5  0 { item(s(X,Y)) } 1 :- edge(X,Y), introduced(X), current(Y), X < Y.
6  0 { item(s(Y,X)) } 1 :- edge(X,Y), introduced(X), current(Y), X > Y.
7  item(s(X,Y)) :- childItem(R,s(X,Y)), extend(R), current(X;Y).
8  %Count number of selected adjacent edges.
9  auxItem(ct(X,N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
       extend(R), introduced(X), currentNode(CR),
       N0 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
10 auxItem(ct(X,N1+N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
       extend(R), current(X), currentNode(CR),
       childAuxItem(R,ct(X,N1,CH1)),
       N0 = #count { Y : item(s(X,Y)), introduced(Y);
                     Y : item(s(Y,X)), introduced(Y) }.
11 auxItem(ct(X,NC-((NCH-1)*N12),CR)) :- current(X), currentNode(CR),
       NCH = #count { CH: childAuxItem(R,n(CH)) }, extend(R), NCH > 1,
       NC = #sum { N,CH: childAuxItem(R,ct(X,N,CH))},
       N12 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
12 auxItem(n(CR)) :- currentNode(CR).
13 %Eliminate answer set candidates which do not lead to a permutation without repetition.
14 :- childAuxItem(R,ct(X,N,CH)), extend(R), removed(X), N != 2.
15 :- auxItem(ct(X,N,_)), N > 2.
16 %Deduce connectedness.
17 auxItem(c(X,Y)) :- item(s(X,Y)).
18 auxItem(c(X,Z)) :- auxItem(c(X,Y)), auxItem(c(Y,Z)).
19 auxItem(c(X,Z)) :- auxItem(c(X,Y)), auxItem(c(Z,Y)), X < Z.
20 auxItem(c(Z,X)) :- auxItem(c(X,Y)), auxItem(c(Z,Y)), X > Z.
21 auxItem(c(X,Z)) :- auxItem(c(Y,X)), auxItem(c(Y,Z)), X < Z.
22 auxItem(c(Z,X)) :- auxItem(c(Y,X)), auxItem(c(Y,Z)), X > Z.
23 auxItem(c(X,Y)) :- childAuxItem(R,c(X,Y)), extend(R), current(X;Y).
24 %Eliminate unconnected answer set candidates.
25 :- 1 #count{ U : current(U) }, removed(X), extend(R),
       not childAuxItem(R,c(X,Y)) : current(Y);
       not childAuxItem(R,c(Y,X)) : current(Y).
26 :- final, not childAuxItem(R,c(X,Y)), X < Y, extend(R), removed(X;Y).
27 %Optional code for the case that the input graph's connectedness was not verified.
28 :- 1 #count { X : bag(N,X), childNode(N) }, not final, not oldVertex.
29 oldVertex :- current(X), not introduced(X).
30 %Calculate costs.
31 cost(0) :- initial.
32 cost(CC+NC) :- not initial, childCost(R,CC), extend(R),
       NC = #sum { W,X,Y : relevantWeight(X,Y,W), introduced(X);
                   W,X,Y : relevantWeight(X,Y,W), introduced(Y) }.
33 currentCost(C) :- C = #sum { W,X,Y : relevantWeight(X,Y,W) }.
34 relevantWeight(X,Y,MW) :- item(s(X,Y)),
       MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
```

Listing A.5: TSP-NR encoding for D-FLAT with default join on weakly normalized tree decompositions.

