Technology Diffusion on Spiders

by

Charupriya Sharma

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Combinatorics and Optimization

Waterloo, Ontario, Canada, 2017

© Charupriya Sharma 2017

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

There has been significant research about cascade effects that occur when information is spread through a network. Most models of such cascade effects are highly-localised, which means that they assume a node's behaviour in a social network is influenced only by its immediate neighbours. Goldberg and Liu (2013) argued that such models may not represent the information diffusion process in several real world scenarios accurately. They proposed the *technology diffusion* model, where a node's decision to accept a new piece of information can be influenced by remote nodes it can communicate with.

Formally, the technology diffusion model is defined as follows : given an undirected graph, $G = (V,E)$, and a threshold function $\theta : V \to \{2,...,|V|\}$ on all nodes $v \in V$, a node v is considered *activated* whenever it becomes adjacent to a connected component containing at least $\theta(v)$ active nodes. The objective, referred to as *seed minimization*, is to find a minimum cardinality set of initially active nodes, called the *seedset*, that triggers a cascade that activates the entire graph.

Efsandiari et al. (2015) showed that technology diffusion is NP-hard even on *spiders* (trees in which at most one vertex, called the root, has degree larger than 2). In this thesis, we show that the problem is polynomial-time solvable on spiders with a constant number of legs, i.e., spiders where the root has a constant degree. We also show that, in this setting, there is a linear program formulation for the problem of polynomial size. Finally, we initiate the study of the *influence maximization* version of technology diffusion, which seeks for a seedset of a fixed size k which maximizes the number of vertices eventually activated. Influence maximization problems of this kind have been studied extensively in localized diffusion models, and therefore it is natural to investigate the same question for technology diffusion models. We show that this problem is also solvable in polynomial-time on spiders with a constant number of legs.

Acknowledgements

I thank my supervisor Laura Sanità for her guidance. This work could not be completed without her support.

Table of Contents

List of Tables vii

List of Figures viii

1 Introduction 1

2 Related Works 6

2.1 Localised Diffusion Models 6

2.1.1 Independent Cascade (IC) Model 7

2.1.2 Linear Threshold (LT) Model 8

2.2 Technology Diffusion 14

2.2.1 Seed Minimization in TD 15

3 Technology Diffusion on Spiders 16

3.1 Connection with Scheduling 17

3.2 Dynamic Programming for TD on Weighted Paths and Spiders 19

3.2.1 TD on Weighted Paths 20

3.2.2 Extension to Weighted Spiders 25

4 Polyhedral Characterization for Dynamic Programming Models on TD 35

4.1 The Paradigm 35

4.2 Polyhedral Characterization 37

4.3 LP formulation for TD on Paths . . . 41
4.3.1 Example . . . 45
4.4 LP formulation for TD on Spiders . . . 46

5 Maximization of Diffusion under Constraints on Seeds 55
5.1 Greedy Algorithm for maxTD . . . 56
5.2 Dynamic Programming for maxTD on a Path . . . 57
5.2.1 Extension to Spiders . . . 63

6 Conclusion 71

References 72

List of Tables

4.1 Arc Costs for Figure 4.3. 48
4.2 Dual Variables for Figure 4.3. 48

List of Figures

3.1 A spider graph with 5 legs. 16
3.2 Converting a pDLS instance into a TD instance. 18
3.3 A TD instance on a path of 7 nodes. 19
3.5 TD instance on a path. 20
3.6 A spider graph with 5 legs. 26

4.1 A TD instance on a path of three nodes. 45
4.3 Example for Figure 4.1. 47

5.1 A maxTD instance with $k = 2$ on a path of $m+11$ nodes. The node labels indicate thresholds. 56

Chapter 1

Introduction

Social networks have been studied across a number of domains for several decades. With the recent growth of companies like Facebook and Twitter, research on social networks is now supplemented with real-world data. It has lead to exciting applications of this research, and has also lead to the formulation of several new lines of enquiry. One such area of research is the study of cascade effects. Cascade effects model the dynamics of spreading information in a social network. This information could be a rumour, a virus or a new technology. Given a social network, it is often useful to find out how rapidly, and how much a newly introduced piece of information is spread throughout the network.

One strong motivation for studying such cascade models is marketing. Traditional mass marketing methods consist of marketing a product to all potential customers. This may not always be the most optimal strategy, as an individual's decision to buy a product could be influenced by their friends and co-workers. Leveraging these social networks is often more profitable than simple mass marketing. Often, companies go for *directed marketing* campaigns. Directed marketing takes several variables about potential customers into account, like previous purchases, demographic factors, and the people they interact with regularly, to narrow down a set of potential customers, and markets products to just these people. The idea is that some 'influential' people will be encouraged to adopt a product (for instance, by giving them free samples) and they will influence their acquaintances to buy that product too, and the product will thus be promoted in the network through a word-of-mouth campaign. If successful, such *viral marketing* campaigns are more profitable and cost-effective than mass marketing [Piatetsky-Shapiro and Masand (1999)] campaigns.

Several startups use social media data to identify influential people. PeerIndex (http://peerindex.com/) and Influencer50 (http://influencer50.com), for example, assign 'influ-

ence scores' to users based on their social media usage, and recommend those with high scores to companies. They also identify key journalists associated with a topic. The companies use this data to plan viral marketing campaigns. Users are encouraged to improve their score, as it would increase their likelihood of getting offers from such companies.

Viral marketing has also been getting attention in computer science and mathematics research. Domingos and Richardson (2001) modelled a social network as a Markov random field [Besag (1974), Jain and Chellappa (1993), Kindermann and Snell (1980)], where each individual's probability of purchasing some product is a function of both the utility of the product for the individual and the influence of other customers. Employing heuristic data mining methods, the authors used their framework to optimize the choice of a set of customers, called *seedset*, to market to. Kempe et al. (2003) later framed the question of identifying this seedset as a discrete optimization problem. They focused on two cascade effect models: the *linear threshold* [Granovetter (1978), Schelling (2006)], and *independent cascade* [Goldenberg et al. (2001a)] models. In these models, an individual's desire to buy a product is a function of the behaviour of their immediate neighbours in a social network. The authors posed the problem of finding a seedset of k initially active vertices(seeds) that cause all vertices to eventually buy the product, where k is a given parameter. Kempe et al. (2003) showed that this problem is NP-hard on independent cascade and the linear threshold model with random thresholds, and presented a greedy hill climbing algorithm that returns a seedset that activates a $(1-1/e)$ fraction of the number of nodes activated by an optimal algorithm.

While there have been extensive studies of cascade effects in social networks, most of these studies use *localized* models, like the two mentioned above, where a node's behaviour is a function of just its neighbours' behaviour. Goldberg and Liu (2013) argued that such highly localized settings may not model the diffusion process in several real-world scenarios accurately. For example, in communication networks, where a node using a new technology can potentially interact with remote nodes using older technologies, it would be beneficial for the node if its remote connection would upgrade to the newer technology it is using. Goldberg and Liu (2013) defined a new model for studying cascade effects, called *technology diffusion*, which is the subject of this thesis. In this model, nodes are impacted if they are adjacent to a connected component of impacted nodes of a sufficient size. We give a more formal description of Goldberg and Liu's model below.

Technology Diffusion (TD). Consider a simple, undirected graph, $G=(V,E)$, that represents our population and their interactions. We wish to model the cascade effects of introducing a new technology in a network. Each node is in one of two states, *active* or *inactive*. All nodes are set to be initially inactive (modelling a population using an older version of a technology). Once a node is active, it can never become inactive again.

Nodes activate (upgrade to using the newly introduced technology) when they obtain sufficient *utility* from the new technology. A *threshold function*, $\theta(u)$, is associated with each node u and it used to determine how large u's utility should be before it activates. In our technology upgrade example, the threshold function could model the cost of upgrading a technology for a node. We define a node u's utility to depend on the size (i.e., the number of vertices) of the connected component containing u and the active nodes reachable by u in G. TD is defined as the following optimization problem:

Technology Diffusion (TD)

INPUT : A simple, undirected graph, $G = (V,E)$, and a threshold function $\theta : V \to \{2,...,|V|\}$.

ACTIVATION PROCESS : We start with a set of active nodes at time step 0, $Y \subseteq V$, called a seedset. The activation process proceeds in discrete time steps. A node u activates at time t if the connected component containing u in the subgraph induced in G by nodes in the set $\{v \in V : v \text{ is active at time } t-1\} \cup \{u\}$ contains at least $\theta(u)$ nodes.

OBJECTIVE : Find a minimum cardinality seedset $Y \subseteq V$ such that if every node in Y is activated at time step 0, then all remaining nodes in V eventually activate.

Goldberg and Liu (2013) proved that TD is NP-hard by reducing it to the *set cover* problem. For this reason, they focused on *approximation algorithms* for TD. An approximation algorithm is an algorithm that returns a solution to a (combinatorial) optimization problem that is provably close to optimal, and runs in polynomial time. An algorithm is called an α-approximation algorithm for a minimization problem P_I, if for every instance I of P_I, it computes a feasible solution to I of value at most α times the value of an optimal solution to I, with running time polynomial in the input size.

Goldberg and Liu presented an approximation algorithm for the Technology Diffusion problem, based on randomized rounding applied to the solution of a linear program, that returns a seedset of size $O(rq \cdot \log(n))$ times that of an optimal seedset, where r is the diameter[1] of the given graph, and q is the number of distinct thresholds used in the instance. Könemann et al. (2013) improved upon this result by presenting a $O(\min\{r,q\}\log(n))$-approximation algorithm by reducing TD to submodular set cover and a quota version of the node weighted Steiner tree.

[1] The length of the longest path in the set of shortest paths between all pairs of vertices in a graph.

Efsandiari et al. (2015) showed that TD is NP-hard even on *spiders* (trees in which at most one vertex,called the root, has degree larger than 2) and that the problem also admits an $O(\log(q))$-approximation on these graphs. All these algorithms mentioned output a seedset that induces a *connected activation sequence*, defined as follows.

Definition 1.0.1. *A seedset Y induces a* ***connected activation sequence*** *if there exists a permutation $\pi = (v_1, \ldots, v_n)$ of V such that : (i) the graph induced by $v_1, \ldots, v_i$ is connected, for all $i = 1, \ldots, n$, and (ii) $v_i \in Y$ whenever $i < \theta(v_i)$.*

Goldberg and Liu (2013) showed that the size of a minimum cardinality seedset that induces a connected activation sequence has cardinality at most twice as that of the optimal seedset.

Our results and techniques: The starting point of our thesis is the work of Efsandiari et al. (2015). As previously mentioned, they give a $O(\log q)$-approximation algorithm for TD on spider graphs, and they achieve that using a reduction to a problem of *scheduling jobs with precedence constraints*. This problem asks for non-preemptive scheduling of a set of n unit-sized jobs on a single machine, in such a way that chain-like precedence constraints among the jobs are satisfied. Each job has a deadline, and the objective is to compute a schedule that obeys the precedence constraints, while minimizing the number of jobs completed after their deadline (more details in Section 3). Efsandiari et al. (2015) proved the following: (i) there is an approximation-preserving reduction from instances of TD on spiders with ℓ legs and q different threshold values where one seeks for connected activation sequences starting at the root of the spider, to instances of the scheduling problem where precedence constraints are expressed as ℓ job chains, and there are q distinct deadlines values; (ii) there is a $O(\log q)$-approximation algorithms for the scheduling problem; (iii) there is an exact polynomial-time algorithm for the scheduling problem if ℓ is a fixed constant.

Combining (i) and (ii) together with the result of Goldberg and Liu on connected activation sequences, the $O(\log q)$-approximation algorithm for TD on spider graphs follow. However, the result in (iii) does not carry over, since there are instances of TD on spider graphs (in fact, even on paths) where an optimal solution does *not* induce a connected activation sequence. This motivated us to investigate the complexity of solving TD in spider graphs with a constant number of legs.

Our first result is an exact polynomial-time algorithm for TD on spider graphs with a constant number of legs. Our algorithm is based on a simple dynamic program, and in fact, it works in the more general weighted setting, i.e., when each vertex of the spider has an associated non-negative weight, and the objective is to minimize the total weight of the chosen seedset, rather than its cardinality.

Given that our problem is algorithmically solvable in polynomial-time, we then investigate whether we can give a Linear Programming (LP) formulation for the problem of polynomial size, and show that indeed this is the case. Specifically, we show that our algorithm fits into the paradigm given by Martin et al. (1990), who developed a polyhedral characterization of discrete dynamic programs, by viewing dynamic programming algorithms as seeking flows in directed hypergraphs.

Finally, we initiate the study of the *influence maximization* version of TD, called maxTD, which seeks a seedset of a fixed size k maximizing the number of vertices that will eventually activate. Influence maximization problems of this kind have been studied extensively in localized diffusion models, and therefore it is natural to investigate the same question for technology diffusion models. Relying once again on dynamic programming, we show that also this problem is solvable in polynomial-time on spiders with a constant number of legs.

Thesis Organization : In Chapter 2, we give a brief overview of some related works, including the independent cascade and linear threshold models, which are among the most studied models of cascade effects in networks. We also give a more detailed description of the results on the technology diffusion problem. In Chapter 3, we describe our polynomial-time dynamic programming algorithm for TD on spiders with a constant number of legs. In Chapter 4, we transfer our results in Chapter 3 to a polyhedral characterization of TD on spiders. Finally, in Chapter 5, we use our dynamic program in Chapter 3, to construct a polynomial time dynamic programming algorithm for maxTD on spiders with a constant number of legs.

Chapter 2

Related Works

In this section we give a brief overview of three diffusion models. All the models are *progressive*, meaning that a node, once activated, does not become inactive. First, we focus on two popular stochastic models widely studied in the literature, *independent cascade* and *linear threshold*. For a more detailed study of these models, we refer the reader to a comprehensive survey by Chen et al. (2013). Then, we look at the results on the technology diffusion model.

2.1 Localised Diffusion Models

Let us start by looking at the definition of a social network. Note that this applies to just the first two localized models - information cascade and linear threshold.

Notational Convention: We model a social network as a directed graph, $G = (V, E)$. The set V is a finite set of n nodes, representing entities in a social network. The set $E \subseteq V \times V$ is the set of directed edges connecting pairs of nodes, and it represents the relationships among entities in the network. The models under consideration assume such relationships to be directed. We also use $N_{in}(v)$ to denote the set of nodes that have arcs incoming to a node, v.

Suppose, we have a certain piece of information, and we want to trigger a cascade of information diffusion from a small set of nodes, with the aim of reaching as many nodes in the social network, G, as possible. The diffusion proceeds in discrete time steps, with time $t = 0, 1, 2, ..., n$. Each node $v \in V$ can have either of two possible states at a time step, inactive or active. The active state of a node represents that it has received the information being propagated through the network, while inactive state represents that the node has not received that information. We assume that all nodes in the input graph are inactive. We define the set $Y_t \subseteq V$ as the set of active

nodes at time t. The set Y_0 is referred to as the seedset, and nodes in this set, as seeds. These seed nodes are the initial nodes selected to propagate some information, for example, they could be the initial set of people who receive the new piece of information.

The *influence* of a set of nodes, $H \subseteq V$, is defined as the number of nodes that are active at $t = n$ in G if H was the seedset, i.e., $Y_0 = H$. This can be seen as the number of nodes H will spread the information to, if all nodes in Y were given the information.

Clearly, computing influence of a set of nodes requires computing the sets Y_t for all time steps for which the activation process occurs. Computing these sets requires specifying how a node changes its state from inactive to active, which is described by a *diffusion model*. A diffusion model for a network $G = (V, E)$ specifies the process of computing sets Y_t for all $t \geq 1$ given the seedset Y_0. Let us now look at two examples of localized diffusion models. Recall that localized models allow only a node's neighbours to impact its behaviour.

2.1.1 Independent Cascade (IC) Model

The independent cascade (IC) model is one of the simplest, and also one of the most widely studied diffusion models. It was inspired from interactions in particle systems [Durrett (1988), Liggett (1985)] studied in probability theory. Research in marketing theory by [Goldenberg et al. (2001a), Goldenberg et al. (2001b)] has also heavily influenced the creation of this model. It also has similarities to cascade models used in sociology and epidemiology [Anderson et al. (1992)] to study the spread of ideas and diseases respectively. The version we describe here is attributed to Kempe et al. (2003).

Independent Cascade (IC)

SETTING : A directed graph, $G = (V, E)$ and a probability function $p : E \rightarrow [0, 1]$ on all the arcs. The probability function represents the likelihood of one node influencing another node.

ACTIVATION PROCESS : We start with a set of active nodes, Y_0. The activation process proceeds in discrete time steps. If a node, v, is activated at time step t, it can activate each of the inactive nodes in its neighbourhood only at time step $t+1$. Each inactive node $u \in N_{in}(v)$ is activated with probability $p(v, u)$, and this activation attempt of u is independent of all previous activation attempts, if any, by u's other active neighbours. Each discrete time step can have multiple nodes being activated. If u has multiple neighbours activating at the same time, activation attempts of u occur in an arbitrary sequence.

The independent cascade model makes activation events along edges mutually independent of each other. This makes it particularly well suited to model simple cascade effects where node activations may be triggered from a single source, such as the spreading of information or viruses. However, several situations exhibit a much higher complexity in activation mechanisms. In particular, situations in which exposure to multiple independent activation sources are needed for an node to change its state. Such situations are more accurately modelled by the linear threshold model, which is described next.

2.1.2 Linear Threshold (LT) Model

In several real world situations, exposure to one activation source, when adopting an innovation may be insufficient for a node to change its state. People may need feedback from multiple sources in their social network before they adopt the innovation. For example, a person thinking about upgrading their phone to a newer, more expensive model may consult several sources before making a decision.

Social scientists have proposed *threshold behaviours* to model these kinds of cascade effects [Granovetter (1978), Schelling (2006)]. These models define an aggregate function of all activation attempts made on a node. This function could simply be a summation of all the attempts, or could be something more complex. When the value of the aggregate function associated to a node exceeds a certain threshold, the node is activated. Linear Threshold is a popular model that

follows this rule. The version of the linear threshold model we describe here is also attributed to Kempe et al. (2003).

Linear Threshold (LT)

SETTING : A directed graph, $G = (V, E)$, and a weight function $b : E \to [0, 1]$ on the edges such that $\sum_{u \in N^{in}(v)} b(v, u) \leq 1$. The weight function represents the extent of one node's influence on another node. We also have a random variable, called a threshold, θ_v, associated with each node, which takes a value uniformly at random from interval $[0, 1]$, and represents the weighted fraction of v's neighbours that have to be active in order for v to activate.

ACTIVATION PROCESS : We start with a set of active nodes, Y_0. The activation process proceeds in discrete time steps. At time step t, all nodes active at time step $t-1$ retain their active state, and any node v is activated at t if the total weight of its active neighbours, $\sum_{u \in N^{in}(v)} b(v, u) \geq \theta_v$.

Linear threshold does not necessarily require multiple sources to activate a node, because we select thresholds randomly from the interval [0,1]. However, it does represent more complex mechanisms than the information cascade model, as thresholds can model the number of activation sources required to change a node's state.

Properties of Influence Functions in IC and LT Models

For both the independent cascade model and the linear threshold models, we can describe an *influence spread function*.

Definition 2.1.1. *An* ***influence spread function****,* $\sigma : 2^V \to \mathbb{R}_+$*, in an information cascade model maps each subset of the nodes* $S \subseteq V$ *to a positive integer, which is the expected number of the active nodes at the end of the activation process if* S *is the set of initially active nodes.*

Wang et al. (2012) and Chen et al. (2010) showed that computing this influence spread function is #P-hard for both the IC and LT models. #P is a complexity class of counting problems associated with decision problems in NP. For example, asking whether there are any Hamiltonian cycles in an input graph is a problem in the complexity class NP. Counting the number of Hamiltonian cycles in an input graph is a problem in the class #P. A problem is called #P-hard if

every problem in #P can be reduced to it by a polynomial time reduction. #P-hardness is at least as hard as NP-hardness, since counting the number of solutions determines whether a solution exists.

Influence spread functions of both the independent cascade model and the linear threshold model exhibit two common properties: *submodularity* and *monotonicity*, which we define below.

Definition 2.1.2. *A function $f : 2^U \to \mathbb{R}_+$ is **submodular** if*

$$f(A \cup \{u\}) - f(A) \geq f(B \cup \{u\}) - f(B)$$

for all $A \subseteq B \subseteq U$ and $u \in U$.

We can interpret submodularity as a diminishing marginal returns property - as the set grows larger, the effect of adding an element to it reduces.

Definition 2.1.3. *A function $f : 2^U \to \mathbb{R}$ is **monotonic** if*

$$f(A \cup \{u\}) \geq f(A)$$

for all $A \subseteq U$ and $u \in U$.

Monotonicity means that adding elements to a set will not reduce the function value. Kempe et al. (2003) showed that the IC and LT models described have influence spread functions that were both monotonic and submodular.

The properties of submodularity and monotonicity are used to design algorithms for the influence maximization problem, which we describe next.

Influence Maximization in IC and LT Models

Often, companies could have an upper limit on the number of potential customers they want to market their product to. However, they still want their sales to be as high as possible. In this case, we need to find a set of nodes such that if activated, would maximize the number of total active nodes in the network. The cardinality of this seedset cannot exceed a fixed upper limit. This is the *influence maximization* problem.

Definition 2.1.4. ***Influence maximization*** *is the following optimization problem : given a graph $G = (V,E)$, a diffusion model on G, and a non negative integer, called a budget, k, the objective*

is to find a seedset $Y \subseteq V$ of cardinality at most k, such that the influence spread of Y, $\sigma(Y)$, under the given diffusion model is maximized. Formally, we compute Y such that

$$Y = \underset{Y' \subseteq V, |Y'| \leq k}{\operatorname{argmax}} \sigma(Y').$$

Domingos and Richardson (2001) introduced influence maximization as an algorithmic tool for viral marketing within a probabilistic framework. The authors modelled a social network as a Markov random field [Besag (1974), Jain and Chellappa (1993), Kindermann and Snell (1980)], where each individual's probability of purchasing some product is a function of both the utility of the product for the individual and the influence of other customers. They used data mining techniques to identify a set of influential nodes to market to. Kempe et al. (2003) later framed the question of identifying this seedset as a stochastic discrete optimization problem. Kempe et al. (2003) also showed influence maximization to be computationally hard for both of the IC and LT models described before by a reduction to set cover. Also, a corollary of their result implies that influence maximization is #P-hard [Chen et al. (2013)].

