Functional Programming and Non-Distributivity in Pathfinding problems

Juan Carlos Sáenz-Carrasco

Thesis submitted to The University of Nottingham for the degree of Master of Philosophy

December 2016

Abstract

Standard approaches for finding the cost and the path of problems in graphs are based on algorithms relying the fulfilment of certain algebraic properties such associativity and distributivity, for instance the multiplication of matrices. This thesis presents an investigation on the implementation of functional paradigm approach to tackle the problem in the lack of the distributivity property for pathfinding problems in graphs.

The literature on the application, design and implementation, of this approach is scarce. The order in which a bicriteria optimization problem is selected affects the fulfilment of at least one of the properties in the calculation for pathfinding problems.

Two well-known algorithms, Dijkstra's shortest path and Floyd-Roy-Warshall all-pairs, are adapted and tuned for their application to the problems investigated here, Maximum Capacity (or Bottleneck)-Shortest path and knapsack problem.

The performance of the functional approach is assessed and results are benchmarked by two evaluations methods, eager and lazy. Also the application of the derived functions into the existing algorithms are evaluated in both pure and imperative-functional versions.

Acknowledgements

Above all, my supervisors Henrik Nilsson, Graham Hutton and my former supervisor Roland Backhouse deserve my thanks for their patience, and support throughout the last three years.

I would also like to thank everyone in the Functional Programming Laboratory who provided help and support in a wide variety of topics. These people include Thorsten Altenkirch, Venanzio Capretta, Florent Balestrieri, Paolo Capriotti, Christian Sattler, Ambrus Kaposi, Nicolai Kraus, Laurence Day, Jennifer Hackett, Bas van Gijzel, Joey Capper, Li Nuo, and Ivan Perez-Dominguez.

This thesis would be impossible to write without the support given by The Universidad Autónoma de Chihuahua and the Programa para el Desarrollo Profesional Docente (PRODEP) of the Mexican Federal Government.

Contents

Preface

This thesis is submitted in partial fulfilment of the requirements for a Master of Philosophy Degree at The University of Nottingham. It comprises a study of pathfinding problems on graphs and their solutions for the bicriteria Bottleneck-Shortest path problem and the knapsack problem in the lazy functional programming language Haskell.

All the work carried out in this document is original material except where otherwise indicated. The programs in this thesis are written in standard Haskell, and executed in The Glorious Glasgow Haskell Compilation System compiler, version 7.6.3. Knowledge of Haskell is assumed, however some of the functions are explained in detail.

Source code of the programs and functions regarding the contributions in this thesis can be found on the website https://www.github.com/jcsaenzcarrasco/MPhil-thesis/

Font convention

We used the Times font in most part of this document, whereas the *italic* style is used to denote specific terms for which we are giving an explanation. Since we are describing Haskell source code, we use the mathsf font when referring to any function or component. Also, source code is bounded by a frame with rounded corners. In the mathematical context, we use calligraphic letters as in $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$ or simply *math*.

Chapter 1

Introduction

1.1 Background and Motivation

Finding paths is a problem that has been studied for a long time and from different disciplines [BC75]. The hard task in unifying definitions and approaches has been noticed in the middle of the previous century [Gur09].

Linear algebra and graph theory are perhaps the most common language and mathematical frameworks to face the solution in pathfinding problems. More recently efforts have involved practical computation in order to solve such problems, as they grow in their input and the need to add more complexity, see [Kin96] and [Dol14].

Standard algorithms and applications for pathfinding problems assume certain algebraic properties to be hold such as associativity and distributivity. Examples of these are the Dijkstra's shortest path algorithm and Floyd-Roy-Warshall all-pairs algorithm. In both cases, the *multiplication* operator distributes over the *addition* operator and also the associativity is assumed [CLRS09].

Our motivation is precisely to find appropriate solutions for those pathfinding problems for which the distributivity property does not longer hold. At least there

are two cases in pathfinding problems for such lack of this mathematical property: the Maximum Capacity and Shortest path in this exact order (MC-SP, for short) and the knapsack problem, both defined in detail in Section 3. The operators of these pathfinding problems are similar as both are looking for the maximal cost (value, capacity, etc) as their first criterion and a minimal expense (weight, distance, etc) as their second criterion. The interesting point here is to develop the necessary operators or functions that allow these problems to be solved through the current algorithms such as that of Dijkstra and Floyd-Roy-Warshall.

1.2 Aims and Scope

It seems that a few publications are regarding to solve pathfinding problems when algorithms are implemented taking into account the lack of the distributivity property, such is the case of Gurney in [GG11] and in [Gur09], but even in this case there has not found an implementation nor in a functional context.

Roland Backhouse showed in [Bac06] that the order in which the composition of two pathfinding problems is built is critical. As an example, computing the bicriteria Min-Cost (shortest path) and Bottleneck (maximum capacity), through the well known Floyd-Roy-Warshall algorithm is perfectly possible, but if the computation is regarded to the composition of Bottleneck and then Min-Cost, then the solution provided by the same algorithm might not be the optimal. Another discussion regarding this topic is treated by Robert Manger in [Man06].

Two main objectives are pursued in this thesis in order to give solution to the problem described above, specifically for the MC-SP and knapsack problems. Answering questions such as How? and Where? these pathfinding problems present the lack of the distributivity property are discussed in Chapter 3. Initially, this investigation considers overcoming the non-distributivity for both problems and secondly

the analysis of whether or not there exists an advantage in the way the functional programming language evaluate its expressions, that is the *eager* and the *lazy* evaluation. This thesis also describes a set of test instances for both evaluations.

1.3 Overview of this thesis

The remainder of this thesis is organised as follows. In the Chapter 2 an introduction of the fundamental mathematical definitions needed within this document is presented, as well as giving necessary background for pathfinding. Different approaches and description of the most common algorithms, in the literature, to deal with pathfinding problems are also described in this chapter.

The application of Nilsson's functions **choose** and **join** to pathfinding problems, specifically those with the lack of distributivity, is treated in Chapter 3. The design, implementation and proof of correctness for those functions are detailed here. The function **choose** refers to an operator used when there is the need to select a path among a several options, and the function **join** is applied when a computation of more than one edge or path in the graph are next to each other. Their complexity are also detailed in this chapter.

Once the non-distributivity is overcome thanks to Nilsson's functions, we discuss their inclusion into different algorithms in Chapter 4, showing results of benchmarking the running of Nilsson's functions in at least three different algorithms and approaches to solve pathfinding problems, that is, single-source, all-pairs and dynamic programming.

Finally, a review of the results, the conclusions and further work are explained in 5.

1.4 Contributions of this Thesis

The contributions of this thesis are summarised as follows:

1. We showed that Nilsson's **choose** and **join** functions overcome the lack of the distributivity property in certain pathfinding problems, at least in the MC-SP and knapsack problems.

2. The MC-SP and knapsack problems were analysed in a case study fashion. We also showed that Nilsson's **choose** is practically the same in both cases.

3. Temporal complexity is calculated for the Nilsson's **choose** and **join** functions.

4. Benchmark results of the running tests are shown having the corresponding pros and cons for lazy and eager evaluations.

Chapter 2

Literature survey

2.1 Fundamental definitions

Graph theory and linear algebra are vast topics. We will mention only those mathematical entities that play a role in our research and in the perhaps more common algorithms to solve pathfiding problems.

2.1.1 Matrices

According to Carré, [Car79], and Backhouse and Carré, [BC75], it was shown that a pathfinding problem can be posed as that of solving a matrix equation. Another good reference on linear equations and matrix operations, for solving pathfinding problems, is Głazek [Gla02].

So, we will focus our data structure in a square matrix, that is, a matrix having n columns and n rows.

2.1.2 Graphs

Definition 1. A **graph** $\mathcal{G}$ is a pair $(\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ is a non-empty set of vertices or nodes, and $\mathcal{E}$ is a set of edges or arcs, where $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$.

Definition 2. A **directed** graph $\mathcal{G}$ is a pair $(\mathcal{V}, \mathcal{E})$ where the elements of $\mathcal{E}$ are ordered, that is, an edge $(x, y) \neq (y, x)$.

For the rest of this document, a **weighted** or **labelled** graph associates a label with every edge in the graph. The labels for labelled graphs in this thesis will be represented by integers unless otherwise is specified. The following is an example of a *labelled directed* graph, where the labels are *pairs* of integers. We will see why pair notation for labels are useful to explain our proposal. In this case, the values of the pairs are not regarded to the values of the vertices.

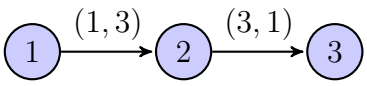


Figure 2.1: Simple Graph

For the purposes of this thesis, we use the types and functions defined in the library Data.Graph, where Graph, Edge, and Vertex are the most common types, and edges and buildG the most common functions in our Haskell code. From now on, vertices and edges are of type Int just for standard purposes. Here is a quick reference of their type signatures:

```
type Vertex  = Int
type Table a = Array Vertex a — Table indexed by a contiguous set of vertices
type Graph   = Table [Vertex] — Adjacency list representation of a graph
type Edge    = (Vertex, Vertex) — As defined
type Bounds  = (Vertex, Vertex) — The bounds of a table

edges :: Graph → [Edge]  — Given a graph, returns all edges in such graph
```

```
buildG :: Bounds → [Edge] → Graph — Build a graph from a list of edges.
                                  — Bounds scope the size of the graph
```

Definition 3. A **path** is a non-empty graph $P = (V, E)$ of the form

$$V = \{x_0, x_1, \ldots, x_k\} \qquad E = \{(x_0, x_1), (x_1, x_2), \ldots, (x_{k-1}, x_k)\},$$

where the x_i are all distinct. The vertices x_0 and x_k are linked by P and are called its *ends*. Note that k is allowed to be zero.

2.2 Pathfinding problems and optimality criteria

Definition 4. A **pathfinding problem** is the problem of finding a path between two vertices in a graph such that at least one constraint is well defined.

In many cases, the constraints are regarded to be order relations. Common operators in pathfinding problems are addition, maximum, minimum, and multiplication. Gondran and Minoux listed some of the most popular pathfinding problems in [GM08] with their definitions, models and algorithms for their solution, including the Maximum Capacity and Shortest path but not as bicriteria problem.

2.2.1 Optimality criteria

By optimality criteria in this Section we are referring to those problems having one or two operators for which a pathfinding problem picks the optimal choice, that is: a maximal or minimal element or elements after a certain number of computations.

In the following Section we will see that both Dijkstra and Floyd-Roy-Warshall algorithms define, in their original versions, a criterion for the optimal result, i.e. single criterion. This criterion is simply a comparison operator, specifically the function *minimum*, denoted also as $\downarrow$.

In other words, the shortest path solution (i.e. length, distance, time or cost) between two nodes is the application of the $\downarrow$ over all the additions between the labels of the edges in the whole graph. Another way to express this is among all the additions of the labels of all the edges in the graph, we will pick the *minimum*. Further details in the next Section.

Order is the key factor in defining an optimization problem, so problems involving sorting, minimizing or maximizing a function by choosing all the elements of the participant set or sets in the problem in matter.

The scope of our investigation is focussed specifically on the criteria for three pathfinding problems:

1. Shortest Path,
2. Maximum capacity, also called Bottleneck,
3. and knapsack

with the corresponding criteria, given a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and n number of vertices:

1. $\downarrow$ and $+$ operators to determine the path with the *minimum* length l from a source vertex s to a target vertex t, where $s, t \in \mathcal{E}$. Here, the label per edge has a simple value l (an integer in this case).
2. *max* or $\uparrow$, $\downarrow$ and $+$ operators to determine the *maximum* capacity c among all the *minimum* lengths l of all the paths between s and t. Again, $s, t \in \mathcal{E}$. Here, the label per edge comprise the pair (c, l).
3. $\uparrow$, $\downarrow$, $+$, and W (an integer representing the capacity of the knapsack) as the operators to determine the *maximum* profit p subject to $w \leq W$, where (p, w) is the label (profit,weight or volume) of each item, which is represented by a vertex in the graph.

Detailed explanation of the criteria and implementation for each pathfinding problem is given in Chapter 3.

2.2.2 Bicriteria

Pathfinding problem 1 has one criterion, the minimum of the additions, whereas pathfinding problem 2 has two criteria, either the minimum of the maximum additions or the maximum of the minimum of the additions, that is a bicriteria problem. Care should be taken at the time to select the order in which these problems are composed. We will study the latter, the $\uparrow$ of the $\downarrow$ or the $+$'s. Specific bicriteria problems are known for shortest paths, as in [Skr00].

The last pathfinding problem in the list, knapsack problem, is itself bicriteria since the operators are *maximum* and *minimum* (of the additions) with an extra predicate *at least*, (i.e. $\leq$) to compare weights against the capacity. Our research is focused in the bounded-knapsack problem.

Definition 5. Given a bag capacity $W > 0$ and $n > 0$ items, where each item has a value $v_i > 0$ and a weight $w_i > 0$, the problem is to *maximize* $\sum_{i=1}^{n} x_i v_i$, subject to $\sum_{i=1}^{n} x_i w_i \leq W$, where x_i is an element of $\{0, 1\}$, representing whether item i is selected or not in the summation. This problem is called *knapsack problem* or *0-1 knapsack problem.*

2.2.3 Pathfinding algebras

Among the algebraic-structure literature dedicated to analyse and solve pathfinding problems, two approaches arise. The one focuses on the *Regular Algebra* and the approach comprising *semirings* or $closed - semirings$ (a variety of names according to different authors). The former, aimed to model pathfinding problems based on the study of regular languages, and the latter for a variety of problems

such shortest-distance problems [Moh02], pathfinding through congruences [GG11], linear equations, regular expressions, dataflow analysis, polynomials, power series and knapsacks as in [Dol14]. For a complete list of applications in this approach, [Gla02] offers a huge compendium. Our work is based on both approaches, as we will refer to them in further sections. Although we will not use in full any of the above algebras, the following definition is useful for our purposes.

Definition 6. A **monoid** is a triple $(S, \star, e)$, where S is a set, $\star$ is a binary operation, called *product* and e is an element of S, called *unit*, satisfying the following properties:

1. $e \star x = x = x \star e$, for all $x \in S$
2. $x \star (y \star z) = (x \star y) \star z$, for all $x, y, z \in S$.

2.3 Algorithms and properties

Among all algorithms regarding the solution on pathfinding problems, we choose Dijkstra's shortest path for the *single-source* approach and Floyd-Roy-Warshall for the *all-pairs* approach. For the knapsack problem, we restrict our study to the *0-1 bounded* definition.

2.3.1 Dijkstra's single-source shortest path algorithm

Depending on how the algorithm is implemented, the most generic form of Dijkstra's algorithm has worst-case running time in $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}| \log |\mathcal{V}|)$. The reader can find a detailed information about this algorithm as well as the different forms of implementation in [CLRS09]. Let us, as a manner of reference, show a pseudocode taken from [Gur09], which is in imperative style.

1. $d(0) := 0$

2. for n in $N \setminus \{0\}$:

3. $\quad d(n) := \infty$

4. $Q := N$

5. while Q is not empty:

6. $\quad$ choose i from Q so that $d(i) \leq d(j)$ for any j in Q

7. $\quad Q := Q \setminus \{i\}$

8. $\quad$ for each j in iE:

9. $\quad\quad d(j) = \min \{d(j), d(i) + w_{(i,j)}\}$

Figure 2.2: pseudocode for Dijkstra's algorithm

where N represents the set of vertices $\mathcal{V}$, E for edges $\mathcal{E}$, and the operator '\' the $difference$ operation for sets. w is the weight of the edge (i, j) and Q is the *queue* of the remaining vertices to be computed; iE mean all edges from vertex i. So far, this algorithm computes the solution of the shortest length or cost but does not record the route or path. We will show in our code how this implementation is done.

2.3.2 Floyd-Roy-Warshall's all-pairs shortest path algorithm

Here is the pseudocode from [Gur09].

1. for each (i, j) in $N \times N$:

2. if (i, j) is in E then:

3. $d(i, j) := w(i, j)$

4. else:

5. $d(i, j) := 0$

6. for each k in N:

7. for each i in N:

8. for each j in N:

9. $d(i, j) := \min\{d(i, j), d(i, k) + d(k, j)\}$

Figure 2.3: pseudocode for the Floyd-Roy-Warshall algorithm

Where, N is our set of vertices $\mathcal{V}$, E the set of edges $\mathcal{E}$, w the weight of the edge (i, j) and once again, there is no tracking of path so far. The running time for this algorithm is $\mathcal{O}(\mathcal{V}^3)$.

2.3.3 Warshall-Floyd-Kleene's closure of a matrix algorithm

Taken from Daniel Lehmann [Leh77], the following is the pseudocode for the Floyd-Warshall-Kleene's algorithm to find the all-pairs shortest path.

$input : M = [m_{ij}]_{1 \leq i,j \leq n}$

1. for $k = 1$ to n do

2. for each $1 \leq i, j \leq n$ do

3. $[M_k]_{ij} := [M_k]_{ij} + [M_k]_{ik} \times ([M_k]_{kk})^* \times [M_k]_{kj}$

4. $R = I_n + M_n$

$output : R$

Figure 2.4: pseudocode for Warshall-Floyd-Kleene algorithm

Here, $+$ and $\times$ are the conventional arithmetic *addition* and *multiplication* operations, and * is called the *closure*, a unary operator defined as: $a^* = 1 + a \times a^* = 1 + a^* \times a$

First, the matrix is divided in four blocks, named sub-matrices A, B, C, and D, as depicted below:

$$M = \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

Then, the closure is computed as follows:

$$M = \begin{bmatrix} A^* + B' \times \Delta^* \times C' & B' \times \Delta^* \\ \Delta^* \times C' & \Delta^* \end{bmatrix}$$

where $B' = A^* \times B$, $C' = C \times A^*$ and $\Delta = D + C \times A^* \times B$. The star operator $(^*)$ stands for

The details about the theorems and proofs are out of reach of this thesis but in [Leh77]. The corresponding implementation in Haskell for this approach can be found in [Dol14].

