Shortest Paths and Network Flow Algorithms for Electrostatic Discharge Analysis

Robert B. Gramacy
rbgramacy@cse.ucsc.edu
U.C. Santa Cruz
Senior Thesis

June 4, 2001

Abstract

Designers of integrated circuits have always been concerned about the damaging effects of ESD (Electrostatic Discharge). However, as their designs become increasingly more complex, the problem of analyzing the schemes they develop to protect against ESD is quickly becoming too difficult perform manually. Current VLSI (Very Large Scale Integration) designs exceed one hundred bondpads, through which the device communicates with the outside environment, and through which harmful amounts of current might enter the chip. A protection scheme for a complex device such as this might require a network of thousands of non-trivial connections between the bondpads to keep harmful amounts of current from the chip's delicate core. In order to measure the robustness of a protection scheme designers would like to be able to take resistance measurements between bondpad pairs. On such large designs, this can be a monumental task. In this paper we focus on two things: the automation of strategies previously performed manually, and an in-depth development of (newer) more accurate techniques provided by network flow algorithms, each for gauging the degree of protection afforded by ESD Protection Schemes. The underlying algorithms we will discuss for automating this ESD Analysis are well understood. However, their application, and the introduction of more advanced simulation techniques make this discussion original. We would like to develop a tool which not only accurately models the protection scheme against an ESD event, but also helps to pinpoint its weaknesses. In many cases it is not obvious that the techniques we suggest can be applied or adapted to the simulation of electrical devices and networks. It is also interesting that such techniques make no direct use of Kirchoff's Voltage or Current Laws (KVL/KCL) yet still produce meaningful and accurate results. This document is intended to summarize standard techniques for analysis of ESD protection schemes, as well as to introduce some clever variations on more complicated algorithms whose intent is to generate more meaningful results never before feasible manually.

1 Overview

One of the most harmful environments for integrated circuits is the human hand. The human body stores electric potential which can cause irreparable damage to integrated circuits if not averted. When static electric potential is unleashed the event is called *Electrostatic Discharge* or ESD. In order to protect these delicate circuits from ESD, designers incorporate protection hardware into the chip's input/output (I/O) package to divert harmful amounts of current away from the core of the device. Once such hardware is developed it needs to be tested, and debugged. It is much cheaper to analyze the integrity of a protection scheme in the *Computer Aided Design* (CAD) environment than it is to perform physical simulations on fabbed chips. Hence, designers are interested in simulating ESD events in the CAD environment so that the layouts/schematics for the protection schemes can be modified prior to an expensive fabrication process. This paper addresses the development of software for the simulation of ESD events, and subsequently the analysis of ESD protection schemes. Such an analysis can occur in several stages with several strategies, from debugging plans and schematics, to an actual event being simulated on the final layout. Automations of schematic level simulations have most recently been addressed in an article supporting a scheme entitled *VerifyESD* [3]. We are concerned with simulations on layouts which are more difficult to do by hand, and for which there is currently no software available. But first, before we get into the actual techniques, it is necessary to situate the problem in a more well-defined environment: enumerate all of the variables, and more specifically depict structure of ESD protection schemes. After constructing this perspective, and following a general discussion of the input and instances of our problem, the complex nature the ESD analysis problem we are considering will become more apparent.

Typically, an integrated circuit is entirely surrounded by an I/O ring wherein any ESD protection hardware resides. The I/O ring interacts with the outside environment through a series of *bondpads*, or pins, through which all electrical current enters and exits the integrated circuit. The basic theme of the ESD protection schemes we wish to analyze is based on diverting dangerous amounts current from one bondpad (the *zapped* bondpad) to a bondpad with a connection to *ground* without it passing through the delicate core of the chip. The success of a protection scheme can be measured by the *voltage drop* between zapped and grounded bondpads. This voltage drop (or *excitation level*) for each bondpad pair is commonly referred to as the *ESD Budget* for that pair. If the excitation level is too high then dangerous amounts of electrical current will likely trigger the gate-oxide, and thereby leak out of the I/O ring and possibly damage the circuit. We are interested in measuring the excitation levels between bondpad pairs in order to aid designers in developing more robust ESD protection schemes for their IC's. The input to such an analysis is the I/O package to be analyzed and the magnitude of the *ESD Current* incident on the device. The currents corresponding to an ESD event are typically based on an experimental *Human Body Model*, or HBM.

In order to construct a more focused discussion of the underlying algorithms for analyzing ESD events, we will not concern ourselves with appropriate inputs or use of output, but rather the analysis of a protection scheme under several reasonable assumptions, and only the variables just mentioned. However, since these simulations do indeed take place in a CAD environment we will concern ourselves with developing an accurate interpretation of the I/O package, its electrical components, and the providing of meaningful output, and a reasonable, flexible and useful interface for the designer. Therefore, the first section of this paper is devoted to CAD issues, the representation of electrical networks (*netlists*), allowing for the capability of the user to define the simulation of the ESD event globally, the individual components used in ESD protection hardware locally, and most importantly to describing a graphical representation of electrical devices. Any tool for analyzing ESD events on an arbitrary layout will have to accurately model the devices it contains. In order to simplify this discussion we concern ourselves with only a basic set of devices, and argue that such a set is sufficient for a thorough discussion and simulation, placing no serious limitations on input or implementation.

In the next section we address the type(s) of analysis which were previously done by hand. These techniques take the form of overestimating excitation levels between pairs of bondpads by guessing the path of least resistance between them. We first formalize this technique and prove that it yields the desired overestimate. Techniques of this flavor have their analog in Shortest Path algorithms such as those suggested by Dijkstra and Floyd-Warshall. We will discuss these algorithms, their utility and complexity, and the meaning of the data they produce. We also discuss the grossness of the overestimate obtained by using these algorithms, devise some techniques for obtaining a less pessimistic overestimate, and argue that any drastic

improvements require a different algorithmic strategy. The expected efficiency of these algorithms in practice can vary greatly from the results suggested by their theoretical analysis. These topics will be discussed as part the decomposition of each algorithm, along with implementation issues, solutions, and shortcuts as they arise in our unique electrical environment.

The third section of the paper is devoted to techniques for solving the voltage drop problem exactly. This will require techniques from the Network Flow class of Linear Programming algorithms. In this section we begin by addressing the motivation for such an approach, as well as its feasibility. Next, we discuss Network Flow algorithms in general, and the assumptions we must impose on the data in order to exploit them. Following, we discuss the class of Network Flow problems which are best suited to our needs: *Minimum Cost Maximum Flow* (MCMF), its algorithms, and a slight variation of this class called *Convex Flows* which best suits the electrical model we are trying to simulate. Once enumerating this class of problems and their motivation, we will discuss the *Successive Shortest Paths Convex Flow* MCMF algorithm in detail, as well as the deviations we take from it to get the output we require with the greatest possible efficiency. This discussion relies heavily on the data structures used to represent graphs, the concept of residue and residue graphs, their properties, and the optimality principles inherent in the MCMF problem. After enumerating the relevant optimality conditions, one in particular is discussed in detail– reduced cost optimality conditions. From this we derive the *Successive Shortest Paths Algorithm* and discuss it in detail. Afterwards, we discuss a variation on the Successive Shortest Paths Algorithm where by scaling flow augmentations and subsequently reducing the scale in phases we can reduce the algorithm's theoretical complexity. Finally, we enumerate any limitations our initial assumptions impose, and suggest implementation schemes which avoid them. As this section is the heart of the paper, it will contain the most rigorous mathematical justification, treatment of the analysis, and detailed description of the techniques. As already mentioned, the class of Minimum Cost Maximum Flow problems yields several different algorithmic approaches of varying complexity. However, since each approach is based on similar optimality conditions, we discuss these conditions in detail, but focus on only a limited set of algorithms which are particularly well-suited to our task, leaving many of the other algorithms which exploit these and similar conditions to our references. In addition, we will also suggest paths of further research and similar techniques though in much less detail.

2 Preprocessing & Graphing Electrical Components

2.1 Preprocessing

We begin by discussing the Computer Aided Design (CAD) environment, the typical input to our ESD analysis problem, and the preprocessing required to obtain such a suitable input. As mentioned in the overview section, there are two types of inputs on which an ESD analysis might be performed: schematic, and layout. The schematic-level can be characterized by a symbolic description of an integrated circuit, consisting of symbols which represent diodes, transistors, etc. On the other hand, a layout describes the integrated circuit and its components in terms of layers of metal, silicon, etc. We are interested in layout-level analysis for the following reasons:

1. There already exist tools for schematic-level ESD analysis (see *VerifyESD* [3]).
2. Layout extractions tend to produce quite large netlists which make other recursive/exhaustive approaches (as suggested by *VerifyESD*) inappropriate.
3. Layout extractions produce netlists which use devices from a small subset of components (resistor, diode, transistor, net connections) and which have simple graph (arc) representations.
4. Schematic-level representations lack any information about the parasitic resistance across metal layers and long busses of metal which greatly impact the voltage drop measurement across a pair of components when the IC is stressed during an ESD event.

Since a typical layout representation itself is a poor representation of a graph and electrical makeup of the device from a programming perspective (simply because is it more of a diagram than a list of connections) we require a tool which can extract the necessary information (parasitic resistances, physical components, etc.) from the diagram, and a tool which can make a so-called *netlist* from this extracted information. It is this netlist which is the input to our ESD analysis routines. The netlist is an ASCII representation of the electrical network represented by a layout. The netlist describes both where the components in the layout are connected, how they are connected, and give a device hierarchy. Since all we wish to measure is the voltage drop (or excitation level) between pairs of components in the I/O portion of the integrated circuit, we are only interested in a subset of the layout, and therefore only a subset of the components and connections in its netlist. Since the purpose of an ESD protection scheme is simply to divert harmful amounts of current away from the core of the chip, we can make several assumptions which further simplify the input:

1. Current incident on the device in a simulation always corresponds to an ESD event. Thus, only the components which might be involved in such an event are relevant to the analysis. All others can be excluded.
2. Since we are only concerned with the direction of the current and the excitation level between pairs of components, we need only consider components which exhibit directional, resistive, and hierarchical properties:
   i. resistors
   ii. diodes
   iii. non-resistive, undirected connections
   iv. transistors
   v. larger (blackbox) single or multi-directional parasitics
   vi. If the netlist has any hierarchy information (which is usually desired), then this hierarchy can be encoded in terms of the above types of connections, and yet require no other devices for our purpose.

The term *blackbox* refers to complex ESD protection devices themselves, such as latchbacks and crowbars which, as a functional unit, behave as one resistive and/or directional device during an ESD event. In other words, it can be thought of as a large diode or resistor (though perhaps with a variable, or threshold triggered voltage drop or resistance).

It is easy enough to see that none of the above assumptions places any real limitation on the types of layouts we can analyze. However, each (especially the last) will often require another level of preprocessing, in the form of compiling the original netlist (*raw* netlist) into another netlist– one more friendly to our analysis. In other words, netlists often come in several flavors, and many do not represent the extracted layout as a single, flat, sequence of connections. Instead, subcomponents of the device may be enumerated individually, and then referred to as devices of other subcomponents, or sections of a larger device. Thus we require a compiler, or set of compilers, in order to translate an arbitrary netlist to one which is explicit and flat (every connection is described at the resistor, diode, connect, or blackbox level). In addition, we want this compiler to preserve hierarchy so that a complimentary compiler can also be written to translate the results back into the layout domain.

We now conclude our discussion of the input so as to proceed with the general discussion of ESD analysis based on the above assumptions.

2.2 Graphing, Representation of Electrical Components

In short, we would like to represent each of the components in the above list (resistors, diodes, non-resistive connections, transistors, blackboxes) graphically. Connection types (i-iii) have a natural *arc* graphical representation, which will be discussed, briefly, below. Connection type (iv), transistors, have a less natural graphical representation. The main reason for this is the following: connection types (i-iii) have two terminals, but transistors, of which there are many flavors, typically have three or more. In the type of analysis we are performing we need not consider transistors as part of our model. That is, they (or groups of them) can often be abstracted by a blackbox component. Therefore, we will leave the discussion of transistors to the Further Considerations section of this paper, and furthermore leave them out of our general analysis, making only anecdotal references to them from time to time. While we do not require blackbox components to have just two terminals, we do insist that in the netlist-preprocessing phase each of its pairs of terminals can be described as a single resistive and/or directional component. In this way, one blackbox with two or more terminals can be described as a network of two terminal devices, each with the same name. This reduces the problem of representing a blackbox component graphically to describing it as sequence of resistors, diodes, and/or non-resistive connections.

Now that we have simplified our input model to only basic two-terminal devices, we can proceed to describe their representation in a relatively straightforward manner.

