The application of the in-tree knapsack problem to routing prefix caches

by

Patrick Kevin Nicholson

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2009

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

Modern routers use specialized hardware, such as Ternary Content Addressable Memory (TCAM), to solve the Longest Prefix Matching Problem (LPMP) quickly. Due to the fact that TCAM is a non-standard type of memory and inherently parallel, there are concerns about its cost and power consumption. This problem is exacerbated by the growth in routing tables, which demands ever larger TCAMs.

To reduce the size of the TCAMs in a distributed forwarding environment, a batch caching model is proposed and analysed. The problem of determining which routing prefixes to store in the TCAMs reduces to the In-tree Knapsack Problem (ITKP) for unit weight vertices in this model.

Several algorithms are analyzed for solving the ITKP, both in the general case and when the problem is restricted to unit weight vertices. Additionally, a variant problem is proposed and analysed, which exploits the caching model to provide better solutions. This thesis concludes with discussion of open problems and future experimental work.

Acknowledgements

First and foremost, I would like to thank my supervisor, Ian Munro, for all of the help he has provided to me during the completion of this thesis. His guidance and suggestions were critical in keeping me focused and providing a direction for my research.

I would also like to thank Suran de Silva, Nilesh Shah, and Shyam Kapadia for all of their help during my internship at Cisco. The many discussions I had with them paved the way for this work.

Finally, I would like to thank my readers, Anna Lubiw and Gordon Agnew, for their very helpful corrections and suggestions to the initial version of my thesis.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivating application: routing prefix caches

1.1.1 The longest matching prefix problem

Routers maintain a table of *routing prefixes*, each of which *match* a contiguous range of the Internet Protocol (IP) address space, and are associated with packet forwarding information. Routing prefixes are expressed in the form a/m, where, in the case of IP version 4, a is a 32-bit address, and m is a mask length, defined over the range $[0, 32]$. The address portion of the prefix is typically expressed as a series of four octets in decimal, each representing 8 bits of the address, with values over the range $[0, 255]$, and concatenated by dots. The mask represents the size of the range that the prefix matches. Specifically, a prefix matches all IP addresses which share the m most significant bits of the prefix's address. To illustrate this, the routing prefix 192.168.2.0/24 matches all addresses between 192.168.2.0 and 192.168.2.255, whereas a prefix 192.168.0.0/16 matches all addresses between 192.168.0.0 and 192.168.255.255. Thus, the $32 - m$ least significant bits of a prefix are 'don't care' bits; their value is inconsequential in determining whether the prefix matches an address.

When a packet arrives at a router, the router must determine where to forward the packet so that it arrives at its destination. This is done by solving the Longest Prefix Matching Problem (LPMP) for the packet's destination address. The solution to the LPMP is the unique routing prefix, if it exists, that not only matches the destination address of the packet, but has a longer mask length than any other

matching prefixes. This problem is fundamental to routing, as it must be solved multiple times for every packet that is transmitted over the Internet.

Since 2000, solving the LPMP has become increasingly problematic due to the explosive growth in the number of routing prefixes for Internet routing tables. As of 2009, these tables contain around 290000 routing prefixes, and this number is expected to increase further [1, 2]. The rapid increase in the number of routing prefixes has led to the need for ever more scalable solutions to the LPMP.

Many interesting algorithms have been proposed to address this need, which can be used to solve the LPMP efficiently in software [32, 9]. However, most high-end routers use specialized hardware, such as Ternary Content Addressable Memory (TCAM), to solve the the LPMP in a single memory access [14]. Although these hardware solutions are incredibly fast, they have other drawbacks. The main ones being that they tend to be expensive and consume a great deal of power, as they are highly parallelized.

1.1.2 A caching model for routing prefix caches

In a distributed forwarding environment, where each router line-card maintains a TCAM storing the entire routing table, one idea to reduce power consumption and cost is to reduce the capacity of the TCAM in each line-card. Effectively, this would turn each line-card TCAM into a routing prefix cache: packets triggering cache hits could be forwarded to their correct port of egress immediately, whereas packets triggering cache misses would have to be forwarded to a central forwarding engine with access to the complete routing table. However, this model of prefix caching introduces two challenges not present in standard caching models, such as page caching.

The first is that *dependencies* exist among the routing prefixes, which means that certain prefixes can only be cached if all of their dependencies are also cached. This can be best illustrated by the following example. Suppose a cache has room for one prefix, and the routing table has only two prefixes, 192.168.2.0/24 and 192.168.2.100/32 (the latter being a subrange of the former). In this case, the prefix 192.168.2.0/24 cannot be cached, as it will cause the cache to give incorrect solutions to the LPMP for packets having the destination address 192.168.2.100. Because each prefix matches a contiguous range of addresses, and these ranges are nested, the dependencies among prefixes can be modelled by a tree (or trie).

In this tree, each prefix p is a vertex, and all prefixes q which are subranges of p become descendants of this vertex. If p has the longest mask of any prefix for which q is a subrange, then the vertex corresponding to p is the parent of the vertex corresponding to q. In terms of this dependency tree, if a vertex is to be cached, then the entire subtree induced by that vertex must be cached as well.

The second challenge is that information regarding which prefixes are the most heavily used is difficult to obtain in this model. At current line rates, this information must be gathered using hardware counters and sampling devices [22], which take time to determine which prefixes are being used most frequently. Because these time-scales are much longer than the per packet rate of the router, enacting a prefix cache replacement policy with per packet granularity—for example, when a cache miss occurs—is difficult. Due to this, a more feasible approach for this model of prefix caching would be to enact *cache replacement in batch* at much longer time-scales. Discussion of the appropriate length of time between replacement operations is outside the scope of this thesis, but the assumption throughout is that a good cache hit ratio can be maintained even when these replacement operations occur at half-minute intervals.

In light of these two challenges, the problem of determining which routing prefixes should be stored in the line-card caches will be referred to as the Routing Prefix Caching Problem (RPCP). Before diving into more details about the RPCP, some of the previous work on routing prefix caching will be reviewed.

1.2 Previous work on routing prefix caching

1.2.1 Overview

The idea of using caching to ease the burden of solving the LMPP is not new. In 1988, Feldmeier proposed the use of a destination address cache to reduce the amount of time required to do routing table lookups [10]. By caching the forwarding decision for incoming destination addresses, solving the LPMP for subsequent requests for that destination address could be avoided. At that time this was proposed, packet forwarding was handled by dedicated Network Processors (NPs), which meant the caching model was very standard: a cache was considered to be fully associative, and a replacement decision would occur at each cache miss. A simulation comparing standard cache replacement policies was conducted using real

packet traces, and it was concluded that use of the proposed caching scheme could increase the speed of NPs by 65%.

The justification for caching destination addresses is based on the notion that incoming destination addresses have an exploitable temporal locality. Evidence to this effect was given in [10], and since then other studies since then have yielded similar results [6, 29]. In 2000, Chiueh and Pradhan proposed a different approach to caching with the Intelligent Host Address Range Cache scheme for NPs [6]. Their scheme seems to be the first scheme to cache routing prefixes rather than destination address alone, as well as to identify the dependency issues associated with prefix caching. By caching prefixes instead of destination addresses, a much larger portion of the address space could reside in the cache. Experimental evidence showed that IHARC provided a factor of 5 improvement in average lookup time over destination address caching alone.

Since 2000, most of the research on caching for network routing has been focused on prefix caches [21, 3], as well as 'multizone' caches, in which the cache is partitioned into different sections based on the length of the prefixes they store [8, 30, 25, 18]. Here the focus will be solely on methods developed to handle routing prefix dependencies, as these methods are applicable to prefix caching in both the NP model, as well as the distributed model described in Section 1.1.2.

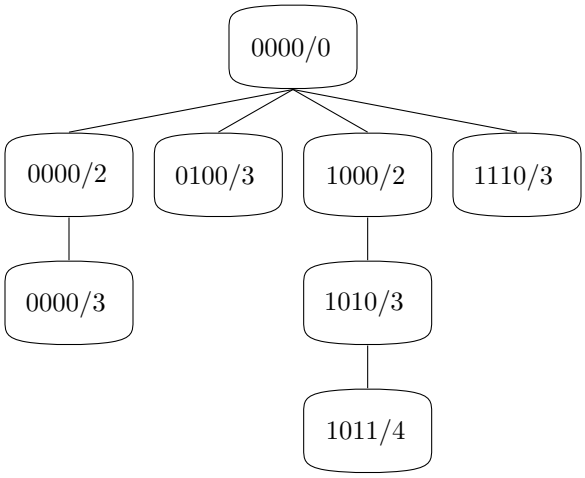

Figure 1.1: An example 4-bit routing table represented as a tree. A prefix c is a child of p if the address of c is matched by p and c has a longer mask length than p.

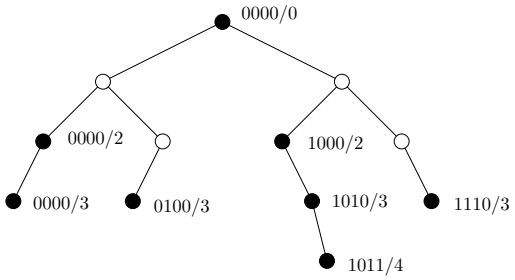

Figure 1.2: The 4-bit example routing table from Figure 1.2 represented as a trie. The black vertices represent prefixes defined in the routing table, while the white vertices are merely structural and do not correspond to actual prefixes. A right branch in the trie represents a 1 and a left branch represents a 0. The trie is traversed by recursively examining the next most significant bit of an address.

1.2.2 Handling prefix dependencies

The predominant strategies for dealing with the problem involve either refusing to cache prefixes with dependencies, or alternatively, modifying the routing table to either remove or reduce the number of dependencies [21, 3, 18]. Before describing the strategies, first consider the example the set of 4-bit routing prefixes in Figure 1.1. In this figure, the routing prefixes are arranged so that child prefixes are dependencies of their parents.

Another way of picturing the routing table is in the form of a binary trie. In a binary trie each edge represents one bit: 1 or 0. Each prefix is stored at a vertex, and the path to a vertex corresponds to bit-wise expansion of the address of the prefix. However, this path is truncated, such that its length is equal to the mask length of the prefix. For example, if the routing table in Figure 1.1 is represented in this manner, it results in the trie in Figure 1.2. To solve the LPMP in terms of this trie, the next most significant bit of the query destination address is recursively examined, and the branch taken in the trie—left or right—is determined by this bit. During this search, the last vertex corresponding to a routing prefix that is visited is the longest matching prefix for that destination address.

All of the strategies for eliminating or reducing routing table dependencies are described in terms of this trie representation of the routing table. These strategies fall into one of three categories:

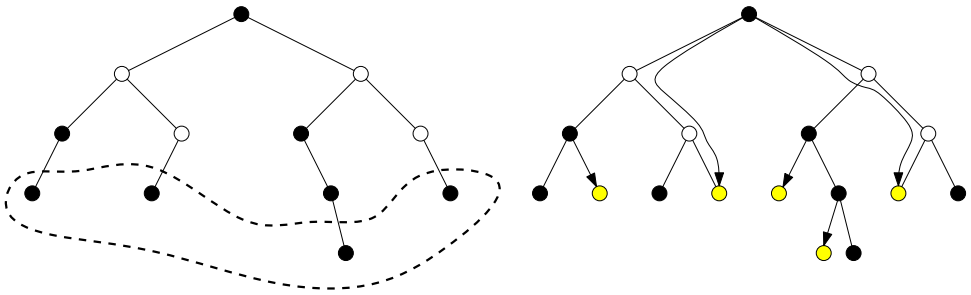

Figure 1.3: Left: The prefixes (black vertices) bounded by the dotted line are the only prefixes which are allowed to be cached under the "No modification" scheme. Right: Under the "Full modification" scheme, many expanded prefixes (yellow leaf vertices) must be added to the trie in order to cover the address space of the associated prefix with dependencies. For example, the prefix 0000/0 has dependencies, and therefore must be expanded into 0110/3 and 1100/3. The arrows denote which internal black vertices correspond to the new expanded yellow vertices.

No Modification

In this method, only prefixes that have no dependencies are allowed to be cached. In the 4-bit example, this means that only the prefixes bounded by the dotted line in Figure 1.3 (left) can be cached. This scheme is called 'No Prefix Expansion' in [21], or 'Parent Restriction' in [3]. Whenever a replacement happens a simple check is performed to determine whether the inserted prefix has dependencies, and, if so, an alternative course of action must be taken. One alternative in this situation is to instead cache the entire destination address of the packet with a full /32 bit mask [21].

Full Modification

All of the prefixes with dependencies are *expanded* into prefixes without dependencies by adding extra prefixes to the routing table [21, 3]. This entails modifying the trie structure of the routing table so that each internal vertex storing a routing table prefix is converted into one or more leaf vertices (see the right side of Figure 1.3). The added leaves contain the forwarding information of the original internal vertex.

One side effect of this method is that the added leaves greatly inflate the size of the routing table. In [21] it was found that, for routing tables with between 15906 and 61832 entries, the inflated tables can vary between 143-218% of the size of the original tables. Another side effect is that additional prefixes can increase the time

to update the routing table by more than constant factor in the worst case. This is due to the extra work required to deal with the added expanded prefixes during a prefix deletion, for example.

A solution to the first problem, called *minimal expansion*, was proposed in [3]. This entails expanding the prefixes in an on-demand fashion, so that new prefixes are only added as they are used. Although this reduces the inflation of the routing tables, minimal expansion still suffers from the increased worst case cost for routing table updates.

Partial Modification

Somewhere in between the previous two approaches is the partial modification scheme. Prefixes with dependencies are expanded, as in the full modification scheme, but in a limited way which only adds one at most expanded prefix per prefix in the routing table. This single new prefix is placed in such a way as to maximize the coverage of the address space [21].

Because the routing table receives exactly one extra entry for each prefix with dependencies the inflation factor is less concerning than that of the full modification method. In [21] it was found that this method caused routing tables to inflate to between 111-120% of their original size, and performed nearly as well as full modification.

Although this deals with the space issue, the trade off is that—as with the no modification scheme— partial modification provides no strict guarantees on caching performance. This is because some portions of the address space may remain unable to be cached. However, in its defense, experimental evidence suggests that it works almost as well as full modification [21].

1.2.3 Summary

Although prior work on prefix caching identifies and handles the problem of dependencies among prefixes, it also introduces new complications. If the full modification scheme is used, the number of additional prefixes required to modify the routing table can more than double it in size in practice. If the no modification, or partial modification schemes are used, then there is no strict guarantee that a heavily

used routing prefix will be able to be cached. Furthermore, both the full modification and partial modification schemes necessitate an extra layer of abstraction for routing table insertions and deletions.

The assumption throughout the literature is that replacement can occur on a per packet basis, and therefore most work recommends implementing standard cache replacement policies. This assumption is valid in the case where a network processor is used to solve the LPMP. However, for the specific distributed hardware forwarding model described in Section 1.1.2, it is not feasible.

1.3 Contributions of this thesis

In this thesis, an approach to solving the RPCP that is more suited to the model discussed in Section 1.1.2 is presented. Cache replacement is considered to be a batch replacement operation, rather than occurring on a per packet basis. Under this model, each cache replacement operation is shown to be equivalent to a special case of the In-tree Knapsack Problem (ITKP) where each vertex has unit weight. Therefore, this thesis addresses the question of how hard it is to solve the RPCP without modifying the routing table, if strict performance guarantees are desired.

Although the RPCP is the motivation for this work, the main results of this thesis are improved algorithms for the ITKP. Chapter 2 details the ITKP, explains its connection to the RPCP, and defines notation that will be used throughout the remainder of this thesis. In Chapter 3, new dynamic programming algorithms are presented to find the optimal solution of the ITKP, and their implications are discussed for the RPCP. In Chapter 4, a new $\frac{1}{2}$-approximation algorithm is presented for the ITKP, for the case where the tree structure need not be constructed. It is shown that this algorithm retains the same error bound, but has a superior running time when applied to the RPCP. In the final chapter, conclusions of this work are discussed along with a list of open problems.

Chapter 2

Modelling prefix caching using the in-tree knapsack problem

2.1 Problem descriptions and notation

2.1.1 Directed Graphs

A *directed graph* G is a tuple (V, E), where V is a set of vertices, and E is a set of edges. Each edge $(u, v) \in E$ is a pair of vertices and represents a connection between them, such that u is a predecessor of v, and v is a successor of u. Define $\textsc{Pre}(v)$ to be the set of vertices which are predecessors of v, and $\textsc{Suc}(v)$ to the set of vertices which are successors of v.

A *cycle* in a directed graph $G = (V, E)$ is a set of vertices $\{v_1, ..., v_k\} \subseteq V$ for any $k \geq 1$, such that $v_1 \in \textsc{Suc}(v_k)$, and $v_i \in \textsc{Suc}(v_{i+1})$ for $1 \leq i < k$. A *directed acyclic graph* is a directed graph that contains no cycles.

2.1.2 The precedence constraint knapsack problem

Given a directed acyclic graph $G = (V, E)$, define a set of vertices $V' \subseteq V$ as *closed under predecessor* with respect to G, if for every edge, $(v, u) \in E$, $u \in V'$ implies $v \in V'$. In other words, if a vertex set is closed under predecessor, then there are no edges directed into it from vertices not contained in the set. In the Precedence Constraint Knapsack Problem (PCKP) a directed acyclic graph $G = (V, E)$ is given, called the precedence graph, as well as a profit function $p : V \mapsto \mathbb{Z}^+$, a

weight function $w : V \mapsto \mathbb{Z}^+$, and a capacity $C \geq \max\{w(v) : v \in V\}$. Let $n = |V|$ represent the number of vertices in the graph, $P = p(V) = \sum_{v \in V} p(v)$ represent the total sum of the vertex profits, and $W = w(V) = \sum_{v \in V} w(v)$ be the total sum of the vertex weights. The optimal solution to an instance of the Precedence Constraint Knapsack Problem (PCKP) is a vertex set $X \subseteq V$ that maximizes

$$Q = \sum_{v \in X} p(v) \ , \tag{2.1}$$

is *closed under predecessor* with respect to G, and satisfies

$$\sum_{v \in X} w(v) \leq C \ . \tag{2.2}$$

2.1.3 In-tree and out-tree knapsack problems

An *in-tree* is a rooted tree with all edges directed toward the root. Similarly, an *out-tree* is a rooted tree with all of its edges directed out from the root.