Chapter 3

Overcoming non-distributivity

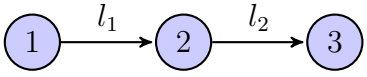
We found in the literature a common way to name the binary operators over the algebraic structures, *addition* and *multiplication*.

We will rename those operators from *addition* to *choose* and *multiplication* to *join*, mainly for the reason that in the graph fashion, is more practical for us to identify a *choice* when a vertex has more than one edge going to or coming from other vertices and a *joint* when two paths or edges are next to each other. In terms of programming language, our function join has the following type signature provided it is a binary operator:

```
join :: WeightType → WeightType → WeightType
```

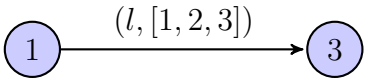
So far WeightType can be any numeric type, and in further sections we will see it becomes a pair of integers.

Another, perhaps trivial, reason is the English description for such names, graphically speaking in

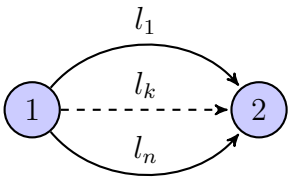


we are *joining* the edges of nodes 1 to 2 and from 2 to 3, in order to get a *cost* and a *path* from 1 to 3, that is, we are computing the labels l_1 and l_2 to get the

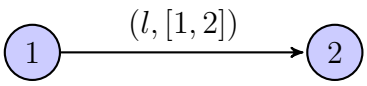
cost, say l under a specific given binary operation called *join*, and also computing the path from 1 to 3, looking like



Similarly, for the *choose* operation, it is case when a specific pathfinding problem 'needs' to make a 'decision' between two or more edges going out from a specific vertex, graphically speaking in



we are 'choosing' one edge from the set between vertices 1 and 2, and as expected, the action will result in a binary operation *choose*. Same as before, we are assuming that computing all the labels over the edges lead to a final result l, and assuming that edges (or paths) must have the same source and target vertices, then



Also, the type signature for our choose function will look as join:

```
choose :: WeightType → WeightType → WeightType
```

For the rest of this chapter we will not bear in mind recording the route or path and we will just focus our attention to the calculation of the cost. In Chapter 4 we retake the computation of the paths for each pathfinding problem.

3.1 The problem: lack of the distributivity

There is no necessity to depict a huge graph to show the absence of the distributivity property in the pathfinding realm. In the figure 3.1 we depict a small but non trivial example.

Suppose the problem is to find, from vertex 1 to vertex 4 the *maximum capacity* among all *shortest paths*, i.e. the MC-SP. Assume further that labels over the edges are natural numbers having the form $(Capacity, Length)$.

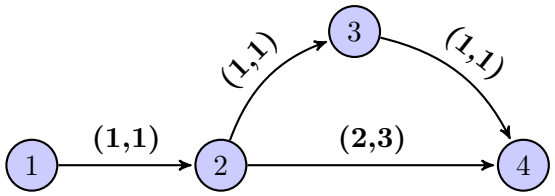


Figure 3.1: Graph with lack of distributivity for MC-SP

Notice that there are two potential paths with the same maximum capacity from vertex 1 to vertex 4 but only one (the optimal) shortest distance between the same vertices. That is why the shortest path was selected as tie breaker in this example.

Operators $\otimes$ and $\oplus$ are defined following the definition of the MC-SP problem in 2.2.1. Given two pairs of type $(Capacity, Length)$, $\downarrow$ representing the function *minimum* and $\uparrow$ representing the function *maximum*.

$$(x_1, y_1) \otimes (x_2, y_2) \quad = \quad (x_1 \downarrow x_2, y_1 + y_2)$$

and

$$(x_1, y_1) \oplus (x_2, y_2) \quad = \quad (x_1 \uparrow x_2, y)$$

where

$$y = \begin{cases} y_1 & \text{if } x_1 > x_2 \\ y_1 \downarrow y_2 & \text{if } x_1 = x_2 \\ y_2 & \text{otherwise} \end{cases}$$

Also, we are assuming that $\otimes$ has precedence over $\oplus$. Algebraically speaking, we denote the solution of the example as the definition for *lack* of distributivity

$$(1,1) \otimes [(1,1) \otimes (1,1) \oplus (2,3)] \quad \neq \quad (1,1) \otimes (1,1) \otimes (1,1) \oplus (1,1) \otimes (2,3) \qquad (3.1)$$

The right hand side computes the correct solution (i.e. cost) but the drawback here is that it takes more steps than the left hand side, in fact, this is the *all-paths enumeration* approach, which is not desirable.

Isolated from SP, the operators of the MC algebra hold the distributivity property, specifically *minimum* ($\downarrow$) does distribute over *maximum* ($\uparrow$). Given $x, y, z \in S$, where S is the carrier set for the MC algebra, we have

$$x \downarrow (y \uparrow z) = (x \downarrow y) \uparrow (x \downarrow z)$$

Now, let us say that x has the greatest value of all, then both sides simplify to $y \uparrow z$.

In the case that y has the greatest value,

$$x \downarrow y = x \uparrow (x \downarrow z)$$

$$= \{ \text{ assumption that } y \text{ is the greatest value of all } \}$$

$$x = x \uparrow (x \downarrow z)$$

$$= \{ \text{ either } z \text{ is greater or lower than } x \}$$

$$x = x$$

Finally, when z is the maximum value of the triple, we can easily see that

$$x \downarrow z = (x \downarrow y) \uparrow x$$

$$= \{ \text{ assumption that } z \text{ is the greatest value of all } \}$$

$$x = (x \downarrow y) \uparrow x$$

$$= \{ \text{ either } y \text{ is greater or lower than } x \}$$

$$x = x$$

We have just proved that MC has the distributivity property, but when it becomes the first algebra in the composition with any other algebra, the distributivity does not hold as we will see in the following Section.

3.2 Case study: Maximum capacity-Shortest path problem (MC-SP)

Initially we will derive a solution for the MC-SP pathfinding problem. Secondly, we will derive a slightly different solution in the knapsack problem.

Let us take a look to the graph example in 3.1 again, but now focusing in the 'inner' graph, that is

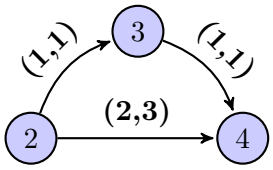


Figure 3.2: Origins of non distributivity for MC-SP

So, computing the optimal path from vertex 2 to vertex 4 in the context of MC-SP, we clearly get the path solution as

```
choose [(2,3)] ( join [(1,1)] [(1,1)] )
= { applying join, i.e. (↓,+), gives [(1,2)] as result }
choose [(2,3)] [(1,2)]
= { applying choose, i.e. (↑,↓), gives [(2,3)] }
[(2,3)]
```

pictorically a graph as

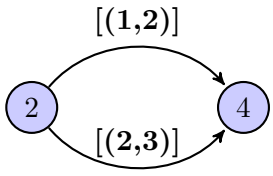


Figure 3.3: computation of *join*

and finally,

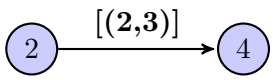


Figure 3.4: computation of *choose*

As we saw in 3.1, we are losing the pair $(1, 2)$, which is needed in order to compute the optimal solution when computing a path from vertex 1 to vertex 4 in the graph depicted in Figure 3.1.

So, the main idea is to store such pairs that are potential or interesting for the computations in advance. That is,

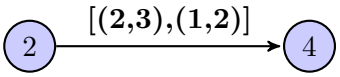


Figure 3.5: ideal computation to preserve optimal solution

It is important to highlight that the list we want should be in order, $[(2,3),(1,2)]$ and not $[(1,2),(2,3)]$, so the optimal result so far is simply the *head* of the list. We can also notice that the potential pairs are comprised by those *capacities* and *lengths* having *lower* values than their predecessors in the list, a relation satisfying

$$p_1 \succ p_2 \succ \dots p_n$$

where $\succ$ is defined pairwise as

$$(c_1, l_1) \succ (c_2, l_2) = c_1 > c_2 \wedge l_1 > l_2$$

Furthermore, to include commutativity in the definition, we can denote it as

$$(c_1, l_1) \succ (c_2, l_2) = (c_1 > c_2 \wedge l_1 > l_2) \vee (c_1 < c_2 \wedge l_1 < l_2)$$

Now, let us formalize our algebra and operators in order to offer a solution to overcome non distributivity in the MC-SP problem.

Suppose $\mathcal{S}_{\mathbf{MCSP}} = (S, +, 0)$ is a symmetric monoid and suppose further $\mathcal{R}_{\mathbf{MCSP}}$ is a relation on S, that is, $\mathcal{R}_{\mathbf{MCSP}} \subseteq S \times S$. Since $+$ in MC-SP is the conventional arithmetic addition, and the 0 is an element in $\mathbb{N}$ or in $\mathbb{R}$, depending on the labels of the graph in matter, then our monoid is symmetric.

Definition 7. We call *monotonic reduction* (*monreduc*) of $\mathcal{R}_{\mathbf{MCSP}}$ on S, for all

elements $x, y, z \in S$,

$$x \; monreduc \; y \equiv \forall z : x + z \; \mathcal{R}_{\mathbf{MCSP}} \; y + z \tag{3.2}$$

Recalling the criteria in MC-SP problem, the relation $\mathcal{R}_{\mathbf{MCSP}}$ is defined as follows:

$$(h_1, d_1) \; \mathcal{R}_{\mathbf{MCSP}} \; (h_2, d_2) \equiv h_1 > h_2 \vee (h_1 = h_2 \wedge d_1 \leq d_2) \tag{3.3}$$

In other words, we are looking for the *pair* having the *maximum* capacity and the *minimum* length or distance. The product operator is then,

$$(h_1, d_1) \star (h_2, d_2) = (h_1 \downarrow h_2, d_1 + d_2) \tag{3.4}$$

Now, we prove the **correctness** for our computations **choose** and **join**, we can also call them $+_{\mathbf{MCSP}}$ and $\star_{\mathbf{MCSP}}$ respectively.

Having a specific carrier set $S = \mathbb{R} \times (\mathbb{R} \cup \{\infty\})$ and applying the quantifiers laws as in [Bac11], we derive the following.

Let us start by applying the definition of *monreduc*. This is:

$$
\begin{array}{ll}
 & (h_1, d_1)\ \mathit{monreduc}\ (h_2, d_2) \\
= & \{\quad \text{definition 7, definition of } \mathcal{R}_{\mathbf{MCSP}}, \text{ and definition of product } \} \\
 & \langle \forall k_1, k_2 :: h_1 \downarrow k_1 > h_2 \downarrow k_1 \vee (h_1 \downarrow k_1 = h_2 \downarrow k_1 \wedge d_1 + k_2 \leq d_2 + k_2) \rangle \\
= & \{\quad \text{property of minimum} \quad \} \\
 & \langle \forall k :: (h_1 > h_2 \wedge k > h_2) \vee (h_1 \downarrow k = h_2 \downarrow k \wedge d_1 + k_2 \leq d_2 + k_2) \rangle \\
= & \{\quad \text{cancellation of addition} \quad \} \\
 & \langle \forall k :: (h_1 > h_2 \wedge k > h_2) \vee (h_1 \downarrow k = h_2 \downarrow k \wedge d_1 \leq d_2) \rangle \\
= & \{\quad \text{range splitting on } k > h_2 \text{ and } \neg(k > h_2) \quad \} \\
 & \quad \langle \forall k :: k > h_2 : h_1 > h_2 \vee (h_1 \downarrow k = h_2 \wedge d_1 \leq d_2) \rangle \\
 & \wedge \langle \forall k : k \leq h_2 : k \leq h_1 \wedge d_1 \leq d_2 \rangle \\
= & \{\quad \text{distributivity and indirect equality} \quad \} \\
 & (h_1 > h_2 \vee (h_1 = h_2 \wedge d_1 \leq d_2)) \wedge (h_2 \leq h_1 \wedge d_1 \leq d_2) \\
= & \{\quad [h_1 > h_2 \Rightarrow h_2 \leq h_1] \quad \} \\
 & h_1 \geq h_2 \wedge d_1 \leq d_2 \quad .
\end{array}
$$

That is, we have proved that for all h_1, h_2, d_1 and d_2,

$$(h_1, d_1)\ \mathit{monreduc}\ (h_2, d_2) \equiv h_1 \geq h_2 \wedge d_1 \leq d_2\ . \tag{3.5}$$

the operators for MC-SP are *correct* with respect to a monotonic relation. But this is the case multiplication *does* distribute over addition. Now we are going to prove *correctness* over keeping the 'interesting' pairs, that is when pairs are *not* related

by *monreduc*. The calculation is as follows:

$$
\begin{aligned}
& \neg((h_1, d_1)\ monreduc\ (h_2, d_2)) \wedge \neg((h_2, d_2)\ monreduc\ (h_1, d_1)) \\
= \ & \{ \quad \text{from 3.5} \quad \} \\
& \neg(h_1 \geq h_2 \wedge d_1 \leq d_2) \wedge \neg(h_2 \geq h_1 \wedge d_2 \leq d_1) \\
= \ & \{ \quad \text{boolean algebra, ordering of numbers} \quad \} \\
& (h_1 < h_2 \vee d_1 > d_2) \wedge (h_2 < h_1 \vee d_2 > d_1) \\
= \ & \{ \quad \text{boolean algebra,} \\
& \qquad [h_1 < h_2 \wedge h_2 < h_1 \equiv false] \\
& \qquad [d_1 > d_2 \wedge d_2 > d_1 \equiv false] \quad \} \\
& (h_1 < h_2 \wedge d_2 > d_1) \vee (h_2 < h_1 \wedge d_1 > d_2) \ .
\end{aligned}
$$

That is, two pairs (h_1, d_1) and (h_2, d_2) are unrelated by *monreduc* whenever

$$(\mathbf{h_1} < \mathbf{h_2} \wedge \mathbf{d_2} > \mathbf{d_1}) \vee (\mathbf{h_2} < \mathbf{h_1} \wedge \mathbf{d_1} > \mathbf{d_2}) \tag{3.6}$$

□

Having the correctness predicate for our functions-operators is not the only concern. We need to bear in mind that such operators should run in the minimum complexity as possible as we have actually $\mathcal{O}(|\mathcal{E}| + |\mathcal{V}| \log |\mathcal{V}|)$ for the single-pair problem and $\mathcal{O}(\mathcal{V}^3)$ for the all-pairs couterpart.

Initially, the lists of pairs are singletons, that is computing *choose* or *join* for the very fist time. As the graph, or the matrix, is traversed (i.e. iterating the above algorithms), the resulting lists can, in the worst case, grow as large as the amount of edge labels in the graph. So, in general we need to perform a *comparison* ($\uparrow$ or $\downarrow$) and an *addition* for each pair in a single pass in the input lists, that is, at most $M + N$, where M and N are the corresponding *lengths* of the input-lists.

Let us call the definition of the following functions **Nilsson choose** and **Nilsson join**, named in that way after Dr. Henrik Nilsson, my supervisor, who encouraged and supported me all along this project.

We want to let the reader know that, from now on, we assume that each edge label will always be defined as singleton, that is, a list with a single pair. In the case that two lists with more than one pair are given as inputs to functions **nch** and **njn**, we assume those lists comply with Definition 3.6 and are sorted in descending order.

3.2.1 *addition* operation: function Nilsson choose, (nch)

We have already defined practically all the constraints for this *addition* operator as well as its correctness. Prior to defining a function, we will describe such constraints (referred to as 'c' and a number).

c1 Given two lists of pairs, of type $(Capacity, Length)$, **nch** will store all pairs satisfying 3.6, in the context of an *addition* computation. Three cases arise. Given the pairs (c_1, l_1) and (c_2, l_2) :

1. $c_1 = c_2$, (c1.1)
2. $c_1 > c_2$, (c1.2)
3. $c_2 > c_1$ (c1.3)

Initially, comparing the heads of the lists (or singletons) is quite simple. When the pairs are part of the tails of the lists, more comparisons are needed, as carrying the length along in order to preserve 3.6.

c2 For any finite-length input lists (i.e. cls_1 and cls_2), function **nch** terminates.

c3 The length of the resulting list after computing **nch**, should be at most the sum of the lengths of the input lists.

```
length nch ≤ length cls1 + length cls2
```

Let us now define the function **nch** in Haskell, starting with the type signature and the base cases.

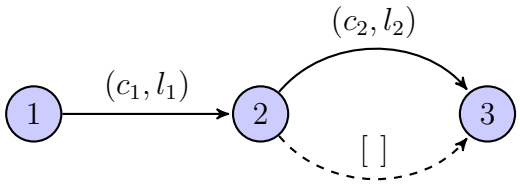
```
nch :: [(Capacity, Length)] → [(Capacity, Length)] → [(Capacity, Length)]
nch [] cls2 = cls2
nch cls1 [] = cls1
```

where Capacity, and Length are type synonyms of

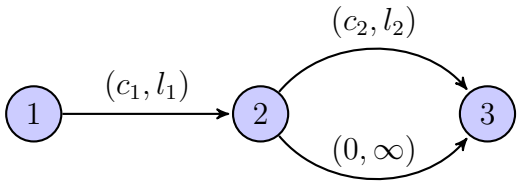
```
type Capacity = Int
type Length   = Int
```

c1 Initially, when lists are singletons, this is trivial. Otherwise, assuming cls_1 and cls_2 satisfy 3.6, then **nch** also does.