The graphs we will use for describing our electrical networks will be defined in the following way: We say that a (weighted and directed) *graph*[1] $G = (N, A)$ is a set of nodes (or vertices) N and arcs (or edges) A, where an arc is an ordered pair of distinct nodes from N. For each arc $(i, j) \in A$ we associate a cost function $c : N^2 \to R$. We say that a (directed) *path* is a sequence $P = \langle a_1, \dots, a_n \rangle$, where $a_i \in N$ and $(a_i, a_{i+1}) \in A$ for integers $i \in \{1, \dots, n-1\}$, and $n \geq 2$. On such a path we say that a_i is the *parent* of a_{i+1}. Since the arcs have a cost associated with them, the cost of traversing this path is $\sum_{i=1}^{n-1} c(a_i, a_{i+1})$. A *cycle* is a path $\langle a_1, \dots, a_n \rangle$ together with the arc (a_n, a_1) or equivalently $\langle a_1, \dots, a_n, a_1 \rangle$. We leave other definitions of graphs and their properties to our references[2], and enumerate them as they become relevant to the discussion. For example, as we begin talking about Network Flows in the third section of this paper we will introduce the concept of flow and residue, demand at a node, etc. An example pictorial representation of a graph is provided in Figure 1.

We now describe the central topic of this section: representing our three connection types graphically. A *resistor* has two terminals, which we call A and B, and the resistance between them we will call r_{AB}, measured

[1] We adopt the Graph representation used in the text by Ahuja, Magnanti & Orlin *Network Flows*[1]. For other, equivalent graph representations, such as $G = (V, E)$, please refer to the references concluding this document.

[2] See [2], [1], and [6]

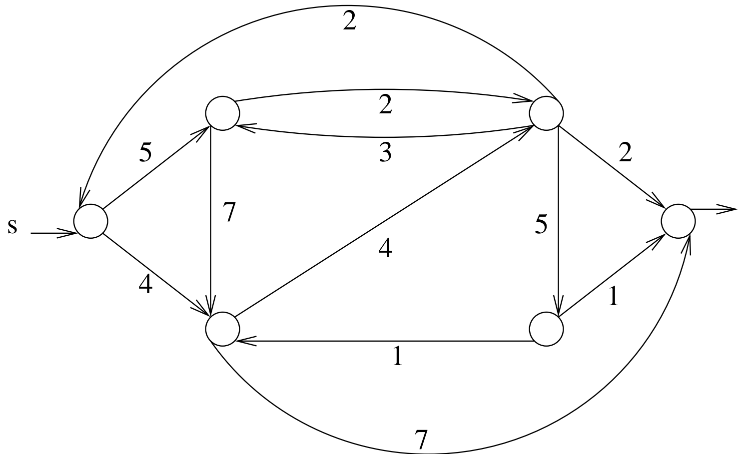


Figure 1: A simple example of a graph. Each arc is associated a cost which is beside it. The nodes s ant t can be thought of as the source and sink for the graph (i.e. the zapped and grounded terminals in an ESD event).

in ohms (Ω). See Figure 2. Current may flow along a resistor from A to B or from B to A. Therefore, a resistor can be thought of simply as a bi-directional arc with a bi-directional cost between nodes A and B which is equal to its resistance. Since our definition refers to arcs as only having one direction we must express our resistor graphically as two arcs between A and B, one for each direction. With this in mind we create an arc (A, B) with $c(A, B) = r_{AB}$, and an arc (B, A) with $c(B, A) = r_{AB}$.

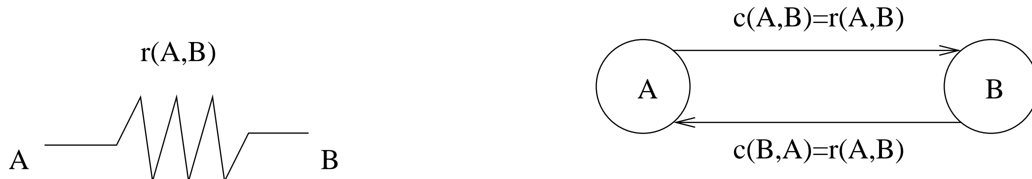


Figure 2: (Left) Electrical representation/notation for a resistor with terminals A and B. $r(A, B) \equiv r_{AB}$ (the resistance between terminals A and B). (Right) Graphical (edge/arc) representation/notation of the resistor on the left.

Similarly, a *diode* has two terminals– again call them A and B– but we only allow current to flow in one direction (see Figure 3). Diodes also have an associated voltage drop[3] ($v(A, B)$), measured experimentally (in volts) with the *Transmission Line Pulse Technique* (TLP). Since we are assuming some current I incident on our ESD protection scheme, we can write the "cost" of this current across a diode in the same units as our resistor, namely ohms. Therefore, we can represent a diode graphically as directed arc (A, B) with $c(A, B) = v_{AB}/I$. This assumes that the diode's reverse direction (opposite the arrow of the diode) has an infinite resistance all of the time, which is equivalent to there being no edge (B, A) in our graph.

[3]usually around 0.7 volts at 1.3 amperes Current, depending on the dimensions and makeup of the diode

Figure 3: (Left) Electrical representation/notation for a diode with terminals A and B. $v(A,B) \equiv v_{AB}$, (the voltage drop from terminal A to terminal B). (Right) Graphical (edge/arc) representation/notation of the diode on the left.

Finally, we have left to describe the graphical representation of a non-resistive, bi-directional connection. However, this should not be too difficult, since this is essentially the same as a resistor with zero resistance. In fact, this is exactly the situation in which such an entry is called for in a netlist. The netlister requires a connection between two (or more) other elements of the layout where one or both (or all) of whose terminals share the same location (e.g. a bondpad and its pin, see Figure 4).

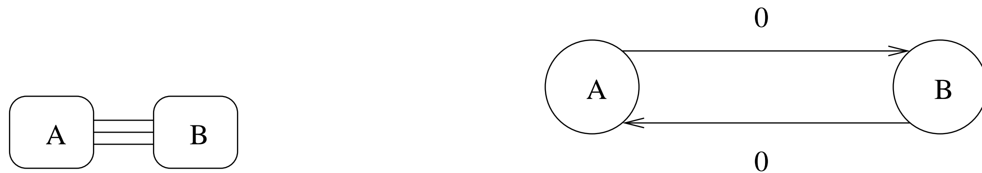


Figure 4: (Left) Electrical representation/notation for a non-resistive connection with terminals A and B. (Right) Graphical (edge/arc) representation/notation of the connection on the left.

A non-resistive bi-directional connection might be used to connect the terminals of two different diodes in order to represent them hierarchically as a back-to-back diode-pair (a standard component in ESD protection schemes). See Figure 5. Since this is a typical construction for ESD protection it is often defined as an abstracted device, or sub-circuit. Therefore, in a netlist it might appear as a single device whose definition is included somewhere else in the netlist (indicating that the sub-circuit named "back2back_diode" is really just a pair of diodes where the anode of one is connected to the cathode of the other, and vice-versa). Arcs in our graphical representation are therefore required in order to "connect" the diodes as described in sub-circuit definition. This is also shown in Figure 5. A more complicated example of such sub-circuit definitions might include an abstracted device which contains more than two shared terminals. In addition, we might come across a reference to an abstracted device which itself contains nested sub-circuits. Non-resistive hierarchical connections handle this nicely. If these connections are named appropriately, the graph can remain backward compatible with the netlist.

With an electrical network composed of the above components we can now create a graph using the arcs described above to represent the connections between terminals which will be represented as nodes. The remaining sections of this paper will cover algorithms which assume input like the graphs we have just discussed. As always, when preparing to discuss algorithms which work with data structures such as graphs we must also talk about how these structures are represented. In this discussion we make use of two representations[4]: *Adjacency List* and *Adjacency Matrix*. These will be discussed in more detail as we

[4] please refer to the references for these and other representations [1], [2], and [6]

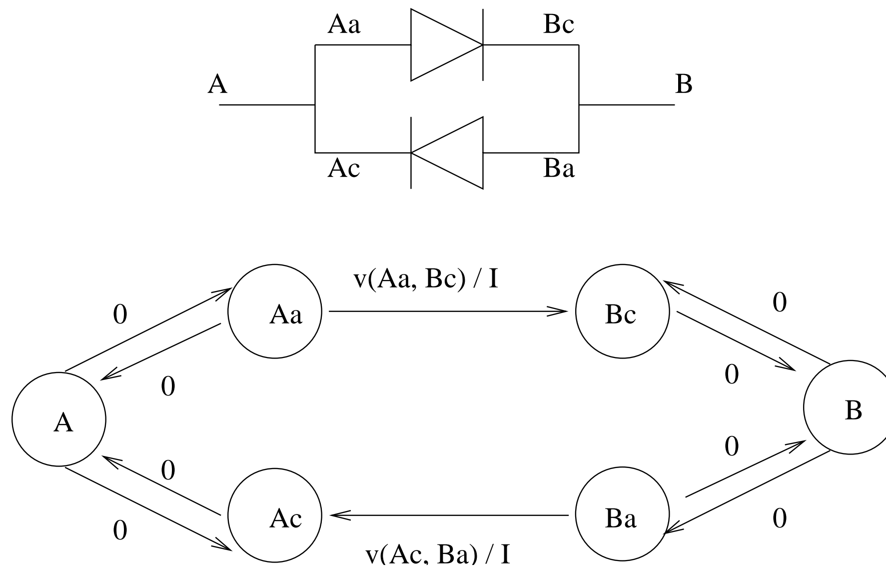


Figure 5: (Top) Back to Back diodes example of two components sharing the same terminals A and B. (Bottom) Graphical representation of the Back-To-Back diode on the Top. Extra arcs with zero resistance must be used to represent the compound device which is really two devices sharing the same terminals A and B. The "a" and "c" respectively in the intermediate node names are used denote the "anode" and "cathode" of each diode.

introduce our analysis techniques in the next sections. Hopefully, at that point, the reasons behind our choosing them will be immediately apparent.

3 Overestimating & Shortest Paths

3.1 Overestimating

We proceed now with our first discussion of estimating the voltage drop, or excitation level, between terminal pairs. This first scheme we will cover is an automation of an imalgum of techniques previously performed by hand, and often "eye-balled" by designers at the schematic level. A technique of this sort involves an overestimation of the voltage drop between pairs of components by computing the voltage drop across the single least resistive path, without considering parallel components or parallel paths of components in series. This approach yields an overestimation because parallel components, or paths on which current can flow, lessens the effective resistance just as parallel river-beds, or channels, allow a greater amount of water to flow downhill. Thus, leaving such "channels" out of the analysis yields a pessimistic estimate on how much water can be expected at the bottom of the mountain. Such techniques, and similar methods for making estimates in the realm of electrical networks, are analogous to shortest path algorithms on graphs. While somewhat feasible at the schematic level, manually running algorithms which are based on this approach is tedious, and for the most part infeasible at the layout level which involves considering parasitic resistances along long metal busses. Since shortest paths problems and their algorithms on graphs are well understood, we shall first proceed with verifying that automatons of this type do indeed yield useful results. Next, we cover some these shortest paths algorithms in detail. Finally, we end the section by discussing how these techniques might be tuned (or optimized) as situated in our CAD environment.

As mentioned above, this section relies on the following elementary observation which we will phrase in the form of a theorem. It is this very result, which should be obvious to any experienced Electrical Engineer, which motivated the previous manual computations. We formalize it here as it is the backbone of the remainder of the section.

Theorem 1 (Resistive Path Overestimation). *The actual resistance between two terminals is greater than or equal to the resistance of any single resistive path between them, without considering parallel components.*

Proof. Here, the proof we offer has more of an Electrical Engineering flavor (involving KVL/KCL and Ohm's Law). In the third (Network Flow) section of this paper we will offer another, more algebraic and graphically motivated, proof based on augmenting flow on parallel paths to yield a lesser resistance.