There is a simple greedy approach to solve the influence maximization problem, relying on the fact that the influence spread function is submodular and monotonic. In fact, this approach works for any diffusion models in which the influence spread function is submodular and monotonic.

Algorithm 1 Greedy Algorithm for Seedset Selection

INPUT: A graph, $G = (V, E)$, budget $k \leq |V|$ and a monotone and submodular set function, $f : V \rightarrow \mathbb{R}_+$.

OUTPUT: A seedset, $Y \subseteq V$, of cardinality k.

```
1: initialize Y ← ∅
2: for all i ∈ [k] do
3:     a ← argmax_{b ∈ V\Y} f(Y ∪ {b}) − f(Y)
4:     Y ← Y ∪ {a}
5: end for
6: return Y
```

Pseudocode for greedy algorithm for seedset selection

The greedy algorithm, described in Algorithm 1, adds one element, a, into the candidate seedset, Y, such that a provides the highest contribution to an objective function f with respect to set Y. This procedure is repeated for k iterations.

When the set function f is monotonic and submodular, Algorithm 1, provides an approximation guarantee, as shown by the following theorem.

Theorem 2.1.5. *(**Nemhauser et al. (1978)**) Given a universe of elements, U, and a function $f : U \to \mathbb{R}_+$ such that f is monotone and submodular and $f(\varnothing) = 0$, let $Y^* \subseteq U$ be the set that maximizes $f(Y)$ among all subsets of U with cardinality at most k. Then for the set Y' computed by Algorithm 1, we have*

$$f(Y') \geq (1 - \frac{1}{e}) f(Y^*)$$

where e is the base of natural logarithm.

Using Theorem 2.1.5, with f as the influence spread function σ, we get that Algorithm 1 guarantees an approximation ratio of $(1 - \frac{1}{e})$. However, Algorithm 1 requires repeated evaluations of $\sigma(Y)$ in Step 3, which we know to be #P-hard. Some approximation algorithms discussed next avoid explicit influence spread computation. Kempe et al. (2003) used Monte Carlo simulations to estimate influence spread in Algorithm 1, getting an approximation guarantee of $(1 - \frac{1}{e} - \varepsilon)$ with running time $O(\varepsilon^{-2} k^3 n^2 m \log n)$, where k is the number of iterations of the greedy algorithm and m is the number of edges in the input graph.

This running time is too high for a graph with a large number of vertices and edges. Leskovec et al. (2007) used lazy evaluations [Minoux (1978)] of the influence spread functions to obtain up to 700 times of speed-up for network optimization problems related to influence maximization. This was shown empirically. Kimura et al. (2007) and Chen et al. (2009) used batch estimates of the influence spread function. Chen et al. (2009) reported that this technique improves running time of the lazy evaluations for influence maximization in the IC models from 15% to 34%.

Even with the lazy evaluations and batch estimates, the resulting algorithm is still not practical for graphs having hundred of thousands of nodes. Chen et al. (2010), Wang et al. (2012). Chen et al. (2010) reported that on a directed graph with 76K nodes and 509K arcs, it took over 60 hours to find 50 seeds.

A number of heuristic algorithms have been proposed to deal with this problem [Chen et al. (2009), Chen et al. (2010), Goyal et al. (2011), Jung et al. (2012), Wang et al. (2012)]. These algorithms avoid Monte Carlo simulations by exploiting specific aspects of the graph structure and the diffusion model to significantly speed up the influence spread function computations.

Seed Minimization in IC and LT Models

In some scenarios, it may be useful to find the smallest set of nodes that have to be activated in order to achieve a certain number of total active nodes in a social network. For example, in a viral marketing campaign, a company could want to know the set of potential customers they have to market their product to so that they achieve some target on the total products sold. To minimize the costs of the campaign, they would want that set to be as small as possible. We call this the *seed minimization* problem.

Definition 2.1.6. ***Seed minimization*** *is the following optimization problem : given a graph* $G = (V,E)$*, a diffusion model on G, and a non-negative integer* $k \leq |V|$*, called coverage, the objective is to find a minimum cardinality seedset* $Y \subseteq V$*, such that the expected influence spread of* Y*,* $\sigma(Y)$*, under the given diffusion model is at least k. Formally, we compute* Y *such that*

$$Y = \operatorname*{argmin}_{Y' \subseteq V,\ \sigma(Y) \geq k} |Y'|$$

Seed minimization contains the set cover problem as a special case [Kempe et al. (2003)]. Feige (1998) showed that set cover is hard to approximate within a factor of $(1-\varepsilon)\ln n$ unless NP has $n^{O(\log\log n)}$-time deterministic algorithms.

Seed minimization is also closely related to the submodular set cover problem. In an instance of the submodular set cover problem, we are given a universe of elements, U, a monotone submodular set function $f : U \to \mathbb{R}_+$, and a cost function, $c : U \to \mathbb{R}_+$. The goal is to find a set $Y \subseteq U$ minimizing $c(Y) := \sum_{y \in Y} c(y)$ such that $f(Y) = f(U)$. Seed minimization corresponds to a generalized version of the submodular set cover problem, where we seek $Y \subseteq U$ such that $f(Y) \geq k$. Wolsey (1982) showed that a greedy algorithm returns a solution, Y, such that $f(Y) = k$, and $c(Y)$ is within a $\ln[k/(k - f(X_{t-1})]$ factor of the optimal solution, where X_i denotes the greedy solution obtained after i iterations and t is the minimum number of iterations needed by the algorithm to achieve a coverage of k, i.e. $f(X_t) = k$.

Goyal et al. (2013) studied seed minimization for a graph with a cost function associated to vertices $c : V \to \mathbb{R}_+$, where the cost of the seedset, Y, is given by $c(Y) = \sum_{v \in Y} c(v)$. They showed that Wolsey's result does not guarantee an approximation for the seed coverage - infact, it can perform arbitrarily badly compared to an optimal algorithm. The authors showed that when the influence spread function is monotone and submodular, a simple greedy algorithm yields a bi-criteria approximation: given a coverage, k and a parameter $\varepsilon > 0$ the greedy algorithm will produce a seedset, Y, such that $\sigma(Y) > k - \varepsilon$ and $c(Y) \leq (1 + \ln(k/\varepsilon))c(Y^*)$, where Y^* is the optimal seedset whose influence spread is at least k. In an instance of the submodular set

cover problem, we are given a universe of elements, U, a monotone submodular set function $f : U \to \mathbb{R}_+$, and a cost function, $c : U \to \mathbb{R}_+$. The goal is to find a set $A \subseteq U$ minimizing $c(A) := \sum_{a \in A} c(a)$ such that $f(A) = f(U)$.

2.2 Technology Diffusion

Both the independent cascade and the linear threshold models described above are localized, i.e., they have an activation process that activates a node based on solely its neighbours' behaviour. The technology diffusion model we defined in the introduction, however, is not localized. Let us look at the model again.

Technology Diffusion (TD)

SETTING: A simple, undirected graph, $G = (V, E)$, and a threshold function on the nodes, $\theta : V \to \{2, ..., |V|\}$.

ACTIVATION PROCESS : We start with a set of active nodes at time step 0, $Y \subseteq V$. The activation process proceeds in discrete time steps. A node u activates at time t if the connected component containing u in the subgraph induced in G by nodes in the set $\{v \in V : v \text{ is active at time } t-1\} \cup \{u\}$ contains at least $\theta(u)$ nodes.

Note that this model has another difference from the IC and LT models - it is defined on an undirected graph. It assumes that as long as two nodes are connected in a network, they can impact each other's behaviour. Also, all edges are weighted equally, which means that all nodes exert the same amount of influence on the nodes that they are connected to. As far as we know, a directed version of the TD model has not been studied.

The influence spread function of the technology diffusion model unfortunately does not exhibit submodular (or supermodular properties) [Goldberg and Liu (2013)]. This makes designing approximation algorithms much harder. The greedy algorithm (Algorithm 1) we used for the IC and LT models does not give the same approximation guarantee for technology diffusion because it relies on the submodularity properties of σ.

As far as we know, there has been no work on influence maximization for this model. However, there are some results on seed minimization, which we describe next.

2.2.1 Seed Minimization in TD

Goldberg and Liu (2013) gave a $O(rq \cdot \log n)$ approximation for seed minimization in the technology diffusion model, where r is the diameter of G, and q is the number of distinct threshold values in the instance. This is a randomized rounding approach applied to the solution of a linear program. They also used a reduction from the set cover problem to show that the problem does not admit a $o(\log n)$ approximation, even if the diameter and number of distinct threshold values are both constants.

Könemann et al. (2013) improved upon this result by presenting a $O(\min\{r, q\} \log n)$-approximation algorithm based on combinatorial techniques. They presented two algorithms. The first was to obtain a $O(r \log n)$-approximation, which they obtained by reducing a technology diffusion instance to a submodular set cover instance. The second algorithm was for the $O(q \log n)$ approximation, which used a reduction of a TD instance to one of the quota version of the node-weighted Steiner tree[1] problem [Konemann et al. (2013), Moss and Rabani (2007)]. They also showed that the seed minimization problem in technology diffusion is as hard to approximate as the quota-version of the unit-weight node-weighted Steiner tree problem.

[1]In an instance of quota-constrained node-weighted Steiner Tree problem, we are given an undirected graph $G = (V, E)$, a root vertex, $r \in V$, vertex weights $w(v)$ for all vertices $v \in V$, and a positive integral quota, $q \in \mathbb{Z}^+$. The goal is to find a tree T containing $r \in T$, that spans at least q vertices, and minimizes $w(T) = \sum_{v \in T} w(v)$.

Chapter 3

Technology Diffusion on Spiders

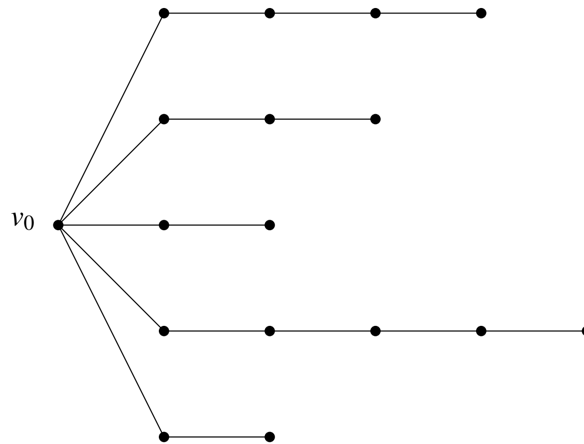


Figure 3.1: A spider graph with 5 legs.

We here focus on the Technology Diffusion problem, defined in the introduction, on spider graphs. Recall that a spider is a tree in which at most one vertex, v_0, (called the *root* of the spider) has degree larger than 2. Note that the edges of a spider can be naturally partitioned into a set of k edge-disjoint paths with the root being one endpoint (see Figure 3.1). We call each such path a *leg* of the spider.

3.1 Connection with Scheduling

Efsandiari et al. (2015) found a connection between TD on spiders and a scheduling problem, called the precedence-constrained single-machine deadline scheduling problem (pDLS). This is a problem of scheduling a set of n jobs non-preemptively on a single machine. Each job, j, has a non-negative processing time, weight, and deadline, and a feasible schedule needs to be consistent with chain-like precedence constraints. A job is late if its processing is completed after its deadline. The goal is to compute a feasible schedule that minimizes the sum of penalties of late jobs. This problem is NP-hard [Lenstra and Kan (1980)].

Precedence-Constrained Single-machine Deadline Scheduling (pDLS)

INPUT : A set $[n] := \{1,...,n\}$ of jobs that need to be scheduled non-preemptively on a single machine. Each job, $j \in [n]$, has a non-negative deadline, $d_j \in \mathbb{N}$, a non-negative processing time, $p_j \in \mathbb{N}$, as well as a non-negative penalty, $w_j \in \mathbb{N}$, associated with it. A schedule pays a penalty of p_j if job j is not processed before its deadline, d_j. Precedence constraints on jobs are given implicitly by a directed acyclic graph, $G = ([n], E)$, which is a collection of ℓ vertex-disjoint paths. Job i has to be processed after job j, where $i, j \in [n]$, if G has a directed i, j -path.

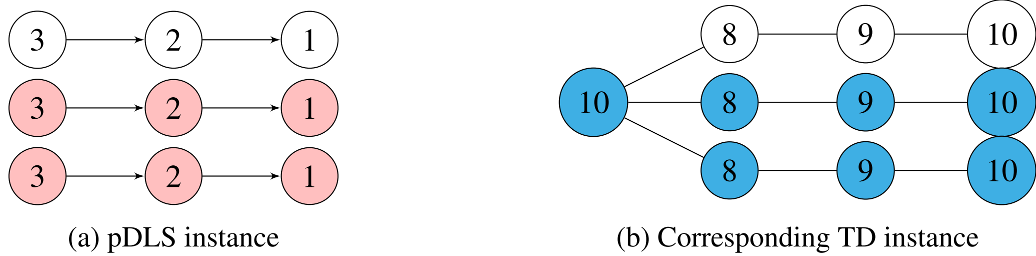
OUTPUT : A feasible schedule that minimizes the total penalty of late jobs, where a schedule is feasible if it is consistent with precedence constraints, and a job j is late if it is completed after its deadline d_j.

Theorem 3.1.1. **(Efsandiari et al. (2015))** *pDLS has a $O(\log q)$ -approximation algorithm, where q is the number of distinct job deadlines in the given instance. Furthermore, pDLS is polynomial time solvable if ℓ is a fixed constant.*

As already mentioned, this problem was shown to have a connection with TD. The following theorem was implicitly stated (without a formal proof) in Efsandiari et al. (2015). We prove it here.

Theorem 3.1.2. *TD in spiders and pDLS with unit processing times and penalties are equivalent whenever in the TD instance we look for a minimum cardinality seedset that induces a connected activation sequence starting at the root.*

Proof. Given an instance of TD on spider graph, $G = (V, E)$, with $V = \{v_0, v_1, v_2, ..., v_{n-1}\}$ and root, v_0, we generate a pDLS instance in the following way : create a job for each vertex $v \in$



(a) pDLS instance

(b) Corresponding TD instance

The graph in (a) shows a pDLS instance with 9 jobs. The deadlines are node labels. The edge directions indicate dependencies, i.e., an edge from a to b indicates that b has to be processed before a. There is no optimal schedule that does not make at least six nodes late. Consider an optimal schedule that makes the six pink nodes late.
The corresponding TD instance is shown in (b). The blue nodes are seeds in a connected activation sequence starting at the root.

Figure 3.2: Converting a pDLS instance into a TD instance.

$V\setminus\{v_0\}$, and let $d_v = n - \theta(v) + 1$, and $p_v = w_v = 1$. Also, create a chain (a directed path) for each leg of the spider in such a way that the job for vertex, v, has to be processed after all its descendants in the spider leg.

We can represent feasible solution of both pDLS and TD by a permutation, P, on the set of nodes, that indicates the order in which the nodes are scheduled or activated, respectively. Consider the permutation of vertices that induce a connected activation starting at the root, $P = \{\pi_0, \pi_1, \ldots, \pi_{n-1}\}$, where $\pi_i = v_i$ for all $i \in \{0, \ldots, n-1\}$. As it grows outward from the root, no vertex precedes its descendant(s) in the spider. We can create a schedule, S, for the pDLS instance by reversing P, i.e., $S = \{\pi_{n-1}, \ldots, \pi_1\}$. This schedule is feasible because every node is processed before its parent. Vice versa, given a permutation S that represents a feasible solution for pDLS, we can reverse it and add v_0 as a first node, to obtain a permutation P for TD that induces a connected activation sequence starting at the root.

A node, v, is defined as a seed when its index (or activation time), i in P, is less than its threshold, i.e., $i < \theta(v)$. In S, this node will have the index $i' = n - i + 1$. A node is late in pDLS if its index in the schedule is more than its deadline, $n - \theta(v) + 1$. By definition, seeds will be at an index $i' > n - \theta(v) + 1$, which is past their deadline in pDLS. This implies that all seeds, other than the root, correspond to late jobs, and vice versa.

Similarly, consider a pDLS instance, where the precedence constraints are given by directed paths. We convert this into a TD instance by rooting the chains at a new common node, and discarding the orientations. This naturally yields a spider (See Figure 3.2). Let the number of nodes in the spider be n. Each node has a threshold $\theta(v) = n - d_v + 1$. The root has threshold set

as n. Let a node, v, with deadline, d_v, be processed late at time, t, in a feasible schedule, S. This gives us $d_v < t$. Reversing S gives us a valid connected activation sequence, P, as every node is processed before its parent in a chain. A late node, processed at time t in S will be activated at $n-t+1$ in P. As $d_v < t$, $n-d_v+1 = \theta(v) > n-t+1$. This means that late jobs correspond to seeds (other than the root) in TD, and vice versa. □

This result of Efsandiari et al. (2015) shows equivalence only for TD instances that have a minimum cardinality seedset inducing a connected activation sequence starting at the root of the spider. However, this is not always the case. Figure 3.3 reports an instance of TD on a path where the minimum seedset Y does not induce a connected activation sequence.

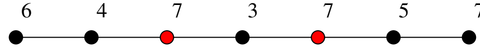


Figure 3.3: A TD instance on a path of 7 nodes.

The node labels indicate the thresholds. The optimal solution requires the activation of the two red nodes with thresholds 7 as seeds. These nodes will eventually activate the entire path. However, the red nodes are not connected. Any seedset inducing a connected activation sequence will contain at least 3 seeds.

For this reason, the second result in Theorem 3.1.1 does not immediately carry over to TD. In particular, the result of Theorem 3.1.1 only shows that finding a minimum cardinality seedset that induces a connected activation sequence, is polynomial-time solvable for TD on spiders with a constant number of legs. In the next section, we will show that the problem remains polynomial time solvable if we remove the restriction of looking for connected activation sequences.

3.2 Dynamic Programming for TD on Weighted Paths and Spiders

We here consider TD in a more general setting, i.e., on vertex-weighted graphs. A graph $G = (V,E,w)$ is vertex-weighted if we have a function on the vertices, $w : V \to \mathbb{R}_+$. Here, the objective of TD is changed from finding the minimum cardinality seedset to activate G. Now, the objective is to find a *minimum weight* seedset to activate all vertices in G. Of course, when $w_v = 1$ for all $v \in V$, we recover our original problem of finding a minimum cardinality seedset.

We will use dynamic programming to solve our problem. Dynamic programming is a powerful technique often used to solve optimization problems. It consists of recursively splitting a

large problem into smaller subproblems, and using the optimal solution of these smaller subproblems to build the optimal solution for the larger ones. Usually, we want the number of distinct subproblems to be small, so that it enables us to solve the optimization problem in polynomial time. Problems like the shortest path, and the longest common subsequence can be solved efficiently using dynamic programming [e.g. see Kleinberg and Tardos (2006)]. In this section, we use this technique to solve Technology Diffusion on spiders with a constant number of legs in polynomial time. To make the explanation simpler, we first describe how to solve the problem on paths, and then extend the result to spiders.

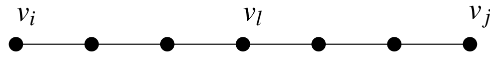


Figure 3.5: TD instance on a path.

3.2.1 TD on Weighted Paths

Let us consider a path, $P = (V, E)$, with $V = \{v_1, ..., v_n\}$ and $E = \cup_{i \in [n-1]} \{(v_i, v_{i+1})\}$. A TD instance on a path, (P, w, θ), has the following property : if a vertex $v_i \in V$ is the last vertex activated in the entire path, and it is activated at time t, then the path segments $v_1, ..., v_{i-1}$ and $v_{i+1}, ..., v_n$ are activated independently of each other before t, i.e, no vertex in the first segment uses a vertex in the second segment to obtain its activation threshold, and vice versa.

This observation gives us a way to decompose our path TD instance into smaller TD instances, which sets up our dynamic programming formulation. Formally, let $D[i, j, l] := (w(Y), Y)$, where $1 \leq i \leq l \leq j \leq n$, be a tuple storing the minimum weight seedset, $Y \subseteq \{v_i, ..., v_j\}$, that activates the segment $v_i, ..., v_l, ..., v_j$, and activates v_l last, and the sum of the weight of the vertices in that seedset, $w(Y)$. We use D to refer to a lookup table for all $D[i, j, l]$ tuple values.

Let us take the base case, where the segment of length 1, i.e., is a single node. Clearly, the node has to be a seed itself to be activated. This gives us

$$D[i, i, i] = \Big(w(v_i), \{v_i\}\Big)$$

for all $i \in [n]$. Now, let us see what values $D[i, j, l]$ takes for segments of length 2 (edges). Note that here l can only take either i or j as its value, and $j = i + 1$, as P is an edge, (i, j). Consider the value of $D[i, j, i]$. This covers the case where v_j is activated first, and v_i second. Since v_j is activated first, it is a seed. Now, we have either of the two following cases :

1. $\theta(\mathbf{v_i}) > \mathbf{2}$
   This means that v_i cannot be activated by v_j, i.e., v_i has to be a seed. This gives $D[i,j,i] = \Big(w(v_i)+w(v_j), \{v_i, v_j\}\Big)$.

2. $\theta(\mathbf{v_i}) = \mathbf{2}$
   This means that v_i can be activated by v_j. This gives $D[i,j,i] = \Big(w(v_j), \{v_j\}\Big)$.