In a tree, the subtree induced by a vertex v consists of v along with every descendant of v. For a given tree with vertex set V, define $S(v)$ to be the subtree induced by vertex v, for all $v \in V$. Conversely, for all $v \in V$, let $A(v)$ be set of vertices on the path from v to the root, inclusive. Figure 2.1 provides an illustration of these two concepts. For brevity, throughout this thesis the term 'subtree' should be interpreted to mean 'subtree induced by some vertex $v \in V$' unless otherwise stated.

In the case of an in-tree, $S(v)$ is closed under predecessor for all $v \in V$. Likewise, in the case of an out-tree, $A(v)$ is closed under predecessor for all $v \in V$. Two special cases of the PCKP are the In-tree Knapsack Problem (ITKP), and the Out-tree Knapsack Problem (OTKP), which occur when the precedence graph is an in-tree and an out-tree, respectively.

2.1.4 The routing prefix caching problem

At this point the link between the ITKP and the Routing Prefix Caching Problem (RPCP) should be clarified. In the RPCP, entries in the cache are replaced in batch whenever enough statistical information is available to indicate which prefixes are

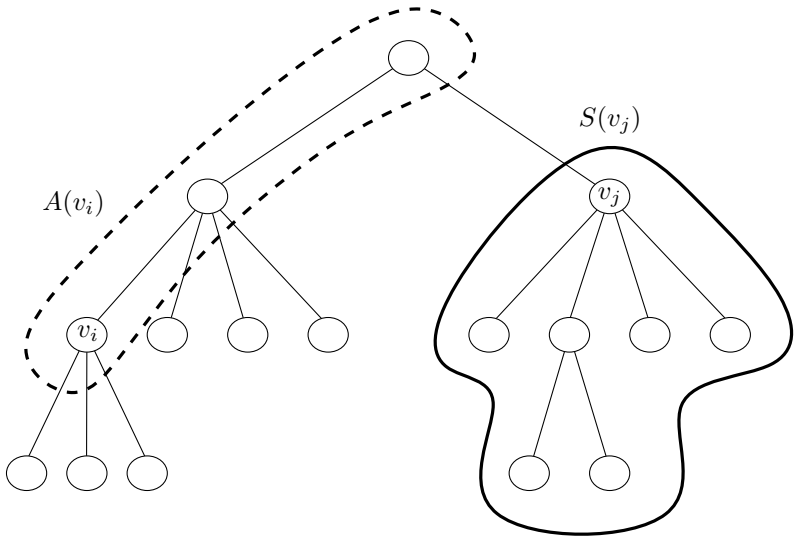

Figure 2.1: An example tree to demonstrate the subtree function $S(v_j)$ and the ancestor function $A(v_i)$, which would result in the set of vertices bounded by the solid line and dashed line, respectively.

being accessed most frequently. Each prefix requires the same amount of TCAM to be stored in the cache. At the time that the replacement occurs, it is assumed that each prefix in the routing table has been mapped to an integer profit value based on these statistics. The profit value is a measure of the benefit that is expected to be obtained by inserting the prefix into the cache.

From the above problem description and the nature of the prefix dependencies, the RPCP can be viewed as a special case of the ITKP where each vertex has unit weight. However, due to the precedence constraints, the ITKP is non-trivial for instances where each vertex has unit weight. This issue is discussed much later in this thesis, at the end of Chapter 4, but now one variant of the RPCP will be described.

2.1.5 Variant problem: punt prefixes

In practice, it might be the case that a particular routing prefix has a very large profit value, but too many unprofitable dependencies which prevent it from being cached. One mechanism to circumvent this issue is the introduction of *punt prefixes*, which trigger a cache miss when they are returned as the result of a LPMP query. At first, caching prefixes that *cause* cache misses may seem strange. However, as the following paragraph will explain, these punt prefixes can result allow better

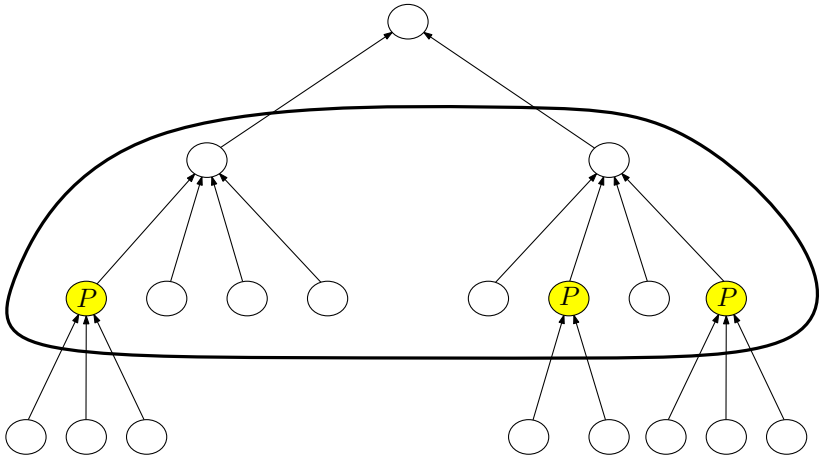

Figure 2.2: An example of how punt vertices relax the requirement that a vertex set be closed under predecessor. Consider the set of vertices bounded by the solid line in the figure, where the three vertices marked by 'P' have been toggled to be punt vertices. Because of these punt vertices, this set of vertices is a valid solution for the RPCP+

solutions to the RPCP.

Suppose that any prefix in the routing table can either be cached as a normal prefix, or as a punt prefix. When a punt prefix is present in the cache, *none* of its dependencies are required to be in the cache. This is because a cache miss is guaranteed to result in the correct forwarding decision for any packets matched by the punt prefix. Thus, in order to cache a given prefix p, not all of its dependencies need to be cached: only those prefixes which are children of p—in terms of the tree representation of the routing table—need to be cached as punt prefixes. Of course, the downside to punt prefixes is that since they cause a cache miss, no profit is gained by caching a prefix as a punt prefix. To distinguish this problem from the RPCP it will be referred to as the Routing Prefix Caching Problem with Punt Prefixes (RPCP+).

The addition of punt prefixes means that the RPCP+ cannot be abstracted directly to the ITKP with unit weights. To illustrate this, consider an instance of the ITKP having an in-tree $G = (V, E)$. To provide an analogous concept to punt prefixes, suppose that each vertex in V has the extra ability to toggle between a normal vertex and a *punt vertex*. A normal vertex v behaves exactly as expected: it has a profit $p(v)$ and weight $w(v)$ associated with it, and if X is a solution containing v, then *all* of the predecessors of v must be in X as well. Alternatively, if v is toggled to be a punt vertex, denoted $B(v)$, it still has weight $w(B(v)) = w(v)$, but $p(B(v)) = 0$, and if $B(v) \in X$, *none* of the predecessors of v are required to be

in X.

To clarify, punt vertices change the requirement that X be closed under predecessor. The new requirement can be explained recursively as follows. Suppose a normal vertex v is in X, having predecessors $\{u_1, ..., u_k\}$. For each u_i, $1 \leq i \leq k$, either $u_i \in X$ as a normal vertex, along with its required predecessors, or $B(u_i) \in X$. Note that in the latter case, even though any descendants of u_i are not required to be in X, they *can* be in X, so long as they also satisfy this new recursive closure definition. Figure 2.2 illustrates how punt vertices can be used.

Even though the RPCP+ is clearly distinct from the ITKP, the same general techniques presented in Chapters 3 and 4 can be applied to both problems. This is due to the fact that the RPCP+ retains a similar tree structure to the ITKP. There may be a better way of representing the RPCP+, though this remains an open question. For further discussion of an alternate framework for the RPCP+, refer to Appendix B after reading Chapters 3 and 4.

2.2 Exact vs. Approximate solutions

Throughout this thesis, there are many algorithms that are presented which sacrifice the optimality of a solution for a decrease in running time. Let Q represent the maximum achievable profit for the optimal solution(s) to the above problems. An ε-approximation algorithm [13] is defined to be any algorithm which returns a solution having profit $\hat{Q}$, such that

$$\frac{Q - \hat{Q}}{Q} \leq \varepsilon \ . \tag{2.3}$$

For approximation algorithms which accept ε as input, the ε parameter typically makes an appearance in the algorithm's time (and possibly space) requirements. Algorithms are called Polynomial Time Approximation Schemes (PTASs) if their running time is polynomial in n for any fixed ε. For example, an ε-approximation algorithm running in $\Theta(n^{1/\varepsilon})$ is considered to be a PTAS. However, note that as ε approaches 0, the running time of the algorithm, though still polynomial, becomes increasingly undesirable.

Another class of approximation algorithms, called Fully Polynomial Time Approximation Schemes (FPTASs), have a more strict property which is absent from

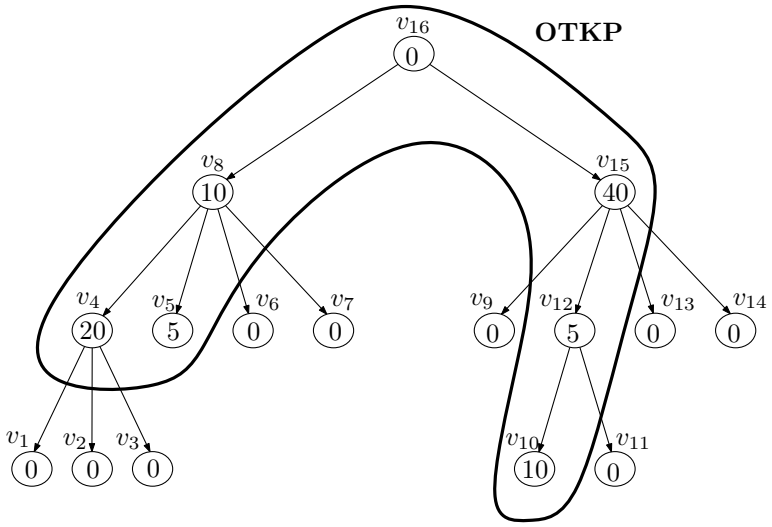

Figure 2.3: An example of an instance of the OTKP. Here, each vertex has unit weight, and the number written on each vertex indicates its profit value. The optimal solution to this OTKP instance for capacity $C = 10$ is bounded by the solid line.

PTASs. The running time of an FPTAS must be polynomial in both n *and* ε. For example, an ε-approximation algorithms requiring $\Theta(n^k/\varepsilon)$ time and space for some constant k would be considered to be FPTASs.

2.3 Numerical Examples

Before moving on to discuss prior work on the ITKP, it would help to provide some numerical examples of each of the above problems.

Example 2.3.1. Figure 2.3 is an example of an instance of the OTKP. This particular instance, which will be our running example, is a special case in which all the vertices have equal unit weight. If the knapsack capacity, $C = 10$, meaning that at most 10 of the vertices can be selected, then the optimal solution (minimizing weight) would be $\{v_4, v_5, v_8, v_{10}, v_{12}, v_{15}, v_{16}\}$, with total profit 90. It is easy to verify that this is the best solution, because each vertex is closed under predecessor, and it contains every vertex with non-zero profit.

Example 2.3.2. The direction of all of the edges in Figure 2.3 are reversed, then the problem becomes an instance of the ITKP, shown in Figure 2.4. Since each vertex has unit weight, this is also an example of the RPCP. For this instance of the problem, the optimal solution for $C = 10$ is less apparent since some of the vertices which

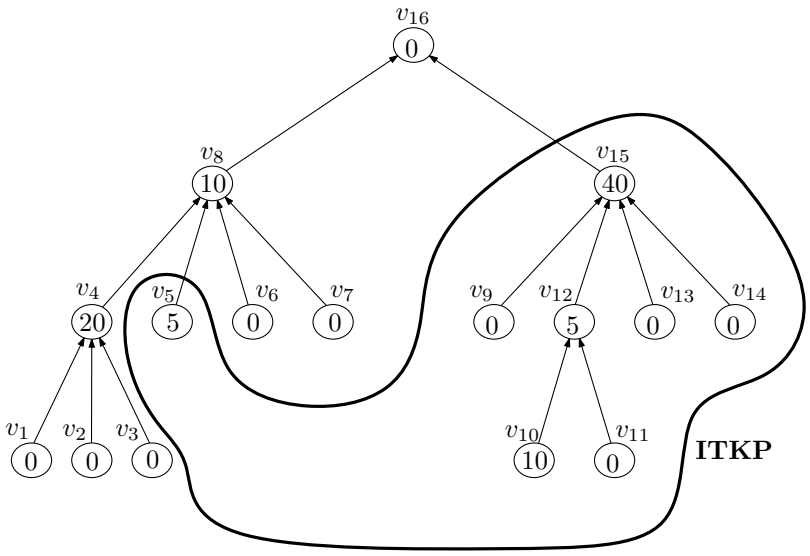

Figure 2.4: An example of an instance of both the ITKP, and the RPCP. Here, each vertex has unit weight, and the number written on each vertex indicates its profit value. The optimal solution to this instance for capacity $C = 10$ is bounded by the solid line.

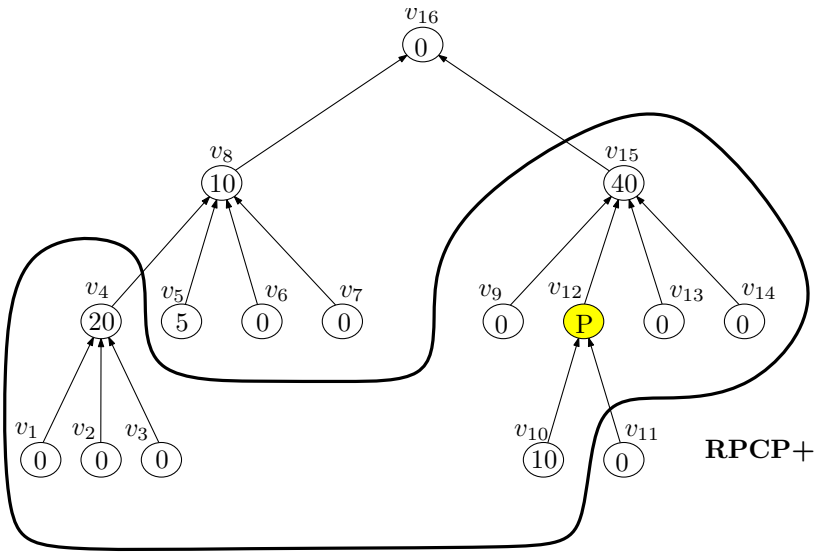

Figure 2.5: An example of an instance of the RPCP+. The optimal solution to this RPCP+ instance for capacity $C = 10$ is bounded by the solid line. In this particular example, converting v_{12} into $B(v_{12})$—denoted by replacing v_{12} with a vertex with a 'P'—has resulted in a higher profit solution than would otherwise be attainable without punt vertices.

have no profit associated with them must be selected in order to be able to reach the most profitable vertices. The optimal solution is $\{v_5, v_9, v_{10}, v_{11}, v_{12}, v_{13}, v_{14}, v_{15}\}$, with total profit 60.

Example 2.3.3. Let the in-tree shown in Figure 2.5 be an instance of the RPCP+, the problem variant that allows punt vertices. The optimal solution for $C = 10$ has a higher profit than the solution from Example 2.3.2 since punt vertices allow us to access more profitable vertices at less cost than before. The optimal solution is $\{v_1, v_2, v_3, v_4, v_9, v_{10}, B(v_{12}), v_{13}, v_{14}, v_{15}\}$, with total profit 70.

2.4 Review of prior work on the ITKP

Other applications of the ITKP include project scheduling and manufacturing, where a given project cannot be started until others been completed [16]. The OTKP can be applied to optimize strip mining operations [17], as well as a number of telecommunication network design problems [7].

The ITKP and OTKP can be shown to be NP-complete through a trivial reduction from the Knapsack Problem (KP) [19]: when the precedence constraints are a forest with no edges these problems reduce exactly to the KP. However, as is the case with the KP, the tree knapsack problems permit algorithms that are *pseudo-polynomial* in both time and space. An algorithm which is pseudo-polynomial runs in time polynomial in the *value* of its input, but exponential in the *size* of the input when it is reasonably represented (i.e. not in unary). For example, an algorithm which runs in $O(nC)$ time and space is pseudo-polynomial since C can be input to the algorithm in $\log C$ bits. Even though $O(nC)$ *seems* to be a polynomial time algorithm, it requires time exponential in the *size* of C.

In this section a quick overview of prior work on the ITKP, OTKP, and PCKP is given. Technical details are mostly omitted, since they are described thoroughly in Chapters 3 and 4. In these chapters the same techniques are used and expanded upon to develop new algorithms for the ITKP, RPCP, and RPCP+.

2.4.1 Subtree density method

Ibarra and Kim [16] seem to be the first to have considered the ITKP, and they developed a PTAS for the problem. Their method, which is analogous to the PTAS

for the KP from [26], is based on the idea of repeatedly selecting subtrees which have high profit per unit of weight, or high *density*. The density of a subtree rooted at vertex v is defined as:

$$\delta(S(v)) = \frac{\sum_{v' \in S(v)} p(v')}{\sum_{v' \in S(v)} w(v')} \ . \tag{2.4}$$

This *subtree density method* entails greedily filling the knapsack capacity with subtrees of maximum density, until the next most dense subtree cannot fit. Suppose the optimal solution has value Q and the subtree density method yields a solution with value Q_0. Ibarra and Kim showed that the difference in profit $Q - Q_0$ is less than than the most profitable subtree in the optimal solution. They also showed that filling the knapsack using the subtree density method can be done in $O(n^2)$ time.