```
%dflat: -e vertex -e edge --tables -n semi --default-join --post-join
%Guess row to be extended.
1 { extend(R): childRow(R,CH) } 1 :- childNode(CH).
%Guess introduced vertices' adjacent edges' selection for the tour.
0 { item(s(X,Y)) } 1 :- edge(X,Y), introduced(X), current(Y), X < Y.
0 { item(s(Y,X)) } 1 :- edge(X,Y), introduced(X), current(Y), X > Y.
item(s(X,Y)) :- childItem(R,s(X,Y)), extend(R), current(X;Y).
%Count number of selected adjacent edges.
auxItem(ct(X,N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
     extend(R), introduced(X), currentNode(CR),
     N0 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
auxItem(ct(X,N1+N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
     extend(R), current(X), currentNode(CR),
     childAuxItem(R,ct(X,N1,CH1)),
     N0 = #count { Y : item(s(X,Y)), introduced(Y);
                   Y : item(s(Y,X)), introduced(Y) }.
auxItem(ct(X,N1+N2-N12,CR)) :-
     childAuxItem(R,ct(X,N1,CH1)), childAuxItem(R,ct(X,N2,CH2)), CH1 != CH2,
     extend(R), currentNode(CR),
     N12 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
auxItem(n(CR)) :- currentNode(CR).
%Eliminate solution candidates that do not lead to a valid permutation with repetition.
:- childAuxItem(R,ct(X,N,CH)), extend(R), minV(X,V), removed(X), N < 2 * V.
:- childAuxItem(R,ct(X,N,CH)), extend(R), maxV(X,V), removed(X), N > 2 * V.
:- childAuxItem(R,ct(X,N,CH)), extend(R), removed(X), N / 2 * 2 != N.
:- maxV(X,V), auxItem(ct(X,N,_)), N > 2 * V.
minV(X,1) :- currem(X), not minStated(X).
maxV(X,1) :- currem(X), not maxStated(X).
minStated(X) :- currem(X), minVisits(X,V).
maxStated(X) :- currem(X), maxVisits(X,V).
minV(X,V) :- currem(X), minVisits(X,V).
maxV(X,V) :- currem(X), maxVisits(X,V).
currem(X) :- current(X).
currem(X) :- removed(X).
```

```
26 %Deduce connectedness.
27 auxItem(c(X,Y)) :- item(s(X,Y)).
28 auxItem(c(X,Z)) :- auxItem(c(X,Y)), auxItem(c(Y,Z)).
29 auxItem(c(X,Z)) :- auxItem(c(X,Y)), auxItem(c(Z,Y)), X < Z.
30 auxItem(c(Z,X)) :- auxItem(c(X,Y)), auxItem(c(Z,Y)), X > Z.
31 auxItem(c(X,Z)) :- auxItem(c(Y,X)), auxItem(c(Y,Z)), X < Z.
32 auxItem(c(Z,X)) :- auxItem(c(Y,X)), auxItem(c(Y,Z)), X > Z.
33 auxItem(c(X,Y)) :- childAuxItem(R,c(X,Y)), extend(R), current(X;Y).

34 %Eliminate unconnected answer set candidates.
35 :- 1 #count{ U : current(U) }, removed(X),
       not childAuxItem(R,ct(X,0,_)), extend(R),
       not childAuxItem(R,c(X,Y)) : current(Y);
       not childAuxItem(R,c(Y,X)) : current(Y).
36 :- final, removed(X;Y), X < Y, not childAuxItem(R,c(X,Y)), extend(R),
       not childAuxItem(R,ct(X,0,_)), not childAuxItem(R,ct(Y,0,_)).
37 %Optional code for the case that the input graph's connectedness was not verified.
38 auxItem(noselection(CR)) :- 1 #count { X : bag(N,X), childNode(N) },
       not final, not oldVertex, auxItem(selection), currentNode(CR).
39 oldVertex :- current(X), not introduced(X).
40 auxItem(selection) :- item(s(X,Y)).
41 auxItem(noselection(CR)) :- childAuxItem(R,noselection(CH)),
       extend(R), currentNode(CR).
42 auxItem(selection) :- childAuxItem(R,selection),
       not auxItem(noselection(CR)), extend(R).
43 :- auxItem(selection), auxItem(noselection(CR)).
44 :- auxItem(noselection(CH1)), auxItem(noselection(CH2)), CH1 < CH2.

