INSTITUT FÜR INFORMATIK
Algorithmische Bioinformatik

Universitätsstr. 1 D–40225 Düsseldorf

Heinrich Heine Universität Düsseldorf

Solving Dominating Set Using Answer Set Programming

My Ky Huynh

Bachelorarbeit

Beginn der Arbeit:	04. November 2019
Abgabe der Arbeit:	04. Februar 2020
Gutachter:	Prof. Dr. Gunnar Klau
	Prof. Dr. Michael Leuschel

Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbstständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

Düsseldorf, den 04. Februar 2020

My Ky Huynh

Abstract

Maximizing photosynthetic gains is one of the plant's many objectives. In this thesis, we present an optimal model for a leaf's venation pattern based on the minimum number of cells that have to turn into vein cells to supply the entire leaf with water and nutrients. The model focuses on the number of vein cells alone and consequently disregards a lot about the vascular system, like the vein hierarchy. To create our model we implement different minimum dominating set variants from graph theory in Answer Set Programming, a form of declarative programming. The most crucial one of these variants is the k-hop connected dominating set which we use to simulate the vascular system. Our results show that Answer Set Programming, while able to compute our model, requires too much solving time for larger inputs to be practical. Especially when compared to Integer Linear Programming, an alternative method for implementing our model. However, the comparison in this thesis is limited to just the k-hop component due to the unavailability of an Integer Linear Programming implementation of our model with connectivity.

Contents

1 Introduction

Plants optimize their architecture to fulfil many different competing objectives [3]. One of these objectives is to maximize photosynthesis production. However, in order for leaves to perform photosynthesis, they need water and nutrients like nitrogen [15] supplied by the vascular system. Leaf-veins consist of xylem cells supplying water from the leaf-stalk into the leaf and phloem cells transporting sugar out of the leaf into the rest of the plant [16]. In order to pass sugar from mesophyll cells, photosynthesis performing cells, to the phloem, mesophyll cells cannot be more than a few cells away from the leaf-veins [14, p. 469]. This means a leaf's venation is crucial for its photosynthetic gains. A way to maximize photosynthesis is to maximize the number of mesophyll cells and minimize the number of vein cells to the smallest number required to supply the entire leaf.

Figure 1: This picture shows a leaves whose venation pattern are clearly visible. This is just a snippet of the actual picture[1].

Here we describe a method for finding a leaf's optimal venation patterns based on graph theory by using Answer Set Programming (ASP), a form of declarative programming based on logic programming [9]. For our results, we use the ASP tools clingo, gringo and clasp from the Potsdam Answer Set Solving Collection called Potassco. To represent a leaf we use a graph with nodes as leaf cells and edges as connections between a cell

[1]Source: https://commons.wikimedia.org/wiki/File:Plant_Leaf_Venation_DSCN9018_28.jpg

and its neighbours. For the sake of convenience the vein hierarchy [16] is disregarded. To find suitable venation patterns, we use variants of the minimum dominating set and gradually modify them until we are able to replicate a pattern resembling a leaf's venation. A dominating set is a subset of nodes such that every node not in the subset has a neighbour that is in the subset. However, finding a minimum dominating set and many of its variants is NP-hard [6]. The goal of this thesis is to find optimal venation patterns with a minimal number of vein cells by implementing the dominating set and its variants in ASP.

A different method to solve this would be Integer Linear Programming (ILP). Comparing some of our results with those of an ILP solution shows that while ASP is not as fast as ILP for larger inputs, it might still be a passable alternative due to its versatile and simple language making it easy to modify programs. Due to the unavailability of a connected solution for ILP, we only compare the dominating set and the k-hop dominating set solutions with each other.

The rest of this thesis is structured in the following way. First, we introduce some basic theory on logic programming and ASP as well as the ASP tools used in this thesis in Section 2. Next in Section 3 we define the methods applied to determine the venation pattern of a leaf which include k-transitive closure and dominating set. Section 4 demonstrates how to implement those methods in ASP. In Section 5 we look at experimental results followed by a discussion on the effectiveness and limitations of the ASP solution as well as some problems and on how our ASP solution competes against one using ILP (Section 6). Finally, we conclude this thesis in Section 7.

2 Logic Programming

Logic Programming is based on first-order logic. Furthermore, one of the main ideas behind logic programming is an algorithm that consists of two components. A logic part describing what the problem is and a control part stating how it is supposed to be solved. This separation allows programmers to only have to specify the logic component while the logic programming system executes the control part [13]. The first part of this section establishes some semantics on logic programming necessary to understand ASP. However, since the first part only establishes the necessities for understanding ASP some parts about logic programming might be left out. Also, the definitions used here are taken from [4, 11, 13], but might be slightly modified to make them more understandable. The second part is dedicated to ASP itself. The third part describes how Potassco tools work.

2.1 Theory

This subsection defines all logic programming terms necessary to understand the basics of ASP in the next subsection and the implementations in Section 4. However, this subsection does not contain a complete guide on logic programming. For more information on logic programming see [13].

An *alphabet*, also called first order language, consists of variables, constants, function

symbols, predicate symbols, connectives, quantifiers and punctuation symbols. For this subsection we denote variables with $var_1, ..., var_n$, constants with $const_1, ..., const_n$, function symbols with $f_1, ..., f_n$ and predicates with $pre_1, ..., pre_n$. Quantifiers are $\exists$ meaning exists and $\forall$ meaning for all. We use $\wedge$ as conjunction and $\vee$ as disjunction. An implication is denoted with $\leftarrow$ where $A \leftarrow B$ means B implies A. A *term* is a constant, a variable or a function symbol with terms as arguments, also called a functional term, e.g., $f(t_1, ..., t_n)$ with $t_1, ..., t_n$ as terms. Similarly to a functional term, an *atom* is a predicate with terms as arguments, e.g., $pre(t_1, ..., t_n)$ with $t_1, ..., t_n$ as terms. Additionally, a *ground atom* is an atom without variables and functions as a propositional variable. This means it is either true or false. A *literal* is either an atom or a negated atom.

Most of the formal definitions for the above-mentioned terms can be found in [13] although specifications like the grounding or particular expressions can be found in [4]. The definitions below come from [4] and a few from [11].

Definition 1. A *normal logic program* is a finite set of rules in form of $h \leftarrow a_1, ..., a_i, \sim a_{i+1}, ..., \sim a_n$ with atoms $h, a_1, ..., a_i, a_{i+1}, ..., a_n$. Every rule consists of

- a head with $head(r) = h$ and
- a body with $body(r) = \{a_1, ..., a_i\} \cup \{\sim a_{i+1}, ..., \sim a_n\}$ with $body(r)^+ = \{a_1, ..., a_i\}$ and $body(r)^- = \{a_{i+1}, ..., a_n\}$

A rule with an empty body $h \leftarrow$, shortened h, is a fact meaning it is true. Rules can also be called clauses.

For a logic program P the set of all terms which can be formed from constants and function symbols is called the *Herbrand Universe* of P, denoted as $HU(P)$. On the other hand, the set of all ground atoms which can be formed from predicates and terms is called the *Herbrand Base* of P, denoted as $HB(P)$. A Herbrand interpretation I is a subset of $HB(P)$ over $HU(P)$. Intuitively, I denotes which ground atoms are true in a given instance. A ground instance of a rule r is obtained by substituting all variables in r with elements of $HU(P)$. All possible ground instances of r are denoted with $grnd(r)$ and the grounding of logic program P is $grnd(P) = \cup_{r \in P} grnd(r)$ [4].

Definition 2 (Negation as failure)**.** Let program P contain a rule r with $\sim a \in body(r)$. If a cannot be proven true then $\sim a$ is true.

Definition 3. The Interpretation I is a model of

- a ground rule $h \leftarrow b_1, ..., b_m, \sim b_{m+1}, ..., \sim b_n$ if either $body^+(r) \not\subseteq I$ or $(h \cup body^-(r)) \cap I \neq \emptyset$ (denoted as $I \models r$)
- a rule r if $I \models r'$ for every $r' \in grnd(r)$ (denoted as $I \models r$)
- a program P if $I \models r$ for $r \in P$ (denoted as $I \models P$).

$A \models B$ denotes A being a model of B.

Definition 4. The *reduct* of a program P with respect to an interpretation M, denoted as P^M is obtained by removing all rules with $\sim a$ in the body for each $a \in M$ and removing all literals $\sim a$ from all other rules. This means if $a \in M$ then every rule with $\sim a$ has to be false. Since for $a \notin M$ $\sim a$ is assumed to be true, M can be seen as an assumption of which negated literals are true or false [4].

Definition 5 (Stable model semantic for ground programs)**.** An interpretation M of a program P is a *stable model* of P, if P^M does not contradict M, meaning M is the minimal model of P, denoted as $M = LM(P^M)$.

Definition 6 (Stable model semantic for programs with variables)**.** An interpretation M is a stable model of a given program P, if M is a stable model of $grnd(P)$.
This means M is a stable model of P if M is a stable model of grounded rules of P.