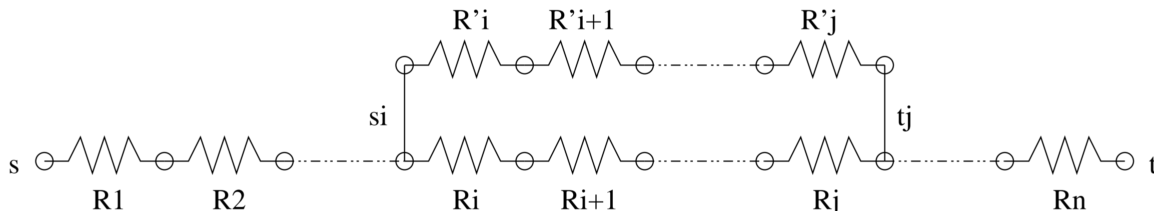
We proceed now with our first proof: Let P be a resistive path from some source s to any sink t. That is

$P =$

which has the following effective resistance, with respect to resistors in series [8] along P, (again, not considering any parallel components)

$$R(P) = R_{st} = \sum_{i=1}^{n} R_i. \tag{1}$$

Suppose, then, that there exists some parallel component to P, say P' such that we have the following picture (resistors along the parallel component are primed) $P||P' =$

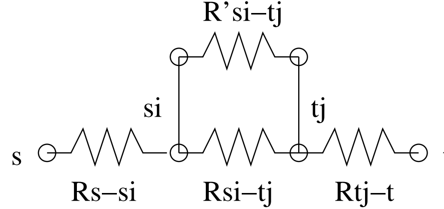


As above, we can break the effective resistance into four sets of resistors in series, two of which are parallel:

$$R'_{s_i t_j} = \textstyle\sum_{k=i}^{j} R'_k$$

$$R_{ss_i} = \sum_{k=1}^{i-1} R_k \ , \quad R_{s_i t_j} = \textstyle\sum_{k=i}^{j} R_k \ , \quad R_{t_j t} = \sum_{k=j+1}^{n} R_k$$

which results in the following picture.



Similarly, (1) can be broken apart and written as

$$R(P) = R_{st} = R_{ss_i} + R_{s_i t_j} + R_{t_j t}. \tag{2}$$

Now we have parallel resistors $R'_{s_i t_j}$ and $R_{s_i t_j}$ which can be combined into one resistor $R_{||}$ by the parallel resistor formula[5]

$$R_{||} = R'_{s_i t_j} || R_{s_i t_j} = \frac{R'_{s_i t_j} * R_{s_i t_j}}{R'_{s_i t_j} + R_{s_i t_j}} \leq R_{s_i t_j} \tag{3}$$

from which it follows that

$$R(P||P') = R_{ss_i} + R_{||} + R_{s_i t_j} \leq R_{ss_i} + R_{s_i t_j} + R_{s_i t_j} = R(P). \tag{4}$$

From this we have that the effective resistance from s to t along any path is greater than the resistance where we include one parallel component for current to flow along. Clearly, by extending this argument to include all parallel components we can conclude that if we allow parallel components from the entire electrical network to be included in our "path", then the actual resistance between s and t $A_{st} \leq R(P)$ for any resistive path P joining terminals s and t. □

Corollary 1 (Least Resistive Path Overestimation). *The total resistance of the least resistive path between two components is greater than or equal to the actual resistance between them.*

Proof. This follows immediately from Theorem 1. □

Therefore, since an electrical network can be represented graphically (as discussed in the previous section) we can use any of the well-known shortest path algorithms on directed graphs to get an overestimate on the total resistance between any two terminals. Such information is useful even with the existence of algorithms which allow us to compute the excitation levels exactly. The algorithms we discuss in this section are far less complex than the network flow algorithms for finding the exact excitation levels (which themselves are far more efficient and applying KVL/KCL to a large network), and they make it easy to identify trouble-spots in the ESD protection network (such as an open circuit between bondpad pairs). If nothing else, the shortest paths algorithms can help to identify "bad" terminal pairs for further analysis. See Example 3 in the final section of this paper.

Designers of ESD protection schemes use these results (Theorem 1 & Corollary 1) in order to estimate, by hand, the voltage drop between bondpad pairs. It is a simple matter to use Ohm's Law to translate resistance into voltage drop along a single path when assuming an incident current (say 1.3 amps). Using the result of Corollary 1 as our motivation, the rest of this section is dedicated to discussing algorithms for computing shortest paths on graphical representations of electrical networks. These algorithms will serve the task of automating the computation of a path-overestimate of the actual excitation level between terminal

[5] see [8] or (9) later in this section

pairs– essentially the process previously performed by hand. After discussing two fundamental shortest paths algorithms in depth, we conclude this section by discussing the accuracy of such overestimates, and some techniques for improving them. Such topics make a nice segue into our third topic of computing the voltage drop between terminal pairs exactly.

The basic idea behind each algorithm we will discuss is that given a source s, if a shortest path to some sink t passes through some intermediate node v, then the path P_v from s to v is itself a shortest path, and $P_{(v,t)}$ from v to t is must also be a shortest path (note that it is possible that $v = s$, and thus $P_{(v,t)} = (v, t)$). To be formal, we say that a path $P = \langle s = a_1 \dots a_k = t \rangle$ between s and t is a *shortest path* if and only if for any other path $P' = \langle s = a'_1 \dots a'_j = t \rangle$ that $\sum_{i=1}^{k-1} c(a_i, a_{i+1}) \leq \sum_{i=1}^{j-1} c(a'_i, a'_{i+1})$

3.2 Single Source Shortest Paths (Dijkstra)

The first algorithm which we will cover resembles the particular techniques which designers have used perform analyses by hand. This algorithm, attributed to Dutch mathematician E. Dijkstra , involves the creation of a shortest paths tree from the source s to all sinks t by adding arcs a one at a time from the arc set A, in a greedy fashion[6]. Formally, Dijkstra's SSSP (Single Source Shortest Paths) algorithm proceeds as follows:

First, we denote the *distance label* of a node v to be the overestimate to the shortest path cost from a source s to this node as follows: If $P_v = \langle s = v_1, v_2 \dots a_k = v \rangle$ then $d(v) = \sum_{i=1}^{k-1} c(v_i, v_{i+1})$. Our goal then will be to find the minimally optimal distance labels for all $v \in N$. Furthermore, we denote the *parent* of a node v to be the last node on the path P_v which is not itself the sink. Dijkstra's SSSP algorithm maintains two sets (with respect to d) which partition the node set N:

i. $S = \{\text{permanently labeled nodes}\}$

ii. $\overline{S} = N - S = \{\text{temporarily labeled nodes}\}$

We begin with s as the only node in the tree. The distance label $d(s) = 0$ is permanent, and $p(s) = nil$. For all $v \in N$, such that $v \neq s$, (temporarily) let $d(v) = \infty$, and grow the tree by finding "better" (shorter in our case) distances to the other nodes as follows.

That is, we start with $S = \{s\}$, and $\overline{S} = N - S$. From s (initially the only permanently labeled node) we consider all nodes v adjacent to s (all nodes v for which $(s, v) \in A$), and calculate

$$\begin{aligned} d'(v) &= d(s) + c(s, v) \\ &= 0 + c(s, v) \\ &= c(s, v). \end{aligned}$$

Then, if $d'(v) < d(v)$, we have found a shorter path to v, and so let $p(v) = s$ and $d(v) = d'(v)$. Otherwise, we do nothing, and move onto the next node adjacent to s until all have been exhausted. We then eliminate s from the current legal set of nodes and continue by letting s be the node v with the smallest $d(v) > d(s)$, and then repeat. That is, put v in S, removing it from $\overline{S}$, thus making its distance label permanent, and repeat as if v were the original source. The process stops only when there are no legal nodes left ($\overline{S} = \emptyset$).

The procedure necessarily terminates since a node is removed from $\overline{S}$ at each iteration. The resulting shortest paths tree can be recovered by using the recorded parent information. That is, the shortest path from the original source s to any node $v \in N$ is the shortest path from s to $p(v)$ with arc $(p(v), v)$. This suggests a recursive procedure for recovering the entire shortest path P_v. Thus, the resulting tree returned by Dijkstra's SSSP algorithm contains all the information necessary to build a shortest paths tree from the original source source s to all nodes $v \in N$. A shortest paths labeling and tree is given in Figure 7.

In the situation where there might be parallel arcs in our graph we will need to keep track of more than just parent node information in order to be able to reconstruct the shortest paths in terms of the arcs involved in the path. This can be done by keeping track of the *parent arc*. That is, line (11) can be followed by

[6] Dijkstra References: [4], [2], and [1].

```
(0)  let w(i,j) = c(i,j) for all (i,j) in A(G)

procedure DijksraSSSP(s,G=(N,A))
begin
(1)   PriorityQueue pq
(2)
(3)   d(v) = infinity, forall v in N
(4)   d(s) = 0
(5)   pq->d(v) = d(v), forall v in N
(6)   s = pq->min() // min over all temp distance labels
(7)   pq->remove(s) // thus making d(s) permanent
(8)   while s != nil do
(9)     foreach v adjacent to s do
(10)      if pq->d(v) < pq->d(s) + w(s,v) then
(11)        p(v) = s
(12)        d(v) = pq->d(s) + w(s,v)
(13)        pq->d(v) = d(v) // (decrease key)
(14)      endif
(15)    enddo
(16)    s = pq->min() // min over all temp distance labels
(17)    pq->remove(s) // thus making d(s) permanent
(18)  enddo
end
```

Figure 6: Pseudocode for Dijkstra Single Source Shortest Paths (SSSP) procedure, using a priority queue abstract data type. The procedure takes a source node $s \in N$ and a graph $G = (N, A)$ as input. The algorithm uses an auxiliary weight vector $w(i,j) = c(i,j)$ instead of the actual costs $c(i,j)$ themselves. This is will make it easier to use Dijkstra's SSSP algorithm on something other than the arc costs in the Network Flows section by simply redefining line (0).

```
(11.5)      parc(v) = (v,s)
```

where (v, s) was the particular arc chosen in line (9). This will come in especially handy in the Network Flows section when we wish to augment some amount of flow along the chosen arcs of a path from the source to the sink.

There is another option for keeping track of arcs in a shortest path which is slightly less desirable in general since it requires manipulating the graph representation. It involves marking the arcs of the graph whenever a node is removed from the priority queue (except for the source). This, of course, cannot be done without the parent arc pointers discussed above. Otherwise, without them we would not know which edge to mark. In this way we add a line (17.5) like

```
(17.5)    G->markEdge(parc(s))
```

and the result will be that the entire shortest paths tree will be marked after a call to *DijkstraSSSP*. On the other hand, the *parc* and p information is usually enough to recover the shortest path and tree information in general. Nonetheless, marking arcs is useful when a procedure requires that the shortest paths tree be traversed breadth first, rather than recursively. In addition, marking arcs can be useful for duplicating the part of the graph which is the shortest paths tree, etc. Of course, it is important to remember to remove any marks before a subsequent call to the marking routine.

A *Priority Queue*[7] data structure is used as an abstraction to many of the operations described above; to keep track of the ordering of nodes with respect to their temporary distance label, adjust that distance

[7] see [9], [6], and [1]

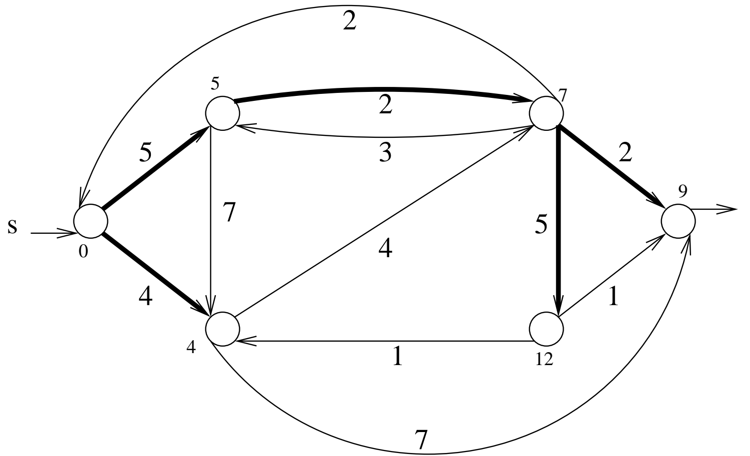


Figure 7: An example of a shortest paths tree on the graph given in Figure 1. The source is s and we can think of all other nodes as sinks, or specifically t. The optimal distance labels $d(\cdot)$ are given next to the nodes. The arcs in the shortest paths tree are darkened.

label, and thus adjust the priority of that corresponding node. In this way, the priority queue indirectly maintains the partition S and $\overline{S}$. Using a minimizing priority queue (the highest priority is the entry with the smallest value) we can write Dijkstra's SSSP algorithm using pseudocode in Figure 6[8].

Theorem 2 (Dijkstra SSSP Correctness). *Dijkstra's SSSP algorithm, given a source node $s \in N$, produces distance labels $d(v)$ which are the cost of the shortest path P_v from s to v for each $v \in N$.*

Proof. The proof of this is left to the Appendix and our references[9]. □

Data Structures and Runtime Analysis:

There are two key abstract data types (ADT's) used the Dijkstra SSSP algorithm. The algorithm relies on the ability to quickly find all of the nodes adjacent to a particular node v (line 9), and the efficiency of all of the priority queue operations, and assignments. Therefore, before discussing the running time of the algorithm, we must address these issues.