The value of $D[i,j,j]$ is initialised similarly. For segments of length longer than 2, we use the observation on the last activated vertices discussed above. First, we need to define an addition operation on these tuples. Addition of two D tuples, $D[i',j',l']$ and $D[i'',j'',l'']$ yields a tuple, containing the sum of weights and union of the seedsets, i.e., if $D[i',j',l'] = \Big(w(Y'),Y'\Big)$ and $D[i'',j'',l''] = \Big(w(Y''),Y''\Big)$, and we wish to calculate $D[i',j',l'] + D[i'',j'',l'']$, we do the following computation :

$$\begin{aligned} D[i',j',l'] + D[i'',j'',l''] &= \Big(w(Y'),Y'\Big) + \Big(w(Y''),Y''\Big) \\ &= \Big(w(Y')+w(Y''), Y' \cup Y''\Big). \end{aligned}$$

Furthermore, we let $w(D[i,j,l]) := w(Y)$ for $D[i,j,l] = (w(Y),Y)$. Now, we write $D[i,j,l]$ as

$$D[i,j,l] = \underset{\substack{D[i,l-1,i'],\\ i'\in[i\ldots l-1]}}{\operatorname{argmin}}\, w(D[i,l-1,i']) + \underset{\substack{D[l+1,j,j'],\\ j'\in[l+1\ldots j]}}{\operatorname{argmin}}\, w(D[l+1,j,j']) + \alpha_{ijl} \tag{3.1}$$

where $\alpha_{ijl} = (w(v_l),\{v_l\})$ if $j-i+1 < \theta(v_l)$, and $(0,\varnothing)$ otherwise. In other words, it indicates whether we make v_l a seed when we activate v_l last during activation of segment $v_i,\ldots,v_l,\ldots,v_j$ or not. If the number of vertices in the segment are less than $\theta(v_l)$, then v_l is a seed.

On the right hand side of the recurrence, the first term finds the value $i' \in \{i,\ldots,l-1\}$, such that if $v_{i'}$ is the last activated vertex in the segment $v_i,\ldots,v_{l-1}$, then the segment $v_i,\ldots,v_{l-1}$ is activated by the seedset with minimum weight. This is done by checking all vertices $v_{i'} \in \{v_i,\ldots,v_{l-1}\}$, and selecting $v_{i'}$ such that $w(D[i,l-1,i'])$ is minimized. Similarly, the second term finds the value $j' \in \{l+1,\ldots,j\}$, such that if $v_{j'}$ is the last activated vertex in the segment $v_{l+1},\ldots,v_j$, then the segment $v_{l+1},\ldots,v_j$ is activated by the seedset with minimum weight. Note that the first term in the recurrence vanishes if $l=i$ and the second term vanishes if $l=j$.

Algorithm 2 defines the method Path_TD, which computes the seedset for a TD instance on a path using the recurrence in Equation 3.1.

Algorithm 2 Path_TD(P, w, θ)

INPUT: A TD instance, (P, w, θ), with n vertices.
OUTPUT: A lookup table, D, that stores all $D[i, j, l]$ tuple values for $1 \leq i \leq l \leq j \leq n$.

1: **for all** $i \in [n]$ **do**
2: $\quad D[i,i,i] = \Big(w(v_i), \{v_i\}\Big)$
3: **end for**

4: **for all** $i \in \{2, \ldots, n\}$ **do**
5: $\quad$ **for all** $j \in \{1, \ldots, n-i\}$ **do**
6: $\quad\quad$ **for all** $l \in \{j, \ldots, i+j\}$ **do**

7: $\quad\quad\quad$ **if** $l == j$ **then**
8: $\quad\quad\quad\quad D[j, i+j, l] = \underset{\substack{D[l+1,i+j,j'],\\ j' \in \{l+1,\ldots,i+j\}}}{\operatorname{argmin}} w(D[l+1, i+j, j'])$

9: $\quad\quad\quad$ **else if** $l == i+j$ **then**
10: $\quad\quad\quad\quad D[j, i+j, l] = \underset{\substack{D[j,l-1,i'],\\ i' \in [j,\ldots,l-1]}}{\operatorname{argmin}} w(D[j, l-1, i'])$

11: $\quad\quad\quad$ **else**
12: $\quad\quad\quad\quad D[j, i+j, l] = \underset{\substack{D[j,l-1,i'],\\ i' \in [j,\ldots,l-1]}}{\operatorname{argmin}} w(D[j, l-1, i']) + \underset{\substack{D[l+1,i+j,j'],\\ j' \in \{l+1,\ldots,i+j\}}}{\operatorname{argmin}} w(D[l+1, i+j, j'])$
13: $\quad\quad\quad$ **end if**

14: $\quad\quad\quad$ **if** $i+1 < \theta(v_l)$ **then**
15: $\quad\quad\quad\quad D[j, i+j, l] += \Big(w(v_l), \{v_l\}\Big)$
16: $\quad\quad\quad$ **end if**

17: $\quad\quad$ **end for**
18: $\quad$ **end for**
19: **end for**

20: **return** D

Pseudocode for method Path_TD(P, w, θ)

We find values of tuples $D[i,j,l]$ via Algorithm 2. Steps 1 - 3 fill in values for segments with one node. Clearly, the minimum seedset for such a segment is just the vertex in the segment. Steps 4-19 fill in values for segments larger than 1. In Step 4, iterator i fixes the size of the segment. In Step 5, iterator j fixes the leftmost vertex of the segment (making the rightmost vertex v_{i+j}). In Step 6, iterator l fixes the last activated vertex, which has an index in the set $\{j,...,j+i\}$. Steps 7-16 initialize tuple values corresponding to smallest possible seedsets that activate sub-segments $v_j,...,v_{l-1}$ and $v_{l+1},...,v_{i+j}$, starting with all the segments of length 2, then 3, and so on. As discussed earlier, this means calculating the tuples $\left\{ \underset{\substack{D[j,l-1,i'], \\ i'\in[j,...,l-1]}}{\text{argmin}} w(D[j,l-1,i']) \right\}$ and $\left\{ \underset{\substack{D[l+1,i+j,j'], \\ j'\in\{l+1,...,i+j\}}}{\text{argmin}} w(D[l+1,i+j,j']) \right\}$ respectively, and adding them. If the length of segment $v_i,...,v_j$ is less than the threshold of the last activated vertex, v_l, then it means that v_l needs to be in the seedseet, because the segment does not contain enough nodes to activate v_l. In this case, Steps 14-16 add $(w(v_l),v_l)$ to the tuple $D[i,j,l]$, putting v_l in the seedset. Step 20 returns the lookup table of D tuple values associated to each sub-segment in the input path.

Algorithm 3 CalculateSeedset(P,D)

INPUT: A path, $P = (v_1,....v_n)$ and a lookup table of tuples, D, calculated via the method Path_TD.

OUTPUT: A tuple containing a positive integer and a subset of vertices in P.

1: $\Big(w(Y),Y\Big) = \underset{\substack{D[1,n,l], \\ l\in\{1,...,n\}}}{\text{argmin}} w(D[1,n,l])$

2: **return** $\Big(w(Y),Y\Big)$

Pseudocode for method CalculateSeedset(P,D)

Theorem 3.2.1. *Algorithm 2 takes $O(n^3 \log n)$ time, and $O(n^3)$ space.*

Proof. Addition of tuples is assumed to be performed in $O(1)$. The loop in Step 1 takes n iterations. Step 2 is a constant time assignment operation. This means Steps 1-3 are done in $O(n)$ time. In Steps 4-6, each of the loops iterators i, j and l iterate over values from a subset of

$\{1,...,n\}$. Finding a minimum weight $D[i,j,l]$ tuple associated to a segment of length n takes at most n comparison operations, as there are n such tuples (the first and last vertices of the path are fixed, and the last activated vertex can be any vertex in the path). Checking if v_l is a seed and updating the tuple (Steps 14-16) takes one comparison operation, one addition operation, and one assignment operation. This means Steps 4-19 are done in $O(n^4)$ time. Finally, Step 20 returns the lookup table, D, in $O(1)$ time. Hence Algorithm 2 takes $O(n^4)$ time in a naive implementation.

Note that we can use a min-heap [Cormen (2009)] to store our D tuple values. If we associate a min-heap with each segment $(v_i,...,v_j)$, and use it to store tuples values of $D[i,j,l]$, where $l \in \{i,...,j\}$, with $w(D[i,j,l])$ as keys, we can compute the minimum weight seedsets for each segment in $\Theta(1)$ time by retrieving the value at the root of the heap. Insertion into a heap takes $O(\log(j-i+1))$ time, as the heap has $j-i+1$ nodes. For a tuple $D[i,j,l]$, each of i, j and l take values from $\{1,...,n\}$, giving us a total of $O(n)$ distinct tuples. Thus, we can perform Steps 4-19 in $O(n^3 \log n)$ time, reducing our total runtime to $O(n^3 \log n)$.

The input path has n vertices, and thus, $n-1$ edges. Storing the graph as an adjacency list would take $O(n)$ space. As mentioned above, there are $O(n^3)$ distinct tuples. Assuming that each tuple can be stored in $O(1)$ space, our algorithm takes $O(n^3)$ space. □

Theorem 3.2.2. *TD on paths can be solved in $O(n^3 \log n)$ time.*

Proof. Theorem 3.2.1 shows that we can compute the table D in $O(n^3 \log n)$ time. The computation of the final seedset is done by the method CalculateSeedset(P,D), which calculates the value of the tuple

$$\underset{\substack{D[1,n,l],\\ l\in\{1,...,n\}}}{\operatorname{argmin}}\ w(D[1,n,l]).$$

This requires a $\Theta(1)$ lookup operation at the root of the min-heap of size n. The result follows. □

Extension to Cycles

This algorithm is directly applicable to cycles. A cycle, C, just adds an edge, (v_n,v_1), to the path $v_1,v_2...,v_n$. If v_i is the last activated vertex in C, and it is activated at time step t, it cuts C to form a path $v_{i+1},...,v_n,v_1,...,v_{i-1}$ at $t-1$. There are n such paths. We can use the Path_TD method on each of these paths, and thus solve TD on C.

Extension to Trees

Clearly, there is a simple extension of this dynamic programming algorithm to trees. Consider a tree, $T = (V, E)$, with n nodes. A TD instance on T would have the following property : if a vertex $v \in V$ is activated last in T, then all the trees in the forest induced in T by $V - \{v\}$, $T[V - \{v\}]$, will be activated independently of each other.

The total number of subtrees in a tree is not necessarily a polynomial in n [Knudsen (2003), Székely and Wang (2005), Székely and Wang (2007)]. In our algorithm, we need an entry for every node in every connected subgraph of the the input graph. This means that our lookup table can have non-polynomial size. Hence, this approach is not suitable for all trees.

However, spiders with a constant number of legs have a polynomial number of subtrees, as we will show later. We use this fact to design an algorithm to solve TD on weighted spiders with a constant number of legs next.

3.2.2 Extension to Weighted Spiders

Here, we formulate a dynamic program for TD in the case where the input graph is a spider graph. We represent a spider with n nodes as $S[\beta_1, ..., \beta_\ell] = (V, E)$ with $\ell < n$ legs, which are indexed by $[\ell]$. The parameter ℓ is a constant, positive integer. The leg lengths are denoted by a set of ℓ non-negative integers, $\{\beta_1, ..., \beta_\ell\}$. Note that $\sum_{i=1}^{\ell} \beta_i = n - 1$. A vertex is represented as $v_{i,a}$, and the subscript indicates that it is the ith vertex from the root in the leg with index a, $a \in \{1, ..., \ell\}$. The root of the spider is denoted by $v_{0,0}$ (See Figure 3.6).

Each leg is represented by the path (V_a, E_a) for all $a \in [\ell]$. The vertex set of each spider leg is given by $V_a := \{v_{1,a}, ..., v_{\beta_a,a}\}$ for all legs $a \in [\ell]$. The edge set of each spider leg is given by $E_a := \{(v_{j,a}, v_{j+1,a}) \mid j \in [\beta_a - 1]\}$ for all legs $a \in [\ell]$. Note that the vertex set is the union of the root vertex and the vertices of the legs, i.e., $V = \{v_{0,0}\} \cup V_1 \cup V_2 \cup ... \cup V_\ell$. The edge set of the spider is given by $E = E_1 \cup ... \cup E_\ell \cup \{(v_{0,0}, v_{1,1}), ..., (v_{0,0}, v_{1,\ell})\}$. We have weights on the vertices assigned by the function, $w : V \to \mathbb{R}_+$, and a threshold function $\theta : V \to \{2, .., n\}$. We also use the notation $\theta(i, a)$ to represent $\theta(v_{i,a})$ for all $v_{i,a} \in V$.

Let $D[S[\beta_1, ..., \beta_\ell], i, j] := \big(w(Y),\ Y\big)$ be a tuple containing the minimum weight seedset, Y, that fully activates the spider, $S[\beta_1, ..., \beta_\ell]$, such that it activates the vertex $v_{i,j} \in S[\beta_1, ..., \beta_\ell]$ last, and the weight of that seedset, $w(Y)$. Furthermore, we define $w(D[S[\beta_1, ..., \beta_\ell], i, j]) := w(Y)$. We will use the notation $D[S, i, j]$ where the list of leg lengths is clear from context.

Also, for any path, P, of the form $(v_{i,a}, v_{i+1,a}, ...v_{k,a})$, with $1 \leq i \leq k \leq \beta_a$, and any $c \in \{i, ..., k\}$, we let the tuple $D[(v_{i,a}, v_{i+1,a}, ..., v_{k,a}), c, a] = \big(w(Y'), Y'\big)$ contain the minimum weight

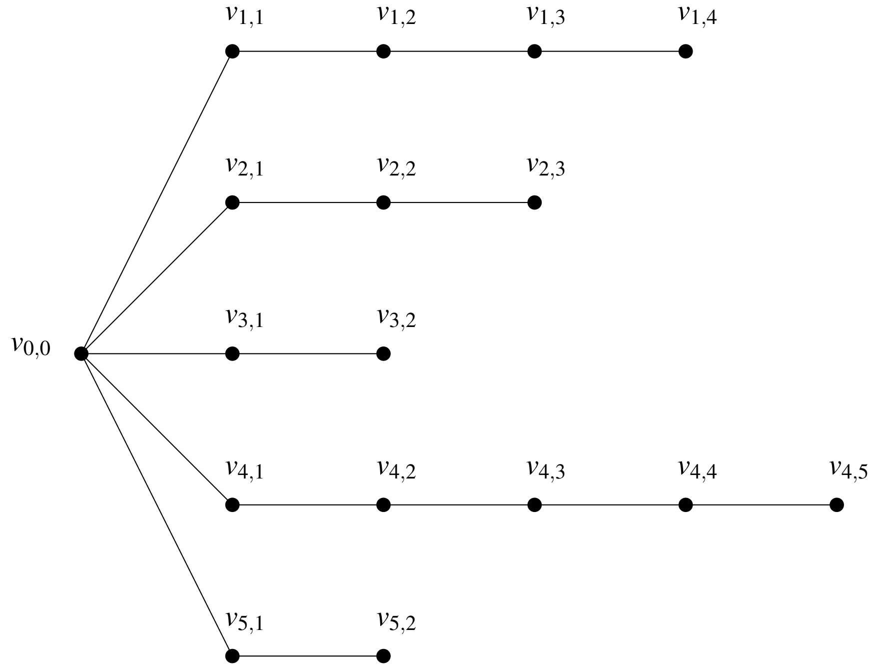


Figure 3.6: A spider graph with 5 legs.

seedset, Y', that activates P, such that $v_{c,a}$ is the last activated vertex, and the weight of that seedset, $w(Y')$. Furthermore, we define $w(D[(v_{i,a}, v_{i+1,a}, ..., v_{k,a}), c, a]) := w(Y')$. Again, we will use the notation $D[P, c, a]$ instead of $D[(v_{i,a}, v_{i+1,a}, ..., v_{k,a}), c, a]$ where the path vertex indices are clear from context. Recall that this tuple can be computed with a Path_TD method call, with P as input.

We define the following operations on the tuples.

1. *Addition of D tuple values*: This yields a tuple containing the sum of weights, and union of the seedsets, i.e., if $D[S', i', a'] = \Big(w(Y'), Y'\Big)$ and $D[S'', i'', a''] = \Big(w(Y''), Y''\Big)$, and we wish to calculate $D[S', i', a'] + D[S'', i'', a'']$, we do the following computation :

$$\begin{aligned} D[S', i', a'] + D[S'', i'', a''] &= \Big(w(Y'), Y'\Big) + \Big(w(Y''), Y''\Big) \\ &= \Big(w(Y') + w(Y''), Y' \cup Y''\Big). \end{aligned}$$

2. $\alpha(S, i, a)$: When i and a are non-negative integers such that $v_{i,a} \in V$, this operation returns

Algorithm 4 Spiders(S,k)

INPUT: A spider, $S[\beta_1,\ldots,\beta_\ell]=(V,E)$, and a positive integer $k\leq|V|$.
OUTPUT: Connected subgraphs of S which have exactly k nodes, and contain the root vertex.

```
1: R = ∅

2: if k == 1 then
3:     R = {({v_{0,0}}, ∅)}

4: else
5:     R' = Spiders(S, k − 1)
6:     for all S'[b_1,...,b_ℓ] ∈ R' do
7:         for all a ∈ [ℓ] do
8:             R = R ∪ {Join(S', b_a + 1, a)}
9:         end for
10:    end for
11: end if

12: return R
```

Pseudocode for Spiders(S,k).

a tuple $(w(Y), Y)$, where Y is set as $\{v_{i,a}\}$ with $w(Y) = w(v_{i,a})$, if the number of vertices in the spider S is less than the threshold of $v_{i,a}$, and $(\varnothing, 0)$ otherwise.

3. *Split*$(S[\beta_1, ..., \beta_\ell], i, a)$: When i and a are positive integers such that $i \leq \beta_a$, $a \in [\ell]$, this operation returns a spider, $S[\beta_1, ..., \beta_a - (\beta_a - i + 1), ..., \beta_\ell] = S[\beta_1, ..., i-1, ..., \beta_\ell]$. In other words, it cuts off the leg segment $v_{i,a}, ..., v_{\beta_a,a}$ from the input spider, S.

4. *Join*$(S[\beta_1, ..., \beta_\ell], i, a)$: When i and a are positive integers such that $i = \beta_a + 1$, $a \in [\ell]$, this operation returns a spider, $S[\beta_1, ..., \beta_a + 1, ..., \beta_\ell]$. In other words, it adds the vertex $v_{i,a}$ to the spider, S.

5. *Spiders*(S, k) : When k is a positive integer such that $k \leq |V|$, this operation returns the set of connected subgraphs of S which have exactly k nodes, and contain the root vertex.

The next four lemmas show that each of these operations can be performed in polynomial time.

Proposition 3.2.3. *The operation $\alpha(S, i, a)$ takes constant running time.*

Proof. This operation requires just checking the condition $\theta(i, a) > |V|$, which is assumed to be computable in constant time. □

Proposition 3.2.4. *The algorithm for Join$(S[\beta_1, ..., \beta_\ell], i, a)$ takes constant running time.*

Proof. This operation requires adding a vertex and an edge to the sets V and E. Assuming that the set of vertices and the set of edges in each leg are stored as a separate linked list, this would require constant time. □

Proposition 3.2.5. *The algorithm for Split$(S[\beta_1, ..., \beta_\ell], i, a)$ takes $O(n)$ running time.*

Proof. This operation requires deleting at most $n-1$ vertices and at most $n-1$ edges. Since all vertices in S have at most ℓ edges incident on them, this operation will require $O(n)$ time. □

Proposition 3.2.6. *The algorithm for Spiders(S, k) takes $O(n^\ell)$ running time.*

Proof. This operation requires arranging $k-1$ vertices into at most ℓ legs such that they form a connected spider graph. As the legs are distinct, there are $\binom{k-1+\ell-1}{\ell-1}$ such arrangements.

Algorithm 4 gives the pseudocode to compute them. It starts with a root vertex (Steps 2-3), and adds one vertex at a time. There are ℓ legs to which this new vertex can be added. So, we do Join operations considering all ℓ arrangements, yielding ℓ distinct spiders. The next new vertex is

added to each of these ℓ spiders. This is done recursively until all spiders created have k vertices (Steps 4-11).

Addition of the ith vertex is done iteratively on $\binom{i+\ell-2}{\ell-1}$ spiders, for all of their ℓ legs. Thus, the running time of the algorithm, $T(k)$, is given by

$$\begin{aligned} T(k) &= 1 + \ell \cdot \sum_{i=2}^{k} \binom{i+\ell-2}{\ell-1} \\ &\leq 1 + \ell \cdot \sum_{i=2}^{n} \binom{i+\ell-2}{\ell-1} \\ &\leq 1 + \ell \cdot n \cdot \binom{n+\ell-2}{\ell-1} \\ &= O\Big(\ell \cdot n \cdot \binom{n+\ell-2}{\ell-1}\Big) \\ &= O(\ell \cdot n^{\ell}) \\ &= O(n^{\ell}). \end{aligned}$$

As ℓ is a constant, running time is polynomial in n. □

To calculate the minimum cardinality seedset of a spider we observe two things:

1. If the last activated vertex is $v_{i,a} \neq v_{0,0}$, then the spider can be seen as split into a smaller spider, $S[\beta_1, ..., \beta_a - (\beta_a - i + 1), ..., \beta_\ell]$, and a leg segment $(v_{i+1,a}, ..., v_{\beta_a,a})$, which are activated independently of each other. As the leg segment is a path, and is activated independently of the all other vertices outside the segment, we can calculate the minimum cardinality seedset to activate it with Equation 3.1.

2. If the last activated vertex is the root, we can calculate minimum cardinality seedsets for each of the ℓ legs using Equation 3.1. This can be computed in polynomial time, specifically, $\ell \cdot O(n^3 \log n) = O(n^3 \log n)$. The minimum seedset required to activate the entire spider will be the union of the minimum seedsets to activate each of the ℓ legs, and the result of $\alpha(S, 0, 0)$, where the latter decides if the root has to be in the seedset.

