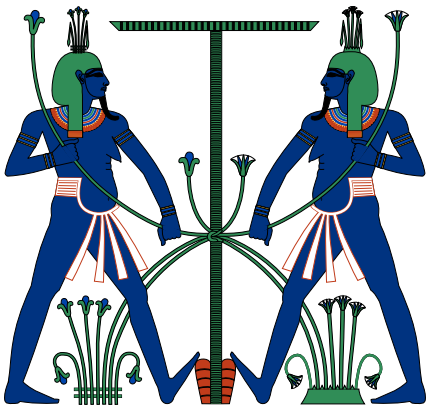
Adam Gleave

Hapi

Fast and scalable cluster scheduling using flow networks



Computer Science Tripos, Part II

St John's College

15th May 2015

Hapi is the ancient Egyptian god of the annual flooding of the Nile. The floods were critical to Egyptian agriculture, depositing fertile soil on the river banks. In line with his importance, Hapi was also the god of both upper and lower Egypt. My project's logo depicts this duality, with the double deity tieing together the symbol of upper Egypt (the lotus, on the left) and lower Egypt (the papyrus, on the right).

Much as the Nile was the lifeblood of Egyptian society, cluster schedulers are critical in modern cloud computing applications. After Hapi has solved the flow problem, the scheduler can deliver work to idle machines, in a periodic pattern reminiscent of the Nile's annual floods. I am proud to say Hapi also embodies the duality of its namesake, bringing together the theoretical and applied in computer science similarly to how the deity brought together the two halves of Egypt.

The logo is by Jeff Dahl, and is inspired by ancient Egyptian depictions. It is available under the GNU Free Documentation License at http://commons.wikimedia.org/wiki/File:Hapy_tying.svg.

Proforma

Name:	**Adam Gleave**
College:	**St John's College**
Project Title:	**Hapi: fast and scalable cluster scheduling using flow networks**
Examination:	**Computer Science Tripos, Part II (June 2015)**
Word Count:	**11,878**
Project Originators:	**Ionel Gog**
Supervisors:	**Ionel Gog & Malte Schwarzkopf**

Original aims of the project

Recent research into cluster scheduling — assigning tasks to machines — has suggested that scheduling can fruitfully be expressed in terms of an optimisation problem over flow networks. The *Hapi* project seeks to develop high-performance algorithms to solve these problems, enabling so-called *flow scheduling* systems to scale to the largest clusters built today. Implementation and performance analysis of an approximation algorithm forms the core success criteria for *Hapi*. Extensions include development of an incremental solver, which would reuse previous optimal solutions to exploit similarities between successive flow networks.

Work completed

Hapi was highly successful, achieving a 14.5× speedup over state of the art implementations and sub-second scheduling latency on a 12,000 machine cluster. I surpassed the success criteria by developing both an approximate and incremental solver. As far as I am aware, both implementations are the first of their kind. The performance of these solvers was thoroughly evaluated on the Firmament flow scheduling platform, including simulation of a cluster from publicly available trace data. In addition, approximation yielded a speedup of between 2–11× on networks arising in other applications, suggesting its use beyond flow scheduling.

Special difficulties

None.

Declaration

I, Adam Gleave of St John's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date: 15th May 2015

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I would like to thank my supervisors, Ionel Gog and Malte Schwarzkopf, for their invaluable advice and encouragement throughout the project.

Chapter 1

Introduction

This dissertation describes the development of novel approaches for solving optimisation problems over flow networks. My interest lies in their use for cluster scheduling, which can be formulated as a minimum-cost flow optimisation problem. I show that my incremental solution technique achieves a 14.5× speedup, compared to state-of-the-art implementations used in prior research. I also explored approximate solution methods. Although I found these to be of limited use in cluster scheduling, approximation achieves speedups between 2–11× in other applications.

1.1 Motivation

Clusters of commodity servers have become the dominant platform for high-throughput computing. Machines in a cluster can collaborate to provide the abstraction of a single "warehouse-scale" computer [3]. Making efficient use of such warehouse-scale computers is a major challenge faced by today's leading web companies, and an active area of distributed systems research.

A *cluster scheduler* chooses which tasks to run on each machine. The choice of scheduler has considerable ramifications on cluster performance and efficiency. Despite this, most approaches rely on *ad hoc* heuristics, which makes it difficult to adapt the scheduler to different cluster designs and application requirements.

The Firmament cluster scheduling platform has been developed to overcome this limitation[1] [60, ch. 5]. In a departure from traditional designs, Firmament represents scheduling as an optimisation problem over a flow network: figure 1.1 shows a simple example.

The arc capacities within the flow network restrict the possible schedules, e.g. limiting the number of tasks which can run on each machine. Cost values specify preferences

[1] Firmament was developed by Malte Schwarzkopf and Ionel Gog, as part of the Cambridge Systems at Scale initiative (http://camsas.org).

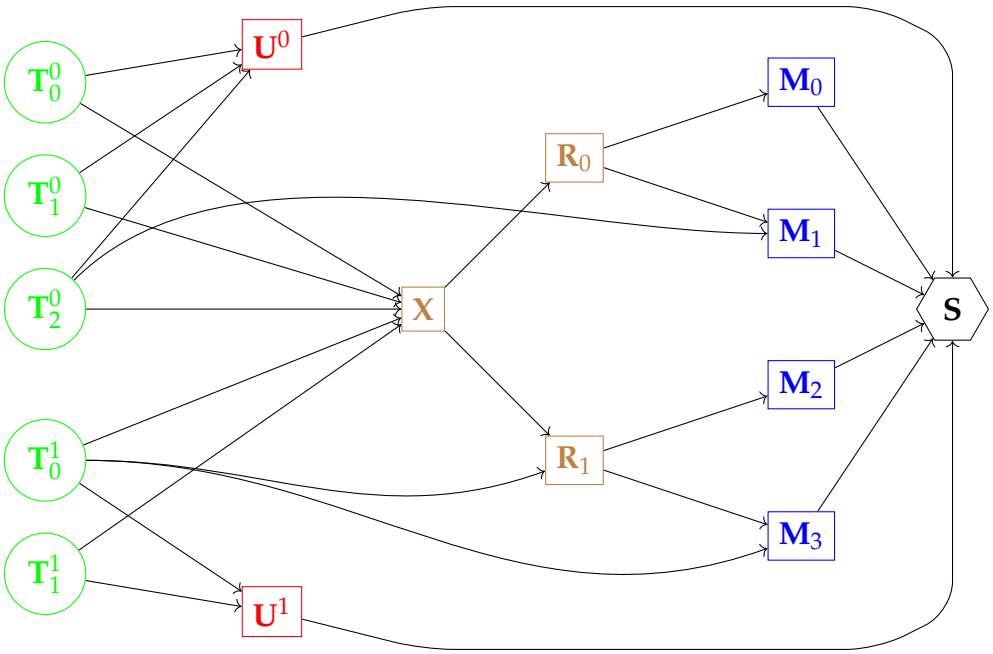


Figure 1.1: A small scheduling flow network. Flow drains left-to-right from ***tasks*** $\mathbf{T}_i^j$ *to sink* **S**. *If the flow passes through a* ***machine*** $\mathbf{M}_l$, *then the task is scheduled there. Alternatively, it may flow through an unscheduled aggregator* $\mathbf{U}_j$, *in which case the task is left unscheduled. Capacities and costs (not shown) determine the scheduling policy.*

between possible assignments. Solving the minimum-cost flow problem on this network yields a schedule that is optimal for the given costs.

This "flow scheduling" approach was originally pioneered by the Quincy scheduler, developed at Microsoft Research [43]. Quincy aims for *data locality*: placing tasks close to where their data is stored. Costs in the network represent bandwidth usage, with the optimal schedule being the one which minimises traffic. With Quincy, cluster throughput increases by 40%, demonstrating the efficacy of this approach.

However, solving the minimum-cost flow problem is extremely computationally expensive at scale. This can be a problem: many applications are sensitive to scheduling latency. Quincy was originally evaluated on a cluster of 243 machines, where the scheduling latency was under 10 ms. On a simulated cluster of 2,500 machines, latency was about a second. Today's warehouse-scale computers consist of tens or hundreds of thousands of servers, and continue to grow.

Throwing more hardware at the problem does not help: flow algorithms have limited parallelism, and the scalar performance of processors has mostly peaked. The only way to get faster is by algorithmic improvements: this is the focus of my project.

In the rest of the dissertation, I explore approaches to improve the performance of algorithms on networks produced by flow schedulers. My goal is to enable these systems to scale to the largest warehouse-scale computers. Moreover, reducing the scheduling latency will allow flow schedulers to be used for applications which require rapid decisions.

1.2 Challenges

Research into the minimum-cost flow problem has been ongoing for over 60 years. There is consequently considerable prior work, outlined in §1.3. I had to assimilate this large body of existing material before I could attempt to improve upon it.

Given that many seasoned researchers have spent their careers working on this problem, realising a significant performance improvement appeared difficult. Not only has there been considerable work to develop efficient algorithms, the reference implementations for these algorithms have been optimised extensively.

While the task seemed daunting, the reward more than justified the risk: success could enable a new generation of schedulers, able to address the challenges facing today's major technology companies.

1.3 Related work

Research into flow networks has been ongoing since the 1940s, driven by their numerous practical applications. The area remains active, with new algorithms and implementation techniques continuing to be devised.

Study of the area began with the transportation problem, a special case of the minimum-cost flow problem. The problem was first formulated by Kantorovich in 1939 [46], although his work did not receive recognition outside Russia until sometime later. Study of the problem in the Western world began with Hitchcock in 1941 [38]. Koopmans followed in 1949, demonstrating the theory applied in an economic context [50]. Kantorovich and Koopmans later received a Nobel prize for their research[2].

Flow problems are a special case of *linear programming* problems. Study of linear programming was originally motivated by flow networks: indeed, the first statement of the general linear programming problem is due to Kantorovich [46]. It became established as a distinct field with the publication in 1949 of Dantzig's seminal work on the now well-known simplex algorithm [14]. One of the earliest applications of this method was to flow networks, with Dantzig specialising the simplex algorithm to the transportation problem in 1951 [15].

Development of flow algorithms continues to the present day. While it is impractical for me to discuss all published methods, I survey contemporary flow algorithms in the remainder of this section.

[2]Hitchcock missed out on the prize as it was awarded in 1975, after he had passed away.

1.3.1 Network simplex

The network simplex algorithm is the oldest flow solution method still in use today. Although the generic network simplex algorithm is not guaranteed to run in polynomial time, many variants have been devised with a polynomial bound [65, 35]. Furthermore, there has been considerable work to develop efficient implementations [52, 36].

1.3.2 Successive shortest path

This algorithm was invented independently by several authors [44, 41, 11]. Edmonds and Karp [19] and Tomizawa [66] independently suggested a technique to maintain non-negative arc costs during the algorithm; this allows for more efficient shortest path computations, considerably improving its performance. The variant given by Edmonds and Karp is notable for being the first (weakly) polynomial time algorithm[3].

Nevertheless, these traditional versions of successive shortest path are inferior to more modern algorithms, such as cost scaling (see §1.3.4). Bertsekas and Tseng developed a more modern variant, *relaxation*, which is competitive with contemporary algorithms [67, 6, 7]. In fact, it is the fastest solver on some networks [47]. Although relaxation is not normally described as a successive shortest path algorithm, it can be shown to performs the same basic operations [1, §9.10], a fact exploited by my incremental solver (see §3.7.3).

1.3.3 Cycle cancelling

Originally proposed by Klein [48], cycle cancelling was initially an exponential time algorithm, but variants have improved on this by carefully choosing which cycles to cancel.

An important special-case is the strongly polynomial minimum-mean cycle cancelling algorithm, devised by Goldberg and Tarjan [33]. Although not the first polynomial time algorithm, it is one of the simplest. Research into variants of cycle cancelling has continued into recent years, with Sokkalingam *et al.* publishing an algorithm with an improved asymptotic bound in 2000 [62].

1.3.4 Cost scaling

The most modern class of minimum-cost flow algorithms, cost scaling, was first proposed in the 1980s by Rock [58] and, independently, Bland and Jensen [8].

[3] In a weakly polynomial algorithm, the maximum cost and capacity of arcs may feature in the polynomial bound. By contrast, for a (strongly) polynomial algorithm, the bound is a function only of the dimensions of the problem: the number of vertices and arcs.

Goldberg and Tarjan developed an improved method in 1990 [34], using the concept of ϵ-optimality due (independently) to Bertsekas [4] and Tardos [63]. This can be viewed as a generalisation of their well-known and highly successful push-relabel algorithm for the maximum flow problem [32].

The algorithm by Goldberg and Tarjan offers some of the best theoretical and practical performance. Moreover, Goldberg *et al.* spent considerable time in the late 1990s developing efficient implementations of this algorithm, including devising heuristics to guide the solver [29, 10].

Work has continued up until the present day. In 2008, Goldberg published an improved version of his push-relabel algorithm for the maximum flow problem [30]. Király and Kovács demonstrated in 2012 that this approach can also be incorporated into the minimum-cost flow algorithm, with similar performance gains [47].

1.3.5 Comparative evaluation

The goal of my project is to develop a solution method that outperforms state-of-the-art solvers, when applied to the problem of flow scheduling. However, this raises the question of how algorithms can be compared to identify the current state of the art.

Asymptotic complexity is sometimes misleading: flow algorithms usually considerably outperform their worst-case time complexity. The relaxation algorithm described in §1.3.2 is the most extreme example of this: in the worst case it is exponential, but in practice it outperforms many strongly polynomial algorithms.

Consequently, comparison of flow algorithms is typically done by empirical benchmarks. The most recent study in this area is due to Király and Kovács, who tested all the algorithms described above [47, 51].

In addition, reference implementations of these algorithms are available. These include CS2 for the cost-scaling algorithm [40] and RELAX-IV for the relaxation algorithm [24]. I compare my solver against these state-of-the-art implementations in my evaluation.

Chapter 2

Preparation

2.1 Cluster scheduling

Much as an operating system scheduler maps threads to processor cores, a cluster scheduler maps tasks to machines. Better scheduling techniques can considerably improve performance: for example, throughput increased by 40% under Quincy [43]. Due to their importance, cluster schedulers are an active area of research.

I start by outlining desirable properties of cluster schedulers. Following this, I summarise the prevailing approaches to scheduling. I conclude by discussing the principles underlying Firmament, showing how it improves upon competing approaches with its use of flow scheduling.

2.1.1 Design goals

To achieve optimal performance, the scheduling policy employed must be *targeted* to the requirements of the workload. The scheduler may need to analyse a large amount of data, such as the resource requirements of each task and the utilisation of each machine, to *effectively* implement a particular policy. All decisions should be made as *fast* as possible, to support interactive applications. Furthermore, the system must be able to *scale* to the size of modern clusters. These goals are summarised in table 2.1.

Goal	Description
Targeted	Application requirements are reflected in the scheduler policy.
Effective	The schedules produced are optimal given the policy objectives.
Fast and scalable	Scheduling latency remains low even for large clusters.

Table 2.1: Cluster scheduler design goals

System	Targeted	Effective	Fast and scalable
Traditional schedulers			
Hadoop Fair Scheduler [69]	◯	✓	◯
Apache Mesos [37]	✓	◯	◯
Sparrow [53]	✗	◯	✓
Flow-based schedulers			
Quincy [43]	✗	✓	✗
Firmament [60, ch. 5]	✓	✓	✗
Firmament using Hapi	✓	✓	✓

Table 2.2: Scheduler feature matrix showing full ✓, partial ◯ or no ✗ support.

2.1.2 Prevailing approaches

Commensurate with their importance, a variety of cluster scheduling designs have been investigated, summarised in table 2.2. Traditionally, schedulers have been *monolithic*, designed for a single workload. For example, the Hadoop Fair Scheduler only manages MapReduce jobs [69].

Although monolithic schedulers may perform well on their intended workload, they cannot be targeted to different applications. Newer schedulers such as Mesos [37] adopt a *two-level* design [61, §3.3]: a centralised allocator dynamically partitions resources in the cluster between *scheduling frameworks*, each of which implement a policy appropriate to the applications they manage. However, as each framework sees only a subset of the resources in the cluster, they are not as effective as monolithic schedulers.

To support interactive applications on large-scale clusters, there has been research into *distributed* scheduling. These methods can be very fast. For example, the Sparrow scheduler achieves sub-100 ms scheduling latency, employing a sampling mechanism to avoid maintaining any centralised state [53, fig. 10]. However, the lack of global state imposes limits on the effectiveness of distributed schedulers.

2.1.3 Firmament and flow scheduling

Flow scheduling models scheduling as an optimisation problem over flow networks (see §2.2.2). The effectiveness of this approach was demonstrated in the pioneering Quincy system, which delivered a 40% increase in throughput and reduced data transfer by 3.9× [43].

Firmament generalises Quincy, allowing a variety of scheduling policies to be specified. Like with two-level schedulers, the policy can be chosen on a per-

application basis. However, flow scheduling does not partition the cluster: solving the optimisation problem globally allows better scheduling decisions to be made.

Flow algorithms are notoriously difficult to parallelise, due to the global nature of the problem. Accordingly, the flow solver quickly becomes a bottleneck, leading some researchers to claim flow scheduling systems cannot scale [9, §6]. In this dissertation, I design and implement new flow algorithms which allow these systems to compete with the fastest distributed schedulers.

2.2 Flow networks

I first define flow networks and their associated optimisation problems, before discussing the details of scheduling using flow networks. Following this, I summarise properties of flow networks useful for the analysis and design of flow algorithms.

2.2.1 Introduction

Definitions and notation

A *flow network* is a weakly connected[1] directed graph $G = (V, E)$.

Each arc $(i, j) \in E$ has an associated *capacity* (or *upper bound*[2]) u_{ij}, and a *cost* c_{ij}.

Each vertex $i \in V$ has an associated supply/demand b_i. Vertex i is said to be a *supply vertex* if $b_i > 0$, a *demand vertex* if $b_i < 0$ and a *transshipment vertex* if $b(i) = 0$.

The problems we will consider involve finding a solution vector $\mathbf{x}$, specifying the flow x_{ij} at each arc $(i, j) \in E$. A solution $\mathbf{x}$ is *feasible*, and we say it is a *flow*, if it satisfies capacity constraints at every arc $(i, j) \in E$:

$$0 \leq x_{ij} \leq u_{ij}, \tag{2.1}$$

and mass balance constraints at each vertex $i \in V$:

$$\sum_{j\,:\,(i,j)\in E} x_{ij} - \sum_{j\,:\,(j,i)\in E} x_{ji} = b_i. \tag{2.2}$$

That is, the net flow out of the vertex is exactly equal to the supply at that vertex (which can be negative, in the case of demand vertices).

[1] A directed graph is weakly connected if the undirected graph formed by replacing all directed arcs with undirected edges is itself connected.

[2] Some authors include a lower bound l_{ij}, but this does not feature in flow scheduling. Moreover, any network featuring lower bounds can be transformed into an equivalent one without [1, p. 39].

Assumptions

The following two assumptions hold for all flow scheduling networks, and will be used in the design and analysis of algorithms in the next chapter.

Assumption 2.2.1 (Integrality)**.** All quantities defining the flow network take on only integer values.

Assumption 2.2.2 (Non-negative arc costs)**.** For all $(i, j) \in E$, $c_{ij} \geq 0$.

The minimum-cost flow problem

The well-known maximum flow problem involves finding a solution vector $\mathbf{x}$ subject to constraints eq. (2.1) and eq. (2.2), i.e. finding a feasible flow.

Flow scheduling requires solving a generalization of this, known as the minimum-cost flow (MCF) problem. Formally, it is:

$$\text{minimise } s(\mathbf{x}) = \sum_{(i,j)\in E} c_{ij}x_{ij} \tag{2.3}$$

subject to the constraint that $\mathbf{x}$ must be a feasible flow.

Note, in general, that the MCF and maximum flow problem may be infeasible. For example, the network might be imbalanced: $\sum_{i\in V} b_i \neq 0$. All networks produced by Firmament are guaranteed to be solvable, however, so I will tend not to discuss this case[3].

2.2.2 Scheduling with flows

I will now show how cluster scheduling can be formulated in terms of the MCF problem, using Quincy as an example. For a detailed description of Quincy, see Isard *et al.* [43]. Firmament generalises Quincy, supporting different scheduling policies via a *cost model* framework. I describe how I ported Quincy to this framework in appendix B. Schwarzkopf [60, ch. 5] provides several examples of other scheduling policies represented in this manner.

Network structure

Each machine in the cluster is represented by a vertex, $\mathbf{M}_l$. The topology of the network follows that of the cluster: each rack is represented by a rack aggregator

[3]For robustness, all algorithms implemented as part of this project detect if a network is infeasible.

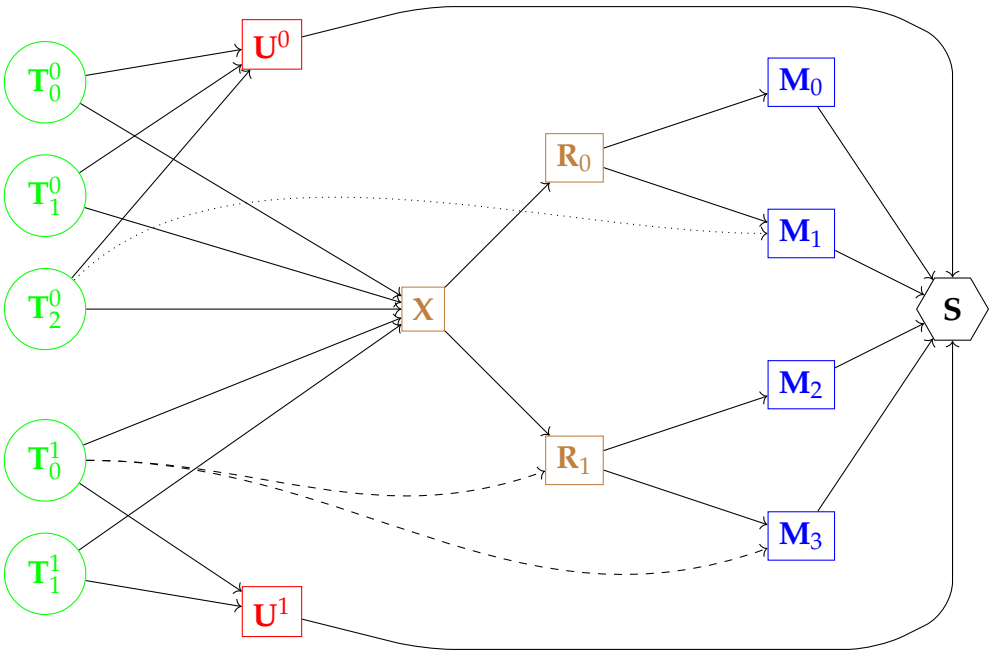


Figure 2.1: A small Quincy scheduling flow network. Vertices are ***tasks*** $\mathbf{T}_i^j$, ***unscheduled aggregators*** $\mathbf{U}^j$, ***cluster and rack aggregators*** $\mathbf{X}$ *and* $\mathbf{R}_l$, ***machines*** $\mathbf{M}_l$, *and the sink S. Task* $\mathbf{T}_2^0$ *is already scheduled on machine* $\mathbf{M}_1$*: the dotted line represents the* ***running arc****. All other tasks are unscheduled. Task* $\mathbf{T}_0^1$ *has two* ***preference arcs*** *to machine* M_3 *and rack aggregator* R_1*, represented by dashed lines.*

vertex $\mathbf{R}_j$, with arcs to machines in that rack. There is also a single cluster aggregator vertex $\mathbf{X}$, with arcs to every rack aggregator.

The scheduler manages *jobs*, which consist of a number of *tasks*. Each job j has an *unscheduled aggregator* $\mathbf{U}_j$. Every task i in the job is represented by a task vertex $\mathbf{T}_i^j$, with unit supply. Tasks must drain into the *sink* vertex $\mathbf{S}$, passing through a machine or unscheduled aggregator along the way[4]. Supply from tasks can only drain into the sink vertex via machines and unscheduled aggregators. If the flow from $\mathbf{T}_i^j$ passes through machine $\mathbf{M}_l$, the task is scheduled there; if it passes through the unscheduled aggregator $\mathbf{U}_j$, the task is left unscheduled.

All tasks have an arc to cluster aggregator $\mathbf{X}$. In addition, a task may have *preference arcs* to racks or machines which store a substantial proportion of the input data. Tasks that are already executing also have a *running arc* to the machine they are currently scheduled on.

Capacities and costs

Arc capacities are used to *restrict* the possible scheduling assignments. In Quincy, the capacities are chosen to provide a lower and upper bound on the number of tasks

[4]In fact, unscheduled aggregators may themselves have demand, consuming the flow before it reaches the sink: see appendix B.2.

that may be scheduled for each job, guaranteeing fairness (see appendix B.2).

The costs associated with arcs specify *preferences* between possible scheduling assignments. Quincy seeks to achieve data locality, setting arc costs proportional to the network resources consumed. For arcs from a task to a machine, this can be computed exactly. When the arc is to a rack (or cluster) aggregator, a conservative approximation is used, assuming the task runs on the *worst* machine in that rack (or the cluster as a whole).

Solving the minimum-cost flow problem on the resulting network finds a scheduling assignment which maximises data locality, subject to fairness constraints.

Properties of these networks

I state below some properties which hold for all scheduling flow networks. These will be useful in the next chapter, when analysing the complexity of algorithms.

Lemma 2.2.1 (Number of vertices in flow scheduling networks)**.** *Let n denote the number of vertices in the network. Then $n = \Theta(\#\,\textit{machines} + \#\,\textit{tasks})$.*

Proof. See appendix B.4. □

Lemma 2.2.2 (Number of arcs in flow scheduling networks)**.** *Let m denote the number of arcs in the network. Then $m = O(n)$. That is, the network is* sparse*.*

Proof. See appendix B.4. □

Lemma 2.2.3 (Size of supply in flow scheduling networks)**.** *The largest supply in the network is a unit supply.*

Proof. The only vertices in the network with any supply are the task vertices, $\mathbf{T}_i^j$. These by definition have unit supply. □

Remark 2.2.1. Most vertices in the network are transshipment vertices, with no demand nor supply. The demand vertices in the network are the sink vertex $\mathbf{S}$ and unscheduled aggregators, $\mathbf{U}^j$, which may have a greater than unit demand.

2.2.3 Key concepts in flow algorithms

In the preceding sections, I have formalised the notion of a flow network and outlined how cluster scheduling can be formulated in terms of the MCF problem. This section introduces some further definitions and properties of flow networks, necessary for the implementation and analysis of algorithms in the next chapter.

Pseudoflows

A *pseudoflow* is a vector $\mathbf{x}$ which satisfies the capacity constraints of eq. (2.1), but which need not satisfy the mass balance constraints of eq. (2.2). While all flow algorithms must return a feasible solution, many operate on pseudoflows in intermediate stages.

The *excess* at a vertex $i \in V$ is defined to be:

$$e_i = b_i + \sum_{\{\, j \mid (j,i) \in E \,\}} x_{ji} - \sum_{\{\, j \mid (i,j) \in E \,\}} x_{ij}. \tag{2.4}$$

Vertex i is said to be an *excess vertex* if $e_i > 0$, and a *deficit vertex* if $e_i < 0$ (with deficit $-e_i$). If $e_i = 0$, i is said to be *balanced*. Note the mass balance constraints of eq. (2.2) hold if and only if all vertices are balanced.

Residual networks

The notion of *residual networks* is used in many flow algorithms. The residual network of G is defined with respect to a (pseudo)flow $\mathbf{x}$, and is denoted by $G_{\mathbf{x}}$. Informally, it represents the actions an algorithm can take to modify the (pseudo)flow $\mathbf{x}$. In the case where $\mathbf{x} = \mathbf{0}$, $G_{\mathbf{0}} = G$ holds.

Formally, define $G_{\mathbf{x}} = (V, E_{\mathbf{x}})$ as a directed graph where:

$$E_{\mathbf{x}} = \{\, (i,j) \in V^2 \mid (i,j) \in E \wedge x_{ij} < u_{ij} \,\} \cup \{\, (j,i) \in V^2 \mid (i,j) \in E \wedge x_{ij} > 0 \,\}. \tag{2.5}$$

The former set contains *forward arcs*, present in the original flow network. Arcs (i,j) which are *saturated*, i.e. $x_{ij} = u_{ij}$, are omitted from the residual network: they do not allow for any additional flow to be pushed. The second set consists of *backward arcs*, the reverse of arcs in the original network. Only arcs with positive flow in the original network have a corresponding backwards arc. Pushing flow along the backward arc corresponds to cancelling flow on the forward arc.

The *residual capacity* of an arc $(i,j) \in E_{\mathbf{x}}$ is defined to be:

$$r_{ij} = \begin{cases} u_{ij} - x_{ij} & \text{, if } (i,j) \text{ is a forward arc;} \\ x_{ij} & \text{, if } (i,j) \text{ is a reverse arc.} \end{cases} \tag{2.6}$$

The cost of forward arcs is the same as in the original network, whereas the cost of a reverse arc (j,i) is defined to be $-c_{ij}$.

Duality and reduced cost

Every linear programming problem may be viewed from two perspectives, as either a *primal* or *dual* problem. Solutions to the dual problem provide a lower bound on

the objective value of the primal problem. The primal version of the MCF problem was stated previously in §2.2.1.

To formulate the dual problem, it is necessary to associate with each vertex $i \in V$ a *dual variable* π_i, called the *vertex potential*. The *reduced cost* of an arc $(i,j) \in E$ with respect to a potential vector $\boldsymbol{\pi}$ is defined as:

$$c_{ij}^{\pi} = c_{ij} - \pi_i + \pi_j. \tag{2.7}$$

The dual problem can now be stated as:

$$\text{maximise } w(\boldsymbol{\pi}) = \sum_{i \in V} b_i \pi_i - \sum_{(i,j) \in E} \max\left(0, -c_{ij}^{\pi}\right) u_{ij} \tag{2.8}$$

with no constraints on $\boldsymbol{\pi}$.

Many flow algorithms seek to solve the dual problem, as this may be computationally more efficient. Others try to enjoy the best of both worlds, operating on both the primal and dual versions of the problem.

Optimality conditions

Below, I give conditions for a solution vector $\mathbf{x}$ to be optimal. These may suggest algorithms for solving the problem, and are needed for correctness proofs in the subsequent chapter.

Theorem 2.2.4 (Negative cycle optimality conditions)**.** *Let $\mathbf{x}$ be a (feasible) flow. It is an optimal solution to the minimum-cost flow problem if and only if the residual network $G_{\mathbf{x}}$ has no negative cost (directed) cycle.*

Proof. See Ahuja *et al.* [1, p. 307]. □

Theorem 2.2.5 (Reduced cost optimality conditions)**.** *Let $\mathbf{x}$ be a (feasible) flow. It is an optimal solution to the minimum-cost flow problem if and only if there exists a vertex potential vector $\boldsymbol{\pi}$ such that the reduced cost of each arc in the residual network $G_{\mathbf{x}}$ is non-negative:*

$$\forall (i,j) \in E_{\mathbf{x}} \cdot c_{ij}^{\pi} \geq 0 \tag{2.9}$$

Proof. See Ahuja *et al.* [1, p. 309]. □

Theorem 2.2.6 (Complementary slackness optimality conditions)**.** *Let $\mathbf{x}$ be a (feasible) flow. It is an optimal solution to the minimum-cost flow problem if and only if there exists a vertex potential vector $\boldsymbol{\pi}$ such that for every arc $(i,j) \in E$:*

$$\text{if } c_{ij}^{\pi} > 0, \text{ then } x_{ij} = 0; \tag{2.10a}$$

$$\text{if } c_{ij}^{\pi} < 0, \text{ then } x_{ij} = u_{ij}; \tag{2.10b}$$

$$\text{if } c_{ij}^{\pi} = 0, \text{ then } 0 \leq x_{ij} \leq u_{ij}. \tag{2.10c}$$

Proof. The result follows immediately from expanding out eq. (2.9), applying the definition of a residual network and performing a case analysis. A detailed proof is given by Ahuja *et al.* [1, p. 310]. □

Notation for complexity analysis

Here, I introduce notation that will be useful for stating the asymptotic complexity of algorithms. Let $n = |V|$ and $m = |E|$ denote the number of vertices and arcs respectively. Let U denote the largest vertex supply/demand, or arc capacity:

$$U = \max\left(\max\left\{|b_i| \,:\, i \in V\right\}, \max\left\{u_{ij} \,:\, (i,j) \in E\right\}\right) \tag{2.11}$$

and let C denote the largest arc cost:

$$C = \max\left\{c_{ij} \,:\, (i,j) \in E\right\}. \tag{2.12}$$

2.3 Project management

Before I could start to develop improved flow algorithms, it was necessary to understand the considerable body of prior work summarised in §1.3 and §2.2. Consequently, reviewing the existing literature formed a key part of the preparatory stage of this project.

Having mastered the background material, I began to refine the initial project proposal (see appendix E) into a concrete plan. To start with, I carefully reviewed the project's requirements, described in the next section. In the final two sections, I describe how I selected a development model and devised a testing strategy appropriate for the project's requirements.

2.3.1 Requirements analysis

The main goals for the project are listed in table 2.3, broadly following the success criteria and extensions suggested in the original project proposal.

Core success criteria were assigned a high priority: work on extensions only began once all success criteria had been met (see §2.4). Most of the risk was concentrated in the approximate solver. This approach had not previously been attempted, so it was difficult to predict how it would perform.

The extensions I deemd more risky. Although the incremental solution approach turned out to be highly successful (see §4.5), at the outset of the project it was unclear whether it was even possible to build such a solver. All extensions were rated medium or high difficulty, because of the considerable implementation work required.

#	Deliverable	Priority	Risk	Difficulty
	Success criteria			
S1	Implement standard flow algorithms	High	Low	High
S2	Design an approximate solver	High	Med.	Low
S3	Integrate system with Firmament	High	Low	Low
S4	Develop a benchmark suite	High	Low	Med.
	Optional extensions			
E1	Design an incremental solver	Med.	High	High
E2	Build cluster simulator	Med.	Low	Med.
E3	Optimise algorithm implementations	Low	Med.	High

Table 2.3: Project deliverables

2.3.2 Development life cycle

A software development life cycle or process provides structure to a project, dividing work into distinct phases to simplify management. This project has clearly defined requirements, suggesting a waterfall model. However, there was considerable uncertainty associated with the design of some deliverables: methods such as incremental development or evolutionary prototyping might be more appropriate in these cases. I adopted the *spiral model*, a risk-driven life cycle generator, which is able to accommodate such varying requirements.

The spiral model indicates that development of deliverables S1 and S3 in table 2.3 should follow the *waterfall* model: they are low risk, with precisely known requirements. By contrast, the test suite — consisting of deliverables S4 and E2 — used an *incremental* development approach, with new tests added after other deliverables were completed.

Deliverables S2, E1 and E3 were all rated as medium or high risk. Their requirements were fairly well known, but there was significant uncertainty as to the appropriate design. *Evolutionary prototyping* was followed in accordance with the spiral model, first constructing a robust prototype which is then repeatedly refined. Building a prototype early was invaluable for risk management: for example, it allowed the viability of the incremental solver (extension E1) to be confirmed before committing significant development time.

2.3.3 Testing strategy

I developed unit tests for each class. Where possible, I used the GTest framework (see appendix A). However, testing of flow algorithms requires large external datasets: a Python test harness was developed to automate such tests. I also performed integration tests with Firmament (see §4.1).

Extensive performance evaluation was conducted. Anticipating this need, success criteria S4 and extension E2 deliver tools for benchmarking. Further details are provided in §3.8, §4.2 and appendix D.2.

2.4 Implementation schedule

For a project of this scope, it is essential to plan the work before commencing development. An implementation schedule was devised, following the high-level strategy described in the preceding section, dividing the project into the following phases:

1. **Approximate solver** – develop an algorithm to find approximate solutions to the MCF problem. Provided success criteria S1, S2 and S3; see §3.6.
2. **Performance evaluation** – run experiments to determine the speed-up offered by the system. This required development of a benchmark suite to facilitate testing, and provided deliverables S4 and E2; see §4.
3. **Incremental solver** – develop an algorithm solving the minimum-cost flow problem on sequences of related flow networks. The method reuses solutions to previous networks in the sequence to improve performance. Provided extension E1; see §3.7.
4. **Optimisations** – previous phases focused on improvements in *algorithmic* techniques. In practice, performance is heavily dependent on the *implementation* of the algorithm. Consequently, optimisations were necessary for the project to be competitive with existing solvers. Delivers extension E3; see §4.3.

All core success criteria were satisfied after the completion of the first two phases. This ensured the key goals of the project were met before work on extensions commenced.

2.5 Choice of tools

2.5.1 Languages

C++

I implemented the minimum-cost flow algorithms in C++. The performance of C++ is on par with low-level languages like C, with a number of excellent optimising compilers[5]. At the same time, it adds support for object oriented programming and other features typical of high-level languages.

[5]Clang and GCC were used in this project.

Python and shell

I used the Python scripting language for the development of test harnesses (see §4.1 and appendix D.2), and for generation of many of the figures appearing in §4. As a high-level language, Python aided productivity, although it was not used for any performance critical code. Shell scripting was used for particularly simple tasks, that involved plumbing together UNIX utilities. Python was favoured for anything more complicated, however, as it allows for a more structured programming style.

