Overcoming Non Distributivity: A Case Study in Functional Programming

Juan Carlos Saenz-Carrasco [*]
Mike Stannett
The University of Sheffield, Department of Computer Science,
Regent Court, 211 Portobello, Sheffield S1 4DP, United Kingdom
jcsaenzcarrasco1@sheffield.ac.uk m.stannett@sheffield.ac.uk

Among hard problems in Computer Science are those that are easy to write but hard to run, especially when the data structures that represent them are persistent. In this paper we present two different problems, in the fashion of case studies, which share a common algebraic problem: 'multiplication' is not distributive over 'addition' (for example, *bi-criteria* and *knapsack* problems). Our proposed solution comprises two simple non-strict pure functions within a standard search algorithm, and we show that this is enough to overcome the lack of distributivity in these situations. Also, we provide benchmarking between the strict and non-strict versions. We believe that this simple solution provides a useful exercise for students of functional programming, since it fruitfully combines several different aspects of the paradigm—persistent data structures, polymorphic data types, and different evaluations are all involved.

1 Introduction

One of the aims of functional programming (FP) is to write down the simplest and clearest specification to solve a problem without regard to how efficient the result might be. In the vast repertoire of hard problems in Computer Science, there are some (reasonably tractable) problems that can be stated in simple and clear terms, but standard solutions of the sort that might be generated by a Haskell compiler can potentially take an undesirable amount of time to run.

In this paper we describe a case study—suitable for discussion in graduate level programming classes—showing how this situation can arise in the context of algebraic situations in which key algebraic properties fail (in our case, failure of multiplication to distribute over addition), and how a change of approach can render them more easily soluble. This case study focuses on the *bottleneck shortest path* (BSP) problem. We analyse the problem, demonstrate the failure of distributivity, and illustrate how our approach can be applied.

The remainder of this paper is as follows. Sect. 2 outlines some key mathematical definitions and their Haskell implementations. The problem we area addressing is introduced through an example in Sect. 3, which is solved by the proposal in Sect. 4. The core of the paper is in Sect. 5 where implementations are given for solving BSP. We conclude by benchmarking various test cases in Sect. 6.

2 Fundamental definitions

For the purposes of this overview we limit BSP to the context of weighted directed acyclic graphs, although it can also be considered for other types of graph [2]. We write $\mathbb{N}$ for the set of natural numbers

[*]Research supported by the Consejo Nacional de Ciencia y Tecnología, CONACYT (the Mexican National Council for Science and Technology) under grant 411550, scholar 580617 and CVU 214885

Submitted to:
TFPIE 2017

```
type Graph = Table [Vertex]
-- Adjacency list representation of a graph, mapping each vertex to its list of successors.

type Table a = Array Vertex a
-- Table indexed by a contiguous set of vertices.

type Vertex = Int
-- Abstract representation of vertices.
```

Figure 1: Representation of graphs in `Data.Graph`. (From [4].)

(including zero), and $\mathbb{R}_+$ for the set of positive real numbers.

Definition 1. A weighted directed **graph** is a pair $G = (V,E)$ comprising a finite non-empty set V of *vertices* and a finite non-empty set $E \subseteq V \times V$ of *edges*, together with a *weight* function $w\colon E \to \mathbb{R}_+$. A **path** is a non-empty graph $P = (V,E)$ of the form

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

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, in which case E is empty and we have a *singleton* path. □

In the basic Haskell representation (Fig. 1) each vertex is assigned an integer value; this allows vertices to be used as array indices. The graph itself is represented as an array which captures, for each index (i.e. vertex), which vertices can be reached from it directly. Since a path is fully determined by the order in which its vertices are arranged, we represent paths in Haskell by declaring `type Path = [Vertex]`.

For the purposes of this case study, we define BSP as a bi-criteria problem. The first (and leading) criterion is to determine the graph's maximum capacity (or widest path); the second is to determine its shortest path (as a tie breaker for the first criterion).