In Algorithm 5, we use the observations above to calculate the tuple values for the connected subgraphs of a spider. We start with an empty tuple lookup table, D, in Step 1. Steps 2-8 iterate over all legs (we define the legs as starting with the root vertex for the purposes of this algorithm), and call the method Path_TD(P_a, w, θ) on each leg, P_a, where $a \in [\ell]$. This method

Algorithm 5 Spider_TD($S[\beta_1,...,\beta_\ell],w,\theta$)

INPUT: A TD instance on a spider, $(S[\beta_1,...,\beta_\ell],w,\theta)$, with n vertices.
OUTPUT: A lookup table of tuples, D.

1: $D \to \varnothing$
2: **for all** $a \in [\ell]$ **do**
3: $P_a := (v_{0,0}, v_{1,a}, v_{2,a}, ..., v_{\beta_a,a})$
4: $D_a = \text{Path_TD}(P_a, w, \theta)$
5: **for all** $D_a[i,j,l] \in D_a$ **do**
6: $D[(v_{i,a},...,v_{j,a}),l,a] = D_a[i,j,l]$
7: **end for**
8: **end for**
9: $D[S[0,...,0],0,0] = \Big(w(v_{0,0}), \{v_{0,0}\}\Big)$
10: **for all** $k \in [n-1]$ **do**
11: **for all** $S'[b'_1,...,b'_\ell] \in \text{Spiders}(S[\beta_1,...,\beta_\ell],k)$ **do**
12: **for all** $b'_j \in \{b'_1,...,b'_\ell\}$ **do**
13: **if** $b'_j < \beta_j$ **then**
14: $\hat{b}_j = b'_j + 1$
15: $\hat{S}[\hat{b}_1,...,\hat{b}_\ell] = \text{Join}(S'[b'_1,...,b'_\ell], \hat{b}_j, j)$
16: **for all** $v_{i,a} \in \hat{S}$ **do**
17: **if** $i == \hat{b}_j$ AND $a == j$ **then**
18: $D[\hat{S},\hat{b}_j,j] = \underset{\substack{D[S',c,d],\\ v_{c,d}\in S'}}{\text{argmin}}\ w(D[S',c,d]) + \alpha(\hat{S},\hat{b}_j,j)$
19: **else if** $i == 0$ AND $a == 0$ **then**
20: $\overline{P}_d := (v_{1,d},...,v_{\hat{b}_d,d})\ \forall d \in [\ell]$
21: $D[\hat{S},0,0] = \sum_{d\in[\ell]} \Big(\underset{\substack{D[\overline{P}_d,c,d],\\ v_{c,d}\in\overline{P}_d}}{\text{argmin}}\ w(D[\overline{P}_d,c,d]) \Big) + \alpha(\hat{S},0,0)$

Pseudocode for method Spider_TD($S[\beta_1,...,\beta_\ell],w,\theta$).

Algorithm 5 Spider_TD$(S[\beta_1,...,\beta_\ell],w,\theta)$ (continued)

```
22:             else
23:                 $\overline{S}[\overline{b}_1,...,\overline{b}_\ell] = \text{Split}(\hat{S},i,a)$
24:                 $\overline{P} := (v_{i+1,a},...,v_{\hat{b}_a,a})$
25:                 $D[\hat{S},i,a] = \underset{D[\overline{S},c,d],\ v_{c,d}\in\overline{S}}{\text{argmin}}\ w(D[\overline{S},c,d]) + \underset{D[\overline{P},c,a],\ v_{c,a}\in\overline{P}}{\text{argmin}}\ w(D[\overline{P},c,a]) + \alpha(\hat{S},i,a)$
26:             end if

27:           end for
28:         end if
29:       end for
30:     end for
31: end for

32: return D
```

Pseudocode for method Spider_TD$(S[\beta_1,...,\beta_\ell],w,\theta)$ (continued).

returns a tuple lookup table for each leg, D_a. We update D by putting the value of tuple $D_a[i, j, l]$ in $D[(v_{i,a}, ..., v_{j,a}), l, a]$. Thus, we can get the minimum weight seedsets of all connected subsegments in each leg from D.

In Step 9, we initialise the tuple value for a spider containing just the root vertex. Steps 10-30 solve TD on connected subgraphs of S that contain the root vertex, increasing the size of the subgraphs by one vertex with every increment in the iterator, k. In Step 11, the iterator $S'[b'_1, ..., b'_\ell]$ iterates over all spiders with k vertices, where the set of these spiders is obtained by using the Spiders(S, k) operation. In Step 12, the iterator b'_j iterates over the set of leg lengths of $S'[b'_1, ..., b'_\ell]$. If $b'_j < \beta_j$, which is the length of the jth leg of the input spider, S, we add one vertex to the jth leg of S', using the Join operation in Step 14, creating the spider $\hat{S}$. Now, we need to calculate $D[\hat{S}, i, a]$ for all $v_{i,a} \in \hat{S}$. As in the case of paths (Theorem 3.2.1), we store $D[\hat{S}, i, a]$ tuple values in a min-heap, where the keys are the $w(D[\hat{S}, i, a])$ values. Every distinct spider is assigned its own separate min-heap, and it stores $D[\hat{S}, i, a]$ values for all $v_{i,a} \in \hat{S}$.

The following lemma shows that each tuple $D[\hat{S}, i, a]$ can be computed in polynomial time.

Lemma 3.2.7. *We need $O(k)$ operations each to compute $D[\hat{S}, i, a]$ for all $v_{i,a} \in \hat{S}$, where $\hat{S}$ has k vertices, i.e., Steps (17-26) of Algorithm 5 can be computed in $O(k)$ time.*

Proof. There are three possible cases which occur when a vertex, $v_{\hat{b}_j, j}$, is joined to spider S' to form spider $\hat{S}$, which has k vertices. Now, we want to compute $D[\hat{S}, i, a]$.

1. If the last activated vertex, $v_{i,a}$ is the same as the newly added vertex, $v_{\hat{b}_j, j}$, then to obtain the minimum seedset required to activate the spider $\hat{S}$, we sum the tuple value corresponding to the minimum seedset required to activate S', and $\alpha(\hat{S}, \hat{b}_j, j)$ where the latter decides if the new vertex has to be in the seedset (Steps 17-18).

   The minimum seedset can be found by looking at the root of the tuple min-heap associated to S'. This is a constant time operation. Insertion into the min-heap can be done in $O(\log k)$ operations, as the number of vertices in $\hat{S}$ is k. As addition and computation of $\alpha(\hat{S}, \hat{b}_j, j)$ takes constant time (Proposition 3.2.3), this case requires $O(\log k)$ time.

2. If the last activated vertex is the root, we can calculate minimum cardinality seedsets for each of the ℓ legs of $\hat{S}$ using Equation 3.1. The minimum seedset required to activate the spider $\hat{S}$ will be the union of the minimum seedsets to activate each of the ℓ legs and the result of $\alpha(\hat{S}, 0, 0)$, where the latter decides if the root has to be in the seedset (Steps 19-21).

Again, the minimum seedset that activates a path, $\overline{P}_d$, can be found by looking at the root of the tuple min-heap associated to $\overline{P}_d$. This is a constant time operation, repeated for $\ell = O(1)$ times. Insertion into the min-heap can be done in $O(\log k)$ operations, as the number of vertices in $\hat{S}$ is k, and that is the upper bound on vertices in $\overline{P}_d$. Hence, the union of the minimum seedsets to activate each of the ℓ legs can be done in $O(\ell + \log k)$ operations and ℓ addition operations. Two additional constant time operations for computing and addition of the result of $\alpha(\hat{S}, 0, 0)$ are required. Thus, this case requires $O(\log k)$ (ℓ is a constant) time.

3. Else, $\hat{S}$ can be seen as split into a smaller spider, $\overline{S}[\overline{b}_1, ..., \overline{b}_\ell]$ and a leg segment $\overline{P} = (v_{i+1,a}, ..., v_{\beta_a,a})$, which are activated independently of each other (as $v_{i,a}$ is activated last). Note that we have calculated tuple values both for $\overline{S}$ (as it has fewer vertices than $\hat{S}$ and would have been considered in a previous iteration of the algorithm), and $\overline{P}$ (as we calculate tuples for all sub-segments in a leg) (Steps 22-25).

   This requires comparing all tuple values associated to $\overline{S}$ and $\overline{P}$ respectively to compute minimum seedsets to activate $\overline{S}$ and $\overline{P}$. As $\overline{S}$ and $\overline{P}$ combined have $k-1$ vertices by construction, the minimum seedset is calculated in $O(1)$ operations, and insertion of the $D[\hat{S}, i, a]$ tuple into the min-heap for $\hat{S}$ takes $O(\log k)$ operations. There are 3 addition operations, and two method calls to $\alpha(\hat{S}, i, a)$ and Split. These methods have been shown to have constant, and $O(k)$ run times respectively ((Proposition 3.2.3) and (Proposition 3.2.5)). Thus, this case takes $O(k)$ operations.

□

Theorem 3.2.8. *Algorithm 5 takes $O(n^{\ell+2})$ running time.*

Proof. Steps 1-8 make ℓ calls to the Path_TD method, which take $O(n^3 \log n)$ time each, as shown in Theorem 3.2.1. Thus ℓ calls will take $O(\ell \cdot n^3 \log n) = O(n^3 \log n)$. Steps 12-30 compute tuples for all connected subgraphs of S that contain the root. Join operations were shown to be performed in constant time (Proposition 3.2.4). As discussed above, a tuple, $D[\hat{S}, i, a]$, can be either of three types. Computation of each tuple, $D[\hat{S}, i, a]$, is dependant on the number of vertices in $\hat{S}$. As discussed earlier, for k vertices, there are $\binom{k+\ell-2}{\ell-1}$ distinct spiders with ℓ legs. Each of these spiders have k associated tuples, one for each vertex fixed as the last activated vertex. Thus, time to compute all tuples, $T(S)$, is given by

$$
\begin{aligned}
T(S) &= \sum_{k=1}^{n} k \cdot k \cdot \binom{k+\ell-2}{\ell-1} \\
&\leq \binom{n+\ell-2}{\ell-1} \sum_{k=2}^{n} k \cdot k \\
&= O(n^{\ell-1} \cdot n^3) \\
&= O(n^{\ell+2})
\end{aligned}
$$

Thus, the total running time is given by $O(n^3 \log n) + O(n^{\ell+2}) = O(n^{\ell+2})$ (assuming $\ell > 1$). □

Algorithm 6 CalculateSeedsetSpider(S, D)

INPUT: A spider, S and a lookup table of tuples, D, calculated via the method Spider_TD.
OUTPUT: A tuple containing a positive integer and a subset of vertices in S.

1: $(w(Y), Y) = \underset{\substack{D[S,i,a], \\ v_{i,a} \in S}}{\text{argmin}}\, w(D[S,i,a])$
2: **return** $\Big(w(Y), Y\Big)$

Pseudocode for CalculateSeedsetSpider(S, D)

Theorem 3.2.9. *TD on a spider with ℓ legs can be solved in $O(n^{\ell+2})$ time.*

Proof. Theorem 3.2.8 shows that we can compute the tuple lookup table D in $O(n^{\ell+2})$ time. The computation of the final seedset is done by the method CalculateSeedsetSpider(S, D), which calculates

$$(w(Y), Y) = \underset{\substack{D[S,i,a], \\ v_{i,a} \in S}}{\text{argmin}}\, w(D[S,i,a])$$

. This requires a $\Theta(1)$ lookup operation at the root of the min-heap associated with S. The result follows. □

Chapter 4

Polyhedral Characterization for Dynamic Programming Models on TD

Although we have a dynamic program to solve TD on paths and spiders in polynomial time, it is worthwhile to transfer this result to derive a polyhedral characterization of TD on paths and spiders.

In particular, we want to give a linear program whose optimal solutions can be mapped onto optimal solutions to our TD instance. For this purpose, we use the directed decision hypergraph paradigm in Martin et al. (1990).

4.1 The Paradigm

Let us first formally introduce the notions of a directed hypergraph.

Definition 4.1.1 (*Ausiello et al. (1998)*)**.** *A* ***directed hypergraph*** *is a pair, $H = (S, A)$, where S is a set of vertices (called states), and A is a set of hyperarcs. Each hyperarc is an ordered pair (H,t), from a non-empty set of states (called the head set), $H \subseteq S$, to a node, $t \in S$, called the tail node.*

Martin et al. (1990) model dynamic programming algorithms as flows in a simple directed hypergraph, $\mathbb{H} = (\mathbb{S} \cup \{\varnothing\}, \mathbb{A})$. The finite set of states, $\mathbb{S} \cup \{\varnothing\}$, represents the vertices of the hypergraph. S represents solutions to intermediate phases (corresponding to subproblems in the dynamic program) of the problem. The null state $\varnothing$ represents the empty solution. The

set of directed hyperarcs is denoted by $\mathbb{A}$. A hyperarc has the form (H,t), where head states $h \in H \subset \mathbb{S} \cup \{\varnothing\}$ are combined to obtain the tail state $t \in \mathbb{S}$. Hyperarcs represent decisions taken to combine solutions to a set of subproblems to arrive to a solution of a larger problem. A complete final solution corresponds to one unit of flow in $\mathbb{H}$ from $\varnothing$ to a final, *global* state, denoted by $\mathscr{o} \in \mathbb{S}$. The hypergraph is required to have the following properties.

1. *Acyclic Property* : The hypergraph is acyclic, i.e., there exists an integer-valued function, σ, on the set $\mathbb{S}$ that defines a total order on $\mathbb{S}$. Formally,

   $$\sigma : \mathbb{S} \to \{1,2,...,|\mathbb{S}|\}$$

   such that

   $$\sigma(h) < \sigma(t) \text{ for all } (H,t) \in \mathbb{A}; h \in H. \tag{4.1}$$

2. Every state $s \in \mathbb{S}$ has at least one incoming arc. With Equation 4.1, this means that we have hyperarcs $(H,t) \in \mathbb{A}$ such that $H = \varnothing$, for some $t \in \mathbb{S}$.

3. The hypergraph, $\mathbb{H}$, has a finite *reference set*, $\mathscr{R}$, associated with it. Each state $s \in \mathbb{S}$ is associated with a non-empty reference subset, $R[s] \subseteq \mathscr{R}$. These subsets need to satisfy the following properties -

   (a) *Consistency Property* : A partial order on elements contained in each reference subset that is consistent with the ordering σ on $\mathbb{S}$. Formally,

   $$R[h] \subseteq R[t] \text{ for all } (H,t) \in \mathbb{A}; h \in H. \tag{4.2}$$

   We require that $\mathscr{o}$ has $R[\mathscr{o}] = \mathscr{R}$.

   (b) *Disjointness Property* : Reference subsets of different heads of the same hyperarc have to be disjoint, i.e.,

   $$R[h_1] \cap R[h_2] = \varnothing \text{ for all } (H,t) \in \mathbb{A};\ h_1,h_2 \in H;\ h_1 \neq h_2. \tag{4.3}$$

   For a hyperarc $(H,t) \in \mathbb{A}$, where h_1 and h_2 are distinct elements in the head set H, the intersection of the reference sets $R[h_1]$ and $R[h_2]$ is null (not to be confused with the null state).

4. Every arc $(H,t) \in \mathbb{A}$ has a non-negative cost associated with it, denoted by $c[H,t]$.

4.2 Polyhedral Characterization

Consider a general instance, $\mathbb{H}$, of the hypergraph model described above. Let $OPT_{\mathbb{H}}$ denote a set of hyperarcs representing the set of optimal decisions. We define a variable, z, s.t.

$$z[H,t] = \begin{cases} 1 & \text{if } (H,t) \in OPT_{\mathbb{H}} \\ 0 & \text{otherwise} \end{cases}$$

for all hyperarcs $(H,t) \in \mathbb{A}$. The following linear program models the problem of finding a minimum cost flow of value 1 from the null state to the final state in $\mathbb{H}$.

$$\begin{array}{lll} \min & & \sum_{(H,t)\in\mathbb{A}} c[H,t] \cdot z[H,t] \\ \text{s.t.} & & \\ & & \sum_{(H,\not o)\in\mathbb{A}} z[H,\not o] = 1. \\ \text{for all } t_1 \in \mathbb{S} - \{\not o\} & \sum_{(H,t_1)\in\mathbb{A}} z[H,t_1] - & \sum_{(H,t_2)\in\mathbb{A} \text{ with } t_1 \in H} z[H,t_2] = 0 \\ \text{for all } (H,t) \in \mathbb{A} & & z[H,t] \geq 0 \end{array} \qquad \text{(PRIMAL)}$$

The objective function states that we need a set of hyperarcs of minimum cost. The first constraint represents the fact that exactly one hyperarc will lead to the optimal solution. The second constraint represents a flow constraint requirement. We use the fact that if a state, t_1, is in the optimal solution, then it has an incoming hyperarc with z value 1. We consider this to be incoming flow of 1 unit. Flow conservation is ensured by requiring that such a state must also send that one unit of flow out, on arcs of the form $(H,t_2) \in \mathbb{A}$, where $t_2 \in \mathbb{S}$. The final constraint imposes a non-negativity constraint on z values.

Martin et al. (1990) showed that these three constraints are enough to ensure the existence of an optimal binary solution vector to the above linear program. We now report the proof for completeness.

To prove this claim, we first take the dual of the above linear program. The dual multiplier for the first constraint is $u[\not o]$, and $u[h]$ is the dual multiplier for the $\sigma(h)^{th}$ row of the second constraint, where $h \in \mathbb{S} - \{\not o\}$. The dual linear program is given below.

$$\begin{aligned}
&\max && u[\circ] \\
&\text{s.t.} && \\
&\text{for all } (H,\circ) \in \mathbb{A} && u[\circ] - \sum_{h \in H} u[h] \leq c[H,\circ] \\
&\text{for all } (H,t) \in \mathbb{A},\ t \neq \circ && u[t] - \sum_{h \in H,\ h \neq \varnothing} u[h] \leq c[H,t]
\end{aligned} \tag{DUAL}$$

Feasible solutions to DUAL and PRIMAL are constructed as in Algorithm 7 and 8 respectively.

Algorithm 7 DUAL Construction

INPUT: A hypergraph, $\mathbb{H} = (\mathbb{S} \cup \{\varnothing\}, \mathbb{A})$, and a cost function, $c : \mathbb{A} \to \mathbb{R}_0^+$.
OUTPUT: A feasible solution for DUAL for $\mathbb{H}$.

1: **for all** $i \in 1, \ldots, |\mathbb{S}|$ **do**
2: Let $t \in \mathbb{S}$ be such that $\sigma(t) = i$.
3: $u^*[t] \leftarrow \min_{(H,t) \in \mathbb{A}} \{\, c[H,t] + \sum_{h \in H,\ h \neq \varnothing} u^*[h] \,\}$
4: **end for**

5: **return** u^*

Pseudocode for constructing a solution to DUAL.

Algorithm 7 computes dual multipliers for each row of the second and the third constraints of PRIMAL.

Algorithm 8 computes a solution to PRIMAL, z^*. We fist set all the z variables to 0 (Steps 1-3). Then, we associate a set H^*_δ, and a state $h^*_\delta \in H^*_\delta$ in that set with each depth δ. At depth $\delta = 0$, these are set as $\{\circ\}$ and $\circ$ respectively (Steps 4-5). When the selected state $h^*_\delta \in H^*_\delta$ is processed, we select a hyperarc $(H^*_{\delta+1}, h^*_\delta)$ such that

$$u^*[h^*_\delta] = c[H^*_{\delta+1}, h^*_\delta] + \sum_{h \in H^*_{\delta+1}} u^*[h] \tag{4.4}$$

and increment the corresponding $z^*[H^*_{\delta+1}, h^*_\delta]$ value and the depth (Steps 6-12). When there are no more elements remaining in the set H^*_δ for any depth δ, the algorithm returns to Step 6 process $H^*_{\delta-1}$ (Steps 13-16).

Theorem 4.2.1 (***Martin et al.*** **(*1990*)).** *The solutions z^* and u^* generated by the primal and dual construction algorithms, Algorithm 8 and Algorithm 7 respectively, are optimal in their respective problems, and z^* is integral.*

Proof. First, we prove that z^* and u^* constructed by primal and dual construction algorithms are feasible.

Consider DUAL. Steps 1-4 in Algorithm 7 are restatements of constraints of DUAL. The initialization of $u^*[t]$ in Step 3 is valid, because there is at least one incoming arc for all states $t \in \mathbb{S}$. Also, the acyclic ordering will ensure that the first state, will have arcs with null heads, i.e., $H = \varnothing$ for all $(H,t) \in \mathbb{A}$ and $\sigma(t) = 1$. Thus, the initialization of $u^*[1]$ will be dual feasible. As the next states in the ordering use u^* values of states with σ values lower than theirs, inductively, all u^* values are dual feasible, and thus u^* is a feasible solution to DUAL.

Now consider the PRIMAL. The initialization of $H^*_{\delta+1}$ in Step 9 such that

$$u^*[h^*_\delta] = c[H^*_{\delta+1}, h^*_\delta] + \sum_{h \in H^*_{\delta+1}} u^*[h] \tag{4.4}$$

is valid because there exists a $H^*_{\delta+1}$ which achieves the minimum in Step 3 of Algorithm 7, and we have proved above that Algorithm 7 generates a feasible solution to DUAL.

A solution that is primal feasible should satisfy the flow constraints of PRIMAL. Initialization of H^*_0 as $\{\mathcal{o}\}$ ensures that exactly one hyperarc with tail as $\mathcal{o}$, $(H,\mathcal{o}) \in \mathbb{A}$, sets $z^*[H,\mathcal{o}] = 1$, which is the first primal constraint. Each $z^*[H,t]$ incremented by 1 in Step 10 has its head, H, set as $H^*_{\delta+1}$, which is in the next depth, in Step 11. This ensures the that all states $h \in H$ maintain flow conservation. This satisfies the second primal constraint, by ensuring the flow balance at all depths. We have showed above that such a $H^*_{\delta+1}$ can always be found. Equality in Equation 4.4 ensures that the primal and dual construction algorithms return solutions, z^* and u^*, that satisfy complementary slackness conditions.