Also notice that empty list here is acting as the unit for *nch* but is not necessary the *zero*. The *zero* for MC-SP problem is in fact the pair with labels $[(0, \infty)]$. Let us see this graphically. In the following examples we are computing the solution for a path from vertex 1 to vertex 3.



Similarly,



where the result in both cases is

$$njn\ [(c_1, l_1)]\ [(c_2, l_2)]$$

since the path solution from vertex 2 to vertex 3 is

$$nch\ [(c_2, l_2)]\ [\,]\ =\ [(c_2, l_2)]\ =\ nch\ [(c_2, l_2)]\ [(0, \infty)]\ .$$

c2 Termination is trivial, in one step, either as cls_2 or as cls_1.

c3 Length of **nch** $\leq \text{cls}_1 + \text{cls}_2$ is also trivial, either the size of cls_2 or the size of cls_1.

Then, non-empty-lists case comprises the analysis of the *heads* of the lists via pattern matching and the analysis of the *tails* via recursion through two auxiliary functions, chAux and chAux′. The output from the aforementioned auxiliary functions altogether the pair resulting from the pattern matching is the output of the nch function. The inputs for the auxiliary functions vary but their output is the same, a list of type [(Capacity, Length)].

```
nch clcls1@((c1, l1) : cls1) clcls2@((c2, l2) : cls2)
    | c1 == c2  = (c1, min l1 l2) : chAux (min l1 l2) cls1   cls2
    | c1 >  c2  = (c1,      l1   ) : chAux       l1      cls1   clcls2
    | otherwise = (c2,           l2) : chAux            l2   clcls1 cls2
    where
```

```
    — code for chAux function (explained below)
```

c1 As in constraint definition c1.1 1, for the head of the list. Equation 3.6 will hold if function **chAux** also holds. Same for second and third guards as in definitions c1.2 2 and c1.3 3.

c2 Termination is done if function **chAux** terminates, see each guard in the pattern matching.

c3 **nch** is storing one pair rather than two (one from cls_1 and cls_2), the length of the remaining list will hold if **chAux** holds.

So far we have computed only the heads of the lists. In order to compute the remaining pairs of such lists, we define a recursion through two functions, **chAux** and **chAux'**.

```
1  chAux :: Length → [(Capacity,Length)] → [(Capacity,Length)] → [(Capacity,Length)]
2  chAux _ []                  []                  = []
3  chAux l []                  ((c2, l2) : cls2) = chAux' l c2 l2 [] cls2
4  chAux l ((c1, l1) : cls1) []                  = chAux' l c1 l1 cls1 []
5  chAux l clcls1@((c1, l1) : cls1) clcls2@((c2, l2) : cls2)
6    | c1 == c2  = chAux' l c1 (min l1 l2) cls1   cls2
7    | c1 >  c2  = chAux' l c1       l1    cls1   clcls2
8    | otherwise = chAux' l c2          l2  clcls1 cls2
9
10 chAux' :: Length → Capacity → Length → [(Capacity, Length)]
11           → [(Capacity, Length)] → [(Capacity, Length)]
12 chAux' l c' l' cls1 cls2
13   | l > l'    = (c', l') : chAux l' cls1 cls2
14   | otherwise =            chAux l   cls1 cls2
```

c1 Line 6 and $13-14$ guarantee c1.1 while lines $7-8$ and $13-14$ do the corresponding to c1.2 and c1.3 accordingly. That is, l is greater than any other l'

since it comes from the head (i.e. from **nch**). Otherwise, the pair in matter is not stored as is shown in line 14.

c2 Termination in line 2 is easy to see, an empty list. Termination for **chAux** in lines $3-8$ rely on **chAux'**. Since **chAux'** returns a list based on the tails of lists send by **chAux**, eventually it will end in line 2.

c3 Length of **chAux** is *zero* in line 2, while length of **chAux** in lines $3-8$ is determined in **chAux'**. Since the length of **chAux'** grows, in line 13, by one pair for every two compared in **chAux**, then total length of **chAux** is $\leq |\text{cls}_1| + |\text{cls}_2|$.

We have proved that **chAux** and **chAux'** terminate, with a length of resulting list at most the size of the sum of input lists and also holding our equation 3.6. Therefore our function **nch** also fulfils the constraints c1, c2 and c3.

3.2.2 *multiplication* operation: function Nilsson join, (njn)

Similar to **Nilsson choose**, there are some requirements for **njn** to be fulfilled. Let us recall the *multiplication* operation for the MC-SP problem:

$$(h_1, d_1) \star (h_2, d_2) = (h_1 \downarrow h_2, d_1 + d_2)$$

Since there is no 'tie' breaker in our equation as in the *addition* operation, we can then apply the operation in two comparisons rather than three, but looking after the preservation of 3.6. We avoid any repeated pair to be stored along one traversal in order to minimise the length of the resulting list. Let the following be our constraints:

c1 Given two lists of pairs, of type $(Capacity, Length)$, **njn** will store all pairs satisfying 3.6, in the context of an *multiplication* computation. Two cases

arise. Given the pairs (c_1, l_1) and (c_2, l_2) :

1. $c_1 \leq c_2$, (c1.1)
2. $c_1 > c_2$, (c1.2)

Same as *nch*, we initially compare the heads of the lists (or singletons). When the pairs are part of the tails of the lists, more comparisons are needed.

c2 For any finite-length input lists (i.e. cls_1 and cls_2), function **njn** terminates.

c3 The length of the resulting list after computing **njn**, should be at most the sum of the lengths of the input lists.

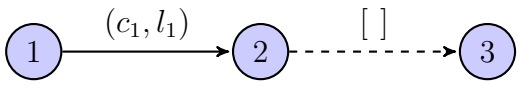
```
length njn ≤ length cls1 + length cls2
```

The base-cases and type signature for **njn** are as follows

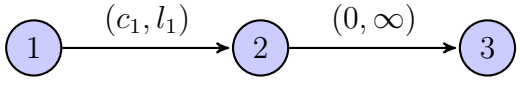
```
njn :: [(Capacity,Length)] → [(Capacity,Length)] → [(Capacity,Length)]
njn [ ] cls2 = [ ]
njn cls1 [ ] = [ ]
```

c1 This case is trivial, an empty list meets 3.6.

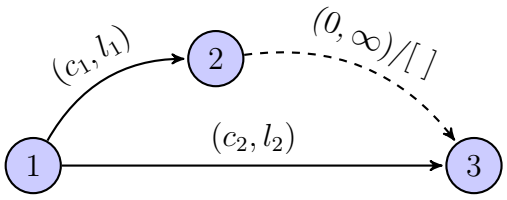
Analogously to *nch*, the empty list, [ ] acts as the *zero*, cancelling the *multiplication*, that is, resulting in computing the empty list for *njn*. Graphically,



Similarly,



where the result in both cases is no solution or solution with *zero* value. Such cases are more 'visible' in the Floyd-Roy-Warshall algorithm, where *zero* labels are placed in the adjacency matrix for every absence of an edge between two nodes. A small example for this is the following.



That is, the solution for a path from vertex 1 to vertex 3, is

$$\begin{aligned} [(c_2,l_2)] &= nch\ [(c_2,l_2)]\ (njn\ [(c_1,l_1)]\ [\,]\ ) &&= nch\ [(c_2,l_2)]\ [\,] \\ &= nch\ [(c_2,l_2)]\ (njn\ [(c_1,l_1)]\ [(0,\infty)]\ ) &&= nch\ [(c_2,l_2)]\ [(0,\infty)] \end{aligned}$$

c2 Termination is also trivial since in any case the empty list completes in just one step

c3 The length of the resulting list so far is zero.

The next definition of **njn** is the general case (non empty lists),

```
njn ((c1, l1) : cls1) ((c2, l2) : cls2)
    | c1 ≤ c2   = jnAux c1 l1 l2 cls1 cls2
    | otherwise = jnAux c2 l2 l1 cls2 cls1
    where
        — code for jnAux function (explained below)
```

c1,c2,c3 Holding 3.6, termination and length will depend on function **jnAux**, but we can see that at least the heads from the original input-lists are treated in such

a way that one of the capacities (i.e. the greatest) is not passed to **jnAux**, reducing in one-pair size the resulting list respect to the lengths of the initial lists.

The following function is defined recursively in order to compute the remaining pairs in the input lists for **njn**.

```
1 jnAux :: Capactiy → Length → Length → [(Capacity,Length)] → [(Capacity,Length)]
2 jnAux c l l' cls1 [] = (c, l + l') : [ (c1, l1 + l') | (c1, l1) ← cls1 ]
3 jnAux c l l' cls1 clcls2@((c2, l2) : cls2)
4   | c ≤ c2    = jnAux c l l2 cls1 cls2
5   | otherwise = (c, l + l') :
6     case cls1 of
7       ((c1, l1) : clss1) | c1 > c2 → jnAux c1 l1 l' clss1 clcls2
8       _                           → jnAux c2 l2 l  cls2  cls1
```

c1 In lines $4-5$ we are matching the corresponding criteria c1.1 and c1.2. Since arithmetic addition is in nature monotonic (lines 2 and 5), the only control we need to guarantee for 3.6 is the order in which capacities are stored, sorted descending, in lines $7-8$. The lengths or distances do not need to be sorted. For the list comprehension, in line 2, this is trivial, having one non empty list and a length or distance from function **njn**.

c2 Although **jnAux** always terminates in line 2 (with a list comprehension), in line 5 a recursive call computes more pairs including the tails from previous calls. Suppose we are in line 3, and suppose further we are computing a very long list clcls_2. Function **jnAux** terminates either shrinking clcls_2 in line 4 by passing its tail (cls_2) or by swapping it in line 8 with an empty list cls_1.

c3 Length of list **jnAux**, in line 2, can be misleading since $|cls_1|+|[\,]| = |cls_1|$, but the list comprehension give us $|cls_1|+1$ because a pair is adding from the left.

The empty list here does come from the original input lists but for internal computation or from **njn**. The simplest case is computing two singletons, therefore the sum of their sizes is 2. Computing **jnAux** returns a singleton, that is size 1, since the inner cls_1 is also empty. So, the remaining cases where cls_1 is a singleton and $|\text{cls}_2|$ is $n \geq 2$, for any finite $n \in \mathbb{N}$, **njn** will return a list of size $n - 1$. Recall that the order of the input lists does not matter.

The pair $(c, l + l')$, in line 5, is added to the resulting list as a consequence of comparing at least three pairs, where c represents one comparison of two pairs (from **njn** or recursively from **jnAux**) and $c2$ (line 3) represents the third pair. This means that size of resulting list will not be greater than the sum of the input lists.

We have proved that **jnAux** terminates, with a length of resulting list at most the size of the sum of input lists and also holding our equation 3.6. Therefore our function **njn** also fulfils the constraints c1, c2 and c3.

3.2.3 distributivity of njn over nch

Let us consider the following ordering to show that *multiplication*, or **njn** does distribute over *addition*, or **nch**.

$$c_1 < c_2 < c_3 \text{ and } l_1 < l_2 < l_3 \text{ in the pairs } (c_1, l_1), (c_2, l_2), (c_3, l_3)$$

where the carrier set is $\mathbb{N}$ or $\mathbb{Z}$ or $\mathbb{R}$. Then, by applying **njn** and **nch** on the left-hand side of the equation of the distributivity property, we have

$$
\begin{aligned}
& [(c_1, l_1)] \ \mathbf{njn} \ \Big([(c_2, l_2)] \ \mathbf{nch} \ [(c_3, l_3)] \Big) \\
= \ & \{ \text{ applying inner } \mathbf{nch}, \text{ and because } c_3 > c_2 \wedge l_2 < l_3 \ \} \\
& [(c_1, l_1)] \ \mathbf{njn} \ [(c_3, l_3), (c_2, l_2)] \\
= \ & \{ \text{ applying } \mathbf{njn} \text{ and since } l_1 + l_2 < l_1 + l_3 \ \} \\
& [(c_1, l_1 + l_2)]
\end{aligned}
$$

Now, on the right-hand side

$$
\begin{aligned}
& \big([(c_1, l_1)] \ \mathbf{njn} \ [(c_2, l_2)] \big) \ \mathbf{nch} \ \big([(c_1, l_1)] \ \mathbf{njn} \ [(c_3, l_3)] \big) \\
= \ & \{ \text{ applying inner } \mathbf{njn}\text{'s } \ \} \\
& [(c_1, l_1 + l_2)] \ \mathbf{nch} \ [(c_1, l_1 + l_3)] \\
= \ & \{ \text{ applying } \mathbf{nch} \text{ and } l_1 + l_2 < l_1 + l_3 \ \} \\
& [(c_1, l_1 + l_2)]
\end{aligned}
$$

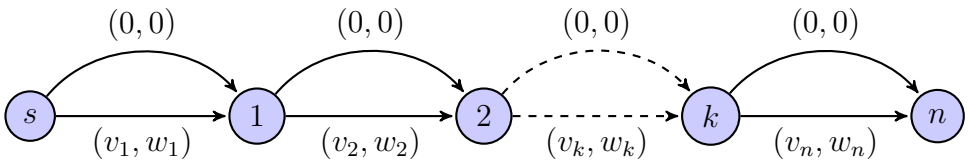
Proof of any other order can be easily showed. We can now, use **nch** and **njn** as the *addition* and *multiplication* operators respectively in the algorithms seen in 2.3. Analysis of the performance of operators **nch** and **njn** altogether with different algorithms will be seen in Chapter 4.

3.3 Case study: knapsack problem

In this section we recall definition 5 on which we will see that the previously defined function **nch** has a straight application to the knapsack problem, while the function **njn** has with some modification, being the control of bag capacity, denoted as W,

the most important.

We depict the following graph as the suitable one for defining knapsack problem.



where the pairs (v_i, w_i) clearly represents the values and weights, and the labels $(0, 0)$ represent $x_i = 0$.

As pathfinding problem, we can define the two operators as follows.

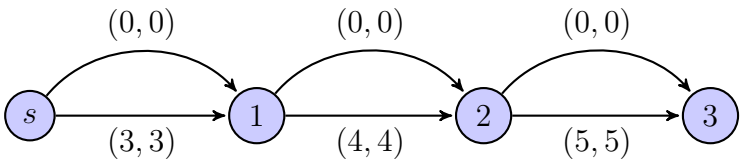
- The *addition* $(\oplus)$, $(\uparrow, \downarrow)$, as taking the maximum value from the given labels over the edges or paths, for all $w_i \leq W$, and in case of a tie, the minimum $\downarrow$ weight is choosen, as the MC-SP problem;
- The *multiplication* $(\otimes)$, $(+, +)$, as the arithmetic addition of the labels from the edges or paths given, where for all $w_i \leq W$, and $w_{i+1} \leq W$ and $w_i + w_{i+1} \leq W$.

We, of course, need to extend such definitions of the operators in order to overcome the non-distributivity, but also including a third component, the capacity W. That is, if we face that the maximum value has a weight over the capacity, we will not add it to our summation (making $x_i = 0$) since it will not fit in the bag. In this sense, we prefer to 'sacrifice' such value for a lower one that matches the capacity.

3.3.1 Non-distributivity in knapsack

Applying the operators *multiplication*, and *addition* defined in 5, to the following example we will see that the former does not distribute over the latter. Suppose the

values $(3, 4, 5)$, weights $(3, 4, 5)$ comprise the items $1, 2, 3$ respectively. The capacity W is 10. So, the following graph depicts such problem.



Each pair $(value, weight) \leq W$, so we will evaluate a comparison with W after each *product*. We have then,

$$\Big((0,0) \oplus (3,3)\Big) \otimes \Big((0,0) \oplus (4,4)\Big) \otimes \Big((0,0) \oplus (5,5)\Big)$$

applying the first two additions,

$$(3,3) \otimes (4,4) \otimes \Big((0,0) \oplus (5,5)\Big)$$

and first *multiplication*,

$$(7,7) \otimes \Big((0,0) \oplus (5,5)\Big)$$

Now, to show the lack of distributivity we present the following inequality:

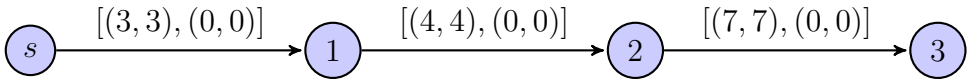
$$\begin{aligned} (7,7) \otimes \Big((0,0) \oplus (5,5)\Big) &= \Big((7,7) \otimes (0,0)\Big) \oplus \Big((7,7) \otimes (5,5)\Big) \\ (7,7) \otimes (5,5) &= (7,7) \oplus (12,12) \\ (12,12) &\neq (7,7) \end{aligned}$$

The distributivity property does not hold in previous example as neither the left hand side complies with the capacity W.

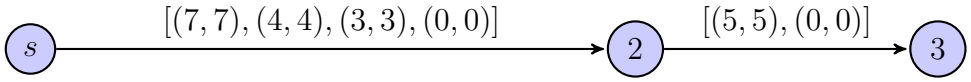
The proposed solution is similar to that in MC-SP problem, that is, storing the

interesting pairs satisfying the criteria in the Definition 5.

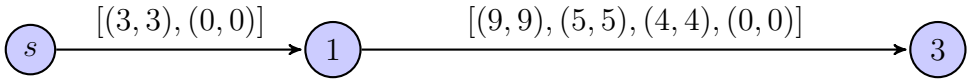
Let us depict a graph with our earlier proposal before to define formally our functions. So, after the first *additions*:



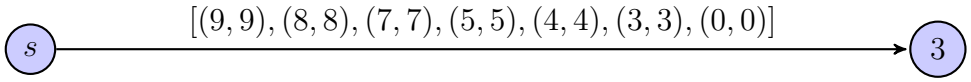
Now, applying *multiplications*, from the left:



or from the right:



and finally



Let us now formalize our proposal. Some definitions are given and equational reasoning afterwards.