Definition 2 (The Bottleneck Shortest Path Problem (BSP))**.** Given a graph $G = (V,E)$ and source and target vertices $s \neq t$ in V, we need to determine the shortest maximum capacity path from s to t, together with its capacity and length. Capacity and distance are given by label functions $c : E \to \mathbb{C}$ and $d : E \to \mathbb{D}$, where $\mathbb{C}$ and $\mathbb{D}$ are value sets representing capacities and distances, respectively. For practical purposes we typically take $\mathbb{C} = \mathbb{D} = \mathbb{N}$. □

2.1 Optimality Criteria

By optimality criteria we mean requirements involving minimality and maximality targets. For example, Dijkstra's and Floyd-Roy-Warshall's pathfinding algorithms seek to minimise path length.

Writing $\downarrow$ for the minimization operator over a set of values, we can formalise this idea by saying that the shortest path solution (i.e. length, distance, time or cost) between two nodes is given by applying $\downarrow$ to a set of path lengths—where these lengths are obtained by *adding* together the values (i.e. labels) attached to each edge in the path.

More generally, we need to identify how edge-weights will be "added" in order to calculate the overall cost of traversing a path. Having done so, we can then apply minimization and/or maximization ($\uparrow$) operators to an approprioate set of "path costs" to find those paths which satisfy the constraint(s) in question.

In this document we focus specifically on distance and capacity constraints, so we assume that edge weights are given as capacity/distance pairs. For example, given source and target vertices s and t in a graph $G = (E, v)$, and a path $p \equiv s \xrightarrow{(c_1,d_1)} \dots \xrightarrow{(c_n,d_n)} t$,

1. the distance associated with p is given by $\sum d_i$, and the shortest path is given by applying $\downarrow$ to the st of all such sums. Thus, solving the shortest-path problem involving applying the $(+)$ and $\downarrow$ operations.
2. the capacity associated with a path is the smallest capacity of its edges. Thus, finding the capacity of a path involves computation of $\downarrow c_i$. Finding the path with maximum capacity involves application of $\uparrow$ to the set of all such path capacities.
3. solving BSP therefore requires computations involving $\uparrow$, $\downarrow$ and $(+)$ (as well as the projection operators from (c_i, d_i) to c_i and d_i, respectively).

3 The non-distributivity problem: an example

Suppose the problem is to find, from vertex 1 to vertex 4 from Figure 2 the maximum capacity among all shortest paths, i.e. the BSP. Assume further that labels over the edges are natural numbers having the type `(Capacity,Distance)` where `type Capacity = Int` and `type Distance = Int`.

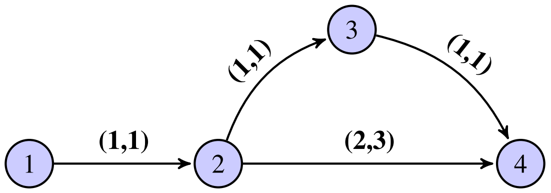


Figure 2: Graph with lack of distributivity for BSP

In this particular case, there are two paths from vertex 1 to vertex 4 but only one is the optimal. Let us call the path p_1 the sequence of vertices $\{1,2,3,4\}$, and p_2 the sequence $\{1,2,4\}$. Assume further that p the optimal path between p_1 and p_2. Since both paths have the same maximum capacity (i.e. 1), is the minimal distance that breaks the tie. So, being p_1 the optimal with $Cost_{p_1} = (1,3)$.

Now, in terms of our pathfinding algebra for BSP, we have that

$$\begin{aligned} Cost_p &= (1,1)\ \texttt{join}\ \Big\langle \big\langle (1,1)\texttt{join}(1,1) \big\rangle\ \texttt{choose}\ (2,3) \Big\rangle \\ &= (1,1)\ (\downarrow,+)\ \Big\langle \big\langle (1,1)\ (\downarrow,+)\ (1,1) \big\rangle\ (\uparrow,\downarrow)\ \ (2,3) \Big\rangle \\ &= (1,1)\ (\downarrow,+)\ \Big\langle (1,2)\ (\uparrow,\downarrow)\ (2,3) \Big\rangle \\ &= (1,1)\ (\downarrow,+)\ (2,3) \\ &= (1,4) \end{aligned}$$

and this is clearly not equal to $Cost_{p_1} = (1,3)$.

This shows that **join** (the *multiplication* operator for BSP) does not distribute over **choose** (the *addition* operator for BSP). The reasons and analysis of why this is happening is beyond the scope of this paper; the interested reader is refered to [1].