45 %Calculate costs.
46 cost(0) :- initial.
47 cost(CC+NC) :- not initial, childCost(R,CC), extend(R),
       NC = #sum { W,X,Y : relevantWeight(X,Y,W), introduced(X);
                   W,X,Y : relevantWeight(X,Y,W), introduced(Y) }.
48 currentCost(C) :- C = #sum { W,X,Y : relevantWeight(X,Y,W) }.
49 relevantWeight(X,Y,MW) :- item(s(X,Y)),
       MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
```

Listing A.6: TSP-R encoding for D-FLAT with default join on semi-normalized tree decompositions.

```
%dflat: -e vertex -e edge --tables -n weak --default-join --post-join

%Guess row to be extended.
1 { extend(R): childRow(R,CH) } 1 :- childNode(CH).

%Guess introduced vertices' adjacent edges' selection for the tour.
0 { item(s(X,Y)) } 1 :- edge(X,Y), introduced(X), current(Y), X < Y.
0 { item(s(Y,X)) } 1 :- edge(X,Y), introduced(X), current(Y), X > Y.
item(s(X,Y)) :- childItem(R,s(X,Y)), extend(R), current(X;Y).

%Count number of selected adjacent edges.
auxItem(ct(X,N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
     extend(R), introduced(X), currentNode(CR),
     N0 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
auxItem(ct(X,N1+N0,CR)) :- 1 #count { CH : childAuxItem(R,n(CH)) } 1,
     extend(R), current(X), currentNode(CR),
     childAuxItem(R,ct(X,N1,CH1)),
     N0 = #count { Y : item(s(X,Y)), introduced(Y);
                   Y : item(s(Y,X)), introduced(Y) }.
auxItem(ct(X,NC-((NCH-1)*N12),CR)) :- current(X), currentNode(CR),
     NCH = #count { CH: childAuxItem(R,n(CH)) }, extend(R), NCH > 1,
     NC = #sum { N,CH: childAuxItem(R,ct(X,N,CH))},
     N12 = #count { Y : item(s(X,Y)); Y : item(s(Y,X)) }.
auxItem(n(CR)) :- currentNode(CR).

%Eliminate solution candidates that do not lead to a valid permutation with repetition.
:- childAuxItem(R,ct(X,N,CH)), extend(R), minV(X,V), removed(X), N < 2 * V.
:- childAuxItem(R,ct(X,N,CH)), extend(R), maxV(X,V), removed(X), N > 2 * V.
:- childAuxItem(R,ct(X,N,CH)), extend(R), removed(X), N / 2 * 2 != N.
:- maxV(X,V), auxItem(ct(X,N,_)), N > 2 * V.

minV(X,1) :- currem(X), not minStated(X).
maxV(X,1) :- currem(X), not maxStated(X).
minStated(X) :- currem(X), minVisits(X,V).
maxStated(X) :- currem(X), maxVisits(X,V).
minV(X,V) :- currem(X), minVisits(X,V).
maxV(X,V) :- currem(X), maxVisits(X,V).
currem(X) :- current(X).
currem(X) :- removed(X).
```

```
26 %Deduce connectedness.
27 auxItem(c(X,Y)) :- item(s(X,Y)).
28 auxItem(c(X,Z)) :- auxItem(c(X,Y)), auxItem(c(Y,Z)).
29 auxItem(c(X,Z)) :- auxItem(c(X,Y)), auxItem(c(Z,Y)), X < Z.
30 auxItem(c(Z,X)) :- auxItem(c(X,Y)), auxItem(c(Z,Y)), X > Z.
31 auxItem(c(X,Z)) :- auxItem(c(Y,X)), auxItem(c(Y,Z)), X < Z.
32 auxItem(c(Z,X)) :- auxItem(c(Y,X)), auxItem(c(Y,Z)), X > Z.
33 auxItem(c(X,Y)) :- childAuxItem(R,c(X,Y)), extend(R), current(X;Y).