Suppose $\mathcal{S}_K = (S, \star, 0)$ is a symmetric monoid and suppose further that $\mathcal{R}_K$ is a binary relation on S, that is, $\mathcal{R}_K \subseteq S \times S$.

The *multiplication* operator is defined as:

$$(v, w) \star (v', w') = (v + v', w + w') \tag{3.7}$$

So, defining the relation $\mathcal{R}_K$ for values: $v, v' \in S$, weights: $w, w' \in S$, and capacity W we have

$$\begin{aligned}(v, w)\ \mathcal{R}_K\ (v', w') = \ & (w \leq W \wedge (v \geq v' \vee w' > W))\ \vee \\ & (w > W \wedge v \geq v' \wedge w' > W)\end{aligned} \tag{3.8}$$

That is, we are looking for the pairs having a maximum value, where the weight should be at most the capacity or over, in the case two values are summed up in the multiplication operation.

Definition 8. We call $mon.\mathcal{R}_K$ of $\mathcal{R}_K$ on S, monotonic reduction to the following. For all elements $x, y, z \in S$,

$$x\ mon.\mathcal{R}_K\ y \equiv \forall z, x \star z\ \mathcal{R}_K\ y \star z \tag{3.9}$$

Now, let us prove the correctness of our operators for the knapsack problem. Having a specific carrier set $S = \mathbb{N} \times \mathbb{N}$, we derive the following. Let us start by

applying the definition of $mon.\mathcal{R}_K$. This is, $v, v', w, w' \in S$

$$
\begin{array}{ll}
 & \langle \forall x, y :: (v + x, w + y) \ \mathcal{R}_K \ (v' + x, w' + y) \rangle \\
= & \quad \{ \text{ definition of } \mathcal{R}_K \ \} \\
 & \langle \forall x, y :: (w + y \leq W \ \ \wedge \ \ (v + x \geq v' + x \ \ \vee \ \ w' + y > W)) \ \vee \\
 & \qquad\qquad (v + x \geq v' + x \ \ \wedge \ \ w + y > W \ \ \wedge \ \ w' + y > W) \rangle \\
= & \quad \{ \text{ cancellation of addition on } v \ \} \\
 & \langle \forall y :: (w + y \leq W \ \ \wedge \ \ (v \geq v' \ \ \vee \ \ w' + y > W)) \ \vee \\
 & \qquad\qquad (v \geq v' \ \ \wedge \ \ w + y > W \ \ \wedge \ \ w' + y > W) \rangle \\
= & \quad \{ \text{ distribution } \wedge \text{ over } \vee \text{ in first disjunct } \} \\
 & \langle \forall y :: ((w + y \leq W \ \ \wedge \ \ v \geq v') \ \vee \ (w + y \leq W \ \ \wedge \ \ w' + y > W)) \ \vee \\
 & \qquad\qquad (v \geq v' \ \ \wedge \ \ w + y > W \ \ \wedge \ \ w' + y > W) \rangle \\
= & \ \{ \text{ case analysis on } w + y \leq W \ \}
\end{array}
$$

$$
\begin{aligned}
&\langle \forall y : w + y \leq W : v \geq v' \ \vee \ w' + y > W \rangle \ \wedge \\
&\qquad \langle \forall y : w + y > W : v \geq v' \ \wedge \ w' + y > W \rangle \\
= \ &\{ \text{ realising free variables } \} \\
&(v \geq v' \ \vee \ \langle \forall y : w + y \leq W : w' + y > W \rangle) \ \wedge \\
&\qquad (v \geq v' \ \wedge \ \langle \forall y : w + y > W : w' + y > W \rangle) \\
= \ &\{ \text{ quantifier in the first conjunct is eliminated since its term} \\
&\text{is out of the scope of the rage } \} \\
&(v \geq v') \wedge (v \geq v' \ \wedge \ \langle \forall y : w + y > W : w' + y > W \rangle) \\
= \ &\{ \text{ calculus on predicates and quantifiers } \} \\
&(v \geq v') \ \wedge \ (v \geq v' \ \wedge \ w' \geq w) \\
= \ &\{ \text{ property of conjunction } \} \\
&v \geq v' \ \wedge \ w' \geq w
\end{aligned}
$$

That is, we have proved that for all v, v', w, w' and capacity W,

$$
(v, w) \ \ mon.\mathcal{R}_K \ \ (v', w') \equiv v \geq v' \ \wedge \ w' \geq w \tag{3.10}
$$

Similar to the MC-SP problem, we would like to keep track of those cases due to the lack of distributivity.

$$
\begin{aligned}
&\neg((v, w) \ monreduc \ (v', w')) \wedge \neg((v', w') \ mon.\mathcal{R}_K \ (v, w)) \\
= \ &\{ \text{ from 3.10 } \} \\
&\neg(v \geq v' \wedge w \leq w') \wedge \neg(v' \geq v \wedge w' \leq w) \\
= \ &\{ \text{ boolean algebra, ordering of numbers } \}
\end{aligned}
$$

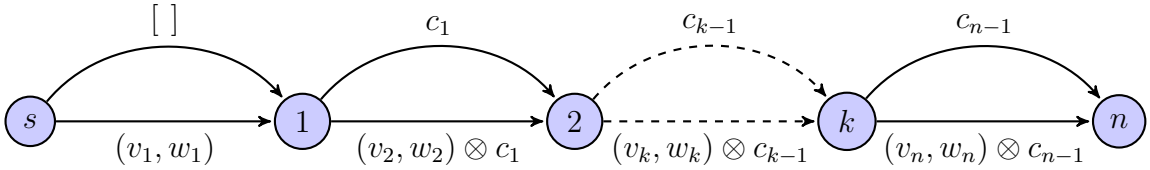
$$
\begin{aligned}
& (v < v' \vee w > w') \wedge (v' < v \vee w' > w) \\
= \quad & \{ \text{ boolean algebra,} \\
& \qquad [v < v' \wedge v' < v \equiv false] \\
& \qquad [w > w' \wedge w' > w \equiv false] \ \} \\
& (v < v' \wedge w' > w) \vee (v' < v \wedge w > w') \quad .
\end{aligned}
$$

That is, two pairs (v, w) and (v', w') are unrelated by $mon.\mathcal{R}_K$ whenever

$$(\mathbf{v} < \mathbf{v}' \wedge \mathbf{w}' > \mathbf{w}) \vee (\mathbf{v}' < \mathbf{v} \wedge \mathbf{w} > \mathbf{w}') \tag{3.11}$$

□

The shape of the graph plays a big role in the knapsack problem, since it will not change because the number of items or type of *values* or *weights* (i.e. integers, real numbers, etc.). The main idea is to seize the opportunity of the shape and from there define our operators *addition* and *multiplication*. We have seen that [ ] behaves as the *zero* for *addition* in our functional programming implementation in the MC-SP problem. We can also observe that the general computation is a big *multiplication* of *choices* (or *additions*). Except for the first calculation (i.e. from vertex s to vertex 1), the remaining computations can be based on previous work, as in dynamic programming. So, our graph turns to be as follows.



In this context, we do not need for the *addition* operation to add a comparison

against W, except on the first evaluation. We will leave this comparison just to the *multiplication* operator.

- The base case, computing the optimal result from s to 1 is $[\,] \oplus (v_1, w_1)$ where $w_1 \leq W$ otherwise we have $[\,]$. We call this computation c_1.
- Next, the *addition* of c_1 and $(v_2, w_2) \otimes c_1$ relies in that *multiplication* verifies both $w_2 \leq W$, and $w_2 + w_1 \leq W$. The result from this *multiplication* is c_1 or (v_2, w_2) or $(v_2, w_2) \otimes c_1$. Now, the addition does not need to compare any weight against W. We call this computation c_2.
- For the i-th step, we *multiply* the previous result, c_{i-1}, with (v_i, w_i) and *add* to the previous result.

3.3.2 *addition* operation: function Nilsson choose, (nchk)

We have seen that nch from the MC-SP problem and nchk are practically the same in structure, that is, the criteria and the binary relation. We now define the Haskell code, starting with the types of values and weights.

```
type Value  = [Int]
type Weight = [Int]
```

Note that the type for the item weight is the same that for the capacity.

```
1 nchk :: [(Value,Weight)] → [(Value,Weight)] → [(Value,Weight)]
2 nchk [] vws2 = vws2
3 nchk vws1 [] = vws1
4 nchk vwvws1@((v1, w1) : vws1) vwvws2@((v2, w2) : vws2)
5     | v1 == v2  = (v1, min w1 w2) : chAux (min w1 w2) vws1    vws2
6     | v1 >  v2  = (v1,      w1    ) : chAux       w1      vws1    vwvws2
7     | otherwise = (v2,          w2) : chAux           w2  vwvws1 vws2
8     where
```

```
9   chAux :: Length → [(Value,Weight)] → [(Value,Weight)]
10                  → [(Value,Weight)]
11  chAux _ []          []              = []
12  chAux w []          ((v2, w2) : vws2) = chAux' w v2 w2 [] vws2
13  chAux w ((v1, w1) : vws1) []          = chAux' w v1 w1 vws1 []
14  chAux w vwvws1@((v1, w1) : vws1) vwvws2@((v2, w2) : vws2)
15    | v1 == v2  = chAux' w v1 (min v1 w2) vws1   vws2
16    | v1 >  v2  = chAux' w v1      v1     vws1   vwvws2
17    | otherwise = chAux' w v2         w2  vwvws1 vws2
18
19  chAux' :: Weigth → Value → Weight → [(Value,Weight)]
20                   → [(Value,Weight)] → [(Value,Weight)]
21  chAux' w v' w' vws1 vws2
22    | w > w'    = (v', w') : chAux w' vws1 vws2
23    | otherwise =            chAux w  vws1 vws2
```

Since we are encoding the same function nch from MC-SP, the constraints are fulfilled in the same way as that function. As a summary we have

1. Function nchk meets 3.11.
2. Function nchk terminates.
3. Length of the function nchk is at most the sum of the two input lists.

3.3.3 *multiplication* operation: function Nilsson join, (njnk)

The constraints for the product operator in the knapsack problem are practically the same as those in nchk, but the advantage that its length is a bit shorter, at most the size of the input list plus one. Let us recall the product definition:

$$(v_1, w_1) \otimes (v_2, w_2) = (v_1 + v_2, w_1 + w_2)$$
$$\text{where } w_1 \leq W \wedge w_2 \leq W \wedge w_1 + w_2 \leq W$$

c1 Given one pair and a list of pairs, of type $(Value, Weight)$, **njnk** will store all pairs satisfying 3.11, in the context of a *multiplication* computation. Two cases arise. Given the pairs (v_1, w_1) and (v_2, w_2) :

1. $w_1 \leq W$
2. $w_1 + w_2 \leq W$

c2 For any finite-length input list (i.e. vwss), function **njnk** terminates.

c3 The length of the resulting list after computing **njnk**, should be at most the length of the input list plus one.

```
length njnk ≤ length vwss + 1
```

The base case and function signature:

```
njnk :: Weight → (Value,Weight) → [(Value,Weight)] → [(Value,Weight)]
njnk wc (v,w) []   = if   w > wc
                     then []
                     else [(v,w)]
njnk wc (v,w) vwss = if   w > wc
                     then vwss
                     else njnAk wc (v,w) vwss
```

c1 In the second line, the case is trivial. We need only to verify that our unique pair does not exceed the capacity in order to satisfy 3.11. For the non empty input list, we simply determine whether the weight in the input pair is at most

the capacity. In case the latter is true we call the function njnAk, otherwise the input list, vwss, is returned. At this point we assume either vwss and result from function njnAk satisfies 3.11.

c2 njnk terminates either in second row with a simple comparison or by calling function njnAk (assuming njnAk terminates).

c3 The length of njnk is 1 or zero in second row. For the third row, this function depends on the result of function njnAk.

For the consecutive pairs in the input list, the following recursive function is defined.

```
1 njnAk :: Weight → (Value,Weight) → [(Value,Weight)] → [(Value,Weight)]
2 njnAk wc (v,w) []          = [(v,w)]
3 njnAk wc (v,w) vwss@(vw:vws)
4     | w + (snd vw) > wc =                    njnAk wc (v,w) vws
5     | otherwise          = (v+fst vw, w+snd vw) : njnAk wc (v,w) vws
```

c1 Case in line 2 is trivial. In line 4, 3.11 is guaranteed since the pair added to the list fulfils the comparison in definition of constraint c1 and because the additions are monotonic. The case in line 5 no pair is added as it does not satisfy the criteria.

c2 Function njnAk terminates in line 2, adding always the pair of the current item in analysis, since its weight was proved to be at most the capacity, wc. Lines 4 and 5 terminate as they recursively call themselves with the tail of the input list, vws.

c3 If the weight of all items (in the input list) are lower than the capacity and their sum with current item weight (the input pair) are also lower than the

capacity (worst case for the length), then line 4 is always called, which turns to be $|vwss|$, i.e. the size of input list. Then we add a pair in line 2, giving us $|vwss| + 1$. For all other cases, in line 5, are taken into account meaning $njnAk \leq |vwss| + 1$.

Since the function njnAk fulfils constraints c1,c2, and c3, the function njnk does as well.

Care should be taken at the time to compute **njnk** consecutively, since it associates only from the right. Recall the specific shape of the graph for the knapsack problem. So, **njnk** will type check only

```
njnk wc (v1 ,w1) ( njnk wc (v2 ,w2) ... (njnk wc (vn ,wn) vws ) ... )
```

where vws is a list of pairs (v, w), and wc is the capacity W.

3.3.4 Distributivity of $njnk$ over $nchk$

Let us consider the following ordering to show that *multiplication*, or **njnk** does distribute over *addition*, or **nchk**.

$$w_1 < w_2 < w_3 \text{ and } v_1 < v_2 < v_3 \text{ in the pairs } (v_1, w_1), (v_2, w_2), (v_3, w_3)$$

Also

$$w_1 + w_2 < w_1 + w_3 < w_2 + w_3 \leq W < w_1 + w_2 + w_3$$

where $(v_i, w_i) \in \mathbb{N}$ for $i \in \{1, 2, 3\}$. Then, by applying **njnk** and **nchk** on the left-hand side of the equation of the distributivity property, we have

$$
\begin{aligned}
& \textbf{njnk}\ W\ (v_1, w_1)\Big([(c_2, l_2)]\ \textbf{nchk}\ [(c_3, l_3)]\Big) \\
= \ & \{\ \text{applying inner } \textbf{nchk}\text{, and } v_3 > v_2 \wedge w_2 < w_3\ \} \\
& \textbf{njnk}\ W\ (v_1, w_1)\ [(v_3, w_3), (v_2, w_2)] \\
= \ & \{\ \text{applying } \textbf{njnk} \text{ and since } w_1 + w_3 \leq W \wedge w_1 + w_2 \leq W\ \} \\
& [(v_1 + v_3, w_1 + w_3), (v_1 + v_2, w_1 + w_2), (v_1, w_1)]
\end{aligned}
$$

Now, on the right-hand side

$$
\begin{aligned}
& \Big(\ \textbf{njnk}\ W\ (v_1, w_1)\ [(v_2, w_2)]\Big)\ \textbf{nchk}\ \Big(\ \textbf{njnk}\ W\ (v_1, w_1)\ [(v_3, w_3)]\Big) \\
= \ & \{\ \text{applying inner } \textbf{njnk}\text{'s, } w_1 + w_2 \leq W \wedge w_1 + w_3 \leq W\ \} \\
& [(v_1 + v_2, w_1 + w_2), (v_1, w_1)]\ \textbf{nchk}\ [(v_1 + v_3, w_1 + w_3), (v_1, w_1)] \\
= \ & \{\ \text{applying } \textbf{nchk}\ \} \\
& [(v_1 + v_3, w_1 + w_3), (v_1 + v_2, w_1 + w_2), (v_1, w_1)]
\end{aligned}
$$

Proving then that **njnk** does distribute over **nchk** (i.e. multiplication over addition) and therefore they can be applied to the corresponding algorithms to compute an optimal solution for the knapsack problem. Same as before, the benchmarking will be analysed in Chapter 4.

3.4 Complexity

In this thesis we will assume eager, not lazy, evaluation as the reduction strategy. The reasons for assuming eager evaluation as well as for focusing on time complexity are well explained in [Bir15].

We have defined four functions in this thesis in order to overcome non distributivity of a *multiplication* operator over an *addition* operator in two pathfinding problems, MC-SP and knapsack. Two of these functions are equal in their definitions, reducing the complexity calculation on three main functions: **nch** (same as **nchk**), **njn**, and **njnk** plus three complementary (i.e. path-tracking) functions: **nchP** (same as **nchkP**), **njnP**, and **njnkP**.