4 The solution: a simple data structure to overcome the problem

As we observed from the example in Section 3, some information is left behind in the calculation of the operations *multiplication* and *addition*. Specifically, the pair $(1,2)$, in the inner bracket where `choose` is located, holds the values for the optimal solution. Since the cost of the path can be computed from a considerably large number of pairs (following the size of the graph), we do not know where the optimal pairs are located. Our goal is then to preserve all the necessary pairs that guarantee the optimal solution. That is, a *list* or pairs preserving the pathfinding algebras.

Let us start defining which pairs should be stored in the *list* for BSP. This leads to the following criteria applicable to both operations `join` and `choose`:

- The elements in the list (pairs of type (`Capacity`, `Distance`)) should be sorted descending (due the maximum operator) by `Capacity`
- In case of a tie in the `Capacity`, we break such tie by the second criterion, in this case the minimum operator
- Pairs with equal values `Capacity` and `Distance` are stored once.

More formally, given a tuple of pairs (`Capacity`,`Distance`), denoted as (c,d), we store the result by any of the operators of BSP as $[(c_i,d_i),(c_j,d_j)]$, where $c_i > c_j \ \wedge \ d_i > d_j$. Detailed criteria is given in Sect. 5 where the lexicographical order is taken into account for breaking ties.

Let us put the above into work for the example in Fig. 2, for which we have

$$Cost_{p_1} = (1,3)$$

and

$$\begin{aligned} Cost_p &= (1,1)\ \texttt{join}\ \Big\langle \big\langle (1,1)\texttt{join}(1,1) \big\rangle\ \texttt{choose}\ (2,3) \Big\rangle \\ &= (1,1)\ \texttt{join}\ \Big\langle [(1,2)]\ \texttt{choose}\ (2,3) \Big\rangle \\ &= (1,1)\ \texttt{join}\ [(2,3),(1,2)] \\ &= [(1,3)] \end{aligned}$$

We can notice also that our functions `join` and `choose` should be initially fed with singletons in order to preserve associativity. Therefore the type signatures are:

```
type BSP = [(Capacity,Distance)]
join    :: BSP → BSP → BSP
choose  :: BSP → BSP → BSP
```

5 Case study: the BSP problem, breaking ties

In this section we describe the algebra and operators for computing the BSP, followed by the implementation in Haskell for such operators. Finally, we show why distributivity is preserved.

5.1 Algebra for BSP

We have briefly explained the kind of graph we rely on, although the types or carrier sets can vary. For practical purpose we limit our carrier set to be $\mathbb{N}$, which in Haskell can be materialise as `Int` provided we avoid the negative numbers.

It is important to highlight that the list of pairs should be in order, 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 *distances* 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

$$(c_1, d_1) \succ (c_2, d_2) = c_1 > c_2 \wedge d_1 > d_2$$

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

$$(c_1, d_1) \succ (c_2, d_2) = (c_1 > c_2 \wedge d_1 > d_2) \vee (c_1 < c_2 \wedge d_1 < d_2)$$

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

Recalling the criteria in BSP, the relation $\mathscr{R}_{\mathsf{BSP}}$ is defined as follows:

$$(c_1, d_1)\ \mathscr{R}_{\mathsf{BSP}}\ (c_2, d_2) \equiv c_1 > c_2 \vee (c_1 = c_2 \wedge d_1 \le d_2) \tag{1}$$

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

$$(c_1, d_1) \otimes (c_2, d_2) = (c_1 \downarrow c_2, d_1 + d_2) \tag{2}$$

The addition operator, $\oplus$ is defined as:

$$(c_1, d_1) \oplus (c_2, d_2) \quad = \quad (c_1 \uparrow c_2, d) \tag{3}$$

where (the lexicographic ordering)

$$d = \begin{cases} d_1 & \text{if } c_1 > c_2 \\ d_1 \downarrow d_2 & \text{if } c_1 = c_2 \\ d_2 & \text{otherwise} \end{cases}$$

Also, we are assuming that $\otimes$ has precedence over $\oplus$.

5.2 Implementation of `chBSP`, the addition operator

We start by defining the cases (or constraints) for the applicability of the `chBSP` operator:

c1 Given two lists of pairs, of type `(Capacity,Distance)`, `chBSP` will store all pairs satisfying 3. Three cases arise. Given the pairs `(c_1,d_1)` and `(c_2,d_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 distributivity.

c2 For any finite-length input lists (i.e. `cds1` and `cds2`), function `chBSP` terminates.