2.5.2 Libraries

Hapi uses two libraries developed at Google — glog and gtest — for logging and unit tests. I also extensively used the Boost family of libraries. Further details are provided in appendix A.

2.5.3 Flow scheduling

The Firmament cluster scheduling platform is used for testing this project. The only other flow scheduling system in existence is Quincy, which pioneered the approach; however, Quincy is not publicly available.

2.5.4 Revision control and backup strategy

The Git distributed revision control system was used to manage the project source code, along with the LaTeX code for this dissertation. Each Git working directory is a clone of the repository, complete with versioning history. The benchmark suite makes extensive use of this distributed nature of Git (see appendix D.2).

To simplify testing, commits were automatically pushed to test machines in the SRG cluster, via a Git hook. Furthermore, changes were regularly pushed to GitHub, a popular repository hosting service.

Given the distributed nature of Git, having multiple working copies as described above automatically provides data replication. However, it would be possible for all copies of the repository to become corrupted due to user error, so this could not be relied upon as a backup strategy. Moreover, some files were not stored in Git[6].

To protect against these threats, nightly backups were taken using a cron job. This took snapshots of the Git repository using `git-bundle`. Copies of other project files were made using `rdiff-backup`, which maintains a history of changes without the overhead of a full version control system like Git.

[6]For example, test data files were often large, so had to be stored outside of Git.

Copies of the resulting snapshots were stored on the university MCS and the Student Run Computing Facility (SRCF) in Cambridge. Additional off-site backups were made to Copy, a cloud storage service based in the US.

2.6 Summary

In this chapter, I have outlined the work undertaken prior to starting implementation for this project. In §2.1, I summarised the state of the art in cluster scheduling, and showed how flow scheduling can improve upon it. Next, I provided some elementary background on flow networks, and outlined how scheduling can be expressed in terms of a flow problem.

After having surveyed the research area, I turned to more practical considerations. I started by conducting a requirements analysis, identifying an appropriate software development life cycle taking into account the risk profile of the requirements. Next, I developed a concrete plan of implementation using the aforementioned techniques. Finally, I identified the tools I would use during implementation. In the next chapter, I describe the algorithms developed in this project.

Chapter 3

Implementation

3.1 Introduction

Considerable research effort has been expended over the past 60 years to produce efficient flow algorithms, as discussed in §1.2 and §1.3. My approach has been to embrace prior work rather than attempting to supplant it, adapting existing algorithms to improve performance for the special case of flow scheduling.

Two strategies seemed particularly promising. *Approximate* solutions to the MCF problem can be found, described in §3.6. Existing algorithms always seek to find optimal solutions: for flow scheduling, we may be happy to trade optimality for reduced scheduling latency. Alternatively, the problem can be solved *incrementally*, outlined in §3.7. The network remains largely unchanged between runs of the scheduler: significant performance improvements can be realised by reoptimising from the last optimal solution found.

These strategies are not solution methods in themselves. Rather, they suggest modifications that can be made to existing minimum-cost flow algorithms. Table 3.1 lists the algorithms I implemented, alongside the strategy they are used in.

I analysed the complexity of each algorithm, both in the general case and for flow scheduling networks. My results are summarised in table 3.2, and show that the algorithmic complexity often improves on flow scheduling networks, sometimes quite significantly: for example, successive shortest path goes from being weakly to strongly polynomial.

This chapter starts by outlining each of the standard algorithms listed in table 3.1, before going on to discuss the approximate and incremental strategy. I include proofs of correctness, and the complexity results given in table 3.2. However, in the interests of brevity, these results are generally summarised in this chapter, with the proofs given in appendix C. I have included proofs in the main text only when they are critical to understanding the implementation of the algorithm, or where they are substantially my own work.

Name	Section	Approximation?	Incremental?
Cycle cancelling	Appendix C.1	✗	✗
Successive shortest path	3.3	✗	✓
Relaxation	3.4	✗	✓
Cost scaling	3.5	✓	✗

Table 3.1: How each algorithm implemented in this project was used.

Name	Section	Complexity (general)	Complexity (flow scheduling)
Cycle cancelling	Appendix C.1	$O\left(nm^2CU\right)$	$O\left(n^3CU\right)$
Successive shortest path	3.3	$O\left(nmU \lg n\right)$	$O\left(n^2 \lg n\right)$
Relaxation	3.4	$O\left(nm^2CU^2\right)$	$O\left(n^3CU\right)$
Cost scaling - FIFO	3.5	$O\left(n^2m \lg(nC)\right)$	$O\left(n^3 \lg(nC)\right)$
Cost scaling - WAVE	3.5	$O\left(n^3 \lg(nC)\right)$	$O\left(n^3 \lg(nC)\right)$

Table 3.2: Summary of asymptotic complexity of flow algorithms.

3.2 Cycle cancelling algorithm

My implementation began with cycle cancelling, a simple flow algorithm due to Klein [48]. Its performance is exponential in the worst case, and so is not a viable option for a high-performance flow solver. However, its simplicity allowed me to implement it early on in the project. This prototype enabled me to verify that I had correctly designed and implemented other classes, such as graph data structures and a parser. Since it is not used in the final version of Hapi, I describe the algorithm in appendix C.1.

3.3 Successive shortest path algorithm

In general, this algorithm runs in weakly polynomial time. However, for the class of networks produced by flow schedulers this improves to a strongly polynomial time bound of $O(n^2 \lg n)$. Successive shortest path lends itself readily to an incremental implementation (see §3.7), but is inappropriate for an approximate solver.

[0]Strongly polynomial variants exist [33, 62], but are still slow.

3.3.1 Algorithm description

Algorithm 3.3.1 Successive shortest path

1: $\mathbf{x} \leftarrow \mathbf{0}$ and $\boldsymbol{\pi} \leftarrow \mathbf{0}$
2: **while** mass balance constraints not satisfied **do**
3: choose excess vertex s and deficit vertex t[1]
4: solve SSSP problem from s to all vertices in the residual network $G_{\mathbf{x}}$, with respect to the reduced costs c_{ij}^{π}
5: let $\mathbf{d}$ denote the vector of shortest path distances, such that d_i is the shortest path from s to $i \in V$
6: $\boldsymbol{\pi} \leftarrow \boldsymbol{\pi} - \mathbf{d}$
7: let P denote a shortest path from s to t
8: $\delta \leftarrow \min\left(e_s, -e_t, \min\left\{r_{ij} : (i,j) \in P\right\}\right)$
9: augment δ units of flow along path P

The successive shortest path algorithm maintains a pseudoflow $\mathbf{x}$ (see §2.2.3) and potentials $\boldsymbol{\pi}$ satisfying reduced cost optimality (see theorem 2.2.5), and attempts to attain feasibility.

Each iteration of the algorithm identifies an excess vertex s and deficit vertex t. Next, the single-source shortest-path (SSSP) problem [13, ch. 24] is solved from s. To maintain reduced cost optimality, the potentials $\boldsymbol{\pi}$ are updated. Finally, the pseudoflow $\mathbf{x}$ is augmented along a shortest path P from s to t. The algorithm terminates when no excess or deficit vertices exist.

The amount of flow δ sent over the path is limited by the minimum residual capacity of arcs along P. δ is further restricted so that a non-negative supply at s and demand at t are maintained. This guarantees the excess at supply vertices and deficit at demand vertices is monotonically decreasing, ensuring each iteration improves the feasibility of the flow.

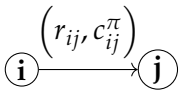
Figure 3.1 illustrates the operation of successive shortest path. Note that updating the potentials in figures 3.1c and 3.1e makes the reduced cost of arcs in the shortest-path tree rooted at **A** zero, allowing the flow to be augmented in figures 3.1d and 3.1f without violating reduced cost optimality.

3.3.2 Optimisations

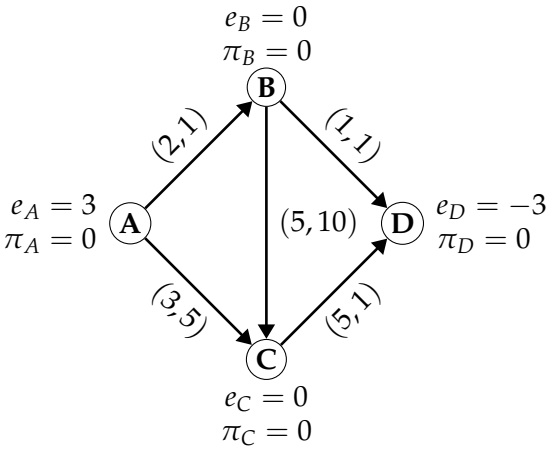
Choice of shortest-path algorithm

The fastest known single-source shortest-path algorithms are all variants of Djikstra's algorithm [1, ch. 4], differing only in the underlying heap data structure used. The

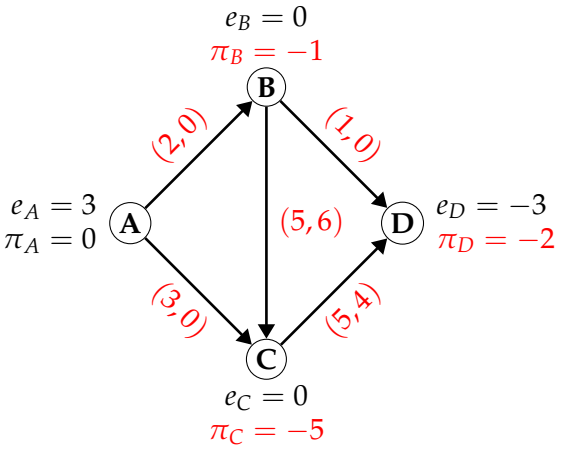
[1] Note so long as the mass balance constraints are unsatisfied, there must exist both an excess vertex s and deficit vertex t. This is because the total sum of excesses must equal the total sum of deficits for any feasible problem.



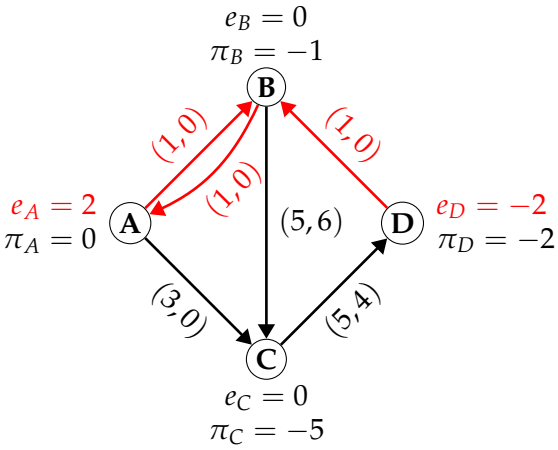
(a) Notation: each arc is labelled with residual capacity and reduced cost.



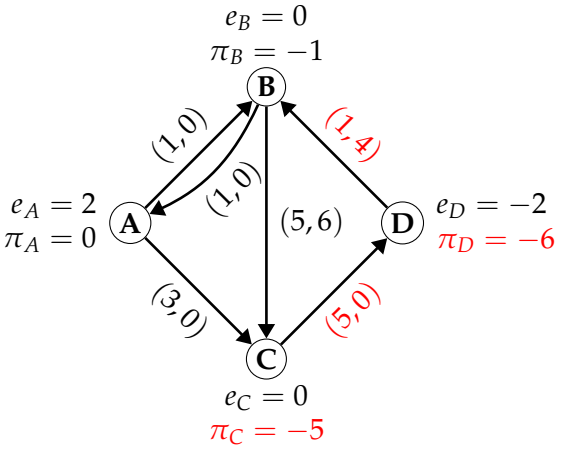
(b) Initial network.



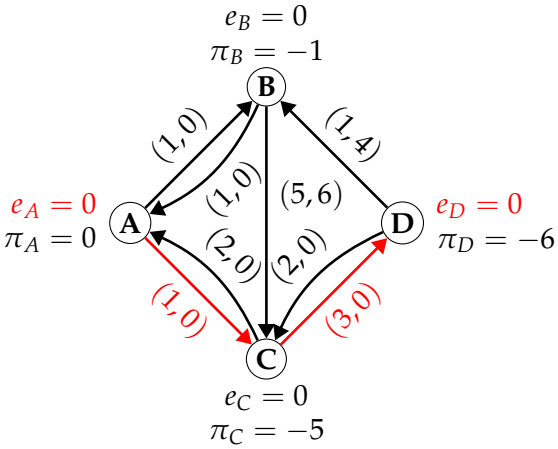
*(c) SSSP problem solved from **A**, and potentials updated.*



(d) Flow augmented along shortest path $A \to B \to D$.



*(e) SSSP problem solved from **A**, and potentials updated.*



(f) Flow augmented along shortest path $A \to C \to D$.

Figure 3.1: Step-by-step trace of the successive shortest path algorithm in action, depicting the residual network. Each node i is labelled with its excess e_i and potential π_i.

asymptotically fastest are based on Fibonacci heaps, with a complexity of $O(m + n \lg n)$. By contrast, Djikstra's algorithm has a complexity of $O(m \lg n)$ when using the popular binary heap data structure[2].

However, computational benchmarks have found Fibonacci heaps to be slower than binary heaps in practice, except on extremely large graphs [47, p. 15]. In light of this, I used a binary heap implementation in this project.

Terminating Djikstra's algorithm early

Djikstra's algorithm is said to have *permanently labelled* a vertex i when it extracts i from the heap. At this point, Djikstra has found a shortest path from s to i. I modified the successive shortest path algorithm to terminate Djikstra as soon as it permanently labels a deficit vertex l: details of the implementation and a proof of correctness are given in appendix C.2.2.

3.3.3 Analysis

Theorem 3.3.1 (Loop invariant of successive shortest path)**.** *Immediately before each iteration of the loop,* $(\mathbf{x}, \boldsymbol{\pi})$ *satisfies reduced cost optimality.*

Proof. See appendix C.2.1. □

Corollary 3.3.2 (Correctness of successive shortest path)**.** *Upon termination,* $\mathbf{x}$ *is a solution to the minimum-cost flow problem.*

Proof. See appendix C.2.1. □

Theorem 3.3.3 (Asymptotic complexity of generic successive shortest path)**.** *Let* $S(n, m, C)$ *denote the time complexity of solving a single-source shortest path problem (SSSP) with non-negative arc costs, bounded above by* C*, over a network with* n *vertices and* m *arcs. Then the time complexity of successive shortest path is* $O(nUS(n, m, C))$.

Proof. Each iteration of the loop body (lines 2 to 9 of algorithm 3.3.1) decreases the excess of vertex s by δ and the deficit of vertex t by δ, while leaving the excess/deficit of other vertices unchanged. Prior to the iteration, $e_s \geq 0$ and $e_t \leq 0$, so the total excess and total deficit in the network are both decreased by δ. By assumption 2.2.1, $\delta \geq 1$. Thus the number of iterations is bounded by the initial total excess. This in turn is bounded by $nU/2 = O(nU)$. So there are $O(nU)$ iterations in total.

Within each iteration, the algorithm solves an SSSP problem on line 4. Since reduced cost optimality is maintained throughout the algorithm, the arc costs in the shortest-

[2] Although binary and Fibonacci heaps have the same asymptotic complexity on flow scheduling networks, since $m = O(n)$ by lemma 2.2.2.

path problem are non-negative[3]. Thus the cost of solving this problem is $S(n, m, C)$.

The cost of lines 3 and 6 are clearly $O(n)$. Lines 8 and 9 are also $O(n)$, since the length of P is bounded by $n - 1$ (shortest paths are acyclic). Certainly $S(n, m, C) = \Omega(n)$, so the cost of each iteration is $O(S(n, m, C))$.

It follows that the overall time complexity of the algorithm is $O(nUS(n, m, C))$. □

Corollary 3.3.4 (Asymptotic complexity of successive shortest path in Hapi)**.** *The implementation in this project has a time complexity of* $O(nmU \lg n)$.

Proof. Djikstra's algorithm with a binary heap data structure is used to solve the SSSP problem, giving $S(n, m, C) = O(m \lg n)$. The result follows by theorem 3.3.3. □

Remark 3.3.1. On flow scheduling networks, by lemma 2.2.3 no vertex has a greater than unit supply. This gives a bound on the total excess of $O(n)$ rather than $O(nU)$. Moreover, by lemma 2.2.2, $m = O(n)$. So the complexity improves to $O(n^2 \lg n)$.

3.4 Relaxation algorithm

Like the successive shortest path algorithm, the relaxation algorithm works by augmenting along shortest paths in the residual network from excess vertices to deficit vertices. Unlike successive shortest path, it uses intermediate information to update vertex potentials as it constructs the shortest-path tree. It is inspired by Lagrangian relaxation [1, ch. 16][22].

Relaxation performs much better empirically than the successive shortest path algorithm, and is one of the fastest algorithms for some classes of flow networks [47]. However, its worst-case complexity is exponential, slower than any other algorithm considered in this dissertation[4]. Like successive shortest path, it is well suited to incremental operation, but is inappropriate for an approximate solver.

3.4.1 The relaxed integer programming problem

Before describing how the relaxation algorithm works, it is necessary to understand the problem it seeks to optimise. I have previously discussed the primal (see §2.2.1) and dual (see §2.2.3) versions of the minimum-cost flow problem. The relaxed version of the problem can be derived by applying Lagrangian relaxation to the primal problem, and has characteristics of both the primal and dual formulations. It is denoted by $\mathrm{LR}(\boldsymbol{\pi})$, and is defined by:

[3] Algorithms such as Djikstra which assume non-negative arc lengths are asymptotically faster than more general algorithms such as Bellman-Ford.

[4] The generic version of cycle cancelling is also exponential in the worst-case, however there are variants with a strongly polynomial bound. By contrast, there are no variants of relaxation achieving polynomial time, although in practice exponential runtime is not observed on realistic flow networks.

$$\text{maximise } w(\boldsymbol{\pi}) = \min_x \left[\sum_{(i,j)\in E} c_{ij}x_{ij} + \sum_{i\in V} \boldsymbol{\pi}_i e_i \right] \tag{3.1}$$

where $\mathbf{x}$ is subject to the capacity constraints first given in eq. (2.1):

$$\forall\,(i,j) \in E \cdot 0 \leq x_{ij} \leq u_{ij}.$$

Lemma 3.4.1. *An equivalent definition for $w(\boldsymbol{\pi})$ is:*

$$w(\boldsymbol{\pi}) = \min_x \left[\sum_{(i,j)\in E} c^{\pi}_{ij}x_{ij} + \sum_{i\in V} \pi_i b_i \right]. \tag{3.2}$$

Proof. Recall the excess at vertex i is defined as:

$$e_i = b_i + \sum_{(j,i)\in E} x_{ji} - \sum_{(i,j)\in E} x_{ij}$$

So:

$$\begin{aligned}
w(\pi) &= \min_x \left[\sum_{(i,j)\in E} c_{ij}x_{ij} + \sum_{i\in V} \pi_i \left(b_i + \sum_{(j,i)\in E} x_{ji} - \sum_{(i,j)\in E} x_{ij} \right) \right] && \text{substituting for } e_i;\\
&= \min_x \left[\sum_{(i,j)\in E} c_{ij}x_{ij} + \sum_{i\in V} \pi_i b_i + \sum_{i\in V} \pi_i \sum_{(j,i)\in E} x_{ji} - \sum_{i\in V} \pi_i \sum_{(i,j)\in E} x_{ij} \right] && \text{expanding;}\\
&= \min_x \left[\sum_{(i,j)\in E} c_{ij}x_{ij} + \sum_{i\in V} \pi_i b_i + \sum_{(j,i)\in E} \pi_i x_{ji} - \sum_{(i,j)\in E} \pi_i x_{ij} \right] && \text{double to single sum;}\\
&= \min_x \left[\sum_{(i,j)\in E} c_{ij}x_{ij} + \sum_{i\in V} \pi_i b_i + \sum_{(i,j)\in E} \pi_j x_{ij} - \sum_{(i,j)\in E} \pi_i x_{ij} \right] && \text{permuting } i \text{ and } j \text{ in 3rd sum;}\\
&= \min_x \left[\sum_{(i,j)\in E} \left(c_{ij} - \pi_i + \pi_j \right) x_{ij} + \sum_{i\in V} \pi_i b_i \right] && \text{factoring;}\\
&= \min_x \left[\sum_{(i,j)\in E} c^{\pi}_{ij}x_{ij} + \sum_{i\in V} \pi_i b_i \right] && \text{substituting reduced cost.}
\end{aligned}$$

□

Remark 3.4.1. Note that eq. (3.1) is expressed in terms of the excesses ($\mathbf{x}$-dependent) and arc costs, whereas eq. (3.2) is expressed in terms of the reduced costs ($\boldsymbol{\pi}$-dependent) and balances.

Lemma 3.4.2. *Let $\mathbf{x}$ be a pseudoflow and $\boldsymbol{\pi}$ vertex potentials. Then $(\mathbf{x}, \boldsymbol{\pi})$ satisfies reduced cost optimality conditions if and only if $\mathbf{x}$ is an optimal solution to* $\mathrm{LR}(\boldsymbol{\pi})$.

Proof. The complementary slackness conditions will be used rather than reduced cost optimality conditions, to simplify the proof. Recall the two conditions are equivalent (see §2.2.3).

By the definition of the relaxed problem and eq. (3.2), $\mathbf{x}$ is an optimal solution to $\mathrm{LR}(\boldsymbol{\pi})$ if and only if it minimises:

$$\sum_{(i,j)\in E} c_{ij}^{\boldsymbol{\pi}} x_{ij} + \sum_{i\in V} \pi_i b_i.$$

subject to the capacity constraints of eq. (2.1).

The second sum, $\sum_{i\in V} \pi_i b_i$, is constant in $\mathbf{x}$. Thus $\mathbf{x}$ is an optimal solution to $\mathrm{LR}(\boldsymbol{\pi})$ if and only if it minimises:

$$\sum_{(i,j)\in E} c_{ij}^{\boldsymbol{\pi}} x_{ij}$$

Note the coefficients $c_{ij}^{\boldsymbol{\pi}}$ are constant in $\mathbf{x}$. Furthermore, each term is independent of the others: by varying x_{ij}, only the term $c_{ij}^{\boldsymbol{\pi}} x_{ij}$ is affected[5]. Thus the sum is minimised when each of its summands is minimised.

When $c_{ij}^{\boldsymbol{\pi}} > 0$, the term $c_{ij}^{\boldsymbol{\pi}} x_{ij}$ is minimised by setting x_{ij} to the smallest value permitted by eq. (2.1), which is zero. When $c_{ij}^{\boldsymbol{\pi}} < 0$, set x_{ij} to the largest possible value, u_{ij}. When $c_{ij}^{\boldsymbol{\pi}} = 0$, the choice of x_{ij} is arbitrary. This is the same as the complementary slackness conditions given in theorem 2.2.6.

Thus complementary slackness optimality is equivalent to $\mathbf{x}$ being an optimal solution to $\mathrm{LR}(\boldsymbol{\pi})$. The result follows by equivalence of reduced cost optimality and complementary slackness optimality. □

Definition 3.4.1. Let z^* denote the optimal objective function value of the minimum-cost flow problem, that is the cost of an optimal flow.

Lemma 3.4.3.

(a) For any potential vector $\boldsymbol{\pi}$, $w(\boldsymbol{\pi}) \leq z^*$.

(b) There exists vertex potentials $\boldsymbol{\pi}^*$ for which $w(\boldsymbol{\pi}^*) = z^*$.

Proof. (Adapted from Ahuja *et al.* [1, lemma. 9.16])

For the first part, let $\mathbf{x}^*$ be a feasible flow with objective function value $s(\mathbf{x}^*) = z^*$: that is, $\mathbf{x}^*$ is a solution to the minimum-cost flow problem given in eq. (2.3).

Using the form of $w(\boldsymbol{\pi})$ stated in eq. (3.1), it follows that:

$$w(\boldsymbol{\pi}) \leq \sum_{(i,j)\in E} c_{ij} x_{ij}^* + \sum_{i\in V} \pi_i e_i$$

by dropping the $\min_x$ and replacing $=$ with $\leq$.

[5] Contrast this to if we were varying a vertex quantity, such as π_i, which would affect many arcs.

Note that the first sum is equal to $s(\mathbf{x}^*) = z^*$. Since $\mathbf{x}^*$ is a feasible flow, the mass balance constraints are satisfied, so:

$$\forall i \in V \cdot e_i = 0$$

and thus the second sum is equal to 0. It follows that:

$$w(\boldsymbol{\pi}) \leq z^* + 0 = z^*.$$

To prove the second part, let $\boldsymbol{\pi}^*$ be vertex potentials such that $(\mathbf{x}^*, \boldsymbol{\pi}^*)$ satisfy the reduced cost optimality conditions given in eq. (2.9)[6]. By lemma 3.4.2, it follows that $\mathbf{x}^*$ is an optimal solution to LR$(\boldsymbol{\pi}^*)$. Thus $w(\boldsymbol{\pi}^*) = z^*$. □

3.4.2 Algorithm description

The algorithm maintains a pseudoflow $\mathbf{x}$ and vertex potentials $\boldsymbol{\pi}$, such that $\mathbf{x}$ is an optimal solution to LR$(\boldsymbol{\pi})$. Equivalently, by lemma 3.4.2, $(\mathbf{x}, \boldsymbol{\pi})$ satisfies reduced cost optimality.

While the pseudoflow $\mathbf{x}$ is infeasible, the algorithm selects an excess vertex s. It then builds a tree in the residual network $G_{\mathbf{x}}$, rooted at s. Only arcs with zero reduced cost are added to the tree, hence it is a shortest-path tree.

The algorithm can perform one of two operations, described below. An operation is run as soon as its precondition is satisfied. After it executes, the tree is destroyed and the process repeats.

The first operation is to *update the potentials*, increasing the value of the objective function while maintaining optimality. That, is $\boldsymbol{\pi}$ is updated to $\boldsymbol{\pi}'$ such that $w(\boldsymbol{\pi}') > w(\boldsymbol{\pi})$ and $\mathbf{x}$ is updated to $\mathbf{x}'$ such that $\mathbf{x}'$ is an optimal solution to LR$(\boldsymbol{\pi}')$.

The second operation is to *augment the flow* $\mathbf{x}$, while leaving potentials $\boldsymbol{\pi}$ unchanged. The new flow $\mathbf{x}'$ remains an optimal solution of LR$(\boldsymbol{\pi})$, and improves feasibility.

The primary goal of the algorithm is to increase the value of the objective function, while the secondary goal is to increase feasibility (leaving the objective function unchanged). Therefore, when both operations are eligible to be executed, updating the potentials is preferred.

A few definitions are necessary before specifying the preconditions for these operations.

Definition 3.4.2 (Tree excess)**.** Let S denote the set of vertices spanned by the tree. Define:

$$e(S) = \sum_{i \in S} e_i. \tag{3.3}$$

[6]Note such a choice of $\boldsymbol{\pi}^*$ is guaranteed to exist since $\mathbf{x}^*$ is an optimal solution to the minimum-cost flow problem.

Definition 3.4.3 (Cuts)**.** A *cut* of a graph $G = (V, E)$ is a partition of V into two sets: S and $\overline{S} = V \setminus S$. We denote the cut by $\left[S, \overline{S}\right]$. Let $\left(S, \overline{S}\right)$ and $\left(\overline{S}, S\right)$ denote the set of forward and backward arcs crossing the cut, respectively. Formally:

$$\left(S, \overline{S}\right) = \{\, (i,j) \in E \mid i \in S \wedge j \in \overline{S} \,\};$$

$$\left(\overline{S}, S\right) = \{\, (i,j) \in E \mid i \in \overline{S} \wedge j \in S \,\}.$$

Definition 3.4.4. Let:

$$r(\boldsymbol{\pi}, S) = \sum_{(i,j) \in \left(S, \overline{S}\right) \wedge c_{ij}^{\pi}} r_{ij}. \tag{3.4}$$

Remark 3.4.2. Since the tree only ever contains arcs of zero reduced cost, $r(\boldsymbol{\pi}, S)$ represents the total residual capacity of the arcs which are eligible for addition to the tree in the current iteration.

Algorithm 3.4.1 Relaxation: main procedure

```
1:  x ← 0 and π ← 0
2:  while network contains an excess vertex s do
3:      S ← { s }
4:      initialise e(S) and r(π, S)
5:      if e(S) > r(π, S) then UpdatePotentials
6:      while e(S) ≤ r(π, S) do
7:          select arc (i, j) ∈ (S, S̄) in the residual network with c_ij^π = 0
8:          if e_j ≥ 0 then
9:              Pred_j ← i
10:             S ← S ∪ { j }
11:             update e(S) and r(π, S)
12:         else
13:             AugmentFlow(j)
14:             break
15:     if e(S) > r(π, S) then UpdatePotentials
```

Algorithm 3.4.2 Relaxation: potential update procedure

Precondition: $e(S) > r(\boldsymbol{\pi}, S)$

```
1:  function UpdatePotentials
2:      for every arc (i, j) ∈ (S, S̄) in the residual network with c_ij^π = 0 do
3:          saturate arc (i, j) by sending r_ij units of flow
4:      compute α ← min { c_ij^π | (i, j) ∈ (S, S̄) and r_ij > 0 }
5:      for every vertex i ∈ S do
6:          π_i ← π_i + α
```

Algorithms 3.4.1 to 3.4.3 formally describe the solution method. A set S of vertices spanned by the shortest path tree is maintained. The arcs in the tree are specified by the predecessor array, Pred. Although $e(S)$ and $r(\boldsymbol{\pi}, S)$ could be computed when

Algorithm 3.4.3 Relaxation: flow augmentation procedure

Precondition: $e(S) \leq r(\boldsymbol{\pi}, S)$ and $e_t < 0$

1: **function** AUGMENTFLOW(t)
2: chase predecessors in Pred starting from t to find a shortest path P from s to t
3: $\delta \leftarrow \min\left(e_s, -e_t, \min\left\{\, r_{ij} \mid (i,j) \in P \,\right\}\right)$
4: augment δ units of flow along path P

needed, they are more efficiently maintained as variables, being updated whenever a vertex is added to the tree.

Each iteration of the outer loop of algorithm 3.4.1 selects an excess vertex, s, to be the root of a new tree. The inner loop adds vertices and arcs to this tree. If a deficit vertex t is encountered, flow is augmented along the (zero-cost) shortest path discovered from s to t, and the inner loop terminates. However, an iteration may end before this: if at any point $e(S) > r(\boldsymbol{\pi}, S)$, the potentials are updated, and a new iteration starts[7].

3.4.3 Analysis

Correctness

Theorem 3.4.4 (Correctness of relaxation)**.** *Immediately before each iteration of the loop,* $(\mathbf{x}, \boldsymbol{\pi})$ *satisfies reduced cost optimality. Moreover, upon termination,* $\mathbf{x}$ *is a solution of the minimum-cost flow problem.*

Proof. See appendix C.3. □

Asymptotic complexity

Analysis of the algorithm's complexity is conspicuously absent from all the standard sources. Authors are, perhaps, hesitant to highlight relaxation's worst-case exponential performance. I provide my own analysis below.

Lemma 3.4.5. *The time complexity of* UPDATEPOTENTIALS *is* $O(m)$.

Proof. The for loop on lines 2 to 3 has cost $O(m)$: the iteration is over $O(m)$ arcs, and saturating an arc is an $O(1)$ operation. Computing α on line 4 is $O(m)$, as $\left(S, \overline{S}\right)$ contains at most m arcs. The number of vertices in S is certainly $O(n)$; it follows that updating potentials on lines 5 to 6 contributes an $O(n)$ cost. The overall time complexity is thus $O(m)$, as $m = \Omega(n)$. □

[7] Note line 5 of algorithm 3.4.1 is just an optimisation.

Lemma 3.4.6. *The time complexity of* AugmentFlow *is* $O(n)$.

Proof. The length of shortest path P is bounded by $n-1 = O(n)$ as it is acyclic. The operations on lines 2 to 4 are linear in the length of P, thus the overall time complexity is $O(n)$. □

Lemma 3.4.7. *The following properties hold:*

(a) UpdatePotentials *is called at most* mCU *times.*

(b) AugmentFlow *is called at most* nU *times in between each call to* UpdatePotentials.

(c) There are at most $nmCU^2$ *iterations of the outer loop on lines 2 to 15 of algorithm 3.4.1.*

Proof. A feasible flow has a cost of at most mCU. By lemma 3.4.3, $w(\boldsymbol{\pi}) \leq z^*$. By lemma C.3.1 and assumption 2.2.1, $w(\boldsymbol{\pi})$ increases by at least one for each call of UpdatePotentials. Moreover, $w(\boldsymbol{\pi})$ never decreases: $\boldsymbol{\pi}$ is only modified by UpdatePotentials. Thus UpdatePotentials is called at most mCU times.

Each call to AugmentFlow decreases the total excess by at least one unit, by lemma C.3.2 and assumption 2.2.1. The total excess in a network is at most nU. The only part of the algorithm which may *increase* the excess at vertices is lines 2 to 3 of UpdatePotentials. Thus, in between calls to UpdatePotentials, at most nU calls to AugmentFlow may be made.

For a bound on the total number of iterations, note an iteration ends after calling either AugmentFlow on line 13 or UpdatePotentials on line 15[8]. By part (a), we know there are at most mCU calls to UpdatePotentials. This together with part (b) implies there are at most $nmCU^2$ calls to AugmentFlow. This gives the required bound on the number of iterations. □

Theorem 3.4.8. *The algorithm runs in* $O(nm^2CU^2)$ *time.*

Proof. Initialisation on line 1 of algorithm 3.4.1 has cost $O(m)$. The loop body on lines 2 to 15, excluding function calls and the inner loop on lines 6 to 14, has cost $O(1)$: initialising S, $e(S)$ and $r(\boldsymbol{\pi}, S)$ are all constant cost.

Now, consider the execution of the inner loop body on lines 6 to 14. To update $r(\boldsymbol{\pi}, S)$ after adding a vertex j to S, it is necessary to scan over the adjacency list $\text{Adj}(j)$ of j. This dominates the other costs of lines 8 to 11: updating the predecessor on line 9 and inserting an element in set S on line 10 are both constant cost.

It is natural to maintain a queue of arcs $(i,j) \in (S, \overline{S})$ in the residual network with $c_{ij}^{\pi} = 0$. This does not increase the asymptotic complexity of lines 8 to 11, since much the same work must already be done when updating r. Using this queue, line 7 has cost $O(1)$. Thus the inner loop body has complexity, excluding procedure calls, of $O(|\text{Adj}(j)|)$.

[8] Of course UpdatePotentials might be called on line 5. This is not guaranteed, however, so cannot form part of the complexity analysis.

Note that each iteration of this loop adds a new vertex to S, and so is executed at most once per vertex. It follows that the total complexity of the inner loop on lines 6 to 14, excluding procedure calls, is $O(m)$. This loop thus dominates the other costs on lines 2 to 15, so the loop body overall has a cost of $O(m)$.

The overall time complexity can now be proved using the above results and the bounds in lemma 3.4.7. UpdatePotentials is called $O(mCU)$ times and has cost $O(m)$, so contributes a total cost of $O(m^2CU)$. AugmentFlow is called $O(nmCU^2)$ times and has cost $O(n)$, so contributes a total cost of $O(n^2mCU^2)$. Finally, the body of the while loop on lines 2 to 15 of algorithm 3.4.1 is executed $O(nmCU^2)$ times. Excluding the cost of the procedure calls, each iteration costs $O(m)$, so this contributes a cost of $O(nm^2CU^2)$. This dominates the other costs, and is thus the overall time complexity of the algorithm. □

Remark 3.4.3. As with remark 3.3.1, there is an $O(n)$ bound on the total excess in flow scheduling networks by lemma 2.2.3, improving lemma 3.4.7(b) from $O(nU)$ to $O(n)$.

Note that, in general, $U = \omega(1)$ as there is no bound on the capacity of arcs in flow scheduling networks. Accordingly, the bound on the maximum cost of a feasible flow remains $O(mCU)$.

The complexity is thus $O(nm^2CU)$ on flow scheduling networks by the proof of theorem 3.4.8, which improves to $O(n^3CU)$ by lemma 2.2.2.

3.5 Cost scaling algorithm

The cost scaling method is due to Goldberg and Tarjan [28]. The generic version runs in weakly polynomial time, with variants offering strongly polynomial bounds. It is one of the most efficient solution methods, in both theory and practice[9]. Cost scaling works by successive approximation, so is readily adapted to create an approximate solver (see §3.6). It is not suitable to being used incrementally, however.

A feasible flow is maintained by the algorithm. The flow need not be optimal, but the error is bounded using the notion of ϵ-optimality. Each iteration of the algorithm refines the solution, halving the error bound, until optimality is achieved.

The next section formally defines ϵ-optimality. Following this, the general solution method is described. Afterwards, specific variants of the algorithm are considered. Finally, techniques to improve the practical performance of implementations are considered.

[9] In computational benchmarks such as the one by Király and Kovács [47], cost scaling is the fastest algorithm for many classes of network. It is occasionally beaten by other algorithms, although its performance is still competitive in these cases. This makes it more robust than some alternative solution methods, such as relaxation (see §3.4), which sometimes outperform it, but are in other cases orders of magnitude slower.

3.5.1 ϵ-optimality

Definition 3.5.1 (ϵ-optimality)**.** Let $\epsilon \geq 0$. A pseudoflow $\mathbf{x}$ is ϵ-optimal with respect to vertex potentials $\boldsymbol{\pi}$ if the reduced cost of each arc in the residual network $G_{\mathbf{x}}$ is greater than $-\epsilon$:

$$\forall (i,j) \in E_{\mathbf{x}} \cdot c_{ij}^{\pi} \geq -\epsilon \tag{3.5}$$

Remark 3.5.1. Note this is a relaxation of the reduced cost optimality conditions given in eq. (2.9). When $\epsilon = 0$, the definition of ϵ-optimality in eq. (3.5) is equivalent to that of eq. (2.9). A stronger result is proved below.