Their PTAS works by choosing sets of disjoint subtrees and placing them in the knapsack. The subtree density method is then run on what remains of the tree after these subtrees have been removed. The number of different sets tried by the PTAS is based on the value of the desired error bound, ε, where ε is a unit fraction. If an error bound of ε is desired, all possible sets of $1/\varepsilon - 1$ or fewer disjoint subtrees are tried, and this number of sets is no more than

$$\sum_{k=1}^{1/\varepsilon - 1} \binom{n}{k} \leq (1/\varepsilon - 1) n^{1/\varepsilon - 1} \ . \tag{2.5}$$

Since the subtree density method must be run on each subset, the total running time of the PTAS becomes:

$$O(\frac{n^{\frac{1}{\varepsilon}+1}}{\varepsilon}) \ . \tag{2.6}$$

Furthermore, for the special case where $\varepsilon = \frac{1}{2}$, they gave an improved algorithm that runs in $O(n^2)$ time and $\Theta(n)$ space. This algorithm is discussed in great detail in Chapter 4, where a new algorithm is proposed which matches this error bound and provides a superior worst case running time of $O(n \log n)$, for the case where the in-tree is constructed and given as input.

2.4.2 Dynamic programming algorithms

By far the most important work on the tree knapsack problems can be attributed to Johnson and Niemi [17]. First, they showed that the "bottom-up" dynamic programming approach on trees, used by Lukes [23] to solve a related tree partition problem, can be applied to both the OTKP and ITKP to solve them in $\Theta(nQ^2)$ time and space. Then, they presented an algorithm that solves instances of the OTKP in $\Theta(nQ)$ time and space, using their so-called "left-right" dynamic programming method for trees. Their left-right approach therefore benefits from a factor of Q time speedup over the bottom-up approach.

Although they did not directly apply the left-right method to the ITKP, they showed that a profit maximization instance of the ITKP is transformable into a profit minimization instance of the OTKP. Thus, their left-right approach can be adapted to solve instances of in-trees in $\Theta(n(P-Q))$ time and space. In comparison, in Chapter 3 it is shown that the left-right method can be adapted to achieve a matching bound of $\Theta(nQ)$ for the ITKP.

Johnson and Niemi also gave a FPTAS for the OTKP based on their dynamic programming algorithm, which requires $\Theta(n^2(\frac{1}{\varepsilon} + \log n))$ time and $\Theta(n^2/\varepsilon)$ space. However, due to the P term when the algorithm is applied to the ITKP, this FPTAS yields a running time of $\Theta(n^3/\varepsilon)$ for that problem. By using the dynamic programming algorithm from Chapter 3 in conjunction with the $\frac{1}{2}$-approximation algorithm from Chapter 4, this bound is improved to $\Theta(n^2/\varepsilon)$ for the ITKP.

Later, Cho and Shaw modified the left-right approach for the OTKP to be pseudo-polynomial in C rather than Q, yielding a $\Theta(nC)$ time and space algorithm for the OTKP [7]. Because of the relationship between the ITKP and OTKP, the minimization version of this algorithm can therefore be used to solve the ITKP in $\Theta(n(W-C))$ time and space[1]. As before, in Chapter 3 it is shown that a matching bound of $\Theta(nC)$ can be achieved for the ITKP.

Shaw and Cho have also developed a branch and bound algorithm that is the current fastest method of solving instances of OTKP in practice [28]. The key operation of this algorithm is the identification of the "critical item" of the ITKP. Finding this critical item is done using an algorithm similar to the subtree density method of [16], and takes $O(n^2)$ time. Therefore, the results in Chapter 4 have relevance to this branch and bound algorithm, as they improve its worst case time

[1]Even though this is not explicitly mentioned in that paper.

Source	Problem	
	ITKP	OTKP
Ibarra and Kim [16] (ε-approximation)	$O(n^{\frac{1}{\varepsilon}+1}/\varepsilon)$ time	–
Ibarra and Kim [16] ($\frac{1}{2}$-approximation)	$O(n^2)$ time, $\Theta(n)$ space	–
Johnson and Niemi [17] (Bottom-up method)	$\Theta(nQ^2)$ time $\Theta(nQ)$ space	$\Theta(nQ^2)$ time $\Theta(nQ)$ space
Johnson and Niemi [17] (Left-right method)	$\Theta(n(P-Q))$ time/space	$\Theta(nQ)$ time/space
Johnson and Niemi [17] (ε-approximation)	$\Theta(n^3/\varepsilon)$ time/space	$\Theta(n^2(1/\varepsilon + \log n)$ time $\Theta(n^2/\varepsilon)$ space
Cho and Shaw [28]	–	$\Theta(nC)$

Table 2.1: A comparison of algorithms for the ITKP and OTKP by source.

bound.

2.4.3 Other work on the PCKP

Several others algorithms have been developed for the PCKP restricted to different classes of precedence graphs. These classes include series parallel graphs [5], two-dimensional partial orders [20], interval orders and bipartite convex orders [33], and general directed acyclic graphs [27]. Because this thesis is narrowed in scope to in-trees, exploration of these other graph classes is left to the reader.

2.4.4 Summary

Several algorithms have been proposed for both the ITKP and OTKP. Table 2.1 compares the time and space requirements of these algorithms. Surprisingly, there is a discrepancy between the bounds for the ITKP and OTKP even they are very closely related. Chapter 3 addresses this discrepancy, and discusses these techniques described in this section in further detail.

Chapter 3

Dynamic programming algorithms

3.1 Overview and assumptions

In this chapter several dynamic programming algorithms are presented for solving the In-tree Knapsack Problem (ITKP), Routing Prefix Caching Problem (RPCP), and Routing Prefix Caching Problem with Punt Prefixes (RPCP+). These algorithms make use of two different methods for doing dynamic programming on trees. The first, called the bottom-up method [23, 17], is presented to help clarify the advantages of second, called the left-right method [17].

For all of the algorithms, it is assumed that the precedence tree[1] is input as a binary tree, and does not need to be constructed. Items with more than two predecessors can be supported by adding "dummy" vertices that have zero profit and weight. Similarly, any forest of two or more trees can be converted into a binary tree by adding a number of dummy vertices at the root. For a forest with n vertices, the equivalent binary tree with dummy vertices will have at most $2n - 1$ vertices. In all of these cases, it is assumed that dummy vertices are added to make the resultant binary tree as balanced as possible. Figure 3.1 illustrates how the tree from Figure 2.4 can be modified into binary form.

The assumption of receiving a tree structure as input may not be appropriate for some applications of ITKP, but in the case of prefix caching, it is reasonable to assume that a tree-like data structure is used to manage the TCAM hardware. The extra requirement that the tree be in binary format is for notational convenience,

[1]The terms in-tree and tree are used interchangeably from now on, as all of the problems discussed are variants of the ITKP.

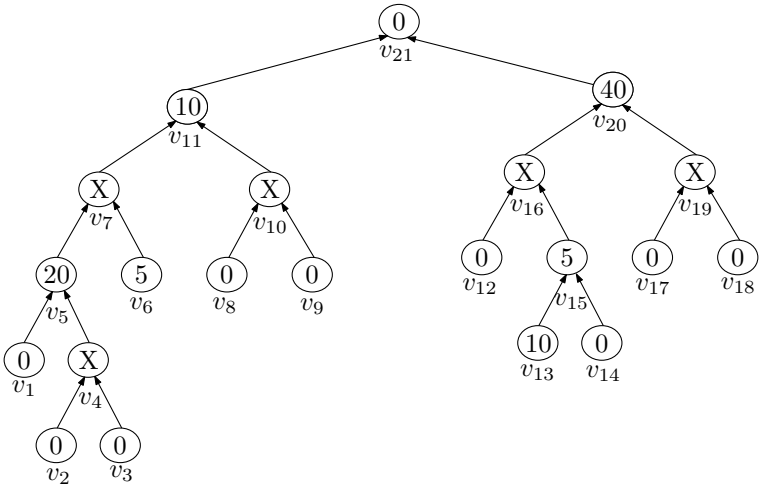

Figure 3.1: An example of how a tree where vertices with more than two predecessors can be converted into a binary tree. The vertices marked with 'X' are dummy vertices, each of which have a weight and profit of zero.

and the algorithms can be modified to resemble those in [17], which directly deal with vertices of arbitrary degree.

Additionally, each of the dynamic programming algorithms presented here traverses the tree in post-order. Denote vertex v_i as the i-th vertex visited during the post-order traversal, $1 \leq i \leq n$, and v_n as the root of the in-tree. The reason for this ordering is so that the optimal solution for vertex v_i is computed after the solutions for all other vertices in $S(v_i)$.

3.2 Bottom-up method for the ITKP

3.2.1 Description

Johnson and Niemi were the first to apply the bottom-up method to the ITKP [17]. The bottom-up method works by combining the optimal solutions to sub-problems, starting at the leaves and ending at the root of the tree. Each vertex is considered for all capacities k, $0 \leq k \leq C$. For a given vertex and capacity, finding the optimal solution is a straightforward process. If there is enough room to take the entire subtree, then that is the right choice, as it clearly maximizes the profit for that sub-problem. Otherwise, a scan needs to be conducted to find an optimal combination of the solutions from the left and right children.

First a recursive formula is described for performing the dynamic programming in terms of solution weights, followed by an alternate formulation which is done in terms of solution profits. The above description of how the algorithm works remains the same, except that instead of maximizing profit for the target weight, the profit version of the algorithm attempts to find the minimum weight solution that achieves the target profit. The reason the profit version is important—other than for providing an alternate parameter over which to do the optimization—is because it, unlike the weight version, can be used to derive an FPTAS [13, 17]. The exact method for turning the profit versions into an FPTAS is described in Section 3.5. This section closes with a description of how to modify the weight version of the recursion to improve the running time for the case when each vertex has unit weight; otherwise known as the RPCP.

3.2.2 Dynamic programming by solution weight

Define a set of vertices V' as *feasible* for a given capacity k, if $\sum_{v \in V'} w(v) \leq k$ and V' is closed under predecessor. Let the $n \times C$ dynamic programming table T_{BU} store the maximum profit feasible subset of $S(v_i)$ for capacity k in each entry $\mathrm{T}_{\mathrm{BU}}[v_i, k]$. The following three cases determine how $\mathrm{T}_{\mathrm{BU}}[v_i, k]$ is computed:

- v_i is a leaf,

$$\mathrm{T}_{\mathrm{BU}}[v_i, k] = \begin{cases} p(v_i) & \text{if } k \geq w(v_i), \\ 0 & \text{otherwise.} \end{cases} \tag{3.1}$$

- v_i has a single child v_c,

$$\mathrm{T}_{\mathrm{BU}}[v_i, k] = \begin{cases} p(S(v_i)) & \text{if } k \geq w(S(v_i)), \\ \mathrm{T}_{\mathrm{BU}}[v_c, k] & \text{otherwise.} \end{cases} \tag{3.2}$$

- v_i has both a left and right child, v_l and v_r respectively,

$$\mathrm{T}_{\mathrm{BU}}[v_i, k] = \begin{cases} p(S(v_i)) & \text{if } k \geq w(S(v_i)), \\ \max\limits_{0 \leq j \leq k} \{\mathrm{T}_{\mathrm{BU}}[v_l, j] + \mathrm{T}_{\mathrm{BU}}[v_r, k-j]\} & \text{otherwise.} \end{cases} \tag{3.3}$$

Theorem 3.2.1. *The weight version of the bottom-up dynamic program for the ITKP requires $\Theta(nC^2)$ time and $\Theta(nC)$ space.*

Proof. The general case of the recursion, Equation 3.3, requires $\Theta(k)$ time. Since the algorithm is run C times on n vertices, this gives a running time of $\Theta(nC^2)$. The space bound is $\Theta(nC)$ for the $n \times C$ dynamic programming table. Correctness is straightforward, since the problem can be shown to exhibit optimal substructure by contradiction on Equation 3.3. □

Equations 3.1-3.3 compute the *value* of the optimal solution. Suppose the root of the in-tree is vertex r. The optimal solution value will be stored in the table entry $\mathrm{T_{BU}}[r, C]$. To find the actual set of vertices that achieve this value, standard dynamic programming techniques can be used which are described in [19]. One method would be to fill a second $n \times C$ table $\mathrm{X_{BU}}$ with the decision that was made at each vertex.

For example, suppose the decision to take the entire subtree is made at vertex v_i for capacity k. A special flag value TAKE could be used to indicate this decision, thus $\mathrm{X_{BU}}[v_i, k] = \textsc{take}$. Otherwise, suppose v_i has two children, and the decision was made to take a weight of j from the left child and $k - j$ from the right child. In this case, $\mathrm{X_{BU}}[v_i, k] = j$, which would indicate what capacity should be examined when the algorithm recurses to the left child (and right child implicitly).

It is important to note that maintaining such a table is up to the discretion of the implementer, as the recursions in Equations 3.1-3.3 are entirely reversible. For this reason, the topic of extracting the actual solution will not be discussed further, as it does not affect the asymptotic running time of any of the algorithms, and is mostly an implementation detail.

3.2.3 Numerical Example

Example 3.2.2. The bottom-up method, presented in Equations 3.1-3.3, would yield the dynamic programming table displayed in Table 3.1 when run on the tree in Figure 3.1 with $C = 10$.

3.2.4 Dynamic programming by solution profit

Let the $n \times C$ dynamic programming table $\mathrm{U_{BU}}$ store the **minimum** weight feasible subset of $S(v_i)$ having profit at least q in each entry $\mathrm{U_{BU}}[v_i, q]$. Since the dynamic programming is done over profits, the subsets need only be feasible with respect

T_{BU}											
	Capacity (C)										
Vertex	0	1	2	3	4	5	6	7	8	9	10
v_1	0	0	0	0	0	0	0	0	0	0	0
v_2	0	0	0	0	0	0	0	0	0	0	0
v_3	0	0	0	0	0	0	0	0	0	0	0
v_4	0	0	0	0	0	0	0	0	0	0	0
v_5	0	0	0	0	20	20	20	20	20	20	20
v_6	0	5	5	5	5	5	5	5	5	5	5
v_7	0	5	5	5	20	25	25	25	25	25	25
v_8	0	0	0	0	0	0	0	0	0	0	0
v_9	0	0	0	0	0	0	0	0	0	0	0
v_{10}	0	0	0	0	0	0	0	0	0	0	0
v_{11}	0	5	5	5	20	25	25	25	35	35	35
v_{12}	0	0	0	0	0	0	0	0	0	0	0
v_{13}	0	10	10	10	10	10	10	10	10	10	10
v_{14}	0	0	0	0	0	0	0	0	0	0	0
v_{15}	0	10	10	15	15	15	15	15	15	15	15
v_{16}	0	10	10	15	15	15	15	15	15	15	15
v_{17}	0	0	0	0	0	0	0	0	0	0	0
v_{18}	0	0	0	0	0	0	0	0	0	0	0
v_{19}	0	0	0	0	0	0	0	0	0	0	0
v_{20}	0	10	10	15	15	15	15	55	55	55	55
v_{21}	0	10	15	15	20	30	35	55	60	60	60

Table 3.1: The dynamic programming table, T_{BU}, created by the bottom-up method when run on the tree found in Figure 3.1. By examining table entry $T_{BU}[v_{21}, 10]$ it can be seen that the optimal solution has a profit value of 60.

to the total capacity C. If there is no feasible solution achieving profit q, then the table entry is denoted at ∞. The following three cases determine how $\mathrm{U}_{\mathrm{BU}}[v_i, q]$ is computed:

- v_i is a leaf,

$$\mathrm{U}_{\mathrm{BU}}[v_i, q] = \begin{cases} \infty & \text{if } q > p(v_i), \\ w(v_i) & \text{otherwise.} \end{cases} \tag{3.4}$$

- v_i has a single child v_c,

$$\mathrm{U}_{\mathrm{BU}}[v_i, q] = \begin{cases} \infty & \text{if } q > p(S(v_i)), \\ \mathrm{U}_{\mathrm{BU}}[v_c, q] & \text{if } w(S(v_i)) > C, \\ \min\left\{w(S(v_i)), \mathrm{U}_{\mathrm{BU}}[v_c, q]\right\} & \text{otherwise.} \end{cases} \tag{3.5}$$

- v_i has both a left and right child, v_l and v_r respectively,

$$\mathrm{U}_{\mathrm{BU}}[v_i, q] = \begin{cases} \infty & \text{if } q > p(S(v_i)), \\ \min\left[\begin{array}{l} \{u = \mathrm{U}_{\mathrm{BU}}[v_l, j] + \mathrm{U}_{\mathrm{BU}}[v_r, q-j] : \\ 0 \le j \le q, u \le C\} \end{array}\right] & \text{if } w(S(v_i)) > C, \\ \min\left[\begin{array}{l} \{w(S(v_i))\} \cup \\ \{u = \mathrm{U}_{\mathrm{BU}}[v_l, j] + \mathrm{U}_{\mathrm{BU}}[v_r, q-j] : \\ 0 \le j \le q, u \le C\} \end{array}\right] & \text{otherwise.} \end{cases} \tag{3.6}$$

It is important to note the $w(S(v_i))$ term which appears in Equations 3.5 and 3.6. This term is necessary for the case when the alternative choice has infinite weight.