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

```
length chBS ≤ length cds1 + length cds2
```

Let us now define the function `chBSP` in Haskell

```
chBSP :: [(Capacity, Distance)] → [(Capacity, Distance)] → [(Capacity, Distance)]
chBSP [] cds2 = cds2
chBSP cds1 [] = cds1
chBSP cdcds1@((c1, d1) : cds1) cdcds2@((c2, d2) : cds2)
    | c1 == c2  = (c1, min d1 d2) : chAux (min d1 d2) cds1   cds2
    | c1 >  c2  = (c1,     d1   ) : chAux      d1     cds1   cdcds2
    | otherwise = (c2,        d2) : chAux         d2  cdcds1 cds2
   where
 chAux :: Distance → [(Capacity,Distance)] → [(Capacity,Distance)] → [(Capacity,Distance)]
 chAux _ []                  []                 = []
 chAux d []                  ((c2, d2) : cds2) = chAux' d c2 d2 [] cds2
 chAux d ((c1, d1) : cds1) []                  = chAux' d c1 d1 cds1 []
 chAux d cdcds1@((c1, d1) : cds1) cdcds2@((c2, d2) : cds2)
  | c1 == c2  = chAux' d c1 (min d1 d2) cds1   cds2
  | c1 >  c2  = chAux' d c1      d1     cls1   cdcds2
  | otherwise = chAux' d c2         d2  cdcds1 cds2

chAux' :: Distance → Capacity → Distance → [(Capacity, Distance)]
          → [(Capacity, Distance)] → [(Capacity, Distance)]
chAux' d c' d' cds1 cds2
  | d > d'    = (c', d') : chAux d' cds1 cds2
  | otherwise =            chAux d  cds1 cds2
```

5.3 Implementation of `jnBSP`, the product operator

Similar to `chBSP`, there are some requirements for `jnBSP` to be fulfilled. 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 distributivity. We avoid any repeated pair to be stored along one traversal in order to minimise the length of the resulting list.

c1 Given two lists of pairs, of type $(Capacity, Distance)$, `jnBSP` will store all pairs satisfying distributivity, in the context of an *multiplication* computation. Two cases arise. Given the pairs (c_1, d_1) and (c_2, d_2) :

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

Same as `chBSP`, 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. cds1 and cds2), function `jnBSP` terminates.

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

```
length jnBPS ≤ length cds1 + length cds2
```

Haskell implementation for `jnBPS` is intentionally omitted due limited space for submission

5.4 `jnBSP` does distribute over `chBSP`

Let us consider the following ordering to show that `jnBSP` does actually distribute over `chBSP`:

$$c_1 < c_2 < c_3 \text{ and } d_1 < d_2 < d_3 \text{ in the pairs } (c_1,d_1),(c_2,d_2),(c_3,d_3)$$

Then, applying the operators we have:

$$\begin{aligned}
& [(c_1,d_1)] \ \texttt{jnBSP} \ \Big([(c_2,d_2)] \ \texttt{chBSP} \ [(c_3,d_3)]\Big) \\
= & \ \{ \text{ applying } \texttt{chBSP}\text{, and because } c_3 > c_2 \wedge d_2 < d_3 \ \} \\
& [(c_1,d_1)] \ \texttt{jnBSP} \ [(c_3,d_3),(c_2,d_2)] \\
= & \ \{ \text{ applying } \texttt{jnBSP} \text{ and since } d_1+d_2 < d_1+d_3 \ \} \\
& [(c_1,d_1+d_2)]
\end{aligned}$$

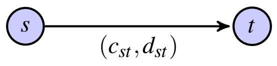
Now, on the right-hand side

$$\begin{aligned}
& \Big([(c_1,d_1)] \ \texttt{jnBSP} \ [(c_2,d_2)]\Big) \ \texttt{chBSP} \ \Big([(c_1,d_1)] \ \texttt{jnBSP} \ [(c_3,d_3)]\Big) \\
= & \ \{ \text{ applying } \texttt{jnBSP}\text{'s } \} \\
& [(c_1,d_1+d_2)]\texttt{chBSP}[(c_1,d_1+d_3)] \\
= & \ \{ \text{ applying } \texttt{chBSP} \text{ and } d_1+d_2 < d_1+d_3 \ \} \\
& [(c_1,d_1+d_2)]
\end{aligned}$$

5.5 Algorithm for BSP

As we stated before, our approach relies 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 capacities set is $\mathbb{C} = \mathbb{N}$ and distances set is $\mathbb{D} = \mathbb{N} \cup \{\infty\}$



and its adjacency matrix

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

where $(0,\infty)$ is the *zero* and $(\infty,0)$ is the *one* (or *unit*) elements of $\mathbb{C} \times \mathbb{D}$, meaning there is no path between two vertices.