There are three extensions of normal logic programs that are crucial in ASP [4]. As before the definitions are taken from [4]. First, the strong negation denoted with $-$. A strong negation is $-a$ only true if a can be proven false which is a contrast to the default negation $\sim$ which uses negation as failure. Secondly, a *disjunction* in the head of a rule $h_1(X) \vee ... \vee h_n(X) \leftarrow l_1(X), ..., l_m(X)$ declares that either $h_1(X)$ to $h_n(X)$ could be true. Lastly, the integrity constraint whose definition is as follows:

Definition 7 (Integrity constraints)**.** A *integrity constraint* is a rule with an empty head of the form $\leftarrow l_1, ..., l_n$ with the literals $l_1, ..., l_n$. It is equal to a rule of the form $false \leftarrow\sim false, l_1, ..., l_n$ where $false$ is a propositional atom.

It means there must not be a model where all literals of the integrity constraint are true. A logic program that uses strong negation is called an *extended logic program (ELP)* whereas a logic program using disjunctions and strong negation is called an *extended disjunctive logic program (EDLP)*. To take strong negations and disjunctions into consideration, the definition of a model needs to be slightly modified.

Definition 8 (Models for ELPs and EDLPs)**.** An interpretation I is a model of

- a ground rule $a_1 \vee ... \vee a_k \leftarrow b_1, ...b_m, \sim c_1, ..., \sim c_n$ if either $\{b_1, ..., b_n\} \not\subseteq I$ or $\{a_1, ..., a_k, \sim c_1, ..., \sim c_n\} \cap I \neq \emptyset$ (denoted as $I \models C$),
- a rule, if I denotes r' for every $r' \in grnd(r)$ (denoted $I \models r$)
- a program P, if $I \models r$ for every rule r in C

Note that $a_1, ..., a_k, b_1, ..., b_m, c_1, ..., c_n$ are atoms or strongly negated atoms.

In the next section, we introduce a few rules specifically used in ASP.

2.2 Answer Set Programming

Answer Set Programming focuses on NP-hard search problems [12] and enables solving all search problems in a uniform way [9]. A logic program P represents an instance

of a problem I where the models of P are solutions for I. An ASP solver computes models of P and outputs a solution for I [4]. Furthermore, ASP operates on a generate and test methodology with optional optimization [9]. First, solution candidates are (non-deterministically) generated. Then the rules test candidates and eliminate those who break them [4]. Optionally, a optimized solution can be found [9]. Logic programs in ASP can also be divided into two classes, problem encoding and problem instance. The problem encoding contains the specification and rules of a problem meaning it describes the problem while the problem instance is a concrete instance of the problem which we want to solve [4].

In order to understand the implementation, there are a few more rules and constraints which we need to illustrate. The following definitions are taken from [2].

Definition 9 (Choice rule)**.** A choice rule is a rule in form of $\{a_1, ..., a_m\} \leftarrow a_{m+1}, ..., a_n$ with $a_1, ..., a_n$ as literals stating that, if the body is true then either a_i or $\sim a_i$ is true for all $a_i, 1 \leq i \leq n$.

Definition 10 (Cardinality constraint)**.** A cardinality constraint is a rule in form of $L\ \{a_1, ..., a_m\}\ U$ that means at least L atoms in the choice rule have to be true and at most U atoms can be true. We also denote the cardinality rule with $U \geq \{a_1, ..., a_m\} \geq L$ or $L \leq \{a_1, ..., a_m\} \leq U$ in the implementation.

For predicates, the cardinality constraint can be extended, e.g., $L\ \{a(X) : p(X,Y)\}\ U$ meaning for every Y there are at least L and at most U values of X such that $a(X)$ is true. ASP also implements aggregates like *count, sum, maximum* and *minimum*, e.g., $\#count\{X : node(X)\} > 0$ meaning the number of nodes must be greater 0 [2]. ASP solvers are typically divided into two levels. First, the grounding step which grounds all rules in a given program P such that $P' \subseteq grnd(P)$ has the same answer sets as P. Secondly, Model Search which computes the answer sets of the grounded program P' [4].

To sum it up, the first phase of ASP solving process is a modelling phase in which the problem and its instances are modelled into logic programs, followed by the grounding phase which eliminates all first-order variables and outputs a propositional program. The solver then solves the propositional program and outputs a model as a solution [9].

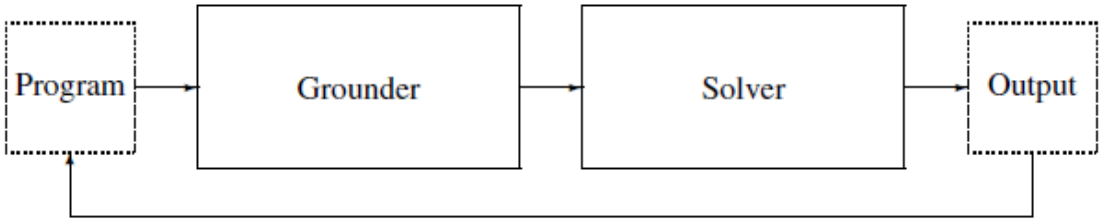


Figure 2: This picture describes the ASP solving process, source: [9]

In the following section, we take a look at clingo, Potassco's ASP Solver.

2.3 Potassco

For our results, we use the ASP tools clingo, gringo and clasp from Potassco.

Potassco's grounder gringo grounds the input and simplifies the rules by eliminating true components [9]. Besides grounding, gringo can also integrate the scripting languages lua and python. Moreover, the input language of gringo implements the optimization statements *#maximize* and *#minimze* to allow the search of an optimal answer set [7].

The actual solver, clasp, was originally designed and optimized for conflict-driven ASP solving [10]. Like other ASP solvers, clasp implements optimization via branch-and-bound search. First clasp searches for a model. Then clasp tries to solve the satisfiability problem of whether there is a model with a lower cost until this problem becomes unsatisfiable. This makes the last model found before the unsatisfiability was established the optimal model [8]. In addition to this, clasp version 3 can also use core guided optimization techniques usually used to solve Maximum Satisfiability (MaxSAT) problems. In unsatisfiability optimization, the solver tries to solve the problem and extracts an unsatisfiable core, if the problem is not satisfiable. A subset of clauses (rules) of the original problem, whose conjunction is still unsatisfiable, is an unsatisfiable core. All soft clauses of the extracted cores are then relaxed, so that the solver can arbitrarily satisfy one of them. A set of clauses of a given program, for which we want to find a subset maximizing the amount of satisfied clauses, is called soft clauses. The solver repeats this process until either a model is found or no more unsatisfiability cores can be extracted. In the first case, the first model found is the optimal model. In the second case, if no more unsatisfiability cores can be extracted then the problem is unsatisfiable. [1]. Due to clasp's multi-threaded architecture both optimization techniques can be combined. Furthermore, clasp also has the option of enumerating optimal models [8].

Clingo combines both gringo and clasp in a single system supporting all features and options of gringo and clasp [9].

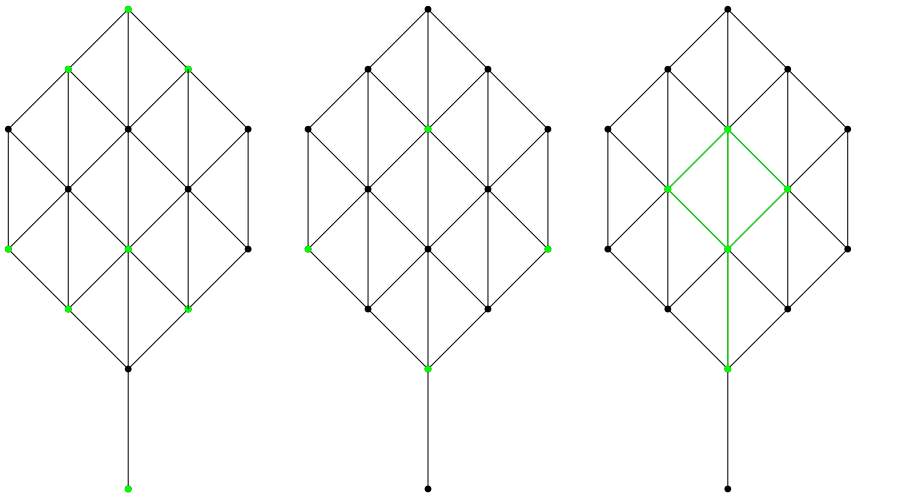
The following section illustrates the methods used to implement the simulation of a leaf's venation pattern.

3 Methods

We represent a leaf as a whole by using a simple undirected graph $G = (V, E)$. The nodes in V represent the leaf's cells, without distinguishing between cell type. The edges represent connections of cells linked to each other through plasmodesmata. As previously mentioned we use the dominating set and its variants to stepwise create venation patterns for a given leaf.
The dominating set binds non-vein-cells to vein cells since non-vein-cells need to be close to vein cells [14, p. 469].