Theorem 3.2.3 (From [17])**.** *The profit version of the bottom-up dynamic program for the ITKP requires $\Theta(nQ^2)$ time and $\Theta(nQ)$ space.*

Proof. The general case of the recursion, Equation 3.6, requires $\Theta(q)$ time. Suppose a value of $\tilde{Q}$ is supplied to the algorithm, such that the memoized version of the above recursion is run on each vertex for all values $0 \le q \le \tilde{Q}$. This algorithm will require $\Theta(n\tilde{Q}^2)$ time and $\Theta(n\tilde{Q})$ space.

Suppose the root of the tree is vertex r. After the algorithm terminates, $\mathrm{U}_{\mathrm{BU}}[r, \tilde{Q}]$ will contain some value. If $\mathrm{U}_{\mathrm{BU}}[r, \tilde{Q}] = \infty$ then it means that $Q < \tilde{Q}$, and

$$Q = \max_{0 \leq k \leq \tilde{Q}} \{k : \mathrm{U}_{\mathrm{BU}}[r, k] \neq \infty\} \quad . \tag{3.7}$$

Otherwise, the value of $\tilde{Q}$ can be doubled, and the algorithm can continue. By starting with a value

$$\tilde{Q} = \max_{v \in V} \{p(v)\} \quad , \tag{3.8}$$

the algorithm can be run, doubling $\tilde{Q}$ until $Q < \tilde{Q}$. Therefore, the algorithm requires $\Theta(nQ^2)$ time and $\Theta(nQ)$ space, since when the algorithm terminates, $\tilde{Q}/2 \leq Q \leq \tilde{Q}$. □

3.2.5 Improved time bound for the RPCP

The bottom-up method, as presented, implies a worst case running time of $O(n^3)$ for the RPCP. This is because $0 \leq C < n$ for the RPCP. However, it can be shown that the running time of the bottom-up method can be improved for RPCP through a slight modification to the algorithm. This is done by exploiting the fact that for the RPCP, $w(v) \leq 1$ for all $v \in V$.

Theorem 3.2.4. *The running time of the bottom-up method can be improved to* $O(\min\{nC^2, n^2\})$ *for the case where* $w(v) \leq 1$ *for all* $v \in V$.

Proof. Consider the following two facts. First, for each vertex $v \in V$, $\mathrm{T}_{\mathrm{BU}}[v, i]$ does not need to be computed for $|S(v)| < i \leq C$, since $w(S(v)) \leq |S(v)|$. In this case, $\mathrm{T}_{\mathrm{BU}}[v, i]$ is guaranteed to be equal to $p(S(v))$. Second, when $\mathrm{T}_{\mathrm{BU}}[v, i]$ is being computed for an i that is larger than the cardinality of the smaller child subtree of v, denoted $|S(v_c)|$, all i comparisons do not need to be made. Rather, $|S(v_c)| + 1$ comparisons suffice, since it is only necessary to determine 'how much' of $S(v_c)$ to take when there is enough capacity to take the entire subtree.

Suppose a problem instance consists of a binary tree with n vertices, rooted at vertex r. Let one of the children of r be a subtree containing α vertices, and the other be a child subtree containing β vertices, where, without loss of generality,

$\alpha \geq \beta$. The running time of the bottom-up dynamic programming method which exploits the above two facts can be expressed as:

$$T(n) = T(\alpha) + T(\beta) + \Delta(r) \tag{3.9}$$

where $\Delta(r)$ is the cost of computing $\mathrm{T_{BU}}[r, i]$ for $0 \leq i \leq n = \alpha + \beta + 1$. Table 3.2 indicates how much work has to be done for each of the $\alpha + \beta + 1$ table entries. $\Delta(r)$ is therefore the sum of each entry in the 'cost' row. Effectively, this is the amount of work to solve the RPCP at a given vertex in the dependency tree for all cache sizes. Therefore, $T(n)$ is an upper bound on the running time of our modified bottom up method for any capacity in the case where all vertices have unit weight.

capacity	1	2	...	β	...	α	$\alpha+1$	...	$\alpha+\beta$	$\alpha+\beta+1$
cost	2	3	...	$\beta+1$	...	$\beta+1$	β	...	2	1

Table 3.2: For a subtree of size $\alpha+\beta+1$ this table indicates how many comparisons must be done to compute the optimal solution for each capacity over the interval $[1, \alpha+\beta+1]$.

The sum, $\Delta(r)$, can be expressed as:

$$\Delta(r) = 2\sum_{i=1}^{\beta}(i+1) + (\alpha-\beta)(\beta+1) + 1 \tag{3.10}$$

$$= \alpha\beta + \alpha + 2\beta + 1 \tag{3.11}$$

Using a guess of $T(n) \leq n^2$ and substituting this and Equation 3.11 back into Equation 3.9, the inequality can be shown to hold. Therefore the running time of the bottom-up method for the RPCP is $O(\min\{nC^2, n^2\})$.

□

3.3 Left-right method for the ITKP

3.3.1 Description

The left-right method, first presented in [17] for the OTKP, is a significant improvement over the bottom-up method which results in a factor of C speedup. Here,

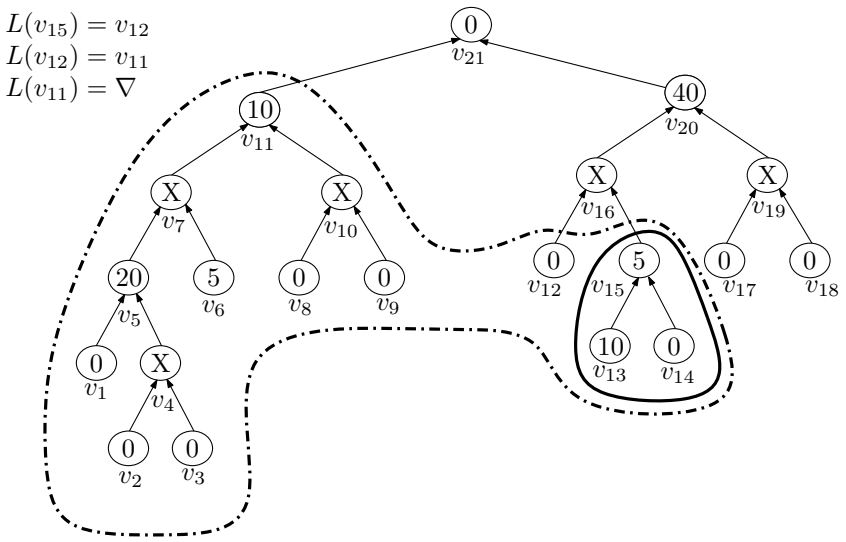

Figure 3.2: Suppose the input to the bottom-up and left-right dynamic programming algorithms is the above tree. During the computation of $\mathrm{T}_{\mathrm{BU}}[v_{15}, k]$, the goal is to find the optimal legal combination of vertices within $S(v_{15})$; indicated by the solid boundary. In contrast, for $\mathrm{T}_{\mathrm{LR}}[v_{15}, k]$, not only is $S(v_{15})$ considered, but all vertices previously visited; as indicated by the dot-dashed boundary.

the method is adapted and applied in an analogous way to the ITKP, to achieve a better running time. As with the bottom-up method, both the weight and profit versions of the dynamic programming recursion will be discussed.

For the bottom-up method, solving a sub-problem at a vertex v_i involves finding the maximum profit feasible subset of vertices in $S(v_i)$. In contrast, the goal of the left-right method is to find the maximum profit feasible subset of all previously visited vertices, $\{v_1, ..., v_i\}$. Recall that the vertices of the tree are visited in post-order. At any point during the post-order traversal, the set of visited vertices induces a forest. The current vertex v_i can be thought of as the root of the rightmost subtree in this forest. Define $L(v_i) = v_j$, such that $1 \leq j < i$ and j is the maximum index such that $v_j \notin S(v_i)$. $L(v_i)$ points to the root of the subtree directly to the left of the subtree rooted at v_i. Figure 3.2 provides an illustration of this concept for clarification. If $L(v_i)$ is undefined, because all previously visited vertices are contained in $S(v_i)$, then this is denoted by $L(v_i) = \nabla$.

The key observation is that for each vertex v_i and capacity k, only a single comparison is necessary to determine whether v_i is part of the optimal solution for the capacity k. If v_i is not part of the solution, then the optimal solution at v_i is identical to the solution at v_{i-1}. Otherwise, if v_i is part of the solution, then so must $S(v_i)$ too, as well as the optimal solution at vertex $L(v_i)$ for the remaining

weight. Since these are the only two possibilities, filling the table is much faster than with the bottom-up method.

3.3.2 Dynamic programming by solution weight

Let the $n \times C$ dynamic programming table T_{LR} store the maximum profit feasible subset of $\{v_1, \ldots v_i\}$ for capacity k in each entry $\mathrm{T}_{\mathrm{LR}}[v_i, k]$. The following two cases can be used to compute each entry $\mathrm{T}_{\mathrm{LR}}[v_i, k]$:

- If the first vertex in the traversal is being visited, i.e. $i = 1$, then

$$\mathrm{T}_{\mathrm{LR}}[v_1, k] = \begin{cases} p(v_1) & \text{if } k \geq w(v_1), \\ 0 & \text{otherwise.} \end{cases} \tag{3.12}$$

- For all other vertices v_i, $1 < i \leq n$,

$$\mathrm{T}_{\mathrm{LR}}[v_i, k] = \begin{cases} \mathrm{T}_{\mathrm{LR}}[v_{i-1}, k] & \text{if } k < w(S(v_i)), \\ p(S(v_i)) & \text{if } L(v_i) = \nabla, \\ \max \left\{ \begin{array}{l} \mathrm{T}_{\mathrm{LR}}[v_{i-1}, k], \;\; p(S(v_i)) + \\ \mathrm{T}_{\mathrm{LR}}[L(v_i), k - w(S(v_i))] \end{array} \right\} & \text{otherwise.} \end{cases} \tag{3.13}$$

Theorem 3.3.1. *The weight version of the left-right dynamic program for the ITKP requires $\Theta(nC)$ time and space.*

Proof. Each entry in the $n \times C$ dynamic programming table can be computed in constant time, since $L(v_i)$ is precomputable. Therefore, the total running time of the algorithm is $\Theta(nC)$, and the space consumption is dominated by the size of the dynamic programming table. □

It is important to note that a more general version of this technique has been applied to directed acyclic graphs [27]. However, the resulting algorithm for directed acyclic graphs is *not* pseudo-polynomial, as no pseudo-polynomial algorithm can exist for the general PCKP [17].

T_{LR}											
	Capacity (C)										
Vertex	0	1	2	3	4	5	6	7	8	9	10
v_1	0	0	0	0	0	0	0	0	0	0	0
v_2	0	0	0	0	0	0	0	0	0	0	0
v_3	0	0	0	0	0	0	0	0	0	0	0
v_4	0	0	0	0	0	0	0	0	0	0	0
v_5	0	0	0	0	20	20	20	20	20	20	20
v_6	0	5	5	5	20	25	25	25	25	25	25
v_7	0	5	5	5	20	25	25	25	25	25	25
v_8	0	5	5	5	20	25	25	25	25	25	25
v_9	0	5	5	5	20	25	25	25	25	25	25
v_{10}	0	5	5	5	20	25	25	25	25	25	25
v_{11}	0	5	5	5	20	25	25	25	35	35	35
v_{12}	0	5	5	5	20	25	25	25	35	35	35
v_{13}	0	10	15	15	20	30	35	35	35	45	45
v_{14}	0	10	15	15	20	30	35	35	35	45	45
v_{15}	0	10	15	15	20	30	35	35	40	45	45
v_{16}	0	10	15	15	20	30	35	35	40	45	45
v_{17}	0	10	15	15	20	30	35	35	40	45	45
v_{18}	0	10	15	15	20	30	35	35	40	45	45
v_{19}	0	10	15	15	20	30	35	35	40	45	45
v_{20}	0	10	15	15	20	30	35	55	60	60	60
v_{21}	0	10	15	15	20	30	35	55	60	60	60

Table 3.3: The dynamic programming table, T_{LR}, created by the left-right method when run on the tree found in Figure 3.1. By examining table entry $T_{LR}[v_{21}, 10]$ it can be seen that the optimal solution has a profit value of 60.

3.3.3 Numerical Example

Example 3.3.2. The left-right method would yield the dynamic programming table displayed in Table 3.3 when run on the tree in Figure 3.1 with $C = 10$.

3.3.4 Dynamic programming by solution profit

As with the bottom-up method, the left-right method can be done over profits as follows. Let the table U_{LR} store the minimum weight feasible solution having profit q that can be obtained from the set of vertices $\{v_1, \ldots v_i\}$ in each entry $U_{LR}[v_i, q]$. The following three cases can be used to compute each entry $U_{LR}[v_i, q]$:

- If the first vertex in the traversal is being visited, or $i = 1$, then

$$\mathrm{U}_{\mathrm{LR}}[v_1, q] = \begin{cases} \infty & \text{if } q > p(v_1), \\ w(v_1) & \text{otherwise.} \end{cases} \tag{3.14}$$

- If $L(v_i) = \nabla$, then

$$\mathrm{U}_{\mathrm{LR}}[v_i, q] = \begin{cases} \mathrm{U}_{\mathrm{LR}}[v_{i-1}, q] & \text{if } w(S(v_i)) > C, \\ \infty & \text{if } p(S(v_i)) < q, \\ \min\{\mathrm{U}_{\mathrm{LR}}[v_{i-1}, q], w(S(v_i))\} & \text{otherwise.} \end{cases} \tag{3.15}$$

- Otherwise, let $r = w(S(v_i)) + \mathrm{U}_{\mathrm{LR}}[L(v_i), q - p(S(v_i))]$,

$$\mathrm{U}_{\mathrm{LR}}[v_i, q] = \begin{cases} \mathrm{U}_{\mathrm{LR}}[v_{i-1}, q] & \text{if } w(S(v_i)) > C, \\ \min\{\mathrm{U}_{\mathrm{LR}}[v_{i-1}, q], r\} & \text{if } p(S(v_i)) < q \text{ and } r \leq C, \\ \mathrm{U}_{\mathrm{LR}}[v_{i-1}, q] & \text{if } p(S(v_i)) < q \text{ and } r > C, \\ \min\{\mathrm{U}_{\mathrm{LR}}[v_{i-1}, q], w(S(v_i))\} & \text{otherwise.} \end{cases} \tag{3.16}$$

Theorem 3.3.3. *The profit version of the left-right dynamic program for the ITKP requires $\Theta(nQ)$ time and space.*

Proof. As with the weight version, each entry in the dynamic programming table requires constant time to fill. Using the same iterative search procedure as in Theorem 3.2.3, the desired bounds are achieved. □

3.4 The bottom-up method for the RPCP+

3.4.1 Description

The methods presented in this chapter for the ITKP can by modified in order to find the optimal solution to the Routing Prefix Caching Problem with Punt Prefixes (RPCP+) as well. This section describes how to modify the bottom-up method to correctly deal with punt vertices in such a way to only increase the time and space complexity by a constant factor. The same kind of modification can

likely be performed on the left-right method, however the constants are larger and the algorithm is less clear. It is also important to note that although vertices have unit weight in the RPCP+, the algorithm presented here can be used to handle arbitrary weights.

The main idea of this approach is to use two separate $n \times C$ dynamic programming tables, $\mathrm{T_{IN}}$ and $\mathrm{T_{EX}}$. Each entry $\mathrm{T_{IN}}[v_i, k]$ corresponds to the maximum profit solution that can be obtained from the subtree rooted at vertex v_i with weight at most k *that includes vertex* v_i. Each entry $\mathrm{T_{EX}}[v_i, k]$ corresponds to the maximum profit solution that can be obtained from the subtree rooted at vertex v_i with weight at most k *that excludes vertex* v_i.

When determining which vertices should be toggled to be punt vertices, it seems necessary to have access to the two tables described above. This is because the decision to toggle a child v_c of a given vertex v_i to be a punt vertex can only be made efficiently if there is knowledge of both the optimal solution which contains v_c, and the optimal solution which does not contain v_c. By comparing these two values it is easier to determine whether or not there is a benefit to toggling v_c to be a punt vertex $B(v_c)$. For instance, if v_i has high profit, and $B(v_c)$ allows v_i to be part of a feasible solution for a given capacity, then this can be determined in a straightforward way if the two tables described are accessible.