34 %Eliminate unconnected answer set candidates.
35 :- 1 #count{ U : current(U) }, removed(X),
       not childAuxItem(R,ct(X,0,_)), extend(R),
       not childAuxItem(R,c(X,Y)) : current(Y);
       not childAuxItem(R,c(Y,X)) : current(Y).
36 :- final, removed(X;Y), X < Y, not childAuxItem(R,c(X,Y)), extend(R),
       not childAuxItem(R,ct(X,0,_)), not childAuxItem(R,ct(Y,0,_)).
37 %Optional code for the case that the input graph's connectedness was not verified.
38 auxItem(noselection(CR)) :- 1 #count { X : bag(N,X), childNode(N) },
       not final, not oldVertex, auxItem(selection), currentNode(CR).
39 oldVertex :- current(X), not introduced(X).
40 auxItem(selection) :- item(s(X,Y)).
41 auxItem(noselection(CR)) :- childAuxItem(R,noselection(CH)),
       extend(R), currentNode(CR).
42 auxItem(selection) :- childAuxItem(R,selection),
       not auxItem(noselection(CR)), extend(R).
43 :- auxItem(selection), auxItem(noselection(CR)).
44 :- auxItem(noselection(CH1)), auxItem(noselection(CH2)), CH1 < CH2.