Definition 11 (Dominating Set)**.** A dominating set for G is a subset $D \subseteq V$ such that all nodes $v \in V \setminus D$ are adjacent to a node $w \in D$. Examples are shown in figure 3.

From here on out D portrays the vein cells of a leaf for all coming definitions. Let k be a positive integer. Since veins are not random points in leaves, but connecting threads



(a) Dominating Set (b) Minimum Dominating Set (c) Minimum Connected Dominating Set

Figure 3: These are dominating set examples where the dominating nodes are marked with the color green.

running through leaves, the vein cells have to be connected. Otherwise, the water coming from the leafstalk would not be able to reach the vein-cells. We use the connected dominating set to not only have the non-vein-cells be dependent on vein-cells but also to ensure that the vein cells are connected.

Definition 12 (Connected Dominating Set)**.** A connected dominating set D is a dominating set for G such that for a subgraph induced by D there is a path from every node $v \in D$ to every other node $w \in D$.

Mesophyll cells can be up to three or four cells away from the vascular system [14, p. 469]. The k-hop dominating set allows for $k-1$ intermediate nodes between a dominating node and a node not in the dominating set.

Definition 13 (k-Hop Dominating Set)**.** A k-hop dominating set for G is a subset $D \subseteq V$ such that all nodes $v \in V \backslash D$ are at most k nodes away from a node $w \in D$.

A 1-hop dominating set is equivalent to a dominating set since 0 intermediate nodes are allowed for $k = 1$.

Definition 14 (k-transitive closure)**.** The graph $G' = (V, E')$ is a k-transitive closure of $G = (V, E)$ if for every path P of the length $2 \leq d \leq k$ in G the graph G' has an edge from every node in the path to every other in the path.

Remark. The above mentioned k-transitive closure definition only applies for undirected graphs. For $k = 1$ one would usually create an edge from a node to itself, but this case is disregarded which is why d must be at least 2.

Examples for k-transitive closures with different k can be seen in figure 4.

The next step is to combine the connectivity of vein cells and the mesophyll cells' ability to be a few cells away from the veins [14, p. 469]. This portrays the vascular system more

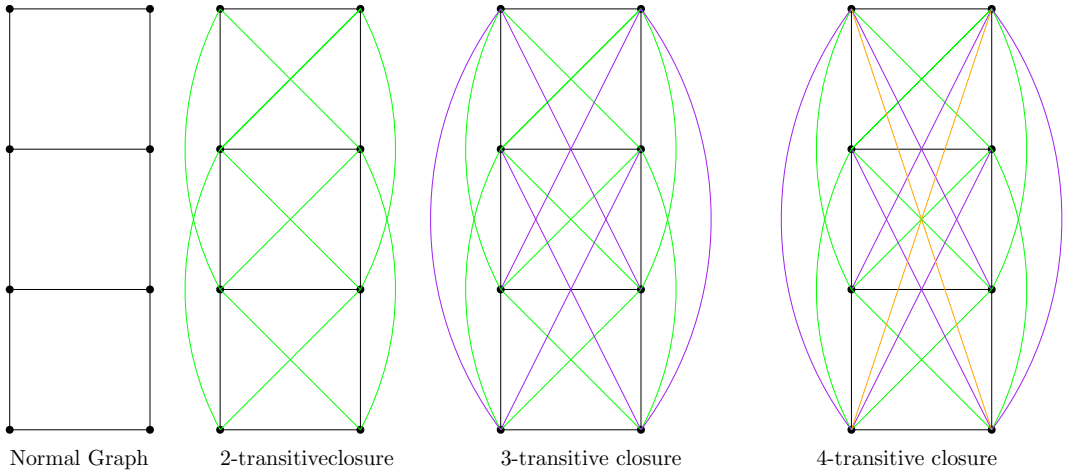


Figure 4: This figure is an example of k-transitive closures of a graph. The green edges display the additional edges for $k = 2$. The purple edges are the additional edges for $k = 3$ and the orange edges are for $k = 4$.

realistically by allowing both the connectivity of vein cells and a certain distance between vein cells and other types of cells.

Definition 15 (k-Hop Connected Dominating Set)**.** A k-hop connected dominating set is a k-hop dominating set for G such that for a subgraph induced by D there is a path from every node $v \in D$ to every other node $w \in D$.

To prevent solutions consisting of mirrored patterns, we define a type of canonical dominating set to eliminate symmetry. For symmetrical examples see figure 5.

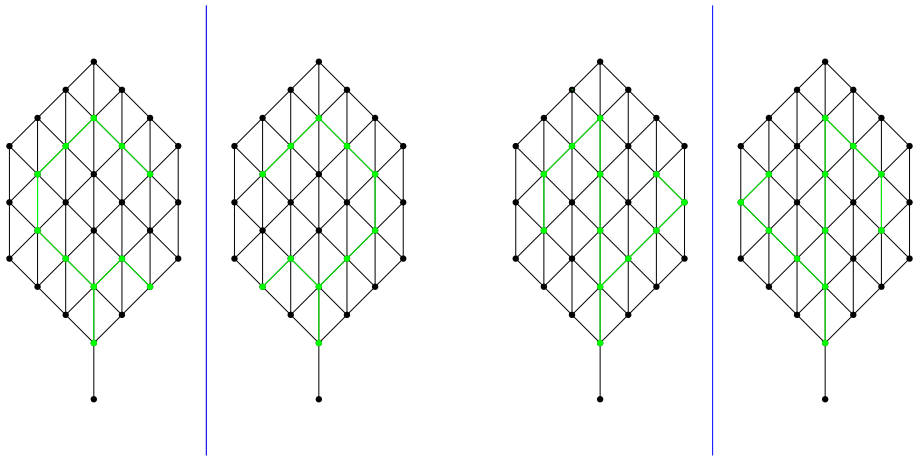


Figure 5: Mirrored patterns we want to avoid or minimize

Definition 16 (Canonical Connected Dominating Set)**.** A canonical connected dominating set D is a connected dominating set such that every path between two nodes $v_h, v_j \in D$ comply with the following rule: For every path between startnode v_h and endnode v_j let the intermediate nodes be v_i with $h < i < j$.

Definition 17 (Canonical k-Hop Connected Dominating Set)**.** A canonical k-hop connected dominating set D is a k-hop connected dominating set such that every path between two nodes $v_h, v_j \in D$ comply with the following rule: For every path between startnode v_h and endnode v_j let the intermediate nodes be v_i with $h < i < j$.

The vascular system is connected to the leafstalk which in turn connects the leaf to the rest of the plant [16]. To depict the vascular system correctly, all possible venation patterns should have one node representing the leafstalk which they are all linked to.

Definition 18 (Rooted k-Hop Connected Dominating Set)**.** A rooted k-hop connected dominating set is a k-hop connected dominating which is also connected to the root node $r \in D$.

Remark. Analogous definitions for other connected variants which are to be rooted.

In the next section, we will implement these methods in ASP to obtain suitable models for venation patterns.

4 Implementation

Now, we describe the implementation of the methods from the previous section in ASP. To do so we first introduce an alphabet for our logic program. Then we implement the variants of the dominating set in ASP pseudocode and elaborate on what the pseudocode does.

4.1 Alphabet

The alphabet used for the implementation is as follows:
We denote variables with upper case characters, e.g., X, and constants with lower case characters, e.g., x. In order to represent the graph as a problem instance we have the predicate symbols *node*, *edge*, *reach*, *connected*, *transitive* and *dom* which can form the atoms *node(_)*, *edge(_,_)*, *reach(_)*, *connected(_,_)*, *transitive(_,_)* and *dom(_)* with _ as placeholder for variables, constants or integers. A dot "." signifies the end of the rule. We denote not as the default negation with negation as failure.

The terms *node(_)* and *edge(_,_)* are used to represent the input graph. For every node in the graph, there is a fact $node(x)$ with x as a unique identifier for the node in the problem instance. For every edge, the fact $edge(x, y)$ represents the edge with x and y as the unique identifiers of the node in the graph sharing the edge. The example below shows a problem instance for an undirected rectangle graph.

```
node(1).    node(2).    node(3).    node(4).
edge(1,2).  edge(2,3).  edge(3,4).  edge(4,1).
edge(2,1).  edge(3,2).  edge(4,3).  edge(1,4).
```

Note that to represent an undirected edge in ASP, we have to use two directed edges.

The term $dom(\_)$, short for dominating, signifies that there is a dominating node with the identifier _. We explain the rest of the predicates as they appear.

We use the aggregate $\#count\{\_\}$ for counting and the aggregate $\#minimize\{\_\}$ for optimization. In addition to that, we introduce the aggregate $\#choose\{\_\}$ which was specifically created for the pseudocode and means that we pick exactly one element of a set that has to be true. We choose to introduce this aggregate in case a different grounder than gringo is used and the aggregate used in gringo might not exist in other grounders.

4.2 Minimum Dominating Set

The pseudocode in algorithm 1 is for the minimum dominating set implementation.