Now, we look at integrality. The primal solution, z^*, will always be a vector of positive integers, because all elements in z^* are initialized to 0, and incremented by 1 in Step 2 and 10 respectively in Algorithm 8. No other increments or decrements take place.

□

Having proved integrality and feasibility, we now prove that the hypergraph characterization solves our original optimization problem.

Algorithm 8 PRIMAL Construction

INPUT: A hypergraph, $\mathbb{H} = (\mathbb{S} \cup \{\varnothing\}, \mathbb{A})$, a cost function, $c : \mathbb{A} \to \mathbb{R}_0^+$ and a feasible solution to DUAL for $\mathbb{H}$.

OUTPUT: A feasible solution for PRIMAL for $\mathbb{H}$.

1: **for all** $(H,t) \in \mathbb{A}$ **do**
2: $\quad z^*[H,t] \leftarrow 0$
3: **end for**

4: $\delta \leftarrow 0$
5: $H_0^* \leftarrow \{o\}$

6: **while** H_δ^* is not empty and $H_\delta^* \neq \{\varnothing\}$ **do**
7: $\quad$ Pick $h_\delta^* \in H_\delta^*$
8: $\quad$ Update $H_\delta^* \leftarrow H_\delta^* \backslash h_\delta^*$
9: $\quad$ Pick $H_{\delta+1}^*$ such that
$$u^*[h_\delta^*] = c[H_{\delta+1}^*, h_\delta^*] + \sum_{h \in H_{\delta+1}^*} u^*[h]$$
10: $\quad z^*[H_{\delta+1}^*, h_\delta^*] \leftarrow z^*[H_{\delta+1}^*, h_\delta^*] + 1$
11: $\quad \delta \leftarrow \delta + 1$
12: **end while**

13: **if** $\delta > 0$ **then**
14: $\quad \delta \leftarrow \delta - 1$
15: $\quad$ Return to Step 6
16: **end if**

17: **return** z^*

Pseudocode for constructing a solution to PRIMAL.

Theorem 4.2.2 (*Martin et al.* (*1990*)). *The polytope given by the constraints of PRIMAL has extreme points as binary vectors.*

Proof. Theorem 4.2.1 shows that Algorithm 8 gives an integer optimal solution for every cost function, implying that each extreme point of the feasible region is an integral vector. If Step 7 does not pick the same state twice, then no $z[H^*_{\delta+1}, h^*_\delta]$ is incremented twice, and the solution would be binary, and correspond to a flow of one unit from the $\varnothing$ state to ϕ.

Let us assume that a state is picked twice. This occurs when the sequence of tail sets and states processed by Algorithm 8 has two sub-sequences of tail sets and states, $s_1 = h^*_0, H^*_0, h^*_1, H^*_1, ..., h^*_\delta$ and $s_2 = h^*_0, H^*_0, h^*_1, H^*_1, ..., h^*_l, H^*_l \overline{h}^*_\rho, \overline{H}^*_\rho, ..., \overline{h}^*_{\delta'}$ These sub-sequences first diverge at some depth ρ, by using different tail states, $\overline{h}^*_\rho \neq h^*_\rho$, and merge again, causing a repeated tail state, $\overline{h}^*_{\delta'} = h^*_\delta$.

Using the consistency property of the paradigm, we have

$$R[h^*_\delta] \subseteq R[h^*_{\delta-1}] \subseteq ... \subseteq R[h^*_{\rho+1}] \subseteq R[h^*_\rho]$$

and

$$R[\overline{h}^*_{\delta'}] \subseteq R[\overline{h}^*_{\delta'-1}] \subseteq ... \subseteq R[\overline{h}^*_{\rho+1}] \subseteq R[\overline{h}^*_\rho].$$

We know that $\overline{h}^*_{\delta'} = h^*_\delta$, which gives us $R[\overline{h}^*_{\delta'}] = R[h^*_\delta]$. Since the reference sets are non empty, we use the two equations above to get

$$R[h^*_\delta] = R[\overline{h}^*_{\delta'}] \subseteq R[\overline{h}^*_\rho] \cap R[h^*_\rho].$$

This is in violation of the disjointness property, as the states $h^*_\rho, \overline{h}^*_\rho \in H^*_\rho$, have intersecting reference sets. □

4.3 LP formulation for TD on Paths

We go back to technology diffusion on a path, $P = (V, E)$, with $V = \{v_1, ..., v_n\}$ and $E = \cup_{i\in[n-1]}\{(v_i, v_{i+1})\}$. Let us look at our dynamic programming formulation again.

$$D[i, j, l] = \underset{\substack{D[i,l-1,i'],\\ i'\in[i...l-1]}}{\operatorname{argmin}} w(D[i, l-1, i']) + \underset{\substack{D[l+1,j,j'],\\ j'\in[l+1...j]}}{\operatorname{argmin}} w(D[l+1, j, j']) + \alpha_{ijl} \quad (3.1)$$

where $\alpha_{ijl} = (w(v_l), \{v_l\})$ if $j - i + 1 < \theta(v_l)$, and $(0, \varnothing)$ otherwise. We represent it in the hypergraph paradigm as follows. Let our hypergraph instance be $\mathbb{H}_p = (\mathbb{S}_p \cup \{\varnothing\}, \mathbb{A}_p)$. We define the properties of $\mathbb{H}_p$ below.

1. The set of states has a state for all entries required in Equation 3.1, i.e., every $D[i,j,l]$ for $1 \leq i \leq l \leq j \leq n$. Additionally, it has a global node to represent the optimal solution, $\ominus_p$, i.e.,

$$\mathbb{S}_p = \{(i,j,l) : 1 \leq i \leq l \leq j \leq n\} \cup \{\ominus_p\}.$$

2. The set of arcs is in the form of (H,t) such that H is the set of path fragments needed to form path fragment t. If $t = (i,j,l)$, which corresponds to the path fragment $(v_i,...,v_j)$ in P, H is the set of states that correspond to path fragments $(v_i,...,v_{l-1})$ and $(v_{l+1},...,v_j)$, i.e., $(i,l-1,q)$ and $(l+1,j,q')$, where $1 \leq i \leq q < l < q' \leq j \leq n$. If $l = i$, the first fragment does not exist. If $l = j$, the second fragment does not exist. In such cases, H contains one state, corresponding to the existing fragment. If $t = (i,i,i)$, then $H = \{\varnothing\}$, as single node fragments cannot be composed of smaller fragments. If $t = \ominus_p$, then the set H corresponds to states representing the entire path, $(v_1,...,v_n)$, i.e., $H = \{(1,n,l)\}$, where $1 \leq l \leq n$.

$$\begin{aligned}
\mathbb{A}_p = &\Big\{(H,t) : H = \{(i,l-1,q),(l+1,j,q')\},\ t = (i,j,l),\ 1 \leq i \leq q < l < q' \leq j \leq n\Big\} \\
&\cup \Big\{(H,t) : H = \{(i,j-1,q)\},\ t = (i,j,j),\ 1 \leq i \leq q < j \leq n\Big\} \\
&\cup \Big\{(H,t) : H = \{(i+1,j,q)\},\ t = (i,j,i),\ 1 \leq i < q \leq j \leq n\Big\} \\
&\cup \Big\{(H,t) : H = \{\varnothing\},\ t = (i,i,i),\ 1 \leq i \leq n\Big\} \\
&\cup \Big\{(H,t) : H = \{(1,n,l)\},\ t = \ominus_p,\ 1 \leq l \leq n\Big\}.
\end{aligned}$$

   Note that the global state, $\ominus_p$, has no outgoing arc.

3. We sort the states into buckets corresponding to the size of the fragments they represent. We begin assigning an order in ascending order of the fragment size, which is given by the difference between the second and the first coordinate of a state plus 1. In each bucket, we can pick any ordering on the states, $\sigma_P : \mathbb{S}_p \rightarrow \mathbb{R}_+$, such that σ_P obeys the lexicographic ordering of the state coordinates, with $\sigma_p(\ominus_p)$ as the largest value. Formally,

$$\sigma_P(i,j,l) < \sigma_P(i',j',l)$$

   if

   (a) $j - i < j' - i'$, or,
   (b) $j - i = j' - i'$ and $i < i'$, or,
   (c) $j - i = j' - i'$, $i = i'$, $j = j'$, and $l < l'$.

The buckets will ensure that smaller fragments have smaller σ_P values, thus ensuring that sub-fragments used to make larger fragments will obey the acyclic property.

4. We define the reference subset of a state, $s \in \mathbb{S}_p$, as the set of vertices in the fragment it represents.
$$R[s] = \begin{cases} \{1,...,n\} & \text{if } s = \sigma_p \\ \{i,...,j\} & \text{if } s = (i,j,l) \end{cases}$$
As the head of each arc contains states representing disjoint fragments, we have the disjointness property. As the tail state represents a fragment joining these head fragments, we get consistency.

5. The cost of a hyperarc represents whether the last activated vertex in the tail fragment, i.e., last coordinate of the tail state, was a seed or not. Arcs incoming to the global state have no cost.
$$c_p(H,t) = \begin{cases} w(v_l) & \text{if } t = (i,j,l) \text{ and } j-i+1 < \theta(v_l) \\ 0 & \text{if } t = (i,j,l) \text{ and } j-i+1 \geq \theta(v_l) \\ 0 & \text{if } t = \sigma_p \end{cases}$$
This function ensures that the hyperarcs used to build the solution have a combined cost equal to the weight of the seeds used in the solution.

We have the following lemmas about the size of the hypergraph, $\mathbb{H}_p$.

Lemma 4.3.1. *$\mathbb{H}_p$ has $O(n^3)$ states.*

Proof. There is one state to represent the optimal solution, and one to represent the null state. All other states have the form (i,j,l), where $1 \leq i \leq l \leq j \leq n$. Thus, there are $O(n^3)$ such states. This is the same as the number of tuple entries required for our dynamic program for TD on paths. □

Lemma 4.3.2. *$\mathbb{H}_p$ has $O(n^5)$ hyperarcs.*

Proof. Let a hyperarc, (H,t). We have the following cases:

1. The tail state is $t = (i,j,l)$ such that $i \neq l \neq j$.
There are $O(n^3)$ such states.We know that the head states are the set $H = \{(i,l-1,q),(l+1,j,q')\}$, where $1 \leq i \leq q < l < q' \leq j \leq n$. There are $l-i$ states of the form $(i,l-1,q)$ and $j-l$ states of the form $(l+1,j,q')$. Thus there are $(l-i)\cdot(j-l) < n^2$ possibilities for H. This case will contribute $O(n^3)\cdot n^2 = O(n^5)$ arcs.

2. The tail state is $t = (i, j, l)$ such that $i = l$.
   There are $O(n^2)$ such states. We know that the head states are the set $H = \{(i+1, j, q)\}$, where $1 \leq i < q' \leq j \leq n$. There are $(j-i)$ states of the form $(i+1, j, q)$. Thus, there are $j - i < n$ possibilities for H. This case contributes $O(n^2) \cdot n = O(n^3)$ arcs.

3. The tail state is $t = (i, j, l)$ such that $j = l$.
   There are $O(n^2)$ such states. We know that the head states are the set $H = \{(i, j-1, q)\}$, where $1 \leq i \leq q < j \leq n$. There are $j - i$ states of the form $(i, j-1, q)$. Thus, there are $j - i < n$ possibilities for H. This case contributes $O(n^2) \cdot n = O(n^3)$ arcs.

4. The tail state is $t = (i, i, i)$.
   There are n such states. In this cases, the head state is $\varnothing$.

5. The tail is σ_p. The head set is $H = \{(1, n, l)\}$ where $1 \leq l \leq n$. There are n possibilities for H. This case will contribute n hyperarcs.

Thus, there are $O(n^5)$ hyperarcs in $\mathbb{H}_p$. □

The construction of $\mathbb{H}_p$ also gives us the following:

Theorem 4.3.3. *Let (P, w, θ) be an instance of TD on a path, P, and $(\mathbb{H}_p, c_p)$ the corresponding weighted hypergraph. Then the cost of an optimal solution to our TD instance is equal to the optimal value of PRIMAL for $(\mathbb{H}_p, c_p)$.*

Proof. The solution to PRIMAL, z^*, gives a sequence of hyperarcs, $m = (H_1, t_1), (H_2, t_2) \ldots, (H_\rho, t_\rho)$ in $\mathbb{H}_p$, where $H_1 = \{\varnothing\}$ and $t_\rho = \sigma_p$ and $z[H_i, t_i] = 1$ for all $i \in [\rho]$. Let us see what the tail states in the hyperarcs in m represent in our TD instance.

For every hyperarc $(H, t) \in m$, we consider the tail state $t = (i, j, l)$. In our TD instance, we construct a solution by activating state v_l at time step $j - i + 1$, as the state represents a fragment $(v_i, \ldots, v_l, \ldots, v_j)$ where v_l is activated last. Every arc has a cost, $c_p(H, t)$. In our TD solution, we pay a cost of $w(v_l)$ for activation of v_l if the fragment size, $j - i + 1$, is less than the threshold value of v_l, $\theta(v_l)$, because that means that v_l is not connected to enough active nodes to be activated, and has to be activated as a seed. If $j - i + 1 \geq \theta(v_l)$ then v_l can be activated without being set as a seed, and hence we pay no cost. Thus, the cost of our solution is equal to the cost of m.

Let us take a solution, $(w(Y), Y)$, of a TD instance returned by Algorithm 3. Each solution is built up of tuples associated with subgraphs in P. Note that all entries in the lookup table, D, used to build the solution output by this algorithm can be mapped to states in $\mathbb{H}_p$, because

all entries have a corresponding state in $\mathbb{H}_p$. Moreover, these states together form a sequence of hyperarcs, m', in $\mathbb{H}_p$, because the entry $D[i,j,l]$ for each segment (corresponding to a tail state, (i,j,l)) in D is calculated using entries of smaller segments, $D[i,l-1,q]$ and $D[l+1,i,q']$ (corresponding to the head set, $\{(i,l-1,q),(l+1,j,q')\}$, where $1 \leq i \leq q < l < q' \leq j \leq n$) that combine to form the segment, $(v_i,...,v_l,...,v_j)$. To complete m' as a solution, we also add the hyperarcs $(\{\varnothing\},(i,i,i))$ and $(\{(1,n,l)\},\not{o}_p)$ to m', where we have used the tuples $D[i,i,i]$ and $D[i,n,l]$ to form the $(w(Y),Y)$. For every tuple $D[i,j,l]$ used to calculate Y, v_l is added to Y if $j-i+1 < \theta(v_l)$. The cost of m' is incremented by $w(v_l)$ if m' includes a state (i,j,l) and $j-i+1 < \theta(v_l)$. This makes the cost of m' same as that of Y. □

An example hypergraph construction for a TD instance on a path of three nodes is given below.

4.3.1 Example

Consider a TD instance on a path of 3 nodes, shown in Figure 4.1. All nodes have weights as 1, and the thresholds are $\{\theta(v_1)=3, \theta(v_2)=2, \theta(v_3)=3\}$. We build the hypergraph $\mathbb{H}_p = (\mathbb{S}_p \cup \{\varnothing\}, \mathbb{A}_p)$ as follows (See Figure 4.3).

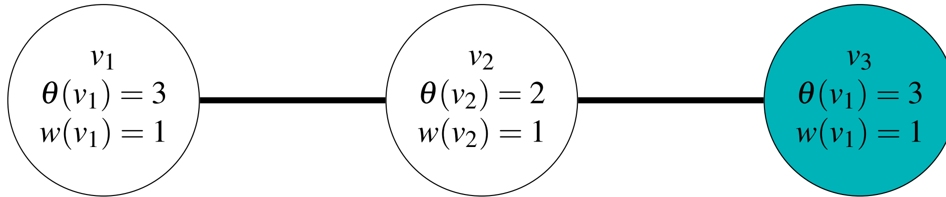


Figure 4.1: A TD instance on a path of three nodes.

Clearly, if v_3 is set as a seed, the path is fully activated.

$$\begin{aligned}\mathbb{S}_p = \{&(1,1,1),(2,2,2),(3,3,3),\\ &(1,2,1)(1,2,2),(2,3,2),(2,3,3),\\ &(1,3,1)(1,3,2),(1,3,3),\not{o}_p\}\end{aligned}$$

$$
\begin{aligned}
\mathbb{A}_p = \Big\{ & \big(\ \{(2,2,2)\},\ (1,2,1)\ \big),\\
& \big(\ \{(1,1,1)\},\ (1,2,2)\ \big),\\
& \big(\ \{(2,3,2)\},\ (1,3,1)\ \big),\\
& \big(\ \{(2,3,3)\},\ (1,3,1)\ \big),\\
& \big(\ \{(3,3,3)\},\ (2,3,2)\ \big),\\
& \big(\ \{(2,2,2)\},\ (2,3,3)\ \big),\\
& \big(\ \{(1,2,1)\},\ (1,3,3)\ \big),\\
& \big(\ \{(1,2,2)\},\ (1,3,3)\ \big),\\
& \big(\ \{(1,1,1),(3,3,3)\},\ (1,3,2)\ \big),\\
& \big(\ \{(1,3,1)\},\ \not{o}_p\ \big),\\
& \big(\ \{(1,3,2)\},\ \not{o}_p\ \big),\\
& \big(\ \{(1,3,3)\},\ \not{o}_p\ \big),\\
& \big(\ \{\varnothing\},\ (1,1,1)\ \big),\\
& \big(\ \{\varnothing\},\ (2,2,2)\ \big),\\
& \big(\ \{\varnothing\},\ (3,3,3)\ \big)\Big\}
\end{aligned}
$$

We give the arc costs in Table 4.1, and the dual variables set according to Algorithm 7 in Table 4.2.

4.4 LP formulation for TD on Spiders

Here, we formulate a dynamic program for TD in the case where the input graph is a spider $S[\beta_1,...,\beta_\ell] = (V,E)$ with n nodes, and $\ell < n$ legs, which are indexed by $[\ell]$, where ℓ is a constant positive integer. Recall that the leg lengths are denoted by a set of ℓ non-negative integers, $\{\beta_1,....,\beta_\ell\}$. A vertex is represented as $v_{i,a}$, and the subscript indicates that it is the ith vertex from the root in the leg with index a, $a \in \{1,...,\ell\}$. The root of the spider is denoted by $v_{0,0}$.

Each leg is represented by the path (V_a,E_a) for all $a \in [\ell]$. The vertex set of each spider leg is given by $V_a := \{v_{1,a},...,v_{\beta_a,a}\}$ for all legs $a \in [\ell]$. The edge set of each spider leg is given by $E_a := \{(v_{j,a},v_{j+1,a}) \mid j \in [\beta_a - 1]\}$ for all legs $a \in [\ell]$. Note that the vertex set is the union of the root vertex and the vertices of the legs, i.e., $V = \{v_{0,0}\} \cup V_1 \cup V_2 \cup ... \cup V_\ell$. The edge set of the spider is given by $E = E_1 \cup ... \cup E_\ell \cup \{(v_{0,0},v_{1,1}),...,(v_{0,0},v_{1,\ell})\}$. We have weights on the vertices

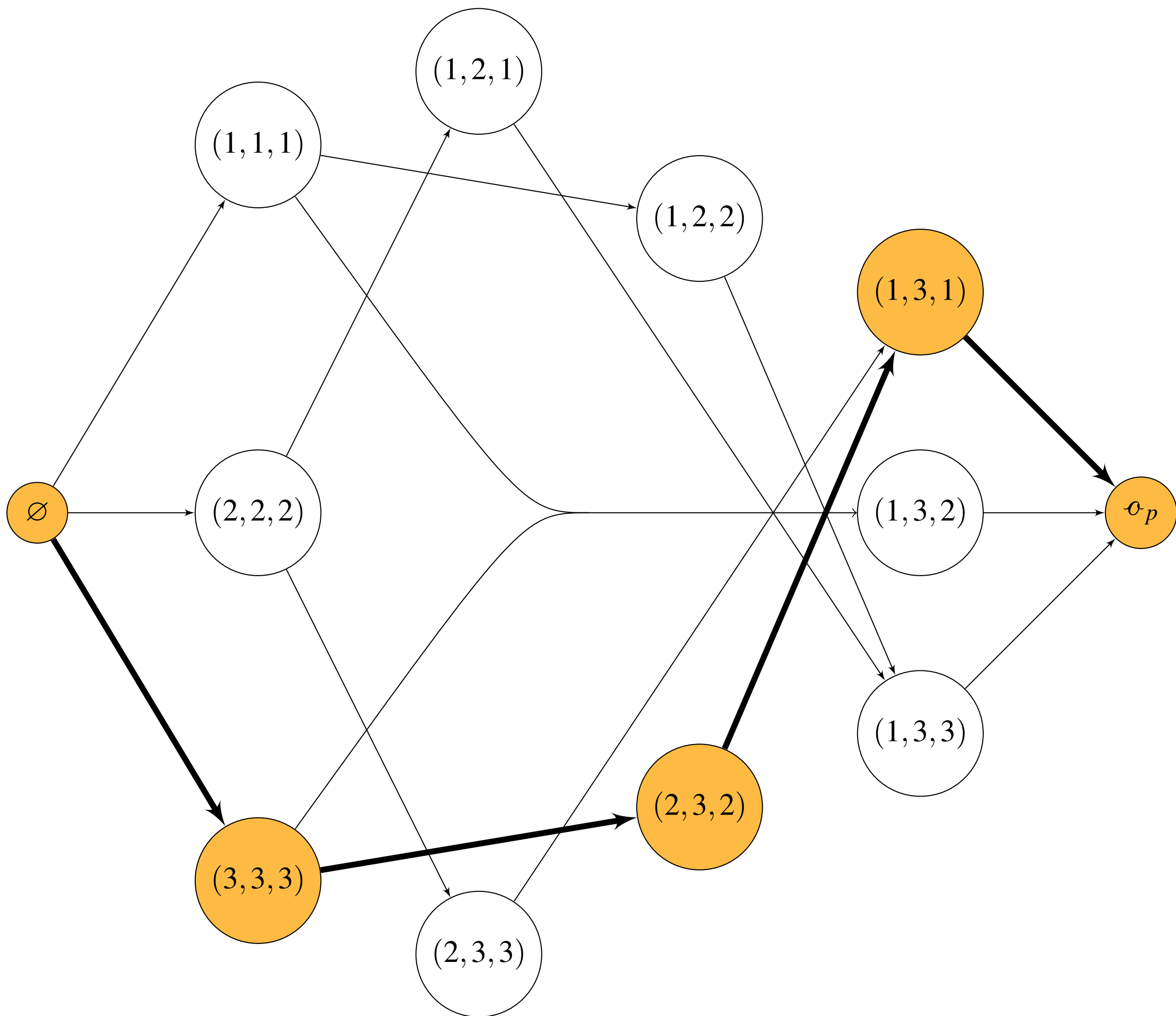


Figure 4.3: Example for Figure 4.1.