3.4.1 Strategy

We follow the strategy given by Stannett in [Sta13]. For any function f defined in a Haskell program, its step-counting function is called T_f. In order to compute T_f we need to know how much it costs to evaluate the various types of expression we encounter during execution of the program. We write $T(e)$ for the cost of evaluating the expression e.

cost:case if $f\ a_1\ a_2\ \ldots\ a_n$ is defined directly by an expression of the form $f\ a_1\ a_2\ \ldots\ a_n = e$ then evaluating f requires us first to perform pattern matching to identify the relevant expression as e (assume this takes one step), and then to evaluate it (which takes $T(e)$ steps). So define

$$T_f\ a_1\ a_2\ \ldots\ a_n = 1 + T(e)$$

cost:const if c is a constant, it costs nothing to evaluate c, i.e.,

$$T(c) = 0$$

cost:var if v is a variable, it costs nothing to look up the value of v, i.e.,

$$T(v) = 0$$

cost:if if e is an expression of the form *if* a *then* b *else* c, then the cost depends on the value of a. In either case we have to evaluate a first (this require $T(a)$ steps); if a is *True*, we need to evaluate b (using $T(b)$ steps) and otherwise c (using $T(c)$ steps). So we define

$$T(\text{if } a \text{ then } b \text{ else } c) = T(a) + (\text{if } a \text{ then } T(b) \text{ else } T(c))$$

cost:prim if p is a primitive operation, like addition, *cons*, $\uparrow$, $\downarrow$, which is implemented as part of the language, we can assume that the implementation is efficient enough that the cost of calling p can be ignored. Consequently, the cost of evaluating $p\ a_1 \ldots a_n$ is just the cost of evaluating the n different arguments. So

$$p\ a_1 \ldots a_n = T(a_1) + \ldots + T(a_n)$$

cost:func if f isn't a primitive function, and the particular evaluation $f\ a_1 a_2 \ldots a_n$ isn't one of the cases used to define f, the cost of evaluating $f\ a_1 a_2 \ldots a_n$ is the cost of evaluating the various arguments, together with the cost of applying f to those results, i.e.,

$$T(f\ a_1\ a_2 \ldots a_n) = T(a_1) + \ldots + T(a_n) + (T_f\ a_1 \ldots a_n)$$

Example: append function

Define

```
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

Then

$$
\begin{aligned}
T_{++}\ [\,]\ \text{ys} &= 1 + T(\text{ys}) && \text{by [cost:case]}\\
&= 1 + 0 && \text{by [cost:var]}\\
&= 1 && \text{by arithmetic}
\end{aligned}
$$

$$
\begin{aligned}
T_{++}\ (\text{x:xs})\ \text{ys} &= 1 + T(\text{x} : (\text{xs} +\!+ \text{ys})) && \text{by [cost:case]}\\
&= 1 + T(\text{x}) + T(\text{xs}+\!+\text{ys}) && \text{by [cost:prim]}\\
&= 1 + 0 + T(\text{xs}+\!+\text{ys}) && \text{by [cost:var]}\\
&= 1 + 0 + T(\text{xs}) + T(\text{ys}) + T_{++}\ \text{xs ys} && \text{by [cost:func]}\\
&= 1 + 0 + 0 + 0 + T_{++}\ \text{xs ys} && \text{by [cost:var]}\\
&= 1 + T_{++}\ \text{xs ys} && \text{by arithmetic}
\end{aligned}
$$

Clearly, the value of ys is irrelevant in this calculation, and we have

$$T_{++}\ \text{xs ys} = 1 + \text{length (xs)} \tag{3.12}$$

3.4.2 Complexity for nch and nchk functions

We start with the step-counting and cost functions (i.e. T_f and $T()$) of the inner most auxiliary function for nch.

```
1  chAux' l c' l' cls1 cls2
2      | l > l' = (c', l') : chAux l' cls1 cls2
3      | otherwise =        : chAux l  cls1 cls2
```

$T_{chAux'}$ l c' l' cls1 cls2

$$
\begin{aligned}
l > l' \text{ guard} \quad &= 1 + T(\text{(c',l'):chAux l' cls1 cls2}) && \text{by [cost:case]} \\
&= 1 + T(\text{(c',l')}) + T(\text{chAux l' cls1 cls2}) && \text{by [cost:prim]} \\
&= 1 + 0 + T(\text{l'}) + T(\text{cls1}) + T(\text{cls2}) + T_{chAux}\ \text{l' cls1 cls2} && \text{by [cost:func]} \\
&= 1 + 0 + 0 + 0 + 0 + T_{chAux}\ \text{l' cls1 cls2} && \text{by [cost:var]} \\
&= 1 + T_{chAux}\ \text{l' cls1 cls2} && \text{by arithmetic}
\end{aligned}
$$

$$
\begin{aligned}
\text{otherwise guard} \quad &= 1 + T(\text{chAux l' cls1 cls2}) && \text{by [cost:case]} \\
&= 1 + 0 + T(\text{l'}) + T(\text{cls1}) + T(\text{cls2}) + T_{chAux}\ \text{l' cls1 cls2} && \text{by [cost:func]} \\
&= 1 + 0 + 0 + 0 + 0 + T_{chAux}\ \text{l' cls1 cls2} && \text{by [cost:var]} \\
&= 1 + T_{chAux}\ \text{l' cls1 cls2} && \text{by arithmetic}
\end{aligned}
$$

From both guards, we have that function chAux′ depends on the length of both lists. Let n be the length of cls1, and let m be the length of cls2, then $T_{chAux'} = 1 + length(m + n)$.

Now, it is the case of function chAux.

```
1 chAux _ [] []    = []
2 chAux l [] ((c2,l2):cls2) = chAux' l c2 l2 [] cls2
3 chAux l ((c1,l1):cls1) [] = chAux' l c1 l1 cls1 []
4 chAux l clcls1@((c1,l1):cls1) clcls2@((c2,l2):cls2)
5     | c1 == c2  = chAux' l c1 (min l1 l2) cls1 cls2
6     | c1 > c2   = chAux' l c1 l1 cls1 clcls2
7     | otherwise = chAux' l c2 l2 clcls1 cls2
```

$$
\begin{aligned}
T_{chAux\ \_\ [\,]\ [\,]} \quad &= 1 + T([\,]) && \text{by [cost:case]} \\
&= 1 + 0 && \text{by [cost:const]} \\
&= 1 && \text{by arithmetic}
\end{aligned}
$$

Lines 2 through 7 having calls to function chAux′, so generalising

$$
\begin{aligned}
T_{chAux}\ \text{l (x:xs) (y:ys)} \quad &= 1 + T(chAux'\ \text{l x y xs ys}) && \text{by [cost:case]} \\
&= 1 + 0s + T_{chAux'}\ \text{l x y xs ys} && \text{by [cost:vars,func]} \\
&= 1 + T_{chAux'}\ \text{l x y xs ys} && \text{by arithmetic}
\end{aligned}
$$

reducing function chAux to $T_{chAux} = 1 + length(n + m)$.

Finally,

```
1  nch []  cls2 = cls2
2  nch cls1 []  = cls1
3  nch clcls1@((c1, l1) : cls1) clcls2@((c2, l2) : cls2)
4      | c1 == c2  = (c1, min l1 l2) : chAux (min l1 l2) cls1   cls2
5      | c1 >  c2  = (c1,      l1    ) : chAux       l1     cls1   clcls2
6      | otherwise = (c2,          l2) : chAux           l2  clcls1 cls2
```

lines 1 and 2 are calculated as

$$
\begin{aligned}
T_{nch}\ [\,]\ \text{cls2} &= 1 + T(\text{cls2}) && \text{by [cost:case]} \\
&= 1 + 0 && \text{by [cost:var]} \\
&= 1 && \text{by arithmetic}
\end{aligned}
$$

$$
\begin{aligned}
T_{nch}\ \text{cls2}\ [\,] &= 1 + T(\text{cls1}) && \text{by [cost:case]} \\
&= 1 + 0 && \text{by [cost:var]} \\
&= 1 && \text{by arithmetic}
\end{aligned}
$$

first guard (line 4) of nch definition in line 3, we have

T_{nch} cls1 cls2

= { by [cost:case] }

$1 + T(\ (c1,min\ l1\ l2):chAux\ (min\ l1\ l2)\ cls1\ cls2\ )$

= { by [cost:prim]}

$1 + T(\ (c1,\ min\ l1\ l2)\ ) + T(\ chAux\ (min\ l1\ l2)\ cls1\ cls2)$

= { by [cost:var] }

$1 + 0 + T(\ chAux\ (min\ l1\ l2)\ cls1\ cls2\ )$

= { by [cost:func] }

$1 + 0 + T(\ min\ l1\ l2\ ) + T(cls1) + T(cls2) + T_{chAux}\ (min\ l1\ l2)\ cls1\ cls2$

= { by [cost:var(s)] }

$1 + 0 + 0 + 0 + 0 + T_{chAux}\ (min\ l1\ l2)\ cls1\ cls2$

= { by arithmetic}

$1 + T_{chAux}\ (min\ l1\ l2)\ cls1\ cls2$

= { by definition of T_{chAux} }

$1 + length(n + m)$

Consequently, $O(\mathsf{nch}) = |cls1| + |cls2|$ where $cls1$ and $cls2$ are the input lists.

3.4.3 Complexity of function njn

Same as before, we calculate T_f and $T()$ of the inner most auxiliary function for njn.

```
1 jnAux c l l' cls1 [] = (c, l + l') : [ (c1, l1 + l') | (c1, l1) ← cls1 ]
2 jnAux c l l' cls1 clcls2@((c2, l2) : cls2)
3     | c ⇐ c2 = jnAux c l l2 cls1 cls2
4     | otherwise = (c, l + l') :
5         case cls1 of
```

```
6        ((c1, l1) : cls1) | c1 > c2 → jnAux c1 l1 l' cls1 c1cls2
7        _                          → jnAux c2 l2 l cls2 cls1
```

From line 1 we have the following

T_{jnAux} c l l' cls1 [ ] =

{ by [cost:case] }

$1 + T$((c,l+l'): [(c,l1+l') | (c1,l1)←cls1])

= { by [cost:prim] }

$1 + T((c,l+l')) + T([(c,l1+l') \mid (c1,l1)\leftarrow cls1])$

= { by [cost:var] }

$1 + 0 + T([(c,l1+l') \mid (c1,l1)\leftarrow cls1])$

= { by [cost:func] }

$1 + 0 + T((c,l1+l')) + T((c1,l1)\leftarrow cls1) + T_{[\leftarrow]}$ (c,l1+l') ((c1,l1)←cls1)

= { by [cost:var] }

$1 + 0 + 0 + 0 + T_{[\leftarrow]}$ (c,l1+l') ((c1,l1)←cls1)

= { by arithmetic }

$1 + T_{[\leftarrow]}$ (c,l1+l') ((c1,l1)←cls1)

recurrence $T_{[\leftarrow]}$ will transfer all values in the list cls1 to the pair (c1, l1), so $T_{[\leftarrow]}$ = length (cls1), hence T_{jnAux} = length (cls1).

In every remaining pattern matching, lines 2 through 7, function jnAux calls itself, in general (i.e. position of arguments is variable) we have

T_{jnAux} c l l' ((c1,l1):cls1) ((c2,l2):cls2) =

{ by [cost:case] }

$1 + T$(jnAux c l l' cls1 cls2)

= { by [cost:func] }

$1 + T$(c) + T(l) + T(l') + T(cls1) + T(cls2) + T_{jnAux}c l l' cls1 cls2

= { by [cost:var] }

$1 + 0 + 0 + 0 + 0 + 0 + T_{jnAux}$c l l' cls1 cls2

= { by arithmetic }

$1 + T_{jnAux}$ c l l' cls1 cls2

Concluding that T_{jnAux} = length (|cls1| + |cls2|).

Finally, the calculation for njn, we have

```
1 njn [] _ = []
2 njn _ [] = []
3 njn ((c1, l1) : cls1) ((c2, l2) : cls2)
4     | c1 <= c2  =  jnAux c1 l1 l2 cls1 cls2
5     | otherwise =  jnAux c2 l2 l1 cls2 cls1
```

Calculation in lines 1 and 2 being trivial, giving $T_{njn} = 1$ while remaining calculation will clearly lead to $T_{njn} = 1+$ length (|cls1| + |cls2|) due it calls jnAux.

3.4.4 Complexity of function njnk

We recall that the *addition* operation in solving the knapsack problem is the same as that for solving the MC-SP problem, so we simply calculate nchk = nch, therefore

```
nchk :: [(Value,Weight)] → [(Value,Weight)] → [(Value,Weight)]
nchk xs ys = nch xs ys
```

provided that types Value must be equal to Capacity, and Weight equal to Length.

```
1 njnAk wc (v,w) []          = [(v,w)]
2 njnAk wc (v,w) (vw:vws)
3   | w + (snd vw) > wc =                    njnAk wc (v,w) vws
```

```
4 | otherwise          = (v+fst vw, w+snd vw) : njnAk wc (v,w) vws
```

In line 1, the calculation for the step-counting function is as

$$\begin{aligned} T_{njnAk}\ \text{wc (v,w) [ ]} &= 1 + T([\text{(v,w)}]) && \text{by [cost:case]} \\ &= 1 + 0 && \text{by [cost:var]} \\ &= 1 && \text{by arithmetic} \end{aligned}$$

The corresponding for line 2 with guards in lines 3 and 4

T_{njnAk} wc (v,w) (vw:vws) =

$$\begin{aligned} & 1 + T(\text{njnAk wc (v,w) vws}) && \text{by [cost:case]} \\ &= 1 + T(wc) + T(\text{(v,w)}) + T(\text{vws}) + T_{njnAk}\text{wc (v,w) vws}) && \text{by [cost:func]} \\ &= 1 + 0 + 0 + 0 + T_{njnAk}\text{wc (v,w) vws} && \text{by [cost:var]} \\ &= 1 + T_{njnAk}\ \text{wc (v,w) vws} && \text{by arithmetic} \end{aligned}$$

$$\begin{aligned} &= 1 + T(\text{(v + fst vw, w + snd vw): njnAk wc (v,w) vws}) && \text{by [cost:case]} \\ &= 1 + T(\text{(v + fst vw, w + snd vw)}) + T(\text{njnAk wc (v,w) vws}) && \text{by [cost:prim]} \\ &= 1 + 0 + T(\text{njnAk wc (v,w) vws}) && \text{by [cost:var]} \\ &= 1 + 0 + T(wc) + T(\text{(v,w)}) + T(\text{vws}) + T_{njnAk}\text{wc (v,w) vws}) && \text{by [cost:func]} \\ &= 1 + 0 + 0 + 0 + 0 + T_{njnAk}\text{wc (v,w) vws} && \text{by [cost:var]} \\ &= 1 + T_{njnAk}\ \text{wc (v,w) vws} && \text{by arithmetic} \end{aligned}$$

In both cases, $T_{njnAk} = 1$+length (vws).

Finally, for the njnk function we have

```
1 njnk wc (v,w) []     = if   w > wc
2                        then []
3                        else [(v,w)]
4 njnk wc (v,w) vwss = if   w > wc
5                        then vwss
6                        else njnAk wc (v,w) vwss
```

The first case, computing a pair with an empty list (lines 1-3). It does not matter the result from w > wc since in both cases (*True* and *False*) the result will turn in constant computation.

T_{njnk} wc (v,w) [ ] =

$1 + T$(if w > wc then [ ] else [(v,w)])	by [cost:case]
$= 1 + T$(w>wc) + (if w>wc then T([ ]) else T([(v,w)]))	by [cost:if]
$= 1 + 0 + 0 + 0$	by [cost:var,const,var]
$= 1$	by arithmetic

Now, processing a pair against a non empty list, lines 4-6, the computation of the conditional w > wc will depend whether its result is *True*, which is in constant time, or *False* which is a recursive call, turning the function njnk in linear time in the size of the input list (vwss) through the function njnAk.

T_{njnk} wc (v,w) vwss =

$1 + T$(if w > wc then vwss else njnAk wc (v,w) vwss)	by [cost:case]
$= 1 + T$(w>wc) + (if w>wc then T(vwss) else T_{njnAk} wc (v,w) vwss)	by [cost:if]
$= 1 + 0 +$ (if w>wc then T(vwss) else T_{njnAk} wc (v,w) vwss)	by [cost:var]
$= 1 + 0 + T$(vwss)	(w>wc) is *True*
$= 1 + 0 + 0$	by [cost:var]
$= 1$	by arithmetic
$= 1 + 0 + T_{njnAk}$ wc (v,w) vwss	(w>wc) is *False*
= 1+ length (vwss)	by definition of T_{njnAk}

So far, we have that the functions computing a non-path-tracking solutions take at most $O(addition\ of\ lengths\ of\ input\ list(s))$. That is,

- nch takes $O(|cls1| + |cls2|)$,
- nchk takes $O(|vws1| + |vws2|)$,
- njn takes $O(|cls1| + |cls2|)$,

- njnk takes $O(|vwss|)$.

For the functions holding (storing) the path or route of the solution, we have

- nchP does not compute any path deletion or insertion.
- nchkP does not compute any path deletion or insertion.
- njnP computes path processing through the function jnPath.
- njnkP computes path processing through the operator (++).

3.4.5 Complexity of function njnP

We analyse the full function to identify the complexity, starting from the external auxiliary function jnPath.