Algorithm 1: Minimum Dominating Set

```
1 {dom(X) : node(X)}.
2 #count{dom(X) : edge(X,Y)} ≥ 1 ← not dom(Y), node(Y).
3 #minimize{#count{dom(X)}}.
```

The choice rule in the first line states that a node is either dominating or not meaning every node could be in the dominating set. This way we generate subsets of nodes which are potential dominating sets. The second line declares that if a node is not in the dominating set then at least one of its edges has a dominating node on the other end. Additionally, the second line is equal to $\leftarrow not\ dominating(Y), node(Y), \{dominating(X) : edge(X,Y)\} < 1$ meaning there must not be the case where a node not in the dominating set does not share an edge with at least one dominating node. Intuitively, this means a node not in the dominating set has at least one dominating neighbour. The last line is the optimization statement specifying that we want to minimize the number of $dom(X)$ that are true meaning we want to minimize the number of dominating nodes needed for the dominating set.

4.3 Minimum Connected Dominating Set

The implementation for the minimum connected dominating set shown in algorithm 2 is an extension of the minimum dominating set shown in algorithm 1.

Algorithm 2: Minimum Connected Dominating Set

```
1 {dom(X) : node(X)}.
2 #count{dom(X) : edge(X,Y)} ≥ 1 ← not dom(Y), node(Y).
3 {connected(X,Y) : edge(X,Y), dom(Y)} ← dom(X).
4 {connected(X,Y) : edge(X,Y), dom(X)} ≤ 1 ← dom(Y).
5 reached(X) ← X = #choose{Y:dominating(Y)}.
6 reached(Y) ← connected(X,Y), reached(X).
7 ← dominating(Z), not reached(Z).
8 #minimize{#count{dom(X)}}.
```

The first two lines work the same way as for the minimum dominating set, as does the last line. The new lines from 3-7 add connectivity and is a barely modified version of the Hamilton cycle encoding taken from a Potassco tutorial [17]. Line 3 states that if X is dominating then all outgoing edges of $dom(X)$ connecting X with another dominating node Y are named *connected*. So does line 4 with the addition that we only allow at most one incoming edge per dominating node to limit the number of possible models. Moreover, we have now created a subgraph comprised of only dominating nodes and arbitrarily chosen edges between dominating nodes. In line 5 we choose an arbitrary startnode then in line 6 we try to recursively reach all other dominating nodes by marking dominating nodes connected to already reached nodes as reached. Line 7 declares that there must not be a dominating node that is not reached. If there is a dominating node that is not reached then generated set is not connected.

4.4 Minimum k-Hop Dominating Set

We do some preprocessing for the k-hop dominating set by performing a k-transitive closure on the original input graph and use the resulting graph G' that has additional edges as a new input graph. As a consequence, we can reuse the encoding of the minimum dominating set for the minimum k-hop dominating set. The algorithm used for the k-transitive closure is shown in algorithm 3:

Algorithm 3: k-transitive closure

```
1  tansitiveGraph := a copy of the inputGraph;
2  k := k-hop number;
3  for all nodes in inputGraph do
4      cnode = current node;
5      currentNeighbours := list of cnode's neighbours;
6      for i ← 0 to (k − 1) do
7          newNeighbours := empty list;
8          for all n in currentNeighbours do
9              newNeighbours := neighbours of n;
10             for j in newNeighbours do
11                 if newGraph does not have undirected edge cnode,j and cnode ≠ j then
12                     add an undirected edge between cnode,j;
13                 end
14             end
15             currentNeighbours := newNeighbours + currentNeighbours;
16         end
17     end
18 end
```

First, we create a copy of our input graph to which we will add the transitive edges. We choose a node, denoted as *cnode*, and save its neighbours in a list called *neighbours*. Then for every node in *neighbours* we take a look at its neighbours, called *newNeighbours*, and see if every node in *newNeighbours* shares an edge with our chosen node, *cnode*. If not

add an edge between them unless $cnode = j$ because we do not want an edge from *cnode* to itself. These are the nodes that are two nodes away from *cnode*. After adding edges between *cnode* and all nodes which were previously two nodes away, we add *newNeighbours* to *currentNeighbours*. This process is done $k - 1$ times. In doing so we walk all possible paths of the length k which has *cnode* as startnode. We do this for every node in the input graph to obtain a k-transitive closure.

4.5 Minimum k-Hop Connected Dominating Set

For the k-hop connected dominating set the preprocessing is slightly different. In addition to adding edges, we mark these additional edges with the predicate *transitive* to distinguish them from the original edges. If we add the additional edges with the predicate *edge* then the solver also uses these additional edges instead of only using the original edges to reach dominating nodes during the connectivity test. This can cause the subgraph induced by the dominating set to not be connected. As a result, we need to slightly modify the encoding of the connected dominating set solution.

Algorithm 4: Minimum k-Hop Connected Dominating Set

```
1 {dom(X) : node(X)}.
2 #count{dom(X) : edge(X,Y) ∨ dom(X) : transitive(X,Y)} ≥ 1 ← not dom(Y),
    node(Y).
3 {connected(X,Y) : edge(X,Y), dom(Y)} ← dom(X).
4 {connected(X,Y) : edge(X,Y), dom(X)} ≤ 1 ← dom(Y).
5 reached(X) ← X = #choose{Y:dominating(Y)}.
6 reached(Y) ← connected(X,Y), reached(X).
7 ← dominating(Z), not reached(Z).
8 #minimize{#count{dom(X)}}.
```

In the choice rule in line 2, we add $transitive(X, Y)$ to signify that a node not in the dominating set must share at least one original edge or one transitive edge with a node in the dominating set.

4.6 Canonical (k-Hop) Connected Dominating Set

Symmetry can cause a lot of problems in combinatorial optimization due to unnecessarily exploring search regions more than once [5]. To break some of the symmetry in our solution we use in Section 3 described canonical dominating set.

Algorithm 5: Minimum k-Hop Connected Dominating Set

```
1 {dom(X) : node(X)}.
2 #count{dom(X) : edge(X,Y)} ≥ 1 ← not dom(Y), node(Y).
3 {connected(X,Y) : edge(X,Y), dom(Y), X < Y} ← dom(X).
4 {connected(X,Y) : edge(X,Y), dom(X), X < Y} ≤ 1 ← dom(Y).
5 reached(X) ← X = #choose{Y:dominating(Y)}.
6 reached(Y) ← connected(X,Y), reached(X).
7 ← dominating(Z), not reached(Z).
8 #minimize{#count{dom(X)}}.
```

As described in Definition 16, we only accept edges that go from nodes with smaller identifying numbers to greater ones during the connectivity test. In order to do so, we add the condition $X < Y$ in the two choice rules in lines 3 and 4, so that $X < Y$ applies for every $connected(X, Y)$. This method has both advantages and disadvantages which are touch upon in the results and discussion sections. For the canonical k-hop connected dominating set, we need to add $\vee dom(X) : transitive(X, Y)$ in Line 2, like we did for the k-hop connected dominating set. Note that the way we number the node is decisive of what kind of symmetry we are breaking.

4.7 Rooted Connected Dominating Set Variants

There are several techniques to root the connected dominating set variants. For our implementation, we add $root(X)$ with X as the identifier of the chosen root node as a fact in the problem instance. After that the code needs to be slightly modified, e.g., a modification for the connected dominating set:

Algorithm 6: Minimum Connected Dominating Set

```
1 {dom(X) : node(X)}.
2 #count{dom(X) : edge(X,Y)} ≥ 1 ← not dom(Y), node(Y).
3 {connected(X,Y) : edge(X,Y), dom(Y)} ← dom(X).
4 {connected(X,Y) : edge(X,Y), dom(X)} ≤ 1 ← dom(Y).
5 reached(X) ← root(X).
6 reached(Y) ← connected(X,Y), reached(X).
7 ← dominating(Z), not reached(Z).
8 #minimize{#count{dom(X)}}.
```

Since we know the root node, we can start the connectivity search at the root node (line 5 in the example above). This saves some time since we do not have to arbitrarily choose a random node to start the search. The root node is also automatically a dominating node since $connected$ is only between nodes in the dominating set. In all the other connected variants the line "$reached(X) \leftarrow X = \#choose\{Y : dominating(Y)\}$" needs to be replace with "$reached(X) \leftarrow root(X)$" to root them.