3.4.2 Dynamic programming over solution weight

Recall that there are also dummy vertices present in the tree which do not correspond to any prefixes in the routing table. These vertices cannot be toggled to be punt prefixes, as a punt vertex can only be an actual prefix that exists in the routing table. Let

$$M(v_i) = \begin{cases} 1 & \text{if } v_i \text{ is a dummy vertex,} \\ 0 & \text{otherwise.} \end{cases} \tag{3.17}$$

If it is not possible to take a vertex v_i for a given capacity k, then $\mathrm{T_{IN}}[v_i, k] = -\infty$. The values of $\mathrm{T_{IN}}[v_i, k]$ and $\mathrm{T_{EX}}[v_i, k]$ can be computed as follows:

- If v_i is a leaf:

$$\mathrm{T}_{\mathrm{IN}}[v_i, k] = \begin{cases} p(v_i) & \text{if } k \geq w(v), \\ -\infty & \text{otherwise.} \end{cases} \tag{3.18}$$

$$\mathrm{T}_{\mathrm{EX}}[v_i, k] = \begin{cases} p(v_i) & \text{if } k \geq w(v), \\ 0 & \text{otherwise.} \end{cases} \tag{3.19}$$

- If v_i has a single child, v_c, let $r = k - (w(v_i) + w(v_c))$:

$$\mathrm{T}_{\mathrm{EX}}[v_i, k] = \max\left\{\mathrm{T}_{\mathrm{EX}}[v_c, k], \mathrm{T}_{\mathrm{IN}}[v_c, k]\right\} \tag{3.20}$$

$$\mathrm{T}_{\mathrm{IN}}[v_i, k] = \begin{cases} p(S(v_i)) & \text{if } k \geq w(S(v_i)), \\ \max\left\{ \begin{array}{l} p(v_i) + \mathrm{T}_{\mathrm{EX}}[v_c, r], \\ p(v_i) + \mathrm{T}_{\mathrm{IN}}[v_c, k - (w(v_i))] \end{array} \right\} & \text{if } r \geq 0 \text{ and } M(v_c) = 0, \\ p(v_i) + \mathrm{T}_{\mathrm{IN}}[v_c, k - (w(v_i))] & \text{if } k \geq w(v_i) \\ -\infty & \text{otherwise.} \end{cases} \tag{3.21}$$

- If v_i has both a left and right child, v_l and v_r respectively:

$$\mathrm{T}_{\mathrm{EX}}[v_i, k] = \max\left\{ \begin{array}{l} \max\left\{\mathrm{T}_{\mathrm{EX}}[v_l, j], \mathrm{T}_{\mathrm{IN}}[v_l, j]\right\} + \\ \max\left\{\mathrm{T}_{\mathrm{EX}}[v_r, k-j], \mathrm{T}_{\mathrm{IN}}[v_r, k-j]\right\} : 0 \leq j \leq k \end{array} \right\} \tag{3.22}$$

$$\mathrm{T}_{\mathrm{IN}}[v_i, k] = \begin{cases} p(S(v_i)) & \text{if } k \geq w(S(v_i)), \\ \max\left[S_{tt} \cup S_{lt} \cup S_{tl} \cup S_{ll} \cup \{-\infty\}\right] & \text{otherwise,} \end{cases} \tag{3.23}$$

where

$$S_{tt} = \left\{ \begin{array}{l} p(v_i) + \mathrm{T}_{\mathrm{IN}}[v_l, j] + \mathrm{T}_{\mathrm{IN}}[v_r, k_{tt} - j] : \\ 0 \leq j \leq k_{tt}, \;\; k_{tt} \geq 0 \end{array} \right\} , \tag{3.24}$$

$$S_{lt} = \left\{ \begin{array}{l} p(v_i) + \mathrm{T}_{\mathrm{IN}}[v_r, j] + \mathrm{T}_{\mathrm{EX}}[v_l, k_{lt} - j] : \\ 0 \leq j \leq k_{lt}, \;\; k_{lt} \geq 0, \;\; M(v_l) = 0 \end{array} \right\} , \tag{3.25}$$

$$S_{tl} = \left\{ \begin{array}{l} p(v_i) + \mathrm{T}_{\mathrm{IN}}[v_l, j] + \mathrm{T}_{\mathrm{EX}}[v_r, k_{tl} - j] : \\ 0 \leq j \leq k_{tl}, \;\; k_{tl} \geq 0, \;\; M(v_r) = 0 \end{array} \right\} , \tag{3.26}$$

$$S_{ll} = \left\{ \begin{array}{l} p(v_i) + \mathrm{T}_{\mathrm{EX}}[v_l, j] + \mathrm{T}_{\mathrm{EX}}[v_r, k_{ll} - j] : \\ 0 \leq j \leq k_{ll}, \;\; k_{ll} \geq 0, \;\; M(v_l) = 0, \;\; M(v_r) = 0 \end{array} \right\} , \tag{3.27}$$

and

$$k_{tt} = k - w(v_i) \ , \tag{3.28}$$

$$k_{lt} = k - (w(v_i) + w(v_l)) \ , \tag{3.29}$$

$$k_{tl} = k - (w(v_i) + w(v_r)) \ , \tag{3.30}$$

$$k_{ll} = k - (w(v_i) + w(v_l) + w(v_r)) \ . \tag{3.31}$$

Each set S_{tt}, S_{lt}, S_{tl}, and S_{ll} correspond to computing the optimal solution when both children are selected, the left child is covered with a punt vertex, the right child is covered with a punt vertex, or both children are covered with punt vertices, respectively. Note the constraints on each set are defined so that they are non-empty only when k is large enough to store the current vertex v_i, plus whatever combination of punt vertices are required.

Theorem 3.4.1. *The bottom-up dynamic program for the RPCP+ requires* $\Theta(nC^2)$ *time and* $\Theta(nC)$ *space.*

Proof. Though the three cases are messier looking than the original bottom-up method, they add no more than a constant factor to the running time, because finding the max of S_{tt}, S_{lt}, S_{tl}, and S_{ll} requires $\Theta(k)$ time. Since the memoized versions of these recursions are run on each vertex for $0 \leq k \leq C$, the overall running time is $\Theta(nC^2)$. Finally, because just two dynamic programming tables need to be maintained, each of size $n \times C$, the space required is still $\Theta(nC)$. □

3.4.3 Numerical Example

Example 3.4.2. The bottom-up method the allows for punt vertices would yield the dynamic programming table displayed in Tables 3.4 and 3.5 when run on the tree in Figure 3.1 with $C = 10$. The optimal profit that can be obtained is $\max\{\mathrm{T}_{\mathrm{EX}}[r, C], \mathrm{T}_{\mathrm{IN}}[r, C]\}$ where r is the root of the precedence tree. In this case, the optimal solution is 70, as in Example 2.3.3.

T_{IN}											
	Capacity (C)										
Vertex	0	1	2	3	4	5	6	7	8	9	10
v_1	$-\infty$	0	0	0	0	0	0	0	0	0	0
v_2	$-\infty$	0	0	0	0	0	0	0	0	0	0
v_3	$-\infty$	0	0	0	0	0	0	0	0	0	0
v_4	$-\infty$	$-\infty$	0	0	0	0	0	0	0	0	0
v_5	$-\infty$	$-\infty$	$-\infty$	$-\infty$	20	20	20	20	20	20	20
v_6	$-\infty$	5	5	5	5	5	5	5	5	5	5
v_7	$-\infty$	$-\infty$	5	5	5	25	25	25	25	25	25
v_8	$-\infty$	0	0	0	0	0	0	0	0	0	0
v_9	$-\infty$	0	0	0	0	0	0	0	0	0	0
v_{10}	$-\infty$	$-\infty$	0	0	0	0	0	0	0	0	0
v_{11}	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	15	15	15	35	35	35
v_{12}	$-\infty$	0	0	0	0	0	0	0	0	0	0
v_{13}	$-\infty$	10	10	10	10	10	10	10	10	10	10
v_{14}	$-\infty$	0	0	0	0	0	0	0	0	0	0
v_{15}	$-\infty$	$-\infty$	$-\infty$	15	15	15	15	15	15	15	15
v_{16}	$-\infty$	$-\infty$	0	10	15	15	15	15	15	15	15
v_{17}	$-\infty$	0	0	0	0	0	0	0	0	0	0
v_{18}	$-\infty$	0	0	0	0	0	0	0	0	0	0
v_{19}	$-\infty$	$-\infty$	0	0	0	0	0	0	0	0	0
v_{20}	$-\infty$	$-\infty$	$-\infty$	$-\infty$	$-\infty$	40	50	55	55	55	55
v_{21}	$-\infty$	$-\infty$	$-\infty$	0	10	15	15	40	50	55	60

Table 3.4: The dynamic programming table (T_{IN}) created by the bottom-up method that allows for punt vertices when run on the tree found in Figure 3.1.

T_{EX}											
	Capacity (C)										
Vertex	0	1	2	3	4	5	6	7	8	9	10
v_1	0	0	0	0	0	0	0	0	0	0	0
v_2	0	0	0	0	0	0	0	0	0	0	0
v_3	0	0	0	0	0	0	0	0	0	0	0
v_4	0	0	0	0	0	0	0	0	0	0	0
v_5	0	0	0	0	0	0	0	0	0	0	0
v_6	0	0	0	0	0	0	0	0	0	0	0
v_7	0	5	5	5	20	25	25	25	25	25	25
v_8	0	0	0	0	0	0	0	0	0	0	0
v_9	0	0	0	0	0	0	0	0	0	0	0
v_{10}	0	0	0	0	0	0	0	0	0	0	0
v_{11}	0	5	5	5	20	25	25	25	25	25	25
v_{12}	0	0	0	0	0	0	0	0	0	0	0
v_{13}	0	0	0	0	0	0	0	0	0	0	0
v_{14}	0	0	0	0	0	0	0	0	0	0	0
v_{15}	0	10	10	10	10	10	10	10	10	10	10
v_{16}	0	10	10	15	15	15	15	15	15	15	15
v_{17}	0	0	0	0	0	0	0	0	0	0	0
v_{18}	0	0	0	0	0	0	0	0	0	0	0
v_{19}	0	0	0	0	0	0	0	0	0	0	0
v_{20}	0	10	10	15	15	15	15	15	15	15	15
v_{21}	0	10	15	15	20	40	50	55	60	60	70

Table 3.5: The dynamic programming table (T_{EX}) created by the bottom-up method that allows for punt vertices when run on the tree found in Figure 3.1.

3.5 FPTAS for the ITKP

In [17] an FPTAS for the both the ITKP and OTKP was developed based on the dynamic programming algorithm specifically designed for the OTKP. Since the OTKP dynamic programming algorithm required $\Theta(n(P-Q))$ time to solve the ITKP, the resultant FPTAS for the ITKP required $\Theta(n^3/\varepsilon)$ time and space, whereas the FPTAS for the OTKP required only $\Theta(n^2(1/\varepsilon+\log n))$ time and $\Theta(n^2/\varepsilon)$ space. Here it is shown that the FPTAS for the ITKP can be improved to $\Theta(n^2/\varepsilon)$ time and space using the adapted left-right method presented in this chapter.

Theorem 3.5.1. *Using the left-right method (Equations 3.14 - 3.16), a $\Theta(n^2/\varepsilon)$ time and space FPTAS for the ITKP can be derived, for $0 < \varepsilon \leq 1$, provided a lower bound $\hat{Q}$ is known, where $\hat{Q} \leq Q \leq 2\hat{Q}$.*

Proof. The FPTAS works by creating a modified instance of the problem where the profits of vertices are scaled in order to guarantee the desired error bound, ε [17, 19]. Suppose the ITKP instance I consists of a graph $G = (V, E)$ and a knapsack capacity C. A modified instance I' can be created, consisting of $G' = (V', E')$ and capacity C in the following way. First, for each vertex $v \in V$, set the corresponding vertex $v' \in V'$ to have profit:

$$p(v') = \left\lfloor \frac{p(v)n}{\hat{Q}\varepsilon} \right\rfloor \ . \tag{3.32}$$

The left-right dynamic program over profits (Equations 3.14-3.16) can find the optimal solution to instance I' in $\Theta(n^2/\varepsilon)$ time and space, due to the fact that the optimal profit of instance I' lies somewhere in the range $(0, \frac{2n}{\varepsilon})$. Suppose the optimal solution for I is set $X \subseteq V$, and the solution from solving instance I' is $X' \subseteq V'$. To avoid overloading the term $\hat{Q}$, $p(X)$ and $p(X')$ will be referred to directly, which are the profit of the optimal solutions to instances I and I', respectively. Let $p'(X') = \frac{p(X')\hat{Q}\varepsilon}{n}$. The rounding process introduced an error of at most $\frac{\hat{Q}\varepsilon}{n}$ per vertex. Since there are n vertices, the total error is:

$$p(X) - p'(X') \leq \hat{Q}\varepsilon \leq p(X)\varepsilon \ . \tag{3.33}$$

Returning to the definition of an ε-approximation (Equation 2.3), it is the case that

$$\frac{p(X) - p'(X')}{p(X)} \leq \frac{p(X)\varepsilon}{p(X)} = \varepsilon \ , \tag{3.34}$$

which completes the proof. □

In Chapter 4 it is shown that finding $\hat{Q}$ can be done in $O(n \log n)$ time, which is dominated by the running time of the above FPTAS. It is also important to note that Theorem 3.5.1 provides the following weaker bound for an FPTAS derived from the bottom-up method.

Corollary 3.5.2. *Using the bottom-up method, a $\Theta(n^3/\varepsilon^2)$ time and $\Theta(n^2/\varepsilon)$ space FPTAS for the ITKP can be derived, for $0 < \varepsilon \leq 1$, provided a lower bound $\hat{Q}$ is known, where $\hat{Q} \leq Q \leq 2\hat{Q}$.*

3.6 Reducing storage requirements

In [17] a method is described for reducing the storage requirement of the dynamic programming algorithms for the OTKP. This same method can be applied to all of the dynamic programming algorithms presented in this chapter. However, here it is only described using the left-right method for the ITKP (see Section 3.3), though it can be applied to any of the other algorithms in an analogous way.

Recall that during the general case of the computation of $\mathrm{T}_{\mathrm{LR}}[v_i, k]$, for $1 \leq k \leq C$, only two other rows of the dynamic programming table need to be inspected. These rows are $\mathrm{T}_{\mathrm{LR}}[v_{i-1}, k]$ and $\mathrm{T}_{\mathrm{LR}}[L(v_i), k]$. Because so much of the dynamic programming table goes unused during this computation, it is only really necessary to keep a small number of table rows in memory in order to compute the optimal solution. Furthermore, many rows can be discarded after they are used, as they will never be used again in the future. Figure 3.3 (left) illustrates this fact by showing both which rows must be accessible during the computation of a vertex, as well as which rows will be needed in the future.

Theorem 3.6.1 (Based on method for the OTKP [17])**.** *To compute the optimal solution* value *for an instance of the ITKP requires $\Theta(nC)$ and $\Theta(C \log n)$ space.*

Proof. If the tree is highly skewed to the right, as in Figure 3.3 (left), the number of rows that must be stored can be $O(n)$. The number of rows that need to be stored is exactly equal to the maximum number of right branches in any path from the root to a given vertex. However, a clever trick can be used to circumvent this problem, since the concept of left and right is not strictly enforced. At a given vertex, v,

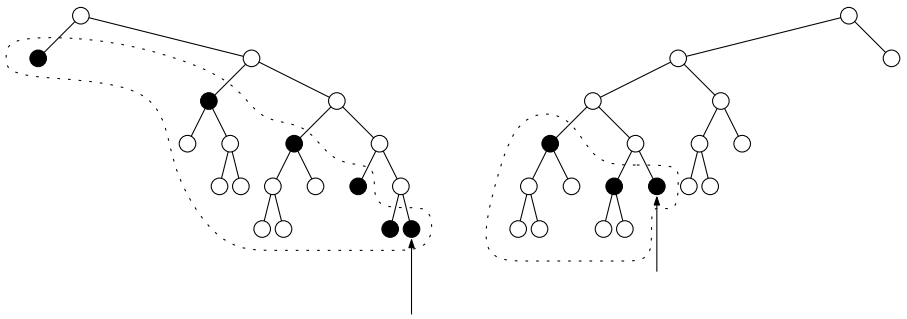

Figure 3.3: Left: Suppose the above tree is an instance of the ITKP, and the table row for the vertex marked by the arrow is currently being computed. The nodes marked in black indicate which table rows must be stored, not only for the computation of the current row, but for any future rows. In this particular case, the marked node represents the vertex having the maximum storage requirements. Right: If the tree on the left is traversed in a way such that the largest subtree is always traversed first, then the number of rows that must be stored is significantly reduced.

the post-order traversal is carried out by recursing on the left child, then the right child, and then visiting v. By examining the size of the left and right child, the maximum number of right branches can be constrained by always traversing the largest child subtree first, thus implicitly making it the left child. Figure 3.3 (right) illustrates how the traversal of a tree is transformed by this action. By doing this, the number of vertices in the right subtree is ensured to be at least halved each time a right branch is taken. Therefore, the maximum number of right branches for a tree with n vertices can be bounded using the following recursion:

$$B(0) = B(1) = 1, \tag{3.35}$$

$$B(n) \leq B(\lfloor \frac{n-1}{2} \rfloor) + 1, \;\; n > 1, \tag{3.36}$$

The general case of the recursion for $B(n)$ comes from the fact that the right child is guaranteed to have no more than half of the remaining vertices in the tree. $B(n)$ is upper bounded by $\log n + 1$, for all $n \geq 1$, which means that at any point, $O(\log n)$ rows of the dynamic programming table need to be stored. □

The above method for reducing space requirements can be implemented in terms of a stack. Each row of the table can be pushed onto the stack as it is computed, and the stack is manipulated in the following way during when visiting vertex v_i:

- If $v_i \neq \textsc{Parent}(v_{i-1})$, the row is computed for the new vertex and is then pushed on top of the stack.
- If $v_i = \textsc{Parent}(v_{i-1})$ and v_{i-1} has no left sibling—either by virtue of being a left sibling or by being an only child—this requires the stack to be popped before the new row is pushed on.
- Otherwise, the stack must be popped twice before the new row is pushed: once to get rid of the row for v_{i-1}, and again to get rid of the row for its left sibling.

Although this method can be used to compute the value of the optimal solution with no increase in running time, identifying the solution requires the above algorithm to be run multiple times.

Theorem 3.6.2 (From [17])**.** *The left-right method using the above storage reduction techniques can be used to solve the ITKP in* $O(\frac{n^2C}{\log n})$ *time and* $O(C \log n)$ *space.*

Proof. In order to recover the solution, store an the last $O(\log n)$ rows of the dynamic programming table, in addition to the $\Theta(\log n)$ rows required to compute the optimal solution value. Then, once the optimal value has been discovered, a part of the solution can be recovered the last rows. The algorithm can then be started over again, this time stopping at last vertex for which no row was stored during the previous iteration. To recover the entire solution would then take $O(n/\log n)$ iterations of an algorithm requiring $\Theta(nC)$ time, matching the bound in the theorem. □

3.7 Summary

In this chapter several algorithms have been presented for solving the ITKP, RPCP, and RPCP+. These algorithms make use of two different techniques for doing dynamic programming on trees. A comparison of the performance of these algorithms can be found in Table 3.6.