```
1  jnPath  []  ys = ys
2  jnPath  xs  [] = xs
3  jnPath  x'@(x:xs)  y'@(y:ys) =
4        if last x' == y then x'++ys
5        else                 y'++xs
```

T_{jnPath} [ ] ys =

{ line 1 }

$1 + T(\text{ys})$ by [cost:case]

$= 1 + 0$ by [cost:var]

$= 1$ by arithmetic

{ line 2 }

$= 1 + T(\text{xs})$ by [cost:case]

$= 1 + 0$ by [cost:var]

$= 1$ by arithmetic

{ line 3 }

$= 1 + T(\text{if last x' == y then x'++ys else y'++xs})$ by [cost:case]

$= 1 + T(\text{last x'==y}) + (\text{if last x' == y then } T(\text{x'++ys}) + T(\text{else y'++xs}))$ by [cost:if]

= { According to GHC.List library, function last has linear complexity }

= 1+ length (x') +(if last x' == y then x'++ys else y'++xs)

= { last x' == y is *True*, line 4 }

= 1+ length (x') $+T$(x'++ys)

= 1+ length (x') +length (x') by 3.12

$= 1 + 2\times$length (x') by arithmetic

= { last x' == y is *False*, line 5 }

= 1+ length (x') $+T$(y'++xs)

= 1+ length (x') + length (y') by 3.12

So, the function jnPath has complexity $O(n \uparrow m)$, where n is two times the length of the first input list whereas m is the length of the second input list.

Now, the analysis of the inner auxiliary function, jnAux,

```
1 jnAux c l l' p p' cls1 [] = (c, l + l', jnPath p p') :
2                            [ (c1, l1 + l', jnPath p' p1) | (c1, l1, p1) <- cls1 ]
3 jnAux c l l' p p' cls1 clcls2@((c2, l2, p2) : cls2)
4       | c <= c2   = jnAux c l l2 p p2 cls1 cls2
5       | otherwise = (c, l + l', jnPath p p') :
6         case cls1 of
7         ((c1, l1, p1) : cls1) | c1 > c2 -> jnAux c1 l1 l' p1 p' cls1 clcls2
```

```
8 |  _                                -> jnAux c2 l2 l  p2 p  cls2 cls1
```

In lines 1 and 2 we have

T_{jnAux} c l l' p p' cls1 [ ] =

{ by [cost:case] }

$1 + T$((c,l+l', jnPath p p'): [(c,l1+l',jnPath p' p1) | (c1,l1,p1)←cls1])

= { by [cost:prim] }

$1 + T$((c,l+l',jnPath p p')) + T([(c,l1+l',jnPath p' p1) | (c1,l1,p1)←cls1])

= { by [cost:var], where $k = (|p| \uparrow |p'|)$ by definition of jnPath }

$1 + k + T$([(c,l1+l',jnPath p' p1) | (c1,l1,p1)←cls1])

= { by [cost:func] }

$1 + k + T$((c,l1+l',jnPath p' p1)) + T((c1,l1,p1)←cls1)+

$T_{[\leftarrow]}$ (c,l1+l',jnPath p' p1) ((c1,l1,p1)←cls1)

= { by [cost:var], where $q = (|p'| \uparrow |p1|)$ by definition of jnPath }

$1 + k + q + 0 + T_{[\leftarrow]}$ (c,l1+l',jnPath p' p1) ((c1,l1,p1)←cls1)

= { by arithmetic }

$1 + k + q + T_{[\leftarrow]}$ (c,l1+l',jnPath p' p1) ((c1,l1,p1)←cls1)

= { let $l = (p' \uparrow p1)$, and length(cls1) from definition of jnPath }

$1 + k + q + l \times$(length (cls1))

Taking into account the worst case (i.e. the length of each input list), we have T_{jnAux} c l l' p p' cls1 cls2 = (length (cls1)) × (length (cls2))

Finally, calculating the complexity of njnP we have

```
1 | njnP [] _ = []
2 | njnP _ [] = []
3 | njnP ((c1, l1, p1) : cls1) ((c2, l2, p2) : cls2)
4 | c1 ≤ c2 = jnAux c1 l1 l2 p1 p2 cls1 cls2
5 | otherwise = jnAux c2 l2 l1 p2 p1 cls2 cls1
```

Lines 1 and 2 are trivial, having T_{njnP} {empty cases} = $O(1)$ whereas lines 3 -

5 depend on the complexity of jnAux function, that is T_{njnP} cls1 cls2 $= O(m \times n)$, where n is the length of list cls1 and m is the length of list cls2.

3.4.6 Complexity of function njnkP

We add the (++) operator to function njnk, in order to store the trace the path of the solution. We start with the auxiliary function njnAkP

```
1 njnAkP wc (v,w,p) []          = [(v,w,p)]
2 njnAkP wc (v,w,p) (vw:vws)
3   | w + (snd3 vw) > wc = njnAkP wc (v,w,p) vws
4   | otherwise              = (v+fst3 vw, w+snd3 vw, (trd3 vw)++p) : njnAkP wc (v,w,p) vws
```

The calculation of the complexity for the function definitions in lines 1 and 3 is quite simple,

{ line 1}

T_{njnAkP} wc (v,w,p) [ ] =

$1 + T$([(v,w,p)])	by [cost:case]
$= 1 + 0$	by [cost:var]
$= 1$	by arithmetic

{ lines 2 and 3 }

T_{njnAkP} wc (v,w,p) (vw:vws) =

$1 + T$(njnAkP wc (v,w,p) vws)	by [cost:case]
$= 1 + T$(wc) $+ T$((v,w,p)) $+ T$(vws) $+ T_{njnAkP}$ wc (v,w,p) vws	by [cost:func]
$= 1 + 0 + 0 + 0 + T_{njnAkP}$ wc (v,w,p) vws	by [cost:var]
$= 1 + T_{njnAkP}$ wc (v,w,p) vws	by arithmetic
= 1+ length (vws)	by def. of njnAk

for the final line we have

T_{njnAkP} wc (v,w,p) (vw:vws) =

{ by [cost:case] }

$= 1 + T$((v+fst3 vw, w+snd3 vw, (trd3 vw)++p) : njnAkP wc (v,w,p) vws)

{ by [cost:prim] }

$= 1 + T$((v+fst3 vw, w+snd3 vw, (trd3 vw)++p)) + T_{njnAkP} wc (v,w,p) vws

{ by [cost:var], let t be the length of list resulting from trd3vw }

$= 1 + t + T_{njnAkP}$ wc (v,w,p) vws

{ from definition of function njnAk }

$= 1 + t +$ length(vws)

Since t was derivable from pairs coming out of list vws, then we have $T_{njnAkP} = 1 + 2\times$ length(vws) = length(vws) in the worst case.

The final function to evaluate is njnkP. Same as function njnk, first cases are trivial leading to a constant complexity $T_{njnkP} = O(1)$ when the list to process is empty. The non-empty case for the input list implemented as the following in line 6,

```
1 njnkP wc (v,w,p) []   = if   w > wc
2                         then []
3                         else [(v,w,p)]
4 njnkP wc (v,w,p) vwss = if   w > wc
5                         then vwss
6                         else njnAkP wc (v,w,p) vwss
```

T_{njnkP} wc (v,w,p) vwss =

$1 + T$(if w > wc then vwss else njnAkP wc (v,w,p) vwss)	by [cost:case]
$= 1 + T$(w>wc) + (if w>wc then T(vwss) else T_{njnAk} wc (v,w,p) vwss)	by [cost:if]
$= 1 + 0 +$ (if w>wc then T(vwss) else T_{njnAk} wc (v,w,p) vwss)	by [cost:var]
$= 1 + 0 + T$(vwss)	(w>wc) is *True*
$= 1 + 0 + 0$	by [cost:var]
$= 1$	by arithmetic
$= 1 + 0 + T_{njnAk}$ wc (v,w,p) vwss	(w>wc) is *False*
$= 1+$ length (vwss)	by definition of T_{njnAkP}

As summary, we have

- nchP same as nch,
- nchkP same as nchk,
- njnP takes $O(|cls1| \times |cls2|)$,
- njnkP takes $O(|vws|)$.

Chapter 4

Benchmarking: eager vs lazy evaluation, path vs non-path tracking

Throughout this thesis we have defined functions in order to solve specific problems in the pathfinding context. Those functions are implemented as purely functional and lazy-evaluated. Another way to compute such functions with the same results is through eager evaluation.

Secondly important for this chapter is the treatment of path tracking, which turns to be the most noticeable difference in performance between the above evaluations. So far, Nilsson's functions cover only the computation of the cost of the specific pathfinding problem.

This chapter comprises two case studies, as before, the MC-SP problem and the knapsack problem, starting with the latter. For each case we will give the performance, that is, times, memory allocation, and the percentage of garbage collection used, bearing in mind that we tried every execution on a processor 2.2 GHz Intel Core i7. Each measurement is the mean of three runs, recording the time taken.

Profiling was used only to determine the performance of different functions on each program. Then a separate run was done without profiling, so that the overheads of profiling do not have a bearing on the results.

4.1 Eager vs lazy evaluation

One important issue about functional programming languages, specifically for those pure functional, is the lazy evaluation. This mechanism allows an expression to be evaluated only when it is needed. On the other hand, an eager process will evaluate all the expressions starting from the inner-most.

Different authors coincide about the definition for a strict function as the eager counterpart in a programming language. Let us take the one in [Bir15]: A Haskell function f is said to be *strict* if $f\ undefined = undefined$, and *nonstrict* otherwise. The $undefined$ value here is also known as *bottom* and denoted with the symbol $\perp$.

Given $x, y \in S$, where S is any set in the family of typeclass Num, we have that all of our operators studied so far are strict, as in

- function maximum, $(\uparrow)$, where $(x \uparrow \perp) = \perp = (\perp \uparrow y) = (\perp \uparrow \perp)$,
- function minimum, $(\downarrow)$, where $(x \downarrow \perp) = \perp = (\perp \downarrow y) = (\perp \downarrow \perp)$, and finally
- arithmetic addition, $(+)$, having $(x + \perp) = \perp = (\perp + y) = (\perp + \perp)$.

So far the functions **choose** and **join** either from MC-SP or knapsack are lazy, that is, they are not evaluated in advance to verify whether or not a $\perp$ is present. However, the difference in time and space consumption is minimum for the *cost* (**not** the path) between evaluations eager and lazy, since their operators are, by nature, strict, as we shown above. In order to turn **choose** and **join** strict, we appeal to the library Control.DeepSeq, which manages the function deepseq, instead of applying

the function seq provided in the native library Prelude, since the former traverses data structures deeply.

We have defined our solution by a pair as data structure for the *cost*. So, we have

```
mkStrictPair x y =
  let xy = (x, y)
  in  deepseq xy xy
```

4.2 Path tracking

There is a general type signature for both MC-SP and knapsack problems,

```
solution :: ((x,y),Path)
```

where (x, y) represents either the cost $(Capacity, Length)$ for MC-SP problem or the cost $(Value, Weight)$ for the knapsack problem. At the end, both problems have the following structure

```
solution :: ((Int,Int),[Int])
```

Since the path is a list (of integers), the operator ++ or *append* is called in order to build the complete path along the graph traversal. Such operator performs slower in eager evaluation respect to the lazy one, mainly because the lazy evaluation does not force *append* to carry out until the very end, computing less lists to be appended.

Now, to turn the path tracking strict, we have

```
mkStrictTriple x y z =
  let xyz = (x, y, z)
  in  deepseq xyz xyz
```

where z is defined of type

```
type Path = [Int]
```

4.3 Case study: knapsack

We have seen in Section 3, specifically in the relation 3.11, that function **nch** and **nchk** are practically the same. It is also the case for path-tracking and eager evaluation analyses, therefore we will focus on **njnK** function.

4.3.1 Eager evaluation

There are few changes to make to the code seen in Chapter 3, specifically in the recursive part since the base case does not compute any pair, so our version of *strict* is as follows

```
njnAkS :: Weight →(Value,Weight) →[(Value,Weight)] →[(Value,Weight)]
njnAkS wc (v,w) []       = [(v,w)]
njnAkS wc (v,w) (vw:vws)
                  if   w + (snd vw) > wc  then  njnAkS wc (v,w) vws
                  else mkStrictPair (v+fst vw) (w+snd vw) : njnAkS wc (v,w) vws
```

4.3.2 Path tracking

Extending the types from the knapsack version in Chapter 3, and adding list manipulation to track paths, we have

```
njnkP :: Weight →(Value,Weight,Path) →[(Value,Weight,Path)] →[(Value,Weight,Path)]
njnkP wc (v,w,p) [] = if    w > wc
                      then  []
                      else  [(v,w,p)]
njnkP wc (v,w,p) vwss = if    w > wc
                        then  vwss
                        else  njnAkP wc (v,w,p) vwss
```

and just *appending* lists in our recursive njnAk function

```
njnAkP :: Weight →(Value,Weight,Path) →[(Value,Weight,Path)] →[(Value,Weight,Path)]
njnAkP wc (v,w,p) []          = [(v,w,p)]
njnAkP wc v1@(v,w,p) (vw:vws)
  | w + (snd3 vw) > wc =                                          njnAkP wc v1 vws
  | otherwise          = (v + fst3 vw, w + snd3 vw, (trd3 vw)++p) : njnAkP wc v1 vws
```

Notice this function is right associative only.

4.3.3 Benchmarking

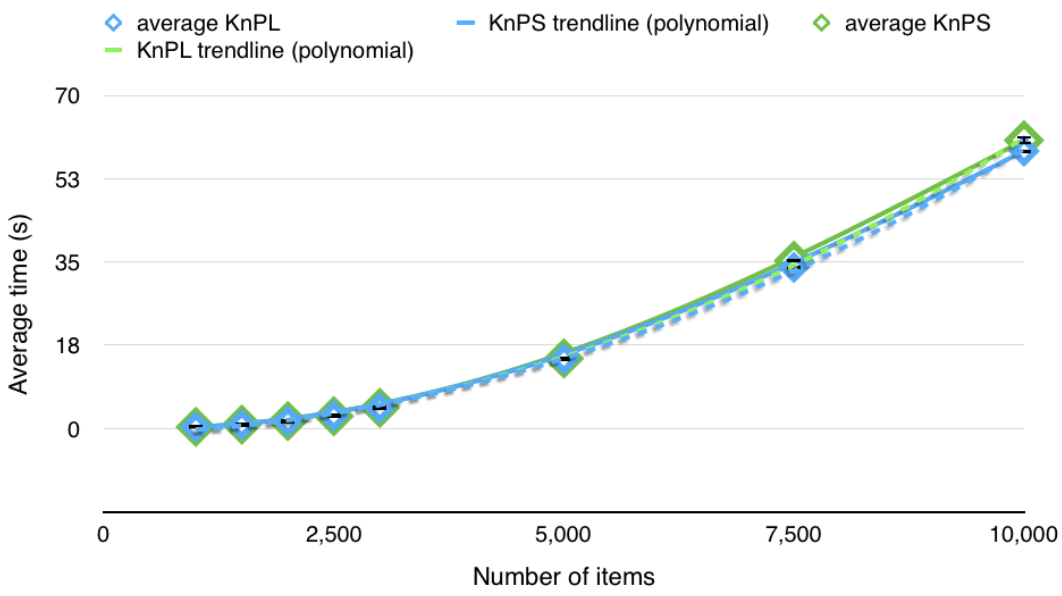
We present data in the form of tables first in order to show the span for items and capacity. The labels of values and weights are randomly generated.

knapsack lazy non-path tracking (KnPL)

items	capacity	average KnPL	data					var	stdev
1,000	9,500	0.323	0.311	0.318	0.320	0.323	0.342	0.000	0.012
1,500	14,250	0.801	0.791	0.838	0.789	0.802	0.785	0.000	0.022
2,000	19,000	1.538	1.530	1.516	1.574	1.552	1.518	0.001	0.025
2,500	23,750	2.629	2.651	2.549	2.623	2.674	2.650	0.002	0.048
3,000	28,500	4.320	4.248	4.362	4.363	4.287	4.338	0.003	0.051
5,000	47,500	14.594	14.747	14.601	14.469	14.469	14.683	0.016	0.125
7,500	71,250	33.845	33.911	34.448	33.623	33.561	33.681	0.131	0.362
10,000	95,000	58.294	59.138	56.739	58.726	58.677	58.192	0.869	0.932

knapsack strict non-path tracking (KnPS)

items	capacity	average KnPS	data					var	stdev
1,000	9,500	0.326	0.315	0.332	0.318	0.322	0.344	0.000	0.012
1,500	14,250	0.804	0.794	0.800	0.805	0.784	0.835	0.000	0.019
2,000	19,000	1.553	1.544	1.539	1.558	1.555	1.570	0.000	0.012
2,500	23,750	2.587	2.594	2.607	2.612	2.561	2.562	0.001	0.024
3,000	28,500	4.336	4.250	4.410	4.337	4.333	4.352	0.003	0.057
5,000	47,500	14.706	14.676	14.598	14.721	14.590	14.944	0.021	0.144
7,500	71,250	35.263	35.647	34.938	34.876	35.307	35.549	0.122	0.349
10,000	95,000	60.571	59.971	60.012	61.720	60.962	60.190	0.573	0.757

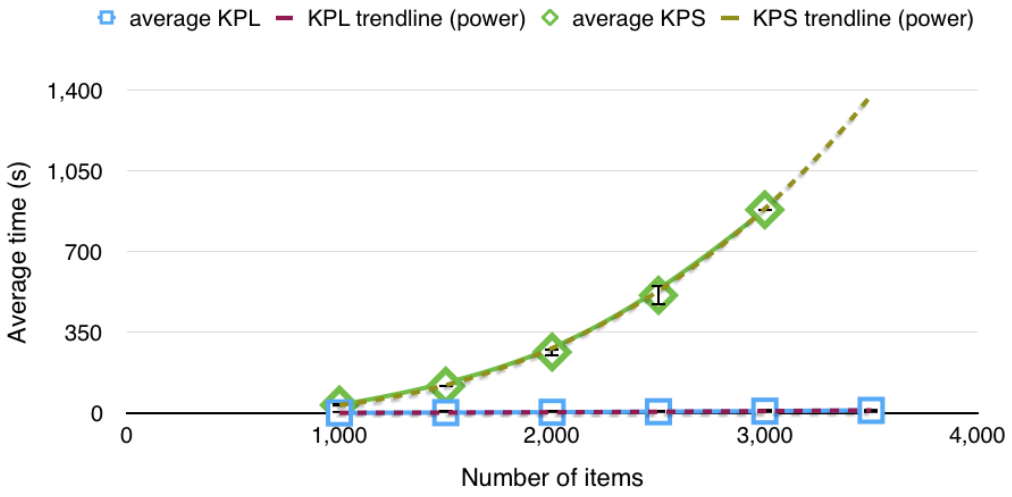


knapsack lazy path tracking (KPL)

items	capacity	average KPL	data					var	stdev
1,000	9,500	0.636	0.630	0.653	0.629	0.630	0.640	0.000	0.010
1,500	14,250	1.681	1.696	1.682	1.679	1.686	1.664	0.000	0.012
2,000	19,000	3.179	3.180	3.144	3.199	3.185	3.188	0.000	0.021
2,500	23,750	5.399	5.360	5.440	5.347	5.444	5.403	0.002	0.045
3,000	28,500	8.068	8.175	8.082	8.024	7.959	8.100	0.007	0.081
3,500	33,250	11.303	11.354	11.438	11.108	11.394	11.221	0.018	0.136

knapsack strict path tracking (KPS)

items	capacity	average KPS	data					var	stdev
1,000	9,500	34.727	34.402	34.777	34.829	34.925	34.702	0.040	0.199
1,500	14,250	116.518	118.861	116.912	116.920	117.192	112.704	5.202	2.281
2,000	19,000	264.391	266.522	268.586	260.074	267.131	259.641	17.711	4.208
2,500	23,750	510.572	514.289	509.180	500.116	517.540	511.734	43.746	6.614
3,000	28,500	881.510	880.634	884.408	881.860	881.744	878.902	4.036	2.009



We just plot some of the strict path-tracking data for knapsack, since the results for $4,096$ items will draw a point really high in the chart as is denoted in the respective table (above). Such huge difference is due the strict application on the operator ++. More details in the differences are shown in the following charts.

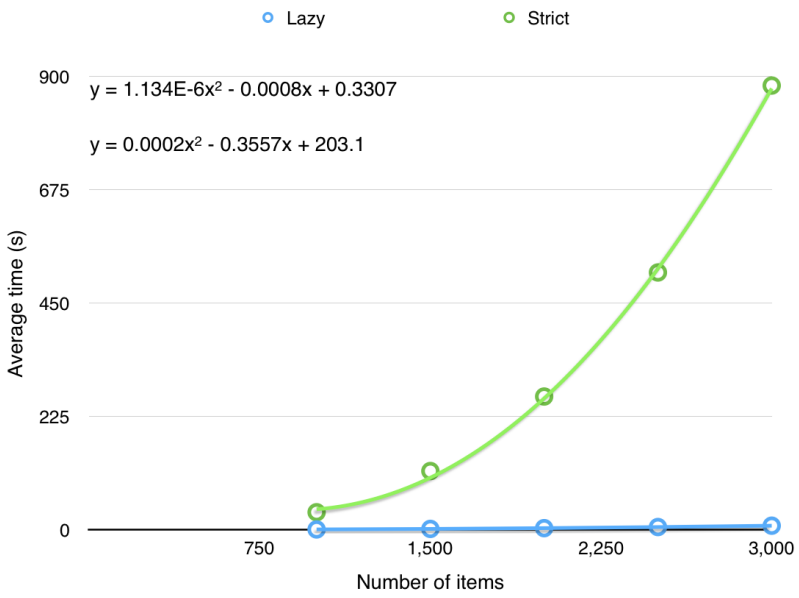


Figure 4.1: Polynomial trendline for lazy and strict versions of knapsack with path tracking

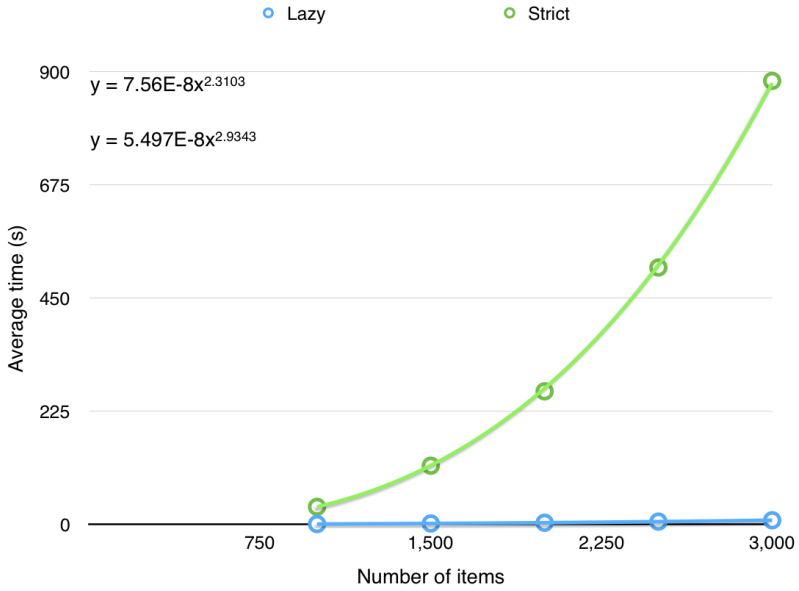


Figure 4.2: Power trendline for lazy and strict versions of knapsack with path tracking

4.4 Case study: MC-SP

In the single source approach we will show the performance of the lazy against strict evaluations for both cases, with and without path tracking, whereas the performance for the all-pairs approach will be based on the lazy non-path tracking only. Charts and tables are shown in each case.

4.4.1 Random graphs

In the many ways of constructing a random graph we will pick the one described in [Kin96], where David King implemented a monadic function to generate random permutations by constructing an array of integers, and swapping each index once with a random index value. That is, given v number of vertices, the function