4.8 Potassco's ASP tools

Potassco's ASP tools gringo, clasp and clingo are available on github[1]. The clingo package includes gringo and clasp. Recommended is installing Anaconda or Miniconda and download clingo with the following command in the Anacanda command:

```
conda install -c potassco clingo
```

This thesis uses the specific versions gringo 5.4.0, clasp 3.3.5 and clingo 5.4.0. Problem class and problem instance are written like normal text files and have the file extension ".lp". Let class.lp represent the problem class and instance.lp the problem instance. All experimental results used with clingo's parallel compete mode with T threads, the first command. The second command prints all optimal solutions. Both commands use the –quiet option meaning we do not want clingo to print the progess just the solutions. Printing the progress adds to the overall runtime.

```
clingo class.lp instance.lp --quiet --parallel-mode=T
clingo class.lp instance.lp --quiet --parallel-mode=T --opt-mode=optN
```

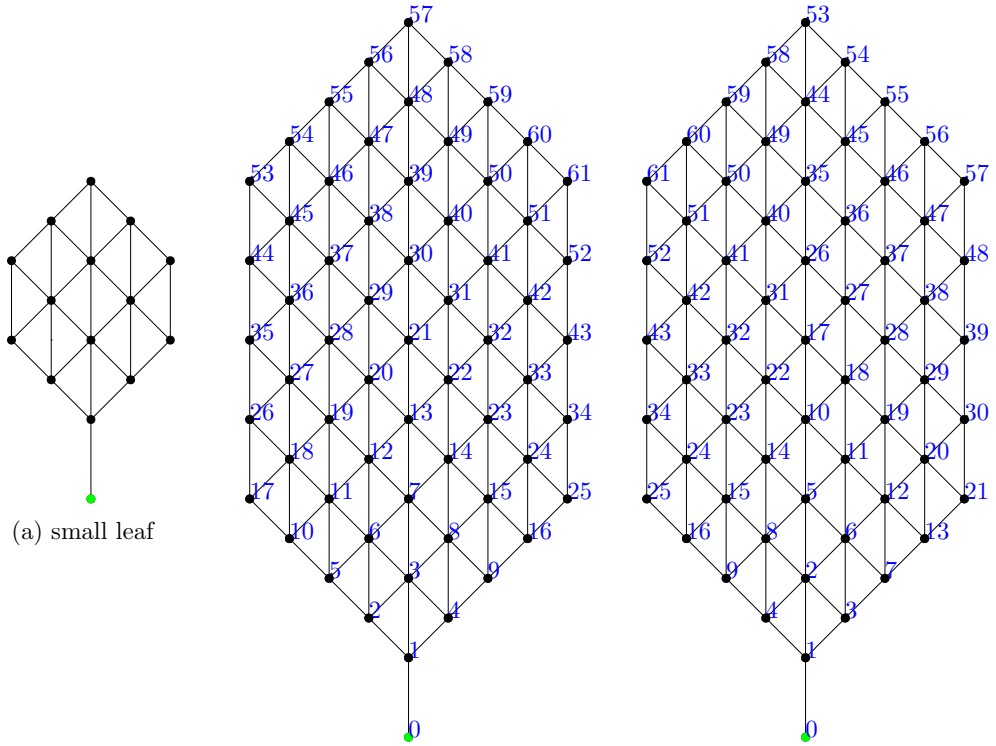
Next, we will take a look at our results.

5 Results

This section shows our runtime results for the different dominating set variants. First, we describe the graphs used for our tests:

Figure 6 shows the three smallest graphs used.

[1] https://github.com/potassco/clingo/releases/

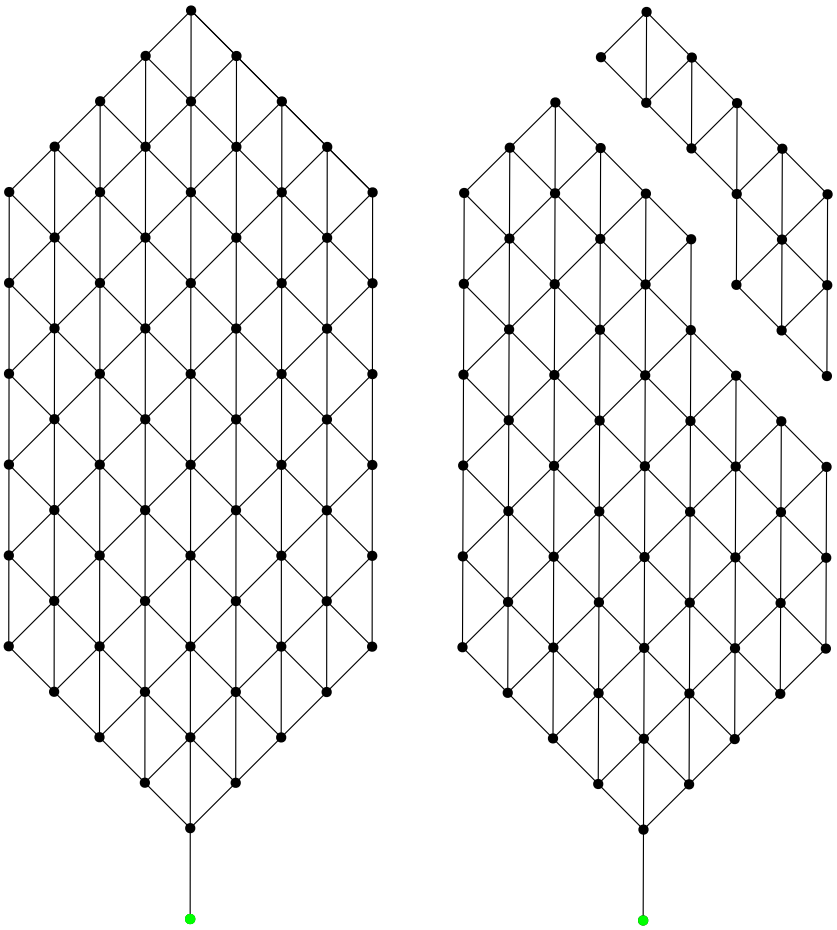


(a) small leaf (b) left-right leaf (c) middle leaf

Figure 6: The three smallest graphs used for the results

The first graph (a), called *small-leaf*, has 15 nodes and is only used since its small size makes it easy to see whether the results are correct or not. The other two graphs share the same structure, both having 62 nodes. Their difference lies in the numbering of their nodes. For the numbering, we categorize the nodes into levels determined by the shortest path between the node and the root node. For example, the root node has the level 0, if a node is 3 nodes away from the root node then it is on level 3. A node that is 3 nodes away has two intermediate nodes between it and the root node. In the graph (b), denoted as *left-right-leaf*, the root node has the number 0. The nodes on the next level are numbered from left to right with the next greater integer. This process is done for all levels. (c), named *middle-leaf*, numbers its nodes as follows. The node in the middle has the smallest integer of the level. Then we number the nodes on the right with the next greater integer. When there are no more nodes left on the right side we return to the middle and repeat the same process for the nodes on the left of the middle. We test these two isomorph graphs to show the impact that the numbering has on the canonical solutions. The graph structure of *left-right-leaf* and *middle-leaf* is taken from an example graph from [18]. The remaining graphs are inspired by *left-right-leaf* and *middle-leaf*.

The next two graphs, as shown in figure 7 are slightly bigger:

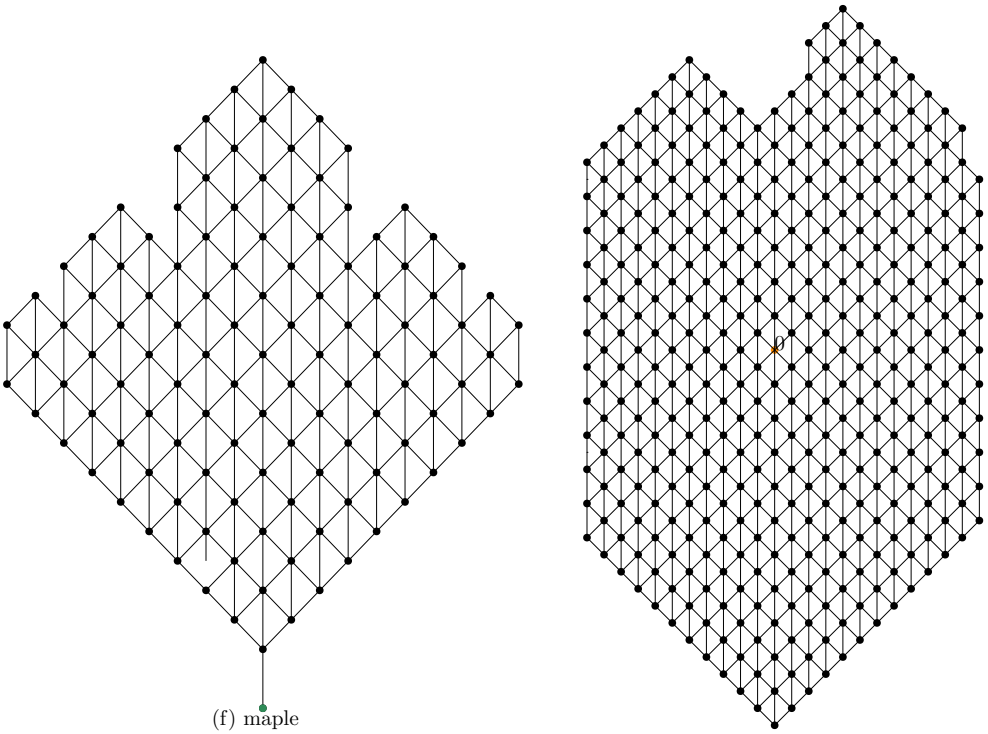


(d) bigger leaf (e) ripped leaf

Figure 7: A slightly bigger test graph and an unconnected one

The graph (d), called *bigger-leaf*, has 9 nodes more than *left-right-leaf*. It is also numbered from left to right. (e), denoted as *ripped-leaf*, has also 71 nodes, but has edges missing such that it is not connected since we also test whether and how fast the connected implementations detect that a graph is disconnected.