Unfortunately, even the most efficient algorithms presented in this chapter are not appropriate for solving the RPCP or RPCP+. This is because in consideration of the sheer number of prefixes in modern routing tables, quadratic space and time

	Problem	Method			
		Bottom-up		Left-right	
		Time	Space	Time	Space
By weight	ITKP	$\Theta(nC^2)$	$\Theta(nC)$	$\Theta(nC)$	$\Theta(nC)$
	RPCP	$O(\min\{nC^2, n^2\})$	$\Theta(nC)$	$\Theta(nC)$	$\Theta(nC)$
	RPCP+	$\Theta(nC^2)$	$\Theta(nC)$	–	–
By profits	ITKP	$\Theta(nQ^2)$	$\Theta(nQ)$	$\Theta(nQ)$	$\Theta(nQ)$
ε-approximation	ITKP	$\Theta(n^3/\varepsilon^2)$	$\Theta(n^2/\varepsilon)$	$\Theta(n^2/\varepsilon)$	$\Theta(n^2/\varepsilon)$

Table 3.6: A comparison of the different dynamic programming methods applied to the ITKP, RPCP, and RPCP+. Table entries that are dashed out do not mean that no bounds *can* be shown for those problems, but rather that the bounds have not been explicitly stated in this chapter. Note that any bounds for the ITKP immediately hold for the RPCP.

is prohibitively expensive. Furthermore, even if the optimality of the solution is sacrificed, the ITKP FPTAS presented in Section 3.5 requires $\Theta(n^2/\varepsilon)$ time and space. For this reason, in the next chapter a different algorithm is examined as a starting point for a more practical solution.

Chapter 4

Greedy algorithms

4.1 Overview

In this chapter, a greedy $\frac{1}{2}$-approximation algorithm for the ITKP is presented that has a running time of $O(n \log n)$, under the assumptions described at the beginning Chapter 3; i.e. that the input to the algorithms is given as a binary tree. In this chapter, the predecessor and successor notation (see Section 2.1.1) is used. These concepts are still defined in terms of the original tree. Using Figure 3.1 as an example, $\textsc{Pre}(v_{20}) = \{v_{12}, v_{15}, v_{17}, v_{18}\}$, and $\textsc{Suc}(v_{12}) = v_{20}$. Furthermore, for convenience, example problems used to demonstrate the tightness of error bounds will discard the binary tree notation.

The layout of this section is as follows. First, the $\frac{1}{2}$-approximation algorithm from [16] called the *subtree density method* is reviewed, and the solution it provides is shown to have two important properties. By exploiting these properties, an alternative algorithm is developed which computes a solution with the same error bound, but a better worst case running time. After presenting these algorithms, the remainder of the chapter discusses their application to both the RPCP and RPCP+.

4.2 The subtree density method for the ITKP

In [16], it is observed that the optimal solution to an instance of the ITKP is likely to contain subtrees which have high density. The density of a subtree rooted at vertex v is defined as:

Algorithm 1 SUBTREE-DENSITY(G, C) [Modified from [16]]

$\hat{X} \leftarrow \emptyset$
$r \leftarrow \text{ROOT}(G)$
for $i \leftarrow 1$ TO n **do**
  $w_s[v_i] \leftarrow w(S(v_i))$
  $p_s[v_i] \leftarrow p(S(v_i))$
  $\delta_s[v_i] \leftarrow$ CALC-DENSITY $[v_i]$
  $m[v_i] \leftarrow$ PROP-MAX $[v_i]$
end for
while TRUE **do**
  $d \leftarrow m[r]$
  if $\delta_s[d] = -\infty$ **then**
    RETURN $\hat{X}$
  else if $w_s[d] + w(\hat{X}) = C$ **then**
    RETURN $\hat{X} \cup S(d)$
  else if $w_s[d] + w(\hat{X}) > C$ **then**
    for $d_0 \in \text{TOP}(A(d))$ **do**
      if $w(S(d_0)) \leq C$ and $p(S(d_0)) > p(\hat{X})$ **then**
        $\hat{X} \leftarrow S(d_0)$
      end if
    end for
    RETURN $\hat{X}$
  else
    $\hat{X} \leftarrow \hat{X} \cup S(d)$
    for $d_0 \in \text{TOP}(A(d) \setminus d)$ **do**
      $w_s[d_0] \leftarrow w_s[d_0] - w_s[d]$
      $p_s[d_0] \leftarrow p_s[d_0] - p_s[d]$
      $\delta_s[d_0] \leftarrow$ CALC-DENSITY $[d_0]$
      $m[d_0] \leftarrow$ PROP-MAX $[d_0]$
    end for
  end if
end while

$$\delta(S(v)) = \frac{\sum_{v' \in S(v)} p(v')}{\sum_{v' \in S(v)} w(v')} \ . \tag{4.1}$$

The subtree density method[1] is presented here in a slightly modified form for analysis purposes. The main idea of the method is to greedily remove the most dense subtrees from the tree, and add them to the current solution. When a subtree rooted at vertex v has been removed, the density of each subtree rooted at the vertices in $A(v) \setminus v$ needs to be recomputed.

It is assumed that for each vertex $v \in V$, $w(S(v))$ and $p(S(v))$ has been computed, which can easily be done in $\Theta(n)$ time. For an instance of the ITKP having binary in-tree $G = (V, E)$ and capacity C, Algorithm 1 can be used to construct a solution $\hat{X}$ iteratively using the subtree density method. However, before examining Algorithm 1, it would be helpful to continue reading as the next few paragraphs describe the algorithm in detail.

Since $\hat{X}$ is constructed iteratively, during the course of the algorithm separate residual values $w_s[v]$, $p_s[v]$, and $\delta_s[v]$ are maintained for each $v \in V$, which represent $w(S(v) \setminus \hat{X})$, $p(S(v) \setminus \hat{X})$, and $\delta(S(v) \setminus \hat{X})$, respectively. Another array, $m[v]$, defined for each $v \in V$, keeps a pointer to the root d of the maximum density subtree such that $S(d) \subseteq S(v) \setminus \hat{X}$. Two 'helper' functions are used, which are defined as follows. The first function, Calc-Density is defined to compute $\delta_s[v]$ for an input vertex v in the following way:

$$\delta_s[v] = \begin{cases} -\infty & \text{if } w(S(v)) > C, \\ p_s[v] / w_s[v] & \text{otherwise.} \end{cases} \tag{4.2}$$

If $p_s[v]$ and $w_s[v]$ are known, Calc-Density $[v]$ can be computed in constant time for any vertex. The second function, Prop-Max is defined to receive a vertex v as a parameter, and to operate under the assumption that $m[v_l]$ and $m[v_r]$ have been computed for the left and right children of v, respectively. Prop-Max $[v]$ returns either the root of the maximum density subtree contained within $S(v) \setminus \hat{X}$, or a special "ignore" flag if $v \in \hat{X}$. This flag is used to avoid selecting v in the future, since it is already part of the solution. Ties can be broken by returning the subtree of lesser weight. If $m[v_l]$ and $m[v_r]$ are precomputed for the left and right children of v, then Prop-Max can be computed in constant time for any

[1] Called "MODALPHA" in the original publication.

vertex. To accomplish this, whenever a loop calling PROP-MAX iterates over a set of vertices, the vertices are visited in topological order. The notation TOP indicates this requirement in the algorithm.

At this point a detailed description of Algorithm 1 will be given. Values are initialized on Lines 1 to 8. The current solution $\hat{X}$ is initialized to be an empty set, and the density values $\delta_s[v]$ for the subtree of each vertex $v \in V$ are initialized using CALC-DENSITY. This function ensures that infeasible subtrees—i.e. subtrees which are too large for the capacity—will not be added to the $\hat{X}$. After the initialization step, PROP-MAX has ensured that $m[r]$, where r is the root of G, stores a pointer to the root of the maximum density subtree of G.

In the main loop (Line 10) $m[r]$ is examined to identify the vertex d that is the root of the maximum density subtree $S(d)$ in the current tree. By attempting to add $S(d)$ to the solution $\hat{X}$, the result is one of the following four cases:

1. If $\delta_s[d] = -\infty$ then $S(d)$ is infeasible for the capacity C. This implies that all feasible subtrees have been added to the $\hat{X}$ already, so $\hat{X}$ is returned (Line 12).

2. If $w(\hat{X}) + w(S(d)) = C$, then the capacity has been filled exactly, and $\hat{X} \cup S(d)$ is returned (Line 14).

3. If $w(\hat{X}) + w(S(d)) > C$, then $S(d)$ cannot fit in the remaining capacity, and the maximum profit subtree which contains $S(d)$, denoted $S(d_0)$, is located. If the profit obtained by choosing $S(d_0)$ is greater than the current solution $\hat{X}$, then $\hat{X}$ is set to $S(d_0)$. Finally, the computed solution $\hat{X}$ is returned (Lines 16-21).

4. Otherwise, if $w(\hat{X}) + w(S(d)) < C$, then $S(d)$ is added to the solution $\hat{X}$, and the densities of all feasible vertices on the path from d to the root of G are recomputed. The process then starts over again (Lines 23-29).

Theorem 4.2.1 (From [16])**.** *Algorithm 1 is a $\frac{1}{2}$-approximation algorithm for the ITKP.*

Proof. If Algorithm 1 terminates on Lines 12 or 14, then $\hat{X}$ is the optimal solution, because the entire capacity has been filled with subtrees of maximum density. Suppose instead that it terminates on Line 21. Let $\hat{X}_0$ be equal to the solution before entering the loop on Line 16, $S(d_0)$ be the maximum profit feasible subtree

identified during the loop, and Q (as before) represent the value of the optimal solution. It must be the case that $p(\hat{X}_0 \cup S(d_0)) > Q$, because $S(d_0)$ contains the most dense subtree $S(d)$ in $V \setminus \hat{X}_0$, and $w(\hat{X}_0) \cup S(d) > C$. Also, it must be the case that $p(\hat{X}_0) \leq Q$ and $p(S(d_0)) \leq Q$ since both sets are feasible solutions. Since Algorithm 1 returns a solution $\hat{X}$ with profit $\hat{Q}$, where

$$\hat{Q} = \max\left\{p(\hat{X}_0), p(S(d_0))\right\} \ , \tag{4.3}$$

it is the case that

$$\hat{Q} \leq Q \leq p(\hat{X}_0) + p(S(d_0)) \leq 2\hat{Q} \ . \tag{4.4}$$

Returning to the definition of an ε-approximation (Equation 2.3) and substituting in Equation 4.4,

$$\frac{Q - \hat{Q}}{Q} \leq \frac{1}{2} \ , \tag{4.5}$$

which completes the proof.

□

In practice it is likely a good idea to run Algorithm 1 again after it terminates, to fill any remaining unused capacity. By starting the algorithm over again, subtrees which cannot fit in the remaining capacity will be excluded during the initial pass through the tree by Calc-Density. Doing this may uncover some number of less dense but still profitable subtrees which should be added to the solution. However, running the algorithm more than once does not improve the error bound in the worst case.

Lemma 4.2.2 (From [16, 26])**.** *The error bound of Algorithm 1 can be approached.*

Proof. Consider a tree with three vertices, v_1, v_2, v_3, and a knapsack of capacity $2p'$. Set $p(v_1) = 2$, and $w(v_1) = 1$, $p(v_2) = w(v_2) = p(v_3) = w(v_3) = p'$. Since Algorithm 1 selects vertices by density, a solution of value $p' + 2$ will be found, whereas the optimal solution has value $2p'$. □

The running time of the subtree density method is given as $O(n^2)$ in [16]. But, since it is assumed here that the binary tree is constructed prior to executing

Algorithm 1, $O(n^2)$ is slightly pessimistic. This is because the running time can only be $O(n^2)$ if the height of the tree is $\Theta(n)$.

Theorem 4.2.3. *Algorithm 1 requires $O(nh)$ time and $\Theta(n)$ space, where h is the height of the binary tree.*

Proof. The space bound is immediate since only a constant amount of extra information needs to be stored per vertex. The time bound comes from the fact that the loop starting on Line 9 can run for $O(n)$ iterations, and during each iteration the inner loop on Lines 23-29 can run as many as $O(h)$ times. This is due to the fact that $|A(v)| \leq h$ for each vertex $v \in V$. □

Even though Algorithm 1 does not compute an optimal solution to the ITKP, it has the advantage of only requiring linear space. This makes the subtree density method more appealing from a practical standpoint than the dynamic programming algorithms presented in Chapter 3, which all require quadratic space.

4.3 Properties of the subtree density method

In this section two properties of the solution provided by Algorithm 1 are described. It is shown that any solution having these two properties must be unique. Establishing this fact allows for easier development of more efficient algorithms which compute the same solution. First, the following two basic lemmas are required.

Lemma 4.3.1. *Let S, R_1 and R_2 be sets of vertices having non-zero weight, and R_1 and R_2 be disjoint:*

1. *If $\delta(R_1) > \delta(S)$ and $\delta(R_2) > \delta(S)$, then $\delta(R_1 \cup R_2) > \delta(S)$.*
2. *If $\delta(R_1) > \delta(S)$ and $\delta(R_2) \geq \delta(S)$, then $\delta(R_1 \cup R_2) > \delta(S)$.*
3. *If $\delta(R_1) < \delta(S)$ and $\delta(R_2) < \delta(S)$, then $\delta(R_1 \cup R_2) < \delta(S)$.*
4. *If $\delta(R_1) < \delta(S)$ and $\delta(R_2) \leq \delta(S)$, then $\delta(R_1 \cup R_2) < \delta(S)$.*

Proof. From the definition of the density function, statement 1 implies:

$$\delta(R_1) = \frac{p(R_1)}{w(R_1)} > \delta(S) = \frac{p(S)}{w(S)} \ , \tag{4.6}$$

and

$$\delta(R_2) = \frac{p(R_2)}{w(R_2)} > \frac{p(S)}{w(S)} \ . \tag{4.7}$$

Since the sets R_1 and R_2 are disjoint, it is true that

$$\delta(R_1 \cup R_2) = \frac{p(R_1 \cup R_2)}{w(R_1 \cup R_2)} = \frac{p(R_1) + p(R_2)}{w(R_1) + w(R_2)} \ . \tag{4.8}$$

Since the weight function is defined over the non-negative integers:

$$\frac{p(R_1) + p(R_2)}{w(R_1) + w(R_2)} > \frac{\frac{p(S)}{w(S)} \left(w(R_1) + w(R_2)\right)}{w(R_1) + w(R_2)} = \delta(S) \tag{4.9}$$

It is clear at this point that the same argument can be used to prove the remaining statements.

□

Lemma 4.3.2 (From [16])**.** *Let R be a tree and $\delta(R) \geq \delta(S)$ for each subtree S of R. Let $S_1, ..., S_m$ be any possibly empty set of subtrees of R and $\bar{R} = R - \cup_{i=1}^{m} S_i$. If $\bar{R} \neq \emptyset$ then $\delta(\bar{R}) \geq \delta(R)$.*

Proof. Assume the contrary, with $\bar{R} \neq \emptyset$. If this is the case, then $\delta(R) > \delta(\bar{R})$. By Lemma 4.3.1 and the definition of R, it is implied that

$$\delta(R) \geq \delta(\cup_{i=1}^{m} S_m) \ . \tag{4.10}$$

Since

$$R = \bar{R} \cup S_1 \cup ... \cup S_m \ , \tag{4.11}$$

it is implied that

$$\delta(R) > \delta(\bar{R} \cup S_1 \cup ... \cup S_m) = \delta(R) \tag{4.12}$$

by Lemma 4.3.1. This is a contradiction, thus the lemma holds. □

Algorithm 1 repeatedly selects the most dense subtrees from G. If the algorithm is run until all feasible subtrees have been exhausted, rather than until the capacity

is filled, it would partition the tree into a *set of regions*[2]. Suppose the algorithm selects the subtrees rooted at vertices $\{v_1, \ldots, v_k\}$, where v_1 is selected first by Algorithm 1. Define the resultant set of regions as $D = \{r_1, \ldots, r_k\}$, where:

$$r_j = S(v_j) \setminus \bigcup_{i=1}^{j-1} r_i \ . \tag{4.13}$$

Notice that every feasible vertex in G is contained by a single region in D, all regions in D are disjoint, and each region is rooted at a single vertex. Region r is referred to as being *below* s if there exist vertices $v \in r$ and $u \in s$ such that $(v, u) \in E(G)$. Likewise, region r is referred to as a *descendant* of region s if the root of r is a descendant of the root of s. The following properties can be used to describe a set of regions.

Definition 4.3.3. A set of regions D has the **monotonicity property**, if for every pair of regions $r \in D$ and $s \in D$, r below s implies $\delta(s) \leq \delta(r)$.

Definition 4.3.4. For a given region r, define a **subregion** r' as a subtree of r that is closed under predecessor with respect to r, such that $r \neq r'$. D has the **region density property**, if for all $r \in D$, $\delta(r) > \delta(r')$, for any subregion r' of r.

From these definitions, the following three lemmas can be proved: the first two are straightforward based on Definitions 4.3.3 and 4.3.4, and the third is by induction.

Lemma 4.3.5. *The set of regions created by Algorithm 1 has the monotonicity property.*

Proof. Suppose there exists regions r and s such that s is below r and $\delta(r) > \delta(s)$. Let T represent the complete set of regions below r, and $t_0 \in T$ be the least dense region in T. Clearly, $\delta(t_0) \leq \delta(s)$. It is also clear that t_0 must have been selected by Algorithm 1 before r, since t_0 is below r, and after all other regions $t \in T$ s.t. $t_0 \neq t$. However, $\delta(r \cup t_0) > \delta(t_0)$ by Lemma 4.3.1, which means that Algorithm 1 would have selected $r \cup t_0$ before t_0. Since this is a contradition, the initial assumption must have been false. □

Lemma 4.3.6. *The set of regions created by Algorithm 1 has the region density property.*

[2]Technically, regions are just subtrees, though not in the sense that subtree is used throughout the rest of this thesis.