The first is quite straightforward. It is clear that for Dijkstra's SSSP algorithm we would like to be able to access all of the nodes adjacent to any node $v \in N$, which we will call $Adj(v)$. For such a purpose we have what is called an *adjacency list* data structure. This data structure depicts the graph $G = (N, A)$ as an array of lists. There is a list for each node $v \in N$, and the elements of the list are the nodes $w \in N$ such that the arc $(v, w) \in A$. Using this data structure, each execution in the *foreach* loop (line 9) is constant.

The second is less straightforward. The reason for this is that there are many different ways to implement a Priority Queue which yield the same worst case analysis with respect to Dijkstra's SSSP algorithm. However, there are some priority queue implementations which are well-suited to our procedure, and perform especially well in sparse networks, as often arise in practice, especially in our domain of I/O peripherals of Integrated Circuits. The two priority queue operations most relevant to Dijkstra's SSSP algorithm are *min* (line 7) and *decrease key* (line 13). Of less significance (but still worth noting) is the operation which manages the creation of the structure itself, and what exactly it means to delete an element from the priority queue. Here, we highlight three (of the many) priority queue data structures. The first we illustrate because its

[8]this code is intended to be a self-contained, easy to implement and read. It resembles some of the similar descriptions provided in (all of) our references. In particular, it most resembles a Java-pseudocode version presented in [2].

[9]see [1], [6], and [2]

implementation is straight-forward, the second because of its theoretical significance, and the third because of its efficiency in sparse networks. None of these will we discuss in great detail however, and, especially in the case of the latter two, we leave the specifics to our references.

As mentioned above, the priority queue operations we are interested in with respect to Dijkstra's SSSP algorithm are:

- *initialize:* create and initialize the structure to be empty.
- *insert:* insert a new element into the priority queue.
- *extract min:* return the entry in the priority queue with greatest priority (in our case this is least cost) and remove it from the priority queue.
- *decrease key:* change the priority of a particular entry to something less than its current value.

The data structure for the first type of priority queue is simply a node-name ordered array pq[]. The structure assumes that the nodes are named $1 \ldots n$, where for each $v_i \in N$ we have pq[i]=$d(v_i)$. This makes it trivial ($O(1)$) to find or change the priority (or cost) of any node. Removing any node from the priority queue can also be done trivially if we add a boolean field to each priority queue entry indicating whether or not to include this node in *min* operations. Creating a new priority queue is $O(n)$ and inserting an element is $O(1)$. The initialization is a trivial $\Theta(n)$ operation to create and initialize an array structure of suitable size, *decrease key* is $\Theta(1)$, and *min* is an $O(n)$ for-loop which must potentially examine all nodes in the array. This data structure is the easiest of the three to implement. As we shall soon see, we might be able to improve upon the complexity of the *min* operation, by using a data structure called a *Fibonacci Heap* as our priority queue. However, we will do no better in the worst case.

	Trivial Array	**Fibonacci Heap**	**Binary Heap**
initialize	$O(n)$	$O(1)$	$O(n)$
insert	$O(1)$	$O(1)$	$O(\log n)$
extract min	$O(n)$	$O(\log n)$	$O(\log n)$
decrease key	$O(1)$	$O(1)$	$O(\log n)$
Dijkstra SSSP	$O(n^2)$	$O(m + n \log n)$	$O(n \log n)$

Figure 8: Complexity Table for three Priority Queue data structures often used in Dijkstra's SSSP algorithm. As usual, $n = |N|$ and $m = |A|$.

The *Fibonacci Heap* is a very powerful data structure which is efficient because it is lazy. This priority queue is desirable when the number of *extract min* operations is small relative to the total number of priority queue operations. It maintains several linked tree structures each of which represent a partial ordering on the elements contained in the priority queue, further maintaining that the element with the greatest priority is at the root of one of these trees, which it keeps a special pointer to. We will not continue with a full description of this structure and its analysis, since such is long and of only theoretical interest, and is well documented[10]. However, we hi-light the key results in the table in Figure 8 which pertain Dijkstra's SSSP algorithm. A technique known as *amortized analysis*[11] is used to measure the complexity of Fibonacci Heap operations. The basic idea comes from the accounting method of a similar name. Since the Fibonacci Heap is lazy, many of its operations are trivial ($O(1)$), but to account for this lazy nature the data structure must execute a more expensive operation every once in a while in order to "clean up" the data structure. That is, every operation except for *extract min* has a constant amortized cost. Intuitively, we can think of the other operations as paying some amount in advance for the mess they are making by being lazy. While the Fibonacci Heap priority queue is theoretically more desirable than the Trivial Array we must still

[10] see [6], Chapter 21

[11] see [9], Chapter 3.5 and [6] Chapter 18

keep in mind their "implementation complexity". Since Fibonacci Heaps are far more complicated, from an implementation standpoint, we can think of the Θ-expressions for the running times of its operations as having large constant terms. It is because of this complexity that in practice Fibonacci Heaps only outperform other implementations– our Trivial Array in particular– for large entry (node) sets. Further analysis of Dijkstra's SSSP algorithm with respect to an arbitrary choice of Priority Queues is left to the Appendix.

Using a Fibonacci Heap implementation of a priority queue we cannot expect to do better than the Trivial Array in the worst case. It is easy to conclude, from the table in Figure 8, that if we were to use a *Fibonacci Heap*, Dijkstra's SSSP algorithm will run in time $O(m + n \log n)$. For dense networks ($m = n(n-1)/2$), this is no better than our Trivial Array implementation. For sparse networks this is an improvement on the worst case. Again, we must also take into consideration the implementation complexity of the Fibonacci Heap versus the Trivial Array. The former, in this case, is far more complex than the latter, which means that we would certainly do better using a Trivial Array on smaller graphs. On the other hand, one might even be able to assume that the graph which represents an ESD protection scheme is extremely sparse by appealing to the geography of I/O rings, thus providing a strong argument in favor of using an approach as suggested by Fibonacci Heaps.

There are definitely other ways of maintaining a priority queue, some which are more appropriate than others. For example, *Dial's*[12], *Johnson's*[13], and the *Radix Heap*[14] Implementations for most cases outperform all of the other methods in practice, and especially in the case of Dial's Implementation are easy to implement. However, each of these approaches have the disadvantage that they require the costs of arcs be integral, whereas extracted resistances are typically rational with large decimal expansions. At first this seems like no big deal because we can always scale rationals by an arbitrarily large powers of 10 to get integers representing the same measurement, but then the time complexity of these priority queue operations are functions of cost of the most expensive arc. This is not likely not to be polynomial in the size of the graph. We will run into this problem again when we talk about Network Flows.

Still, there are other examples of Priority Queues, such as the standard[15] *Binary Heap* Implementation, which are indeed well-suited to our model. The time complexity for this implementation (with respect to Dijkstra's Algorithm) are included in the table of Figure 8 for this very reason. Note that for sparse graphs this implementation is preferred to the Trivial Array, but for dense graphs it has a theoretically worse bound. Fibonacci Heaps have better bounds than Binary Heaps for all occasions.

In conclusion, Dijkstra's SSSP algorithm makes for a nice manual, or interactive tool for use in any environment. With appropriate input and output formats it is convenient to make requests about arbitrary pairs of terminals in the input graph, even sets of pairs of terminals, with a quick turnaround. If our graph is backward compatible with the netlist, and the corresponding extracted layout representation of the ESD protection device, then we might be able to display the shortest paths in the layout itself by highlighting the components that were chosen by the algorithm. The only downside to this is that unless we have a clever way of recording the results of previous calls to the Dijkstra SSSP subroutine, we may be forced to unnecessarily recalculate a shortest paths tree, for say two non-successive shortest paths requests which have the same source. When a request calls for a set of pre-designated pairs of shortest paths then we can serialize the process by ensuring that we invoke the sub-routine for each source only once. However, when shortest paths are required for all pairs of terminals in a device, or should we find it beneficial to record, and reference a pre-computed table of shortest paths, there is a more clever and faster way of accomplishing this.

3.3 All Pairs Shortest Paths (Floyd-Warshall)

If we are interested in voltage drop information for all pairs of components in an ESD Protection Scheme we can run Dijkstra's SSSP algorithm for every source node in the graph. Doing this would require a mechanism for keeping track of the distance labels of sink nodes corresponding to each source node, requiring $\Theta(n^2)$

[12] see page 113 of [1]
[13] see [7]
[14] see page 116 of [1]
[15] can be found in any algorithms text such as [2], [9], and [6]

space. Calling Dijkstra's SSSP algorithm for each source would then be in time $\Theta(n^3)$, and we can do no better for dense graphs. However, in this section we introduce a simple dynamic programming[16] algorithm for the *All Pairs Shortest Paths* problem called *Floyd-Warshall APSP*[17]. Although the Floyd Warshall APSP algorithm is theoretically as complex as Dijkstra's SSSP algorithm run for each source node, the simplicity of the Floyd-Warshall APSP algorithm makes using it over the later more desirable when an analysis on the entire network is desired. As we will soon see, since the heart Floyd-Warshall APSP algorithm can be written with just three "tight" loops (no complicated priority queue operations) the constant terms hidden in the Θ expressions for its runtime are small compared to those for the Dijkstra SSSP algorithm. Furthermore, keeping track of permanent distance labels for all pairs of source and sink nodes is implicit, and only minor modifications are needed in order to keep track of the actual shortest paths between all pairs of terminals.

There are two key ingredients that go into designing a dynamic programming problem: (1) a recurrence that characterizes optimal solutions, which in our case are shortest paths, and (2) the ability to represent the input and optimal solutions to subproblems so that they can be easily referenced (e.g. as in a table). We address the second ingredient first as it allows us to bring up another common way of representing graphs, and after which it will be clear how the first ingredient can take advantage of such a representation.

The input graph $G = (N, A)$, where $N = \{v_1, v_2, \dots, v_n\}$, to the Floyd-Warshall APSP algorithm is an *Adjacency Matrix*. This matrix is actually an $n \times n$ two-dimensional array A (or table, where $n = |N|$). The entries in the table are filled in in the following way:

$$A[i][j] = \begin{cases} \infty & \text{if } (v_i, v_j) \notin A(G) \\ c(v_i, v_j) & \text{the cost of } (v_i, v_j) \in A(G) \end{cases} \tag{5}$$

Therefore an arc $(v_i, v_j) \in A$ is represented by its cost $c(v_i, v_j)$, and the entries of the table $A[i][j]$ are ∞ if there is no such arc in A. There is a slight limitation in representing the graph of an electrical network in this fashion since there may be multiple arcs between any pair of nodes. As we will see in the next part of this section, with a little preprocessing this is no limitation after all.

However, for the Floyd-Warshall APSP algorithm it is useful to think of the entries in the table A in a slightly different way. Given the Adjacency Matrix A for a graph $G = (N, A)$ we can think of the entries $A[i][j]$ as the cost of the shortest path from v_i to v_j which uses only one arc. Or, alternatively, $A[i][j]$ is the cost of the shortest path from v_i to v_j which uses no intermediate nodes. With this in mind we develop some new notation. Let $D_k[i][j]$ be the cost of the shortest path from v_i to v_j which can use any of $\{v_1, \dots, v_k\} \subseteq N$ as intermediate nodes. Therefore, we clearly have $D_0[i][j] = A[i][j]$. As before, with the Dijkstra SSSP algorithm, we can think of D_k as a table of temporary distance labels, and the entries of D_n as permanent distance labels.

With this new notation we can now present the recurrence, or principle of optimality, underling the Floyd-Warshall APSP algorithm: Given a graph $G = (N, A)$ which can be represented by an Adjacency Matrix A, the shortest path $D_k[i][j]$ from v_i to v_j which uses only elements of $\{v_1, \dots, v_k\} \subseteq N$ as intermediate nodes is given by:

$$D_k[i][j] = \begin{cases} A[i][j] & \text{if } k = 0 \\ \min\{D_{k-1}[i][j], D_{k-1}[i][k] + D_{k-1}[k][j]\} & \text{otherwise} \end{cases} \tag{6}$$

This recurrence is valid for the following reason. It is easy to see that the shortest path, which might include the first k nodes, is either the same as the shortest path which includes the first $k-1$ nodes (that is, the shortest path does not include the kth node) in which case we have $D_k[i][j] = D_{k-1}[i][j]$. Otherwise the kth node is included, and then we have $D_k[i][j] = D_{k-1}[i][k] + D_{k-1}[k][j]$. This result leads to a very easy algorithm to implement. The pseudocode for the complete procedure can be found in Figure 9. The result is a table $D_n[i][j]$ of permanent distance labels which represents the cost of the shortest paths from v_i to v_j for $v_i, v_j \in N$.