```
randomPerm :: Int → [Int]
```

will generate a list with the vertices ordered randomly. Then, given v as the number of vertices and e as the number of edges, the function

```
randomE :: Int → Int → [Edge]
```

will return a list of pairs of vertices (i.e. edges) also sorted randomly. From these two functions a random graph is generated:

```
randomGraph :: Int → Int → Graph
```

where Graph has been previously defined as:

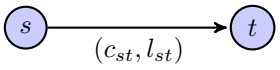
```
type Graph = Table [Vertex]
```

Finally, we generate the labels randomly, so we can compute the functions **nch** and **njn** in an arbitrary graph of size v and as dense as the size of e (the number of edges).

4.4.2 Single source approach

Before starting to the code or the algorithm for the single source approach for our proposal, that is using **nch** and **njn**, we will explain the kind of graph and its traversing. We will focus for this approach on a directed acyclic graph, so the maximum number of edges is $\frac{v(v-1)}{2}$, where v is the number of vertices.

The trivial step is the one having two vertices and therefore only one edge. Assume the carrier set $S = \mathbb{N} \cup \{\infty\}$

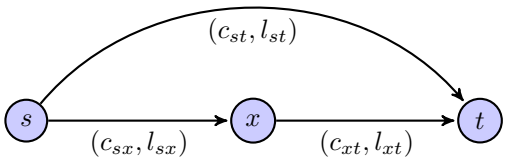


and its adjacency matrix

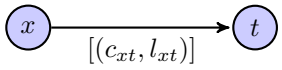
	s	t
s	$(0, \infty)$	(c_{st}, l_{st})
t	$(0, \infty)$	$(0, \infty)$

where $(0, \infty)$ is the *zero* element of S, meaning there is no path from between two vertices.

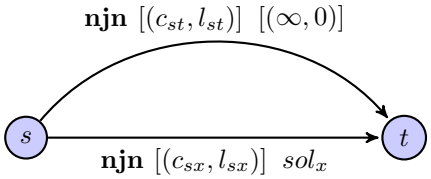
Another way to view the current solution for bigger graphs, is through recursion. So, having the following graph



we can start our computations from the target vertex t backwards, and storing partial solutions. Let us say that sol_x, reading as *solution at* x, refers to



And therefore



Let us depict a final example of a fully connected graph to describe the complete situation for the single source approach.

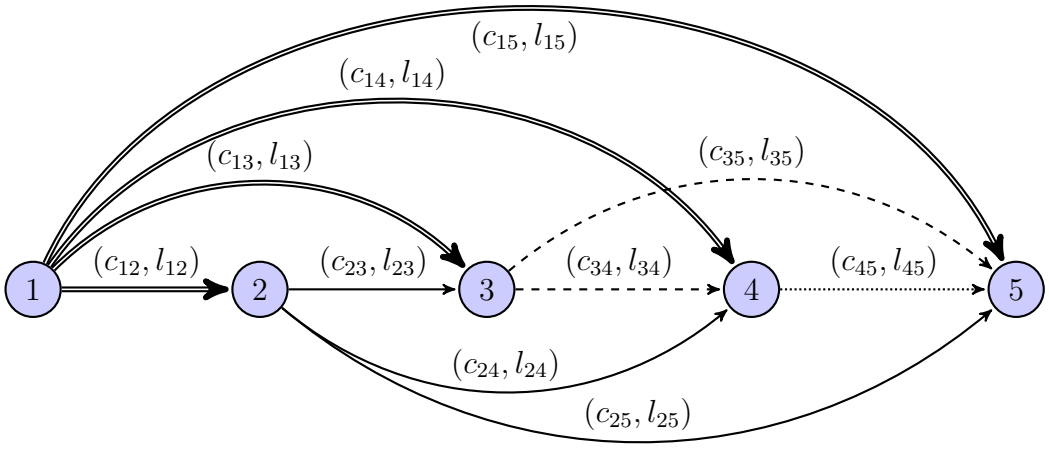


Figure 4.3: Reusing previous computations

The proposal solution for computing MC-SP problem from vertex 1 to vertex 5 is:

double arrow	single arrow (sol_2)	dashed arrow (sol_3)
nch	**nch**	**nch**
njn (c_{15}, l_{15}) sol_5	**njn** (c_{25}, l_{25}) sol_5	**njn** (c_{35}, l_{35}) sol_5
njn (c_{14}, l_{14}) sol_4	**njn** (c_{24}, l_{24}) sol_4	**njn** (c_{34}, l_{34}) sol_4
njn (c_{13}, l_{13}) sol_3	**njn** (c_{23}, l_{23}) sol_3	
njn (c_{12}, l_{12}) sol_2		

where sol_4, dotted arrow, is the pair (c_{45}, l_{45}) and sol_5, not arrow at all, is the pair $(\infty, 0)$, the *unit* element.

The above table shows that **njn** is performed as expected whereas each function **nch** computes a list of *joins*, we will call it **chooses** in the following code.

Finally, we can store the results from each sol_i in (descending) order to avoid repeated computations. We will do so in an unidimensional array, where i in sol_i represents the vertex i.

Let us call our single-source approach function to solve MC-SP problem as solutionSS.

```
solutionSS :: Int → Int → Int → [(Capacity,Length)]
solutionSS v e seed = sol ! 1
 where
  sol = array (1,v) ([(v,oneNonPath) ] ++
                     [(i,chooses    (randomweights !! (i−1))) | i ← [v−1,v−2..1] ])

  randomgraph    = elems $ randomGraphFilled v e seed
  randomlist     = zip (randomList 20 seed) (randomList 50 (seed+1))
  randomweights = mixListsPairs randomgraph randomlist

  chooses :: [(Int,Int,Int)] → [(Capacity,Length)]
  chooses []       = zeroNonPath
  chooses (x:xs) = nch (njn [(snd3 x, trd3 x)] (sol ! (fst3 x)) ) ( chooses xs )
```

This code practically follow the definition in the above graphs and tables, where sol is an array holding partial solutions starting from sol_t (oneNonPath or $[(\infty, 0)]$, the unit of *multiplication*). The computation starts with the target vertex and then we compute further solutions backwards. Function chooses computes **nch**'s as far as its input list has pairs of random values, otherwise it returns $[(0, \infty)]$ which is the unit of *addition*.

This function, solutionSS, always terminates since v is a finite number of vertices and the internal list $[v-1, v-2 \ldots, 1]$ bounds its size. Function randomweights returns a triple with the index i representing the current vertex and random values for a *Capacity* and a *Length*. If, after a random graph is generated, there is not transitivity (path connection) from the start vertex s to the target vertex t, then the *zero* or $[(\infty, 0)]$ is returned as solution by solutionSS.

Strict evaluation

We start analysing the pairs. We place our strictness function just in one line of code for **nch** (now **nchS**) and two lines for **njn** (now **njnS**). For **nchS**,

```
...
    | c1 == c2  = mkStrictPair c1 (min l1 l2) : chAux (min l1 l2) cls1 cls2
...
```

and for **njnS**,

```
...
  jnAux c l l' cls1 [] = mkStrictPair c (l + l') :
                         [mkStrictPair c1 (l1 + l') | (c1, l1) ← cls1]
...
    | otherwise = mkStrictPair c (l + l') :
...
```

Similar to knapsack problem, lazy and eager evaluation without tracking paths, have practically the same results for performance, since the operators involved are strict.

Path tracking

Recall the types for both functions **nch** and **njn** are:

```
nch, njn :: [(Capacity,Length)] → [(Capacity,Length)] → [(Capacity,Length)]
```

Having then

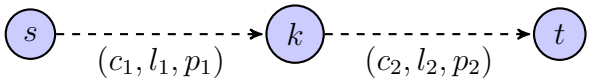
```
nch, njn :: [(Capacity,Length,Path)] → [(Capacity,Length,Path)]
            → [(Capacity,Length,Path)]
```

Since the nature of **nch** is precisely to pick a solution between two or more choices, this function should not affect the path of such choices, therefore this func-

tion has trivial implementation regarding paths. The focus is then in **njn**, according to the following. That is, applying the function **njn**, we have, in general

```
njn cls1 cls2 = [(c1 ↑ c2 , l1 + l2 , p1 ++ p2)]
```

Graphically we have,



where both paths p_1 and p_2 include the vertex k in their lists, resulting in the append operation above, ++, being no trivial, since our function **njn** is commutative. The resulting path of this *multiplication* is the same, it does not matter if the inputs are

$$[s, \ldots, k] \ `njn' \ [k, \ldots, t] = [s, \ldots, k, \ldots, t]$$

or

$$[k, \ldots, t] \ `njn' \ [s, \ldots, k] = [s, \ldots, k, \ldots, t]$$

Then we will need an auxiliary function to glue two paths.

```
jnPath :: Path → Path → Path
jnPath [] ys = ys
jnPath xs [] = xs
jnPath x'@(x:xs) y'@(y:ys)
  | last x' == y = x' ++ ys
  | otherwise    = y' ++ xs
```

Since the append operation just takes $\mathcal{O}(n)$ where n is the length of the first input list, the complexity of our functions **nch** and **njn** remains $\mathcal{O}(m + n)$, see 3.4.2.

This new function is being called every time **njn** adds a new pair (now a triple) into the resulting list. Here is the code where this function is called by **njn**. Specifically the changes are in **jnAux**

```
jnAux :: Capactiy → Length → Length → Path → Path
          → [(Capacity,Length,Path)] → [(Capacity,Length,Path)]

jnAux c l l' p p' cls1 [] =   (c,   l + l', jnPath p p') :
                              [ (c1, l1 + l', jnPath p' p1) | (c1, l1, p1) ← cls1 ]