Proof. Suppose there exists a subregion r' of region r such that $\delta(r') \geq \delta(r)$. This is an immediate contradiction, as it implies that r', rather than r, should have been selected as the maximum density subtree during the execution of Algorithm 1. This is due to the fact that ties are broken by selecting the subtree of lesser weight[3]. $\square$

Lemma 4.3.7. *For a given in-tree $G = (V, E)$, the set of regions $D = \{r_1, \ldots, r_k\}$ that has both the monotonicity property and region density property is unique.*

Proof. The proof is by strong induction on the number of vertices, n, in the in-tree. Base Case: The theorem holds trivially for $n = 0$, $n = 1$. Induction: Assume the theorem holds for all trees with $0 \leq n \leq k-1$ vertices. Suppose there are two sets of regions, D and $\bar{D}$, of a tree $G = (V, E)$ with k vertices, for which the monotonicity and region density properties hold and $D \neq \bar{D}$. D and $\bar{D}$ each contain one region that contains the root of G. Call these regions r and $\bar{r}$, respectively. It must be the case that $r \neq \bar{r}$, otherwise, by the inductive hypothesis, it can be shown that $D = \bar{D}$, which is a contradiction. Without loss of generality, assume that $\bar{r}$ contains a vertex not in r. There are two cases:

i) $\bar{r}$ does not 'divide' any regions in D. Rather, $\bar{r}$ is the union of r and a set of regions $T = \{t_1, \ldots, t_j\} \subset D$. Each region T is either below some other region in T, or below r. By the monotonicity property of D, $\delta(t_i) \geq \delta(r)$ for $1 \leq i \leq j$. This implies that $\delta(\cup_{i=1}^{j} t_i) \geq \delta(r)$ by Lemma 4.3.1. Let t_m be the maximum density region in T. Because t_m is a subregion of $\bar{r}$, $\delta(\bar{r}) > \delta(t_m)$ due to the region density property of $\bar{D}$. However, since $\delta(t_m) \geq \delta(\cup_{i=1}^{j} t_i) \geq \delta(r)$, it is implied that $\delta(t_m) \geq \delta(r \cup t_i) = \delta(\bar{r})$ by Lemma 4.3.1, which is a contradiction.

ii) $\bar{r}$ divides at least one region $t \in D$ into two parts: t_u the upper part of t, and a lower forest $F = t \setminus t_u = \{f_1 \ldots f_j\}$. By the inductive hypothesis, the possibly empty set of regions below any $f_i \in F$ must be unique. Call one such set of regions Y. By the region density property of D, $\delta(t) > \delta(f_i)$ for $1 \leq i \leq j$. Since any subregion of f_i is also a subregion of t, no subregion of any $f_i \in F$ can have greater density than any region in Y. This implies that there exists at least one region $s \in \bar{D}$ which is a descendant region of $\bar{r}$, and either $s = f_i$, or s is a subregion of f_i for some $1 \leq i \leq j$. The region density

[3]If ties are broken by selecting the subtree of greater weight, then the less than operator for the monotonicity property becomes strict, and equality is allowed for the region density property. Either way all of the proofs in this section still hold.

property of D indicates that $\delta(t) > \delta(s)$, since f_i is a subregion of t. From this point, there are two possibilities which both arrive at a contradiction:

(a) If t_u is a subregion of $\bar{r}$, then $\delta(\bar{r}) > \delta(t_u)$ by the region density property of $\bar{D}$, and $\delta(t_u) \geq \delta(t)$ by Lemma 4.3.2. But if this is the case, then $\bar{D}$ does not have the monotonicity property, since $\delta(\bar{r}) > \delta(t) > \delta(s)$, and a contradiction has been reached.

(b) If t_u is not a subregion of $\bar{r}$, then find some $z_0 \in D$ which is a descendant region of t and shares a root with a subregion z of $\bar{r}$, such that $z \subseteq z_0$. Such a region must exist, and by Lemma 4.3.2 and the region density property, it is implied that $\delta(z) \geq \delta(z_0)$. Since $\delta(z_0) \geq \delta(t)$ by the monotonicity property of D, $\delta(z) \geq \delta(t) > \delta(s)$. Again, a contradiction has been reached, since this implies that $\delta(\bar{r}) > \delta(z) > \delta(s)$ which violates the monotonicity property of $\bar{D}$.

Since, in all cases, a contradiction is reached if $D \neq \bar{D}$, the statement must hold for trees of size k. □

4.4 The region-heap algorithm for the ITKP

In [4], an $O(n \log n)$ algorithm for solving min-cost flow problems with upper and lower capacities in trees is described. The algorithm uses mergable heaps to improve upon the previous worst case time bound for that problem. Here, it is shown that mergeable heaps can also be used to improve the time bound of the subtree density method for the ITKP.

Lemma 4.3.7 implies that if another algorithm can identify the set of regions which has the region density and monotonicity properties, it will guarantee the same error bound as Algorithm 1. Such an algorithm works as follows. The set of regions is computed for the tree by visiting the vertices in post-order[4]. At each vertex, $v \in G$, a data structure D is created by merging data structures D_l and D_r computed for the left and right children of v, v_l and v_r, respectively. D is created in such a way as to contain the same unique set of regions that would be created by Algorithm 1 when executed on $S(v)$. To create D for a given vertex v, execute Algorithm 2.

[4]Any topological ordering will suffice.

Algorithm 2 REGION-HEAP(v, D_l, D_r)

```
1: D ← MERGE(D_l, D_r)
2: if w(S(v)) ≤ C then
3:    R ← {v}
4:    M ← GET-MIN(D)
5:    while M ≠ ∅ and δ(M) < δ(M ∪ R) do
6:       DELETE-MIN(D)
7:       R ← R ∪ M
8:       M ← GET-MIN(D)
9:    end while
10:   INSERT(D, R)
11: end if
12: RETURN D
```

Lemma 4.4.1. *The set of regions stored in D, the data structure returned by Algorithm 2, has the monotonicity property.*

Proof. If the data structures D_l and D_r are assumed to have the monotonicity property, then D can be shown to have the monotonicity property in the following way. When the loop on Line 5 terminates, it must be the case that $\delta(R) \leq \delta(M)$ by Lemma 4.3.1. Since D at this point is a subset of the regions in D_l and D_r—because up until this point only been deleting regions from $D_l \cup D_r$ —it must be the case that the regions in D have the monotonicity property. Finally, since M is the least dense region in D before R is inserted, R cannot violate the monotonicity property for any region in D. Therefore D has the monotonicity property after R is inserted. Since D_l and D_r *do* have the monotonicity property when they contain either zero or one regions in the base case, the above argument suffices to prove the lemma. □

Lemma 4.4.2. *The set of regions stored in D, the data structure returned by Algorithm 2, has the region density property.*

Proof. As with the previous proof, it is assumed that the regions stored in D_l and D_r have the region density property. Since R is initialized to be a single vertex, it clearly has the region density property. Therefore, prior to the first iteration of the loop on Line 5, both R and M have the region density property.

Choose any subregion T of $R \cup M$. T is either M, a subregion of M, a subregion of R, or comprised of two parts $T_u \cup M$ where $T_u \subset R$. If $T = M$, T is a subregion of M, or T is a subregion of R, then the region density property holds trivially due

to the initial assumptions and the termination conditions of the loop. Otherwise, since $\delta(R) > \delta(T_u)$, it is the case that $\delta(R/T_u) > \delta(T_u)$ by Lemma 4.3.2, which implies that $\delta(R/T_u) > \delta(M)$. Then, by Lemma 4.3.1, $\delta(R/T_u) > \delta(T_u \cup M)$. This means that $\delta(R \cup M) > \delta(T_u \cup M)$ also by Lemma 4.3.1. Thus, R has the region density property after the first iteration of the loop. The same argument follows for every subsequent iteration, and therefore the region density property holds for all regions in D after R is inserted. □

A *pairing heap* is a self-adjusting data structure that is designed to provide operation time bounds comparable to a Fibonacci heap, be simple to implement, and also work well in practice [12, 31].

Lemma 4.4.3. *If Algorithm 2 is implemented using a pairing heap, then the total time required to run Algorithm 2 at each vertex in the in-tree is $O(n \log n)$.*

Proof. At each vertex, a single MERGE operation, and a single INSERT operation must be performed. Within the while loop on line 5, potentially many DELETE-MIN and GET-MIN operations are performed. However, note that each region in D is rooted at a single vertex, and once deleted, that region never is inserted into D again. Therefore, the number of DELETE-MIN and GET-MIN operations is bounded by n. It is important to mention that the union operation on line 7 can be done in constant time, since each region can be stored in the heap with a pointer to its root, total profit, and total weight. A sequence of m operations, MERGE, INSERT, DELETE-MIN and GET-MIN, each take amortized $O(\log m)$ time using a pairing-heap. Since there are at most n of each of these operations, the running time is $O(n \log n)$. □

Theorem 4.4.4. *When the in-tree is already constructed, Algorithm 2 can be used as part of a $\frac{1}{2}$-approximation algorithm for the ITKP which takes $O(n \log n)$ time.*

Proof. Follows from Theorem 4.2.1 and Lemmas 4.3.5, 4.3.6, 4.3.7, 4.4.1, 4.4.2, and 4.4.3. Once the data structure at the root of the in-tree has been created, the regions can be extracted. The set of regions can be extracted in sorted order to find the $\frac{1}{2}$-approximation solution in $O(n \log n)$ time. Alternatively, the regions can be extracted in linear time unsorted, and the solution can be determined in linear time using the standard iterative median searching technique, described in [19]. □

4.5 Discussion of the RPCP

4.5.1 Error bounds

Algorithms 1 and 2 both achieve an error bound of $\frac{1}{2}$ in the general case of the ITKP. Unfortunately, even when each vertex has unit weight, the following lemma illustrates that the error bound remains tight.

Lemma 4.5.1. *The error bound of Algorithms 1 and 2 can be approached, even with the added restriction that for each $v \in V$, $w(v) \leq 1$.*

Proof. Consider a forest with three subtrees, having roots v_1, v_2, v_3, and knapsack capacity $2p'$. Set $p(v_1) = 2$, and $p(v_2) = p(v_2) = p'$. Create $p' - 1$ descendants for both v_2 and v_3, each of which have unit weight, and zero profit. The greedy strategy will result in a solution with profit $p' + 2$, whereas the optimal solution will have profit $2p'$. □

Although Lemma 4.5.1 shows how an instance of ITKP with general weights can be reduced to the unit weight case, the reduction requires a more than linear number of additional vertices. If the size of each individual subtree in the input is constrained to be less than $\frac{C}{2}$, then the error bound does improve. However, in general, a better algorithm for the unit weight case appears to be non-trivial.

4.5.2 Time bounds

The reason the running time of Algorithm 1 is $O(nh)$ for the ITKP is due to the fact that when the knapsack capacity and vertex weights are arbitrary, it is trivial to construct instances where $O(n)$ regions must be selected to fill the capacity. However, a stronger claim can be made about the running time of Algorithm 1 when it is applied to the RPCP.

Theorem 4.5.2. *The running time of Algorithm 1 is $O(Ch + n)$ when applied to the RPCP.*

Proof. Because $1 \leq C \leq n$ for all instances of the RPCP, and dummy vertices are never leaves in the binary tree, it is guaranteed that the main loop of Algorithm 1 will terminate after at most C iterations. This is because every region selected must contain at least one unit of weight. This simple observation leads to a better

running time for Algorithm 1 of $O(Ch + n)$ time, where the additive factor of n comes from the initial preprocessing of the binary tree. □

Furthermore, a similar observation can be made about the running time of Algorithm 2 when it deals with unit weight vertices.

Theorem 4.5.3. *The running time of Algorithm 2 is $O(n \log C)$ when applied to the RPCP.*

Proof. This proof relies on constant amortized time bounds for the operations INSERT, GET-MIN, and MERGE in a pairing heap [15]. Consider the fact that DELETE-MIN is only executed in heaps that are of size at most C, since new regions are only added if they rooted at feasible vertices[5]. For a given subtree of size at most C, the cost of constructing the regions for the subtree is at most $O(C \log C)$. There can be at most $\frac{n}{C}$ such subtrees, which, after they are constructed, can be linked together in $O(n)$ time since MERGE only requires constant amortized time. Therefore the total amount of work is less than or equal to $O(n) + O(\frac{n}{C} C \log C)$, which is $O(n \log C)$. □

4.6 Discussion of the RPCP+

So far in this chapter the question of how to handle punt vertices has not been addressed. Figure 4.1 demonstrates that if punt vertices are allowed, then the greedy algorithms presented thus far do not provide any error bound.

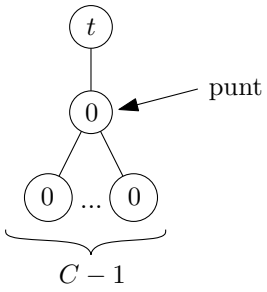

Figure 4.1: If the subtree density method were run on this example, a solution with no profit would be found, since the vertex with profit t is not feasible. However, if punt vertices are allowed, a solution with profit equal to t is clearly possible.

[5]This analysis ignores dummy vertices, since they can only increase the size of a given subtree by at most a factor of 2.

Algorithm 3 PUNT-SET-DENSITY(G, C)

$\hat{X} \leftarrow \emptyset,\ \ r \leftarrow \textsc{Root}(G)$
for $i \leftarrow 1$ TO n **do**
  $w_s[v_i] \leftarrow w(B^*(v_i))$
  $\delta_s[v_i] \leftarrow \textsc{Compute-PS-Density}(v_i)$
  $m[v_i] \leftarrow \textsc{Prop-Max-PS}[v_i]$
end for
while TRUE **do**
  $d \leftarrow m[r]$
  if $\delta_s[d] = -\infty$ **then**
    RETURN $\hat{X}$
  else if $w_s[d] + w(\hat{X}) = C$ **then**
    RETURN $\hat{X} \cup B^*(d)$
  else if $w_s[d] + p(\hat{X}) > C$ **then**
    if $p(B^*(d)) > \hat{Q}$ **then**
      $\hat{X} \leftarrow B^*(d)$
    end if
    RETURN $\hat{X}$
  else
    $\hat{X} \leftarrow \hat{X} \cup B^*(d)$
    MARK(d)
    if $d \neq r$ **then**
      $d_s \leftarrow \textsc{Suc}(d)$
      $w_s[d_s] \leftarrow w(B^*(d_s)) - w(d)$
      $\delta_s[d_s] \leftarrow \textsc{Compute-PS-Density}(d_s)$
    end if
    for $d_p \in \textsc{Pre}(d)$ **do**
      $w_s[d_p] \leftarrow w(B^*(d_p)) - w(d_p)$
      $\delta_s[d_p] \leftarrow \textsc{Compute-PS-Density}(d_p)$
    end for
    for $d_0 \in \textsc{Top}\left(\bigcup_{d_p \in \textsc{Pre}(d)} A(d_p)\right)$ **do**
      $m[d_0] \leftarrow \textsc{Propagate-Max-PS}(d_0)$
    end for
  end if
end while

In this section an algorithm is presented to deal with punt sets using the same general technique as Algorithm 1. For a given tree $G = (V, E)$ a *punt set* is defined for each non-dummy vertex $v \in V$ to be the union of v with the minimum number of punt vertices required to make v closed under predecessor with respect to G. The punt set $B^*(v)$ of vertex v is defined as:

$$B^*(v) = \{v\} \cup \bigcup_{v' \in \textsc{Pre}(v)} B(v') \ . \tag{4.14}$$

The profit $p(B^*(v))$ and weight $w(B^*(v))$ of each punt set for all non-dummy vertices in V can be computed in linear time. After this initial preprocessing, Algorithm 3 can be executed, which proceeds in a similar manner as Algorithm 1.

The main idea of the algorithm is that each punt set can be treated as an individual item to be placed in the knapsack, and the standard greedy strategy of inserting items by density can be used. When a punt set is inserted into the knapsack, its root is marked (Line 20) so that it is never considered again in the future. Each time a punt set rooted at vertex v is inserted into the knapsack, it causes other punt sets to lose weight in one of two ways. The first way is that the punt set rooted at the successor of v, v_s, loses weight equal to the weight of v, since $B(v)$ is no longer required to make $B^*(v_s)$ closed under predecessor. The second way is that each punt set rooted at a predecessor of v, v_p, loses its weight equal to $w(v_p)$, since $B(v_p)$ has already been inserted into the knapsack. This means that if $B^*(v_p)$ is subsequently inserted, there is no need to account for the weight of v_p, as it has already been accounted for when $B(v_p)$ was added.

As with the original algorithm, separate residual values, $w_s[v]$ and $\delta_s[v]$, are maintained for each non-dummy vertex $v \in V$. These values represent $w(B^*(v) \setminus \hat{X})$, and $\delta(B^*(v) \setminus \hat{X})$, respectively[6]. There is also another array, $m[v]$, which keeps a pointer to the root d of the maximum density punt set such that $B^*(d) \subseteq S(v) \setminus \hat{X}$. Algorithm 3 uses two helper functions which are defined analogously to those from Algorithm 1. The first function, Calc-Density-PS is defined to compute $\delta_s[v]$ for a punt set rooted at vertex v in the following way:

$$\delta_s[v] = \begin{cases} -\infty & \text{if } w(B^*(v)) > C, \\ p(B^*(v))/w_s[v] & \text{otherwise.} \end{cases} \tag{4.15}$$

[6]The residual profit does not need to be kept for this algorithm, since the profit of a punt set does not change during the course of the algorithm.