[16] a template for designing algorithms where the principle of optimality for a problem can be written as a recurrence, and the problem can be solved quickly by *memoizing* the optimal solutions of subproblems and using them to find the optimal solutions to larger subproblems using the recurrence. The most benefit is gained when many larger subproblems share the same smaller subproblems.

[17] attributed to both Floyd for APSP and Warshall for *Transitive Closure* who separately developed similar algorithms for solving these slightly different problems at about the same time.

```
procedure FloydWarshallAPSP(A[1..n][1..n])
begin
(1)    float D[1..n][1..n][1..n]
(2)    D[0] = A
(3)
(4)    for k=1 to n do
(5)      for i=1 to n do
(6)        for j=1 to n do
(7)          if D[k-1][i][k]+D[k-1][k][j] < D[k-1][i][j] then
(8)            D[k][i][j] = D[k-1][i][k]+D[k-1][k][j]
(9)          else
(10)           D[k][i][j] = D[k-1][i][j]
(11)         endif
(12)       enddo
(13)     enddo
(14)   enddo
end
```

Figure 9: Pseudocode for the Floyd-Warshall All Pairs Shortest Paths (APSP) procedure. The procedure takes an *Adjacency Matrix* representation of a graph $G = (N, A)$ as input.

The Floyd-Warshall APSP algorithm clearly runs in time $\Theta(n^3)$, and space $\Theta(n^3)$. However, since we only really need D_{k-1} in order to compute D_k, if we are clever we can use only two $n \times n$ tables, and thereby save a great deal of space. In fact, we can get away with having just one $n \times n$ sized table D by noticing that the table D_k is filled in top down. In either case we use only $\Theta(n^2)$ space.

Next, since the Floyd-Warshall APSP algorithm in Figure 9 only keeps track of distance labels, there is still the issue of recovering the actual shortest path information. However, this too is easy, since we can do pretty much the same thing we did for Dijkstra's SSSP algorithm. This means keeping $n \times n$ table of parents P_k, where $P_0[i][j] = i$, and we add lines to the pseudocode in Figure 9, described in Figure 10.

```
(8.5)          P[k][i][j] = P[k-1][k][j]
(10.5)         P[k][i][j] = P[k-1][i][j]
```

Figure 10: Additions to the Floyd-Warshall SSSP algorithm to keep track of path information by recording the parent of a sink in a table P_k.

P_k like D_k, as in Figure 10, requires $\Theta(n^2)$ space for each of the n tables corresponding to the n tables of distance labels, $\Theta(n^3)$ total. However, the same observation can be made for P_k as for D, and we can get away with using only one table P, thereby using space in $\Theta(n^2)$. Therefore, since we have added only a constant amount of work to the entire procedure, including the code for path information, this does not change our asymptotic measure of space or time. We formalize these results in the form of a theorem.

Theorem 3 (Floyd-Warshall Complexity). *Given an* Adjacency Matrix *representation of a graph* $G = (N, A)$ *the Floyd-Warshall APSP algorithm runs in time* $\Theta(n^3)$ *and uses space in* $\Theta(n^2)$*, where* $n = |N|$.

In summary, while Dijkstra's SSSP algorithm makes for a nice interactive Shortest Paths analysis tool, we consider the Floyd-Warshall algorithm as being the most efficient procedure for characterizing an entire network– producing a complete *Shortest Paths Certificate* of sorts for the entire network. This is because of the compactness of its loops, and because it does not require any complicated abstract data types, like priority queues. For this reason the constant in the Θ expression for the Floyd-Warshall APSP algorithm is likely to be much smaller than that for the Dijkstra APSP modification. In addition, the Floyd-Warshall

APSP algorithm is far easier to implement, especially in a low-level programming language. Moreover, compilers are good at optimizing loops, but do not know much about priority queues. Lastly, the two tables produced by the Floyd-Warshall APSP algorithm are computationally and implementationally easy to store in memory or on disk for later reference– at least much more so than trying a scheme which provides similar results using Dijkstra's SSSP algorithm– or as a "snapshot" in the development process of an I/O protection scheme, thus making the procedure a nice "batch process" on large networks. Of course, if all we require is the budget information between bondpad pairs, which make up only a small fraction of the total number of possible terminal pairs in an I/O portion of an integrated circuit, we may be doing much more work than required. In this case the Floyd-Warshall APSP algorithm still runs in $\Theta(n^3)$, whereas running the Dijkstra SSSP routine with each of k bondpads as a source will run in $O(kn^2)$, which is likely to be faster when $k \ll n$.

3.4 Graph-size Reductions, Optimizations & Improving Overestimates

As mentioned in the previous section, the *Adjacency Matrix* data structure for representing a graph $G = (N, A)$ cannot represent parallel arcs in A. This is simply a result of the fact that there is only one location $A[i][j]$ in the table, which means that there is room for only one arc like (v_i, v_j). Since it is possible, and even likely, that the electrical networks we are dealing with have parallel connections (parallel diodes, resistors, etc), something must certainly be done in order to accommodate this. The first solution to come to mind is probably a non-trivial extension of the the adjacency matrix format, perhaps in terms of creating a linked list at each matrix entry location. However, we will be much better off in the long run if we can find a way of getting rid of parallel arcs in the graph G all together. That is, without destroying any of the underlying electrical network which the graph G represents. As we will soon see, this limitation of the adjacency matrix data structure is a blessing in disguise. Not only will it be advantageous to eliminate parallel arcs for adjacency matrix representations, but also for the adjacency lists representation as well. While Dijkstra's SSSP algorithm can handle parallel arcs, a smaller network is certainly more desirable.

The preprocessing step which we will soon propose for eliminating parallel arcs relies on an important property of electrical networks. Furthermore, it suggests not only an algorithm for eliminating parallel arcs, which eliminates our representation problem for the adjacency matrix, but it also reduces the arc set A, thereby yielding faster shortest paths algorithms. Most importantly however, performing the following reduction guarantees that our shortest paths algorithm will produce a more accurate resulting ESD Budget overestimate.

We generalize the following basic property of parallel resistors in an electrical network: When there exist any number of resistors $R_1, R_2, \ldots, R_n$ in parallel, that is, each R_i has the same terminal pair as others (call them A, and B), the effective resistance from A to B, R_{eff} is less that that of any one of the resistors R_i. In the simplest case of a pair of parallel resistors R_1, R_2, where $R_1 = R_2 = R$, we can expect to have the effective resistance $R_{\text{eff}} = R/2$ (much like two downhill river-beds which begin and end at the top and bottom of the hill respectively, and have the same slope– the total amount of water allowed to flow downhill is twice that of any single river-bed) between A and B. The general case of n resistors in parallel is easiest to work out if we think in terms of the conductance $\frac{1}{R} = \frac{I}{V}$. Since the current from A to B is just the sum of the currents on each resistor I_i, assuming one unified voltage, we have that the total conductance from A to B is the sum of the conductances:

$$\frac{1}{R_1} + \frac{1}{R_2} + \cdots + \frac{1}{R_n} = \frac{1}{R_{\text{eff}}} \tag{7}$$

We introduce the notation $||$ to mean parallel, and write:

$$R_1 || R_2 || \cdots || R_n = R_{\text{eff}} \tag{8}$$

Again, going back to the simple case of two resistors we have:

$$R_1 || R_2 = \frac{R_1 * R_2}{R_1 + R_2} \tag{9}$$

And we now have a way of getting rid of parallel arcs which represent parallel resistors without violating any electrical properties, and thus leaving the underlying network unchanged. It should be obvious that all other connection types (diode, hierarchical) can be handled in the same way, since in the first chapter we were able to argue that each can be viewed as a resistor of sorts. However, it might be advantageous to execute this preprocessing as a two part process. First, eliminate parallel resistors in the most raw (or low) level netlist representation of the electrical network. Second, after translating the netlist, thereby encoding hierarchy, etc, eliminate any parallel components which are left by the encoding, if any (this depends on encoder, perhaps a good encoder would not leave any).

Having eliminated all parallel arcs in a graph $G = (N, A)$ (producing $G = (N, A')$, $A' \subseteq A$) we have not only solved our adjacency matrix representation problem, and reduced the size of the arc set A, but even more significantly we have eliminated some parallel paths. All in all, this is a quite desirable side effect, since our Shortest Paths algorithms, which before did not take any parallel paths into account, when run on the preprocessed graph G' now inherently consider paths with "degree 1" parallelism. That is, if it chooses an arc $a \in A'$ to be part of a shortest path, it has effectively chosen to augment current on perhaps an entire network of parallel arcs in A (which a replaced in A'), thus yielding a tighter overestimate for the actual voltage drop of the component (or components) whose shortest path contain the arc a. This is motivation in and of itself for the development of such a preprocessing scheme as was previously motivated mostly by convenience.

Unfortunately, there is at least one problem with eliminating arcs (or components) from the graph $G = (N, A)$ which represents an electrical network. Since we have already shown that after preprocessing the electrical network we produce is effectively equivalent, there is no problem here. The problem arises in the CAD environment. While the underlying network remains unchanged, there is an inherent loss of information. When the results of a Shortest Paths analysis are completed and fed back to the calling (CAD) environment, it may be that there are parallel arcs used in some shortest path, but only one of them is accounted for by G, or any of the shortest paths trees from G. This means that if we want an accurate "backward comparability" after preprocessing, we need to somehow keep track of eliminated components, and the remaining arcs of A' which represent them. This can easily be done by generating a list of equivalences for the arcs of A' for translation of the output of the shortest paths algorithm back to the CAD environment.

While on the topic, we might wish to consider another preprocessing step which consolidates elements in series. However, there is no accuracy advantage to this, and this has an even greater negative impact on backward compatibility (from G back to the CAD netlist and graphical representation of the underlying electrical network). Furthermore, this is only possible if the nodes of components in series have absolutely no arcs leaving or entering them not in the series (or, for some reason we can be guaranteed that they will never be on a shortest path). Or, perhaps we might consider eliminating all "degree 2" parallel arcs. However, pretty soon, the complexity of the preprocessing will exceed that of the actual shortest paths procedure. In the next section we present a far more powerful technique for estimating the actual voltage drop more accurately.

There are a number of other things that can be done in order to reduce the size of the input netlist which apply in general. The first of these simply involves the symbolic removal of components from the analysis. That is, if it is known in advance that some components in the layout (or, in particular, large sets of components representing un-affected portions of the device) will not be used in a shortest path, then they can be removed, thereby reducing the size of the input graph. This can be handled by the preprocessing compiler discussed in the previous section. This may also be useful for finding "alternate shortest paths". That is, if it is known in advance (say by previous simulations) that certain components play a pivotal role in many shortest paths, it might be interesting to examine a simulation where that component is off limits. Of course we would like to do this without actually modifying the layout itself. Therefore, we should like the ability to flag components to be removed by the preprocessor. For example, we may want to go so far as to remove all internal nodes along this shortest path, to find the "degree 2" shortest path, and then combine the results as discussed previously. This can be repeated until no such path exists, and we have yet another way of further parallelizing the shortest path result. Of course, neither of these strategies guarantees more accurate budget information, but prove to be useful ways of speeding up the analysis, and allow the user to get more meaningful results quickly, as well as making the tool more flexible.

4 Convex Flows

4.1 Calling All Paths

Our previous discussion was motivated by a simple observation at the network level. Namely, by ignoring parallel paths/components, and focusing on just a single resistive path from the source to the sink, we could get a great deal of useful information about the ESD budget rather quickly. This part of our discussion focuses on computing the resistance (thereby essentially computing the excitation level) between terminal pairs exactly. This means that we must take into consideration all parallel paths and components. Or, to be more exact, we must consider the entire network with the possibility of current flow on every arc if we are to attempt to compute the effective resistance between terminals exactly.

The end result of our discussion is that there is a class of *Network Flow* algorithms called *Convex Minimum Cost Maximum Flow* (or simply Convex Flow) which suits our problem nicely. However, unlike in the previous section, rather than having the some proven property of electrical networks motivate our discussion, we proceed first with a general discussion of Flows, Networks and Minimum Cost Maximum Flow algorithms, then conclude the section with results motivated by the algorithms themselves. Once we show that this result does indeed accurately report the excitation level between terminal pairs, we can then offer a more algorithmic explanation of our first theorem for overestimating in the previous section. After the motivation and basic algorithms are laid out, we will discuss ways of optimizing such procedures by taking advantage of our domain, and a clever scaling technique. Unfortunately, a general discussion of Minimum Cost Maximum Flows is quite disjoint from the topic of electrical networks, properties, or current flows. For this reason we take a brief departure from the electrical domain. We return to the topic of computing the excitation level between terminal pairs exactly when we begin discussing Convex Flows.