The yellow states and bold hyperarcs indicate the solution to the TD instance. The seedset is given by the last activated nodes in the tails of hyperarcs in the solution which have non-zero cost. In this instance, the seedset is $\{v_3\}$.

Table 4.1: Arc Costs for Figure 4.3.

	$\mathbb{A}_p$	c_p
1	$\big(\,\{(2,2,2)\},\ (1,2,1)\,\big)$	1
2	$\big(\,\{(1,1,1)\},\ (1,2,2)\,\big)$	0
3	$\big(\,\{(2,3,2)\},\ (1,3,1)\,\big)$	0
4	$\big(\,\{(2,3,3)\},\ (1,3,1)\,\big)$	0
5	$\big(\,\{(3,3,3)\},\ (2,3,2)\,\big)$	0
6	$\big(\,\{(2,2,2)\},\ (2,3,3)\,\big)$	1
7	$\big(\,\{(1,2,1)\},\ (1,3,3)\,\big)$	0
8	$\big(\,\{(1,2,2)\},\ (1,3,3)\,\big)$	0
9	$\big(\,\{(1,1,1),(3,3,3)\},\ (1,3,2)\,\big)$	0
10	$\big(\,\varnothing,\ (1,1,1)\,\big)$	1
11	$\big(\,\varnothing,\ (2,2,2)\,\big)$	1
12	$\big(\,\varnothing,\ (3,3,3)\,\big)$	1
13	$\big(\,(1,3,1),\ \not{o}_p\,\big)$	0
14	$\big(\,(1,3,2),\ \not{o}_p\,\big)$	0
15	$\big(\,(1,3,3),\ \not{o}_p\,\big)$	0

Table 4.2: Dual Variables for Figure 4.3.

$\boldsymbol{\sigma}_p$	$\mathbb{S}_p$	u^*
1	$(1,1,1)$	1
2	$(2,2,2)$	1
3	$(3,3,3)$	1
4	$(1,2,1)$	2
5	$(1,2,2)$	1
6	$(2,3,2)$	1
7	$(2,3,3)$	2
8	$(1,3,1)$	1
9	$(1,3,2)$	2
10	$(1,3,3)$	1
11	$\not{o}_p$	1

assigned by the function $w : V \to \mathbb{R}_+$, and a threshold function $\theta : V \to \{2,..,n\}$. We use the notation $\theta(i,a)$ to represent $\theta(v_{i,a})$ for all $v_{i,a} \in V$. Let a path, $(v_{i,a},...,v_{j,a})$, be represented by $P[i,j,a]$.

We map our dynamic programming model on a TD instance on a spider in the hypergraph paradigm as follows. Let our hypergraph instance be $\mathbb{H}_s = (\mathbb{S}_s \cup \{\varnothing\}, \mathbb{A}_s)$. We define the properties of $\mathbb{H}_s$ below.

1. The set of states has a state for all entries in the tuple lookup table, D, used in the dynamic programming solution of TD in spiders, i.e., we have a state for every entry $D[G,i,j]$ such that $G = (V',E')$ is a connected subgraph in S and $v_{i,j}$ is a vertex in G. We also have a global node to represent the optimal solution, σ_s.

$$\mathbb{S}_s = \{(G,i,j) : G = (V',E') \text{ is a connected subgraph in } S,\ v_{i,j} \in V'\} \cup \{\sigma_s\}.$$

2. Hyperarcs are in the form (H,t) such that H is the set of subgraphs needed to form subgraph t. If $t = (\hat{S}[b_1,...,b_j,...,b_\ell],i,j)$, with $i < b_j$, which corresponds to the spider $\hat{S}[b_1,...,b_j,...,b_\ell]$ in S, H is the set of states that correspond to a smaller spider $S'[b_1,...,(i-1),...,b_\ell]$ and a path, $P[i+1,b_j,j]$, i.e., $(S'[b_1,...,(i-1),...,b_\ell],i',j')$ for some $v_{i',j'} \in S'$ and $(P[i+1,b_j,j],i'',j)$, for some $0 < i < i'' \le b_j$.

   If the last activated vertex in the tail state is at the end of a leg, i.e., if $t = (\hat{S}[b_1,...,b_j,...,b_\ell],b_j,j)$, the set H contains one state, corresponding to the spider $S'[b_1,...,b_j-1,...,b_\ell]$.

   If the last activated vertex in the tail state is the root, i.e., if $t = (\hat{S}[b_1,...,b_\ell],0,0)$, the set H contains at most ℓ states, each corresponding to a path starting at the first vertex of a leg in $\hat{S}$. i.e., $H = \{(P[1,b_j,j] :\ \forall j : b_j > 0\}$.

   If we have a tail state which corresponds to a path, $(v_{a,j},...,v_{c,j})$ in S, i.e., $t = (P[a,c,j],l,j)$, the set H contains states corresponding to smaller paths, $(v_{a,j},...,v_{l-1,j})$ and $(v_{l+1,j},...,v_{c,j})$, i.e., the states $(P[a,l-1,j],l_1,j)$ and $(P[l+1,c,j],l_2,j)$, for some $0 \le a \le l_1 < l < l_2 \le c \le \beta_j$. If the last activated vertex, l, in the path is at either end of the path, then the set H will have just one state, because in this case the last activated vertex does not split the path into two smaller disjoint sub-paths.

   If the tail state is a single vertex in S, i.e., $t = (v_{i,j},i,j)$, then $H = \{\varnothing\}$, as single node subgraphs cannot be composed of smaller subgraphs. If $t = \sigma_p$, then the set H corresponds to states representing the entire spider, $S[\beta_1,...,\beta_\ell]$, i.e., $H = \{(S[\beta_1,...,\beta_\ell],i,j)\}$, for some $v_{i,j} \in S$.

$$\mathbb{A}_p = \Big\{(H,t) : H = \{(S'[b_1, ..., (i-1), ...b_\ell], i', j'), (P[i+1, b_j, j], i'', j)\},$$
$$t = (\hat{S}[b_1, ..., b_j, ..., b_\ell], i, j),\ v_{i',j'} \in \hat{S},\ 0 < i < i'' \leq b_j,\ 0 \leq b_k \leq \beta_k\ \forall k \in [\ell]\Big\}$$
$$\cup \Big\{(H,t) : H = \{(S'[b_1, ..., b_j - 1, ..., b_\ell], i', j')\},\ t = (\hat{S}[b_1, ..., b_j, ..., b_\ell], b_j, j),$$
$$v_{i',j'} \in S',\ 0 < b_k \leq \beta_k\ \forall k \in [\ell]\Big\}$$
$$\cup \Big\{(H,t) : H = \{(P[1, b_j, j] :\ \forall j : b_j > 0\},\ t = (\hat{S}[b_1, ..., b_\ell], 0, 0), 0 \leq b_k \leq \beta_k\ \forall k \in [\ell]\Big\}$$
$$\cup \Big\{(H,t) : H = \{(P[a, l-1, j], l_1, j), (P[l+1, c, j], l_2, j)\},\ t = (P[a, c, j], l, j),$$
$$0 \leq a \leq l_1 < l < l_2 \leq c \leq \beta_j,\ j \in [\ell]\Big\}$$
$$\cup \Big\{(H,t) : H = \{(P[a, l-1, j], l_1, j)\},\ t = (P[a, l, j], l, j),\ 0 \leq a \leq l_1 < l \leq \beta_j,\ j \in [\ell]\Big\}$$
$$\cup \Big\{(H,t) : H = \{(P[l+1, c, j], l_2, j)\},\ t = (P[l, c, j], l, j),\ 0 \leq l < l_2 \leq c \leq \beta_j,\ j \in [\ell]\Big\}$$
$$\cup \Big\{(H,t) : H = \{\varnothing\},\ t = (v_{i,j}, i, j),\ v_{i,j} \in S\Big\}$$
$$\cup \Big\{(H,t) : H = \{(S[\beta_1, ..., \beta_\ell], i, j)\},\ t = \phi_p,\ v_{i,j} \in S\Big\}.$$

Note that the global state, ϕ_p, has no outgoing arc.

3. We sort the states into buckets corresponding to the number of vertices in the subgraphs they represent. In each bucket, we can pick any ordering on the states, $\sigma_s : \mathbb{S}_s \to \mathbb{R}_+$ such that it obeys the lexicographic ordering of the state coordinates, with $\sigma_s(\phi_s)$ as the largest value. Formally, if we have two connected subgraphs of S, $\hat{G} = (\hat{V}, \hat{E})$ and $G' = (V', E')$, and $v_{i,j} \in \hat{V}$ and $v_{i',j'} \in V'$.

$$\sigma_S(\hat{G}, i, j) < \sigma_S(G', i', j')$$

if

(a) $|\hat{V}| < |V'|$, or,
(b) $|\hat{V}| = |V'|$ and $i < i'$, or,
(c) $|\hat{V}| = |V'|$, $i = i'$ and $j < j'$.

However, if $|\hat{V}| = |V'|$, $i = i'$ and $j = j'$, there are three possible cases.

(a) $\hat{G}$ and G' are paths of the form $P[\hat{a},\hat{c},j]$ and $P[a',c',j]$, where i is some integer in $[\hat{a},\hat{c}] \cap [a',c']$. Note that both paths have to be subgraphs in the same leg, as $j = j'$. Here, $\sigma_S(\hat{G},i,j) < \sigma_S(G',i',j')$ if $\hat{a} < a'$.

(b) $\hat{G}$ is a path, and G' is a spider, of the forms $P[\hat{a},\hat{c},j]$ and $S'[b'_1,...,b'_\ell]$ respectively, where $v_{i,j} \in \hat{V} \cap V'$. Here, $\sigma_S(\hat{G},i,j) < \sigma_S(G',i',j')$ if

  i. any of the first $j-1$ legs of S' have non-zero length, i.e., there exists $b'_m > 0$ for some $m < j$, or,

  ii. $b'_m = 0$ for all $m < j$ and $\hat{c} - \hat{a} + 1 \leq b_j$.

(c) $\hat{G}$ and G' are spiders, denoted by $\hat{S}[\hat{b_1},..,\hat{b_\ell}]$ and $S'[b'_1,...,b'_\ell]$, where $v_{i,j} \in \hat{V} \cap V'$. Here, $\sigma_S(\hat{G},i,j) < \sigma_S(G',i',j')$ if there is some $m \leq \ell$ such that $\hat{b_c} = b'_c$ for all $c < m$ and $\hat{b_m} < b'_m$.

The buckets will ensure that smaller subgraphs have smaller values, thus ensuring that subgraphs used to make larger subgraphs will obey the acyclic property.

4. We define the reference subset of a state, $s \in \mathbb{S}_s$, as the set of vertices in the subgraph it represents.

$$R[s] = \begin{cases} V & \text{if } s = \phi_s \\ V' & \text{if } s = (G'(V',E'),i,j) \end{cases}$$

As the head of each arc contains states representing disjoint subgraphs, we have the disjointness property. As the tail state represents a subgraph joining these smaller head subgraphs, we get consistency.

5. The cost of a hyperarc represents whether the last activated vertex in the tail subgraph, i.e., last coordinate of the tail state, is a seed or not. Arcs incoming to the global state have no cost.

$$c_s(H,t) = \begin{cases} w(v_{i,j}) & \text{if } t = (G'(V',E'),i,j) \text{ and } |V'| < \theta(v_{i,j}) \\ 0 & \text{if } t = (G'(V',E'),i,j) \text{ and } |V'| \geq \theta(v_{i,j}) \\ 0 & \text{if } t = \phi_s \end{cases}$$

This function ensures that the hyperarcs used to build the solution have a combined cost equal to the weight of the seeds used in the solution.

We have the following lemmas about the size of $\mathbb{H}_s$.

Lemma 4.4.1. *$\mathbb{H}_s$ has $O(n^{\ell+1})$ states.*

Proof. Let us first consider states corresponding to a subgraph that contains the root node of the spider, $S[\beta_1,...,\beta_\ell]$. As discussed earlier, there are $\binom{k+\ell-2}{\ell-1}$ unique spiders with k nodes and at most ℓ legs. Subgraphs that do not contain the root node are paths. A path, $v_{i,a},...,v_{j,a}$, is uniquely defined by its terminal vertices, $v_{i,a}$ and $v_{j,a}$. Each pair of terminal vertices has to belong to the same leg of the spider, S. This gives us at most $\beta_1^2+...+\beta_\ell^2$ unique paths. For a subgraph of size k, there are k different states, as there are k choices for the last activated vertex. We also have one node to represent the optimal solution, $\mathscr{o}_s$. Finally, we get :

$$
\begin{aligned}
|\mathbb{S}_s| &\leq \sum_{k=1}^{n} \binom{k+\ell-2}{\ell-1} \cdot k + \sum_{i=1}^{\ell} \beta_i^2 \cdot \beta_i + 1 \\
&\leq \binom{n+\ell-2}{\ell-1} \sum_{k=1}^{n} k + n^3 + 1 \\
&= O(n^{\ell-1} \cdot n^2 + n^3) \\
&= O(n^{\ell+1})
\end{aligned}
$$

assuming that $\ell > 1$. The second inequality follows from the fact that $\sum_{i=1}^{\ell} \beta_i = n-1$. $\square$

Lemma 4.4.2. *$\mathbb{H}_s$ has $O(n^{2\ell+1})$ hyperarcs.*

Proof. Let us consider a hyperarc (H,t). If the tail state corresponds to a subgraph containing a single node in S, then there is just one choice for the head set - $\{\varnothing\}$. This gives us n hyperarcs.

If the tail state is $\mathscr{o}_s$, then are n choices for the head set, one each for every vertex in S set as the last activated vertex. This gives us an additional n hyperarcs.

Now, let $t = (G,i,j)$, where $G = (V',E')$ is a connected subgraph in S, and $v_{i,j} \in V'$. If $v_{i,j} \neq v_{0,0}$, the set H contains two (or one) states corresponding to connected subgraphs obtained by splitting G at $v_{i,j}$. These subgraphs are unique. The head states can have the last activated vertices set as any vertex in $V' - \{v_{i,j}\}$. Thus, for each such tail state, there are at most $(|V'|-1)\cdot(|V'|-2) < |V'|^2$ choices for the head set.

If $v_{i,j} = v_{0,0}$, the set H contains at most ℓ states corresponding to disjoint legs obtained by splitting G at the root. These legs are unique. Let $G = \hat{S}[b_1,...,b_\ell]$. The head states can have the last activated vertices set as any vertex in $V' - \{v_{i,j}\}$. Specifically, for the kth leg, the last activated vertex can be any one of b_k choices. Thus, for each such tail state, there are at most $\Pi_{k=1}^{\ell} \max\{1,b_k\} \leq |V'|^\ell$ choices for the head set. Using the argument in Lemma 4.4.1 to count tail states, we get :

$$
\begin{aligned}
|\mathbb{A}_s| &\leq \sum_{k=2}^{n} \binom{k+\ell-2}{\ell-1} \cdot k \cdot k^{\ell} + \sum_{i=1}^{\ell} \beta_i^3 \cdot (\beta_i - 1)(\beta_i - 2) + n + n \\
&\leq \binom{n+\ell-2}{\ell-1} \sum_{k=2}^{n} k^{\ell+1} + n^5 + 2n && [\sum_{i=1}^{\ell} \beta_i = n-1] \\
&= O(n^{\ell-1} \cdot n^{\ell+2} + n^5) \\
&= O(n^{2\ell+1})
\end{aligned}
$$

assuming $\ell > 1$. □

The construction above gives us the following:

Theorem 4.4.3. *Let (S, w, θ) be an instance of TD on a spider, S, and $(\mathbb{H}_s, c_s)$ the corresponding weighted hypergraph. Then the cost of an optimal solution to our TD instance is equal to the optimal value of PRIMAL for $(\mathbb{H}_s, c_s)$.*

Proof. The solution to PRIMAL, z^*, gives a sequence of hyperarcs, $m = (H_1, t_1), (H_2, t_2) \ldots, (H_\rho, t_\rho)$ in $\mathbb{H}_s$, where $H_1 = \{\varnothing\}$ and $t_\rho = o_s$ and $z[H_i, t_i] = 1$ for all $i \in [\rho]$. Let us see what the tail states in the hyperarcs in m represent in our TD instance.

For every hyperarc $(H, t) \in m$, we consider the tail state $t = (G', i, j)$. In our TD instance, we construct a solution by activating node $v_{i,j}$ at time $|V'|$, as the state represents a subgraph of S, $G' = (V', E')$, where $v_{i,j}$ is activated last. Every arc has a cost, $c_s(H, t)$. In our TD solution, we pay a cost of $w(v_{i,j})$ for activation of $v_{i,j}$ if the number of vertices in subgraph G', $|V'|$, is less than the threshold value of $v_{i,j}$, $\theta(v_{i,j})$, because that means that $v_{i,j}$ is not connected to enough active nodes to be activated, and has to be activated as a seed. If $|V'| \geq \theta(v_{i,j})$ then $v_{i,j}$ can be activated without being set as a seed, and hence we pay no cost. Thus the cost of our solution is equal to the cost of m.

Let us take a solution to a TD instance, $(w(Y), Y)$, returned by Algorithm 6. Each solution is built up of tuples associated with subgraphs in S. Note that all entries in the lookup table, D, used to build the solution to this algorithm can be mapped to states in $\mathbb{H}_s$, as all entries have a corresponding state in $\mathbb{H}_s$. Moreover, these states together form a sequence of hyperarcs, m', in $\mathbb{H}_s$, because the entry, $D[\hat{S}[b_1, \ldots, b_\ell], i, j]$ for each subgraph (corresponding to a tail state, $(S[b_1, \ldots, b_\ell], i, j))$ in D is calculated using entries of smaller subgraphs, $D[S'[b_1, \ldots, (i-1), \ldots b_\ell], i', j']$ and $D[P[i+1, b_j, j], i'', j]$ (corresponding to the head set, $\{S'[b_1, \ldots, (i-1), \ldots b_\ell], i', j'),$ $(P[i+1, b_j, j], i'', j)\}$ where $v_{i',j'} \in S'$, $i < i'' < b_j$ and $0 \leq b_k \leq \beta_k \; \forall k \in [\ell]$), that combine

to form the segment, $\hat{S}[b_1,...,b_\ell]$. For every tuple $D[\hat{S}[b_1,...,b_\ell],i,j]$ used to calculate Y, $v_{i,j}$ is added to Y if $|\hat{S}| < \theta(v_{i,j})$, where $\hat{S} = (\hat{V},\hat{E})$. We also add the arcs $(\{\varnothing\},(v_{i,j},i,j))$ and $(\{(S[\beta_1,...,\beta_\ell],i',j')\},\multimap_s)$ to complete m' as a solution to the TD instance, where $D[(v_{i,j}),i,j]$ and $D[S[\beta_1,...,\beta_\ell],i',j]$ were tuples used to form $(w(Y),Y)$. The cost of m' is incremented by $w(v_{i,j})$ if m' includes a state (i,j,l) and $|\hat{S}| < \theta(v_{i,j})$. This makes the cost of m' same as that of Y. □

Chapter 5

Maximization of Diffusion under Constraints on Seeds

We here consider the maximization version of the Technology Diffusion problem, referred to as *maxTD*. In an instance of maxTD, we are given a graph, $G = (V,E)$, a positive integer, $k \leq |V|$, and thresholds, $\theta(v) \in \{\theta_1,...,\theta_q\}$ for each node $v \in V$. Each of the thresholds is a positive integer in the set $\{2,...,|V|\}$. For this section, we assume that all vertices have equal weights , i.e., $w : V \rightarrow 1$. As in the Technology Diffusion problem, we consider dynamic processes in which each vertex $v \in V$ is either active or inactive, and where an inactive vertex v becomes active if, in the graph induced by it and the active vertices, v lies in a connected component of size at least $\theta(v)$. The goal in maxTD is now to find a seedset $Y \subseteq V$ of cardinality at most k of initially active vertices that maximizes the number of active nodes in the graph, i.e., $\sigma(Y)$, at the end of the diffusion process, where diffusion is the process of activating vertices not in the seedset.

maxTD

INPUT : A simple, undirected graph, $G = (V,E)$, a threshold function, $\theta : V \rightarrow \{2,...,|V|\}$, and a positive integer, $k \leq |V|$.

OUTPUT: A seedset $Y \subseteq V$ of cardinality at most k such that if every node in Y is activated at time step 0, the total number of nodes eventually activated is maximized.

5.1 Greedy Algorithm for maxTD

As we have discussed in Chapter 2, greedy algorithms have been used quite successfully to give (approximate) solutions to some diffusion problems in the localized setting. We here show that the natural greedy algorithm can perform quite badly on instances of maxTD.