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

$$\begin{tikzcd} s \arrow[r, "{(c_{sx},d_{sx})}"'] \arrow[rr, bend left, "{(c_{st},d_{st})}"] & x \arrow[r, "{(c_{xt},d_{xt})}"'] & t \end{tikzcd}$$

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

$$\begin{tikzcd} x \arrow[r, "{[(c_{xt},d_{xt})]}"'] & t \end{tikzcd}$$

And therefore

$$\begin{tikzcd} s \arrow[r, "{\texttt{jnBSP}\ [(c_{sx},d_{sx})]\ sol_x}"'] \arrow[r, bend left, "{\texttt{jnBSP}\ [(c_{st},d_{st})]\ [(\infty,0)]}"] & t \end{tikzcd}$$

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

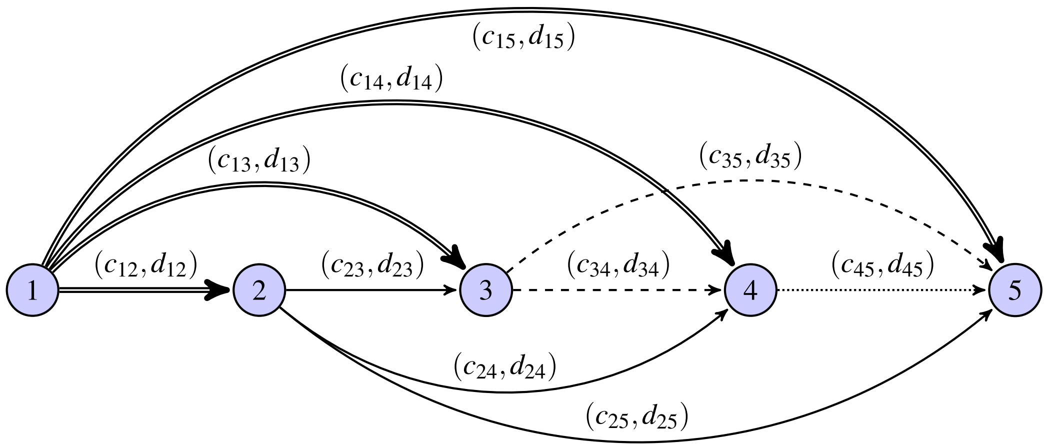


Figure 3: Reusing previous computations

The proposal solution for computing BSP problem from vertex 1 to vertex 5 is:

double arrow	single arrow (sol_2)	dashed arrow (sol_3)
`chBSP`	`chBSP`	`chBSP`
`jnBSP` (c_{15},d_{15}) sol_5	`jnBSP` (c_{25},d_{25}) sol_5	`jnBSP` (c_{35},d_{35}) sol_5
`jnBSP` (c_{14},d_{14}) sol_4	`jnBSP` (c_{24},d_{24}) sol_4	`jnBSP` (c_{34},d_{34}) sol_4
`jnBSP` (c_{13},d_{13}) sol_3	`jnBSP` (c_{23},d_{23}) sol_3	
`jnBSP` (c_{12},d_{12}) sol_2		

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

The above table shows that `jnBSP` is performed as expected whereas each function `chBSP` 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 BSP problem as *solutionSS*.

```
solutionSS :: Int → Int → Int → [(Capacity,Distance)]
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,Distance)]
  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 `chBSP`'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 `Distance`. 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`.

6 Test cases and benchmarking

Throughout this document 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 section is the treatment of path tracking, which turns to be the most noticeable difference in performance between the above evaluations. So far, `jnBSP` and `chBSP` functions cover only the computation of the cost of BSP.

This section comprises different implementation approaches. 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.

6.1 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. So far the functions `jnBSP` and `chBSP` 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
```