Theorem 3.5.1. *Let $0 \leq \epsilon < 1/n$, and let $\mathbf{x}$ be an ϵ-optimal feasible flow. Then $\mathbf{x}$ is optimal.*

Proof. (From Goldberg and Tarjan [28, theorem 3.4])

Let C be a simple cycle in $G_{\mathbf{x}}$. C has at most n arcs. By definition 3.5.1, $c_{ij}^{\pi} \geq -\epsilon$ for each arc $(i,j) \in C$. It follows the cost of C is $\geq -n\epsilon$.

The condition $\epsilon < 1/n$ implies $n\epsilon > -1$. By assumption 2.2.1, cost of C must then be ≥ 0. The result follows from the negative cycle optimality conditions given in theorem 2.2.4. □

3.5.2 High-level description of the algorithm

Algorithm 3.5.1 Cost scaling

```
1: x ← result of maximum-flow algorithm        ▷ establishes x feasible
2: π ← 0
3: ε ← C                                       ▷ where C is largest arc cost, see §2.2.3
4: while ε ≥ 1/n do                            ▷ loop until x is optimal
5:     ε ← ε/α                                 ▷ α is a constant scaling factor, commonly set to 2
6:     REFINE(x,π,ε)
```

The procedure described in algorithm 3.5.1 is inspired by theorem 3.5.1. First, the algorithm finds a feasible flow $\mathbf{x}$. The potentials $\boldsymbol{\pi}$ are initialised to zero, so reduced costs are equal to arc costs. $(\mathbf{x}, \boldsymbol{\pi})$ is then trivially C-optimal, and so ϵ is initialised with this value. Next, the algorithm iteratively improves the approximation using the REFINE routine.

This routine is described in algorithm 3.5.2. Lines 2 to 4 ensure the reduced cost optimality conditions given in eq. (2.9) are satisfied, making $\mathbf{x}$ 0-optimal. Lines 5 to 8 bring the pseudoflow back to feasibility by applying a sequence of the basic operations, PUSH and RELABEL, both of which preserve ϵ-optimality.

One might wonder why ϵ is not immediately set to a value $< 1/n$ in lines 5 to 8: only one iteration of REFINE would then be needed. This approach is taken in an algorithm due to Bertsekas [5], but it results in exponential time complexity. In general, there is a trade-off when choosing the scaling factor α (which controls the rate at which

Algorithm 3.5.2 Cost scaling: generic REFINE routine

Precondition: $\mathbf{x}$ is a pseudoflow
Postcondition: $\mathbf{x}$ is a feasible flow and $(\mathbf{x}, \boldsymbol{\pi})$ is ϵ-optimal

1: **function** REFINE($\mathbf{x}$,$\boldsymbol{\pi}$,ϵ)
2: **for** every arc $(i,j) \in E$ **do** ▷ initialisation
3: **if** $c_{ij}^{\pi} > 0$ **then** $x_{ij} \leftarrow 0$
4: **if** $c_{ij}^{\pi} < 0$ **then** $x_{ij} \leftarrow u_{ij}$
5: **while** mass balance constraints not satisfied **do** ▷ main loop
6: select an excess vertex s
7: **if** $\exists (s,j) \in E_{\mathbf{x}} \cdot c_{sj}^{\pi} < 0$ **then** PUSH(s,j)
8: **else** RELABEL(s,ϵ)

ϵ is decreased). Increasing α decreases the number of calls needed to REFINE, but increases the time each call takes[10].

Lines 5 to 8 of REFINE are non-deterministic: there may be multiple excess vertices s, and multiple arcs $(s,j) \in E_{\mathbf{x}}$ satisfying the conditions given. The correct final solution is obtained whatever choices are made, as shown in the next section. Moreover, some complexity bounds can be proved for this generic version of the algorithm. However, both the theoretical and practical performance of the algorithm depends upon the rule used to select operations (see §3.5.3).

Algorithm 3.5.3 Cost scaling: the basic operations, push and relabel

Precondition: $e_i > 0$, $(i,j) \in E_{\mathbf{x}}$ and $c_{ij}^{\pi} < 0$
1: **function** PUSH(i,j)
2: send $\min\left(e_i, r_{ij}\right)$ units of flow from i to j

Precondition: $e_i > 0$ and $\forall (i,j) \in E_{\mathbf{x}} \cdot c_{ij}^{\pi} \geq 0$
1: **function** RELABEL(i,ϵ)
2: $\pi_i \leftarrow \min\{\, \pi_j + c_{ij} + \epsilon \mid (i,j) \in E_{\mathbf{x}} \,\}$

The basic operations PUSH and RELABEL are described in algorithm 3.5.3. PUSH sends as much flow as possible along arc (i,j) from excess vertex i to vertex j, without exceeding the excess at i or the capacity constraint on arc (i,j). RELABEL increases the potential of i, decreasing the reduced cost c_{ij}^{π} of arcs leaving i, allowing more PUSH operations to take place.

Analysis

Theorem 3.5.2 (Correctness of cost scaling)**.** *Upon termination of the algorithm described in algorithm 3.5.1,* $\mathbf{x}$ *is a solution of the minimum-cost flow problem.*

[10] This trade-off is discussed further in §4.3.3.

Proof. See appendix C.4.1. □

Theorem 3.5.3 (Complexity due to basic operations in cost scaling)**.** *Within an invocation of* Refine*, the basic operations contribute a complexity of* $O(n^2m)$*.*

Proof. See appendix C.4.2. □

Remark 3.5.2. It is not possible to prove a bound on the complexity of the generic refine routine given in algorithm 3.5.2, as it depends on how vertices and arcs are chosen on how excess vertices are selected on lines 6 and 7. Bounds on specific implementations of refine are given in §3.5.3.

Theorem 3.5.4 (Complexity of cost scaling)**.** *Let* $R(n, m)$ *be the running time of the* Refine *routine. Then the minimum-cost flow algorithm described in algorithm 3.5.1 runs in* $O\left(R(n, m) \lg(nC)\right)$ *time.*

Proof. See appendix C.4.2. □

3.5.3 Implementations of the algorithm

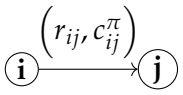
The preceding section presented a generic version of the Refine routine, algorithm 3.5.2. The order in which the basic operations are to be applied is intentionally left unspecified in lines 5 to 8 of the algorithm. Both the practical and theoretical performance of cost scaling is affected by this order. I implemented two variants, *FIFO* and *wave*, which differ in the rule used to select basic operations. Both versions were first proposed by Goldberg and Tarjan [34].

FIFO refine

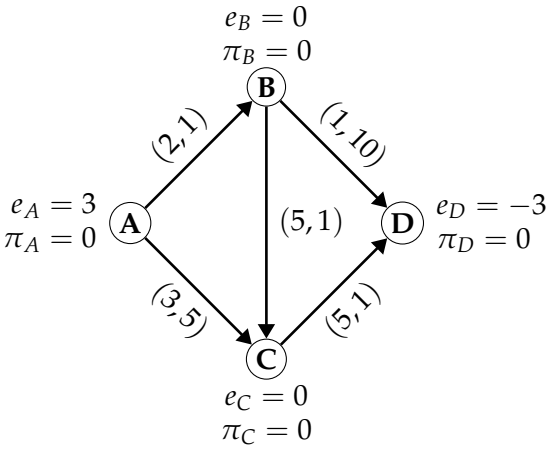
Algorithm 3.5.4 Cost scaling: FIFO Refine routine

```
1: function Refine(x,π,ϵ)
2:     initialisation as in lines 2-4 of algorithm 3.5.2
3:     Q ← [s ∈ V : e_s > 0]                    ▷ Q is a queue of excess vertices
4:     while Q not empty do
5:         pop head of Q into s
6:         Discharge(s,ϵ)                       ▷ May add vertices to Q
7:         if Relabel called by Discharge then
8:             add s to rear of Q
9:             break
```

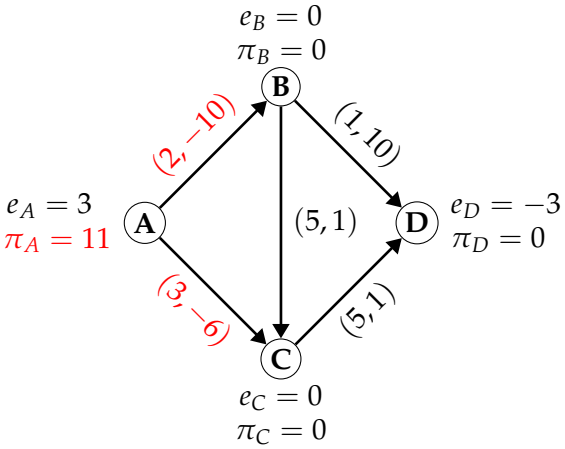
Algorithm 3.5.4 maintains a first-in-first-out (FIFO) queue Q of excess vertices (see §2.2.3). The initial order of vertices in the queue is arbitrary. At each iteration, an excess vertex s is removed from the head of the queue, and *discharged* by applying a sequence of Push and Relabel operations (described below). During execution of



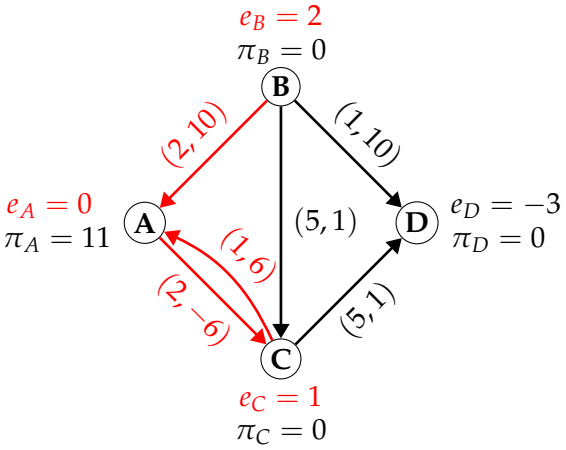
(a) Notation: each arc is labelled with residual capacity and reduced cost.



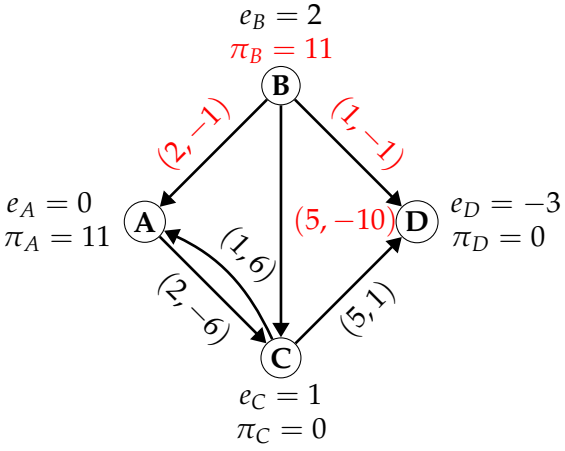
(b) Initial network. $Q = [A]$.



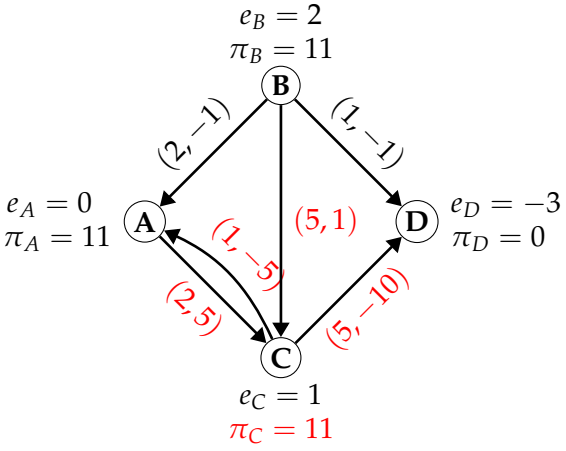
(c) Discharge(A)*: immediately calls* Relabel(A) *as there are no outgoing arcs with negative reduced cost. Q remains $[A]$, as A is popped and then immediately pushed again.*



(d) Discharge(A)*: calls* Push(A, B) *and* Push(A, C)*, leaving $Q = [B, C]$.*



(e) Discharge(B)*: immediately calls* Relabel(B)*, leaving $Q = [C, B]$.*



(f) Discharge(C)*: immediately calls* Relabel(C)*, leaving $Q = [B, C]$.*

Figure 3.2: Partial trace of FIFO Refine *from algorithm 3.5.4, depicting the residual network after each call to* Discharge*. Each node i is labelled with its excess e_i and potential π_i. ϵ is taken to be 10, as would be the case on the first call to* Refine *from algorithm 3.5.1 (note the maximum cost C in the network is 10.)*

Algorithm 3.5.5 Cost scaling: DISCHARGE and helper routine PUSHORRELABEL

Precondition: $e_s > 0$

```
1: function DISCHARGE(s,ϵ)
2:     repeat
3:         PUSHORRELABEL(s,ϵ)
4:     until e_s ≤ 0
```

Precondition: $e_s > 0$

```
1: function PUSHORRELABEL(s,ϵ)
2:     let (i, j) be the current arc of s
3:     if (i, j) ∈ E_x and c^π_ij < 0 then PUSH(i,j)
4:     else
5:         if (i, j) last arc on the adjacency list for s then
6:             RELABEL(s)
7:             current arc ← first arc in adjacency list
8:         else
9:             current arc ← next arc in adjacency list
```

the algorithm, new vertices may gain an excess; in this case, they are added to the rear of Q.

DISCHARGE(s) is described in algorithm 3.5.5. It may be applied to any excess vertex, and performs a sequence of push operations until $e_s = 0$ or RELABEL is called. For each vertex $i \in V$, an adjacency list is maintained, in a fixed (but arbitrary) order. The helper routine PUSHORRELABEL walks over this adjacency list, performing PUSH operations when possible. When the end of this adjacency list is reached, no further PUSH operations can be performed, and RELABEL is invoked.

Note any new excess vertices generated during execution of the algorithm must be added to the rear of Q. PUSH(i, j) may make j an excess vertex: algorithm 3.5.3 must be modified to check for this case. The code for REFINE in algorithm 3.5.2 must also be updated. REFINE pops s from Q and then calls DISCHARGE. If RELABEL is called by DISCHARGE, s will still be an active vertex when DISCHARGE returns, and must be added back to the queue Q.

Figure 3.2 illustrates the operation of FIFO REFINE, showing how each call to DISCHARGE modifies the potentials $\mathbf{ß}$ and flow $\mathbf{x}$ on a simple network.

Theorem 3.5.5. *The algorithm with FIFO refine has a running time of* $O(n^2 m \lg(nC))$.

Proof. See Goldberg and Tarjan [34, theorem 6.2]. □

Remark 3.5.3. In fact, this complexity bound holds for any refine routine which repeatedly applies PUSHORRELABEL. However, it has been conjectured by Goldberg that the FIFO ordering of vertices in this variant results in a tighter bound of $O(n^3 \lg(nC))$. Proving or disproving this assertion is an open research problem.

Remark 3.5.4. On flow scheduling networks, $m = O(n)$ by lemma 2.2.2, so the bound is trivially $O(n^3 \lg(nC))$ in any case.

Wave refine

Algorithm 3.5.6 Cost scaling: wave REFINE routine

```
1:  function REFINE(x,π,ε)
2:      initialisation as in lines 2-4 of algorithm 3.5.2
3:      L ← list of vertices in V                    ▷ order of vertices is arbitrary
4:      repeat
5:          for each vertex i in list L do
6:              if e_i > 0 then
7:                  DISCHARGE(i,ε)
8:                  if RELABEL called by DISCHARGE then
9:                      move i to front of L
10:     until no excess vertex encountered in for loop
```

Algorithm 3.5.6 describes an approach which can be proved to achieve the $O(n^3 \lg(nC))$ bound on any flow network. The method maintains a list L of all vertices in the network, rather than the queue Q maintained by the FIFO method. The algorithm preserves the invariant that L is topologically ordered with respect to the *admissible graph*: the subgraph of the residual network containing only arcs with negative reduced cost.

Initially, L contains vertices in arbitrary order. But the admissible graph is empty after line 2 of algorithm 3.5.6, since lines 2 to 4 of algorithm 3.5.2 saturate any negative reduced cost arcs, making them drop out of the residual network. Thus, L is trivially topologically ordered.

PUSH operations cannot create admissible arcs, and so preserve the topological ordering. An operation RELABEL(s) can create admissible arcs. However, there are guaranteed to be no admissible arcs entering s immediately after the operation [28, lemma 6.5]. Moving s to the beginning of L therefore ensures a topological ordering is maintained.

Theorem 3.5.6. *The algorithm with wave refine has a running time of* $O(n^3 \lg(nC))$.

Proof. Using the invariant that L is topologically ordered, it is possible to prove an $O(n^2)$ bound on the number of passes over the vertex list [28, lemma 7.3]. An $O(n^3)$ bound on the running time of the wave REFINE routine can be shown as a corollary of this [28, theorem 7.4]. The result follows by theorem 3.5.4. □

3.5.4 Heuristics

So far, this dissertation has only considered means to improve the theoretical performance of cost scaling. Considerable research effort has gone into improving its practical performance by means of *heuristics* that guide the actions of the algorithm [29].

I decided not to implement any heuristics. The heuristics are already well-studied: implementing them would not have produced any new insight, and would have diverted considerable development time from other efforts. I judged that it was better to keep my implementation simple, allowing me to rapidly explore fundamentally different approaches. However, for the sake of completeness I have summarised the heuristics in appendix C.4.3.

3.6 Approximation algorithms

Feasible solutions to flow scheduling networks correspond to scheduling assignments. A solution's cost indicates the degree to which the assignment meets the scheduler's policy goals. The algorithms considered so far find *minimal* cost flows, producing scheduling assignments which *maximise* the policy objectives.

Finding the best solution may take some time, yet minimising scheduling latency is itself important (see §2.1). I develop an approximate solution method, allowing the trade-off between accuracy and speed to be chosen explicitly.

I believe this is the first study of approximation algorithms for the minimum-cost flow problem. There is some prior work on finding approximate solutions for the related NP-complete problems of multi-commodity flows [26] and dynamic flows [39]. These techniques are of limited applicability to minimum-cost flow problems[11], however, and Hapi uses entirely different methods for approximation.

3.6.1 Choice of algorithms

The solution returned by an approximate algorithm must be *feasible*, although it need not be *optimal*. Of the algorithms considered in sections 3.2 to 3.5, only cycle cancelling and cost scaling maintain feasibility as an invariant[12].

Both cycle cancelling and cost scaling find successive approximations to an optimal solution, and so can readily be adapted to find approximate solutions. In cycle cancelling, the cost of the solution monotonically decreases after each iteration

[11] For example, approximation methods for multi-commodity flow problems often work by solving many single-commodity flow problems. Obviously this reduction method does not help when the problem is already single-commodity.

[12] Successive shortest path and relaxation, by contrast, operate on a pseudoflow and maintain optimality as an invariant.

(shown in the proof of lemma C.1.3). However, a large decrease in the cost of the solution may occur at *any* point in the algorithm's execution.

By contrast, in cost scaling the error bound ϵ undergoes exponential decay[13], reaching a small value many iterations before the algorithm would normally terminate. It will therefore produce a better approximation than cycle cancelling. Cost scaling is also considerably faster than cycle cancelling, making it the clear choice for an approximate solver.

3.6.2 Adapting cost scaling

Cost scaling finds successive ϵ-optimal solutions, where ϵ is decreased by a constant factor α after every iteration (see algorithm 3.5.1). It terminates when $\epsilon < 1/n$, at which point the solution is optimal by theorem 3.5.1. Relaxing this terminating condition produces an approximate solver.

Ideally, it would be possible to specify a bound on the relative error of the solution: for example, terminating once the error is smaller than 1%. Unfortunately, computing the relative error requires knowing the minimum cost, which can only be determined by finding an optimal solution.

It is possible to derive a bound on the absolute error of the solution, in terms of ϵ. Unfortunately, the bound is of limited practical use. Definition 3.5.1 of ϵ-optimality has the simple corollary that:

$$\epsilon = \max_{(i,j) \in E_{\mathbf{x}}} -c_{ij}^{\pi}$$

ϵ therefore depends on the 'worst' arc in the residual network: i.e. the one with the most negative reduced cost. If every arc were to attain this worst case, the error could be large even for small values of ϵ. But in practice, this does not occur. The bound therefore tends to significantly overestimate the solution error.

As accurate analytic bounds on the error are not available, I had to develop heuristic terminating conditions. These are the focus of the rest of this section.

Cost convergence

The first few iterations of cost scaling tend to produce large reductions in the cost. When the solution is nearly optimal, the change in cost between successive iterations is typically small. This is illustrated in figure 3.3, which shows cost scaling running on a scheduling flow network.

[13] The solution cost *tends* to also undergo exponential decay. However, this is not guaranteed: the error bound is not tight, and the cost on occasion may even increase.

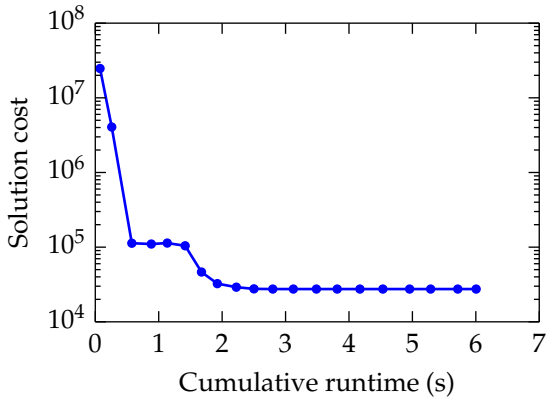


Figure 3.3: Successive approximations to an optimal solution. Each iteration of cost scaling is marked by a point on the graph. The spacing between points on the x-axis indicate the time taken by Refine*. This varies throughout execution of the algorithm, but does not show any trend:* Refine *runs for a similar length of time immediately before optimality as for the first iteration. Note that the y-axis, representing solution cost, is on a log scale. The results are for a scheduling flow network from the dataset described in §4.4.1.*

Cost convergence exploits this property, by measuring the difference between the cost after successive iterations. If the cost changes by a factor less than some threshold t, the algorithm terminates.

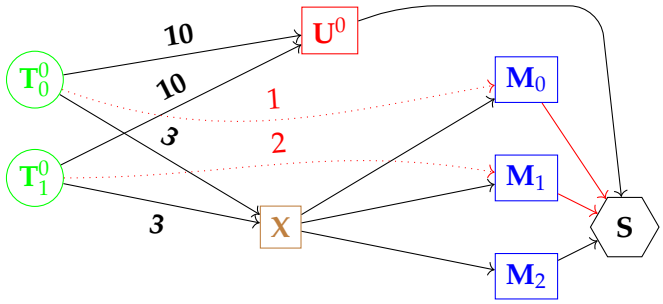
Task migration convergence

Cost convergence is a general approach, applicable to any class of flow network. By contrast, this heuristic exploits the nature of flow scheduling. Vertices are labelled as representing either a task, machine or other entity. Task assignments are computed from the flow (see §2.2.2), and the number of changes from the last iteration is measured. The algorithm terminates when this drops below a threshold t.

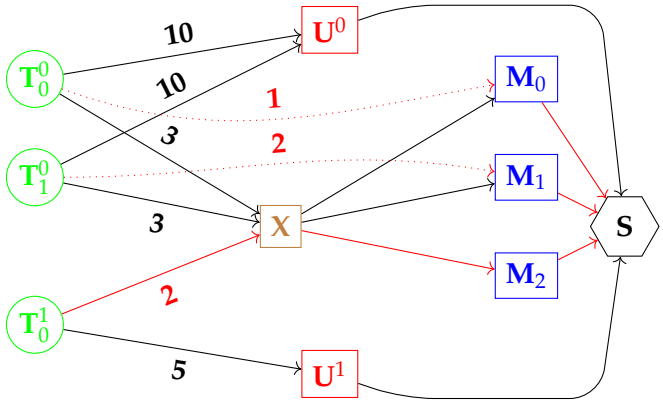
3.7 Incremental algorithms

Flow scheduling systems produce a sequence of closely related flow networks. Each new network in the sequence is generated in response to cluster events. Figure 3.4 gives an example of the changes triggered by task submission, one of the most common events. Other events include task completion, machine addition and removal.

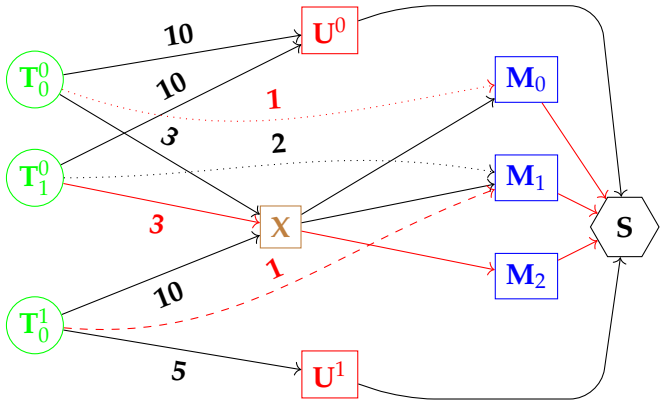
These events cause localised changes to the flow network. Consequently, the optimal flow tends to remain mostly the same. My method of *incremental* solution is designed to exploit this. The algorithm reuses the solution for the old flow network, *reoptimising* to produce an answer for the new network. This approach proved to be extremely successful, producing a factor of 14.5× speedup in my tests (see §4.5). I believe my



(a) The initial network, showing tasks $\mathbf{T}_0^0$ and $\mathbf{T}_1^0$ running on $\mathbf{M}_0$ and $\mathbf{M}_1$ respectively.



(b) A job is submitted, consisting of a single task $\mathbf{T}_0^1$. It has no preference arcs, so its flow drains through $\mathbf{X}$ and into the idle machine $\mathbf{M}_2$.



(c) An alternative to (b), where the task has a strong preference for running on machine $\mathbf{M}_1$, which is busy executing task $\mathbf{T}_1^0$. The cost of the running arc $\mathbf{T}_1^0 \to \mathbf{M}_1$ is lower than the cost of $\mathbf{T}_1^0 \to \mathbf{X}$, as the task is partially complete (see appendix B.3). Nevertheless, the cheapest option is to migrate $\mathbf{T}_1^0$ *to $\mathbf{M}_2$, satisfying $\mathbf{T}_0^1$'s strong preference.*

*Figure 3.4: (b) and (c) show the result of incremental changes to the original network (a). Arcs are labelled with their cost; any unlabelled arc has a cost of zero. The optimal solution is shown by highlighting the arcs which carry flow in red. Dotted lines represent **running arcs**, dashed lines **preference arcs** (see §2.2.2).*

implementation is the first of its kind: there is certainly no mention of this method in the literature.

Flow scheduling is ideally suited to an incremental solution method. However, the approach is much less compelling for traditional applications of flow networks: this may go some way to explaining the lack of prior work. As previously mentioned, most network changes in flow scheduling are small. This alone does not make the problem easy: reoptimising after even a single change is, in general, as hard as solving the problem from scratch[14]. Informally, the problem is that since the optimisation problem is global, an update in one region of the network may trigger a cascading sequence of changes in the optimal flow for distant regions.

Crucially, however, scheduling policies are designed to produce stable assignments of tasks to machines. Preempting a task is an expensive operation, which should be performed sparingly. Consequently, the cost model for any viable scheduling policy naturally avoids frequent changes to the optimal flow. This is precisely the property required for incremental solvers to perform well.

I describe the sources I drew on to develop my incremental solver in the next section, §3.7.1. Following this, I discuss my approach to building such a solver in §3.7.2. After this background, I evaluate the suitability of different flow algorithms to the incremental problem in §3.7.3. Following this, §3.7.5 provides details of the solvers I implemented.

3.7.1 Related work

The Quincy paper highlighted incremental solvers as an area for further research, sparking my interest in such an approach [42, §6.5]. There has been some study of reoptimisation and sensitivity analysis on flow networks, which are related to the incremental problem, although these areas have received comparatively little attention. I survey the existing work below.

Amini and Barr published a comprehensive empirical evaluation of reoptimisation algorithms in 1993 [2]. However, the study tested algorithms such as the out-of-kilter method, now long obsolete. Moreover, the largest network studied had only 1,500 vertices, orders of magnitude smaller than the ones considered in this project.

The most recent work is a computational study of algorithms for reoptimisation after arc cost changes, published by Frangioni and Manca in 2006 [25]. Although now almost a decade old, the algorithms evaluated are still in contemporary use. However,

[14] To see this, consider two networks A and B. WLOG, assume they each have a single supply vertex s_x and demand vertex t_x for $x \in A, B$. Remove the supply and demand at those vertices, introducing new supply and demand vertices s and t. Add arcs $s \to s_x$ with capacity b_{s_x}, and $t_x \to t$ with capacity $-b_{t_x}$, for $x \in A, B$. Let C be a cost larger than the cost of any flow in networks A or B; set the cost of $s \to s_A$ to C and $s \to s_B$ to $2C$. The resulting network is effectively equivalent to just network A. Dropping the cost on $s \to S_B$ from $2C$ to 0 makes the resulting network effectively equivalent to B. Solving this incremental change is thus as hard as solving the entire network B.

the results of the study may not generalise to flow scheduling, where any parameter – not just arc costs – can change.

Sensitivity analysis on flow networks seeks to determine how uncertainty in network parameters causes uncertainty in the optimal solution [1, §9.11]. In scientific applications, this may be used to test the robustness of a flow model. Small changes in the input should produce similarly small changes in the output: otherwise, the model is too sensitive to measurement error. Although the goals of sensitivity analysis are considerably different to those of an incremental solver, some of the underlying techniques can be adapted.

3.7.2 Architecture of an incremental algorithm

Previous sections in this chapter described the implementation of pre-existing algorithms. These are designed to solve the minimum-cost flow problem "from scratch": finding an optimal solution without the benefit of any existing state. An obvious incremental approach is to simply rerun one of these algorithms on the updated network, using the previous state as a starting point.

This does not quite work, however: most flow algorithms maintain an invariant on the state of the solver during intermediate computations. After a network update, this invariant may fail to hold. When the invariant is a precondition for the algorithm, as in relaxation, reoptimisation might fail entirely. In other algorithms, such as cost scaling, the invariant is used only to prove complexity bounds. Reoptimisation will succeed, but may run slower than expected.

Consequently, I adopt a three-step approach for my incremental algorithm:

1. Receive changes to the flow network, updating any internal data structures.
2. Manipulate the saved state, to satisfy the invariant under the updated network.
3. Run the algorithm to reoptimise.

In practice, implementation of this approach is considerably more challenging than suggested by this summary. Step 1 superficially appears trivial, but many graph data structures cannot be efficiently modified. Considerable research effort has gone into devising *dynamic* data structures to resolve this problem [64, 20]. In addition to the network itself, many algorithms rely on auxiliary data structures[15], which must also be updated.

Step 2 varies considerably between algorithms, and is discussed in §3.7.5. Somewhat ironically, step 3 is the simplest, with the core of the algorithm remaining mostly unmodified. Since step 2 ensures that the invariant holds, the optimisation procedure can be run as usual after disabling any initialisation code.

[15]For example, many algorithms maintain a set of excess vertices to improve performance.

3.7.3 Choice of algorithms

The approach suggested above could be applied to any flow algorithm considered in this dissertation. In this section, I consider which algorithms are most suited to operating incrementally. I start with cost scaling, which has excellent performance on the from scratch problem (see §3.5). Next, I consider successive shortest path and relaxation (see §3.3 and §3.4). I do not discuss cycle cancelling, as it is considerably slower than the other algorithms.

Cost scaling

Despite excellent performance on the full problem, Goldberg's cost scaling algorithm fares poorly on the incremental problem.

Cost scaling maintains the invariant that $\mathbf{x}$ is a feasible flow and $(\mathbf{x}, \boldsymbol{\pi})$ is ϵ-optimal. This invariant is not required for correctness, but is needed for proofs of its complexity.

The starting state may be neither feasible nor optimal on the new network. To satisfy the invariant, feasibility must first be restored, such as by running a max-flow algorithm[16], which will tend to make the flow less optimal.

Next, the value of ϵ for which $(\mathbf{x}, \boldsymbol{\pi})$ is ϵ-optimal must be calculated. This can be computed by a single pass over the arcs in the residual network. Alternatively, the potential refinement heuristic described in appendix C.4.3 could be used: this might find a smaller value of ϵ by updating the potentials $\boldsymbol{\pi}$.

The key problem is that ϵ may be very large even after a small update. Recall that ϵ-optimality (see definition 3.5.1) is defined as a property satisfied by *every* arc. ϵ is thus a measure of the *maximum* error in the solution. An update as simple as changing the cost on a single arc can make ϵ large, even though the rest of the network remains 0-optimal. Since ϵ often ends up close (in logarithmic terms) to its starting value C, the algorithm performs almost as many iterations of Refine as it would in the full case.

The computational study of cost reoptimisation by Frangioni and Manca [25] supports my assertion. Whereas other algorithms enjoyed speedups of 20× or more, cost scaling had only a 2× speedup. Moreover, its performance was unpredictable, varying considerably even within networks of the same class.

Successive shortest path and relaxation

Both the successive shortest path and relaxation algorithm work by sending flow along *augmenting paths*. These paths start at an excess vertex, and reach an excess

[16] In fact, it is perfectly legitimate to skip this stage and run Refine directly, which will also restore feasibility. However, the analysis is clearer when considering these stages separately.

vertex traversing only arcs of zero or negative reduced cost. Although the two algorithms differ considerably, they have similar incremental properties. In particular, note they both maintain the invariant that $(\mathbf{x}, \boldsymbol{\pi})$ satisfies reduced-cost optimality conditions, seeking to successively improve the feasibility of $\mathbf{x}$[17].

Changes to the network can violate reduced cost optimality, but the pseudoflow can be modified inexpensively to maintain optimality (see §3.7.4 for my method). The algorithm can then be run as usual to reoptimise, restoring feasibility. This takes time proportional to the reduction in feasibility of the solution caused by the updates[18]. This is a very desirable property for flow scheduling, where most changes will only impact the feasibility of a small region of the network.

The relaxation algorithm is evaluated empirically in the study by Frangioni and Manca. As I discussed in §3.4, the practical performance characteristics of the algorithm are heavily dependent on the class of problem. This is apparent from the study: relaxation is competitive on the NETGEN, GRIDGEN and GOTO instances but performs poorly on PDS instances [25, tables 1 to 4]. Relaxation is fast on flow scheduling networks even as a from scratch solver (see figure 4.16b), so there are reasons to be optimistic about its incremental performance.

There are no benchmarks in the literature for reoptimisation using successive shortest path. However, since it belongs to the same class of algorithms as relaxation, I would predict a similar relative speedup.

To conclude, there are substantial theoretical and practical reasons to favour the augmenting path algorithms. Based on this support, I decided to implement incremental versions of successive shortest path and relaxation, described further in §3.7.5.

3.7.4 Maintaining reduced cost optimality

Augmenting path algorithms maintain the invariant that $(\mathbf{x}, \boldsymbol{\pi})$ satisfies reduced cost optimality conditions, that is:

$$\forall (i,j) \in E_{\mathbf{x}} \cdot c_{ij}^{\pi} \geq 0.$$

In from scratch optimisation, both $\mathbf{x}$ and $\boldsymbol{\pi}$ are initialised to $\mathbf{0}$, guaranteeing these conditions hold. When reoptimising, it is the responsibility of the caller to ensure the starting state satisfies the invariant.

[17] This is most clear in the successive shortest path algorithm, which improves the feasibility of $\mathbf{x}$ on every iteration. It is not strictly speaking the case for the relaxation algorithm, since the UpdatePotentials operation may make the flow *less* feasible. But, indirectly, UpdatePotentials is still working towards feasibility since it allows for more AugmentFlow operations to take place.

[18] Note that the feasibility may be reduced directly and indirectly. Direct decreases in feasibility occur as an immediate consequence of a network update: for example, adding a new source and sink vertex. The feasibility may also be compromised indirectly: the operations needed to maintain reduced cost optimality might require pushing flows along arcs, resulting in excesses or deficits at vertices.

In this section, I describe the method I developed to restore reduced cost optimality after the network is updated. Note the pseudoflow $\mathbf{x}$ may become less feasible as a side-effect of this procedure, although I minimise this where possible. The routine only modifies the pseudoflow $\mathbf{x}$: the potential π_i of an existing vertex $i \in V$ is never changed.

Vertex updates

Adding or removing a vertex without arcs cannot violate reduced cost optimality, as $E_{\mathbf{x}}$ is unchanged. Of course, a vertex is added or removed along with its associated arcs, but arc updates are considered in the next section. It follows that no action need be taken on vertex removal. When a vertex i is added, the potential vector $\boldsymbol{\pi}$ must be extended to include an element π_i. The choice of the value π_i is arbitrary: it is initialised to zero in my implementation.

It is also possible to change the supply or demand b_i of an existing vertex $i \in V$. Since this does not change the reduced cost of any arc, it cannot violate the optimality conditions, so no action is taken.

Adding or removing an arc

Note that having an arc $(i, j) \in E$ of zero capacity is equivalent to having no such arc in the network, $(i, j) \notin E$. In both cases, the flow x_{ij} is always zero, and the arc is absent from the residual network: $(i, j) \notin E_{\mathbf{x}} \wedge (j, i) \notin E_{\mathbf{x}}$.