45 %Calculate costs.
46 cost(0) :- initial.
47 cost(CC+NC) :- not initial, childCost(R,CC), extend(R),
       NC = #sum { W,X,Y : relevantWeight(X,Y,W), introduced(X);
                   W,X,Y : relevantWeight(X,Y,W), introduced(Y) }.
48 currentCost(C) :- C = #sum { W,X,Y : relevantWeight(X,Y,W) }.
49 relevantWeight(X,Y,MW) :- item(s(X,Y)),
       MW = #min { W,X,Y : weight(X,Y,W); W,X,Y : weight(Y,X,W) }.
```

Listing A.7: TSP-R encoding for D-FLAT with default join on weakly normalized tree decompositions.

Bibliography

[1] Michael Abseher, Bernhard Bliem, Günther Charwat, Frederico Dusberger, Markus Hecher, and Stefan Woltran. D-FLAT: Progress report. Technical report, Technical Report DBAI-TR-2014-86, Vienna University of Technology, 2014.

[2] Rachit Agarwal, Philip B. Godfrey, and Sariel Har-Peled. Approximate distance queries and compact routing in sparse graphs. In *INFOCOM 2011. 30th IEEE International Conference on Computer Communications, Joint Conference of the IEEE Computer and Communications Societies, 10-15 April 2011, Shanghai, China*, pages 1754–1762. IEEE, 2011.

[3] Mario Alviano, Carmine Dodaro, Wolfgang Faber, Nicola Leone, and Francesco Ricca. WASP: A native ASP solver based on constraint learning. In *Logic Programming and Nonmonotonic Reasoning, 12th International Conference, LPNMR 2013, Corunna, Spain, September 15-19, 2013. Proceedings*, volume 8148 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 2013.

[4] David L. Applegate. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.

[5] David L. Applegate, Robert Bixby, William Cook, and Vašek Chvátal. *On the Solution of Traveling Salesman Problems*. Rheinische Friedrich-Wilhelms-Universität Bonn, 1998.

[6] Stefan Arnborg, Derek G. Corneil, and Andrzej Proskurowski. Complexity of finding embeddings in a k-tree. *SIAM J. Algebraic Discrete Methods*, 8(2):277–284, 1987.

[7] Richard Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM*, 9(1):61–63, 1962.

[8] Bernhard Bliem, Michael Morak, and Stefan Woltran. D-FLAT: Declarative problem solving using tree decompositions and answer-set programming. *TPLP*, 12(4-5):445–464, 2012.

[9] Bernhard Bliem, Reinhard Pichler, and Stefan Woltran. Declarative dynamic programming as an alternative realization of Courcelle's theorem. In *Parameterized and Exact Computation - 8th International Symposium, IPEC 2013, Sophia Antipolis, France, September 4-6, 2013, Revised Selected Papers*, volume 8246 of *Lecture Notes in Computer Science*, pages 28–40. Springer, 2013.

[10] Hans L. Bodlaender. A linear-time algorithm for finding tree-decompositions of small treewidth. *SIAM J. Comput.*, 25(6):1305–1317, 1996.

[11] Hans L. Bodlaender and Arie M. C. A. Koster. Combinatorial optimization on graphs of bounded treewidth. *Comput. J.*, 51(3):255–269, 2008.

[12] Hans L. Bodlaender and Arie M. C. A. Koster. Treewidth computations I. upper bounds. *Inf. Comput.*, 208(3):259–275, 2010.

[13] Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Commun. ACM*, 54(12):92–103, 2011.

[14] Günther Charwat. Tree-decomposition based algorithms for abstract argumentation frameworks. Master's thesis, Vienna University of Technology, 2012.

[15] Vasek Chvátal, William J. Cook, George B. Dantzig, Delbert R. Fulkerson, and Selmer M. Johnson. Solution of a large-scale traveling-salesman problem. *J. of the Operations Research Society of America*, pages 7–28, 2010.

[16] Keith L. Clark. Negation as failure. In *Logic and Data Bases*, pages 293–322. Springer, 1977.

[17] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog (4. ed.)*. Springer, 1994.

[18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[19] Harlan Crowder and Manfred W. Padberg. Solving large-scale symmetric travelling salesman problems to optimality. *Mgmt. Sci.*, 26(5):495–509, 1980.

[20] Rina Dechter. Bucket elimination: A unifying framework for reasoning. *Artif. Intell.*, 113(1-2):41–85, 1999.

[21] Artan Dermaku, Tobias Ganzow, Georg Gottlob, Benjamin J. McMahan, Nysret Musliu, and Marko Samer. Heuristic methods for hypertree decomposition. In *MICAI 2008: Advances in Artificial Intelligence, 7th Mexican International Conference on Artificial Intelligence, Atizapán de Zaragoza, Mexico, October 27-31, 2008, Proceedings*, volume 5317 of *Lecture Notes in Computer Science*, pages 1–11. Springer, 2008.

[22] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1):269–271, 1959.

[23] Marco Dorigo and Luca M. Gambardella. Ant colonies for the travelling salesman problem. *BioSystems*, 43(2):73–81, 1997.

[24] Stuart Dreyfus. Richard Bellman on the birth of dynamic programming. *Operations Research*, 50(1):48–51, 2002.

[25] Wolfgang Dvorák, Reinhard Pichler, and Stefan Woltran. Towards fixed-parameter tractable algorithms for abstract argumentation. *Artif. Intell.*, 186:1–37, 2012.