6.2 Path tracking

There is a general type signature for both BSP and knapsack problems,

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

where (x,y) represents either the cost `(Capacity,Distance)`. 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]
```

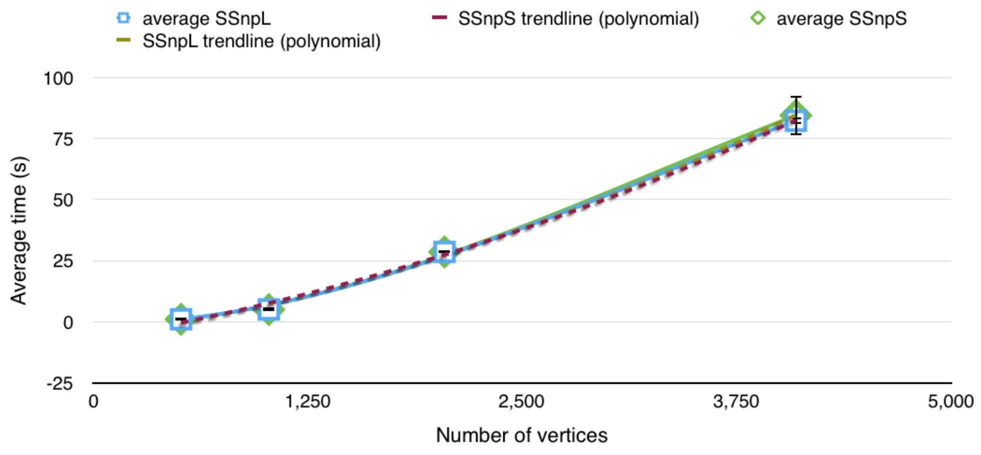
The following are the corresponding data tables and graphics of the single-source approach.

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

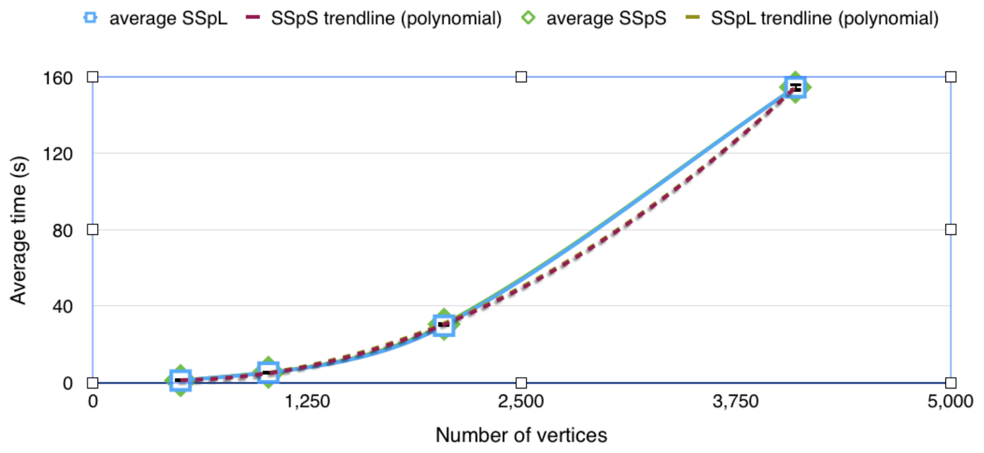


single source MC-SP non path tracking lazy (SSpL)

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



6.3 All pairs approach

We have solved the BSP 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 `chBSP` and `jnBSP`, hold this property.

In this section we will describe and adapt FRW into an algorithm that solves the BSP 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 `chBSP` and `jnBSP` 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 BSP 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 $(\downarrow, +, \infty, 0)$.

6.4 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) = chBSP (g!(x,y)) (jnBSP (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 ]
```

6.5 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.