Adding or removing an arc thus reduces to changing the capacity on the arc, covered in the next section. Removing an arc (i, j) is equivalent to decreasing its capacity to 0. The addition of an arc (i, j) with cost c_{ij} and capacity u_{ij} can be modelled as updating increasing the capacity of an arc (i, j) with cost c_{ij} from 0 to u_{ij}.

Updating arc capacity

Changing the capacity of an arc (i, j) from u_{ij} to u'_{ij} does not change c_{ij} or the potentials $\boldsymbol{\pi}$. The reduced cost c^{π}_{ij} therefore remains unchanged. However, it may be necessary to update x_{ij}. The procedure depends on whether the capacity is increasing, or decreasing.

- **Decreasing capacity.** Set $x_{ij} \leftarrow \min\left(x_{ij}, u'_{ij}\right)$ to ensure the capacity constraints of eq. (2.1) continue to hold[19]. The below case analysis shows this preserves reduced cost optimality.

[19] Whilst the primary goal of this method is to preserve reduced cost optimality, a more primitive requirement is that $\mathbf{x}$ continues to be a pseudoflow!

Suppose $(i,j) \notin E_{\mathbf{x}}$: the arc was not present in the residual network prior to the update. Then the arc was previously saturated, with $x_{ij} = u_{ij}$, and $x_{ij} = u'_{ij}$ now holds. Thus the arc remains saturated, and so $(i,j) \notin E_{\mathbf{x}}$ is still true.

Otherwise, $(i,j) \in E_{\mathbf{x}}$. Since reduced cost optimality was satisfied prior to the update, $c^{\pi}_{ij} \geq 0$ must have held. Since c^{π}_{ij} is not changed by the procedure, this will continue to hold. Note that if $x_{ij} \geq u'_{ij}$, then $(i,j) \notin E_{\mathbf{x}}$ after the update, but this is harmless.

- **Increasing capacity.** There are two cases to consider. If $c^{\pi}_{ij} \geq 0$, then no action need be taken. However, if $c^{\pi}_{ij} < 0$ then x_{ij} must be set to u'_{ij}, so that (i,j) remains absent from the residual network.

Updating arc cost

Changing the cost of an arc (i,j) from c_{ij} to $c'_{ij} = c_{ij} + \delta$ changes the reduced cost from c^{π}_{ij} to $c'^{\pi}_{ij} = c^{\pi}_{ij} + \delta$. It is simplest to consider the complementary slackness conditions of eq. (2.10), which are equivalent to the reduced cost conditions of eq. (2.9) by theorem 2.2.6.

1. **Case** $c'^{\pi}_{ij} > 0$**:** set $x_{ij} \leftarrow 0$. Note that this will already be the case if $c^{\pi}_{ij} > 0$.
2. **Case** $c'^{\pi}_{ij} = 0$**:** this case is a no-op. Complementary slackness is satisfied for all values of x_{ij} satisfying the capacity constraints $0 \leq x_{ij} \leq u_{ij}$.
3. **Case** $c'^{\pi}_{ij} < 0$**:** set $x_{ij} \leftarrow u_{ij}$. Note that this will already be the case if $c^{\pi}_{ij} < 0$.

3.7.5 Implementations

I developed three incremental solvers. I adapted my implementations of the successive shortest path and relaxation algorithm, described in §3.3 and §3.4, to support incremental operation. In addition, I extended RELAX-IV [6, 24] — a highly optimised reference implementation of the relaxation algorithm — to operate incrementally.

Extending this project's implementations

It proved relatively straightforward to modify my existing implementations of successive shortest path and the relaxation algorithm. This reflects the emphasis placed on modularity and design flexibility during earlier development.

The method to preserve reduced cost optimality, described in §3.7.4, was implemented using a wrapper class encapsulating graph objects. The wrapper first updates the flow to maintain optimality, and then forwards the changes to the underlying graph object.

Since the wrapper provides the same interface as the graph classes, it can be used as a drop-in replacement. This minimises the changes needed in the rest of the codebase. The algorithms themselves required little modification: I merely added a flag to disable the initialisation routine when operating incrementally.

Extending RELAX-IV

The RELAX-IV solver was considerably more challenging to modify. Whereas this project's implementations attempted to strike a balance between flexibility and efficiency, performance was the overriding objective for RELAX-IV.

The source code is 5,000 lines in total, with some individual functions longer than 800 lines. The original Fortran version of the code is due to Bertsekas, and dates to 1988 [6]. I worked with a C++ port by Frangioni and Gentile from 2003 [24].

Frangioni had already added support for cost reoptimisation for his aforementioned computational study [25]. It appears he started to add support for other forms of updates; however, this development must have been interrupted, since my testing uncovered numerous bugs. The large, legacy codebase made these time consuming to fix. I intend to report the bugs discovered to the authors, making my patches available.

The rest of the changes I made were comparatively straightforward. RELAX-IV allocated data structure memory statically. This is problematic for an incremental solver, as the size of the network is not known in advance. I added support to dynamically resize the data structures, employing the usual strategy of exponential growth in size to achieve memory allocation in amortized linear time[20].

RELAX-IV does not reuse vertex identifiers, causing a memory leak. I modified the algorithm to maintain a set of free vertices. New vertices are allocated an identifier from this set, with the underlying data structures expanded only when there are no free vertices.

Each arc (i, j) is identified by an array index in RELAX-IV. Of course, the flow scheduling system is not privy to this internal index, and can only identify arcs by their source and destination vertices i and j. Finding the index requires scanning the adjacency list of i to find the arc to j (or vice-versa), an $O(n)$ operation. My implementation makes a time-memory tradeoff, maintaining a hash table mapping pairs (i, j) to indices, providing $O(1)$ lookups.

[20] Spikes in scheduling latency are undesirable, even if the average latency remains low. Since the solver is nowhere close to exhausting available memory, my implementation initially allocates twice as much memory as currently needed, in a bid to avoid needing to ever grow the array.

3.8 Extensions to Firmament

The last two sections described my implementation of approximate and incremental solution methods. This section considers their use as part of a flow scheduling platform, Firmament [60, ch. 5]. Integration with Firmament was straightforward: the standard DIMACS format is used to exchange flow networks and their solution [18]. However, I significantly extended Firmament in other ways.

The most substantial change was adding support for incremental solvers. Although Firmament already generated incremental updates to the flow network, the vast majority of such updates were spurious[21]. Parsing the changes took longer than reoptimising! I patched Firmament to generate only authentic changes, considerably improving performance.

I implemented other extensions to assist performance evaluation. Firmament includes a cluster *simulator*, allowing experiments to be conducted on a virtual cluster. However, the simulator was never intended to evaluate flow schedulers, and required substantial modification.

The simulator only supported the Octopus cost model, a simplistic policy which merely balances the load across machines (see §4.2.2). I implemented the Quincy cost model, described in §2.2.2, to provide a more realistic basis for comparison. This required building entirely new simulated components, such as a distributed filesystem, described further in §4.2.3.

I made various other minor extensions: for example, building a statistics module to record scheduling latency and other properties of the cluster, such as the number of running tasks. Moreover, I added numerous configuration options to the simulator: for example, allowing the number of machines in the cluster to be varied.

3.9 Summary

This chapter began by summarising several standard flow algorithms which were implemented in this project. Next, I turned my attention to how these algorithms may be extended to improve performance on the flow scheduling problem. In §3.6, I described how I modified the cost scaling algorithm to build an approximate solver, allowing accuracy to be traded for speed. In §3.7, I modified the successive shortest path and relaxation algorithm to operate incrementally. As far as I am aware, both implementations are the first of their kind.

I concluded with a summary of the extensions I made to the Firmament flow scheduling system, including adding support for the Quincy cost model. In the next chapter, I demonstrate the considerable speedup that results from using the solvers developed in this project with Firmament.

[21] For example, arc change events were generated for every arc from a task to an unscheduled aggregator, irrespective of whether any parameters had changed.

Chapter 4

Evaluation

The project was highly successful. All success criteria set forth in the project proposal (see appendix E) have been met, and many optional extensions implemented, as shown by table 4.1. The incremental solver achieves sub-second scheduling latency on a 12,000 machine cluster, with a 14.5× speedup over state-of-the-art reference implementations. Approximation yielded a small performance gain on flow scheduling, but significantly underperformed the incremental solver. However, it achieved an over 10× speedup on other classes of flow networks, suggesting it may be profitably employed in other domains.

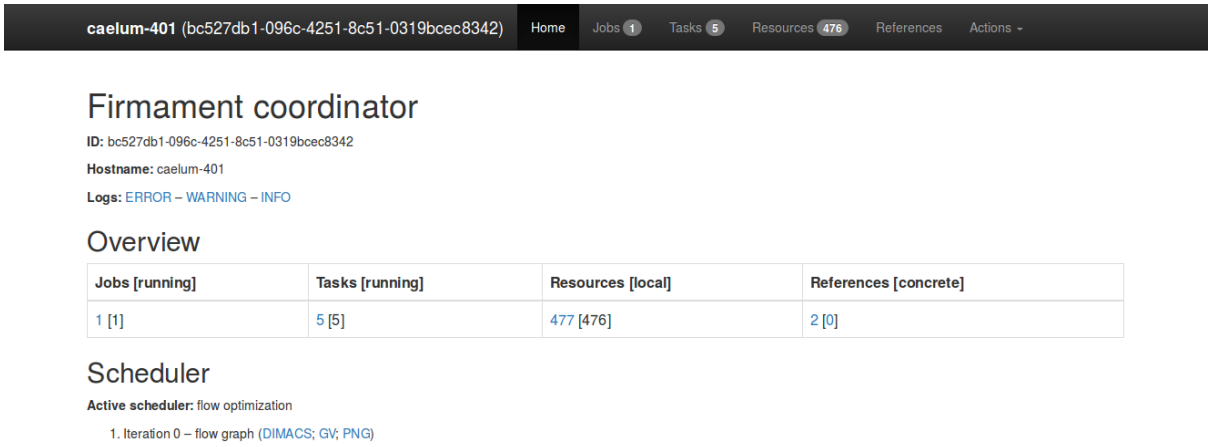
4.1 Correctness testing

I developed unit tests for each component, which were re-run as a regression test at the end of each development cycle in accordance with the spiral model (see §2.3.2). Standard unit test frameworks are not designed to work with large, external datasets such as flow networks. Accordingly, I developed my own test harness in Python to automate verification of flow algorithms. I wrote unit tests for smaller components, such as data structures, using the GTest framework (see appendix A).

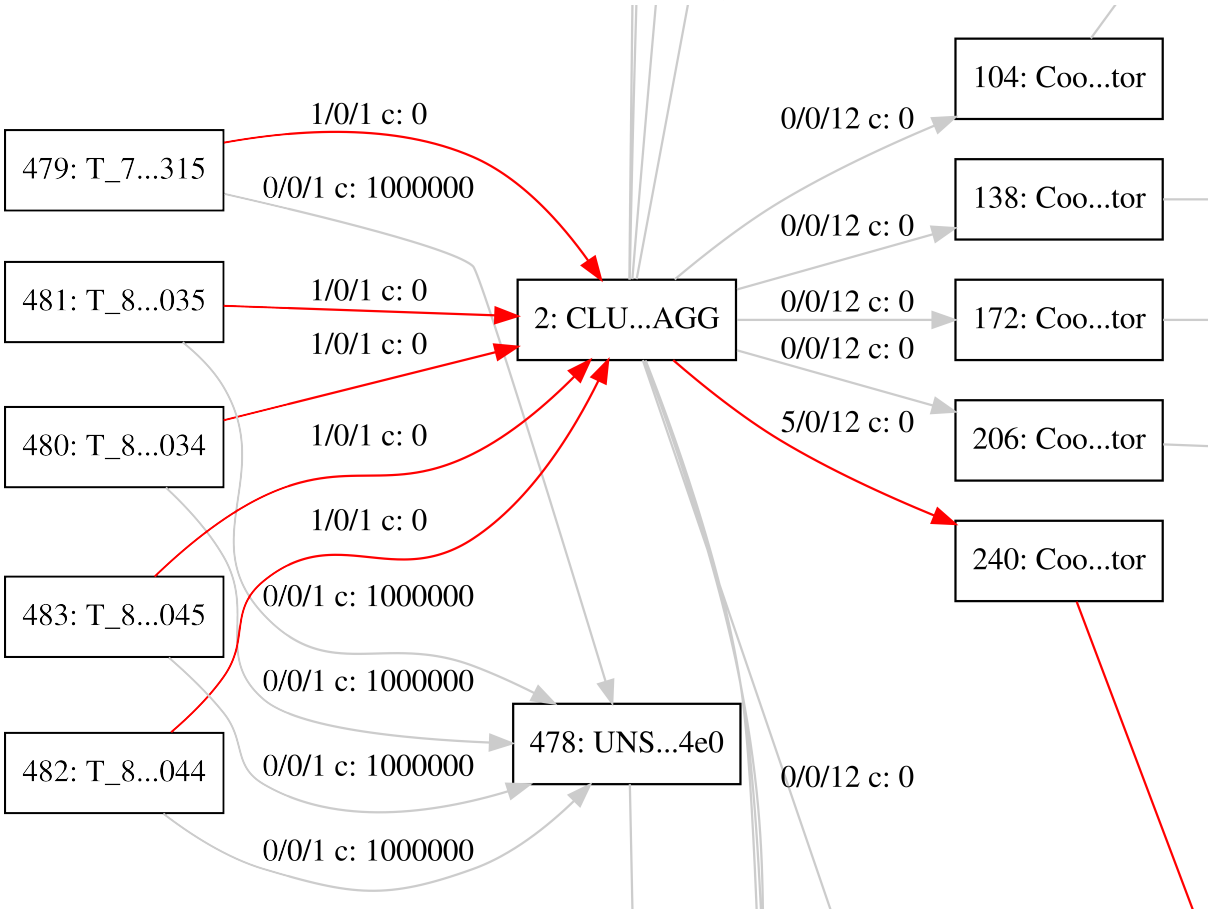
Hapi is intended to used as part of a larger cluster scheduling system. To demonstrate this capability, I integrated Hapi's solvers integrated into the Firmament system. Figure 4.1 shows the system in action.

4.2 Performance testing strategy

The goal of this project is to improve the performance of flow schedulers: consequently, benchmarking forms the majority of this evaluation. The dataset used in tests is described in §4.2.1 to §4.2.3. Finally, §4.2.3 outlines the methodology used for the experiments.



(a) An overview from the web user interface for Firmament.



*(b) An extract of a flow graph, showing the scheduling of a small number of tasks (left hand side). Arcs are **red** if they carry flow, grey otherwise.*

Figure 4.1: Firmament scheduling tasks using Hapi, on a cluster of 14 machines.

#	Deliverable	Section
	Success criteria	
S1	Implement standard flow algorithms	3.2 to 3.5
S2	Design an approximate solver	3.6
S3	Integrate system with Firmament	4.1
S4	Develop a benchmark suite	Appendix D.2
	Optional extensions	
E1	Design an incremental solver	3.7
E2	Build cluster simulator	3.8
E3	Optimise algorithm implementations	4.3

Table 4.1: Summary of work completed: deliverables for the project, with the section describing their implementation.

4.2.1 Simulating a Google cluster

This project sought to develop algorithms which achieve sub-second scheduling latency on warehouse-scale computers, comprising many thousands of compute nodes. Unfortunately, as such a cluster was not available to me, I used simulation to evaluate Hapi.

Firmament includes a cluster simulator, which I modified extensively to support this project (see §3.8). To ensure the test is realistic, the simulator replays a trace of events from a production Google cluster of 12,000 machines [68, 57, 56]. This is by far the most detailed trace released by any major cluster operator, enabling an accurate simulation.

However, the trace is not perfect. To protect commercially sensitive information, much of the data was obfuscated by Google before release. Moreover, some data was simply never recorded: for example, the input file sizes of tasks are missing.

Although there is sufficient information to reconstruct the topology of the flow network, some cost models require data that is absent from the trace. The next two sections describe the cost models I used, and how the limitations of the dataset were overcome.

4.2.2 Octopus cost model

The Octopus model implements a simple load balancing policy. Tasks and machines are assumed to be homogeneous. The cost of arcs into a machine are proportional to the load on that machine, resulting in tasks being preferentially scheduled on idle machines.

This policy is too simplistic to be used in production. However, it serves as a useful baseline for comparison. Producing flow networks that are easy to solve compared to more realistic cost models, it provides a lower bound on the real-world scheduling latency that can be achieved.

4.2.3 Quincy cost model

Flow scheduling was pioneered by the Quincy system, developed at Microsoft Research [43]. To enable a direct comparison between the two systems, I implemented the Quincy cost model in Firmament (see §2.2.2 and §3.8). Firmament does not currently support preemption, which required a slight modification to the cost model[1]. Otherwise, the model is as described in the original paper.

Quincy seeks to optimise for data locality, scheduling tasks close to where their input data is stored, producing flow network that are considerably more complex than those of the Octopus cost model. The cost of an arc from a task to a machine is proportional to the network bandwidth which would be consumed were the task to be placed there. Network bandwidth consumption is in turn dependent upon the location of the task's input data in the cluster. Such detailed information is not present in the trace, and so must be estimated.

Upon starting the simulator, a virtual distributed filesystem is built. Each machine is given 6 TB of simulated storage space, in line with the specifications of machines in contemporary Google clusters [21]. Each file is replicated across three machines, in accordance with industry standards. Files are divided into 64 MB blocks, as in the Google File System [27].

A collection of files is generated to saturate the available storage capacity[2], with file sizes randomly sampled from a distribution. Unfortunately, Google has not released any information on the file size distribution observed inside the cluster. Instead, I used data from a Facebook cluster to estimate a distribution [12].

When a task is submitted, an estimate of its input size S is made. A set of input files is then assigned to the task, by randomly sampling from the distributed filesystem until the cumulative size reaches S. Google has not released any information on the input size distribution. For consistency, I have used data from the same Facebook cluster to estimate a distribution [12].

However, it is possible to do better than simply randomly sampling from this distribution. The Google cluster trace contains information about the runtime of individual tasks, and there is a correlation between runtime and input size. Consequently, I have taken the approach of computing the cumulative probability of

[1] I would expect the runtime of all solvers to be larger with preemption enabled, as there would be more arcs in the network. However, I would not anticipate changes in the relative performance of algorithms.

[2] In practice, a cluster's disk utilisation would not actually be full, but this distinction is unimportant for my simulation.

Name	Percentage of Google cluster	Number of machines
Small	1%	120
Medium	5%	600
Large	25%	3000
Warehouse-scale	100%	12,000

Table 4.2: Cluster sizes used in benchmarks

the runtime for each submitted task, assigning an input size of the same cumulative probability. So, for example, a task of median runtime will be assigned a median input file size.

I believe that the simulation represents a realistic workload (although not the exact Google workload), which could plausibly occur in a production cluster today. The distributions used are provided in appendix D.4.

The exact performance of the algorithms, of course, depends on the architecture and workload of individual clusters. Most metrics used in this chapter therefore measure *relative* performance when comparing different algorithms or optimisations. Absolute runtimes are occasionally reported: while indicative of real-world performance, care should be taken not to generalise beyond the experiments conducted.

4.2.4 Evaluation methodology

I took care to minimise experimental error throughout the evaluation process. However, error cannot be completely eliminated: some variation in runtime is inevitable when executing on a time-sharing operating system. Confidence intervals were computed using Student's t-distribution to quantify the error, which in most cases is negligible.

Each test was executed multiple times, with the runtimes across test runs aggregated to increase precision. Five iterations were performed by default, with more test runs being performed if the confidence interval was too wide.

Significant effort was made to minimise variations in performance between runs. Tests took place on machines dedicated to the experiment, ensuring no interference from other workloads. All machines have the same specification (see appendix D.1), allowing for absolute runtimes to be compared between experiments.

The algorithms were evaluated at multiple scales, ranging from small clusters that might be used within a private company to warehouse-scale computers used to support large cloud applications. Table 4.2 summarises the sizes used throughout the rest of the chapter.

Running the experiments manually would be error-prone, as well as extremely time consuming. I therefore developed a test harness in Python to automate the process,

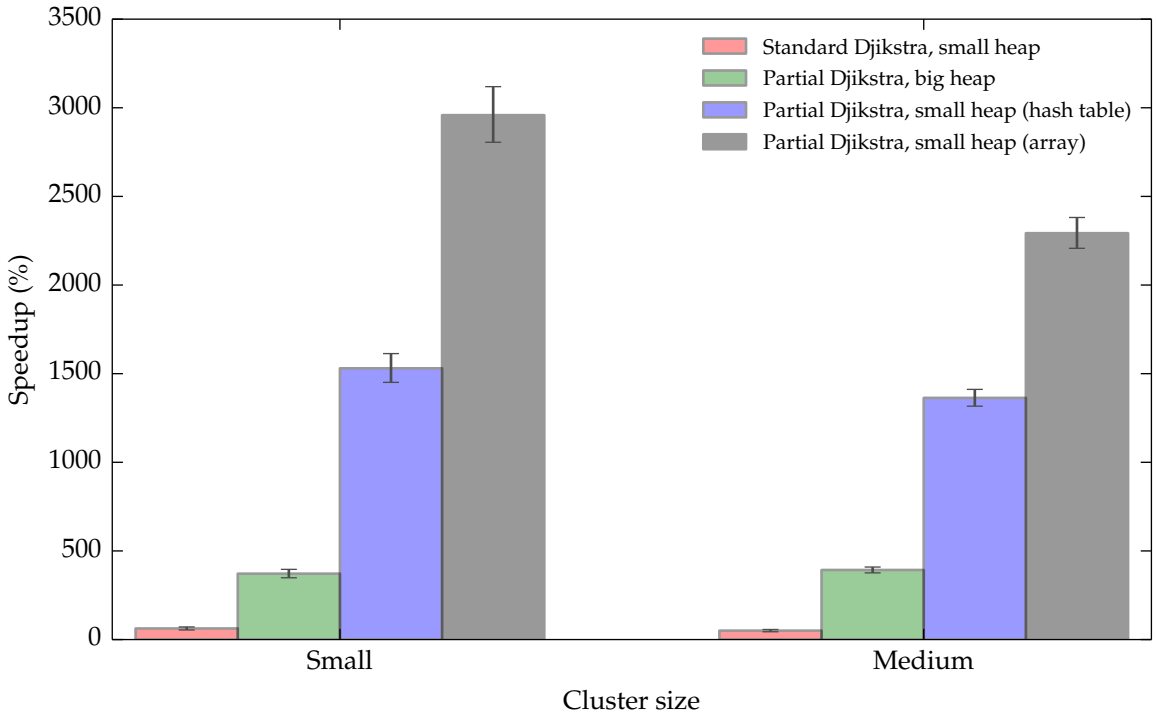


Figure 4.2: Optimisations for successive shortest path: speedup relative to the naïve implementation with standard Djikstra and big heap. The bars represent the mean speedup over 10 replications, with error bars indicating the 95% confidence interval.

described further in appendix D.2. This ensures the above methodology is followed for all tests.

4.3 Optimisations

This project is primarily concerned with algorithmic improvements. However, implementation decisions can have a considerable impact on the performance of flow solvers. In this section, I evaluate the optimisations applied to each implementation. In addition, I describe how I choose the parameters for each algorithm, and evaluate the impact of compiler choice on performance.

4.3.1 Successive shortest path algorithm

As explained in §3.3.2, it is possible to terminate Djikstra's algorithm as soon as it finds a shortest path to a deficit vertex, without needing to compute shortest paths to all vertices in the graph. This so-called *partial Djikstra* approach yields considerable performance gains. As figure 4.2 shows, it provides a speedup of over 4×.

This inspired a related optimisation, *small heap*. Djikstra's algorithm works by maintaining a priority queue of vertices, implemented using a binary heap in this

project (see §3.3.2 for justification of this choice). This results in a $\Theta(\lg l)$ complexity for priority queue operations, where l is the length of the queue.

In the *big heap* implementation, the heap is initialised to contain all vertices in the graph. *Small heap*, by contrast, is initialised containing just the source vertex. Vertices are inserted into the queue as they are encountered during graph traversal.

Initialising a heap with n elements has cost $\Theta(n)$; by contrast, inserting the elements one at a time has a cost of $\Theta(n \lg n)$. Big heap, therefore, has a lower initialisation cost — at least under standard Djikstra, where each vertex is eventually inserted into the queue. However, the number of elements l in the heap is larger throughout the lifetime of big heap, making each operation more expensive.

When using standard Djikstra, small heap provides a very small (but statistically significant) speedup. However, a substantial performance gain is achieved when using small heap in conjunction with partial Djikstra. This is because with early termination, the typical length l of the queue is much smaller than the number of vertices n in the graph. As figure 4.2 shows, partial Djikstra with small heap provides a speedup of around $5\times$ over partial Djikstra with big heap, and over $20\times$ compared with the original implementation.

To support decreasing the key of elements in the priority queue (a common operation in Djikstra's algorithm), it is necessary to maintain a *reverse index* mapping vertex IDs to elements in the priority queue. In the big heap, this was implemented using an *array* of length n. For the small heap, it is possible to reduce memory consumption by using a *hash table*, storing mappings only for vertices v present in the priority queue.

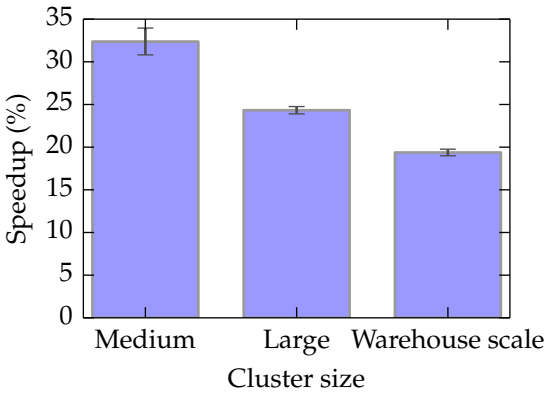
Both the array and hash table offer $O(1)$ lookup. The array might be expected to be faster: there is no need to compute a hash function, and the result is always returned after a single memory read. However, the hash table may be faster for large heaps: it can be more efficiently cached, since there is less "dead space" than in the array.

Yet, figure 4.2 clearly show that the array implementation outperforms the hash table version. The array may never be sufficiently large for capacity cache misses to become a problem with the 15 MB last level cache (LLC) on the evaluation machines.

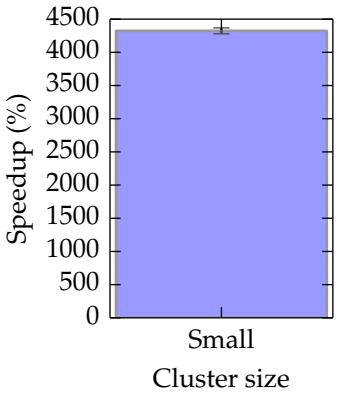
4.3.2 Relaxation algorithm

The relaxation algorithm repeatedly accesses the set of arcs $(S, \overline{S})$ crossing the cut. The main relaxation procedure in algorithm 3.4.1 and AugmentFlow in algorithm 3.4.3 only consider arcs $(i, j) \in (S, \overline{S})$ for which the reduced cost c_{ij}^{π} is zero. By contrast, UpdatePotentials in algorithm 3.4.2 considers only those arcs with positive reduced cost.

In the naïve implementation of the algorithm, the entire list of arcs in the graph is searched until an arc satisfying the above conditions is found. It is possible to instead maintain a cache of arcs $(S, \overline{S})$ crossing the cut. This imposes an additional



(a) Quincy cost model.



(b) Octopus cost model.

Figure 4.3: Optimisations for relaxation: speedup from maintaining cache of arcs crossing the cut. The bars represent the mean speedup over 30 replications, with error bars indicating the 95% confidence interval.

cost whenever S is updated, but allows for iteration over the set to proceed more efficiently. At no additional cost, we can partition this cache into two sets: those arcs with zero reduced cost and those with positive reduced cost[3]. This is more efficient, as the algorithm never needs to access the entire set $(S, \overline{S})$.

The effect of this optimisation varies considerably between flow networks, although it yielded a positive speedup in all experiments performed. Under the Quincy cost model, there is only a modest speedup of just over 20%. By contrast, in the Octopus cost model arc caching yields a more than 40× speedup[4].

4.3.3 Cost scaling

Wave vs FIFO

Two versions of Goldberg's cost scaling algorithm are described in §3.5.3, *"wave"* and *"FIFO"*. The "wave" implementation has time complexity $O\left(n^3 \lg(nC)\right)$ compared to a $O\left(n^2 m \lg(nC)\right)$ running time for "FIFO". However, for flow scheduling networks the asymptotic time complexities are the same, as $m = O(n)$ by lemma 2.2.2.

Figure 4.4 shows that "FIFO" considerably outperforms "wave" on this problem instance, with a speedup of at least 20×. I suspect this is because the "FIFO" queue contains only active vertices (see algorithm 3.5.4). By contrast, in "wave" each iteration involves a pass over every vertex in the graph (see algorithm 3.5.6), imposing a significant overhead.

[3]Note, by the reduced cost optimality conditions, no arcs in the residual network ever have negative reduced cost.

[4]Results are given only for the small dataset, as the tests timed out on larger network.

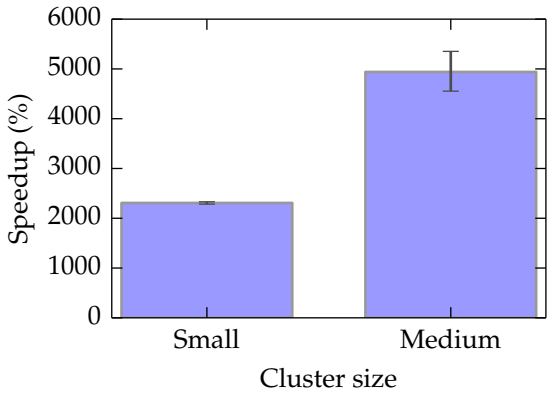
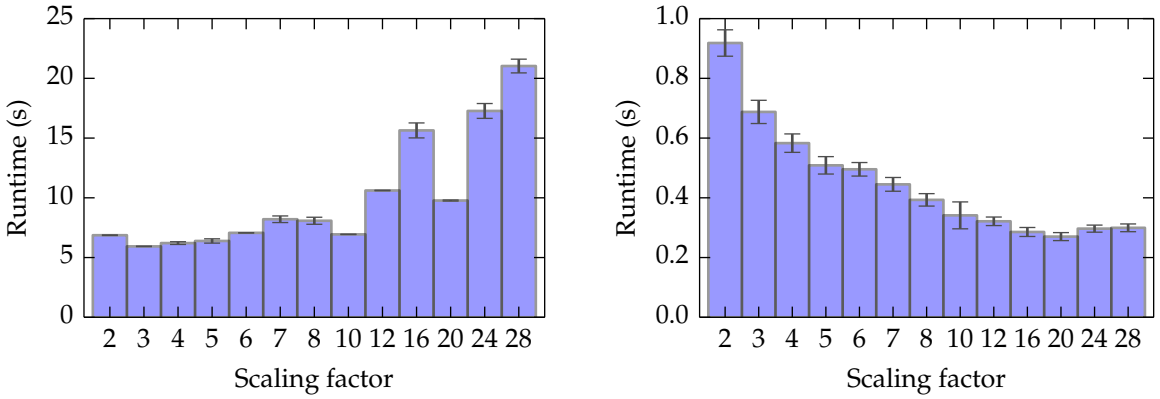


Figure 4.4: Optimisations for cost scaling: speedup from using a FIFO vertex queue, rather than the wave topological ordering of vertices. The bars represent the mean speedup over 10 replications, with error bars indicating the 95% confidence interval.



(a) My implementation.

(b) Goldberg's CS2 implementation [28, 40].

Figure 4.5: Parameter selection for cost scaling: choosing a scaling factor α. Experiment performed on the medium (600 machine) dataset. The bars represent the mean runtime over 5 replications, with error bars indicating the 95% confidence interval.

Moreover, the larger the size of the network, the larger the speedup enjoyed under "FIFO". This suggests that, although "wave" may have a tighter bound in general, "FIFO" may have a better asymptotic bound on flow scheduling networks.

Scaling factor

Cost scaling computes a series of ϵ-optimal solutions, where ϵ is reduced by a *scaling factor* α between calls to REFINE. A value of $\alpha = 2$ was proposed in the original paper by Goldberg and Tarjan [28], and is used in algorithm 3.5.1. However, this choice is not necessarily optimal. Larger values of α will result in fewer calls to REFINE, but each call will take longer to execute. The optimal choice for α varies depending on the problem instance [29, §7].

Level	Description
`-O0`	No optimisation.
`-O1`	Optimise to reduce code size and execution time. Do not perform optimisations which could greatly increase compilation time.
`-O2`	Perform optimisations which do not involve a space-speed tradeoff.
`-O3`	Optimise even more, possibly at the expense of increased binary size.

Table 4.3: Optimisation levels supported by `gcc` *and* `clang`.

Figure 4.5 shows how the runtime varies by scaling factor. The experiment was conducted on both my version of the algorithm, and Goldberg's reference implementation, CS2 [40].

My implementation performs best with a small scaling factor, between 2 and 6. The runtime is very similar within this range, although $\alpha = 3$ offers a small speedup. Performance significantly degrades at $\alpha = 12$ and above.

Goldberg's solver exhibits the opposite behaviour, enjoying considerable performance gains with larger scaling factors. There are diminishing returns after around $\alpha = 12$, with performance starting to degrade from around $\alpha = 24$.

I believe this behaviour is due to Goldberg's extensive use of heuristics (see appendix C.4.3) in his implementation. Although these improve the overall performance of the algorithm, they add overhead to each execution of Refine. This favours larger values of α, which cause fewer calls to Refine.

In subsequent tests, a scaling factor of $\alpha = 3$ is used for my implementation and $\alpha = 20$ for Goldberg's. These offer small, but statistically significant speedups, over the default cost scaling factors of 2 (my implementation) and 12 (Goldberg's implementation).

4.3.4 Compilers

Optimising compilers apply transformations which preserve the semantic meaning of the program, while (typically) improving performance. To isolate the impact of compiler choice and optimisations, I compared two optimising compilers: the GNU C Compiler (GCC) and Clang. (see table 4.3). Higher levels tend to produce faster code, but this is not guaranteed. Some aggressive optimisations — such as inlining functions — may actually harm performance. Consequently, I also compared the different optimisation levels.

Figure 4.6 shows there is a dramatic improvement from applying the lowest level of optimisations, `-O1`. GCC's `-O1` setting is fairly aggressive, achieving performance comparable to `-O2`. Clang's is more relaxed, but their performance at `-O2` and `-O3` are both comparable.

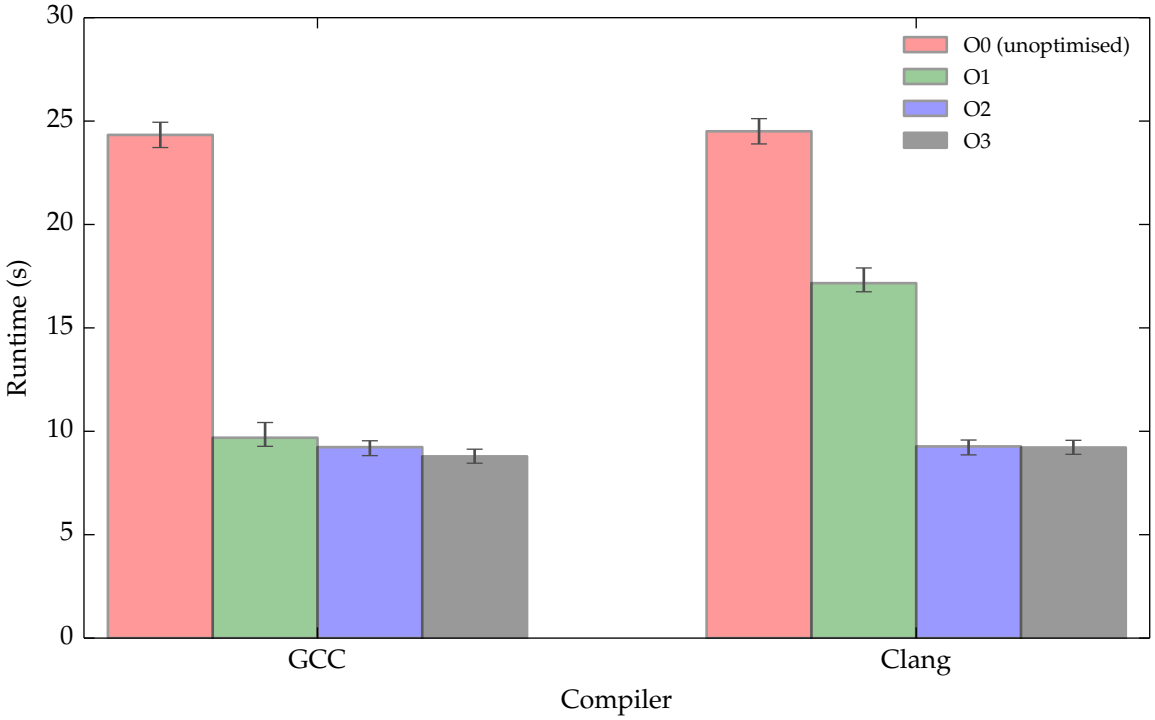


Figure 4.6: Choosing a compiler for cost scaling: runtime on GCC and Clang at varying optimisation levels. Experiment conducted on medium (600 machine) cluster. The bars represent the mean runtime over 5 replications, with error bars indicating the 95% confidence interval.