Let us consider a natural greedy algorithm, Algorithm 9, for maxTD.

Algorithm 9 Greedy algorithm for maxTD

INPUT: A graph, $G = (V, E)$, a threshold function, $\theta : V \rightarrow \{2, ..., |V|\}$, and a positive integer, $k \leq |V|$.

OUTPUT: A subset of nodes, $Y \subseteq V$, such that $|Y| \leq k$.

1: $Y \rightarrow \varnothing$.
2: Select $u \in V$ such that $\sigma(Y \cup \{u\})$ is maximized.
3: $Y = Y \cup \{u\}$.
4: If $|Y| < k$, and there are inactive vertices remaining in G, go to Step 2.

5: **return** Y

Pseudocode for greedy algorithm for maxTD.

We now report an instance where the greedy algorithm for maxTD performs arbitrarily badly compared to an optimal algorithm.

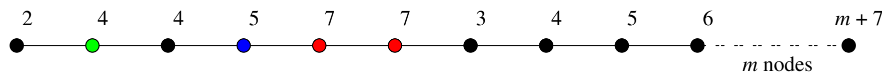


Figure 5.1: A maxTD instance with $k = 2$ on a path of $m + 11$ nodes. The node labels indicate thresholds.

Consider the maxTD instance in Figure 5.1 with $k = 2$ on a path of $m + 11$ nodes. The greedy algorithm will pick the green node as seed at the first iteration, hence activating the left most node with threshold 2. Then, it will pick the blue node as the second seed, activating the node between the two seeds with threshold as 4. No more nodes will be activated.

Note that the optimal solution will pick the two red nodes, both with thresholds 7 as seeds, and will activate the entire graph. This shows that the performance of the greedy algorithm can be arbitrarily bad.

5.2 Dynamic Programming for maxTD on a Path

Let us first study the problem on a graph that is a path. A maxTD instance on a path, (P, θ, k), where P is a path of the form $v_1, v_2, ..., v_n$, has the following property - if a vertex $v_i \in V$ is not activated by time unit t, then vertices in the path segments $v_1, ..., v_{i-1}$ and $v_{i+1}, ..., v_n$ are activated independently of each other before time t, i.e., no vertex in the first segment uses a vertex in the second segment to obtain its utility, and vice versa. If no such vertex exists, then the path can be activated completely by a seedset of cardinality at most k before time t, and this seedset can be calculated by Equation 2. This property can be exploited in a dynamic programming formulation.

Let $mD[i, j, k, l] := (A(Y),\ Y)$ be a tuple, where $1 \leq i \leq l \leq j \leq n$, containing (i) a seedset, Y, where Y has cardinality at most k(an input parameter), that activates the maximum number of vertices in the segment $v_i, ..., v_l, ..., v_j$, and (ii) the set of vertices activated by seedset Y, $A(Y)$, with the following property: if k is less than the required number of seeds to activate the segment $v_i, ..., v_l, ..., v_j$, the vertex v_l is not activated, else $A(Y)$ contains all vertices in the segment $v_i, ...v_l, .., v_j$. Note that with a large enough k, all vertices could be activated.

Furthermore, we define $A(mD[i, j, k, l]) := |A(Y)|$. We will use mD to refer to a lookup table for all $mD[i, j, k, l]$ tuple values. For segments of length 1, i.e., segments containing just one node, clearly, $i = j = l$. This gives us

$$mD[i,i,k,i] = \begin{cases} \Big(\{v_i\}, \{v_i\}\Big) & \text{if } k > 0 \\ \Big(\varnothing, \varnothing\Big) & \text{otherwise.} \end{cases}$$

For segments of length 2, i.e. edges of the type (v_i, v_{i+1}), note that l can take either i or $i+1$ as its value, as P is a path. Consider the value of $mD[i, i+1, k, l]$. Let us assume without loss of generality that $l = i$. Now, we have either of the four following cases :

1. $k = 0$
   Since we're allowed no seeds, no nodes can be activated. We set $mD[i, i+1, k, i] = \Big(\varnothing, \varnothing\Big)$.

2. $k \geq 1$ and $\theta(v_i) = 2$
   This means that v_i can be activated by v_{i+1}. Since we are allowed at least one seed, we set $mD[i, i+1, k, i] = \Big(\{v_i, v_{i+1}\}, \{v_{i+1}\}\Big)$.

3. $k = 1$ and $\theta(v_i) > 2$
   This means that v_i cannot be activated by v_{i+1}. Since we are allowed to have one seed, we set $mD[i, i+1, k, i] = \Big(\{v_{i+1}\}, \{v_{i+1}\}\Big)$.

4. $k \geq 1$ and $\theta(v_i) > 2$
   This means that v_i cannot be activated by v_{i+1}. Since we are allowed to have two seeds, we set $mD[i,i+1,k,i] = \Big(\{v_i, v_{i+1}\}, \{v_i, v_{i+1}\}\Big)$.

Before we define what values mD contains for larger segments, we define some operations.

1. *Addition of mD tuple values*: This yields a tuple containing the union of sets of activated vertices in each tuple, and union of the seedsets, i.e., if $mD[i',j',k',l'] = \Big(A(Y'),\, Y'\Big)$ and $mD[i'',j'',k'',l''] = \Big(A(Y''),\, Y''\Big)$, and we wish to calculate $mD[i',j',k',l'] + mD[i'',j'',k'',l'']$, we do the following computation :

$$\begin{aligned} mD[i',j',k',l'] + mD[i'',j'',k'',l''] &= \Big(A(Y'),\, Y'\Big) + \Big(A(Y''),\, Y''\Big) \\ &= \Big(A(Y') \cup A(Y''),\, Y' \cup Y''\Big). \end{aligned}$$

2. $m\alpha(i,j,k,D)$: This operation checks if a segment $v_i,...,v_j$ can be activated by at most k seeds, by looking at the tuple lookup table, D, where D is a lookup table storing the result of the Path_TD(P,w,θ) method with $w(v_i) = 1$ for all $i \in [n]$. It returns a tuple, $(\{v_i,...,v_j\}, Y)$, where $Y \subseteq \{v_i,...,v_j\}$ is a seedset, that fully activates the segment $(v_i,...,v_j)$, and $|Y| \leq k$. If no such seedset exists, this operation returns the tuple $(\varnothing, \varnothing)$.

Now, for segments of size greater than 2, we have either of the two possible cases in a tuple $mD[i,j,k,l]$:

1. The vertex v_l is not activated, and the segments left and right to v_l are partially activated, independently of each other, using a total of at most k seeds.

2. No inactive vertex exists, and the segment $(v_i,...,v_j)$ can be completely activated using at most k seeds.

Using these insights, we formally define mD as follows. First, let τ be the set of $k+2$ tuples defined as

$$\begin{aligned} \tau[i,j,k,l] = \Big\{ &\operatorname*{argmax}_{\substack{mD[i,l-1,r,l_1],\\ l_1 \in \{i,...,l-1\}}} A(mD[i,l-1,r,l_1]) + \operatorname*{argmax}_{\substack{mD[l+1,j,k-r,l_2],\\ l_2 \in \{l+1,...,j\}}} A(mD[l+1,j,k-r,l_2]) : \forall r = 0,...,k \Big\} \\ &\cup \Big\{ m\alpha(i,j,k,D) \Big\} \end{aligned} \tag{5.1}$$

Algorithm 10 $m\alpha(i,j,k,D)$

INPUT: Three positive integers, $1 \leq i,\ j,\ k \leq n$, and a lookup table, D, which is the result of a Path_TD(P,w,θ) call .

OUTPUT: A tuple containing a positive integer and a subset of vertices in P.

1: $\Big(w(Y),Y\Big) = \text{CalculateSeedset}((v_i,...,v_j),D)$
2: **if** $|Y| \leq k$ **then**
3: **return** $\Big(\{v_i,...,v_j\},Y\Big)$
4: **end if**
5: **return** $(\varnothing,\varnothing)$

Pseudocode for $m\alpha(i,j,k,D)$.

Note that in the first case, the first term vanishes if $l = i$ and the second term vanishes if $l = j$. Now, we define $mD[i,j,k,l]$ as

$$mD[i,j,k,l] = \underset{(A(Y),Y)\in\tau[i,j,k,l]}{\text{argmax}} A(Y) \tag{5.2}$$

Algorithm 11 first calls the Path_TD method on the input, then fills in tuples for segments of length 1 in Steps 3 - 11. Then, Steps 12-32 fill in tuples for larger segments, increasing the size by one vertex at a time. In Step 12, iterator i fixes the size of the segment. In Step 13, iterator j fixes the leftmost vertex of the segment (making the rightmost vertex v_{i+j}). In Step 14, iterator l fixes the last activated vertex, which has an index in the set $\{j,...,j+i\}$. In Step 15, iterator k' iterates over the set $\{0,....,k\}$ fixing the budget for the cardinality of the seedset. In Steps 16-22, we calculate the set of tuples, $\tau[j,i+j,k',l]$ for the segment $(v_i,...,v_l,...v_{i+j})$. In Step 23, we set the value of $mD[j,i+j,k',l]$ as the tuple $(A(Y),Y)$ in $\tau[j,i+j,k',l]$ which has the maximum $A(Y)$ value.

Lemma 5.2.1. *Algorithm 10 takes constant time.*

Proof. By Theorem 3.2.2, Step 1 takes $\Theta(1)$ time. The result follows. □

Algorithm 11 Path_maxTD(P, θ, k)

INPUT: A maxTD instance, (P, θ, k), with n vertices.
OUTPUT: A lookup table, mD, that stores all $mD[i, j, k, l]$ tuple values for $1 \leq i \leq l \leq j \leq n$.

1: $w : V \to 1$
2: $D = \text{Path_TD}(P, w, \theta)$

3: **for all** $i \in [n]$ **do**
4:     **for all** $k' \in 0...k$ **do**

5:         **if** $k' = 0$ **then**
6:             $mD[i, i, k', i] = \Big(\varnothing, \varnothing\Big)$
7:         **else**
8:             $mD[i, i, k', i] = \Big(\{v_i\}, \{v_i\}\Big)$
9:         **end if**

10:     **end for**
11: **end for**

12: **for all** $i \in \{2, ..., n\}$ **do**
13:     **for all** $j \in \{1, ..., n - i\}$ **do**
14:         **for all** $l \in \{j, ..., i + j\}$ **do**
15:             **for all** $k' \in \{0, ..., k\}$ **do**

16:                 **if** $l == j$ **then**
17:

$$\tau[j, i+j, k', l] = \Big\{ \operatorname*{argmax}_{\substack{mD[l+1, i+j, k', j'], \\ j' \in [l+1, ..., i+j]}} A(mD[l+1, i+j, k', j']) \Big\} \cup \Big\{ m\alpha(j, i+j, k', D) \Big\}$$

18:                 **else if** $l == i + j$ **then**
19:

$$\tau[j, i+j, k', l] = \Big\{ \operatorname*{argmax}_{\substack{mD[j, l-1, r, i'], \\ i' \in [j, ..., l-1]}} A(mD[j, l-1, r, i']) \Big\} \cup \Big\{ m\alpha(j, i+j, k', D) \Big\}$$

Pseudocode for Path_maxTD(P, θ, k).

Algorithm 11 Path_maxTD(P,θ,k) (continued)

20: **else**

21:

$$\tau[j,i+j,k',l] = \Big\{ \underset{\substack{mD[l+1,i+j,k',j'],\\ j'\in[l+1,\ldots,i+j]}}{\text{argmax}} A(mD[l+1,i+j,r,j']) + \underset{\substack{mD[j,l-1,r,i'],\\ i'\in[j,\ldots,l-1]}}{\text{argmax}} A(mD[j,l-1,r,i']) : \forall r = 0,\ldots,k' \Big\} \cup \Big\{ m\alpha(j,i+j,k',D) \Big\}$$

22: **end if**

23: $mD[j,i+j,k',l] = \underset{(A(Y),Y)\in\tau[j,i+j,k',l]}{\text{argmax}} A(Y)$

24: **end for**

25: **end for**

26: **end for**

27: **end for**

28: **return** mD

Pseudocode for Path_maxTD(P,θ,k) (Continued).

Theorem 5.2.2. *A minimum cardinality seedset for a maxTD instance on a path can be computed in $O(k \cdot n^4)$ time.*

Proof. Consider Algorithm 11. Step 1 assigns the weight of 1 to each vertex. This can be done in $O(n)$ operations. From Theorem 3.2.1 we have that Step 2 takes $O(n^3 \log n)$ time.

Filling in tuples for segments of size 1 requires one condition check, and one assignment operation. There are a total of $(k+1) \cdot n$ tuples. Thus, Steps 3-11 take $O(k \cdot n)$ operations.

As in TD on paths, we create $(k+1) \cdot n^2$ max-heaps, one each to represent a sub-segment in the input with an associated seed budget, which is an integer in $[0,k]$. We store tuples in max-heaps according to the segment they represent, with keys as the size of the set of active nodes associated with the tuple. Now, to calculate the set $\tau[j,i+j,k',l]$, we perform the following computation :

$$\tau[j,i+j,k',l] = \Big\{ \underset{\substack{mD[l+1,i+j,k',j'],\\ j' \in [l+1,\dots,i+j]}}{\operatorname{argmax}} A(mD[l+1,i+j,r,j']) + \underset{\substack{mD[j,l-1,k'-r,i'],\\ i' \in [j,\dots,l-1]}}{\operatorname{argmax}} A(mD[j,l-1,k'-r,i']) : \forall r = 0,\dots,k' \Big\} \cup \Big\{ m\alpha(j,i+j,k',D) \Big\}$$

(in Steps 20-22). This requires $2(k'+1)$ max-heap root retrieval operations and $(k'+1)$ addition operations, where $(k'+1)$ is the maximum cardinality of the seedset obtained by combining the tuples from the first and the second term. Also, $m\alpha(j,i+j,k',D)$ can be calculated in $\Theta(1)$ time. Thus, Steps 20-22 take at most $O(k)$ operations. Steps 16-17 cover the case where l is the first vertex in the segment, and Steps 18-19 cover case where l is the last vertex in the segment. These have one term less than Steps 20-22 to calculate, and will take at most as much time as Steps 20 -22. Therefore, Steps 16 - 22 can be computed in $O(k)$ time. Now, we need the tuple in $\tau[j,i+j,k',l]$ with the highest total number of active vertices in the segment $v_j,\dots,v_{i+j}$. This is done by

$$mD[j,i+j,k',l] = \underset{(A(Y),Y) \in \tau[j,i+j,k',l]}{\operatorname{argmax}} A(Y)$$

(Step 23). As $\tau[j,i+j,k',l]$ has $k'+2$ tuples, calculating the RHS is done in $O(k'+2) = O(k)$ operations. Insertion into the max-heap takes $O(i+1) = O(\log n)$ operations. Iterators i, j and l loop over the set $\{1,\dots,n\}$. Iterator k' iterates over $\{0,\dots,k\}$. Thus, Steps 16-23 are performed $O(k \cdot n^3)$ times.

The seedset for the input path is finally calculated via

$$\underset{\substack{mD[1,n,k,l],\\ l\in[n]}}{\text{argmax}}\ A(mD[1,n,k,l]).$$

As discussed earlier, this takes $\Theta(1)$ operations. Thus, a maxTD instance on a path can be solved in $O(k \cdot n^3(k+\log n)) = O(k \cdot n^4)$ time. □

5.2.1 Extension to Spiders

Here, we formulate a dynamic program for maxTD in the case where the input graph is a spider $S[\beta_1,...,\beta_\ell] = (V,E)$, with n nodes, and ℓ legs, which are indexed by $[\ell]$, where ℓ is a constant positive integer less than n. As in the previous sections, the leg lengths are denoted by a set of ℓ non-negative integers, $\{\beta_1,....,\beta_\ell\}$. A vertex is represented as $v_{i,a}$, and the subscript indicates that it is the ith vertex from the root in the leg with index a, where $a \in \{1,...,\ell\}$. The root of the spider is denoted by $v_{0,0}$.

Each leg is represented by the path (V_a,E_a) for all $a \in [\ell]$. The vertex set of each spider leg is given by $V_a := \{v_{1,a},...,v_{\beta_a,a}\}$ for all legs $a \in [\ell]$. The edge set of each spider leg is given by $E_a := \{(v_{j,a},v_{j+1,a}) \mid j \in [\beta_a - 1]\}$ for all legs $a \in [\ell]$. Note that the vertex set is the union of the root vertex and the vertices of the legs, i.e., $V = \{v_{0,0}\} \cup V_1 \cup V_2 \cup ... \cup V_\ell$. The edge set of the spider is given by $E = E_1 \cup ... \cup E_\ell \cup \{(v_{0,0},v_{1,1}),...,(v_{0,0},v_{1,\ell})\}$. We have weights on the vertices assigned by the function $w : V \to \mathbb{R}_+$, and a threshold function $\theta : V \to \{2,..,n\}$. We use the notation $\theta(i,a)$ to represent $\theta(v_{i,a})$ for all $v_{i,a} \in V$.

Let $mD[S[\beta_1,....,\beta_\ell],k,i,j] := \big(A(Y),Y\big)$ be a tuple containing (i) a seedset, Y, where Y has cardinality at most k, that activates the maximum number of vertices in the spider, $S[\beta_1,....,\beta_\ell]$, and (ii) the set of vertices in S activated by that seedset, $A(Y)$, with the following property : if k is less than the required number of seeds to activate S, $v_{i,j} \in S$ is not activated, else $A(Y)$ contains all vertices in S. Furthermore, we define $A(mD[S[\beta_1,....,\beta_\ell],k,i,j]) := |A(Y)|$. We will use the notation $mD[S,k,i,j]$ where the list of leg lengths is clear from context. Also, mD is used to refer to the lookup table of all $mD[S,k,i,j]$ tuples.

Also, let there be a path, $P = (v_{i,a},v_{i+1,a},...,v_{l,a})$, such that $v_{c,a}$ is a vertex and $c \in \{i,...,l\}$. The tuple, $mD[(v_{i,a},v_{i+1,a},...,v_{l,a}),k,c,a] = \big(A(Y'),Y'\big)$, contains (i) the seedset, Y', that activates the maximum number of vertices in the leg segment, P, such that $|Y| \le k$, and (ii) the set of vertices eventually activated by that seedset, $A(Y')$, with the following property : if k is less than the required number of seeds to activate P, $v_{c,a} \in P$ is not activated, else $A(Y)$ contains all vertices in P. Furthermore, we define $A(mD[(v_{i,a},v_{i+1,a},...,v_{l,a}),k,c,a]) := |A(Y')|$. Again,

we will use the notation $mD[P,k,c,a]$ instead of $mD[(v_{i,a},v_{i+1,a},...,v_{l,a}),k,c,a]$ where the path vertex indices are clear from context. Recall that this tuple can be computed with a Path_maxTD method call, with P as input.

We define the following operations.

- *Addition of mD tuple values*: This yields a tuple containing the union of the sets of active vertices, and union of the seedsets, i.e., if $mD[S',k',i',a'] = \Big(A(Y'),Y'\Big)$ and $mD[S'',k'',i'',a''] = \Big(A(Y''),Y''\Big)$, and we wish to calculate $mD[S',k',i',a'] + mD[S'',k'',i'',a'']$, we do the following computation :

$$\begin{aligned} mD[S',k',i',a'] + mD[S'',k'',i'',a''] &= \Big(A(Y'),Y'\Big) + \Big(A(Y''),Y''\Big) \\ &= \Big(A(Y') \cup A(Y''),\ Y' \cup Y''\Big). \end{aligned}$$

- $mS\alpha(S,D,k)$: This operation checks if a graph, $S(V,E)$, can be activated by at most k seeds, by checking the lookup table, D, where D is a lookup table storing the result of the Spider_TD(S,w,θ) method with $w(v_{i,j}) = 1$ for all $v_{i,j} \in S$. It returns a tuple (V,Y) where $Y \subseteq V$ is a seedset, that fully activates the spider S, and $|Y| \leq k$. If no such seedset exists, this operation returns the tuple $(\varnothing,\varnothing)$.

Algorithm 12 $mS\alpha(S,D,k)$

INPUT: A spider graph, $S = (V,E)$, a lookup table, D, which is the result of a Spider_TD(P,w,θ) call, and a positive integer, k .
OUTPUT: A tuple containing a positive integer and a subset of vertices in S.

1: $\Big(w(Y),Y\Big)$ = CalculateSeedsetSpider(S,D)
2: **if** $|Y| \leq k$ **then**
3:    **return** $\Big(V,Y\Big)$
4: **end if**
5: **return** $(\varnothing,\varnothing)$

Pseudocode for operation $mS\alpha(S,D,k)$.

Lemma 5.2.3. *The operation $mS\alpha(S,D,k)$ takes constant time.*

Proof. By Theorem 3.2.9, Step 1 takes $\Theta(1)$ time. The result follows. $\square$

To calculate the seedset of a maxTD instance on a spider, we use the recurrence for maxTD on paths [Equation 5.2] that we defined previously. Consider spider $S[\beta_1,....,\beta_\ell]$ in Figure 3.6. We observe two things.

1. If the vertex $v_{i,a}$ is not activated, then the spider can be seen as split into a smaller spider, $S[\beta_1,....,\beta_a-(\beta_a-i+1),...,\beta_\ell]=S[\beta_1,....,(i-1),...,\beta_\ell]$, and a leg segment $(v_{i+1,a},...,v_{\beta_a,a})$, which are (partially) activated, independently of each other. As the leg segment is a path, and is activated independently of the all other vertices outside the segment, we can calculate the minimum cardinality seedset to activate the maximum vertices in it with Equation 5.2.

2. If the root is not activated, we can calculate seedsets using Equation 5.2 for each leg, such that their union maximizes the total number of active vertices in each of the ℓ legs.