jnAux c l l' p p' cls1 clcls2@((c2, l2, p2) : cls2)
  | c ≤ c2    = jnAux c l l2 p p2 cls1 cls2
  | otherwise = (c, l + l', jnPath p p') :
    case cls1 of
      ((c1, l1, p1) : clss1) | c1 > c2 → jnAux c1 l1 l' p1 p' clss1 clcls2
       _                               → jnAux c2 l2 l p2 p cls2 cls1
```

4.4.3 Benchmarking

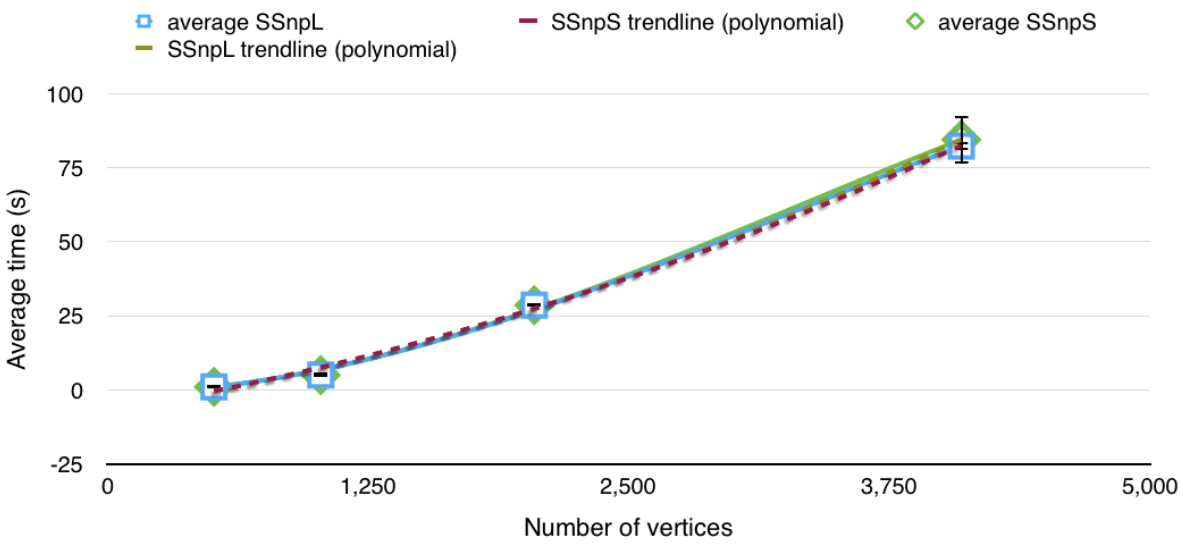
The following are the corresponding data tables and graphics of the single-source approach.

single source MC-SP non path tracking lazy (SSnpL)

vertices	edges	average SSnpL	data					var	stdev
512	131,072	0.972	0.981	0.973	0.961	0.960	0.987	0.000	0.012
1,024	524,288	4.986	4.987	5.010	5.000	4.979	4.952	0.000	0.022
2,048	2,097,152	28.557	28.773	28.829	28.193	28.722	28.267	0.091	0.302
4,096	8,388,608	82.235	87.395	84.490	87.004	84.242	68.045	64.960	8.060

single source MC-SP non path tracking strict (SSnpS)

vertices	edges	average SSnpS	data					var	stdev
512	131,072	0.964	0.981	0.961	0.937	0.974	0.966	0.000	0.017
1,024	524,288	4.933	4.946	4.832	4.998	4.890	4.997	0.005	0.072
2,048	2,097,152	28.518	28.525	28.104	29.647	28.016	28.300	0.436	0.661
4,096	8,388,608	84.390	85.034	85.993	83.427	84.654	82.841	1.595	1.263

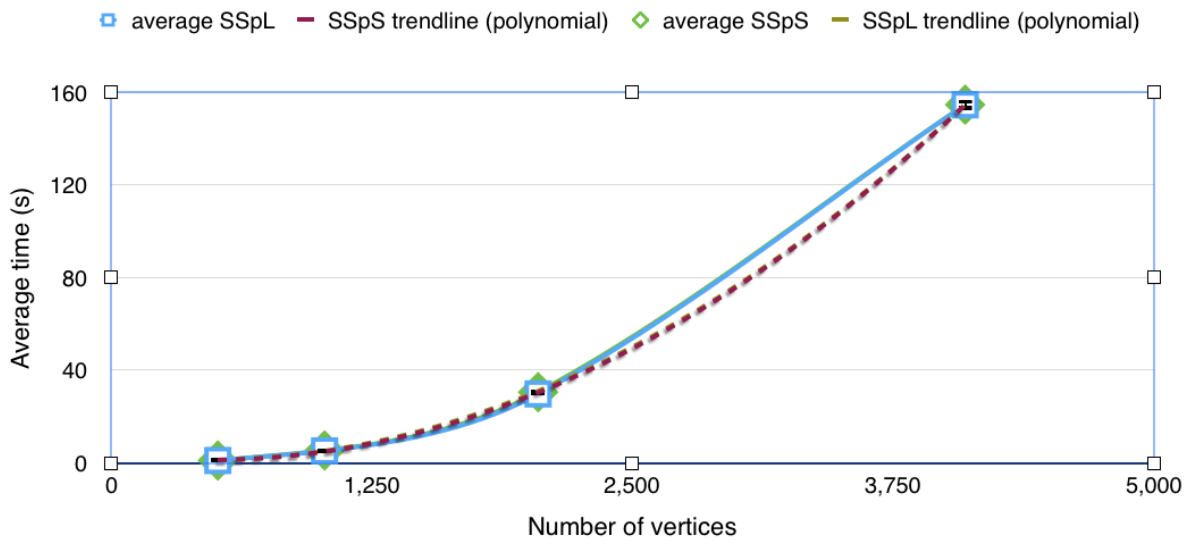


single source MC-SP non path tracking lazy (SSpL)

vertices	edges	average SSpL	data					var	stdev
512	131,072	1.033	1.027	1.012	1.007	1.080	1.041	0.001	0.029
1,024	524,288	5.202	5.189	5.219	5.173	5.266	5.164	0.002	0.041
2,048	2,097,152	29.778	29.835	29.734	29.777	29.853	29.693	0.005	0.067
4,096	8,388,608	154.604	156.339	155.582	152.767	154.588	153.744	2.020	1.421

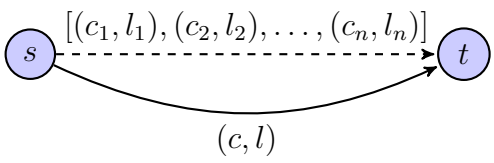
single source MC-SP path tracking strict (SSpS)

vertices	edges	average SSpS	data					var	stdev
512	131,072	1.037	1.047	1.009	1.030	1.059	1.041	0.000	0.019
1,024	524,288	5.374	5.403	5.318	5.264	5.324	5.559	0.013	0.115
2,048	2,097,152	30.531	30.745	30.305	30.151	30.696	30.758	0.080	0.283
4,096	8,388,608	154.662	153.730	155.285	156.756	156.266	151.274	4.923	2.219



4.4.4 Special case

It is the case when there are basically to main paths from the source vertex to the target vertex.



We assume there is one path and one edge. The path, denoted here with a dashed arrow, is comprised by pairs that meet the rule 3.6. Since there is only an edge to be computed with the list, we derive the following.

```
chOneEdge :: (Capacity,Length) → [(Capacity,Length)] → [(Capacity,Length)]
chOneEdge (c,l)     []          = [(c,l)]
chOneEdge (c,l) ((c',l'):cls)
   | c  > c'    = [(c,l)]
   | c == c'    = [(c, l ↓ l')]
   | otherwise = [(c',l')]
```

In this approach it does not matter if the list *cls* contains undefined elements, since it will never be compared. This function fulfils the same constraints and rules as in **nch**, as no matters what, it returns always a singleton.

4.4.5 All pairs approach

We have solved the MC-SP problem, with a dynamic programming approach, but only for cases where graphs are acyclic. For the rest types of graphs, we appeal to Floyd-Roy-Warshall algorithm (FRW). This algorithm assumes that the operations over a square matrix, modelling the graph, are associative and that the *multiplication* distributes over *addition*. We can now, by the application of **nch** and **njn**, hold this property.

In this section we will describe and adapt FRW into an algorithm that solves the MC-SP problem in the setting of a functional programming, calling this function frw. We will define two versions for this purpose, a purely functional and a stateful one (monadic version). Details on implementations of FRW are not the aim of this thesis but those regarding our functions **nch** and **njn** are. Same as in single source implementation, we will show the benchmarking for the performance on both versions of all pairs approach.

We also saw that the closure operator is part of the Warshall-Floyd-Kleene algorithm. For the purposes of MC-SP problem, we have that, for all $x \in S$, where S is the carrier set of the this problem,

$$x^* = 0 \uparrow x \uparrow (x \downarrow x) \uparrow \ldots \uparrow x^* = x,$$

for the MC counterpart where elements and operators are $(\uparrow, \downarrow, 0, \infty)$. On the other hand, we have that

$$x^* = \infty \downarrow x \downarrow (x + x) \downarrow \ldots \downarrow x^* = x,$$

for the SP counterpart where elements and operators are $(\downarrow, +, \infty, 0)$.

4.4.6 Pure functional version

We start with a general perspective; the function frw takes a graph (in its matrix representation) and returns a graph (matrix as well) with the solutions.

```
frw :: GraphFRW → GraphFRW
frw g = foldr induct g (range (limitsG g))
```

where GraphFRW is type synonym of:

```
type Pair     = [(Capacity,Length)]
type GraphFRW = Array Edge Pair
```

From the vast variety of ways to write a solution to FRW, we picked up the function foldr. Since foldr, from the Prelude library, traverse a data structure given an initial value and computing each element of that data structure with given a function, we just substitute the types of foldr by

1. induct as the function to be applied to every element of the data structure

2. g a graph, in this case our original adjacency matrix.

3. a list of vertices, obtained from `range (limitsG g)` .

The induct function will compute the k-th matrix, or k-th path. That is, for every element in the list of vertices in `range (limitsG g)` , a new matrix is created through `mapG` , which is called from `induct k g` .

```
induct :: Vertex → GraphFRW → GraphFRW
induct k g = mapG (const.update) g
 where
   update :: Edge → Pair
   update (x,y) = nch (g!(x,y)) (njn (g!(x,k)) (g!(k,y)) )
```

```
mapG :: Ix a ⇒ (a → b → c) → Array a b → Array a c
mapG f a = array (bounds a) [ (i, f i (a!i)) | i ← indices a ]
```

4.4.7 Monadic version

This is perhaps a more straightforward implementation from the original FRW algorithm. Given n vertices and a graph xs, function frw computes FRW through the function runST. It returns, similar to the pure functional frw, a graph with the solutions (i.e. all pairs). The *graph* here is also another way to refer to a square matrix, which in this case is mutable.

```
frw n xs = runST $
  do {
   xa ← nLA ((1,1),(n,n)) xs;
   xb ← nLA ((1,1),(n,n)) [];
   let loop(k,i,j) =
         if k > n
         then return ()
         else
```

```
            if i > (n-1)
            then do
                transfer xb xa n 1 2
                loop(k+1,1,2)
            else
                if j > n
                then loop (k,i+1,i+2)
                else do
                        compute xa xb k i j
                        loop(k,i,j+1)
    in loop(1,1,2);
    getElems xa }
```

where function compute is the core of our interest since it performs the the *addition* and *multiplication* operations in order to solve MC-SP problem.

```
compute :: MyA s → MyA s → Int → Int → Int → ST s ()
compute xa xb k i j =
    do
        ij ← readArray xa (i,j)
        ik ← readArray xa (i,k)
        kj ← readArray xa (k,j)
        writeArray xb (i,j) (nch ij (njn ik kj))
```

The following chart and tables are calculated with non path-tracking and computing the just the index $(1, v)$ of the square matrix in order to avoid the display of the whole matrix. Due the huge amount of memory taken by the random generation of pairs (labels on the graphs), we profiled specifically the functions frwF, frwFS, frwM, and frwMS for pure functional and monadic implementations of the FRW algorithm. Then we include the random graph by reading it from a text file.

Here is a sample of profiling the single source approach:

COST CENTRE	MODULE	% time	% alloc.
randomList	Graphs	51.7	51.7
listST.constST	Graphs	18.5	10.8
mapST	Graphs	14.0	17.3
applyST	Graphs	5.8	6.1
randomPerm.swapArray	Graphs	5.1	3.0
randomPerm	Graphs	3.9	10.8
solutionSS	Main	1.0	0.3

Table 4.1: Profiling sspt, single source with path-tracking

4.4.8 Benchmarking

Purely functional lazy all pairs (PapL)

vertices	edges	average PapL	data					var	stdefv
100	5,000	0.895	0.873	0.901	0.903	0.896	0.904	0.000	0.013
200	20,000	6.229	6.070	5.918	6.070	6.594	6.493	0.088	0.296
300	45,000	18.688	18.477	18.810	18.679	18.819	18.655	0.019	0.139
400	80,000	43.478	44.168	43.019	43.391	43.287	43.525	0.183	0.428
600	3,600	529.018	525.715	520.614	521.113	564.365	513.282	410.265	20.255

Purely functional strict all pairs (PapS)

vertices	edges	average PapS	data					var	stdefv
100	5,000	0.915	0.873	0.928	0.923	0.946	0.905	0.001	0.028
200	20,000	5.732	5.794	5.767	5.503	5.790	5.805	0.017	0.129
300	45,000	17.400	17.267	17.508	17.572	17.232	17.423	0.022	0.148
400	80,000	42.403	41.854	42.742	42.842	42.357	42.219	0.161	0.402
600	3,600	462.722	507.986	438.929	432.406	506.187	428.104	1,655.40	40.687

Monadic lazy all pairs (MapL)

vertices	edges	average MapL	data					var	stdefv
100	5,000	0.312	0.304	0.320	0.321	0.299	0.315	0.000	0.010
200	20,000	1.819	1.830	1.844	1.779	1.836	1.804	0.001	0.027
300	45,000	5.934	5.959	5.914	5.962	5.863	5.972	0.002	0.046
400	80,000	12.478	12.419	12.204	12.748	12.463	12.554	0.039	0.198
600	3,600	48.262	48.316	48.126	48.312	48.273	48.282	0.006	0.078

Monadic strict all pairs (MapS)

vertices	edges	average MapS	data					var	stdefv
100	5,000	0.312	0.322	0.297	0.309	0.311	0.319	0.000	0.010
200	20,000	1.824	1.820	1.831	1.816	1.803	1.848	0.000	0.017
300	45,000	5.955	5.926	5.912	6.039	6.045	5.852	0.007	0.084
400	80,000	12.496	12.523	12.445	12.459	12.464	12.587	0.004	0.059
600	3,600	47.973	48.230	48.079	47.787	47.626	48.142	0.065	0.255

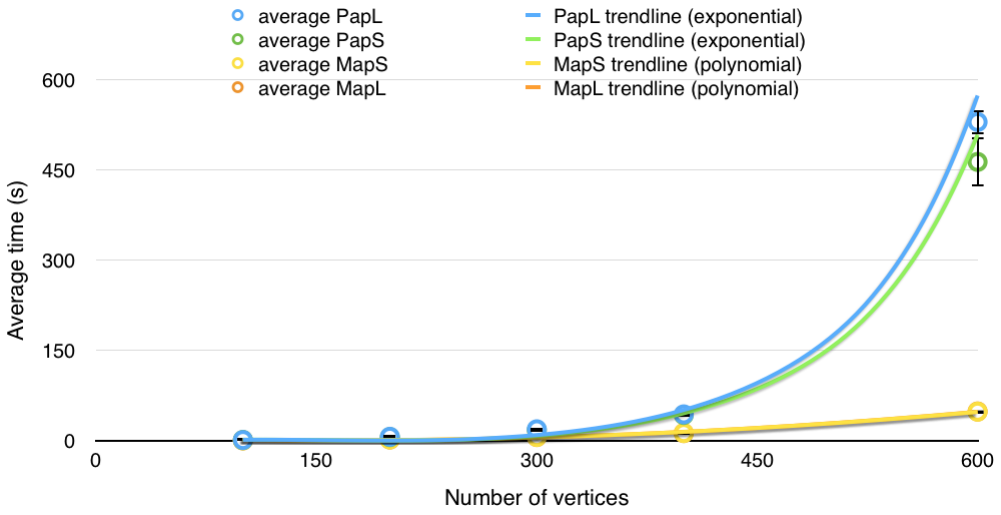


Figure 4.4: Pure functional and monadic implementations for the all-pairs approach

The main difference between *monadic* and *pure-functional* is that the former updates the information over the same data structure (i.e. array) where the latter generates a new one each time is needed. This gives the advantage to the monadic version not only in the speed side, but in the number of problems it can handle over the same computer. See in figure 4.4, that the pure-functional version was able to compute up to 400 vertices whereas the monadic counterpart did it up to the double, and in much less time.

Chapter 5

Conclusion and Further Work

This chapter summarises the previous chapters, and gives a discussion of further work.

We have showed that the application of Nilsson's **choose** and **join** was successful in overcoming the lack of the algebraic distributivity property in at least two pathfinding problems, the joined Maximum-capacity Shortest path and the Knapsack. Two different approaches were tested and proved for such functions application, namely single-source and all-pairs.

We also showed the benefits of the lazy evaluation over the eager one by proving and testing different implementations. In fact, several analyses, charts and tables showed that lazy is faster than its strict counterpart specifically when a graph is medium to highly dense. It is the construction of lists of paths, as part of the solution for a pathfinding problem, where laziness pays off.

Further work

This section summarises some of the possibilities for further work.

There were many aspects that were not considered and that would make the work

more complete. Parallelism, which was not included at all. Functional programming is really good in this matter since the order in which the functions are evaluated is not fixed, therefore some aspects of the current work could be part for this topic, specifically matrix multiplication and dynamic programming.

Another topic of further work is the derivation of a framework that comprises the analysis of operators and the order of the criteria for the pathfinding algebras prior to be composed.

For the order in which the pathfinding algebras are composed is not clear if the analysis we have done can be extended to n number of problems. For example, having Maximum reliability, maximum capacity and shortest path in that precise order.

Work therefore needs to be done to explore such extensions in the context of reasoning and programming.

Bibliography

[Bac06] Roland Backhouse. Regular algebra applied to language problems. *The Journal of Logic and Algebraic Programming*, 66:71–111, 2006.

[Bac11] Roland Backhouse. *Algorithmic Problem Solving*. Wiley, 2011.

[BC75] Roland Backhouse and B.A. Carré. Regular algebra applied to path-finding problems. *J. Inst. Maths Applics*, 15:161–186, 1975.

[Bir15] Richard Bird. *Thinking Functionally with Haskell*. Cambridge University Press, 2015.

[Car79] Bernard Carré. *Graphs and Networks*. Clarendon Press, Oxford, 1979.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[Dol14] Stephen Dolan. Fun with semirings, a functional pearl on the abuse of linear algebra. *ICFP*, pages 101–109, 2014.

[GG11] Alexander Gurney and Timothy Griffin. Pathfinding through congruences. 2011.

[Gla02] Kazimierz Glazek. *A Guide to the Literature on Semirings and their Applications in Mathematics and Information Sciences: With Complete Bibliography*. Springer, 2002.

[GM08] Michel Gondran and Michel Minoux. *Graphs, Dioids and Semirings*. Springer, 2008.

[Gur09] Alexander Gurney. *Construction and verification of routing algebras*. PhD thesis, Cambridge University, 2009.

[Kin96] David J. King. *Functional Programming and Graph Algorithms*. PhD thesis, University of Glasgow, 1996.

[Leh77] Daniel J. Lehmann. Algebraic structures for transitive closure. *Theoretical Computer Science*, 4(1):59–76, 1977.

[Man04] Robert Manger. A new path algebra for finding paths in graphs. *26th International Conference in Information Technology Interfaces ITI 2004*, pages 657–662, 2004.

[Man06] Robert Manger. Composite path algebras for solving path problems in graphs. *Ars Combinatoria*, 78:137–150, 2006.

[Moh02] Mehryar Mohri. Semiring frameworks and algorithms for shortest-distance problems. *Journal of Automata, Languages and Combinatorics*, 7:321–350, 2002.

[Skr00] Anders J. V. Skriver. A classification of bicriterion shortest path (bsp) algorithms. *Asia-Pacific Journal of Operational Research*, 17:199–212, 2000.

[Sta13] Mike Stannett. Com2001: Advanced programming topics a practical guide using haskell and java, 2013.