[26] Thomas Eiter, Wolfgang Faber, Nicola Leone, and Gerald Pfeifer. Declarative problem-solving using the DLV system. In *Logic-Based Artif. Intell.*, pages 79–103. Springer, 2000.

[27] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A user's guide to gringo, clasp, clingo, and iclingo, 2008.

[28] Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The potsdam answer set solving collection. *AI Communications*, 24(2):107–124, 2011.

[29] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. Conflict-driven answer set solving. In *IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007*, pages 386–392, 2007.

[30] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp* : A conflict-driven answer set solver. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 260–265. Springer, 2007.

[31] Martin Gebser, Torsten Schaub, and Sven Thiele. Gringo : A new grounder for answer set programming. In *Logic Programming and Nonmonotonic Reasoning, 9th International Conference, LPNMR 2007, Tempe, AZ, USA, May 15-17, 2007, Proceedings*, volume 4483 of *Lecture Notes in Computer Science*, pages 266–271. Springer, 2007.

[32] Allen Van Gelder, Kenneth A. Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[33] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 1070–1080. MIT Press, 1988.

[34] Georg Gottlob, Nicola Leone, and Francesco Scarcello. Hypertree decompositions and tractable queries. *J. Comput. Syst. Sci.*, 64(3):579–627, 2002.

[35] Georg Gottlob, Reinhard Pichler, and Fang Wei. Bounded treewidth as a key to tractability of knowledge representation and reasoning. *Artif. Intell.*, 174(1):105–132, 2010.

[36] Georg Gottlob and Stefan Szeider. Fixed-parameter algorithms for artificial intelligence, constraint satisfaction and database problems. *Comput. J.*, 51(3):303–325, 2008.

[37] Jens Gramm, Arfst Nickelsen, and Till Tantau. Fixed-parameter algorithms in phylogenetics. *Comput. J.*, 51(1):79–101, 2008.

[38] Gregory Gutin and Abraham P. Punnen. *The Traveling Salesman Problem and Its Variations*, volume 12. Springer, 2002.

[39] Steffen Hölldobler and Lukas Schweizer. Answer set programming and clasp a tutorial. In *Young Scientists' International Workshop on Trends in Information Processing*, page 77, 2014.

[40] Xiuzhen Huang and Jing Lai. Parameterized graph problems in computational biology. In *Proceeding of the Second International Multi-Symposium of Computer and Computational Sciences (IMSCCS 2007), August 13-15, 2007, The University of Iowa, Iowa City, Iowa, USA*, pages 129–132. IEEE, 2007.

[41] Dieter Jungnickel and Tilla Schade. *Graphs, Networks and Algorithms*. Springer, 2008.

[42] Roland Kaminski and Benjamin Kaufmann. *Answer Set Solving in Practice*, volume 19. Morgan & Claypool Publishers, 2012.

[43] Matthieu Latapy and Clémence Magnien. Measuring fundamental properties of real-world complex networks. *CoRR*, abs/cs/0609115, 2006.

[44] Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[45] Vladimir Lifschitz. Answer set programming and plan generation. *Artif. Intell.*, 138(1-2):39–54, 2002.

[46] Vladimir Lifschitz. What is answer set programming? In *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 1594–1597. AAAI Press, 2008.

[47] Shen Lin and Brian W. Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516, 1973.

[48] John D. C. Little, Katta G. Murty, Dura W. Sweeney, and Caroline Karel. An algorithm for the traveling salesman problem. *Operations Research*, 11(6):972–989, 1963.

[49] Victor W. Marek and Mirosław Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm*, volume cs.LO/9809032. Springer, 1998.

[50] Guy Melançon. Just how dense are dense graphs in the real world?: a methodological note. In *Proceedings of the 2006 AVI Workshop on BEyond time and errors: novel evaluation methods for information visualization, BELIV 2006, Venice, Italy, May 23, 2006*, pages 1–7. ACM Press, 2006.

[51] Ilkka Niemelä and Patrik Simons. Smodels - an implementation of the stable model and well-founded semantics for normal LP. In *Logic Programming and Nonmonotonic Reasoning, 4th International Conference, LPNMR'97, Dagstuhl Castle, Germany, July 28-31, 1997, Proceedings*, volume 1265 of *Lecture Notes in Computer Science*, pages 421–430. Springer, 1997.

[52] Raymond Reiter. On closed world data bases. In *Logic and Data Bases*, pages 55–76, 1977.

[53] Neil Robertson and Paul D. Seymour. Graph minors. III. planar tree-width. *J. Comb. Theory, Ser. B*, 36(1):49–64, 1984.

[54] Julia Robinson. On the hamiltonian game (a traveling salesman problem). *RAND Research Memorandum RM-303*, 1949.

[55] Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

[56] Mikkel Thorup. All structured programs have small tree-width and good register allocation. *Inf. Comput.*, 142(2):159–181, 1998.

[57] David B. Wagner. Dynamic programming. *The Mathematica Journal*, 5(4):42–51, 1995.