The last two graphs in figure 8 are the biggest graph used for testing. The graph (f) on the left, called *maple*, is inspired by a maple leaf while the one on the right called (g) *asymmetric* is inspired by an asymmetrical alocasia leaf which has the petiole connected on the underside of the leaf meaning the root is not on the edge of the graph, but rather somewhere closer to the middle. We number *maple*'s nodes from left to right. On the other hand, we number *asymmetric* also with the level system, but since the root is somewhere in the middle of the leaf (see the orange dot in figure 8) we start numbering each level at the top, 12 o'clock and then proceed clockwise. *Maple* has 118 nodes and *asymmetric* has 378.



(f) maple (g) asymmetric

Figure 8: The two biggest test graphs: (f) maple and (g) asymmetric

The result tables are structured as follows, in the first column we have the names of the test graphs used. The second column contains the number of threads used to solve the problem. Then we have a column named "Time single" which states the time needed to find a single optimal model followed by column "Time all" stating the time needed to find all optimal solutions. Both times are given in seconds. The next column shows the minimal amount of nodes needed for an optimal solution. Lastly, the "Models" column shows the number of optimal models that our implementation can find. Note that these are all optimal solutions, but some of them happen to reoccur due to how we implement finding connectivity. We discuss this more in the discussion section. If we cannot find a single optimal solution within 20 minutes and 24 cores or 4 cores for the non-connected solutions, we cancelled the solving process. This means the column for the optimal solution contains bounds for the optimum. The bounds are denoted with "[L;U]" which means there are no models with the lower bound L and there is at least one model found for the upper bound U. We also did not try to find all optimal solutions since we already failed at finding one meaning the number of optimal models is unknown. Optionally, if we can compare our ASP solution with an ILP solution then there is a column between name and threads which specifies whether the results are from the ASP or ILP solution. The ILP results are directly under the ASP results. For the comparison with ILP, we only use maximal 4 threads for both ASP and ILP. Note that there are no tests for finding all possible solutions in ILP since it is not built into the ILP solution given to us. The k-hop dominating set solution in ILP was given to us by Professor Gunnar Klau though we removed some lines which printed what edges were added since this seemed to add to

the programs runtime. Now we take a look at the results starting with the minimum dominating set in ASP and ILP as shown in table 1.

	Solution in	Threads	Time single	Time all	Optimum	Models
(a) small-leaf	ASP	1	0.000	0.000	4	47
	ILP	1	0.0	-	4	-
(b) left-right-leaf	ASP	2	0.007	0.018	12	1091
	ILP	4	0.016	-	12	-
(c) middle-leaf	ASP	2	0.007	0.024	12	1091
	ILP	4	0.016	-	12	-
(d) bigger-leaf	ASP	2	0.007	0.010	13	43
	ILP	4	0.031	-	13	-
(e) ripped-leaf	ASP	2	0.007	0.011	14	441
	ILP	4	0.016	-	14	-
(f) maple	ASP	2	0.031	0.016	20	1
	ILP	4	0.016s	-	20	-
(g) asymmetric	ASP	4	1252.886	-	[59;75]	-
	ILP	4	17.009	-	62	-

Table 1: Minimum Dominating Set Results for ASP and ILP

As we can see in table 1, our ASP solution can compete with the ILP solution for the dominating set for the first 6 test graphs. Both of them need less than 50 milliseconds for solving the minimum dominating set. With the exception of *small-leaf*, ILP even uses two threads more than ASP. The reason why we have a different thread number for the ILP and ASP results is because we want to compare the fastest way of solving for both solutions. During our tests more threads for small inputs sometimes required more time in ASP than less threads for the same input. Also, the ILP solution decides for itself how many threads it wants to use. The ASP solution is faster than the ILP solution for the first five graphs although since the differences are only a few milliseconds we can overlook them. However, ILP is faster than ASP for bigger input graphs since ILP can solve *asymmetric* within 17 seconds while ASP cannot solve *asymmetric* within 20 minutes. For the 2-hop dominating set, the result are slightly different as shown in table 2.

As before ASP seems to require less time than ILP for all graphs except *small-leaf*, *ripped-leaf* and *asymmetric*, but again the differences are only a few milliseconds. However, the 2-hop dominating set was able to solve *asymmetric* in less than five minutes although the number of optimal solutions is not solvable within 20 minutes. The ILP solution is much faster requiring less than a second. The ILP solution can also solve ripped-leaf with only one thread.

	Solution in	Threads	Time single	Time all	Optimum	Models
(a) small-leaf	ASP	1	0.000	0.000	1	1
	ILP	1	0.0	-	1	-
(b) left-right-leaf	ASP	2	0.012	0.014	5	6
	ILP	4	0.062	-	5	-
(c) middle-leaf	ASP	2	0.014	0.013	5	6
	ILP	4	0.016	-	5	-
(d) bigger-leaf	ASP	2	0.015	0.021	6	1177
	ILP	4	0.016	-	6	-
(e) ripped-leaf	ASP	2	0.013	0.057	7	20500
	ILP	1	0.016	-	7	-
(f) maple	ASP	2	0.016	0.031	9	806
	ILP	4	0.031	-	9	-
(g) asymmetric	ASP	4	264.726	1231.504	25	2020+
	ILP	4	0.577	-	25	-

Table 2: Minimum 2-hop Dominating Set Results for ASP and ILP

	Solution in	Threads	Time single	Time all	Optimum	Models
(a) small-leaf	ASP	1	0.000	0.000	1	7
	ILP	1	0.0	-	1	-
(b) left-right-leaf	ASP	2	0.019	0.021	3	18
	ILP	4	0.016	-	3	-
(c) middle-leaf	ASP	2	0.020	0.020	3	18
	ILP	4	0.016	-	3	-
(d) bigger-leaf	ASP	2	0.023	0.039	4	2659
	ILP	1	0.016	-	4	-
(e) ripped-leaf	ASP	2	0.019	0.020	4	81
	ILP	1	0.0	-	4	-
(f) maple	ASP	2	0.031	0.047	5	569
	ILP	4	0.0	-	5	-
(g) asymmetric	ASP	4	1320.739	-	[13;17]	Unknown
	ILP	4	0.188	-	14	-

Table 3: Minimum 3-hop Dominating Set Results for ASP and ILP

For the 3-hop dominating set results in table 3, ILP is faster than ASP for all test graphs except *small-leaf* where they have the same runtime. Additionally, the 3-hop dominating set solution is not able to solve *asymmetric* within 20 minutes in contrast to the 2-hop dominating set. The ILP solution is able to solve the 3-hop dominating set in less than a second.

Next, we take a look at the connected variants beginning with the minimum connected dominating set in table 4. The first row of table 4 shows that *small-leaf* is quite easy to solve needing less than 10 milliseconds for finding one and finding all solutions. However, our implementation needs more than twice as much time for solving *middle-leaf* than

	Threads	Time single	Time all	Optimum	Models
(a) small-leaf	2	0.006	0.005	5	24
(b) left-right-leaf	4	18.445	141.970	21	20088
	4	*54.968	-	21	-
(c) middle-leaf	4	313.022	678.339	21	20088
	4	*156.962	-	21	-
(d) bigger-leaf	24	561.827	4135.029	24	69482
(e) ripped-leaf	1	0.012	-	UNSAT	0
(f) maple	24	1202.277	-	[33;40]	Unknown
(g) asymmetric	24	1379.182	-	[28;145]	Unknown

Table 4: Minimum Connected Dominating Set Results

it does for *left-right-leaf* despite both having the same structure and results in optimum and number of models. This is neither an outlier nor a result of limiting the number of incoming edges during the connectivity test in our implementation since it happens repeatedly during testing. The results marked with the symbol * show the results without limiting the number of incoming edges. The unconstrained results have no results for finding all optimal models since the program did not finish after 3 hours for *middle-leaf*. The table also shows that our connected dominating set implementation does not scale well since it needs 24 threads and over 500 seconds for solving *bigger-leaf* which has only 9 nodes more than *left-right-leaf*. Finding all optimal solutions for *bigger-leaf* even takes over 1 hour. This is the only time we did not cancel around the 20 minute mark. On the other hand, it almost immediately recognizes that *ripped-leaf* is not solvable on account of being a disconnected graph without needing more than one thread. This is true for all remaining tables, so we do not mention ripped-leaf again. Our implementation neither comes close to solving *maple* nor *asymmetric* within 20 minutes even with 24 threads.

In addition to this, the number of optimal models is incredibly high which is like previously mentioned due to how we implement the connectivity test. A method to reduce both the time and the number of models found is with the help of the canonical connected dominating set solution.