4.2 Networks, Minimum Cost Maximum Flow & Residue Graphs

First off, some definitions and the description of a helpful augmentation of our graphical representation are in order. Minimum Cost Maximum Flow (MCMF) is an optimization problem whereby we think of the arcs of the graph as conduits on which some amount of commodity can flow while accruing travel-cost between locations which are designated as nodes. We think of certain nodes as suppliers of this commodity (sources) and others as having some demand (sinks) for the commodity. Still, there are some nodes/locations which neither supply nor demand anything, but still may be used as intermediates in some flow from a source to a sink node. This problem would be less interesting without the idea of an arc in the network having a maximal capacity. Without such a constraint the problem essentially reduces to being a Shortest Paths problem. This is especially true in the case where there is only one source and an arbitrary number of sinks. As we saw in the previous section, Dijkstra's SSSP algorithm solves this problem quite nicely. For this reason we introduce the notion of a capacity constraint on an arc. When we come back to talking about electrical networks and Convex Flows we will argue that no such constraint is necessary, or even useful. However, we will be assuming a much more complicated cost model where it will not be advantageous to send all of the flow along a shortest path. Note that this capacity constraint is in addition to the cost $c(i, j)$ (or c_{ij}) already associated with an arc.

Formally, we say that an arc $(i, j) \in A$ has capacity $u(i, j)$ (or u_{ij}). This means that we are not allowed transport more than u_{ij} units of flow along $(i, j) \in A$. Similarly, we designate x_{ij} to be the current (nonnegative) amount of commodity flowing (or simply, flow) on arc (i, j). This means that at all times during the running of any algorithm we must have the following invariant[18]:

$$0 \leq x_{ij} \leq u_{ij}. \tag{10}$$

[18]we might also like to include lower bounds l_{ij} to model things like transistors. This will be discussed in the Further Considerations section at the end of this paper

Although this is not necessarily true of all MCMF algorithms[19], the algorithms we discuss always maintain the above invariant.

With the concept of flow along an arc emerges a slightly different idea of a cost across that arc. We think of c_{ij} (constant) as being the cost of shipping one unit of commodity, or flow, from node i to node j. Therefore, we have that the actual cost of shipping along an arc $(i, j) \in A$ varies linearly[20] with the amount of flow on that arc. We could define

$$C(x_{ij}) = (x_{ij} + 1) * c_{ij} \tag{11}$$

to be our new arc-cost to send an additional unit of flow along (i, j). This is useful as an illustration of what we mean by linearization of the cost of an arc. However, this will not get much use in our further discussion. A generalization of (11) will be useful for the Convex Flow problem. Furthermore, we define $b : N \to R$ be the supply/demand vector, or function. In an instance of the MCMF problem each node $i \in V$ is either a supply node, corresponding to $b(i) > 0$, a demand node when $b(i) < 0$, or it is an intermediate node, whereby $b(i) = 0$. The vector b is referred to as the *mass balance constraint* for the MCMF problem. Note that in order for the problem to make sense we must have

$$\sum_{\forall i \in N} b(i) = 0 \tag{12}$$

which simply states that the total supply must equal the total demand. We call an input graph $G = (N, A)$ with these extra specifications a *network*.

Having developed some new notation it is now possible to formally describe Minimum Cost Maximum Flow as an optimization problem. These formalizations are an intermediate step toward formulating the actual optimization problem for our electrical network simulations. The program we are really interested in will vary slightly from the one below in that the objective function (13) will be convex. This will be motivated by a quadratic dissipation of power across resistors, where in general we will be interested in finding the equilibrium current across each resistor in the network. The optimality results and implementation of algorithms for each model are similar. However, assuming a linear cost model will simplify the following discussion, which will easily be generalized in the next section. For now, we will be interested in the following:

$$\text{minimize} \quad \sum_{j=1}^{n} c_{ij} x_{ij} \tag{13}$$

$$\text{subject to} \quad \sum_{j:(i,j)\in A}^{n} x_{ij} - \sum_{j:(i,j)\in A}^{n} x_{ji} = b(i) \qquad \forall i \in N \tag{14}$$

$$0 \le x_{ij} \le u_{ij} \qquad \forall (i, j) \in A. \tag{15}$$

Notice that (14) corresponds to enforcing that the inflow of any node i is equal to its outflow. Also, for supply nodes the outflow is positive, and for demand nodes the outflow is negative (corresponding to positive inflow), and for all other nodes the inflow/outflow must be is zero. Typically, the flow on the entire network, or equivalently the arc set A, is referred to as x (using vector notation) rather than x_{ij} $\forall (i, j) \in A$. A flow x which satisfies both the arc capacity constraints u (15) defined similarly over A, and the mass balance constrains (14), is said to be *feasible*. When referring to an optimal flow (one which minimizes (13)), we usually write x^* to notate that x is optimal, rather than just x.

Such a formulation as shown above is common for optimization problems, especially those of a Linear Programming flavor. Although we do not use Linear Programming directly to solve MCMF problems, the algorithms we discuss do take advantage of some nice results from Linear Programming such as duality, reduced cost, complimentary slackness, etc. Nonetheless, our discussion of such properties, especially reduced cost, is completely disjoint from a Linear Programming approach. It is noteworthy to mention that

[19] Pre-Flow Push, etc, which will be discussed briefly after we enumerate the Successive Shortest Paths Algorithm in this section

[20] Again, as we will see when we discuss Convex flows, we have useful ways of interpreting the cost of an arc in the network, which help us more accurately simulate the underlying graph as an electrical network.

combinatorial optimization problems formulated as in (13-15) are often easily adapted to be solved using the simplex method employed by many commercial Linear Programming solvers[21]. However, we do not discuss adaptations of MCMF problems to the simplex method in any further detail as it will be difficult to port the end result (Convex Flow algorithms) to this domain. Furthermore, Network Flow algorithms are somewhat of a special case of the general Linear Programming class of optimization problems, and one can typically do better than the simplex method (and even better than the specialized Network Simplex Method) by employing a more direct approach.

We will be interested in exploiting algorithms which work on Graph Abstract Data Types. Therefore, for the time being we must pull away from the Linear Programming MCMF problem, and pay closer attention to the problem presented from a graph implementation standpoint. The data structure that will be most useful is essentially the same Adjacency List structure used by Dijkstra's SSSP algorithm. However, it will be useful to keep track of more information (flow, capacity, etc) about each arc in the graph. Actually, Dijkstra's SSSP algorithm will be a subroutine to the main MCMF algorithm we will present shortly. The most important deviation from the representation that was discussed in the previous two sections is the concept of a *residue graph*. The algorithms which we present for the MCMF optimization problem converge to a solution by making what seems to be the best move available at the time. Such a "move" is usually along the lines of augmenting some amount of flow along a subgraph (path, cycle, or subnetwork) of $G = (N, A)$, usually to satisfy the supply and demand of a source and sink pair. However, we should like to keep the option open of sending flow along the opposite direction of an arc to reduce the cost of a flow, or to satisfy some other demand. A *residue graph* $G(x)$ is essentially the original graph G, together with extra arcs which facilitate the ability to remove flow, or send flow backwards on an arc. That is, if any directed arc $(i, j) \in A$ has been assigned flow x_{ij} we have that in the *residue graph* $G(x)$ there is a directed arc (j, i) with capacity x_{ij} ($u_{ji} = x_{ij}$) and cost $-c_{ij}$ ($c_{ji} = -c_{ij}$). In some sense, $G(x)$ is a function of G and x, which is the reason it is written in this fashion. See Figure 11 for an example network, and Figure 12 for an example flow.

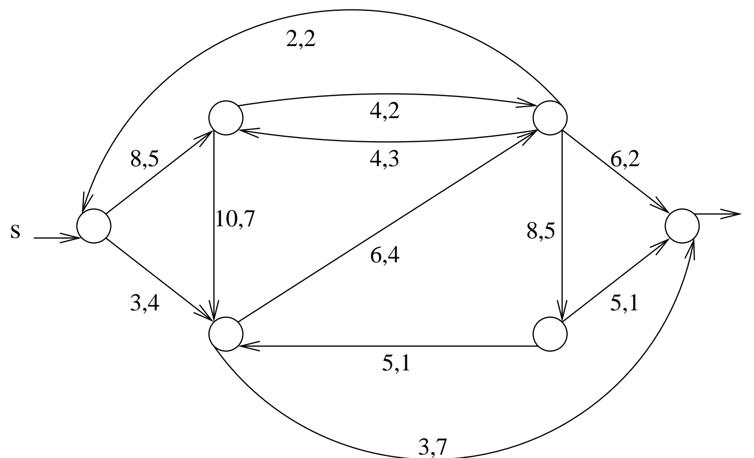


Figure 11: A simple example of a network. Each arc is associated a capacity, and a cost (written u, c).

More formally, the *residual network* $G(x)$ of an input network G is defined in the following way. For each directed arc (i, j) in G we have the arc (i, j) with capacity u_{ij} and cost c_{ij} together with a new arc (j, i) with capacity x_{ij} and cost $-c_{ij}$, both in $G(x)$. If ever an arc in $(i, j) \in A(G(x))$ has zero (non-positive) remaining capacity ($c_{ij} - x_{ij}$) this arc is not considered to (currently) be part of the residual network. Usually this entails a symbolic removal of the arc from an implementation standpoint, since we are likely to need it later. This allows us to think of $G(x)$ starting out as G, since before any flow is augmented, those arcs not originally in G should have zero capacity. Similarly, $G(x)$ should finish again as G. That is, in the end there

[21] Lindo, Mathematica / Maple, etc; see also [4] and [10], and other LP references.

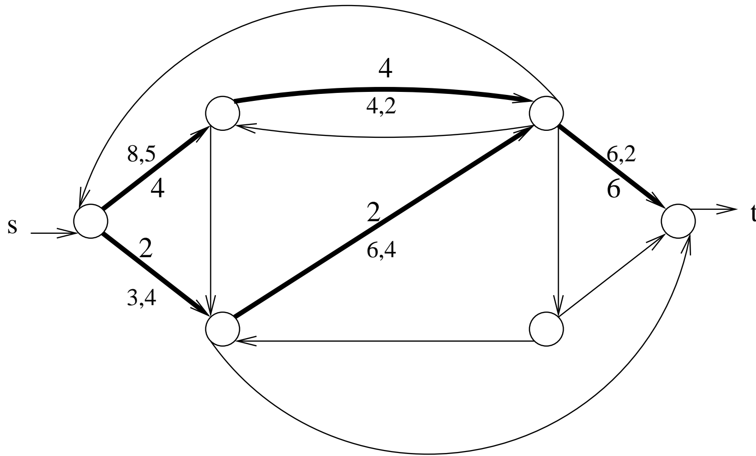


Figure 12: An example of a valid flow for the network given in Figure 11. Note that the mass-balance and capacity constraints are satisfied when s is taken to be the source, and t the sink. The value of the flow is 6 (s sources 6 and t sinks 6). As in Figure 12 the pair u, c represent the cost and capacity of each arc. The third value represents the flow along that arc. Arcs without flow are left unlabeled.

should be no flow on any of the arcs not originally in G.

We refer to arcs in $G(x)$ which were actually present in the original graph G as *forward* arcs. Arcs of $G(x)$ which were not originally in G, but are byproducts of the residue of x, will be referred to as *reverse* arcs (or sometimes *residue* arcs). Reverse arcs exist solely so that we can augment flow backwards on an existing forward arc, thereby actually reducing the flow on the corresponding forward arc. In other words, it does not make sense for flow to ever be augmented on a reverse arc, since such an operation really corresponds to a decrease in flow on the corresponding forward arc. This supports the decision for the capacity of a reverse arc equaling the flow on the corresponding forward arc. See Figure 13 for general illustration of the construction of a residual network.

For bookkeeping purposes, we introduce the notion of an *antagonist* (or anti-parallel) arc. An *antagonist* of a forward arc is a reference to its corresponding reverse arc. Similarly, the *antagonist* of a reverse arc is a reference to the corresponding forward arc. This serves two purposes. The first purpose is that it allows our discussion of arcs in $G(x)$ to be more general. Rather than explicitly indicate whether an arc is a forward or reverse arc when referring to its anti-parallel mate, we can reference its antagonist. In general, we shall refer to the antagonist of an arc $(i, j) \in A(G(x))$ as $\overline{(i, j)}$ regardless of whether (i, j) is a forward or reverse arc. The second reason for introducing the notion of an antagonist is implementation driven. Each arc in the graph implementation of $G(x)$ will have an *antagonist pointer*. This pointer will allow an arc easy access to its corresponding anti-parallel mate. This way, whenever flow is augmented along any arc, the corresponding adjustments to the anti-parallel arc can be made in constant time (say by a similar routine with the antagonist as an argument). Using an adjacency list data structure, these pointers represent the bookkeeping necessary to represent undirected arcs in a directed network, as well as paired parallel arcs. Without antagonist pointers it would be difficult to keep track of which arc in all of $A(G(x))$ is the corresponding antagonist for any given arc $(i, j) \in A(G)$. In general there may be several directed parallel or anti-parallel arcs between nodes i and j.