This test was repeated for all other implementations in the project. The complete results are given in appendix D.3, but are similar to figure 4.6. GCC `-O3` is used in all subsequent tests: it is always competitive with Clang `-O3`, and sometimes has a slight performance edge.

4.4 Approximation algorithm

Unlike with other algorithms, evaluation of approximate solution methods (see §3.6) must consider not just their speed, but also their accuracy[5]. To complicate matters further, both the speed and accuracy of the approximation algorithm vary depending on the terminating condition chosen.

I introduce an *oracle* policy to ease interpretation of the results. The oracle terminates the algorithm as soon as an intermediate solution is found that reaches the target accuracy. Of course, an oracle policy is unrealistic: the entire challenge of approximation is that the minimum cost is not known *a priori*. The oracle policy therefore provides an upper bound on the speedup that can possibly be achieved by approximation.

When evaluating the heuristic policies (see §3.6.2), I split each dataset into training and test data in a 1:3 ratio. The parameters are chosen from the training data in order to achieve a target accuracy A at least a proportion p of the time. In the tests below, both A and p were set to 99%. Naturally, relaxing these constraints allows for a greater speedup.

The accuracy and speed of the algorithm are evaluated by running it on the (unseen) test data, with the parameters chosen in the previous step. This split of test and training data guards against overfitting.

All tests were conducted using my implementation of cost scaling, described in §3.5. Since the experiments in this section measure runtime *relative* to the same algorithm used as an optimal solver, I believe the results should generalise to other implementations of cost scaling.

There is one notable exception to this. The oracle policy achieves a speedup even at 100% accuracy in these experiments, since optimality is often reached when $\epsilon > 1/n$. The *price refinement* heuristic (see appendix C.4.3), present in highly optimised implementations such as Goldberg's CS2 [40], is able to detect such early optimality. Implementations with price refinement therefore achieve the same performance as the oracle model at 100% accuracy. Further work is required to determine if any other differences exist.

[5]Defined to be the ratio of the cost of an optimal solution, to the cost of the solution returned.

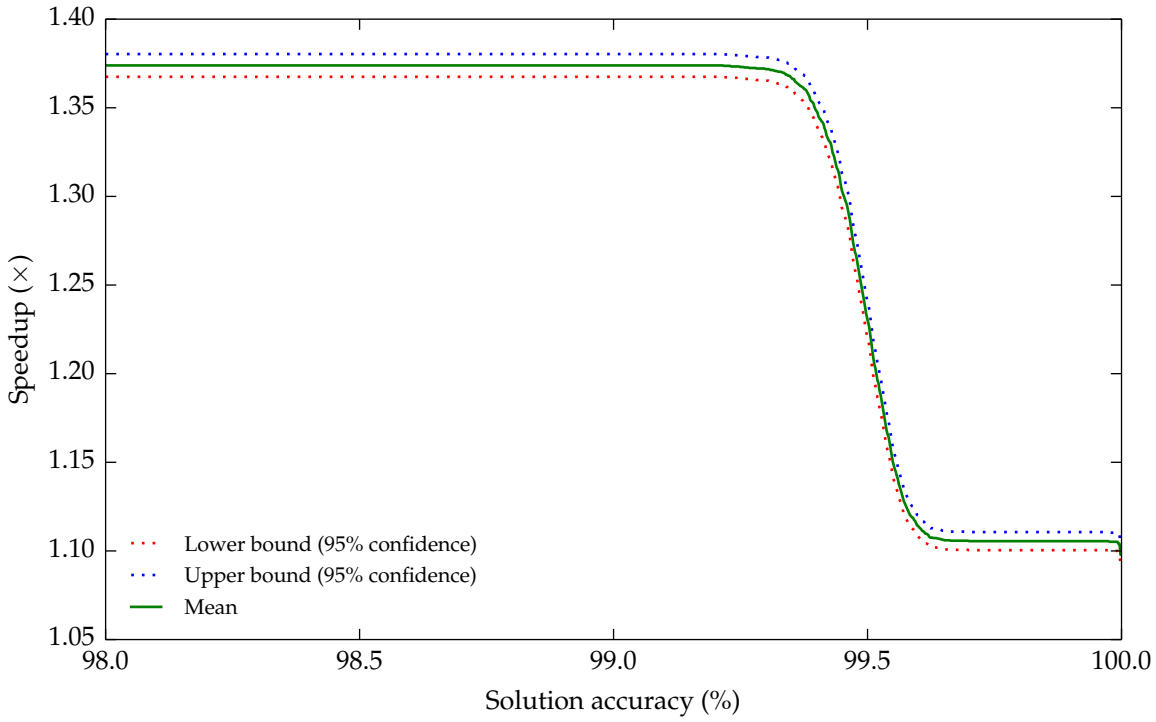


Figure 4.7: Upper bound on speedup from approximation on flow scheduling networks, computed using the oracle policy. The experiment was conducted over 1,000 independently generated networks.

4.4.1 Performance on flow scheduling

Evaluation took place on flow networks produced by the cluster simulator under the Quincy cost model (see §4.2). A total of 1,000 different networks were produced by varying the seed used to initialise the random number generator. This had the effect of changing the generated simulated filesystem, and the input files associated with each task, with consequent changes to costs and preference arcs.

The flow networks produced by the above method are snapshots of the cluster at the beginning of the trace. An alternative would be to take snapshots from multiple points in time, evaluating the algorithm on the time series. However, many tasks run for the entire duration of the trace. Consequently, there is significant correlation between flow networks in the time series. As a result, the training and test data could be very similar when taking snapshots at points in time.

By contrast, networks produced by varying the seed are considerably more dissimilar, providing a more challenging dataset.

Figure 4.7 shows the maximum speedup that can be achieved by approximation, using the oracle policy. At the 99% target accuracy, this gives an upper bound of just over 135%. The confidence intervals are tight, never exceeding 2%, showing remarkable consistency across the dataset.

I implemented two heuristics, *cost convergence* and *task assignment convergence* (see

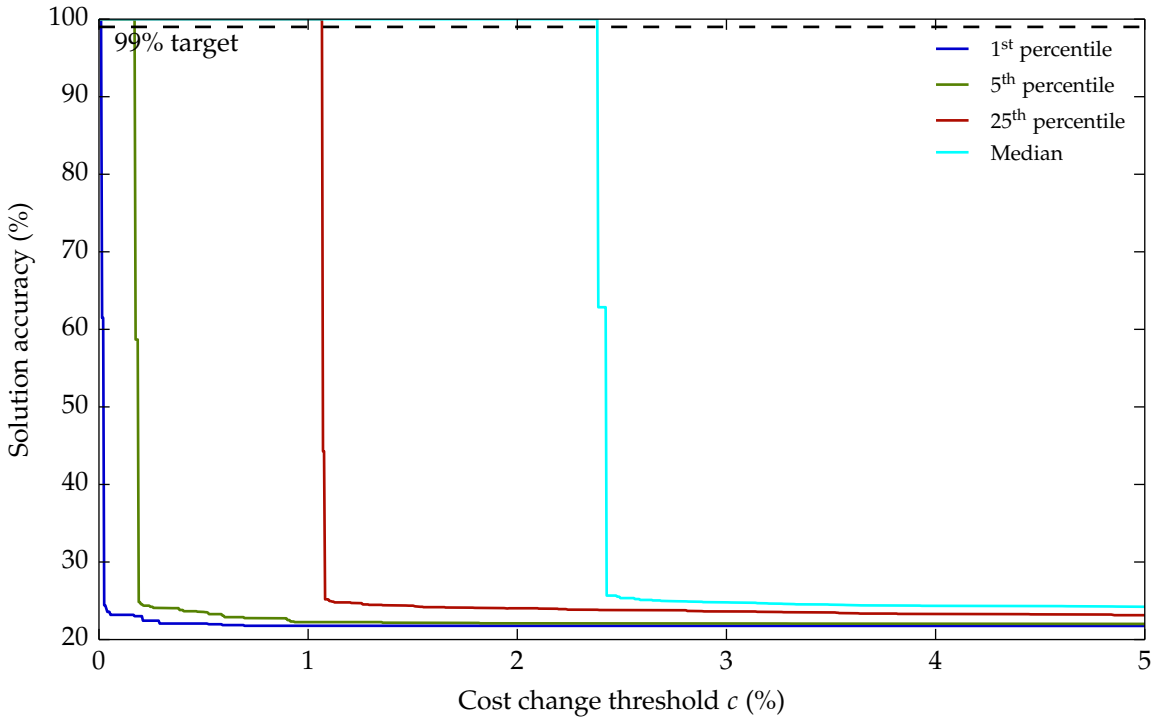


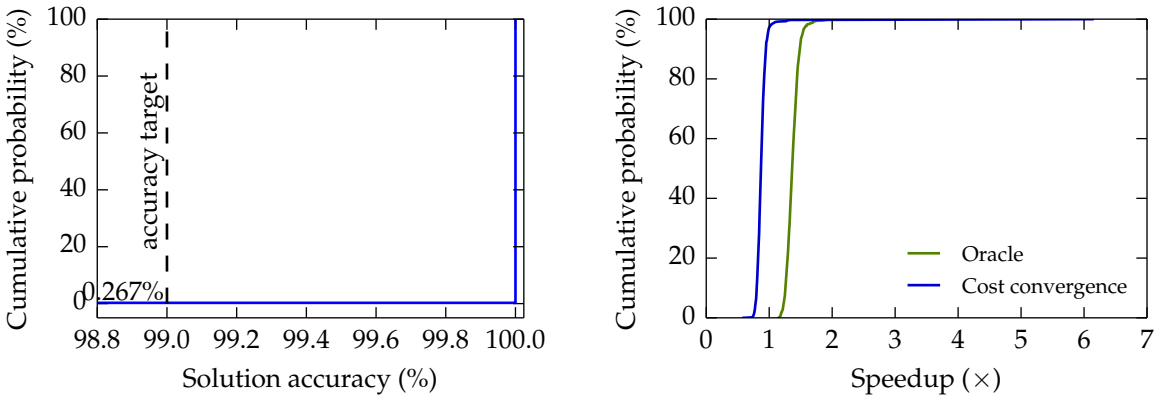
Figure 4.8: Parameter choice for cost convergence on flow scheduling networks: selecting a cost threshold c. Cost convergence was applied, for various parameters c, to each of the 250 networks in the training set. The solution accuracy measures how close the resulting solutions were to optimality. c was selected to be 0.01%*, where the* ***1st percentile*** *cuts the 99% target accuracy line.*

§3.6.2). The subsequent sections evaluate how close these heuristic policies come to achieving the bound set by the oracle policy.

Cost convergence

Under cost convergence, the algorithm terminates when the cost changes by less than a factor c. Figure 4.8 shows the accuracy achieved at various percentiles for different parameters. The **1st percentile** line, representing the accuracy which is met or exceeded 99% of the time in the training set, drops precipitously at the left of the graph. Consequently, c is set to a very low value of 0.01%. This is not encouraging: such a low threshold will tend to give a correspondingly low speedup.

Figure 4.9a shows that 100% accuracy is achieved in 749 out of 750 cases in the test set. The remaining case, however, has a disappointing 25% accuracy (off the axis of the graph). There is a significant speedup in all cases, shown by figure 4.9b, but performance is noticeably behind that of the oracle model. This is unsurprising: since optimality is reached in most cases, the algorithm has performed more work than was necessary.



(a) Accuracy CDF. *(b) Speedup CDF.*

Figure 4.9: Performance of cost convergence policy on flow scheduling networks. Results computed over the 750 networks in the test set.

Task migration convergence

The task migration convergence heuristic measures the number of tasks that have been reassigned ('migrated') between machines, terminating when it is less than a threshold t. Although it was hoped that this domain-specific heuristic might fare better than the more general cost convergence policy, figure 4.10 is not encouraging.

Although the number of task migrations falls somewhat as the algorithm converges, it remains high throughout execution. In fact, tasks are migrated even once optimality is reached (but while $\epsilon > 1$)! Tasks with only a single input file have preference arcs with equal cost to each machine that has a replica of the file. These tasks may be migrated freely between machines in their preference set, without changing the cost of the overall solution.

This is reflected in the fact that accuracy in figure 4.10 only drops at a relatively large migration threshold, of over 6000. When it does fall, it does so precipitously, indicating the heuristic is highly sensitive to parameter setting.

The performance shown in figure 4.10 is equally disappointing. A similar accuracy pattern emerges in figure 4.11a as for cost convergence (*cf.* figure 4.9a). The speedup CDF in figure 4.11b is still worse. In the vast majority of cases, *no* speedup occurs: the algorithm does not terminate even once optimality is reached, as tasks continue to be migrated. A handful of cases enjoyed a considerable speedup, but these correspond to the 1.6% of cases which failed to meet the 99% accuracy target. Hence, these can hardly be counted as a success.

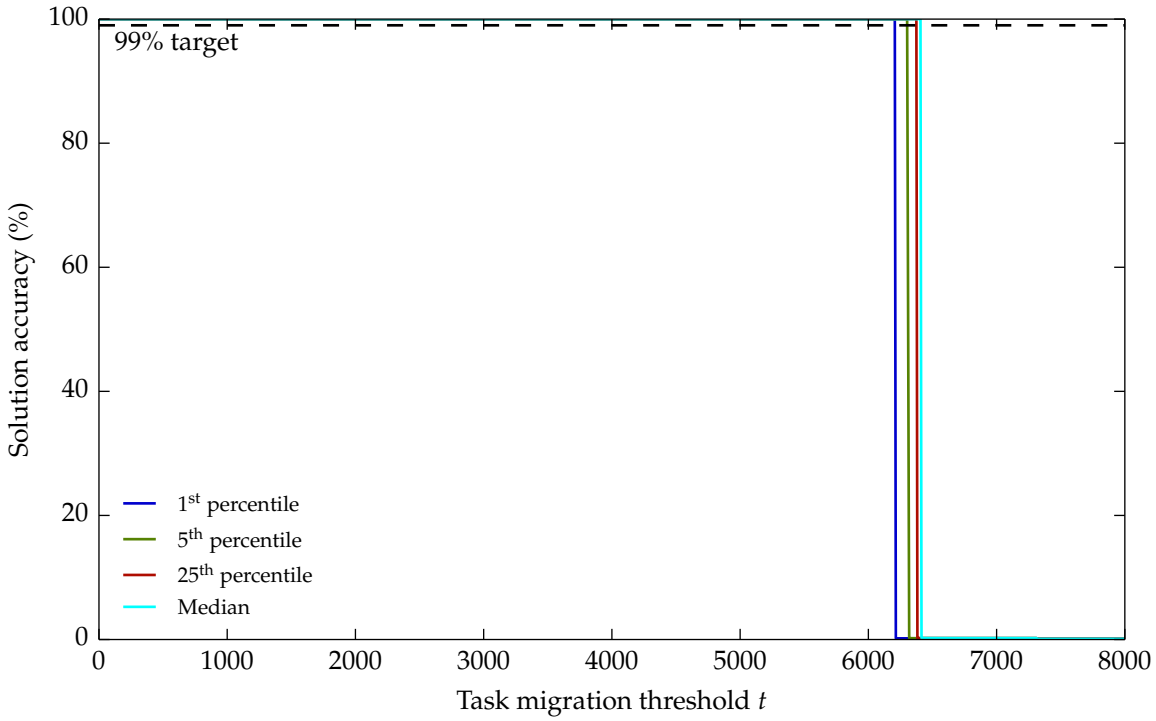
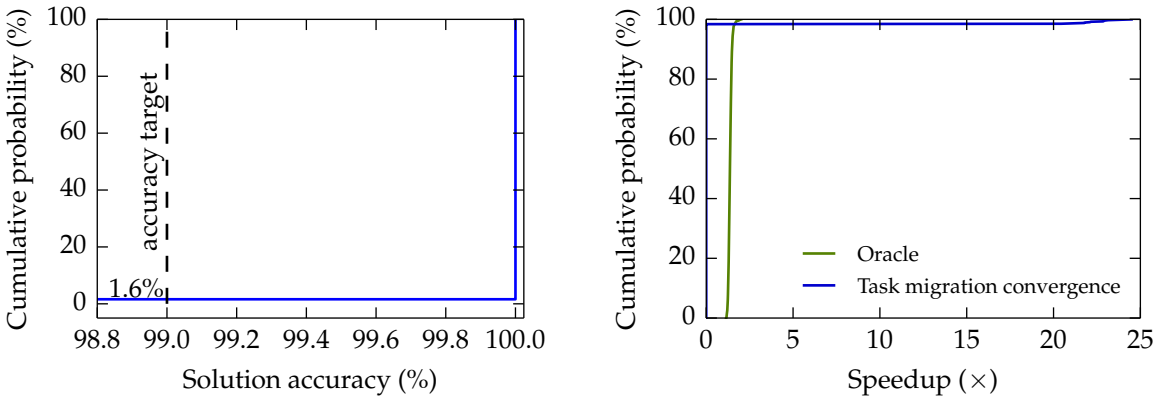


*Figure 4.10: Parameter choice for task migration convergence on flow scheduling networks: selecting a task migration threshold t. As with figure 4.8, the policy was applied to the 250 networks in the training set. A threshold $t = 6206$ is chosen, where the **1st percentile** cuts the 99% target accuracy line.*



(a) Accuracy CDF. *(b) Speedup CDF.*

Figure 4.11: Performance of task migration policy on flow scheduling networks. Results computed over the 750 networks in the test set.

Dataset	Description
NETGEN-8	Sparse networks, with an average outdegree of 8. Arc capacities and costs are uniformly random from $[1, 1000]$ and $[1, 10000]$. The number of supply and demand vertices is $\sqrt{n}$, with an average per-vertex supply of 1,000.
NETGEN-SR	Dense networks, with an average outdegree of $\sqrt{n}$. Other parameters are the same as NETGEN-8.
NETGEN-LO-8 NETGEN-LO-SR	Like NETGEN-8 and NETGEN-LO-SR, except the average per-vertex supply is 10. The arc capacities are therefore only "loose" bounds.
GOTO-8	Sparse networks, with an average outdegree of 8. Maximum arc capacity of 1,000, and cost of 10,000.
GOTO-SR	Dense networks, with an average outdegree of $\sqrt{n}$. Other parameters are the same as GOTO-8.

Table 4.4: Non-scheduling datasets used for benchmarking the approximation algorithm. These parameter choices for NETGEN and GOTO are due to Király and Kovács [47].

Conclusion

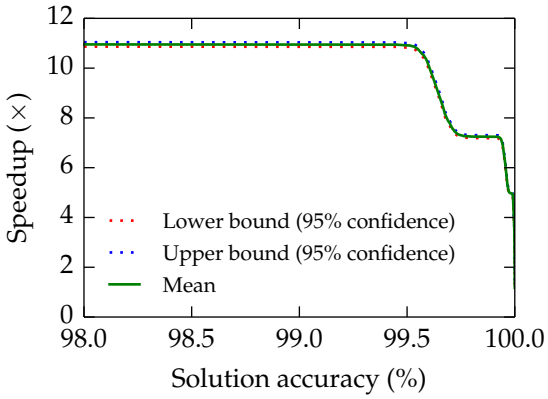
Approximation algorithms do not appear to be a viable means for improving the performance of flow schedulers. The heuristics implemented in this project perform poorly under the Quincy cost model. It is possible that more sophisticated heuristics might improve upon this. However, the oracle policy gives a 135% upper bound on the speedup. Since order of magnitude performance gains are required to make flow scheduling scale to very large clusters, approximation is not a promising approach for this application.

4.4.2 Performance in other applications

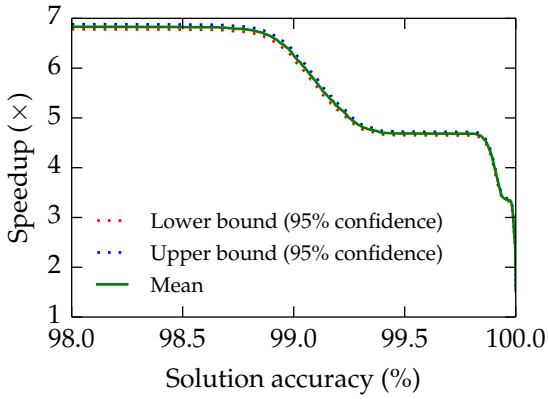
The method of approximation that I developed can be applied to any flow network. Although its performance was disappointing on flow scheduling networks, it turns out to provide a considerable performance boost on other types of network. To illustrate this, I employ two network generators, NETGEN [49] and GOTO [31]. These have a long history of use in computational benchmarks, having been employed in the first DIMACS Implementation Challenge [45].

Table 4.4 summarises the datasets used in this test. 1,000 networks were generated of each type, by varying the random seed. 250 of these networks were used for training, with the remaining 750 being test data.

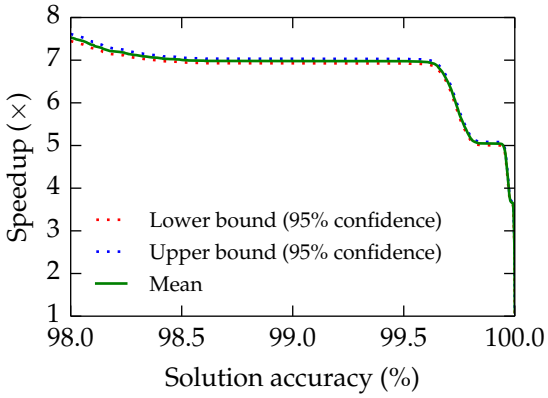
Figure 4.12 shows the maximum speedup that can be achieved on each class of networks. Note that a considerably greater speedup is possible on networks produced by NETGEN, in the first two rows, than those of GOTO, on the last row. This highlights that performance is highly dependent on the class of flow networks. The tightly constrained networks, in the top row, enjoy more of a speedup than the loosely constrained ones, in the second row. The speedup on the dense network NETGEN-



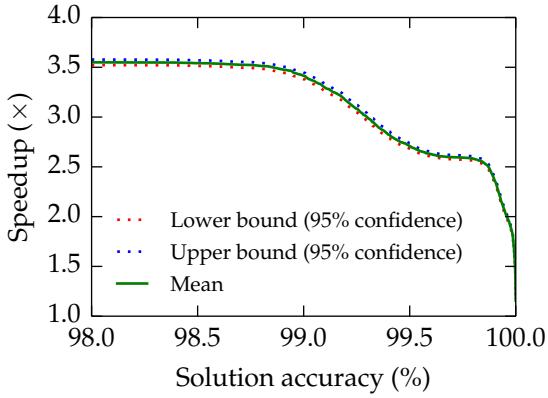
(a) NETGEN-8.



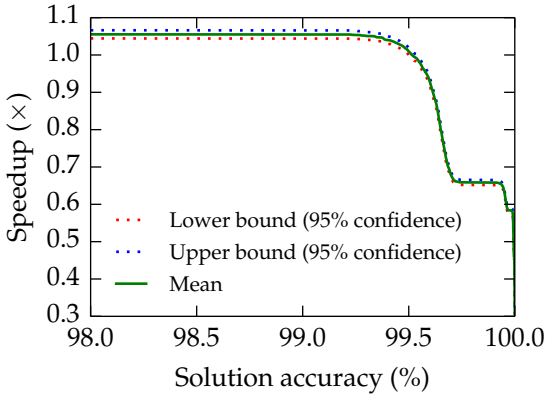
(b) NETGEN-LO-8.



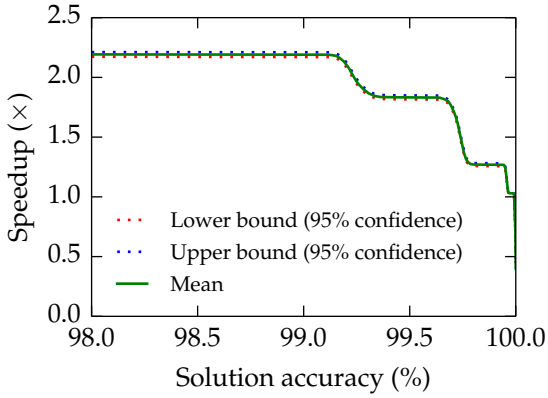
(c) NETGEN-SR.



(d) NETGEN-LO-SR.



(e) GOTO-8.



(f) GOTO-SR.

Figure 4.12: Upper bound on speedup from approximation on general flow networks, computed using the oracle policy. The experiment was conducted over 1,000 independently generated networks.

Dataset	Proportion of solutions meeting target accuracy	Mean accuracy	Worst-case accuracy
NETGEN-8	98.3%	99.6%	96.9%
NETGEN-SR	98.8%	99.7%	98.3%
NETGEN-LO-8	100.0%	99.9%	99.1%
NETGEN-LO-SR	99.3%	99.9%	98.0%
GOTO-8	98.5%	98.9%	45.3%
GOTO-SR	99.0%	99.1%	27.3%

Table 4.5: Accuracy of approximation algorithm on general flow networks. The second column shows the proportion of networks where the 99% target accuracy was met or exceeded. The third and fourth column show the mean and minimum accuracy achieved.

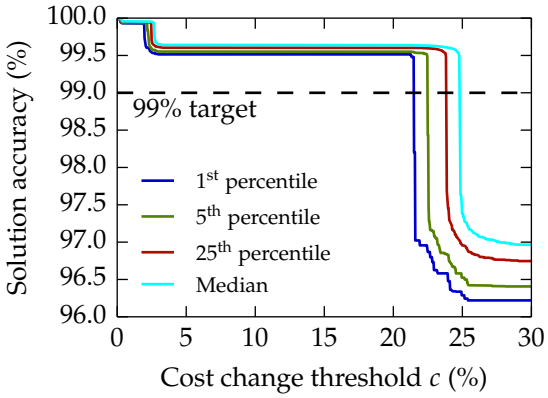
SR is greater than for the sparse network NETGEN-8, but this pattern is reversed for GOTO-SR and GOTO-8.

The relationship between the threshold c for the cost convergence heuristic and accuracy is shown in figure 4.13. For NETGEN-8 and NETGEN-SR, there is a gentle decline in accuracy as c is increased, indicating that the heuristic performance is robust to small changes in c. However, accuracy on the other classes of networks is very sensitive to c, falling rapidly after a certain point.

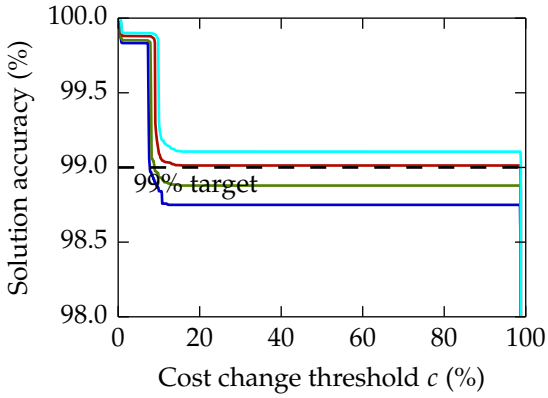
The accuracy under cost convergence is shown in figure 4.14, along with the value of c chosen by the previous step. For each class of networks, the 99% target accuracy is achieved in at least 98% of cases, as shown in table 4.5. The accuracy distribution on GOTO datasets has a long tail, with accuracy below 50% in the worst case. The heuristic is considerably more reliable on NETGEN: the worst-case accuracy across any NETGEN instance is 96.9%, which is still close to optimal.

Figure 4.15 shows the speedup achieved by cost convergence, along with the upper bound provided by the oracle. For the majority of networks (NETGEN-8, NETGEN-SR, GOTO-8 and GOTO-SR), cost convergence comes very close to achieving this upper bound. Indeed, the oracle model is almost totally obscured by the cost convergence model in the graphs on the left-hand side.

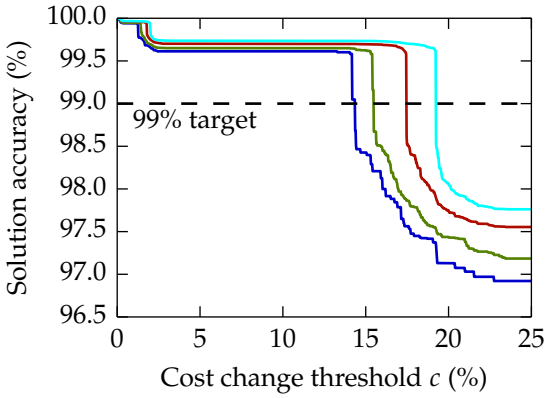
Cost convergence also provides a robust speedup on the "loose" NETGEN instances, NETGEN-LO-SR and NETGEN-LO-8, of 268% and 469% respectively. However, this is significantly less than the maximum achievable speedup of 343% and 622%. The reason for this is clearly visible from figures 4.13b and 4.13d. The 25^{th} percentile and **median** lines remain above the target accuracy line until $c > 80\%$. However, to ensure that this target accuracy is reliably achieved, c is set much more conservatively to a value of $c < 10\%$. This conservative choice is also reflected in figures 4.14b and 4.14d: the accuracy CDF is shifted further to the right than for other graphs, with almost all solutions achieving an accuracy in excess of 99.8%.



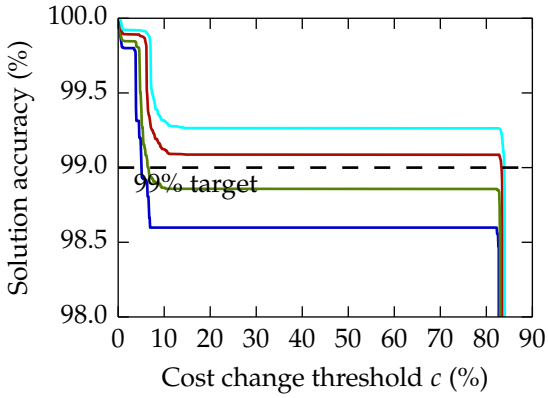
(a) NETGEN-8.



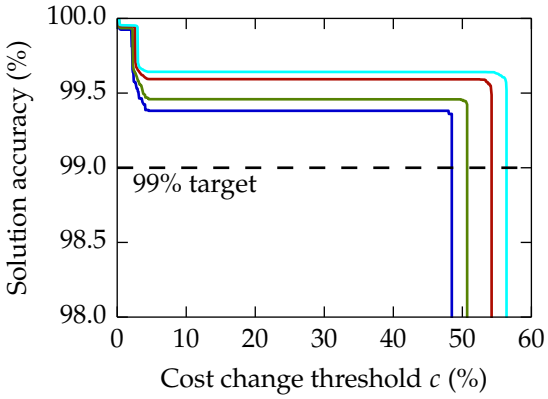
(b) NETGEN-LO-8.



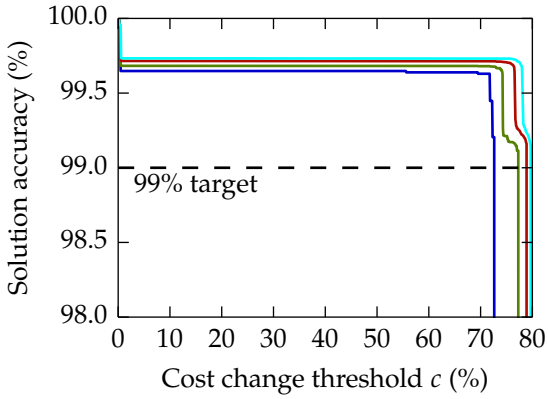
(c) NETGEN-SR.



(d) NETGEN-LO-SR.

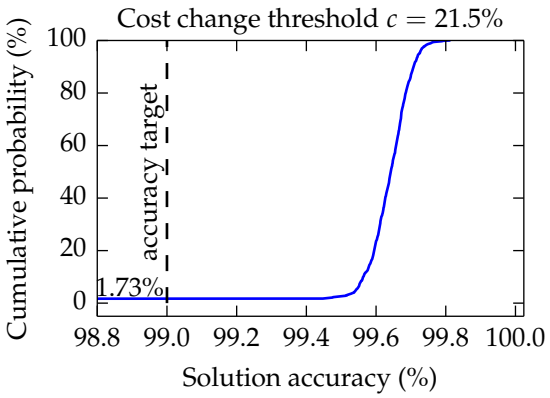


(e) GOTO-8.

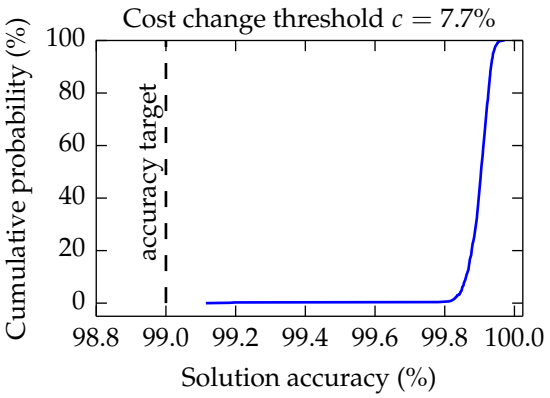


(f) GOTO-SR.

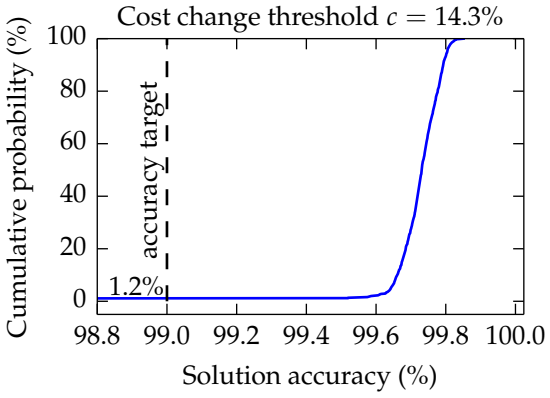
*Figure 4.13: Parameter choice for cost convergence on general flow networks: selecting a cost threshold c. Cost convergence was applied, for various parameters c, to each of the 250 networks in the training set. The solution accuracy measures how close the resulting solutions were to optimality. Key: **1st percentile**, **5th percentile**, **25th percentile**, **median**.*



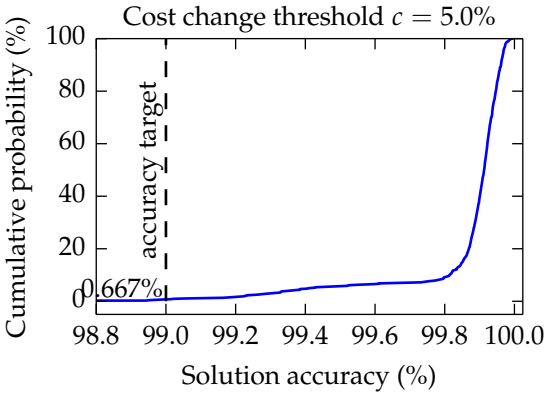
(a) NETGEN-8.



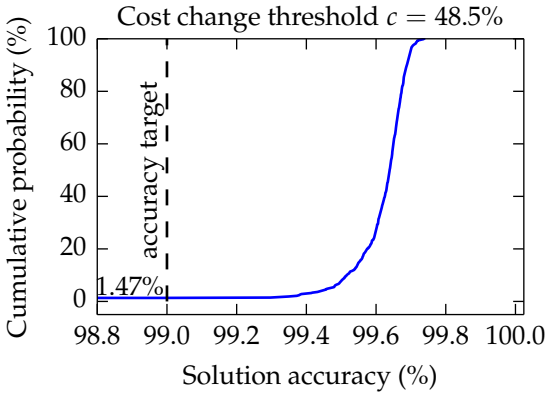
(b) NETGEN-LO-8.



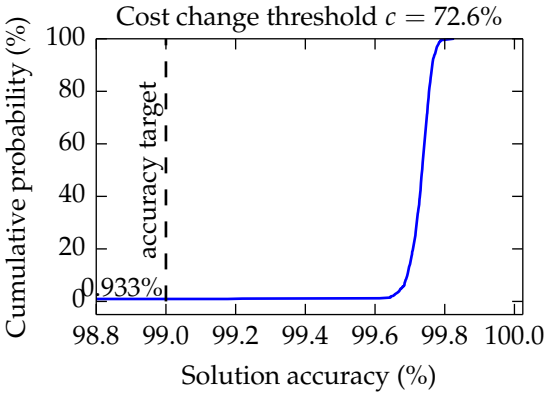
(c) NETGEN-SR.



(d) NETGEN-LO-SR.

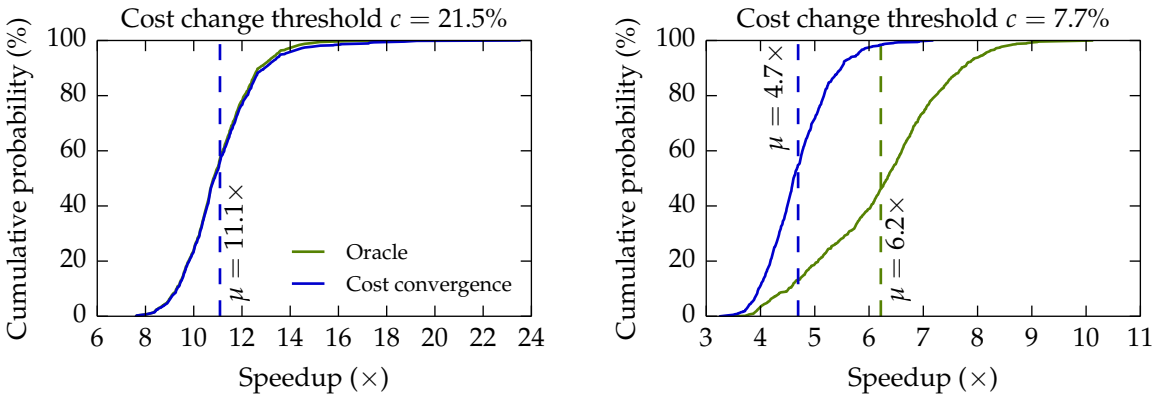


(e) GOTO-8.