	Threads	Time single	Time all	Optimum	Models
(a) small-leaf	2	0.004	0.004	5	7
(b) left-right-leaf	4	0.047	0.197	21	117
(c) middle-leaf	4	0.103	0.424	22	1152
(d) bigger-leaf	4	0.112	0.221	24	156
(e) ripped-leaf	1	0.011	-	UNSAT	0
(f) maple	4	6.807	19.777	41	374848
(g) asymmetric	24	1200.975	-	[72;134]	Unknown

Table 5: Minimum Canonical Connected Dominating Set Results

As shown in table 5 the canonical connected dominating set is significantly faster when compared to the results in table 4. Both *left-right-leaf* and *middle-leaf*'s runtime was reduced to under 1 second, but *middle-leaf* still needs about twice as long as *left-right-leaf*. Furthermore, *bigger-leaf* can be solved in less than a second with only 4 threads. *Maple*

is also solvable with 4 threads needing only 7 seconds for a single optimal solution and around 20 seconds for all optimal solutions. *Asymmetric* is still not solvable for a single solution within 20 minutes, but the canonical connected solution finds better bounds faster than the normal connected solution. *Small-leaf* has no significant change in runtime, but the canonical connected dominating set reduces the number of models from 24 to 7. This seems to be the case for all test leaves since the number of models from *middle-leaf* and *left-right-leaf* also went down from 20088 to 1152 and 117 respectively. Moreover, *bigger-leaf*'s optimal model count lessens from 69482 to 156. These are good results, but comparing table 5 with table 4 we see that the optimums are not correct, e.g., the optimal solution of *middle-leaf* is 22 for the canonical connected dominating set instead of the correct 21. Also, table 5 shows that *maple*'s optimal solution has 41 nodes even though the normal connected dominating set implementation found a solution with 40 nodes as shown in table 4. This means, due to its strong dependency on how the graph is numbered, the canonical connected dominating set solutions are not exact, but approximations of minimum solutions.

Now, we take a look at the results of the k-hop connected for $k = 2$ in table 6 and $k = 3$ in table 7.

	Threads	Time single	Time all	Optimum	Models
(a) small-leaf	2	0.006	0.006	1	1
(b) left-right-leaf	4	0.621	2.111	12	118
(c) middle-leaf	4	1.054	3.698	12	118
(e) ripped-leaf	1	0.018	-	UNSAT	0
(d) bigger-leaf	4	17.451	29.086	13	4
(f) maple	24	1205.141	-	[19;24]	Unknown
(g) asymmetric	24	1356.188	-	[20;95]	Unknown

Table 6: Minimum 2-hop Connected Dominating Set Results

The 2-hop connected dominating set implementation solves the graphs quite fast without the need for canonical symmetry breaking. While our solution cannot solve *maple* and *asymmetric* within 20 minutes with 24 threads, the bounds for the optimal solution are much better than compared to table 4. *Small-leaf*, *left-right-leaf* and *middle-leaf* only need up to a second to find one optimal solution and up to four seconds to find all optimal solutions. Whereas the connected dominating set solution struggles to solve *bigger-leaf* with 24 cores, the 2-hop connected dominating set solution only needs up to 30 seconds fo find all optimal solutions with only 4 cores. The 3-hop connected dominating set solution solves the test graphs even faster as seen in table 7.

While the 3-hop connected solution is still not able to solve *maple* and *asymmetric* within 20 minutes, its bounds are better than those of the 2-hop connected solution in 6. *Maple*'s bounds have only a difference of 3 rather then 5 and *asymmetric*'s bounds have a difference of 49 rather than 75. The 3-hop solution reduces the solving time of *small-leaf*, *left-right-leaf* and *middle-leaf* to less than half a second even when searching for all optimal solutions. For *bigger-leaf*, it needs less than a second for finding an optimal solution and a few seconds to find all optimal solutions with 4 cores. However, it is odd that *middle-leaf* and *left-right-leaf* have a different number of optimal solutions in table 7 even

	Threads	Time single	Time all	Optimum	Models
(a) small-leaf	2	0.007	0.006	1	7
(b) left-right-leaf	4	0.099	0.171	7	124
(c) middle-leaf	4	0.140	0.205	7	128
(d) bigger-leaf	4	0.828	3.292	8	74
(e) ripped-leaf	1	0.021	-	UNSAT	0
(f) maple	24	1413.329	-	[15;18]	Unknown
(g) asymmetric	24	1480.644	-	[14;63]	Unknown

Table 7: Minimum 3-hop Connected Dominating Set Results

though they have the same structure and the same number of optimal models in table 6.

For the canonical k-hop dominating set, rooted k-hop dominating set and rooted canonical k-hop dominating set we use $k = 2$ for our results. The canonical 2-hop connected

	Threads	Time single	Time all	Optimum	Models
(a) small-leaf	2	0.005	0.005	1	1
(b) left-right-leaf	4	0.046	0.088	12	13
(c) middle-leaf	4	0.026	0.049	13	70
(d) bigger-leaf	4	0.084	0.101	13	1
(e) ripped-leaf	1	0.015	-	UNSAT	0
(f) maple	4	3.796	17.017	25	9784
(g) asymmetric	24	1273.791	-	[44;81]	Unknown

Table 8: Minimum Canonical 2-hop Connected Dominating Set Results

dominating set, as shown in table 8, has an even lower runtime than the 2-hop connected dominating set solution in table 6. While not able to solve *asymmetric*, our result has better bounds than those in table 6 with a difference from 37 instead of 75. However, like the canonical connected dominating set solution, it is also an approximation. *Middle-leaf* has an optimum of 13 even though there is a non-canonical solution with 12 nodes. *Maple* is also solvable, but solution with fewer nodes can be found by the non-canonical solution as shown in table 6 which finds a solution with 24 nodes.

Now we take a look at the rooted solution results starting with the non-canonical one. Compared to the not rooted 2-hop connected dominating set results in table 6, the rooted

	Threads	Time single	Time all	Optimum	Models
(a) small-leaf	2	0.006	0.006	3	2
(b) left-right-leaf	4	0.419	1.595	14	132
(c) middle-leaf	4	0.331	2.037	14	132
(d) bigger-leaf	4	0.677	1.538	15	4
(e) ripped-leaf	1	0.012	-	UNSAT	0
(f) maple	24	1262.068	-	[25;27]	Unknown
(g) asymmetric	24	1187.724	-	[21;89]	Unknown

Table 9: Minimum Rooted 2-hop Connected Dominating Set Results

solution is slightly faster, reducing the runtime of *left-right-leaf* and *middle-leaf* to less than half a second to find a single solution and only needing about 2 seconds to find all optimal solutions. While these improvement are hardly noticeable, the improvement for *bigger-leaf*, *maple* and *asymmetric* are much more apparent. Finding all optimal solutions for *bigger-leaf* requires less than two seconds in comparison to the 30 seconds in table 6 and finding a single optimal solution requires less than a second instead of 17 seconds. Moreover, both *maple* and *asymmetric*, while still not solvable within 20 minutes, have better bounds than before. However, this comparison is to be taken with a grain of salt due to the rooted version needing the root to be in the dominating set meaning there might be more nodes in the rooted 2-hop connected dominating set than in the normal connected version, which lowers the runtime. The canonical rooted 2-hop dominating set results as shown in table 10 are also slightly faster than the non-canonical version as shown in table 9. It also can solve *maple* with only 4 threads instead of 24 and has even better bounds for *asymmetric*. However, like all other canonical versions before, the minimum canonical rooted 2-hop dominating set is also an approximation as seen when comparing the optimums of the canonical and non-canonical versions.

	Threads	Time single	Time all	Optimum	Models
(a) small-leaf	2	0.005	0.005	3	1
(b) left-right-leaf	4	0.013	0.015	14	13
(c) middle-leaf	4	0.015	0.023	15	70
(d) bigger-leaf	4	0.016	0.017	15	1
(e) ripped-leaf	1	0.011	-	UNSAT	0
(f) maple	4	0.187	0.307s	27	10272
(g) asymmetric	24	1365.163	-	[53;83]	Unknown

Table 10: Minimum Canonical Rooted 2-hop Connected Dominating Set Results

In conclusion, ASP can compete with ILP for the (k-hop) dominating set with small input graphs. However, ILP can easily solve bigger input graphs that ASP struggles with. For the (k-hop) connected variants ASP can solve the problem faster with greater k. While the canonical connected versions can solve problems that the non-canonical versions struggle with, they are not exact thus not completely reliable.

6 Discussion

The venation patterns created by our implementation are not perfect models for an optimal leaf venation since they only present the number of cells which have to be replaced with vein cells to supply the entire leaf and maximize the amount of photosynthesis performing cells. The model completely disregards the vein hierarchy and among others that the vascular system structure has also different tasks like protecting against damage [16]. Our model also disregards that plants try to minimize both the total branch/vein length and the transport distance for nutrients in their architecture [3].