Code omitted due limited space for submission

where function compute is the core of our interest since it performs the the *addition* and *multiplication* operations in order to solve BSP 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) (chBSP ij (jnBSP ik kj))
```

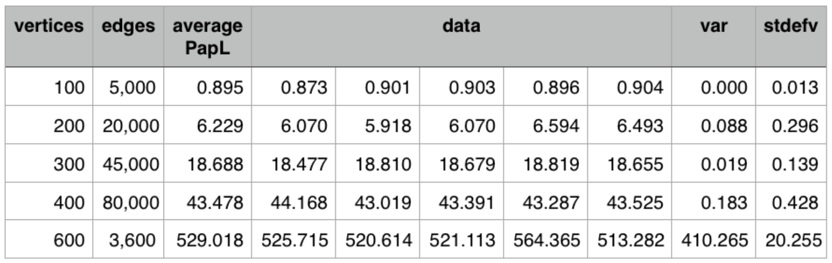
The following chart and tables are calculated with non path-tracking and computing the just the index $(1,v)$ of the square matrix in order to avoid the display of the whole matrix. Due the huge amount of memory taken by the random generation of pairs (labels on the graphs), we profiled specifically 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

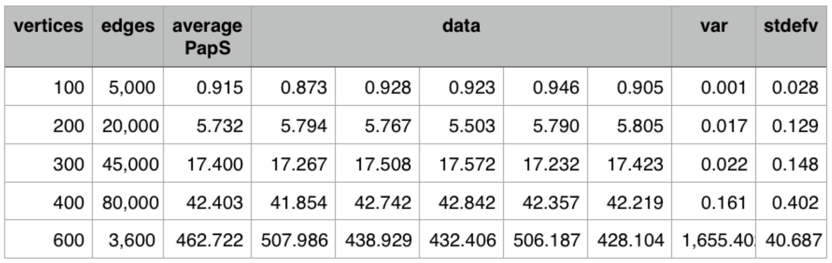
Table 1: Profiling sspt, single source with path-tracking

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

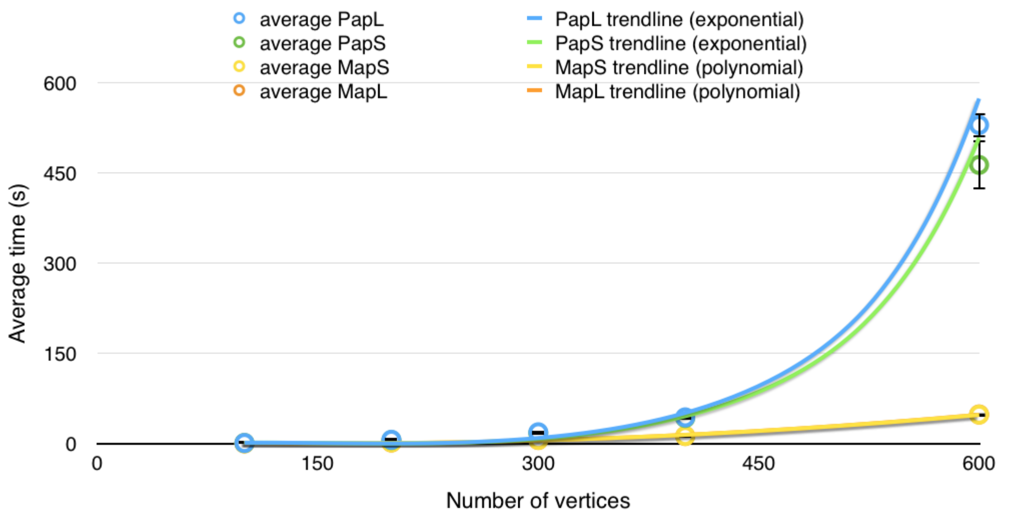


Figure 4: Pure functional and monadic implementations for the all-pairs approach

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, 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.

7 Acknowledgements

JCS would like to thank Henrik Nilsson for his invaluable insights during the early stages of this work.

References

[1] Roland Backhouse (2006): *Regular algebra applied to language problems*. *The Journal of Logic and Algebraic Programming* 66(2), pp. 71–111.

[2] Refael Hassin, Jérôme Monnot & Danny Segev (2010): *The complexity of bottleneck labeled graph problems*. *Algorithmica* 58(2), pp. 245–262.

[3] David J. King & John Launchbury (1995): *Structuring Depth-first Search Algorithms in Haskell*. In: *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, ACM, New York, NY, USA, pp. 344–354, doi:10.1145/199448.199530. Available at `http://www.researchgate.net/publication/2252048_Structuring_Depth-First_Search_Algorithms_in_Haskell`.

[4] Hackage Listing (Accessed 17 May 2017): *Data.Graph*. Available at `http://hackage.haskell.org/package/fgl-5.5.3.1/docs/Data-Graph.html`. Based on [3].