Since $w(B^*(v))$ is precomputed, CALC-DENSITY-PS $[v]$ can be computed in constant time for any vertex. The second function, PROP-MAX-PS is defined to receive a vertex v as a parameter, and to operate under the assumption that $m[v_l]$ and $m[v_r]$ have been computed for the left and right children of v, respectively. PROP-MAX-PS $[v]$ returns either the root of the maximum density punt set contained within $S(v) \setminus \hat{X}$, or a special "ignore" flag if no remaining punt sets are contained within $S(v)$. As before, ties can be broken by returning the punt set of lesser weight. Since $m[v_l]$ and $m[v_r]$ are assumed to be precomputed, PROP-MAX-PS can be computed in constant time for any vertex.

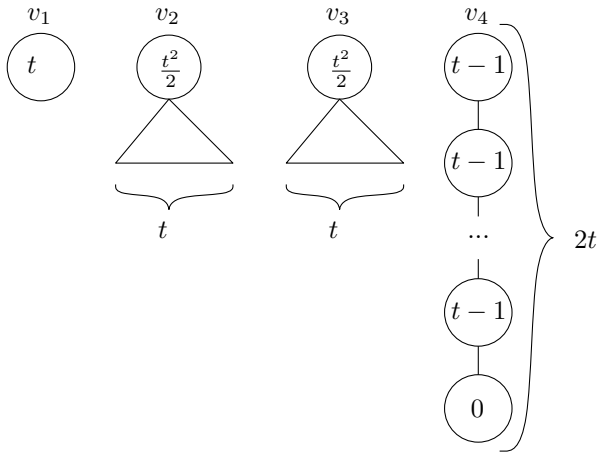

Figure 4.2: In this figure there are 3 disjoint subtrees rooted at vertices v_1, v_2, v_3, and a chain of vertices rooted at v_4. The punt sets rooted at v_2 and v_3 require t punt vertices to cover the t predecessors of these vertices which have zero profit, thus $\delta(B^*(v_2)) = \delta(B^*(v_3)) = \frac{t}{2}$. Meanwhile, $\delta(B^*(v_4)) = \frac{t-1}{2}$. The greedy strategy would select v_1 and either v_2 or v_3, then stop. The optimal algorithm would select the entire chain rooted at v_4. This is because even though each individual punt set has a low density, selecting the chain results in a better solution.

Conjecture 4.6.1. *Algorithm 3 is a $\frac{3}{4}$-approximation algorithm for RPCP+, and improves to an $\frac{1}{2}$-approximation if the algorithm is run multiple times to fill any remaining capacity.*

Although there is currently no proof for the above statement, the following lemma illustrates a potential worst case, explaining the conjecture in the process.

Lemma 4.6.2. *The error bound of $\frac{3}{4}$ can be approached for Algorithm 3.*

Proof. The problematic instance is illustrated in Figure 4.2 with capacity $C = 2t$. Algorithm 3 will find a solution with profit $t + \frac{t^2}{2}$, while the optimal solution has profit

$$(2t-1)(t-1) = 2t^2 - 3t + 1 \tag{4.16}$$

Thus the error bound approaches $\frac{3}{4}$ as $t \to \infty$. □

If Algorithm 3 is run subsequently to fill any remaining capacity, the error bound improves for this particular example. It seems that to achieve the error bound of $\frac{3}{4}$, vertices which form a chain must be present, such as the one rooted at v_4 in the example. To avoid selecting the chain, large high density punt sets such as the ones rooted at v_2 and v_3 must also be present. However, if the algorithm is run again after terminating, these large punt sets will be ignored they are too large to fit in the remaining capacity. By running the algorithm multiple times, it seems that some portion of the remaining profit in the tree will be extracted each time, improving the error bound to $\frac{1}{2}$.

Theorem 4.6.3. *Algorithm 3 requires $O(Ch+n)$ time and $\Theta(n)$ space, where h is the height of the binary tree.*

Proof. Potentially, $O(C)$ punt sets can be extracted before the algorithm terminates, since $1 \leq C < n$. This means that the for loop on Lines 30-32 can be executed $O(C)$ times. Within this loop itself, there are at most h iterations, where h is the height of the tree, plus a number of iterations which is strictly less than the number of predecessors of d. These extra iterations, along with the cost of the for loop on lines 26-29 can be charged to the $O(n)$ initialization step of the algorithm. This is because each vertex is a predecessor for at most one other vertex, and thus will only be "touched" by the algorithm in this particular manner at most one time. Therefore, the algorithm requires $O(n)$ time for the initialization step, and $O(Ch)$ time for the main loop, which achieves the desired bound. □

The running time of Algorithm 3 matches that of Algorithm 1. However, due to the nature of punt sets, it appears that the mergeable heap technique from Algorithm 2 cannot be applied to the RPCP+.

4.7 Summary

In this chapter several algorithms have been presented which make use of the subtree density method for solving the ITKP described in [16]. Table 4.1 gives a

Problem	Algorithm 1	Algorithm 2	Algorithm 3
ITKP	$O(nh)$	$O(n \log n)$	–
RPCP	$O(Ch + n)$	$O(n \log C)$	–
RPCP+	–	–	$O(Ch + n)$

Table 4.1: Comparison of the greedy algorithms presented in this Chapter for solving the ITKP, RPCP, and RPCP+. Each of these algorithms has a space requirement of $\Theta(n)$. The algorithm for the ITKP and RPCP provide an error bound of $\frac{1}{2}$.

comparison of the running times of these algorithms when applied to the ITKP, RPCP, RPCP+.

The most notable result in this section is the improved running time for the ITKP provided by Algorithm 2, which yields a solution at least $\frac{1}{2}$ as profitable as the optimal solution. It was also shown that the subtree density method could be adapted for the RPCP+, though proving that the error bounds match for the two problems remains a conjecture.

Since all of the algorithms presented in this chapter only require $\Theta(n)$ space, they provide a practical starting point for an online solution to the RPCP and RPCP+. The next step in this research is to perform an experimental comparison of these algorithms to determine their relative performance under real-world conditions. It is very likely that the worst case situations described in this chapter will not happen in practice as they were particularly contrived for the RPCP and RPCP+.

Chapter 5

Conclusions and future work

In this thesis it has been shown that the problem of maintaining routing prefix caches in high speed routers—the Routing Prefix Caching Problem (RPCP)—can be viewed as a special case of the In-tree Knapsack Problem (ITKP), where each vertex has unit weight. Furthermore, an interesting and practical variant of the RPCP, called the Routing Prefix Caching Problem with Punt Prefixes (RPCP+), has been defined as well.

The first set of algorithms presented for solving these three problems made use of dynamic programming. The most notable of these algorithms uses the left-right dynamic programming method to solve the ITKP (Section 3.3). This algorithm improves upon prior time bounds for solving the ITKP both optimally and approximately, since it can be used to derive a FPTAS as well. An immediate consequence of these dynamic programming algorithms is that only polynomial time and space are required to find optimal solutions to the RPCP and RPCP+. However, given the real-world input sizes to the RPCP and RPCP+, dynamic programming would not be practical for these problems.

Algorithms in the second set were based on a previously known greedy $\frac{1}{2}$-approximation algorithm, and sacrifice the optimality of a solution for a superior running time and linear storage requirement. Several modifications were made to this algorithm to both improve the asymptotic running time for the ITKP, as well as to adapt it to the RPCP+. In contrast to the dynamic programming algorithms, these greedy algorithms should be simple and efficient enough to be used as the basis for an online solution to the RPCP and RPCP+.

This thesis concludes with a list of open problems and suggested future work:

1. There are many practical and experimental issues left unaddressed by this thesis. A clear next step would be to gather experimental evidence to show how the various algorithms for the RPCP and RPCP+ perform using real-world Internet trace data and routing tables.

2. Can a better framework be given for the RPCP+, thus simplifying the algorithms for solving it optimally?

3. The bottom-up dynamic programming method can be modified to provide a superior running time for the RPCP. Can a similar improvement be made for the left-right method when applied to the RPCP? Specifically, is there some way of exploiting unit weight vertices to provide guarantees on the number of dominated states in the dynamic programming table? If so, the time and space bounds could be improved by doing the dynamic programming using linked lists[1].

4. There are many trade-offs still unexplored for the greedy $\frac{1}{2}$-approximation algorithms. By grouping vertices, or seeding the greedy algorithm with judiciously chosen subsets of vertices, can the an optimal solution to the RPCP be found in quadratic (or sub-quadratic) time and *linear space*? Specifically, for the case where $C = \Theta(n)$, can an ε-approximation be found for the RPCP which runs in $O(2^{1/\varepsilon} n \log n)$ time and $\Theta(n)$ space?

5. Can an algorithm identify the set of regions for an instance of the ITKP in linear time? The deletion operations in the pairing heap are what increase the running time to $O(n \log n)$. Can the techniques described in [11] be adapted for this particular problem?

6. Finally, can the conjectured error bound for the greedy algorithm for the RPCP+ be proved or disproved? Does it match the bound of the greedy algorithms for the ITKP? If not, how can the algorithm be modified to make them match?

[1] See [19] for more details on dynamic programming with linked lists.

APPENDICES

Appendix A

List of Abbreviations

KP Knapsack Problem

FPTAS Fully Polynomial Time Approximation Scheme

IP Internet Protocol

ITKP In-tree Knapsack Problem

LPMP Longest Prefix Matching Problem

NP Network Processor

OTKP Out-tree Knapsack Problem

PCKP Precedence Constraint Knapsack Problem

RPCP Routing Prefix Caching Problem

RPCP+ Routing Prefix Caching Problem with Punt Prefixes

PTAS Polynomial Time Approximation Scheme

TCAM Ternary Content Addressable Memory

Appendix B

Further discussion of the RPCP+

In this section the problem of finding a better abstraction for the RPCP+ is discussed. Recall from Chapter 4 that for a given in-tree $G = (V, E)$ a *punt set* is defined for each vertex $v \in V$ to be the union of v with the minimum number of punt vertices required to make v closed under predecessor with respect to G. A punt set $B^*(v)$ is:

$$B^*(v) = \{v\} \cup \bigcup_{v' \in \textsc{Pre}(v)} B(v') \ . \tag{B.1}$$

The problem of maximizing profit by selecting individual punt sets is odd since adding a punt set p to the solution reduces the weight of other punt sets which share vertices with p. In a more abstract sense, this problem can be generalized to the following optimization problem:

$$\begin{aligned} &\max \sum_{i=1}^{n} p_i x_i \\ &\text{s.t.} \sum_{i=1}^{n} \sum_{j=1}^{n} w_{ij} x_i x_j \leq C \\ &\quad x_i \in \{0,1\},\ \ p_i \in \mathbb{Z}^+,\ \ w_{ii} = \mathbb{Z}^+,\ \ w_{ij} \in \mathbb{Z} \ . \end{aligned} \tag{B.2}$$

In this problem each x_i represents the decision to take the i-th item, which has profit p_i and weight w_{ii}. If items i and j overlap, then w_{ij} and w_{ji} contain negative integers, so that the weight of the overlap is not counted twice. This overlap is

analogous to the way vertices can overlap between punt sets. Thus, the matrix (w_{ij}) has positive integers along its diagonal, and all other values are negative. This optimization problem is similar to both the quadratic knapsack problem [19], as well as the knapsack problem discussed in [24].

The quadratic knapsack problem can be shown to be strongly NP-hard since it generalizes CLIQUE [19]. Not surprisingly, the above optimization problem can be shown to be strongly NP-hard in the same way.

Theorem B.0.1. *The optimization problem described by Equation B.2 is strongly NP-hard since it generalizes CLIQUE.*

Proof. Input: an undirected graph $G = (V, E)$ and an integer $k > 0$, i.e. does G contain a clique of size k or greater?

This can be converted into an instance of Equation B.2 by associating an item with each vertex in G and then setting:

- $p_i = 1$ for all i, $1 \leq i \leq n$.
- $w_{ii} = t$ for some $t > k$, for all i, $1 \leq i \leq n$.
- $w_{ij} = \begin{cases} \frac{-(t-1)}{(k+1)} & \text{if } (i,j) \in E, \\ 0 & \text{otherwise.} \end{cases}$
- $C = k$.

If an algorithm can solve Equation B.2, then it can also solve CLIQUE using the above transformation. If the feasible solution to this transformed problem has profit k, the solution must be a CLIQUE of size k in the original input graph. This is because such a solution consists of k vertices and such a set of vertices must have weight:

$$tk - e\frac{(2(t-1))}{k+1} \tag{B.3}$$

where e is the number of edges between the vertices. The 2 comes from the fact that the matrix is symmetric. For this value to be equal to k demands $e = \binom{k}{2}$.

□

Further study of this problem is left as future work.

References

[1] BGP Reports, March 2009. Retrieved March 25th, 2009 from `http://bgp.potaroo.net/`.

[2] CIDR Report, March 2009. Retrieved March 25th, 2009 from `http://www.cidr-report.org/as2.0/`.

[3] M.J. Akhbarizadeh and M. Nourani. Efficient prefix cache for network processors. *Proc. of IEEE Symposium on High Performance Interconnects*, 12:41–46, 2004.

[4] W. Bein and P. Brucker. An O(nlogn)-algorithm for solving a special class of linear programs. *Computing*, 42(4):309–313, 1989.

[5] S. Chakrabarti and S. Muthukrishnan. Resource scheduling for parallel database and scientific applications. In *Proc. of ACM Symposium on Parallel Algorithms and Architectures*, pages 329–335. ACM New York, NY, USA, 1996.

[6] T.C. Chiueh and P. Pradhan. Cache memory design for network processors. *Proc. International Symposium on High-Performance Computer Architecture*, 6:409–418, 2000.

[7] G. Cho and D.X. Shaw. A Depth-First Dynamic Programming Algorithm for the Tree Knapsack Problem. *INFORMS Journal on Computing*, 9(4):431, 1997.

[8] I.L. Chvets and M.H. MacGregor. Multi-zone caches for accelerating IP routing table lookups. *Workshop on High Performance Switching and Routing*, pages 121–126, 2002.

[9] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. *Proceedings of the ACM SIGCOMM'97 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 3–14, 1997.

[10] D.C. Feldmeier. Improving gateway performance with a routing-table cache. *INFOCOM'88. Networks: Evolution or Revolution? Proceedings. Seventh Annual Joint Conference of the IEEE Computer and Communcations Societies., IEEE*, pages 298–307, 1988.

[11] G.N. Frederickson. An optimal algorithm for selection in a min-heap. *Information and Computation*, 104:197–197, 1993.

[12] M.L. Fredman, R. Sedgewick, D.D. Sleator, and R.E. Tarjan. The pairing heap: A new form of self-adjusting heap. *Algorithmica*, 1(1):111–129, 1986.

[13] M.R. Garey and D.S. Johnson. Computers and Intractability: A Guide to the Theory of NP-Completeness. 1979.

[14] P. Gupta. *Algorithms for Routing Lookups and Packet Classification.* PhD thesis, Stanford University, 2000.

[15] J. Iacono. Improved upper bounds for pairing heaps. *Lecture notes in computer science*, pages 32–45, 2000.

[16] O.H. Ibarra and C.E. Kim. Approximation algorithms for certain scheduling problems. *Math. Oper. Res.*, 3(3):197–204, 1978.

[17] D.S. Johnson and K. Niemi. On Knapsacks, partitions, and a new dynamic programming technique for trees. *Math. Oper. Res.*, 8(1):1–14, 1983.

[18] S. Kasnavi, P. Berube, V.C. Gaudet, and J.N. Amaral. A multizone pipelined cache for IP routing. *IFIP Networking Conference*, 2005.

[19] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems.* Springer, 2004.

[20] S.G. Kolliopoulos and G. Steiner. Partially ordered knapsack and applications to scheduling. *Discrete Appl. Math.*, 155(8):889–897, 2007.

[21] H. Liu. Routing prefix caching in network processor design. *Proceedings of the Tenth International Conference on Computer Communications and Networks*, 10:18–23, 2001.

[22] Y. Lu, B. Prabhakar, and F. Bonomi. ElephantTrap: A low cost device for identifying large flows. In *High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on*, pages 99–108, 2007.

[23] J.A. Lukes. Efficient Algorithm for the Partitioning of Trees. *IBM Journal of Research and Development*, 18(3):217–224, 1974.

[24] L.A. McLay and S.H. Jacobson. Integer knapsack problems with set-up weights. *Computational Optimization and Applications*, 37(1):35–47, 2007.

[25] J.J. Rooney, J.G. Delgado-Frias, and D.H. Summerville. Associative ternary cache for IP routing. *IEE Proceedings - Computers and Digital Techniques*, 151(6):409–416, 2004.

[26] S. Sahni. Approximate Algorithms for the 0/1 Knapsack Problem. *Journal of the ACM (JACM)*, 22(1):115–124, 1975.

[27] N. Samphaiboon and Y. Yamada. Heuristic and Exact Algorithms for the Precedence-Constrained Knapsack Problem. *Journal of Optimization Theory and Applications*, 105(3):659–676, 2000.

[28] D.X. Shaw and G. Cho. The critical-item, upper bounds, and a branch-and-bound algorithm for the tree knapsack problem. *Networks*, 31(4):205–216, 1998.

[29] W.L. Shyu, C.S. Wu, and T.C. Hou. Efficiency analyses on routing cache replacement algorithms. *Proceedings of IEEE International Conference on Communications*, 4, 2002.

[30] W.L. Shyu, C.S. Wu, and T.C. Hou. Multilevel aligned IP prefix caching based on singleton information. *Proceedings of the Third IEEE Global Telecommunications Conference*, 3, 2002.

[31] J.T. Stasko and J.S. Vitter. Pairing heaps: experiments and analysis. *Communications of the ACM*, 30(3):234–249, 1987.

[32] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed IP routing lookups. *ACM SIGCOMM Computer Communication Review*, 27(4):25–36, 1997.

[33] G.J. Woeginger. On the approximability of average completion time scheduling under precedence constraints. *Discrete Appl. Math.*, 131(1):237–252, 2003.