As seen in the results our ASP implementation is impractical for larger input graphs. Especially when considering our comparison with the ILP solution for the simple k-hop variants. The normal connected solution already struggles with *middle-leaf* needing over

5 minutes for solving even though middle leaf only has 62 nodes. Our canonical solutions are faster, but not exact and are too dependent on how we number the nodes of a graph. This means it is possible for the solver to determine that solving the canonical (k-hop) dominating set for a connected graph is unsatisfiable because of the way the nodes are numbered. Furthermore, while the left-right numbering for the canonical solution is more correct, it creates a left heavy venation pattern for *left-right-leaf* and while *middle-leaf* is an approximation and generally requires more solving time, it comes closer to the kind of venation pattern we want to have, as figure 9 shows.

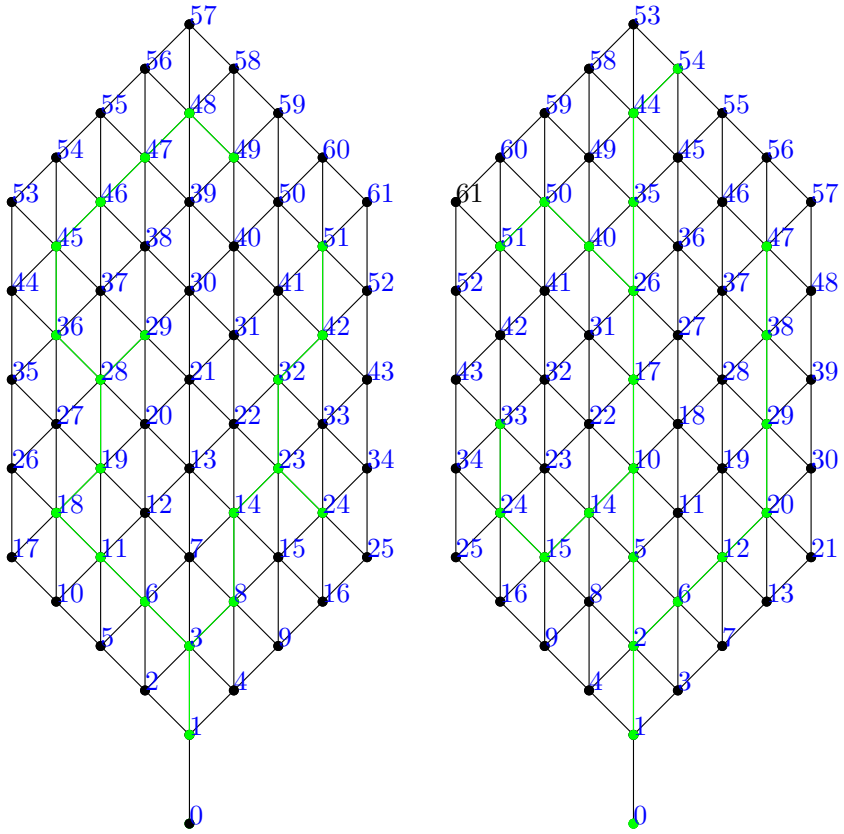


Figure 9: A Disadvantage of the canonical method

Moreover, the non-canonical connected version seems to be dependent on the node numbering since *left-right-leaf* requires less solving time than *middle-leaf* even though they only differ in node numbering.

Our k-hop implementations generally seem to require less time with greater k, but since mesophyll cells can only be a few cells away from the vascular system [14, p. 469] we cannot simply increase k to find a faster solution. Especially if we want to know the solution for a certain k.

As previously mentioned, the number of optimal solutions are extremely high but contain recurrences due to the way we implement our connectivity test. Since a choice rule chooses which edge between two dominating nodes is used for the connectivity test the

solver can output two solutions with the same dominating set, but two different sets of *connected* edges. Limiting the number of incoming edges reduces the number of recurrences, but does not eliminate all of them. It is possible to declare all edges between dominating nodes as *connected* nodes, but this method required more time during testing.

As mentioned in the results section, there are different ways of rooting the connected dominating set variants. One way to root the connected dominating set is to simply add $dominating(x)$. with x as identifier of our chosen root node as a fact in our problem instance. However, this does not remove the arbitrarily choosing of startnodes for the connectivity search. Another way is to declare that for every problem instance, the root node has always a certain identifier x, e.g., $x = 0$. This might become problematic though if one wants to choose the numbering of oneself for some kind of other type symmetry breaking, which is why we choose to do the rooting with the predicate $root(x)$.

During the comparison with ILP, we choose to only use two threads for running the ASP programs while ILP used different numbers of threads for different graphs and different ks. We want to compare ASP and ILP when they are at their fastest. Additionally, solving in ASP sometimes requires more time when using many threads for small inputs. Our comparison of ILP and ASP results might be unfair since we did not use the same amount of threads for both solutions, but both were under the same conditions with the exception that we chose how many threads our ASP solution used while the ILP solution chooses for itself how many threads it needs. Ultimately, ILP is the better solution for the (k-hop) dominating set since it can solve larger inputs in contrast to ASP.

7 Conclusions

Our model finds the minimal amount of vein cells required to supply the entire leaf, but it still disregards a lot about the structure of a leaf.

In addition to this, our ASP implementation, while functioning, is impractical, especially when compared to ILP. Moreover, the canonical method used to reduce the runtime is just an approximation with a lot of disadvantages seeing as it can exclude the types of patterns we actually want as results.

The next step for the ASP solution is to try and find a better encoding for our model since the encoding of an implementation strongly influences its runtime as mentioned in the discussion. Moreover, finding an efficient way to number graph nodes is also another goal since the numbering has a great influence on the runtime and the optimum as shown in our results. Additionally, one could expand the model by minimizing the branch lengths of the veins to make the model even more realistic and to see if more constraints can reduce the solving time. Also, trying out different solvers or Potassco tools are also an option.

On the other hand, implementing the complete model in ILP should be the current focus since we do not know for certain how well it performs for the complete model. After comparing the implementation of our complete model in ILP with our ASP solution we can decide which solution we want to expand on to make the model more authentic.

8 Code

The code for our implementation and our example graphs can be found on https://gitlab.cs.uni-duesseldorf.de/van.mantgem/dominating-set-using-asp. The link was last visited: 04.02.2020.

9 Acknowledgements

I would like to express my gratitude to Prof. Gunnar Klau for giving me the opportunity to write this bachelor thesis. I would also like to thank Eline van Mantgem for giving me advice, answering my questions and for the weekly meetings.

References

[1] B. Andres, B. Kaufmann, O. Matheis, and T. Schaub. Unsatisfiability-based optimization in clasp. In *Technical Communications of the 28th International Conference on Logic Programming (ICLP'12)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2012.

[2] G. Brewka, T. Eiter, and M. Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011.

[3] A. Conn, U. V. Pedmale, J. Chory, and S. Navlakha. High-resolution laser scanning reveals plant architectures that reflect universal network design principles. *Cell systems*, 5(1):53–62, 2017.

[4] T. Eiter, G. Ianni, and T. Krennwallner. Answer set programming: A primer. In *Reasoning Web International Summer School*, pages 40–110. Springer, 2009.

[5] T. Fahle, S. Schamberger, and M. Sellmann. Symmetry breaking. In *International Conference on Principles and Practice of Constraint Programming*, pages 93–107. Springer, 2001.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. Freeman, 1979.

[7] M. Gebser, R. Kaminski, B. Kaufmann, M. Lindauer, M. Ostrowski, J. Romero, T. Schaub, and S. Thiele. Potassco user guide, version 2.2.0. *URL: https://github.com/potassco/guide/releases/*, january 2019.

[8] M. Gebser, R. Kaminski, B. Kaufmann, J. Romero, and T. Schaub. Progress in clasp series 3. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 368–383. Springer, 2015.

[9] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. Schneider. Potassco: The potsdam answer set solving collection. *Ai Communications*, 24(2):107–124, 2011.

[10] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. In *International Conference on Logic Programming and Nonmonotonic Reasoning*, pages 260–265. Springer, 2007.

[11] S. Hölldobler and L. Schweizer. Answer set programming and clasp-a tutorial. In *YSIP*, pages 77–95, 2014.

[12] V. Lifschitz. *Answer set programming*. Springer International Publishing, 2019.

[13] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1987.

[14] P. S. Nobel. *Physicochemical and Environmental Plant Physiology*. Elsevier, 4. edition, 2009.

[15] J. M. Posada, R. Sievänen, C. Messier, J. Perttunen, E. Nikinmaa, and M. J. Lechowicz. Contributions of leaf photosynthetic capacity, leaf angle and self-shading to the maximization of net photosynthesis in acer saccharum: a modelling assessment. *Annals of botany*, 110(3):731–741, 2012.

[16] L. Sack and C. Scoffoni. Leaf venation: structure, function, development, evolution, ecology and applications in the past, present and future. *New phytologist*, 198(4):983–1000, 2013.

[17] T. Schraub. Optimization in clasp 3, part one. https://www.youtube.com/watch?v=23KyrdzHVOA.

[18] M. van Aalst. Optimality principles in leaf venation patterns. Master's thesis, Heinrich Heine University Düsseldorf, October 2019.

List of Figures

List of Tables