Furthermore, it will be useful to assume that the network represented by $G(x)$ contains an uncapacitated directed path between each pair of nodes in N. This assumption imposes no theoretical or implementational restriction on the input graphs since we can always add arcs of infinite capacity, with infinite cost, into the network between all possible sources and sinks. Clearly, such an arc will not be in any minimum cost solution to the MCMF problem unless it has no feasible solution.

In light of the previous comments, $G(x)$ will typically serve as a general representation of the input

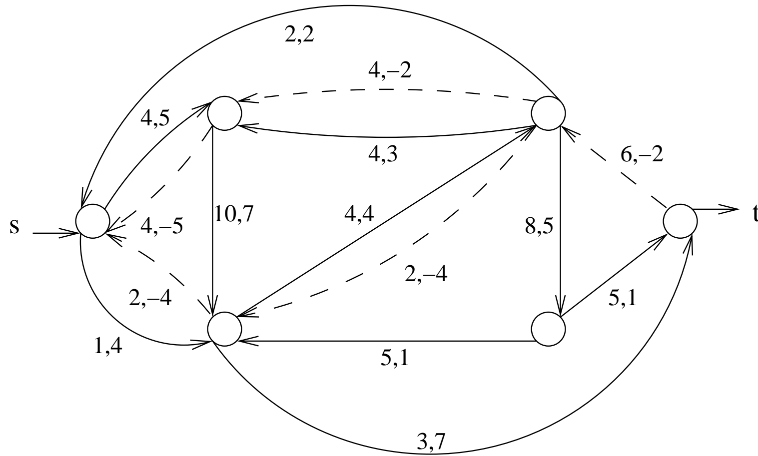


Figure 13: For the flow given in Figure 12 we produce the corresponding residue graph. Arcs which are dashed are reverse arcs. Solid arcs are forward arcs. This time the arcs are labeled $(u - x, c)$ by remaining capacity, and cost. Note that the reverse arcs whose antagonist have no flow have been left out, since they have no capacity. Likewise forward arcs which are carrying flow to their capacity are not included.

graph G at some intermediate stage of an MCMF computation. While at times we may refer to G or $G(x)$ separately, only as input and output to our routines will the graph G by itself be the sole description of the underlying electrical network. Similarly, a network is only ever described in terms of $G(x)$ alone when discussing the network at some intermediate point in a computation. That being said, we now turn to some key properties of our networks from which our algorithms for the MCMF problem are derived.

4.3 Principles and Conditions of Optimality

There are two key properties of network flow problems as phrased in terms of residual networks $G(x)$ which pretty much write their own algorithms. Unfortunately, since this is not a paper on Graph Theory in general, in some cases we will be forced to leave much of the dirty work to our references in order to move on to more interesting results. The algorithm which can be developed from our first principle is straightforward. The second principle (the Reduced Cost optimality conditions) is a useful extension of the first, and the algorithms which we discuss for the remainder of the paper employ this so-called "certificate of optimality". The algorithms which are developed from each result will make extensive use of subroutines and data structures already familiar to us at this point (Dijkstra SSSP, and adjacency lists).

Negative Cycle:
This first optimality condition applies to an existing feasible flow on a intermediate network. Given a feasible flow x and corresponding (intermediate) residual network $G(x)$ the *Negative Cycle Optimality Condition* describes the following certificate of optimality. That is, the feasible flow x is optimal if and only if it satisfies this condition. However, this condition is also a principle which can be employed in order to improve the flow x with respect to the objective (13). First, a few comments on negative cycles are in order. A *negative cost directed cycle* in a any graph $G = (N, A)$ (we will use $G(x)$) is a cycle $W = \langle v_1, v_2, \ldots, v_{n-1}, v_n = v_1 \rangle$ such that $(v_i, v_{i+1}) \in A$, $u_{v_i v_{i+1}} - x_{v_i v_{i+i}} > 0$ (i.e. the flow can still be augmented along (v_i, v_j)), for $i \in \{1, 2, \ldots, n-1\}$, and most importantly $\sum_{i=1}^{n-1} c_{v_i v_{i+1}} < 0$ (negative total cost on W). Note that augmenting flow along such a cycle does not violate the mass-balance constraints b, because the augmented inflow equals the augmented outflow along the arcs of the cycle.

Theorem 4 (Negative Cycle Optimality Conditions). *A feasible solution x^* is an optimal solution to the Minimum Cost Maximum Flow problem if and only if the residual network $G(x^*)$ contains no negative cost directed cycles.*

Proof. Showing the first direction of the if and only if statement is rather straightforward. Suppose that x^* is an optimal solution to the Minimum Cost Maximum Flow Problem. Furthermore, suppose (for the sake of contradiction) that there exists a negative cycle W in $G(x^*)$. But then, clearly we can augment some positive amount flow along the arcs of W thereby improving the objective function value (13). This contradicts that x^* is was an optimal flow. Therefore we conclude that a feasible solution x^* is not optimal an solution to the Minimum Cost Maximum Flow problem if the residual network $G(x^*)$ contains a negative cost directed cycle.

Showing the other side of this if and only if statement is less straightforward, and will rely on several key properties and results concerning *augmenting cycles*[22] and *flow decomposition*[23]. While we state and use these results here, we leave their formulation and proofs to our references [1]. The following, then, is more of a proof sketch than a true proof.

Define $\text{Cost}(x)$ to be the total cost of a flow x in $G(x)$. More precisely,

$$\text{Cost}(x) = \sum_{(i,j)\in A(G(x))} x_{ij}c_{ij}. \tag{16}$$

The *augmenting cycle property* states that for any two feasible flows x and x' we have that the flow x is the same as x' plus the flow on at most $m = |A(G(x'))|$ directed cycles in $G(x')$. Furthermore, if c is the sum of the cost of augmenting flow on these cycles in $G(x')$, then the total cost of x with respect to x' is $\text{Cost}(x) = \text{Cost}(x') + c$. Or, equivalently $c = \text{Cost}(x) - \text{Cost}(x')$.

Suppose then that $G(x)$ contains no negative cycle. Moreover, let x^* be an optimal flow with respect to (13) such that $x^* \neq x$, so we have that

$$\text{Cost}(x^*) \leq \text{Cost}(x) \tag{17}$$

as x^* is optimal. By the *augmenting cycle property* the difference between the flow x^* and x can be decomposed into at most m directed cycles in $G(x)$. In addition, the sum of the flows on these cycles must be $c = \text{Cost}(x^*) - \text{Cost}(x)$. Since $G(x)$ contains no negative cycles we must have that $c \geq 0$. Or, equivalently:

$$\begin{aligned} 0 \leq c &= \text{Cost}(x^*) - \text{Cost}(x) \\ 0 &\leq \text{Cost}(x^*) - \text{Cost}(x) \\ \text{Cost}(x) &\leq \text{Cost}(x^*). \end{aligned}$$

From (17) this implies that $\text{Cost}(x) = \text{Cost}(x^*)$, showing that the flow x is optimal, as desired.

Therefore, we conclude that a feasible solution x^* is an optimal solution of the Minimum Cost Maximum Flow problem if and only if the residual network $G(x^*)$ contains no negative cost directed cycles. □

This optimality condition immediately suggests an algorithm for solving the MCMF optimization problem. The procedure simply entails finding a feasible flow x in G and then while the resulting $G(x)$ contains a negative cycle, detect one of these cycles and then augment flow along it. An approach of this sort is one which maintains feasibility while moving toward optimality. Implementing such a routine requires an efficient algorithm for identifying negative cycles. A slight modification to the Floyd-Warshall APSP algorithm can accomplish this. If any intermediate adjacency matrix A_k in the Floyd-Warshall APSP algorithm run on $G(x)$ has a negative entry on its diagonal (i.e. $A_k[i,i] < 0$ for some $0 \leq i \leq |N|$), then there is a negative cycle which begins and ends at node i. Finding the other nodes and arcs on the cycle is then a simple matter of tracing through the associated predecessor matrix.

Of course, then there is the question of establishing an initial feasible flow (without regard to cost) to start off this *Cycle Canceling* algorithm. This is basically an optimization problem in its own right, known as the *Maximum Flow* problem. While there are well-known algorithms for solving such problems (including

[22] Theorem 3.7 [1]
[23] Theorem 3.5 [1]

almost trivial applications of the simplex method) we end this discussion here in search of much more efficient algorithm which solves the MCMF problem in one swoop. This algorithm relies on a re-phrasing of the negative cycle optimality condition.

Reduced Cost:

Our second optimality condition relies heavily on the following observation. For the shortest paths problem (in particular, one that can be solved by the Dijkstra SSSP algorithm) as long as there are no negative cycles in the graph $G = (N, A)$ we had that an arc (i, j) is only present in the shortest paths tree (part of the optimal solution) for some source s if

$$d(j) \leq d(i) + c_{ij}. \tag{18}$$

Recall that if there were negative cycles in G the shortest paths problem would not be well-defined. Rewritten as

$$c_{ij}^d = c_{ij} + d(i) - d(j) \geq 0 \quad \forall (i, j) \in A \tag{19}$$

we define the *reduced cost* of an arc $(i, j) \in A$ for the shortest paths problem. c_{ij}^d can be thought of as measuring the cost of the arc (i, j) relative to the shortest path distances from some source to its end vertices. Note also that c_{ij}^d is always nonnegative. What these reduced costs tell us is that if we knew the shortest paths tree distances d with respect to some source, finding the actual shortest paths would be easy since every arc in the tree would have zero reduced cost. Our goal will be to enumerate similar conditions for the Minimum Cost Maximum Flow problem.

With the goal in mind of producing similar reduced costs for arcs in $G(x)$ with respect to the MCMF problem, we replace d in (19) by the unknown potential vector π. Without actually quantifying π we can define the *reduced cost* of an arc for an instance of an MCMF problem and show some important properties with respect to these reduced costs which are independent of a chosen potential. Eventually we will prove a theorem which will produce a suitable potential, and enumerate the *reduced cost optimality conditions.* Our ultimate goal will be to show that an optimal solution to the MCMF problem must satisfy these conditions. This result will give us a nice handle on an iterative method for solving the MCMF problem much in the same way as (18) did for the shortest paths problem.

With this in mind, let the reduced cost for the MCMF problem on graph the $G(x)$ be written as

$$c_{ij}^{\pi} = c_{ij} - \pi[i] + \pi[j] \geq 0 \qquad \forall (i, j) \in A(G(x)). \tag{20}$$

Here $\pi[]$ can be thought of as a mechanism for altering the cost of an arc (i, j) so as to reflect its utility to some source, without making it negative. This cost is the reduced cost c_{ij}^{π}.

Lemma 1 (Directed Paths with respect to c_{ij}^{π}). *For any directed path*

$$P = \langle v = v_1, v_2, \ldots, v_k = w \rangle$$

we have that

$$\sum_{(i,j) \in P} c_{ij}^{\pi} = \left(\sum_{\forall (i,j) \in P} c_{ij} \right) - \pi[v] + \pi[w] \tag{21}$$

Proof. From the definition of c_{ij}^{π} we have that

$$\sum_{(i,j) \in P} c_{ij}^{\pi} = \left(\sum_{(i,j) \in P} c_{ij} - \pi[i] + \pi[j] \right) \tag{22}$$

$$= \left(\sum_{(i,j) \in P} c_{ij} \right) + \left(\sum_{(i,j) \in P} -\pi[i] + \pi[j] \right). \tag{23}$$

But then the second sum in the last line above is a telescoping sum.

$$\begin{aligned}\sum_{(i,j)\in P} -\pi[i]+\pi[j] &= \pi[v_1]-\pi[v_2]) + (\pi[v_2]-\cdots-\pi[v_{k-1}]) + (\pi[v_{k-1}]-\pi[v_k]) \\ &= \pi[v_1]-\pi[v_k] \\ &= \pi[v]-\pi[w].\end{aligned}$$

From this result and (23) we have that

$$\sum_{(i,j)\in P} c^{\pi}_{ij} = \left(\sum_{(i,j)\in P} c_{ij}\right) - \pi[v] + \pi[w]$$

as desired. □

This lemma tells us that the reduced cost along path a $P = \langle v = v_i, v_2, \ldots, v_k = w\rangle$ has little to do with the choice of potential function π in that it is only affected by π at the origin and terminus of P. The choice of potential has even less of an impact on the cost of directed cycles.