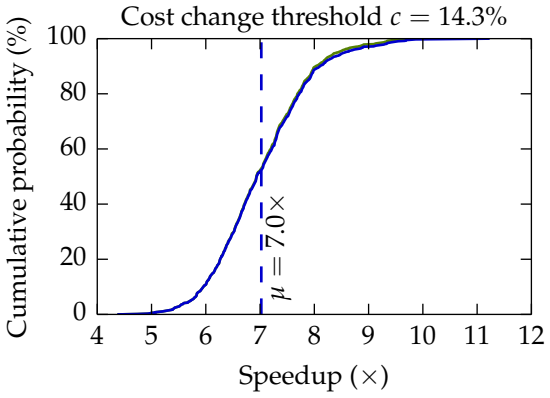
(f) GOTO-SR.

Figure 4.14: Accuracy of cost convergence policy on general flow networks. CDFs computed over the 750 networks in the test set.

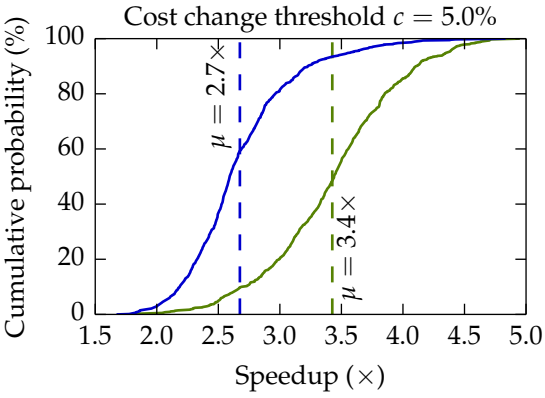


(a) NETGEN-8

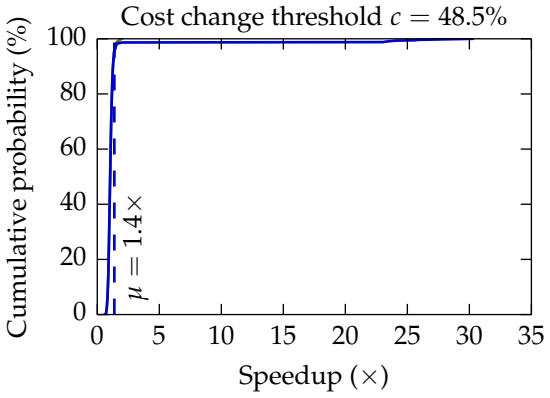
(b) NETGEN-LO-8



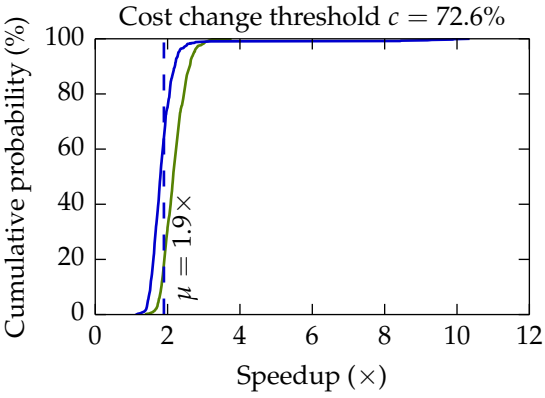
(c) NETGEN-SR



(d) NETGEN-LO-SR



(e) GOTO-8



(f) GOTO-SR

*Figure 4.15: Speedup of approximation algorithm on general flow networks. CDFs compare **cost convergence** to the upper bound given by the **oracle policy**.*

4.5 Incremental algorithm

Incremental solution methods (see §3.7) operate on sequences of related flow networks. Their performance is evaluated in this section by measuring the *scheduling latency* on the Google cluster trace (see §4.2). The simulation was run for an hour of cluster time[6], and used the Quincy cost model.

Time is divided into *scheduling rounds*. When an event (such as task submission) is received during an ongoing scheduling round, the event is added to a queue. A new scheduling round is started as soon as the old round completes, if any events are pending. If the queue is empty, the simulator waits until a new event is received before starting a scheduling round.

The scheduling latency for round n is defined to be the *longest* time any event spent waiting in the queue during scheduling round $n-1$, plus the runtime of the flow algorithm on scheduling round n. This represents the performance that would be observed by users in a real application.

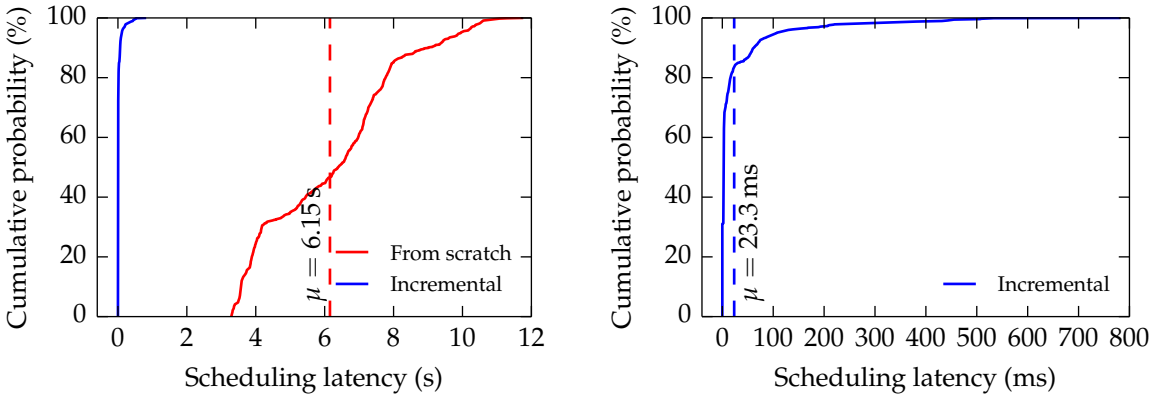
This approach is necessary: unlike algorithms that run from scratch, the runtime of incremental algorithms depends on the number of changes to the flow network. Decreasing the runtime of an incremental algorithm will therefore tend to produce an even greater decrease in the scheduling latency, since the number of pending events in the queue is smaller. Conversely, small increases in the runtime of incremental solvers are amplified when measuring scheduling latency.

The next section compares the performance of the successive shortest path (see §3.3) and relaxation (see §3.4) algorithms operating incrementally and as a from scratch solver. This overstates the true performance gains somewhat, however, as cost scaling (see §3.5) outperforms both algorithms when operating from scratch. The second section therefore compares my implementation of cost scaling to the incremental approach. Finally, I compare the performance of my incremental solution methods with those of highly optimised reference implementations of from scratch solvers.

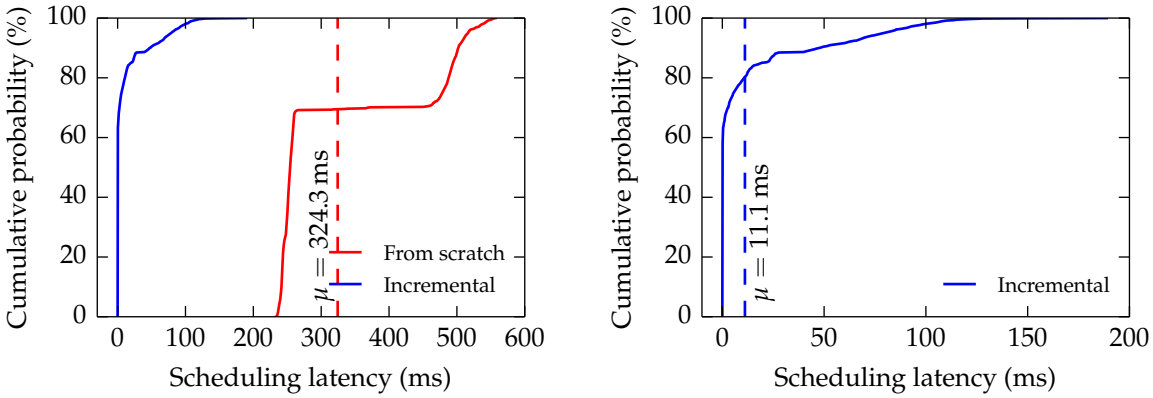
4.5.1 Relative performance

Figure 4.16 shows a dramatic speedup from using the augmenting path class of algorithms (successive shortest path and relaxation) in an incremental mode of operation. For successive shortest path (SSP), scheduling takes at most 800 ms in the incremental case, compared to a best case of over 3 s in the from scratch case. The difference is even more pronounced when considering the *mean* scheduling latency, which is 6.15 s in the from scratch case but a mere 23.3 ms for the incremental solver: a speedup of over 260×.

[6] Although a longer simulation time is possible, the workload in the Google trace does not vary substantially so this would be of little benefit.

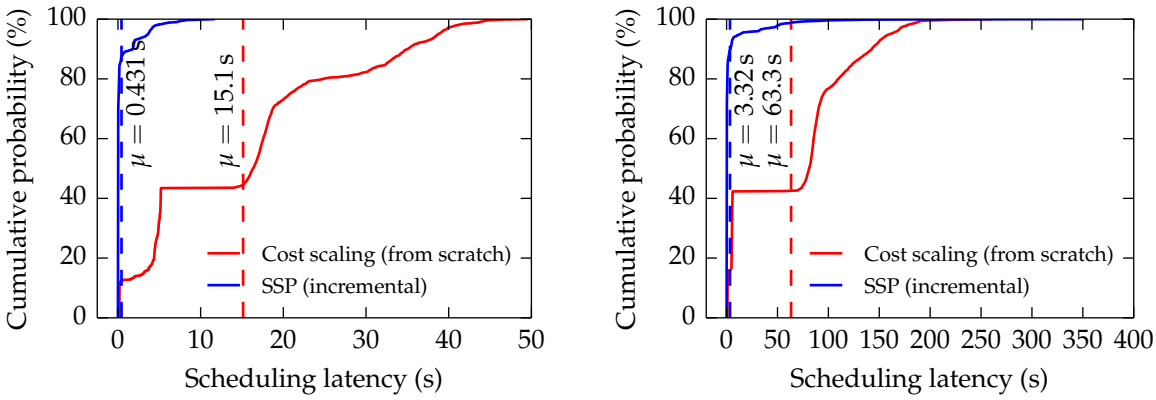


(a) Successive shortest path on medium (600 machine) cluster.



(b) My implementation of relaxation on large (3,000 machine) cluster.

Figure 4.16: Performance in incremental vs from scratch mode. The CDFs in the left column compare the same implementation operating in different modes; those in the right column zoom in on the incremental mode. All experiments were replicated 5 times.



(a) Large cluster (3,000 machines). *(b) Full Google cluster (12,000 machines).*

Figure 4.17: CDF showing performance of incremental successive shortest path (SSP) vs from scratch cost scaling. Experiments were replicated 5 times.

For relaxation, the incremental solver enjoys a speedup of 29×. Although this is a smaller speedup than enjoyed by SSP, incremental relaxation is in fact substantially faster: its mean scheduling latency is less than half that of incremental SSP, despite a 5× larger test dataset. From scratch relaxation is also substantially faster[7] than from scratch SSP, which explains the comparatively low speedup.

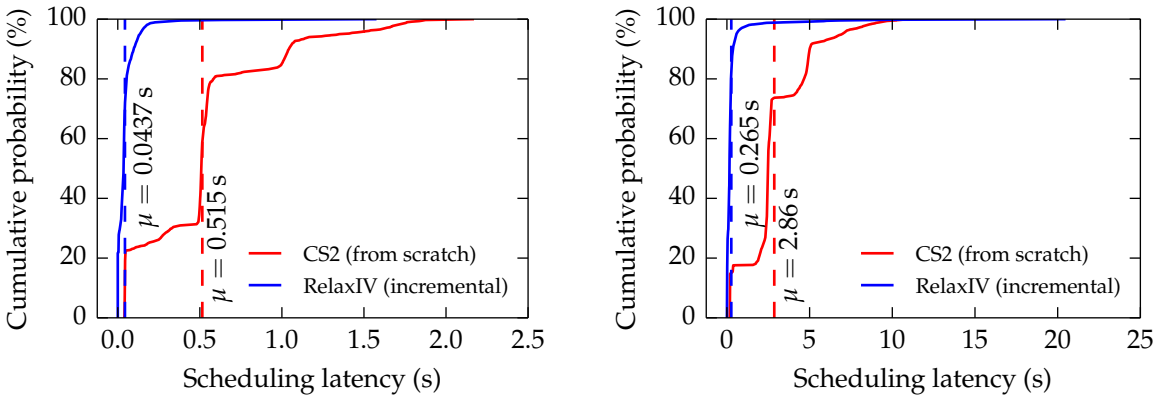
Both incremental solvers have a long tail. This is a reflection of the distribution of events in the cluster trace. In most scheduling rounds, there are only a handful of changes, and the incremental solver returns quickly. Occasionally, however, thousands of events may occur in a small time window: for example, when a large job is submitted. In this case, reoptimisation may take a while, although it is still faster than solving from scratch.

4.5.2 Best-in-class comparison

The previous section showed incremental augmenting path algorithms significantly outperformed their corresponding from scratch versions. This is encouraging, but as both successive shortest path and relaxation are comparatively slow when running from scratch, there is a possibility that a different from scratch algorithm might outperform my incremental solvers.

Figure 4.17 compares incremental SSP to cost scaling (see §3.5), a competitive from scratch algorithm. As predicted, the speedup achieved is less than those reported in the previous section. However, it is still highly substantial: I find my incremental approach substantially outperforms cost scaling, delivering an 18× speedup on the full (12,000 machine) Google cluster.

[7] In fact, it is almost as fast as cost scaling for this class of flow network.



(a) Large cluster (3,000 machines). *(b) Full Google cluster (12,000 machines).*

Figure 4.18: CDF showing performance of my incremental relaxation algorithm against optimised reference implementation of from scratch cost scaling. Experiments were replicated 5 times.

4.5.3 Comparative evaluation

Researchers have spent considerable time developing optimised implementations of flow algorithms. Since the focus of this project is on developing improved algorithmic techniques, I have spent little time optimising my implementations. Although I have attempted to ensure that all implementations in this project are optimised to an equal degree, there is a risk that the results in the previous section do not generalise to state-of-the-art reference implementations.

To investigate this, I extended the reference implementation of the relaxation algorithm, RELAX-IV [24], to support operating incrementally (see §3.7.5). For the from scratch solver, I selected Goldberg's heavily optimised implementation of his cost scaling algorithm, CS2 [40]. This has been found to be one of the fastest flow solvers in independent benchmarks [47], corroborated by my own tests.

Quincy also used CS2: Isard *et al.* report a scheduling latency of ~1 s on a 2,500 machine simulated cluster [43]. Figure 4.18a shows the results of my experiment on a similarly sized cluster, in which CS2 achieves a scheduling latency of ~500 ms. My incremental solver is faster at every percentile, achieving a 12× average speedup.

I went beyond the evaluation of Isard *et al.* by performing the experiment on the full (12,000 machine) Google cluster. Figure 4.18b shows that my incremental solver delivers an average speedup of 11×. This is comparable to the speedup on the 3,000 machine cluster, suggesting my algorithm scales as well as CS2. Most tasks are scheduled without any perceptible delay: the mean scheduling latency is just 265 ms. In fact, sub-second scheduling latency is achieved in 97.4% of cases, as shown in figure 4.19.

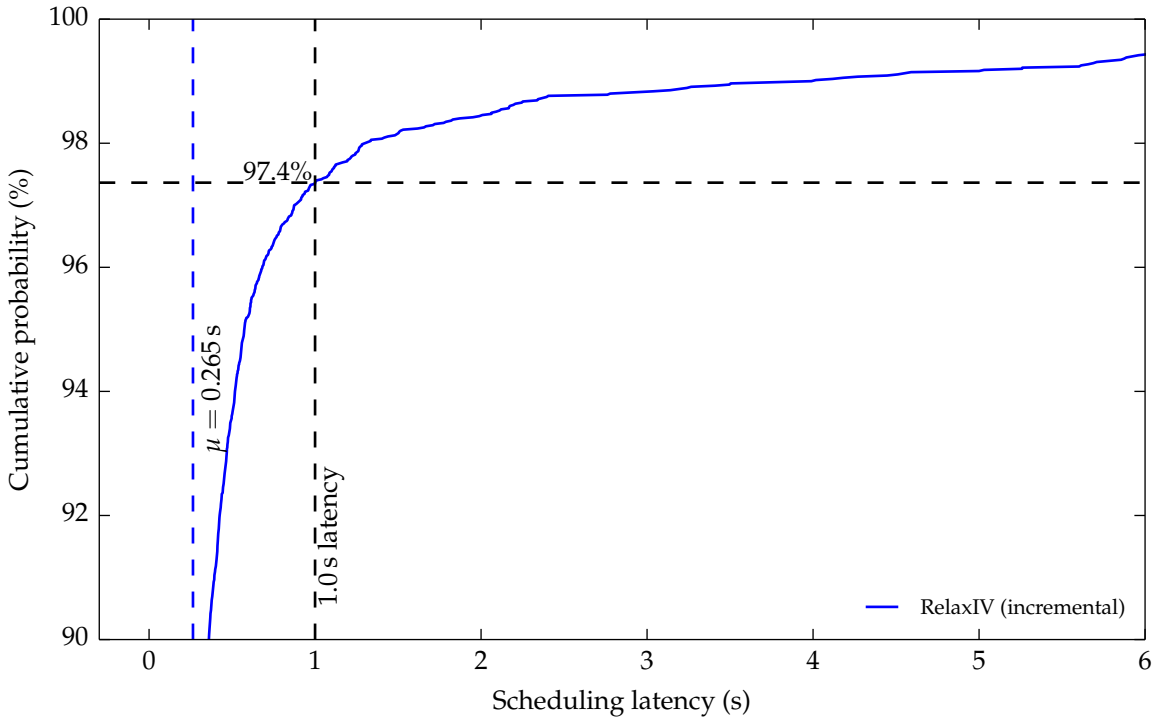


Figure 4.19: Zooming in on figure 4.18b. This CDF shows my incremental relaxation algorithm achieves sub-second scheduling latency on the full Google cluster (12,000 machines) in 97.4% of cases. The line for CS2 is not visible, as it is off the x-axis scale.

4.6 Summary

In this chapter, I have outlined the correctness tests performed in this project, including unit tests for each algorithm and integration tests with the Firmament system. Following this, I described how a realistic simulation of a Google cluster was developed upon publicly available trace data. I then conducted benchmarks on the approximate and incremental solver. I conclude in the next chapter by summarising these results and by discussing areas for further investigation.

Chapter 5

Conclusions

This dissertation has described the implementation and evaluation of high-performance flow solvers. As far as I am aware, both the approximate and incremental solvers are the first of their kind. The incremental solver outperforms state of the art implementations by an order of magnitude, achieving sub-second scheduling latency on the largest clusters used today. I found approximation to be unsuitable for flow scheduling, however the results are promising for its use in other domains.

5.1 Achievements

I have exceeded the initial project requirements by developing an incremental solver, in addition to the core requirement of an approximate solver. Furthermore, I have significantly extended Firmament to complete my evaluation, including adding support for the Quincy cost model.

The incremental solver has broken new ground, with no comparable published work. This implementation has direct real-world applicability, allowing flow schedulers to compete with traditional heuristic approaches on scheduling latency. Further research is required to develop flow schedulers into a finished product; however, early results suggest significant gains in performance and efficiency of data centres can be realised [60].

I believe this is also the first investigation into approximate solution methods for the (single-commodity) minimum-cost flow problem. Although I found approximation to be of limited use for flow scheduling, considerable performance gains of over 10× are realised on other classes of networks. Furthermore, the cost of the solutions is very close to the minimum, with mean accuracy above 98.9% in all cases. Consequently, I would anticipate approximation to be useful in a variety of other applications, where speed is more important than strict optimality.

5.2 Lessons learnt

It was necessary to assimilate the large body of existing research into flow problems before I could attempt to improve upon it. Although I enjoyed studying the area, I underestimated the amount of time this preparatory work would take. In retrospect, I perhaps started implementation work too early, and should have waited until I had a firm understanding of the field.

5.3 Further work

The project is complete in the sense that it satisfies the requirements outlined in my project proposal (see appendix E). However, there is scope both to improve the solvers, and to further investigate their performance:

- **New incremental solvers:** it may be possible to adapt the network simplex algorithm to operate incrementally. Although this appears to be considerably more challenging than the modifications required for augmenting path algorithms, if successful it could yield a still greater speedup.
- **Approximation heuristics:** the terminating conditions, cost convergence and task migration convergence, are relatively simplistic. More sophisticated approaches may improve the reliability of the algorithm, and achieve speedups closer to that of the oracle policy.
- **Other applications:** the focus of this project was on flow scheduling, however flow problems arise in a wide range of domains. The approximation method seems to be very generally applicable, as explained above: I would like to conduct a more thorough investigation into its performance in other domains.

My incremental solver has made a key contribution to the Firmament system, enabling it to scale to large clusters. I plan to work with my supervisors to refine my work, including a description of the incremental technique in forthcoming papers on Firmament. In addition, I hope to publish a general version of these results, so that those working in other application areas can benefit from this work.

Bibliography

[1] Ravindra K. Ahuja, Thomas L. Magnanti and James B. Orlin. *Network Flows: Theory, Algorithms and Applicationsa*. Prentice Hall, 1993 (cited on pages 4, 9, 14–15, 23, 26, 28, 45, 99, 101–103).

[2] Mohammad M. Amini and Richard S. Barr. 'Network reoptimization algorithms: a statistically designed comparison'. In: *ORSA Journal on Computing* (1993) (cited on page 44).

[3] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. 2009 (cited on page 1).

[4] Dimitri P. Bertsekas. *A distributed algorithm for the assignment problem*. Technical report. Lab for Information and Decision Systems, MIT, 1979 (cited on page 5).

[5] Dimitri P. Bertsekas. 'A distributed asynchronous relaxation algorithm for the assignment problem'. In: *24th IEEE Conference on Decision and Control*. 1985, pages 1703–1704 (cited on page 34).

[6] Dimitri P. Bertsekas and Paul Tseng. 'The relax codes for linear minimum cost network flow problems'. In: *Annals of Operations Research* (1988) (cited on pages 4, 49–50).

[7] Dimitri P. Bertsekas and Paul Tseng. *RELAX-IV: A Faster Version of the RELAX Code for Solving Minimum Cost Flow Problems*. Technical report. MIT, 1994 (cited on page 4).

[8] Robert Bland and David Jensen. *On the Computational Behavior of a Polynomial-Time Network Flow Algorithm*. Technical report. Cornell University Operations Research and Industrial Engineering, 1985 (cited on page 4).

[9] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian et al. 'Apollo: scalable and coordinated scheduling for cloud-scale computing'. In: *The 11th USENIX Conference on Operating Systems Design and Implementation*. 2014 (cited on page 9).

[10] Ursula Bünnagel, Bernhard Korte and Jens Vygen. 'Efficient implementation of the Goldberg–Tarjan minimum-cost flow algorithm'. In: *Optimization Methods and Software* (1998) (cited on pages 5, 107–108).

[11] R. G. Busacker and P. J. Gowen. *A Procedure for Determining a Family of Minimum–cost Network Flow Patterns*. Technical report. The John Hopkins University, 1960 (cited on page 4).

[12] Yanpei Chen, Sara Alspaugh and Randy Katz. 'Interactive analytical processing in big data systems: a cross-industry study of MapReduce workloads'. In: *Proceedings of the VLDB Endowment* (Aug. 2012) (cited on pages 56, 111).

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein. *Introduction to Algorithms*. 3rd. The MIT Press, 2009 (cited on pages 23, 98).

[14] George B Dantzig. 'Programming in a linear structure'. In: *Econometrica* (1949) (cited on page 3).

[15] George B Dantzig. 'Application of the simplex method to a transportation problem'. In: *Activity Analysis of Production and Allocation* (1951) (cited on page 3).

[17] Jürgen Dedorath, Jordan Gergov and Torben Hagerup. 'More efficient parallel flow algorithms'. In: *Algorithms and Computations*. 1995 (cited on page 120).

[18] DIMACS. *The First DIMACS International Algorithm Implementation Challenge: Problem Definitions and Specifications*. 1991. URL: ftp://dimacs.rutgers.edu/pub/netflow/general-info/specs.tex (cited on page 51).

[19] Jack Edmonds and Richard M. Karp. 'Theoretical improvements in algorithmic efficiency for network flow problems'. In: *Journal of the ACM* (Apr. 1972) (cited on page 4).

[20] David Eppstein, Zvi Galil and Giuseppe F. Italiano. *Dynamic Graph Algorithms*. Technical report. ALCOM-IT, 1996 (cited on page 45).

[21] Andrew Fikes. 'Storage architectures and challenges'. In: *Google Faculty Summit*. 2010 (cited on page 56).

[22] Marshall L. Fisher. 'The Lagrangian relaxation method for solving integer programming problems'. In: *Management Science* (1981) (cited on page 26).

[23] Lester R Ford and Delbert R Fulkerson. 'Maximal flow through a network'. In: *Canadian Journal of Mathematics* (1956) (cited on page 97).

[24] Antonio Frangioni, Claudio Gentile and Dimitri P. Bertsekas. *MCFClass RelaxIV, Version 1.80*. 2011. URL: http://www.di.unipi.it/optimize/Software/MCF.html#RelaxIV (cited on pages 5, 49–50, 78).

[25] Antonio Frangioni and Antonio Manca. 'A computational study of cost reoptimization for min-cost flow problems'. In: *INFORMS Journal on Computing* (2006) (cited on pages 44, 46–47, 50).

[26] Naveen Garg and Jochen Könemann. 'Faster and simpler algorithms for multicommodity flow and other fractional packing problems'. In: *SIAM Journal on Computing* (2007) (cited on page 40).

[27] Sanjay Ghemawat, Howard Gobioff and Shun-Tak Leung. 'The Google file system'. In: *The 19th ACM Symposium on Theory of Computing*. SOSP '03. 2003 (cited on page 56).

[28] A. Goldberg and R. Tarjan. 'Solving minimum-cost flow problems by successive approximation'. In: *The 19th Annual ACM Symposium on Theory of Computing*. 1987 (cited on pages 33–34, 39, 61, 105–106, 117).

[29] Andrew V. Goldberg. 'An efficient implementation of a scaling minimum-cost flow algorithm'. In: *Journal of Algorithms* (1997) (cited on pages 5, 40, 61, 107–108, 117).

[30] Andrew V. Goldberg. 'The partial augment-relabel algorithm for the maximum flow problem'. In: *Algorithms - ESA 2008*. 2008 (cited on page 5).

[31] Andrew V. Goldberg and Michael Kharitonov. 'On implementing scaling push-relabel algorithms for the minimum-cost flow problem'. In: *Network Flows and Matching: First DIMACS Implementation Challenge*. AMS, 1993 (cited on page 69).

[32] Andrew V. Goldberg and Robert E. Tarjan. 'A new approach to the maximum-flow problem'. In: *Journal of the ACM* (Oct. 1988) (cited on page 5).

[33] Andrew V. Goldberg and Robert E. Tarjan. 'Finding minimum-cost circulations by canceling negative cycles'. In: *Journal of the ACM* (Oct. 1989) (cited on pages 4, 22).

[34] Andrew V. Goldberg and Robert E. Tarjan. 'Finding minimum-cost circulations by successive approximation'. In: *Mathematics of Operations Research* (1990) (cited on pages 5, 36, 38).

[35] Donald Goldfarb and Jianxiu Hao. 'Polynomial-time primal simplex algorithms for the minimum cost network flow problem'. In: *Algorithmica* (1992) (cited on page 4).

[36] Michael D Grigoriadis. 'An efficient implementation of the network simplex method'. In: *Netflow at Pisa*. Springer, 1986 (cited on page 4).

[37] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz et al. 'Mesos: a platform for fine-grained resource sharing in the data center'. In: *The 8th USENIX Conference on Networked Systems Design and Implementation*. 2011 (cited on page 8).

[38] Frank L Hitchcock. 'The distribution of a product from several sources to numerous localities'. In: *Journal of Mathematical Physics* (1941) (cited on page 3).

[39] Bruce Hoppe and Éva Tardos. 'Polynomial time algorithms for some evacuation problems'. In: *The Fifth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '94. Society for Industrial and Applied Mathematics, 1994 (cited on page 40).

[40] IG Systems Inc. *CS2 Software, Version 4.6*. 2009. URL: https://github.com/iveney/cs2 (cited on pages 5, 61–62, 64, 78).

[41] Masao Iri. 'A new method for solving transportation-network problems'. In: *Journal of the Operations Research Society of Japan* (1960) (cited on page 4).

[42] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell and Dennis Fetterly. 'Dryad: distributed data-parallel programs from sequential building blocks'. In: *The 2nd ACM SIGOPS Symposium on Operating Systems Principles* (Mar. 2007) (cited on page 44).

[43] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar and Andrew Goldberg. 'Quincy: fair scheduling for distributed computing clusters'. In: *The 22nd ACM SIGOPS Symposium on Operating Systems Principles*. 2009 (cited on pages 2, 7–8, 10, 56, 78, 92, 94, 115, 119).

[44] William S. Jewell. *Optimal Flow Through Networks*. Interim Technical Report. MIT, 1958 (cited on page 4).

[45] David S. Johnson and Catherine C. McGeoch. *Network Flows and Matching: First DIMACS Implementation Challenge*. Center for Discrete Mathematics and Theoretical Computer Science. AMS, 1993 (cited on page 69).

[46] Leonid Vitalievich. Kantorovich. 'Mathematical methods of organizing and planning production'. In: *Management Science* (1960). Translation of Russian work from 1939 (cited on page 3).

[47] Zoltán Király and Péter Kovács. 'Efficient implementations of minimum-cost flow algorithms'. In: *CoRR* (2012) (cited on pages 4–5, 25–26, 33, 69, 78, 107, 117).

[48] Morton Klein. 'A primal method for minimal cost flows with applications to the assignment and transportation problems'. In: *Management Science* (1967) (cited on pages 4, 22).

[49] D. Klingman, A. Napier and J. Stutz. 'NETGEN: a program for generating large scale capacitated assignment, transportation, and minimum cost flow network problems'. In: *Management Science* (1974) (cited on page 69).

[50] T. C. Koopmans. 'Optimum untilization of the transportation system'. In: *Econometrica* (1949) (cited on page 3).

[51] Péter Kovács. 'Minimum-cost flow algorithms: an experimental evaluation'. In: *Optimization Methods Software* (Jan. 2015) (cited on page 5).

[52] Andreas Löbel. *Solving Large-Scale Real-World Minimum-Cost Flow Problems by a Network Simplex Method*. Technical report. ZIB, 1996 (cited on page 4).

[53] Kay Ousterhout, Patrick Wendell, Matei Zaharia and Ion Stoica. 'Sparrow: distributed, low latency scheduling'. In: *The 24th ACM Symposium on Operating Systems Principles*. SOSP '13. 2013 (cited on page 8).

[54] Git Project. *Git Version Control System*. 2015. URL: http://git-scm.com/ (cited on page 109).

[55] G. Ramalingam and Thomas Reps. 'An incremental algorithm for a generalization of the shortest-path problem'. In: *Journal of Algorithms* (1996) (cited on page 117).

[56] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz and Michael A. Kozuch. 'Heterogeneity and dynamicity of clouds at scale: Google trace analysis'. In: *The 3rd ACM Symposium on Cloud Computing*. SOCC '12. 2012 (cited on pages 55, 111).

[57] Charles Reiss, John Wilkes and Joseph L. Hellerstein. *Google cluster-usage traces: format and schema*. Technical Report. Revised 2012.03.20. Posted at URL http://code.google.com/p/googleclusterdata/wiki/TraceVersion2. Google Inc., Nov. 2011 (cited on pages 55, 119).

[58] H. Röck. 'Scaling techniques for minimal cost network flows'. In: *Discrete Structures and Algorithms*. Hanser, 1980 (cited on page 4).

[59] Liam Roditty and Uri Zwick. 'On dynamic shortest paths problems'. In: *Algorithmica* (2011) (cited on page 117).

[60] Malte Schwarzkopf. 'Operating system support for warehouse-scale computing'. PhD thesis. University of Cambridge, 2015 (cited on pages 1, 8, 10, 51, 81, 89).

[61] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek and John Wilkes. 'Omega: flexible, scalable schedulers for large compute clusters'. In: *The 8th ACM European Conference on Computer Systems*. 2013 (cited on page 8).

[62] P. T. Sokkalingam, Ravindra K. Ahuja and James B. Orlin. 'New polynomial-time cycle-canceling algorithms for minimum-cost flows'. In: *Networks* (2000) (cited on pages 4, 22).

[63] Éva Tardos. 'A strongly polynomial minimum cost circulation algorithm'. In: *Combinatorica* (July 1985) (cited on page 5).

[64] Robert E. Tarjan. 'Network flows'. In: *Data Structures and Network Algorithms*. SIAM, 1983. Chapter 8 (cited on page 45).

[65] Robert E. Tarjan. 'Efficiency of the primal network simplex algorithm for the minimum-cost circulation problem'. In: *Mathematics of Operations Research* (1991) (cited on page 4).

[66] N. Tomizawa. 'On some techniques useful for solution of transportation network problems'. In: *Networks* (1971) (cited on page 4).

[67] Paul Tseng and Dimitri Bertsekas. 'Relaxation methods for minimum cost ordinary and generalized network flow problems'. In: *Operations Research* (1988) (cited on page 4).

[68] John Wilkes. *More Google cluster data*. Nov. 2011. URL: http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html (cited on pages 55, 119).

[69] Matei Zaharia. *The Hadoop fair scheduler*. 11th July 2008. URL: https://issues.apache.org/jira/browse/HADOOP-3746 (cited on page 8).

[70] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker and Ion Stoica. 'Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling'. In: *The 5th European Conference on Computer Systems*. 2010 (cited on page 115).

Appendix A

Libraries used in Hapi

This project made extensive use of the Boost family of libraries. Boost has been extensively peer-reviewed and tested, and is designed to integrate well with the C++ standard library. I used Boost's Program Options library for command-line argument parsing. To measure algorithm runtime, I employed Boost's Timer library. I also used Boost for a variety of smaller tasks, such as string manipulation during parsing.

For logging, I used the glog library which was developed by Google. glog is highly configurable, allowing logging levels to be specified on a per-file or even per-function basis. glog also supports checking a variety of assertions: these were used extensively during development to verify invariants of algorithms and data structures.

I developed unit tests with the aid of the gtest library, also due to Google. It follows the popular xUnit architecture, which I was already familiar with from JUnit. Further, I felt it was one of the best designed unit testing libraries: it has support for a wide range of assertions, and test failure produces informative error messages.

Appendix B

Details of flow scheduling networks

In Firmament, scheduling policies are represented in terms of a *cost model*. As the name suggests, this specifies the costs of arcs. It also determines the network structure, grouping tasks and resources into *equivalence classes*. In this section, I describe how I ported Quincy to this model. Details of the implementation work are provided in §3.8 and §4.2.3. For a more thorough description of the general framework and other examples of cost models, see Schwarzkopf [60, ch. 5].

B.1 Network structure

Figure B.1 depicts a Quincy flow network, on a cluster of four machines scheduling two jobs with a total of five tasks.

Each task i in job j is represented by a vertex $\mathbf{T}_i^j$, with a single unit of supply. Machines are represented by vertices $\mathbf{M}_l$, with arcs to a sink vertex $\mathbf{S}$. Task vertices have arcs to machine vertices or aggregators, the arc cost specifying the task's preference for being scheduled on that machine.

There may be more tasks than can be scheduled. To ensure the flow problem is always feasible, *unscheduled aggregators* $\mathbf{U}^j$ exist for each job j, allowing excess flow to drain to the sink $\mathbf{S}$. Each task vertex $\mathbf{T}_i^j$ has an arc to its unscheduled aggregator $\mathbf{U}_j$, with the cost specifying the penalty for being unscheduled.

Although it would be possible for each task to have an arc to every machine, this would result in a much larger network than necessary. Firmament has support for *equivalence classes*, which may be over tasks or resources[1]. Each equivalence class has an aggregator vertex, with arcs to all vertices in the class. In this way, multiple arcs connecting to vertices in the same class can be combined into a single arc.

The Quincy system provides a concrete example of an equivalence class. Each rack has an associated *rack aggregator* vertex $\mathbf{R}_k$, with arcs to every machine in the

[1] In the simplest model, each machine is a resource. However, a more fine-grained representation than this is possible, such as modelling individual processor cores.