Using these insights, we calculate mD tuple values for the connected subgraphs of a spider in Algorithm 13. Steps 1-2 retrieve D tuple values via a Spider_TD(S,w,θ) method call. Step 3 allocates an empty lookup table, mD. Steps 4-12 iterate over all legs (we start the legs at the root vertex for the purposes of this algorithm) and call the method Path_maxTD(P_a,θ,k') on each leg, P_a, where $a\in[\ell]$ and $k'\in 0...k$. This method returns a tuple lookup table for each leg, mD_a. We update mD by putting the value of tuple $mD_a[i,j,\hat{k},l]$ in $mD[(v_{i,a},...,v_{j,a}),\hat{k},l,a]$. Thus, we can get seedsets of cardinality at most k' for all connected sub-segments in each leg from mD. We store these mD tuple values for each leg segment in max-heaps (Theorem 5.2.2), where we have a separate max-heap for each $k'\in 0...k$ for every subgraph.

Steps 13-54 solve maxTD on connected subgraphs of S that contain the root vertex, increasing the size of the subgraphs by one vertex with every increment in the iterator, n'. In Step 14, the iterator $S'[b'_1,...,b'_\ell]$ iterates over all spiders with n' vertices, where the set of these spiders is obtained by using the Spiders(S,n') operation. In Step 15, the iterator b'_j iterates over the set of leg lengths of $S'[b'_1,...,b'_\ell]$. If $b'_j<\beta_j$, which is the length of the jth leg of the input spider, S, we add one vertex to the jth leg of S', using the Join operation in Step 18, creating the spider $\hat{S}$. In Step 19, iterator k' sets the seedset budget, taking values from $\{0,..,k\}$. In Step 20, iterator $v_{i,a}$ iterates over all vertices in $\hat{S}$.

Now, we need to calculate $mD[\hat{S},k',i,a]$ for all vertices $v_{i,a}\in\hat{S}$ As in the case of paths (Theorem 5.2.2), we store $mD[\hat{S},k',i,a]$ tuple values in a max-heap, where the keys are the

Algorithm 13 Spider_maxTD$(S[\beta_1,...,\beta_\ell],\theta,k)$

INPUT: A maxTD instance on a spider, $(S[\beta_1,...,\beta_\ell],\theta,k)$, with n vertices.
OUTPUT: A lookup table of tuples, mD.

```
1:  w : V → 1
2:  D = Spider_TD(S, w, θ)
3:  mD → ∅
4:  for all a ∈ [ℓ] do
5:      P_a := (v_{0,0}, v_{1,a}, v_{2,a}, ..., v_{β_a,a})
6:      for all k' ∈ 0...k do
7:          mD_a = Path_maxTD(P_a, θ, k')
8:          for all mD_a[i, j, k̂, l] ∈ mD_a do
9:              mD[(v_{i,a}, ..., v_{j,a}), k̂, l, a] = mD_a[(v_{i,a}, ..., v_{j,a}), k̂, l, a]
10:         end for
11:     end for
12: end for

13: for all n' ∈ [n − 1] do
14:     for all S'[b'_1, ..., b'_ℓ] ∈ Spiders(S[β_1, ..., β_ℓ], n') do
15:         for all b'_j ∈ {b'_i, ..., b'_ℓ} do

16:             if b'_j < β_j then
17:                 b̂_j = b'_j + 1
18:                 Ŝ[b̂_1, ..., b̂_ℓ] = Join(S'[b'_1, ..., b'_ℓ], b'_j + 1, j)

19:                 for all k' ∈ [k] do
20:                     for all v_{i,a} ∈ Ŝ do

21:                         if i == b̂_j AND a == j then
22:                             τ[Ŝ, k', i, a] = { argmax_{mD[S',k',c,d], v_{c,d} ∈ S'} A(mD[S', k', c, d]) } ∪ { mSα(Ŝ, D, k') }
```

Pseudocode for Spider_maxTD$(S[\beta_1,...,\beta_\ell],\theta,k)$.

Algorithm 13 Spider_maxTD($S[\beta_1,...,\beta_\ell],\theta,k$) (continued)

23: **else if** $i == 0$ AND $a == 0$ **then**

24: $\overline{P}_d := (v_{1,d},...,v_{\hat{b}_d,d})$

$$F[k'] := \left\{\{k'_1,...,k'_\ell\} : k'_1+...+k'_\ell = k',\ k'_d \geq 0\ \forall d \in [\ell]\right\}$$

$$\tau[\hat{S},k',i,a] = \left\{ \sum_{d\in[\ell]} \left(\underset{\substack{mD[\overline{P}_d,k'_d,c,d],\\ v_{c,d}\in\overline{P}_d}}{\text{argmax}} A(mD[\overline{P}_d,k'_d,c,d]) \right) : \{k'_1,...,k'_\ell\} \in F[k'] \right\} \cup \left\{ mS\alpha(\hat{S},D,k') \right\}$$

25: **else**

26: $\overline{S}[\overline{b}_1,...,\overline{b}_\ell] = \text{Split}(\hat{S},i,a)$

$$\overline{P} := (v_{i+1,a},...,v_{\hat{b}_a,a})$$

$$\tau[\hat{S},k',i,a] = \left\{ \underset{\substack{mD[\overline{S},k'-r,c,d],\\ v_{c,d}\in\overline{S}}}{\text{argmax}} A(mD[\overline{S},k'-r,c,d]) + \underset{\substack{mD[\overline{P},r,c,a],\\ v_{c,a}\in\overline{P}}}{\text{argmax}} A(mD[\overline{P},r,c,a]) : \forall r \in 0,...,k' \right\} \cup \left\{ mS\alpha(\hat{S},D,k') \right\}$$

27: **end if**

28: $$mD[\hat{S},k',i,a] = \underset{(A(Y),Y)\in\tau[\hat{S},k',i,a]}{\text{argmax}} A(Y)$$

29: **end for**

30: **end for**

31: **end if**

32: **end for**

33: **end for**

34: **end for**

35: **return** mD

Pseudocode for Spider_maxTD($S[\beta_1,...,\beta_\ell],\theta,k$) (continued).

$A(mD[\hat{S},k',i,a])$ values. Every distinct spider is assigned its own set of $k+1$ separate max-heaps, and it stores $mD[\hat{S},k',i,a]$ values for all $v_{i,a} \in \hat{S}$ and $k' \in 0...k$.

Lemma 5.2.4. *We need $O(k^{\ell}+n)$ operations each to compute $mD[\hat{S},k,i,a]$ for all $v_{i,a} \in \hat{S}$, where $\hat{S}$ has n vertices.*

Proof. First, we define a set tuples, $\tau[S,k,i,a]$, which contains (i) all tuples of the form $(A(Y),Y)$, where Y is the seedset that has cardinality at most k, and $A(Y)$ is the set of vertices eventually activated in S by Y such that $v_{i,a} \in S$ is the last activated vertex, or not activated, and (ii) the tuple returned by a $m\alpha(i,j,k,D)$. As in the case of paths, we can define $mD[S,k,i,a]$ as

$$mD[S,k,i,a] = \underset{(A(Y),Y)\in\tau[S,k,i,a]}{\text{argmax}} A(Y).$$

To see what tuples $\tau[S,k,i,a]$ contains, we go back to the formation of $\hat{S}$ in Step 18. There are three possible cases which occur when a vertex, $v_{\hat{b}_j,j}$, is joined to spider S' to form spider $\hat{S}$, which has n' vertices.

1. The vertex, $v_{i,a}$, is the same as the newly added vertex, $v_{\hat{b}_j,j}$. If $v_{i,a}$ is not activated, we obtain the seedset of cardinality k' required to activate the maximum number of vertices in the spider $\hat{S}$, by the mD tuple value corresponding the seedset required to activate the maximum number of nodes in S' (Steps 21-22). The seedset can be found by looking at the root of the mD tuple max-heap associated to S' with seedset budget k'. This is a constant time operation.

   If $v_{i,a}$ is activated, then we obtain the seedset of cardinality k' required to activate the spider $\hat{S}$, from the tuple returned by $m\alpha(\hat{S},D,k')$. (Steps 21-22). This is shown to be a constant time operation in Lemma 5.2.3. Thus, this case takes $O(1)$ operations.

2. The vertex, $v_{i,a}$ is the same as the root. If it is not activated, we can calculate seedsets for maximizing the total number of active nodes in each of the ℓ legs of $\hat{S}$ by using Equation 5.2 on each leg. The seedset required to activate the maximum number of nodes in the spider $\hat{S}$ will be the union of the seedsets associated to each of the ℓ legs. We consider all seedset unions which have cardinality at most k', and pick the one that maximizes the total number of active vertices in $\hat{S}$. There are $\binom{k'+\ell-1}{\ell-1} = O((k')^{\ell-1})$ such unions possible (we distribute k' seeds over ℓ legs). The set $F[k']$ contains all these distributions.

$$F[k'] := \left\{\{k'_1,...,k'_\ell\} : k'_1 + ... + k'_\ell \le k',\ k'_d \ge 0\ \forall d \in [\ell]\right\}$$

To generate $F[k']$, we need $k' \cdot O((k')^{\ell-1}) = O((k')^{\ell})$ operations. Now, we calculate seedsets for all such unions. Let $\overline{P}_d := (v_{1,d},...,v_{\hat{b}_d,d})$ denote the dth leg of $\hat{S}$, starting from the the first vertex. Calculating the seedset of cardinality k'_d that activates the maximum number of vertices in $\overline{P}_d$ is done by $\underset{\substack{mD[\overline{P}_d,k'_d,c,d],\\ v_{c,d}\in\overline{P}_d}}{\text{argmax}} A(mD[\overline{P}_d,k'_d,c,d])$. Calculating a seedset for $\hat{S}$ of cardinality at most k' for the distribution $\{k'_1,...,k'_\ell\}$, where the dth leg of $\hat{S}$ is allocated a seedset budget of k'_d, is done by computing $\left(\underset{\substack{mD[\overline{P}_d,k'_d,c,d],\\ v_{c,d}\in\overline{P}_d}}{\text{argmax}} A(mD[\overline{P}_d,k'_d,c,d])\right)$. Looking up a mD tuple value involves looking at the root of the mD tuple max-heap associated to $\overline{P}_d$ with a seedset budget of k'_d. This is a constant time operation. We do ℓ such operations - one for each leg - for each distribution, $\{k'_1,...,k'_\ell\}$. There are $\ell-1$ addition operations. Hence, for all distributions, we do $O((2\ell-1)\cdot(k')^{\ell-1}) = O((k')^{\ell-1})$ operations.

If $v_{i,a}$ is activated, then we obtain the seedset of cardinality k' required to activate the spider $\hat{S}$, from the tuple returned by $m\alpha(\hat{S},D,k')$ (Steps 23-24). This is shown to be a constant time operation in Lemma 5.2.3. Thus, this case takes $O((k')^{\ell})$ operations.

3. The vertex $v_{i,a}$ is not the root, or the newly added vertex, $v_{\hat{b}_j,j}$. If $v_{i,a}$ is not activated, the spider $\hat{S}$ can be seen as split into a smaller spider, $\overline{S}[\overline{b}_1,...,\overline{b}_\ell]$ and a leg segment $\overline{P} = (v_{i+1,a},...,v_{\beta_a,a})$, which are activated independently of each other. We have already calculated mD tuple values for $\overline{S}$ (as it is fewer vertices than $\hat{S}$ and would have been considered in a previous iteration of the algorithm), and $\overline{P}$ (as we calculate mD tuples for all sub-segments in a leg).

   Computing $\tau[\hat{S},k',i,a]$ requires adding all pairs of mD tuple values associated to $\overline{S}$ and $\overline{P}$ such that their seedsets combined have a cardinality of k'. There are $k'+1$ such pairs (allocating a budget of $k'-r$ to $\overline{S}$ and r to $\overline{P}$ for all $r\in 0...k'$). Each seedset computation is a constant time computation, requiring a max-heap lookup. There are $k'+1$ addition operations Thus for all pairs, we do $O(3(k'+1)) = O(k')$ operations.

   If $v_{i,a}$ is activated, then we obtain the seedset of cardinality k' required to activate the spider $\hat{S}$, from the tuple returned by $m\alpha(\hat{S},D,k')$. (Steps 25-26). This is shown to be a constant time operation in Lemma 5.2.3.

   The method call to Split has been shown to have $O(n')$ run time (Lemma 3.2.5). Thus, this case takes $O(n'+k')$ operations.

Now that we have obtained $\tau[\hat{S},k',i,a]$, we can compute $mD[S,k,i,a] = \underset{(A(Y),Y)\in\tau[S,k,i,a]}{\text{argmax}} A(Y)$.

The set $\tau[\hat{S},k',i,a]$ has $O((k')^{\ell-1})$ tuples (Case 2), assuming $\ell > 1$. Thus, the LHS will require $O((k')^{\ell-1})$ comparison operations. A max-heap insertion will take $O(\log n')$ operations. Thus, a mD tuple can be computed in $O\big(\max\{(k')^{\ell},\ k'+n'\} + (k')^{\ell-1} + \log n'\big) \leq O(k^{\ell}+n)$. ☐

Theorem 5.2.5. *A minimum cardinality seedset for a maxTD instance on a spider with a constant number of legs can be computed in polynomial time.*

Proof. Steps 1 -2 make one call to the Spider_TD method, which takes $O(n^{\ell+2})$ operations (Theorem 3.2.8). Steps 3-12 make $\ell \cdot (k+1)$ calls to the Path_maxTD method, which take $O(k \cdot n^3)$ time each, as shown in Theorem 5.2.2. Thus, $\ell \cdot (k+1)$ calls will take $O(k^2\ell \cdot n^3) = O(k^2 \cdot n^3)$ operations. Steps 13-35 compute mD tuples for all connected subgraphs of S that contain the root. Join operations were shown to be performed in constant time. As discussed above, computation of each tuple, $mD[S,k,i,a]$ takes $O(k^{\ell}+n)$ operations (Lemma 5.2.4). Also, as discussed earlier, for a given size n', there are $\binom{n+\ell-2}{\ell-1} = O((n')^{\ell-1})$ distinct spiders with ℓ legs. Each of these spiders have $n' \cdot (k+1)$ associated mD tuples. Thus, to compute all tuples, we need $T(n)$ operations :

$$
\begin{aligned}
T(n) &= O(n^{\ell+2}) + O(k^2 \cdot n^3) + \sum_{n'=1}^{n} O((n')^{\ell-1}) \cdot n' \cdot (k+1) \cdot O(k^{\ell}+n) \\
&= O(n^{\ell+2}) + \cdot k \cdot O(k^{\ell}+n) \sum_{n'=1}^{n} (n')^{\ell} \\
&= O(kn^{\ell+1}(k^{\ell}+n)).
\end{aligned}
$$

The seedset for the input spider is finally calculated via

$$
\underset{\substack{mD[S,k,i,j],\\ v_{i,j}\in S}}{\operatorname{argmax}}\ A(mD[S,k,i,j]).
$$

This is a constant time operation. Therefore, a maxTD instance on a spider can be solved in $O(kn^{\ell+1}(k^{\ell}+n))$ time. ☐

Chapter 6

Conclusion

We presented an exact polynomial-time dynamic programming algorithm for TD on spider graphs with a constant number of legs. It also works in the more general weighted setting, i.e., when each vertex of the spider has an associated non-negative weight, and the objective is to minimize the total weight of the chosen seedset, rather than its cardinality.

We also gave a linear programming formulation of this problem by showing that our dynamic programming algorithm fits into the directed hypergraph paradigm given by Martin et al. (1990).

Finally, we studied the *influence maximization* version of TD, called maxTD, which seeks a seedset of a fixed size k maximizing the number of vertices that will eventually activate. Relying once again on dynamic programming, we showed that this problem is also solvable in polynomial-time on spiders with a constant number of legs.

Several open problems remain. For example, designing a constant factor approximation algorithm for TD and maxTD on all spiders. Another would be designing an $O(\log n)$-approximation algorithm on general graphs.

A natural generalization of the TD model would be to have the input graph as directed. As far as we know, there has been no work on a directed version of TD.

References

Anderson, R. M., May, R. M., and Anderson, B. (1992). *Infectious diseases of humans: dynamics and control*, volume 28. Wiley Online Library.

Ausiello, G., Italiano, G. F., and Nanni, U. (1998). Hypergraph traversal revisited: Cost measures and dynamic algorithms. In *International Symposium on Mathematical Foundations of Computer Science*, pages 1–16. Springer.

Besag, J. (1974). Spatial interaction and the statistical analysis of lattice systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 192–236.

Chen, W., Lakshmanan, L. V., and Castillo, C. (2013). Information and influence propagation in social networks. *Synthesis Lectures on Data Management*, 5(4):1–177.

Chen, W., Wang, Y., and Yang, S. (2009). Efficient influence maximization in social networks. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 199–208. ACM.

Chen, W., Yuan, Y., and Zhang, L. (2010). Scalable influence maximization in social networks under the linear threshold model. In *Data Mining (ICDM), 2010 IEEE 10th International Conference on*, pages 88–97. IEEE.

Cormen, T. H. (2009). *Introduction to algorithms*. MIT press.

Domingos, P. and Richardson, M. (2001). Mining the network value of customers. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 57–66. ACM.

Durrett, R. (1988). *Lecture notes on particle systems and percolation*. Brooks/Cole Pub Co.

Efsandiari, H., Hajiaghyi, M., Könemann, J., Mahini, H., Malec, D., and Sanità, L. (2015). Approximate deadline-scheduling with precedence constraints. In *Algorithms-ESA 2015*, pages 483–495. Springer.

Feige, U. (1998). A threshold of ln n for approximating set cover. *Journal of the ACM (JACM)*, 45(4):634–652.

Goldberg, S. and Liu, Z. (2013). The diffusion of networking technologies. In *Proceedings of the Twenty-Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1577–1594. Society for Industrial and Applied Mathematics.

Goldenberg, J., Libai, B., and Muller, E. (2001a). Talk of the network: A complex systems look at the underlying process of word-of-mouth. *Marketing letters*, 12(3):211–223.

Goldenberg, J., Libai, B., and Muller, E. (2001b). Using complex systems analysis to advance marketing theory development: Modeling heterogeneity effects on new product growth through stochastic cellular automata. *Academy of Marketing Science Review*, 2001:1.

Goyal, A., Bonchi, F., Lakshmanan, L. V., and Venkatasubramanian, S. (2013). On minimizing budget and time in influence propagation over social networks. *Social Network Analysis and Mining*, 3(2):179–192.

Goyal, A., Lu, W., and Lakshmanan, L. V. (2011). Simpath: An efficient algorithm for influence maximization under the linear threshold model. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 211–220. IEEE.

Granovetter, M. (1978). Threshold models of collective behavior. *American journal of sociology*, pages 1420–1443.

Jain, A. K. and Chellappa, R. (1993). *Markov Random Fields: Theory and Applications*. Academic Press.

Jung, K., Heo, W., and Chen, W. (2012). Irie: Scalable and robust influence maximization in social networks. In *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pages 918–923. IEEE.

Kempe, D., Kleinberg, J., and Tardos, É. (2003). Maximizing the spread of influence through a social network. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 137–146. ACM.

Kimura, M., Saito, K., and Nakano, R. (2007). Extracting influential nodes for information diffusion on a social network. In *AAAI*, volume 7, pages 1371–1376.

Kindermann, R. and Snell, J. L. (1980). *Markov random fields and their applications*, volume 1. American Mathematical Society.

Kleinberg, J. and Tardos, E. (2006). *Algorithm design*. Pearson Education India.

Knudsen, B. (2003). Optimal multiple parsimony alignment with affine gap cost using a phylogenetic tree. In *International Workshop on Algorithms in Bioinformatics*, pages 433–446. Springer.

Könemann, J., Sadeghian, S., and Sanità, L. (2013). Better approximation algorithms for technology diffusion. In *European Symposium on Algorithms*, pages 637–646. Springer.

Konemann, J., Sadeghian, S., and Sanita, L. (2013). An lmp o (log n)-approximation algorithm for node weighted prize collecting steiner tree. In *Foundations of Computer Science (FOCS), 2013 IEEE 54th Annual Symposium on*, pages 568–577. IEEE.

Lenstra, J. K. and Kan, A. R. (1980). Complexity results for scheduling chains on a single machine. *European Journal of Operational Research*, 4(4):270–275.

Leskovec, J., Krause, A., Guestrin, C., Faloutsos, C., VanBriesen, J., and Glance, N. (2007). Cost-effective outbreak detection in networks. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 420–429. ACM.

Liggett, T. M. (1985). Introduction. In *Interacting Particle Systems*, pages 1–5. Springer.

Martin, R. K., Rardin, R. L., and Campbell, B. A. (1990). Polyhedral characterization of discrete dynamic programming. *Operations research*, 38(1):127–138.

Minoux, M. (1978). Accelerated greedy algorithms for maximizing submodular set functions. *Optimization techniques*, pages 234–243.

Moss, A. and Rabani, Y. (2007). Approximation algorithms for constrained node weighted steiner tree problems. *SIAM Journal on Computing*, 37(2):460–481.

Nemhauser, G. L., Wolsey, L. A., and Fisher, M. L. (1978). An analysis of approximations for maximizing submodular set functionsi. *Mathematical Programming*, 14(1):265–294.

Piatetsky-Shapiro, G. and Masand, B. (1999). Estimating campaign benefits and modeling lift. In *Proceedings of the fifth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 185–193. ACM.

Schelling, T. C. (2006). *Micromotives and macrobehavior*. WW Norton & Company.

Székely, L. A. and Wang, H. (2005). On subtrees of trees. *Advances in Applied Mathematics*, 34(1):138–155.

Székely, L. A. and Wang, H. (2007). Binary trees with the largest number of subtrees. *Discrete Applied Mathematics*, 155(3):374–385.

Wang, C., Chen, W., and Wang, Y. (2012). Scalable influence maximization for independent cascade model in large-scale social networks. *Data Mining and Knowledge Discovery*, 25(3):545.

Wolsey, L. A. (1982). An analysis of the greedy algorithm for the submodular set covering problem. *Combinatorica*, 2(4):385–393.