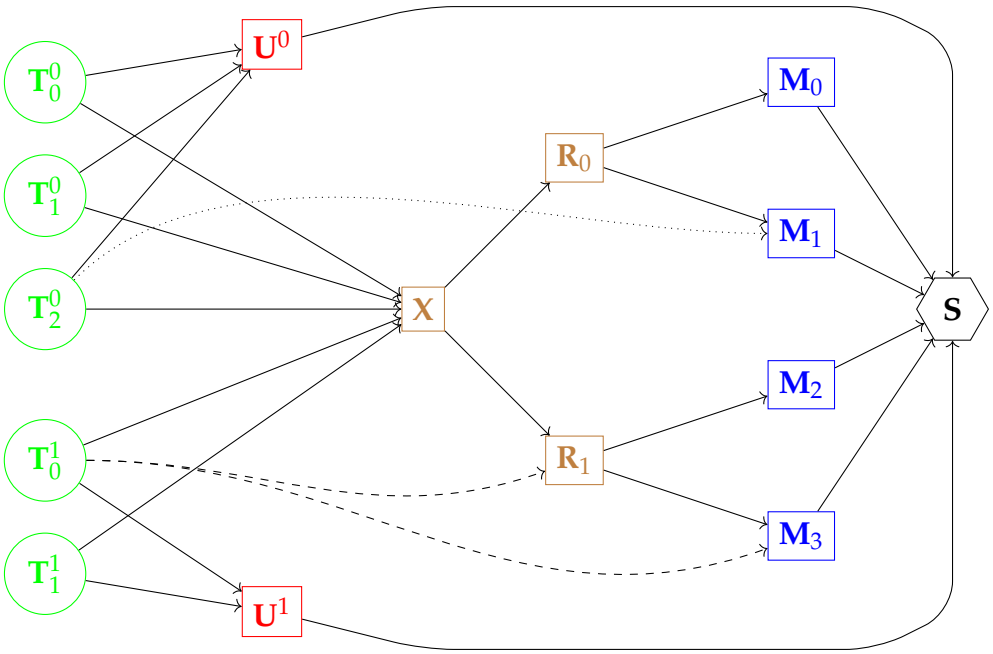


*Figure B.1: A small Quincy scheduling flow network. Vertices are **machines** $\mathbf{M}_l$, **cluster and rack aggregators** X and $\mathbf{R}_l$, **tasks** $\mathbf{T}_i^j$, **unscheduled aggregators** $\mathbf{U}^j$ and sink S. Task $\mathbf{T}_2^0$ is already scheduled on machine $\mathbf{M}_1$: the dotted line represents the **running arc**. All other tasks are unscheduled. Task $\mathbf{T}_0^1$ has two **preference arcs** to machine M_3 and rack aggregator R_1, represented by dashed lines.*

rack. Moreover, a single *cluster aggregator* **X** is introduced, with arcs to each rack aggregator. Each task has a single arc to **X**, and *preference arcs* to favoured machines and racks.

B.2 Capacities

Figure B.2 shows arc capacities for the previous example network, along with a possible solution to the network.

Every arc leaving a task vertex $\mathbf{T}_i^j$ has unit capacity. Note $\mathbf{T}_i^j$ has unit supply and no incoming arcs, so any (non-zero) capacity would be equivalent to unit capacity.

The arcs $\mathbf{M}_l \rightarrow \mathbf{S}$ from machine vertices to the sink vertex have capacity K, where K is a parameter specifying the number of tasks which may concurrently run on a machine[2].

The arcs $\mathbf{R}_k \rightarrow \mathbf{M}_l$ also have capacity K, since up to K jobs may be scheduled on each machine in the rack via the aggregator. The arcs $\mathbf{X} \rightarrow \mathbf{R}_k$ have capacity mK, where m

[2]In the original Quincy system, $K = 1$. However, this is clearly a poor utilisation of modern multi-core machines. In fact, performance may be improved by setting $K >$ # of cores, on simultaneous multi-threading (SMT) machines.

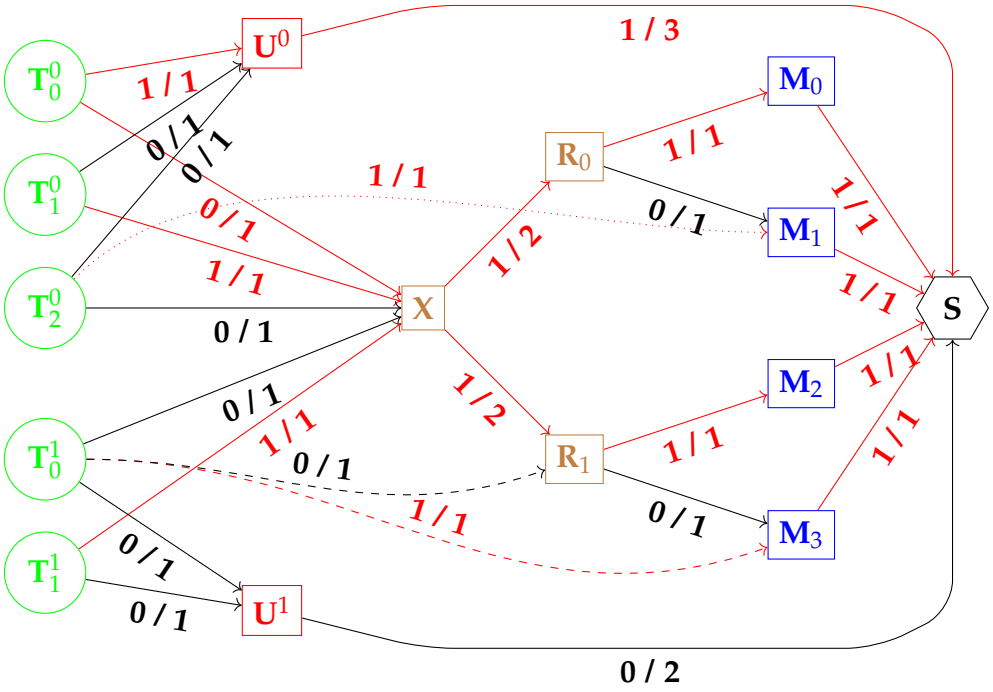


Figure B.2: The network previously depicted in figure B.1 with an example flow. Each arc has a label `flow / capacity`*, with those carrying flow in* ***red****. Note* $K = 1$ *for this example: at most one task can run on each machine. The scheduling assignment depicted is* T_0^0 *unscheduled,* T_0^1 *on* M_1 *(i.e. it is not moved) and* T_0^1 *on its preferred machine* M_3 *with* T_0^2 *and* T_1^1 *being scheduled on any idle machine.*

Quincy sets the upper bound F_j *on the number of running tasks to equal the number of machines divided by the number of jobs. So,* $F_j = 2$ *in this example. Accordingly,* $\mathbf{U}_0$ *has a demand of* 1*. Hence the flow entering* $\mathbf{U}_0$ *from* $\mathbf{T}_0^0$ *does not leave the vertex.*

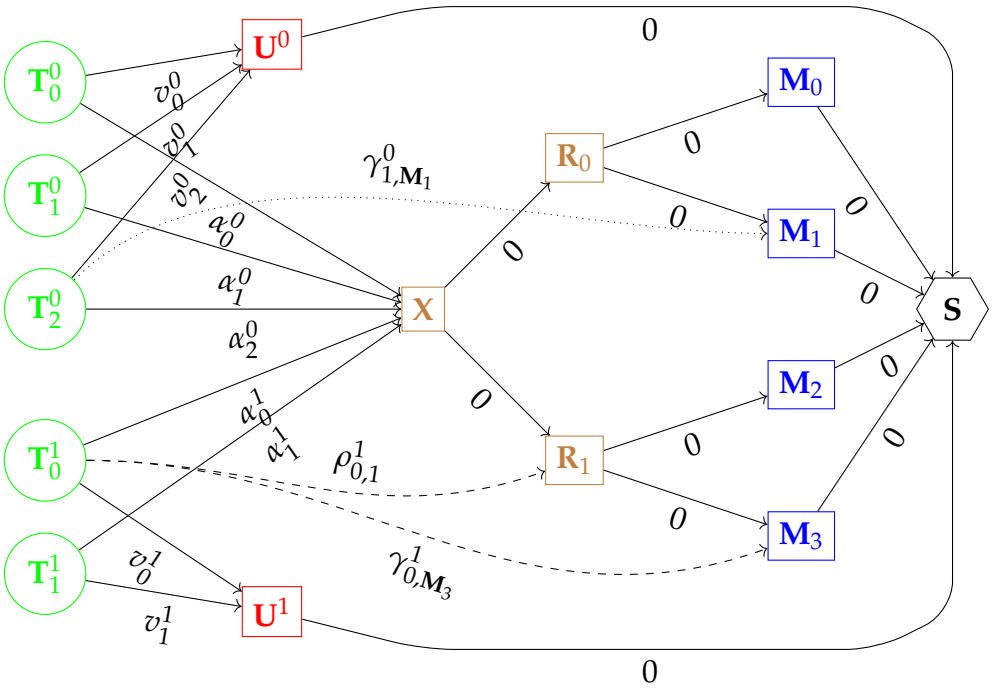


Figure B.3: The network previously depicted in figure B.1, with arcs labelled by their cost. Note only arcs leaving tasks have a cost in Quincy.

is the number of machines in each rack[3].

The arcs $\mathbf{U}^j \to \mathbf{S}$ can be uncapacitated, allowing any number of tasks in job j to be unscheduled. Firmament and Quincy, however, choose to enforce a lower bound E_j and upper bound F_j on the number of tasks that can be scheduled for each job. This can provide some degree of fairness, and in particular specifying $E_j \geq 1$ ensures starvation freedom [43, p. 19].

Let N_j be the number of tasks submitted for job j. To enforce the upper bound F_j, give $\mathbf{U}^j$ a demand of $N_j - F_j$. At least this many tasks in job j must then be left unscheduled, guaranteeing no more than F_j tasks are scheduled.

To enforce the lower bound, set the capacity on $\mathbf{U}^j \to \mathbf{S}$ to $F_j - E_j$. The maximum number of unscheduled tasks is thus the $N_j - F_j$ which are absorbed at $\mathbf{U}^j$, plus the $F_j - E_j$ draining via $\mathbf{U}^j \to \mathbf{S}$. This gives an upper bound of $N_j - E_j$ unscheduled tasks, guaranteeing that at least E_j tasks remain scheduled.

B.3 Costs

Quincy seeks to achieve *data locality*: scheduling tasks close to their input data, in order to minimise network traffic. Consequently, the arc costs (given in table B.1)

[3] In fact, it would be perfectly legitimate to leave these arcs uncapacitated: the capacity on $\mathbf{M}_l \to \mathbf{S}$ will ensure there are not too many tasks scheduled. However, these capacity constraints may improve the efficiency of the solver.

Cost	Arc	Meaning
$\gamma_{i,m}^{j}$	$\mathbf{T}_i^j \to \mathbf{M}_m$	Cost of scheduling on machine $\mathbf{M}_m$.
$\rho_{i,l}^{j}$	$\mathbf{T}_i^j \to \mathbf{R}_l$	Cost of scheduling on worst machine in rack $\mathbf{R}_l$.
α_i^j	$\mathbf{T}_i^j \to \mathbf{X}$	Cost of scheduling on worst machine in cluster.
v_i^j	$\mathbf{T}_i^j \to \mathbf{U}^j$	Cost of leaving task $\mathbf{T}_i^j$ unscheduled.

Table B.1: Costs in the Quincy model. Any arcs not mentioned in the table have a cost of zero[4].

Variable	Meaning
$\chi^X\left(\mathbf{T}_i^j\right), \chi_l^R\left(\mathbf{T}_i^j\right), \chi_m^M\left(\mathbf{T}_i^j\right)$	Data transfer for task $\mathbf{T}_i^j$ across the core switch.
$\mathcal{R}^X\left(\mathbf{T}_i^j\right), \mathcal{R}_l^R\left(\mathbf{T}_i^j\right), \mathcal{R}_m^M\left(\mathbf{T}_i^j\right)$	Data transfer for task $\mathbf{T}_i^j$ across the top-of-rack switch.
θ_i^j	Number of seconds task $\mathbf{T}_i^j$ has spent scheduled.
ν_i^j	Number of seconds task $\mathbf{T}_i^j$ has spent unscheduled.

Table B.2: Variables in the Quincy model.

depend on bandwidth consumption of the task. Figure B.2 show these costs on the example network.

The model assumes that tasks process data residing in a distributed file system. Files may be accessed from any machine, with the file being fetched from remote machines if there is no local copy.

Retrieving a file from a machine in another rack is more expensive – both in terms of request latency and network resources consumed – than from a neighbouring machine. Reading the data from a directly connected disk is cheaper still. The Quincy scheduler quantifies the data transfer costs for each scheduling assignment. Solving the resulting minimum-cost flow problem yields an assignment which minimises the total data transfer cost, achieving data locality.

Table B.2 gives the variables maintained by the Quincy scheduler. All arc costs are functions of these variables. The system breaks down the data transfer of a task $\mathbf{T}_i^j$ into two components: transfer across the core switch, $\chi\left(\mathbf{T}_i^j\right)$, and top of rack switches, $\mathcal{R}\left(\mathbf{T}_i^j\right)$.

Parameter	Meaning
ϵ	Cost of transferring 1 GB across core switch
ψ	Cost of transferring 1 GB across top of rack switch
ω	Wait-time cost factor for unscheduled aggregators

Table B.3: Parameters for the Quincy model.

For a particular machine $\mathbf{M}_m$, the data transfer $\chi_m^M\left(\mathbf{T}_i^j\right)$ and $\mathcal{R}_m^M\left(\mathbf{T}_i^j\right)$ can be computed exactly. However, when it comes to determining the cost of arcs to aggregator vertices $\mathbf{R}_l$ or $\mathbf{X}$, an exact figure is not possible. Instead, Quincy takes a conservative upper bound: the *greatest* data transfer that results from scheduling $\mathbf{T}_i^j$ on the *worst* machine in rack $\mathbf{R}_l$ or the cluster as a whole, respectively.

The system also keeps track of the time θ_i^j and ν_i^j a task spends scheduled and unscheduled, respectively. If the task is stopped and then restarted, these times continue to accumulate. This property is needed to guarantee that the scheduler makes progress [43, p. 19]..

Quincy is controlled by three simple parameters, given in table B.3. ϵ and ψ specify the cost of data transfers across core and top of rack switches respectively. In general, we would expect $\epsilon > \psi$, since the core switch is more likely to become a bottleneck. ω controls the penalty for leaving a task unscheduled[5]. Increasing ϵ and ψ causes the scheduler to more aggressively optimise for data locality, whereas increasing ω decreases queuing delay.

It is now possible to state formulae for the arc costs listed in table B.1. For convenience, define the data transfer cost function:

$$d_b^A\left(\mathbf{T}_i^j\right) = \epsilon\chi_b^A\left(\mathbf{T}_i^j\right) + \psi\mathcal{R}_b^A\left(\mathbf{T}_i^j\right)$$

The cost of scheduling on the worst machine in the cluster and the worst machine in rack $\mathbf{R}_l$ can be immediately stated from this:

$$\alpha_i^j = d^X\left(\mathbf{T}_i^j\right);$$
$$\rho_i^{j,l} = d_l^R\left(\mathbf{T}_i^j\right).$$

It is tempting to use the same form for the cost, $\gamma_{i,m}^j$ of scheduling on a particular machine $\mathbf{M}_m$. When $\mathbf{T}_i^j$ is not running on $\mathbf{M}_m$ (either it is unscheduled, or is running on a different machine), this is valid.

However, this formula is inappropriate when $\mathbf{T}_i^j$ is already scheduled on $\mathbf{M}_m$: $d_m^M\left(\mathbf{T}_i^j\right)$ will overestimate the cost of the remaining data transfer. To account for the work already invested in $\mathbf{T}_i^j$, the time it has been running is subtracted from the data transfer cost, giving:

$$\gamma_{i,m}^j = d_m^M\left(\mathbf{T}_i^j\right) - \begin{cases}\theta_i^j & \text{if } \mathbf{T}_i^j \text{ running on } \mathbf{M}_m;\\ 0 & \text{otherwise.}\end{cases}$$

The only remaining cost is v_i^j, the penalty associated with leaving a task $\mathbf{T}_i^j$ unscheduled. Quincy sets set it proportional to the length of time it has been unscheduled:

$$v_i^j = \omega\nu_n^j$$

[5]It is possible to vary ω between jobs in order to encode a notion of priority.

B.4 Properties

Although the costs assigned to arcs vary between policies, the structure of the network remains the same. Some simple properties are proved below, which will be useful when analysing the asymptotic complexity of algorithms.

Lemma 2.2.1 (Number of vertices in flow scheduling networks)**.** *Let n denote the number of vertices in the network. Then $n = \Theta(\#\ \textit{machines} + \#\ \textit{tasks})$.*

Proof. There is one vertex $\mathbf{T}_i^j$ for every task and one vertex $\mathbf{M}_m$ for every machine, so trivially $n = \Omega(\#\ \text{machines} + \#\ \text{tasks})$.

It remains to show $n = O(\#\ \text{machines} + \#\ \text{tasks})$. Consider each class of vertex in turn.

As stated above # task vertices = # tasks and # machine vertices = # machines. There is an unscheduled aggregator $\mathbf{U}^j$ for each job j, and # jobs $= O(\#\ \text{tasks})$. There is at least one machine in every rack, so # rack aggregator vertices $= O(\#\ \text{machines})$.

In addition, there is also a cluster aggregator vertex $\mathbf{X}$ and sink vertex $\mathbf{S}$ which are independent of the number of tasks and machines, contributing $O(1)$ vertices.

Thus:

$$n = O(\#\ \text{tasks} + \#\ \text{machines} + 1) = O(\#\ \text{tasks} + \#\ \text{machines})$$

Hence:

$$n = \Theta(\#\ \text{machines} + \#\ \text{tasks})$$

□

Lemma 2.2.2 (Number of arcs in flow scheduling networks)**.** *Let m denote the number of arcs in the network. Then $m = O(n)$. That is, the network is* sparse*.*

Proof. Consider the outgoing arcs from each class of vertex. Since every arc is outgoing from exactly one vertex, this will count every vertex exactly once.

Each machine $\mathbf{M}_l$ and unscheduled aggregator $\mathbf{U}^j$ has a single outgoing arc, to sink vertex $\mathbf{S}$. This contributes $O(\#\ \text{machines})$ arcs. The sink vertex $\mathbf{S}$ has no outgoing arcs.

Rack aggregators $\mathbf{R}_l$ have outgoing arcs to each machine in their rack. Each machine is present in exactly one rack, so these contribute collectively $O(\#\ \text{machines})$ arcs. The cluster aggregator $\mathbf{X}$ has outgoing arcs to each rack; since $O(\#\ \text{racks}) = O(\#\ \text{machines})$, this contributes $O(\#\ \text{machines})$ arcs.

It remains to consider task vertices $\mathbf{T}_i^j$. The number of arcs leaving the task vertex has a constant upper bound. The system computes a *preference list* of machines and racks, and includes arcs only to those vertices (and the cluster aggregator $\mathbf{X}$). Thus this contributes $O(\#\ \text{tasks})$ arcs.

Hence:

$$m = (\#\text{ machines} + \#\text{ tasks})$$

By lemma 2.2.1, it follows:

$$m = O(n)$$

□

Remark B.4.1. In fact, $m = \Theta(n)$: the network is connected so certainly $m = \Omega(n)$. However, I will only use the bound $m = O(n)$.

Appendix C

Details of algorithms

C.1 Cycle cancelling

C.1.1 Algorithm description

Cycle cancelling iteratively reduces the cost of the solution by sending flow along negative-cost cycles. This is inspired by the negative cycle optimality conditions, stated in theorem 2.2.4:

Theorem 2.2.4 (Negative cycle optimality conditions)**.** *Let* $\mathbf{x}$ *be a (feasible) flow. It is an optimal solution to the minimum-cost flow problem if and only if the residual network* $G_{\mathbf{x}}$ *has no negative cost (directed) cycle.*

Algorithm C.1.1 Cycle cancelling

```
1: x ← result of maximum-flow algorithm          ▷ establishes x feasible
2: while G_x contains a negative cost cycle do
3:     identify negative cost cycle W             ▷ e.g. using Bellman-Ford
4:     δ ← min_{(i,j)∈W} r_ij
5:     augment δ units of flow along cycle W
```

The algorithm is initialised with a feasible flow $\mathbf{x}$. This may be found by any maximum-flow algorithm, such as Ford-Fulkerson [23]. The feasibility of the solution $\mathbf{x}$ is maintained at all times, and its cost is successively reduced.

Each iteration of the algorithm identifies a directed cycle of negative cost in the residual network $G_{\mathbf{x}}$. Note negative cycles can occur despite assumption 2.2.2 of non-negative arc costs, as reverse arcs in the residual network have the opposite sign to forward arcs.

Negative cost cycles are *cancelled* by pushing as much flow as possible along the cycle. This causes it to 'drop out' of the residual network: one or more arcs along the cycle become saturated, and so are no longer present in the residual network. Sending flow along a negative cost cycle reduces the cost of the solution, bringing it closer to

optimality. The algorithm terminates when no negative cost directed cycles remain, guaranteeing optimality by theorem 2.2.4.

Note this generic version of the algorithm does not specify *how* negative cycles are to be identified. I used the well-known Bellman-Ford [13, p. 651] algorithm, although other more efficient methods exist. However, cycle cancelling was only implemented in order to have a known-working algorithm early on in development. Even the fastest variants are too slow for practical purposes.

Correctness

I will show that, if the algorithm terminates, it produces the correct result.

Lemma C.1.1. *Immediately before each iteration of the loop,* $\mathbf{x}$ *is a feasible solution.*

Proof. For the base case, $\mathbf{x}$ is feasible after initialisation, by correctness of the maximum-flow algorithm used.

For the inductive case, suppose $\mathbf{x}$ is feasible immediately prior to an iteration of the loop body. The body pushes flow along a cycle. This maintains feasibility: the excess at the vertices along the cycles remain zero, with any increase in the flow leaving a vertex being counterbalanced by an equal increase in the flow entering that vertex. □

Theorem C.1.2 (Correctness of cycle cancelling)**.** *Upon termination,* $\mathbf{x}$ *is a solution of the minimum-cost flow problem.*

Proof. By lemma C.1.1, $\mathbf{x}$ is a feasible solution upon termination. The algorithm only terminates when no negative-cost directed cycles exist. It follows by theorem 2.2.4 that $\mathbf{x}$ is optimal. □

Termination and asymptotic complexity

I will now show that the algorithm always terminates, and provide a bound on its complexity.

Lemma C.1.3. *The algorithm terminates within* $O(mCU)$ *iterations*[1].

Proof. Each iteration of the algorithm identifies a cycle w, of cost $c < 0$, and pushes $\delta = \min_{(i,j)\in w} r_{ij}$ units of flow along the cycle. Note $\delta > 0$, otherwise the cycle would not exist in the residual network.

The objective function value changes by $c\delta$. By assumption 2.2.1, c and δ must both be integral. So as $c < 0$ and $\delta > 0$, it follows $c \leq -1$ and $\delta \geq 1$ so $c\delta \leq -1$. That is, the cost is decreased by at least one unit each iteration.

[1] See §2.2.3 for a definition of m, C, U and other variables used in complexity analysis.

The number of iterations is thus bounded by cost of the initial feasible flow. mCU is an upper bound on the cost of any flow, hence the result. □

Theorem C.1.4 (Asymptotic complexity of cycle cancelling)**.** *The asymptotic complexity is* $O(nm^2CU)$.

Proof. Note that Bellman-Ford runs in $O(nm)$ time. Augmenting flow along the cycle is of cost linear in the length of the cycle, and so is certainly $O(n)$. Thus each iteration runs in $O(nm)$. By lemma C.1.3, it follows the complexity of the algorithm is $O(nm^2CU)$. □

Remark C.1.1. On networks produced by flow schedulers, the asymptotic complexity is $O(n^3CU)$ by lemma 2.2.2.

C.2 Successive shortest path

C.2.1 Correctness analysis

Lemma C.2.1 and corollary C.2.2 prove properties of the algorithm, allowing theorem 3.3.1 to show that reduced cost optimality is maintained as an invariant. Correctness of the algorithm follows in corollary 3.3.2 by using the terminating condition of the algorithm.

Lemma C.2.1. *Let a pseudoflow* $\mathbf{x}$ *satisfy the reduced cost optimality conditions of eq.* (2.9) *with respect to potentials* $\boldsymbol{\pi}$*. Let* $\mathbf{d}$ *represent the shortest path distances in the residual network* $G_{\mathbf{x}}$ *from a vertex* $s \in V$ *to all other vertices, with respect to the reduced costs* c_{ij}^{π}*. Then:*

(a) $\mathbf{x}$ *also satisfies reduced cost optimality conditions with respect to potentials* $\boldsymbol{\pi}' = \boldsymbol{\pi} - \mathbf{d}$.

(b) The reduced costs $c_{ij}^{\pi'}$ *are zero for all arcs* (i, j) *in the shortest-path tree rooted at* $s \in V$.

Proof. See Ahuja *et al.* [1, lemma 9.11]. □

Corollary C.2.2. *Let a pseudoflow* $\mathbf{x}$ *satisfy the reduced cost optimality conditions, with respect to some potentials* $\boldsymbol{\pi}$*. Let* $\mathbf{x}'$ *denote the pseudoflow obtained from* $\mathbf{x}$ *by sending flow along a shortest path from vertex* s *to some other vertex* $k \in V$*. Then* $\mathbf{x}'$ *also satisfies the reduced cost optimality conditions, with respect to potentials* $\boldsymbol{\pi}' = \boldsymbol{\pi} - \mathbf{d}$.

Proof. (Adapted from Ahuja *et al.* [1, lemma 9.12])

By lemma C.2.1(a), $(\mathbf{x}, \boldsymbol{\pi}')$ satisfies the reduced cost optimality conditions.

Pushing flow along an arc $(i, j) \in G_{\mathbf{x}}$ might add its reversal (j, i) to the residual network. Let P be a shortest path from s to k. By lemma C.2.1(b), it follows that any arc $(i, j) \in P$ has $c_{ij}^{\pi'} = 0$. So $c_{ji}^{\pi'} = 0$. Thus any arcs are added to the residual network by augmenting flow along P have a zero reduced cost, and so still satisfy the reduced cost optimality conditions of eq. (2.9). □

Theorem 3.3.1 (Loop invariant of successive shortest path)**.** *Immediately before each iteration of the loop,* $(\mathbf{x}, \boldsymbol{\pi})$ *satisfies reduced cost optimality.*

Proof. (Induction)

For the base case, note that $(\mathbf{0}, \mathbf{0})$ satisfies reduced cost optimality. $G_{\mathbf{0}} = G$ holds, i.e. the residual and original network are the same. Moreover, all arc costs c_{ij} are non-negative (by assumption 2.2.2) and so the reduced costs $c^{\mathbf{0}}_{ij} = c_{ij}$ are also non-negative. Thus reduced cost optimality holds.

Now, assume that reduced cost optimality holds immediately prior to execution of the loop body. The body computes the shortest path distances $\mathbf{d}$ from a vertex s, and updates $\boldsymbol{\pi}$ to become $\boldsymbol{\pi}'$ as defined in lemma C.2.1. It then pushes flow along a shortest path from s to another vertex, yielding a new flow of the same form as $\mathbf{x}'$ in corollary C.2.2. It follows by corollary C.2.2 that $(\mathbf{x}', \boldsymbol{\pi}')$ satisfies reduced cost optimality at the end of the loop body. Hence, the inductive hypothesis continues to hold. □

Corollary 3.3.2 (Correctness of successive shortest path)**.** *Upon termination,* $\mathbf{x}$ *is a solution to the minimum-cost flow problem.*

Proof. The algorithm terminates when the mass balance constraints of eq. (2.2) are satisfied. At this point, the solution $\mathbf{x}$ is feasible (see §2.2.3).

By theorem 3.3.1, we know the algorithm maintains the invariant that $\mathbf{x}$ satisfies reduced cost optimality.

It follows that $\mathbf{x}$ is both optimal and a feasible flow upon termination, so $\mathbf{x}$ is a solution to the minimum-cost flow problem. □

C.2.2 Terminating Djikstra's algorithm early

Djikstra's algorithm is said to have *permanently labelled* a vertex i when it extracts i from the heap. At this point, Djikstra has found a shortest path from s to i.

I modified the successive shortest path algorithm to terminate Djikstra as soon as it permanently labels a deficit vertex l. Although this does not affect asymptotic complexity, it may considerably improve performance in practice.

Lemma C.2.3. *Define:*

$$d'_i = \begin{cases} d_i & \text{if } i \text{ permanently labelled} \\ d_l & \text{otherwise} \end{cases}$$

Suppose the triangle equality holds on $\mathbf{d}$*, that is:*

$$\forall (i,j) \in E_{\mathbf{x}} \cdot d_j \leq d_i + c^{\boldsymbol{\pi}}_{ij} \tag{C.1}$$

Then it also holds on $\mathbf{d}'$*:*

$$\forall (i,j) \in E_{\mathbf{x}} \cdot d'_j \leq d'_i + c^{\boldsymbol{\pi}}_{ij}$$

Proof. When Djikstra's algorithm is terminated early, the only shortest path distances known are those to permanently labelled vertex. But vertices are labelled in ascending order of their shortest path distance. As l is permanently labelled, it follows that for any unlabelled vertex i:

$$d_l \leq d_i \tag{C.2}$$

But l is the last vertex to be labelled, so it follows that for any permanently labelled vertex i:

$$d_i \leq d_l \tag{C.3}$$

Now, let $(i,j) \in E_{\mathbf{x}}$. It remains to prove $d'_j \leq d'_i + c^{\pi}_{ij}$, for which there are four possible cases.

i and j permanently labelled $d'_i = d_i$ and $d'_j = d_j$, so result follows by eq. (C.1).

i and j not labelled $d'_i = d_l = d'_j$, so result follows by non-negativity of reduced costs c^{π}_{ij}.

i permanently labelled, j not $d'_j = d_l$ by definition, and $d_l \leq d_j$ by eq. (C.2), so $d'_j \leq d_j$. By definition $d'_i = d_i$, so it follows by eq. (C.1) that $d'_j \leq d'_i + c^{\pi}_{ij}$.

i not labelled, j permanently labelled By definition, $d'_j = d_j$. By eq. (C.3), $d_j \leq d_l$, so $d'_j \leq d_l$. By definition, $d'_i = d_l$, so $d'_j \leq d'_i$. Result follows by non-negativity of c^{π}_{ij}. □

Lemma C.2.4. *Recall lemma C.2.1. Let us redefine:*

$$\pi'_i = \begin{cases} \pi_i - d_i & \text{if } i \text{ permanently labelled;} \\ \pi_i - d_l & \text{otherwise.} \end{cases}$$

The original result (a) still holds. The result (b) holds along the shortest path from s to l[2]*.*

Proof. The original proof for the lemma [1, lemma 9.11] uses the triangle inequality stated in eq. (C.1).

By lemma C.2.3, $\mathbf{d}'$ also satisfies the triangle equality. The original proof for (a) thus still holds, as it makes no further assumptions on $\mathbf{d}'$.

As for (b), every vertex i along the shortest path from s to l has been permanently labelled, and so $d'_i = d_i$. Hence the original proof still holds along this path. □

[2]Note that this is all that is needed for the correctness of the algorithm, as this is the only path along which we augment flow.

Any constant shift in the potential for every vertex will leave reduced costs unchanged, so $\boldsymbol{\pi}'$ may equivalently be defined as:

$$\pi'_i = \begin{cases} \pi_i - d_i + d_l & \text{if } i \text{ permanently labelled;} \\ \pi_i & \text{otherwise.} \end{cases}$$

This is computationally more efficient, as it reduces the number of elements of $\boldsymbol{\pi}$ that must be updated.

C.3 Correctness analysis of relaxation

First, I show UpdatePotentials and AugmentFlow preserve reduced cost optimality. Using these lemmas, I show that reduced cost optimality is maintained as an invariant throughout the algorithm. It follows that, if the algorithm terminates, then $\mathbf{x}$ is a solution to the minimum-cost flow problem. I also prove some other useful properties of UpdatePotentials and AugmentFlow.

Lemma C.3.1 (Correctness of UpdatePotentials)**.** *Let $e(S) > r(\boldsymbol{\pi}, S)$. Let $(\mathbf{x}, \boldsymbol{\pi})$ satisfy reduced cost optimality conditions. Then, after executing* UpdatePotentials*, the new pseudoflow and vertex potentials $(\mathbf{x}, \boldsymbol{\pi})$ continue to satisfy reduced cost optimality conditions, and $w(\boldsymbol{\pi}') > w(\boldsymbol{\pi})$.*

Proof. (Adapted from Ahuja *et al.* [1, p. 334])

Arcs with a reduced cost of zero are saturated by lines 2 to 3, and drop out of the residual network as a result. This preserves reduced cost optimality: the flow on arcs with a reduced cost of zero is arbitrary. Moreover, it leaves $w(\boldsymbol{\pi})$ unchanged. Recall the formulation of $w(\boldsymbol{\pi})$ given in eq. (3.2):

$$w(\boldsymbol{\pi}) = \min_x \left[\sum_{(i,j)\in E} c^{\pi}_{ij} x_{ij} + \sum_{i \in V} \pi_i b_i \right]$$

The second sum is unchanged, as potentials $\boldsymbol{\pi}$ are unchanged. The first sum is also unchanged, as x'_{ij} differs from x_{ij} only when the reduced costs are $c^{\pi}_{ij} = 0$.

Note that there is now $r(\boldsymbol{\pi}, S)$ more flow leaving vertices in S, so the tree excess is now:

$$e'(S) = e(S) - r(\boldsymbol{\pi}, S)$$

By the precondition, $e'(S)$ is (strictly) positive.

After lines 2 to 3, all arcs in the residual network crossing the cut $\left(S, \overline{S}\right)$ have positive reduced cost[3]. Let $\alpha > 0$ be the minimal such remaining reduced cost.

[3]Since none had negative reduced cost, by assumption of reduced cost optimality on entering the procedure.

$\boldsymbol{\pi}'$ is now obtained from $\boldsymbol{\pi}$ by increasing the potential of every vertex $i \in S$ in the tree by α. Recall the formulation of the objective function in terms of costs and excesses, given in eq. (3.1):

$$w(\boldsymbol{\pi}) = \min_{x} \left[\sum_{(i,j)\in E} c_{ij} x_{ij} + \sum_{i \in V} \boldsymbol{\pi}_i e_i \right]$$

The first sum is unchanged: modifying the potentials $\boldsymbol{\pi}$ does not change the original arc cost c_{ij} or flow x_{ij}. For the second sum:

$$\begin{aligned}
\sum_{i\in V} \pi'_i e'_i &= \sum_{i \in S} (\pi_i + \alpha) e'_i + \sum_{i \in \overline{S}} \pi_i e'_i \\
&= \alpha \sum_{i \in S} e'_i + \sum_{i \in V} \pi_i e'_i \\
&= \alpha e'(S) + \sum_{i \in V} \pi_i e'_i
\end{aligned}$$

Since $w(\boldsymbol{\pi})$ is unchanged after updating $\mathbf{x}$, it follows:

$$w(\boldsymbol{\pi}') = w(\boldsymbol{\pi}) + \alpha e'(S)$$

$\alpha > 0$ and $e'(S) > 0$ have already been shown; by assumption 2.2.1, it follows that $\alpha \geq 1$ and $e'(S) \geq 1$ by. Thus $\alpha e'(S) \geq 1$, so $w(\boldsymbol{\pi}') > w(\boldsymbol{\pi})$.

It remains to show that updating potentials maintains reduced cost optimality. Note that increasing the potentials of vertices in S by α decreases the reduced cost of arcs in $(S, \overline{S})$ by α, increases the reduced cost of arcs in $(\overline{S}, S)$ by α and leaves the reduced cost of other arcs unchanged.

Prior to updating potentials, all arcs in the residual network had (strictly) positive reduced costs. Consequently, increasing the reduced cost cannot violate reduced cost optimality[4], but decreasing the reduced cost might. Consequently, before updating the potentials, $c^{\pi}_{ij} \geq \alpha$ for all $(i,j) \in (S, \overline{S})$ with $r_{ij} > 0$. Hence, after the potential update, $c^{\pi'}_{ij} \geq 0$ for these arcs. □

Lemma C.3.2 (Correctness of AUGMENTFLOW)**.** *Let $e(S) \leq r(\boldsymbol{\pi}, S)$, and $e_t < 0$. Let $(\mathbf{x}, \boldsymbol{\pi})$ satisfy reduced cost optimality conditions. Then, after executing* AUGMENTFLOW*, there is a new pseudoflow $\mathbf{x}'$ and $(\mathbf{x}', \boldsymbol{\pi})$ still satisfy reduced cost optimality conditions. Moreover, under $\mathbf{x}'$ the excess at vertex s and deficit at vertex t decreases, without changing the excess/deficit at any other vertex.*

Proof. (Adapted from Ahuja *et al.* [1, p. 336])

Line 2 finds a shortest path P from the excess vertex s to a deficit vertex t. Lines 3 to 4 then send as much flow as possible along path P, subject to:

1. satisfying the capacity constraints at each arc on P, and
2. ensuring $e_s \geq 0$ and $e_t \leq 0$.

[4]Note increasing the reduced cost from zero to a positive number could violate optimality, but this case has been excluded.

The restriction on e_s and e_t ensures that the feasibility of the solution is improved, by ensuring the unmet supply/demmand $|e_i|$ monotonically decreases for all $i \in V$[5].

Since an equal amount of flow is sent on each arc in P, the excess at vertices other than the start s and end t of path P are unchanged. □

Theorem 3.4.4 (Correctness of relaxation)**.** *Immediately before each iteration of the loop,* $(\mathbf{x}, \boldsymbol{\pi})$ *satisfies reduced cost optimality. Moreover, upon termination,* $\mathbf{x}$ *is a solution of the minimum-cost flow problem.*

Proof. For the same reason as given in theorem 3.3.1, the initial values of $(\mathbf{x}, \boldsymbol{\pi})$ satisfy reduced cost optimality conditions. AUGMENTFLOW and UPDATEPOTENTIALS are only invoked when their preconditions are satisfied, and so by lemmas C.3.1 and C.3.2 it follows that reduced cost optimality is maintained. Moreover, the main algorithm given in algorithm 3.4.1 does not update $\mathbf{x}$ or $\boldsymbol{\pi}$ except via calls to AUGMENTFLOWS and UPDATEPOTENTIALS. So, reduced cost optimality is maintained as an invariant throughout the algorithm.

The algorithm terminates when the mass balance constraints are satisfied, so $\mathbf{x}$ is feasible. Thus upon termination, $\mathbf{x}$ is feasible and satisfies reduced cost optimality conditions. Therefore, $\mathbf{x}$ is a solution to the minimum-cost flow problem. □

C.4 Cost scaling

C.4.1 Correctness analysis

I show that PUSH and RELABEL preserve ϵ-optimality. I then prove the correctness of REFINE from these results, with correctness of the cost scaling algorithm following as a corollary.

Lemma C.4.1 (Correctness of PUSH)**.** *Let* $(\mathbf{x}, \boldsymbol{\pi})$ *be* ϵ*-optimal, and the precondition for* PUSH(i, j) *hold:* $e_i > 0$, $(i, j) \in E_{\mathbf{x}}$ *and* $c^{\pi}_{ij} < 0$*. Then* $(\mathbf{x}, \boldsymbol{\pi})$ *continues to be* ϵ*-optimal after* PUSH*.*

Proof. PUSH(i, j) increases the flow on arc (i, j). By assumption, (i, j) satisfied ϵ-optimality prior to PUSH. (i, j) may drop out of the residual network after increasing the flow, but this cannot violate optimality. If (i, j) remains in the residual network, (i, j) continues to satisfy ϵ-optimality, since the reduced cost is unchanged.

However, sending flow along (i, j) could add arc (j, i) to the residual network. By the precondition $c^{\pi}_{ij} < 0$, it follows $c^{\pi}_{ji} > 0 \geq -\epsilon$. Thus (j, i) satisfies ϵ-optimality.

No other changes are made which could affect the ϵ-optimality conditions given in eq. (3.5), so PUSH preserves ϵ-optimality. □

[5] If the algorithm were allowed to 'overshoot' and turn s into a deficit vertex or t into an excess, this property might not hold.

Lemma C.4.2 (Correctness of RELABEL)**.** *Let* $(\mathbf{x}, \boldsymbol{\pi})$ *be* ϵ*-optimal, and the precondition for* RELABEL(i) *hold:* $e_i > 0$ *and* $\forall (i,j) \in E_{\mathbf{x}} \cdot c_{ij}^{\pi} \geq 0$.

Let $\boldsymbol{\pi}'$ *denote the potentials after* RELABEL*. Then* $\pi_i' \geq \pi_i + \epsilon$ *and* $\pi_j' = \pi_j$ *for* $j \neq i$*. Moreover,* $(\mathbf{x}, \boldsymbol{\pi}')$ *continues to be* ϵ*-optimal.*

Proof. (Adapted from Goldberg and Tarjan [28, lemma 5.2])

By the precondition, $\forall (i,j) \in E_{\mathbf{x}} \cdot c_{ij}^{\pi} \geq 0$. Substituting for the definition of reduced costs in eq. (2.7) gives $\forall (i,j) \in E_{\mathbf{x}} \cdot c_{ij} + \pi_j \geq \pi_i$. Thus:

$$\pi_i' = \min \{ \pi_j + c_{ij} + \epsilon \mid (i,j) \in E_{\mathbf{x}} \} \geq \pi_i + \epsilon$$

RELABEL does not modify any other components of $\boldsymbol{\pi}$, so $\pi_j' = \pi_j$ for $j \neq i$.

Increasing π_i has the effect of decreasing the reduced cost c_{ij}^{π} of outgoing arcs (i,j), increasing c_{ji}^{π} for incoming arcs (j,i) and leaving the reduced cost of other arcs unchanged. Increasing c_{ji}^{π} cannot violate ϵ-optimality, but decreasing c_{ij}^{π} might. However, for any $(v,w) \in E_{\mathbf{x}}$:

$$c_{vw} + \pi_w - \min \{ \pi_j + c_{ij} \ : \ (i,j) \in E_{\mathbf{x}} \} \geq 0$$

Thus by definition of $\boldsymbol{\pi}'$, and taking out the constant ϵ:

$$c_{vw} + \pi_w - \pi_i' \geq -\epsilon$$

And so $c_{vw}^{\pi'} \geq -\epsilon$, as required for ϵ-optimality. □

Lemma C.4.3 (Correctness of REFINE)**.** *Let the precondition for* REFINE *hold:* $\mathbf{x}$ *is a pseudoflow. Then upon termination of* REFINE*, the postcondition holds:* $(\mathbf{x}, \boldsymbol{\pi})$ *is* ϵ*-optimal.*

Proof. (Adapted from Goldberg and Tarjan [28, theorem 5.4]) The initial flow after lines 2 to 4 of algorithm 3.5.2 is 0-optimal (and so certainly ϵ-optimal). ϵ-optimality is preserved by subsequent PUSH and RELABEL operations by lemmas C.4.1 and C.4.2. Hence ϵ-optimality is maintained as an invariant.

Given that REFINE has terminated, the mass balance constraints must be satisfied by the loop condition on line 5. Thus $\mathbf{x}$ must also be a flow upon termination. □

Theorem 3.5.2 (Correctness of cost scaling)**.** *Upon termination of the algorithm described in algorithm 3.5.1,* $\mathbf{x}$ *is a solution of the minimum-cost flow problem.*

Proof. After each iteration of the algorithm, $(\mathbf{x}, \boldsymbol{\pi})$ satisfies ϵ-optimality by lemma C.4.3. Upon termination, $\epsilon < 1/n$. So by theorem 3.5.1, $\mathbf{x}$ is an optimal solution. □

C.4.2 Termination and complexity analysis

Lemma C.4.4. *The basic operations given in algorithm 3.5.3 have the following time complexities:*

(a) $\textsc{Push}(i, j)$ *runs in* $O(1)$ *time.*

(b) $\textsc{Relabel}(i)$ *runs in* $O(|\text{Adj}(i)|)$ *time, that is linear in the number of adjacent arcs.*

Proof. Immediate from inspection of algorithm 3.5.3. □

Definition C.4.1. An invocation of $\textsc{Push}(i, j)$ is said to be *saturating* if $r_{ij} = 0$ after the operation; otherwise, it is nonsaturating.
Remark C.4.1. Note a push operation is saturating if and only if $e_i \geq r_{ij}$ prior to calling Push.

Lemma C.4.5. *Within an invocation of* Refine*, the following upper bounds apply to the number of times each basic operation is performed:*

(a) $O(n^2)$ Relabel *operations.*

(b) $O(nm)$ *saturating* Push *operations.*

(c) $O(n^2m)$ *nonsaturating* Push *operations.*

Proof. See Goldberg and Tarjan [28, lemma 6.3, lemma 6.4, lemma 6.7]. □

Theorem 3.5.3 (Complexity due to basic operations in cost scaling)**.** *Within an invocation of* Refine*, the basic operations contribute a complexity of* $O(n^2m)$.

Proof. Immediate from lemmas C.4.4 and C.4.5. □

Remark C.4.2. It is not possible to prove a bound on the complexity of the generic refine routine given in algorithm 3.5.2, as it depends on how vertices and arcs are chosen on how excess vertices are selected on lines 6 and 7. Bounds on specific implementations of refine are given in §3.5.3.

Theorem 3.5.4 (Complexity of cost scaling)**.** *Let* $R(n, m)$ *be the running time of the* Refine *routine. Then the minimum-cost flow algorithm described in algorithm 3.5.1 runs in* $O(R(n, m) \lg(nC))$ *time.*

Proof. (Adapted from Goldberg and Tarjan [28, theorem 4.1])

The algorithm terminates once $\epsilon < 1/n$, which takes place after $\log_2\left(\frac{C}{1/n}\right) = \log_2(nC)$ iterations of lines 4 to 6. The loop condition on line 4 and the assignment on line 5 are both $O(1)$, so the dominant cost of each iteration is the execution of Refine. Lines 4 to 6 thus contribute a cost of $O(R(n, m) \lg(nC))$. This dominates the initialisation on lines 4 to 6, and thus is the overall cost of the algorithm. □

C.4.3 Heuristics

Goldberg has proposed a number of heuristics to improve the real-world performance of his cost scaling algorithm [29], described in §3.5. These have been found to result in considerable real-world improvements in efficiency [10, 47]. Note that their effectiveness depends on the problem instance. Moreover, several of the heuristics such as arc fixing and potential update are highly sensitive to parameter settings or other implementation choices.

Potential refinement

The REFINE$(\mathbf{x}, \boldsymbol{\pi}, \epsilon)$ routine is guaranteed to produce an ϵ-optimal flow. However, it may also be ϵ'-optimal for $\epsilon' < \epsilon$. This heuristic decreases ϵ while modifying the potentials $\boldsymbol{\pi}$, without changing the flow $\mathbf{x}$. This has been found to yield a 40% performance improvement [10].

Push lookahead

Before performing PUSH(i, j), this heuristic checks whether j is a deficit vertex ($e_j < 0$) or if j has an outgoing admissible arc. If so, the PUSH operation proceeds.

Otherwise, the PUSH operation is aborted, and RELABEL is performed instead. Were the PUSH operation to be performed, the flow would get 'stuck' at vertex j, and end up being pushed back to i. This heuristic has been found to significantly reduce the number of PUSH operations performed [29].

Arc fixing

The algorithm examines a large number of arcs, sometimes unnecessarily. It can be proved that for any arc (i, j) with reduced cost satisfying $\left|c_{ij}^{\pi}\right| > 2n\epsilon$, the flow x_{ij} is never again modified. These arcs can safely be *fixed*: removed from the adjacency list examined by the algorithm.

This heuristic takes things a step further, *speculatively* fixing arcs whenever $\left|c_{ij}^{\pi}\right|$ is above some threshold β. Speculatively fixed arcs may still need to have their flow updated, but this is unlikely. Consequently, the algorithm examines these arcs very infrequently, unfixing arcs found to violate reduced cost optimality conditions.

Potential update

The RELABEL operation given in algorithm 3.5.3 updates the potential at a single vertex. This heuristic is based around an alternative *set relabel* operation, which updates potentials at many vertices at once.

Regular relabel operations are still used, with set relabel called periodically. The optimum frequency is dependent on both the problem and implementation, but calling set relabel every $O(n)$ regular relabels has been found to be a good rule of thumb [29].

This heuristic improves performance when used on its own, but has little effect when used in conjunction with potential refinement and push lookahead described above. In fact, it may even *harm* performance in some cases, although it has been found to be more beneficial the larger the network instance [10].

Appendix D

Details of tests

D.1 Machine specifications

All test machines had the same specification, with:

- Intel Xeon E5-2430L CPU.
  - 6 physical cores, each running at 2.4 GHz.
  - In addition, hyperthreading (Intel's term for simultaneous multithreading) is supported, so each physical core presents as two logical cores. For benchmarking, this is undesirable, as two test tasks running on the same physical core could interfere with each other. Accordingly, the number of test instances per machine was limited to the number of physical cores.
- 64 GB of system memory.
- Ubuntu 14.04 *Trusty* operating system.

D.2 Test harness

Running the tests manually would be unwieldly and error-prone. I thus developed a test harness in Python to automate the process. The configuration of the harness includes a list of implementations and datasets. Individual test cases in the configuration specify which implementation to test and what datasets to use, along with other parameters.

In addition to testing the final versions of the solvers, the harness was used to evaluate the impact of optimisations implemented throughout the project, as reported in §4.3. To support this, the harness was integrated with the Git version control system [54]. Versions of an implementation can be specified by a Git commit ID.

I anticipated that this project would involve running a large number of experiments. The harness was therefore designed to make the process as simple as possible. Test

```
STANDARD_TRACE_CONFIG_TEMPLATE = {
    "small": { "percentage": 1 },
    "medium": { "percentage": 5 },
    "large": { "percentage": 25 },
    "full_size": { "percentage": 100 },
}
_STANDARD_TRACE_CONFIG_1HOUR_EXTENSION = {
  "trace": "small_trace",
  "runtime": absoluteRuntime(3600)
}
STANDARD_TRACE_CONFIG_1HOUR =
  { k : extend_dict(v, _STANDARD_TRACE_CONFIG_1HOUR_EXTENSION)
        for k, v in STANDARD_TRACE_CONFIG_TEMPLATE.items()}
```

(a) Definition of a dataset to be used in tests of the incremental solver.

```
INCREMENTAL_TESTS_ANYONLINE = {
    # ...
    "same_ap": {
      "traces": STANDARD_TRACE_CONFIG_1HOUR,
      "timeout": 60,
      "iterations": 5,
      "tests": {
          "full":           { "implementation": "f_ap_latest" },
          "incremental":    { "implementation": "i_ap_latest" },
        },
    },
    # ...
}
```

(b) Definition of a test of my incremental solver (the results of which appear in figure 4.16a). Note it is defined in less than ten lines of code, important given the number of tests which needed to be specified.

Figure D.1: Extract from the configuration of my benchmark harness.

cases can be specified succinctly, shown by figure D.1. Running the experiment is simple: the harness checks out appropriate versions of the code and builds the implementations without user intervention, returning once the results of the test are complete.

Efficiency was also a key requirement: with many experiments, it is important that they complete as fast as possible. The harness reuses intermediate computation where possible. Implementations are only checked out and built once. A cache of generated flow networks is maintained, so that the overhead is only incurred on the first run.

D.3 Evaluation of compilers

Figure D.2 shows the complete set of results from the compiler optimisation experiment described in §4.3.4. In all cases, GCC O3 was either the fastest compiler, or highly competitive (within a few percent) of the fastest.

Interestingly, optimisations yield the biggest performance improvement on implementations where little development time has been spent on hand-optimising the code. The naïve implementation of cycle cancelling in figure D.2a enjoys a very considerable speedup from optimisation, although it remains one of the slowest algorithms overall[1]. My other implementations enjoy moderate speedups. By contrast, Goldberg's cost scaling implementation experiences only a slight speedup of around 25%. Frangioni's relaxation implementation has statistically indistinguishable performance at O3 and at O0.

D.4 Probability distributions in cluster simulator

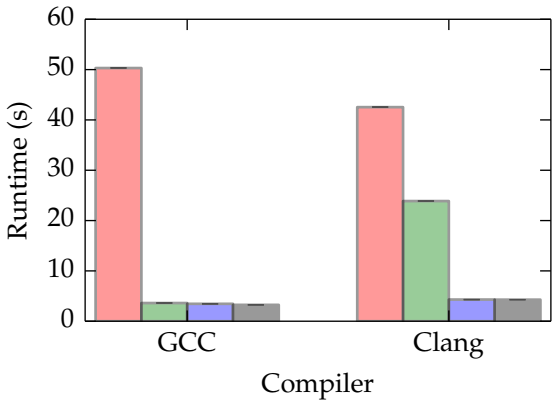
D.4.1 Task runtime

Figure 2 in Reiss *et al.* [56] shows an inverted CDF of task runtimes in the Google trace. On the log-log scale, the inverted CDF is approximately a straight line, justifying an inverted power law $F(x) = 1 - ax^k$. Parameters $a = 0.298$ and $k = -0.263$ were computed from points on the figures, $(x_0, y_0) = \left(10^-2, 1.8 \times 10^4\right)$ and $(x_1, y_1) = \left(6 \times 10^2, 10^3\right)$.

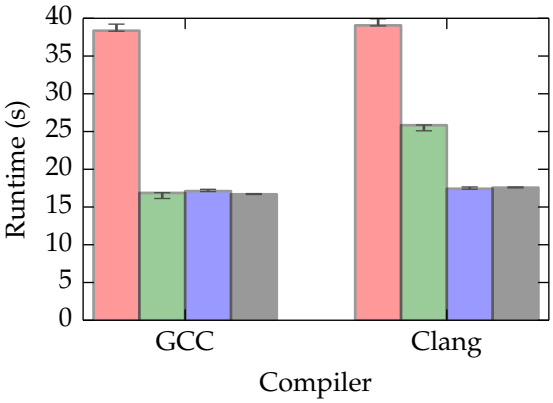
D.4.2 Task input size and file size

Figures 1 and 3 in Chen *et al.* [12] provide CDFs for file size and task input size from a contemporary Facebook cluster. At very small and very large sizes, the empirical CDF does not follow any standard distribution. However, at intermediate sizes the CDF

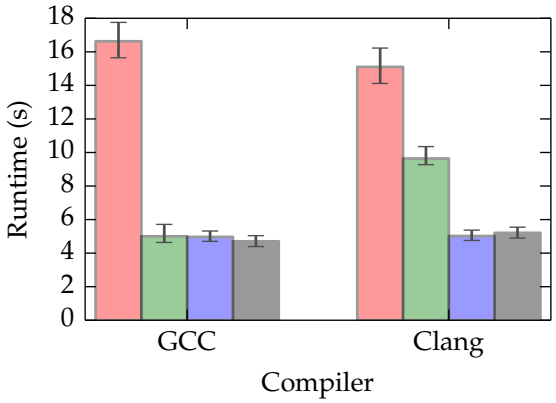
[1]Note different algorithms were evaluated on different sized datasets in this test.



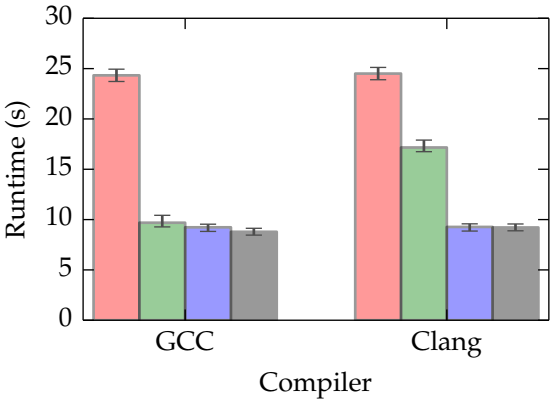
(a) Cycle cancelling, small (120 machine) cluster



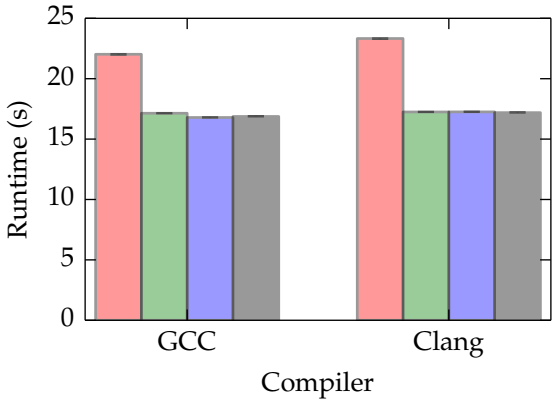
(b) Successive shortest path, medium (600 machine) cluster



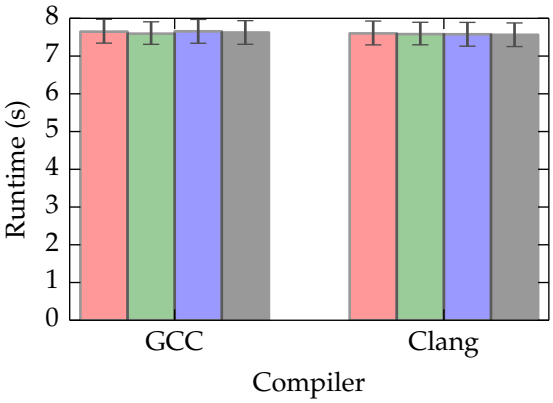
(c) Relaxation, large (3,000 machine) cluster



(d) Cost scaling, medium (600 machine) cluster



(e) Goldberg's cost scaling, warehouse scale (12,000 machine) cluster



(f) Frangioni's relaxation algorithm, medium (600 machine) cluster

Figure D.2: Choosing a compiler: runtime on GCC and Clang at varying optimisation levels. Dataset sizes were chosen so that each algorithm would have a runtime of the order of magnitude of 10s in tests. The bars represent the mean runtime over 5 replications, with error bars indicating the 95% confidence interval. Key: O0 (unoptimised), O1, O2, O3.

is approximately a straight line with the a log-scale x axis, suggesting a distribution proportional to $\lg(x)$.

The task input size CDF shows that a small proportion (less than 1%) of files are extremely large, being a terabyte or more. This long tail I believe is an artefact of MapReduce. These "files" will likely never be read in their entirety, with individual tasks processing only small segments. As there is no way to infer the true number of bytes read from the data, it seems prudent to instead truncate the distribution at the upper end. I chose an upper bound of 10 GB for file sizes, and 20 GB for input sizes of tasks.

The Google File System has a block size of 64 MB, imposing a lower bound on the distribution. Whereas only a negligible proportion of points were above the upper bound, around 50% of files and task inputs are less than or equal to 64 MB.

To model this, a mixed discrete continuous distribution was used. Letting X denote the number of blocks in a file, or in a task's input:

$$P(X = 1) = a$$

$$P(X > B_{\max}) = 0$$

$$P(1 < X \leq x) = a + \frac{1-a}{\log B_{\max}} \lg x, \text{ for } x \in (B_{\min}, B_{\max})$$

where $B_{\max} = 160$ blocks (10 GB) for file size $B_{\max} = 320$ blocks (20 GB) for input size, and $a = 50\%$ in both cases.

Appendix E

Project proposal

Adam Gleave
St John's College
arg58

Part II Project Proposal

Distributed scheduling using flow networks

Thursday 23rd October 2014

Project Originators: Ionel Gog

Resources Required: Yes, please see section Special resources

Project Supervisors: Ionel Gog

Director of Studies: Dr Robert Mullins

Overseers: Dr Stephen Clark & Dr Pietro Lió

Introduction & description

The usage of clusters of commodity computers has now become the ubiquitous paradigm within major web companies. Whilst these so-called *warehouse-scale computers* offer many advantages, programming for this environment is often a challenge.

Most development for warehouse-scale computers takes place on top of some distributed system framework. These take many forms, with the data-flow oriented MapReduce the best known example. A common theme is that these frameworks include a *scheduler*, mapping individual tasks to machines.

The goal of the scheduler is to make the best possible use of the finite resources of the cluster. To aid it in this mission, the framework may provide the scheduler with additional information. For example, the programmer could use flags to specify whether the task is CPU and/or memory hungry. In a data-flow engine, the framework knows where the inputs to the task are stored.

The state of the art within major cluster operators is proprietary. However, widely-used public implementations tend to use a simplistic queue-based approach. An example of this is the scheduler for Hadoop, by far the most widely used open-source framework.

The Hadoop scheduler achieves admirable data locality: scheduling tasks close to the the sources of their input data. However, it has a notable drawback: very unfair schedules are produced in some common cases.

Alternatives have been proposed, addressing the issue of fairness [70]. However, these approaches are still flawed. Most importantly, they ignore factors besides data locality that are relevant to scheduling: for example, the CPU and memory usage of tasks.

Ad hoc approaches to scheduling can never capture the full complexity of a system as large as warehouse-scale computers. For a comprehensive solution, I would argue it is necessary to build a model of the data centre. This is precisely the approach taken by the Quincy system, developed at Microsoft Research [43], which models the data centre as a flow network. As expected, their system considerably outperformed a queue-based approach.

Disappointingly, however, the cost of solving the min-cost flow problem was prohibitively expensive. Despite the schedules providing excellent performance, the runtime of the scheduler was increased by such a great amount that it negated the performance benefits for short-lived jobs. For sufficiently long-lived jobs, it may offer a performance advantage (with the scheduling cost being amortized over a longer time period), but it raises questions as to the scalability of the system in larger data centres.

My proposal is to implement flow algorithms more suitable to this problem domain. Whereas Quincy used a standard algorithm to find an optimal solution to the flow

network, it seems likely that an approximate solution would work almost as well. Even if the schedules resulting from an approximate algorithm were noticeably different, it is likely they would still outperform a queue-based approach. Of course, an approximate solution can be found much more quickly.

Another approach is to exploit unique characteristics of the flow networks being modelled, which general purpose flow algorithms cannot take advantage of. For example, the scheduler is rerun every time a new task is submitted to the scheduler, and whenever a task completes in the system (resulting in a node becoming idle). The flow network will therefore be *mostly unchanged* between each iteration of the scheduler. A so-called 'incremental' solver exploiting this could yield drastic performance gains, as has been the case in related problem domains (such as shortest path problems).

Unfortunately there is no guarantee that this problem is tractable, so it cannot form a core deliverable for this project. However, incremental changes are rare in flow networks. The fact that there is no extant algorithm may therefore reflect more on the uniqueness of this problem, than on the intrinsic difficulty of developing such an algorithm.

Whilst an improved flow solving algorithm for this problem would have immediate practical impact, there are other areas to explore which are of considerable theoretical interest. The model used in the Quincy paper considered only a small subset of the information that is available to a scheduler. For example, it would be possible to take into account the predicted runtimes of jobs. There are also a number of parameters in the paper that must be hard-coded by the operator. Making the system tune these automatically has the potential to improve both performance and usability.

Special resources

Personal machine

My personal Linux-based laptop, for development purposes. This is an Intel core i5 machine with 6 GB RAM. I accept full responsibility for this machine and I have made contingency plans to protect myself against hardware and/or software failure.

To protect myself against data loss in the event of hardware failure or accidental deletion, I will be using the Git version control system, pushing changes to GitHub's servers (outside of Cambridge). Furthermore, I will make automated and manual backups to the MCS and an external hard disk.

Systems Research Group (SRG) cluster

Whilst most of the development and testing of the project can take place on my personal machine, I anticipate occasionally requiring more resources:

- Dedicated server or high-specification virtual machine. For testing new algorithms or cost functions, the results will be computed much more quickly on a server than on my laptop. It would be possible for me to continue development without this resource, but at a slower pace.
- Occasional access to the SRG cluster. Whilst most of the testing can be carried out on a single machine, it is prudent to verify that the expected performance is attained in practice. This can be done by modelling the SRG data centre, and evaluating the scheduler on real-life jobs. I would anticipate requiring access to the SRG cluster towards the end of the project for a brief period of time for these tests.

Starting point

Whilst working on this project, I will be building on the following resources:

- **Existing research**
  There is considerable existing research into flow algorithms. Implementation of an existing algorithm, to use as a performance baseline, forms one of the first deliverables of the project. There are a number of algorithms in the literature suitable for this purpose [29, 47].

  I also anticipate building on the work of existing researchers when devising my own algorithm. For example, Goldberg[28] describes an algorithm using successive approximation to find an optimal solution. This forms an excellent starting point for developing an approximate solver.

  Whilst a literature review shows little in the way of research into incremental flow algorithms, there are a number of published algorithms in related fields. It is probable that some of the techniques used to solve related problems, such as shortest path problems, will also be of use here [55, 59].

- **Firmament**
  A distributed data-flow execution engine under development in the SRG, with lead developer Malte Schwarzkopf. The project includes a representation of data centres as a flow network.

- **Programming experience**
  I will be drawing heavily on my prior programming experience, gained from the Tripos and summer internships. Most of my experience has been in either functional languages, such as OCaml, or scripting languages, such as Python. By contrast, for this project I will likely be working in a systems programming language, such as C or C++. My experience in these languages is limited to that gained from the Tripos. Familiarizing myself with the implementation language will form part of my preparation time.

Structure of the project

I propose to split the project into several phases. This will simplify project management, with each phase having associated milestones. All phases contribute towards the overall goal of improving scheduling performance.

Phase 1 – core implementation

In the first phase, a simple scheduler will be built. Data centres will be modelled as flow networks. Two different algorithms will be supported, to allow for comparison. Broadly, this phase will be divided into the following three tasks:

1. Implementation of a standard algorithm for solving min-cost problems. Some research and experimentation may be required to identify the fastest algorithm for the class of flow networks we will be working with.
2. Development of an algorithm which provides approximate solutions to the min-cost problem. This should take a parameter giving an upper bound for:

$$\frac{\text{Cost of returned solution}}{\text{Minimum cost}}$$

3. Integration of both of the above with Firmament, an execution engine for distributed clusters.

Making the initial steps simple serves several purposes. Firstly, by keeping the complexity of the code to a minimum, I will be able to familiarize myself with the process of developing new algorithms. Secondly, by postponing optimisations and enhancements to later in the project, progress can be observed early on by reaching a tangible milestone.

Phase 2 – testing and performance evaluation

On completion of the first phase, the performance of the scheduler should be evaluated. The overall runtime taken to execute a job is determined by two factors:

- *scheduling overhead*: the time taken to schedule tasks to machines. This includes not just the runtime of the scheduler, but also the time spent by idle nodes in the cluster, who are blocked waiting for the scheduler to complete.
- *computation time*: the time taken for the tasks to run on each machine.

The key concept of the project is that by paying more scheduling overhead, the computation time can be decreased due to a more efficient scheduling allocation. However, finding an optimal solution to the flow problem involves paying

considerable scheduling overhead. We should be willing to increase the scheduling overhead only so long as computation time decreases by a greater amount.

For a comprehensive evaluation, it is therefore necessary for both aspects to be measured. It is simple to compare the scheduling overhead of different algorithms: simply run them both on the same model, and measure the runtime. Since there is a monotonic relationship between the runtime of the scheduler and the total scheduling overhead, whichever algorithm has a lower runtime also has a lower scheduling overhead.

Putting a numerical value on the scheduling overhead is more tricky, however: it depends on the number of idle nodes in the cluster. There are similar problems with measuring computation time, which depends on the type of jobs scheduled. For a comprehensive evaluation, the scheduler will need to be tested on a real cluster, with a simulated workload.

Ensuring the simulated workload is representative of actual usage patterns may be a challenge. Google has released a trace from one of their data centres [68, 57], which could be used to determine an appropriate cross-section of tasks. Alternatively, it would be possible to use similar benchmarks to those in the Quincy paper [43].

There is a risk that the results returned from testing on a cluster may not generalize to other data centres. However, this is a problem common to all research in this area. To mitigate this as far as possible, the algorithm should be tested on synthetic models of much larger data centres, to verify the scheduling overhead does not increase faster than expected.

An area of considerable interest is determining how accurate to make the approximate solution. Demanding more accuracy will increase the scheduling overhead but decrease the computation time, so this is a special case of the trade-off described above. For a specific workload on a particular cluster, it is possible to determine this value empirically.

Phase 3 – enhancements and optimisations

The final phase will be dedicated to optimizing the flow solving algorithm, and adding extensions to the data centre model. My current ideas are summarized in Possible extensions. It is also anticipated that new ideas may emerge during preparatory research, and implementation of the algorithms above.

Possible extensions

The idea of modelling data centres as flow networks is relatively recent, having been published only in 2009. Consequently, there are a wide variety of promising areas which have yet to be explored, giving considerable scope for extension. Here, I present a few of the most promising ideas:

- **Incremental flow algorithm** – this holds the potential of a considerable reduction in scheduling overhead. Crucially, it is likely that such an algorithm would yield not just a constant factor reduction, but also a decrease in the average-case asymptotic complexity. This would allow the algorithm to scale up to the biggest data centres both today, and in the future.

- **Parallel flow algorithm** – the cost of running the scheduling algorithm is negligible. Most of the scheduling overhead comes instead from other nodes in the cluster being unnecessarily left idle, waiting for the cluster.

  Because of this, we care more about the *span* of the flow algorithm, than we do about the total work expended. It would therefore be highly desirable to use a parallel algorithm for this task.

  Unfortunately, flow algorithms are known to be hard to parallelise. It is therefore unclear how tractable this problem is, although some progress has been made [17].

- **Automatic parameter tuning** – there are a number of parameters present in the system, which it is unclear how best to set. One such example is the parameter controlling the accuracy of the approximation. There are also parameters which control the cost imposed on traffic across the network.

  It would be highly desirable, in terms of both performance and usability, for the system to automatically set these parameters. As the optimal parameter is likely to vary depending on the workload, this may outperform any hard-coded value, even if considerable effort was expended to choose that value.

- **Multi-dimensional cost function** – in the original Quincy paper, the cost of edges is determined purely by the amount of data that needs to be transferred. We have other information available to us, such as the CPU and memory usage of a task. A multi-dimensional cost function, such as a weighted sum of these factors, may provide superior performance.

- **Predicting the future** – in general, better solutions are possible to the offline scheduling problem than to online scheduling. In offline scheduling, we have perfect information as to the resource requirements and runtime of each job. Unfortunately, in practice we only have this information after the event: we must write online schedulers.

  Whilst we cannot hope to invent a crystal ball, many jobs run on clusters have predictable resource requirements. It may be possible to estimate the resource requirements of a newly submitted task, from the results of previous runs. Alternatively, the developer of the job can provide hints to the scheduler. By incorporating this information, it is possible for the scheduler to make significantly better allocations.

- **Many-to-one allocation** – only one-to-one allocation of tasks to machines was investigated in the Quincy paper. Extending this to be a many-to-one allocation may enable a performance boost. For example, a cluster may have a mixture of

computationally-intensive and IO-intensive tasks. If two such tasks run on the same machine, the CPU-intensive task can run whilst the IO-intensive task is blocked, increasing throughput.

There is a danger this could actually harm performance for certain workloads: for example, two memory-intensive tasks running on the same machine might result in thrashing occurring.

A simple approach would be to add flags to each job, describing its resource requirements, and then pair tasks suitably. A more sophisticated technique would be to make the flow model more fine-grained. Rather than just describing the network, it could be extended to represent the CPU and memory resources in each machine. For this to be successful, some estimate of resource requirements for a newly submitted job will be necessary. This extension is therefore closely related to the above enhancement.

Success criteria

1. **Standard algorithm** – Implementation of an existing min-cost flow algorithm, described in the literature.
2. **Approximate algorithm** – Implementation of an algorithm providing an *approximate* solution to the min-cost flow problem.
3. **Speed evaluation** – Measuring the runtime of the approximate and standard algorithm on example flow networks.
4. **Simulated workload** – Devise a set of programs that can be used as a simulated workload, when evaluating scheduling performance.
5. **Performance evaluation** – Testing the overall performance of the two algorithms on a real cluster, scheduling the above tasks. The results of this test will depend not just on how fast the scheduler runs, measured above, but also on the decisions made by the scheduler.
6. **Accuracy trade-off** – Running the approximate algorithm for longer will yield a more accurate solution. Explore the effect of increasing and decreasing the accuracy of the algorithm on overall system performance.

Timetable: work plan and milestones

I have split the entire project into work packages, as recommended in the Pink Book. For most of the project these are fortnightly, but become longer towards the tail-end of the project.

By the end of Lent term, I intend to have completed implementation and testing, and have produced a first draft of my dissertation. This affords sufficient time to polish

the dissertation over the Easter vacation, and revise courses for the exams during Easter term. If necessary, the Easter vacations could also be used as buffer time, to resolve any outstanding issues with the software.

- **10th October to 24th October**
  - Background research into the problem area.
  - Writing the project proposal.

- **24th October to 7th November**
  - Familiarize myself with the Firmament code base.
  - Additional research into flow algorithms.
  - Configure development environment.
  - Test backup procedures

- **7th November to 21st November**
  - Implementation of baseline algorithm.
  - Begin investigating workloads for performance testing of the scheduler.

- **24st November to 5th December**
  - Design of approximation algorithm.
  - Integration with Firmament code base.
  - Initial performance test on SRG cluster.

5th December – end of full Michaelmas term

- **5th December to 19th December**
  - 5th-12th December: Vacation, no Internet access.
  - Further time for testing.

Milestone: Phase 1 and phase 2 complete

- **19th December to 2nd January**
  - Write progress report.
  - Buffer time.

  Milestone: Progress report draft

- **2nd January to 16th January**

  Start implementation of phase 3 (optimisations and enhancements):

  - Begin implementation of the extensions outlined in Possible extensions on page 119.
  - Pursue additional extension possibilities which have emerged from research or development earlier in the project.

13th January *– start of full Lent term*

- **16th January to 30th January**
  - Complete and hand-in progress report. Update to include any new developments.
  - Write presentation for overseers and other students. To include preparation of figures or other material summarizing test results.
  - Continue implementation of extensions..

  Deadline: Hand-in the progress report

  Milestone: Presentation ready for early February

- **30th February to 13rd February**
  - Development of extensions continues.
  - Deliver progress presentation.
  - Buffer time.

- **13th February to 27th February**

  Final opportunity for last minute enhancements or additions.

  - In-depth testing of the scheduler. This should produce data suitable for the dissertation write-up, which commences in the next work package.
  - Wrap up any final extensions.

  Milestone: Scheduler implemented and tested, including the optional extensions previously selected.

  Code freeze: only bug fixes allowed henceforth.

- **27th February to 13th March**

  Begin writing dissertation. Buffer time for any outstanding issues.

  13th March *– end of full Lent term*

- **13th March to 27th March**

  Write evaluation section, based on test results found previously. Write outline of other sections.

- **27th March to 10th April**

  Write all remaining sections. Typeset to standard suitable for review by supervisor and others.

  Milestone: Dissertation draft

- **10th April to 24th April**

  Finish dissertation, incorporating feedback from reviewers.

  21st April *– start of full Easter term*

- **24th April to 15th May** (note: extra long)

  Buffer time. Address any outstanding issues with the dissertation.

  Deadline: Hand dissertation in, and upload source code archive.