Lemma 2 (Directed Cycles with respect to c^{π}_{ij}). *For any directed cycle $W = \langle v = v_i, v_2, \ldots, v_k = v\rangle$ we have that*

$$\sum_{(i,j)\in W} c^{\pi}_{ij} = \sum_{(i,j)\in W} c_{ij} \tag{24}$$

Proof. The proof of (2) is almost exactly the same as the proof of Lemma 1. However, instead of (24) we have $\pi[u]-\pi[u]=0$, and thus from (23) we have $\sum_{(i,j)\in W} c^{\pi}_{ij} = \sum_{(i,j)\in W} c_{ij}$, as desired. □

The previous lemma is very useful because it provides an anchor to the cycle canceling optimality condition. Namely, regardless of our choice of potential, if W is a negative cycle with respect to c_{ij} then it is also a negative cost cycle with respect to the reduced costs c^{π}_{ij}.

With these properties of reduced costs c^{π}_{ij} under our belt we now have the artillery necessary to proceed with showing and formalizing the result which will be the backbone of the remainder of the paper.

Theorem 5 (Reduced Cost Optimality Conditions). *A feasible solution x^* to the minimum cost maximum flow problem on $G = (N, A)$ is an optimal solution if and only if there exists a set of node potentials π such that*

$$c^{\pi}_{ij} = c^{\pi}_{ij} - \pi[i] + \pi[j] \geq 0 \quad \forall (i,j) \in A(G(x)). \tag{25}$$

Proof. We shall show this result by showing that the reduced cost optimality conditions ($c^{\pi}_{ij} \geq 0$) are equivalent to the negative cycle optimality conditions of Theorem 4.

First, suppose that the flow x^* satisfies the negative cycle optimality conditions with respect to $G(x^*)$. That is, $G(x^*)$ contains no negative cycles. Now, choose any $s \in N$, and let $d(j)$ be the shortest path from s to every node $j \in N$. Since there are no negative cycles in $G(x^*)$ these distances are well-defined. From (18) we have that

$$d(j) \leq d(i) + c_{ij} \quad \forall (i,j) \in A(G(x^*)).$$

However, this can be rewritten as

$$c_{ij} + d(i) - d(j) \geq 0 \tag{26}$$

$$c_{ij} - (-d(i)) + (-d(j)) \geq 0 \tag{27}$$

for all $(i,j) \in A(G(x^*))$. If we substitute

$$\pi[i] = -d(i) \quad \forall i \in N \tag{28}$$

into (27) and call the result c^π_{ij} we get

$$c^\pi_{ij} = c_{ij} - \pi[i] + \pi[j] \geq 0. \tag{29}$$

Thus we have defined the potentials π which satisfy the reduced cost optimality conditions, namely $c^\pi_{ij} \geq 0$, subsequently establishing (25).

On the other hand suppose that the flow x^* satisfies the reduced cost optimality conditions. That is, for every arc $(i,j) \in A(G(x^*))$, $c^\pi_{ij} \geq 0$. Lemma 2 says that for any directed cycle W in $G(x^*)$

$$\sum_{(i,j)\in W} c^\pi_{ij} = \sum_{(i,j)\in W} c_{ij} \geq 0$$

and so $G(x^*)$ can contain no negative cycle, as desired. □

In general, we say that a flow x *satisfies* the reduced cost optimality conditions ($c^\pi_{ij} \geq 0$) when the reduced costs c^π_{ij} are nonnegative for $(i,j) \in A(G(x))$, where $G(x)$ is the residue graph imposed by x.

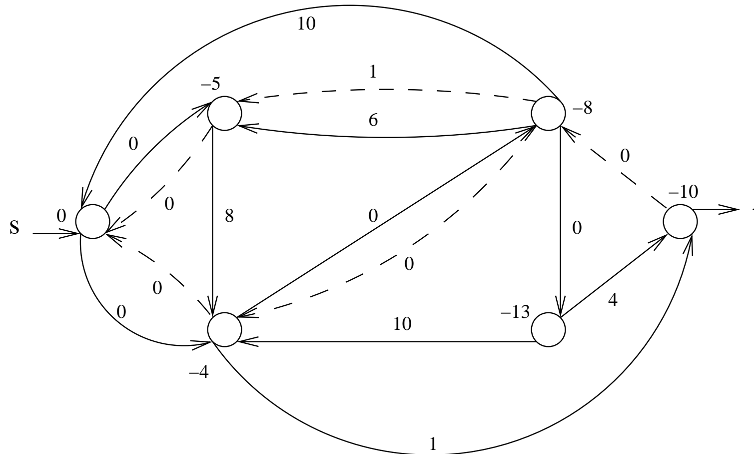


Figure 14: Here we have assigned potentials to each node of the residue graph given in Figure 13. The arc labels represent the reduced costs. Note that they are all non-negative. As we will see shortly, these potentials were not chosen arbitrarily, and neither was the flow which generated this residue graph.

The result provided by Theorem 5 suggests a natural algorithm. Since the node potentials turned out to be closely related to shortest paths distances there is likely to be a shortest paths subroutine in a larger algorithm which exploits the reduced cost optimality conditions. See Figure 14 for an example of reduced costs on a residue graph. We proceed now with the description of such an algorithm. As it turns out, the resulting technique generalizes nicely to the Convex Minimum Cost Maximum Flow problem which best suits our ESD event simulation problem.

4.4 Successive Shortest Paths Algorithm

By employing Theorem 5 the Successive Shortest Paths algorithm maintains optimality while attempting to attain feasibility. While the algorithm never violates the capacity constraint (10), an intermediate flow in

the algorithm may not satisfy the demand described by the vector b. Such a flow which satisfies the capacity constraint but fails to satisfy the mass balance constraint of some of the nodes in N is called a *pseudoflow*. We define the *imbalance* of a node i with respect to a pseudoflow x as

$$e(i) = b(i) + \left(\sum_{\{j:(j,i) \in A(G(x^*))\}} x_{ij} \right) - \left(\sum_{\{j:(i,j) \in A(G(x^*))\}} x_{ij} \right) \quad \forall i \in N. \tag{30}$$

Like the mass balance constraint b, if $e(i) > 0$ then we say that $i \in N(G(x^*))$ is in *excess*, and when $e(i) < 0$ we refer to i as being in *deficit*. Otherwise, if $e(i) = 0$ we must have that $e(i) = b(i)$, and in this situation we say that the node i is in balance. Similarly, as in (12) we enforce that

$$\sum_{i \in N} e(i) = 0. \tag{31}$$

Therefore, if we define the sets E for excess, and D for deficit

$$E = \{i \in N : e(i) > 0\} \tag{32}$$
$$D = \{i \in N : e(i) < 0\} \tag{33}$$

we also have that (as required by (12))

$$\sum_{i \in E} e(i) = -\sum_{i \in D} e(i). \tag{34}$$

In our original formulation of the successive shortest paths algorithm the imbalance e of the intermediate graph $G(x)$ will always be in direct correspondence with the mass balance constraints b. That is, once a node i is in balance (i.e. $e(i) = 0$) it will remain in balance for the remainder of the algorithm. However, when we employ a so-called *capacity scaling* technique for boosting the performance of the algorithm, such will often not be the case. In addition, it is possible that the reduced costs become negative in this technique, which will require some work around when we get to that section.

In the next lemma and theorem let $d(\cdot)$ represent the shortest paths distances from some node s to all other nodes $t \in N$ with respect to the reduced costs c^{π}_{ij}. Therefore, from (18) we have that

$$d(j) \leq d(i) + c^{\pi}_{ij} \quad \forall (i,j) \in A(G(x)) \tag{35}$$

where π can be any valid node potential. Let x be a pseudoflow which satisfies c^{π}_{ij}.

Lemma 3 (Dynamic Node Potentials). *If x is a pseudoflow which satisfies $c^{\pi}_{ij} \geq 0$, then x also satisfies $c^{\pi'}_{ij} \geq 0$ where $\pi' = \pi - d$; where $d()$ is computed with respect to c^{π}_{ij} and some source s.*

Proof. Since the pseudoflow x satisfies the reduced cost optimality conditions with respect to π, we have that $c^{\pi}_{ij} \geq 0$. Substituting the definition $c^{\pi}_{ij} = c_{ij} - \pi[i] + \pi[j]$ into (35) we get

$$d(j) \leq d(i) + c_{ij} - \pi[i] + \pi[j] \quad \forall (i,j) \in A(G(x)). \tag{36}$$

Switching things around a bit:

$$\begin{aligned} c_{ij} + d(i) - d(j) - \pi[i] + \pi[j] &\geq 0 \\ c_{ij} - \pi[i] + d(i) + \pi[j] - d(j) &\geq 0 \\ c_{ij} - (\pi[i] - d(i)) + (\pi[j] - d(j)) &\geq 0 \end{aligned}$$

and letting $\pi[\cdot]' = \pi[\cdot] - d(\cdot)$

$$\begin{aligned} c_{ij} - \pi'[i] + \pi'[j] &\geq 0 \\ c^{\pi'}_{ij} &\geq 0 \end{aligned}$$

establishes the desired result. $\square$

This result suggests a legal dynamic realignment of the node potentials for the reduced cost of an intermediate solution to the MCMF problem. When push comes to shove we need to be able to assign flow to the arcs of the network. We would like to be able to do so in such a way so as to continue to maintain the optimality of the solution. Of course, the measure of this optimality is our reduced cost optimality conditions. We now claim that these conditions hold up under flow augmentation along shortest paths with respect to the reduced costs in the residual network, so long as we redefine the potential after every augmentation along such a path (as described by the previous lemma).

Theorem 6 (Successive Shortest Paths). *Suppose that in $G(x)$ there remain nodes s with excess and t with deficit. Then we can obtain a valid pseudoflow x' from x by sending flow along a shortest path in $G(x)$ with respect to c^{π}_{ij}, such that x' also satisfies the reduced cost optimality conditions.*

Proof. Consider the situation where, after augmenting flow along a shortest path, we redefine the node potentials as in Lemma 3 (whose result is such that the original flow x will also satisfy these new conditions). The situation is no different for the flow x', except that augmenting flow on an arc $(u, v) \in A(G(x))$ may introduce a new *antagonist* arc $\overline{(u, v)} = (v, u) \in A(G(x'))$ that was not present in $A(G(x))$. In such an event we must be careful to maintain that (v, u) satisfies the reduced cost optimality condition as well. Recall that (v, u) did not previously have to satisfy the reduced cost optimality conditions since it was not present in $A(G(x))$. Therefore, in order to establish the desired result, we must verify that $c^{\pi'}_{vu} \geq 0$ when (u, v) is introduced. However, consider the very next shortest path P from some source s to some sink t with respect to $c^{\pi'}_{ij}$. In this situation we necessarily have that

$$d(j) = d(i) + c^{\pi}_{ij} \quad \forall (i, j) \in P \tag{37}$$

since this is the very definition of an arc (i, j) being on a shortest path. Proceeding as in the proof to Lemma 3, now instead of $c^{\pi'}_{ij} \geq 0$ we get $c^{\pi'}_{ij} = 0$ for each arc $(i, j) \in P$. But then our arc (u, v) must have been on a shortest path in order to have its flow augmented, and $x_{uv} = 0$ as a forward arc (or $x_{vu} = c_{vu}$ as a reverse). Therefore, $c^{\pi'}_{uv} = 0$, and since

$$\begin{aligned}
-c^{\pi'}_{ij} &= -(c_{ij} - \pi'[i] + \pi'[j]) \\
&= -c_{ij} + \pi'[i] - \pi'[j]) \\
&= +c_{ji} - \pi'[j] + \pi'[i] \\
&= c^{\pi'}_{ji}
\end{aligned}$$

we must also have that $c^{\pi'}_{vu} = 0$, asserting that (v, u) satisfies the reduced cost optimality conditions with respect to π'. Therefore, all $(i, j) \in A(G(x'))$ satisfy the reduced cost optimality conditions. □

These results translate nicely into an iterative algorithm for solving the MCMF problem. Note that the node potentials play a key role in this algorithm. If we manipulate them correctly then not only do they provide us with a certificate of optimality, but they ensure that the arc costs with respect to c^{π}_{ij} are always nonnegative. This means that we can use an algorithm like Dijkstra SSSP for finding shortest paths from nodes with excess to others with deficit. Note that the reduced costs and the flow given for the network in Figure 14 was obtained by two iterations of the Successive Shortest Paths algorithm algorithm. Since the reduced costs induced by two $\pi = \pi - d$ updates, and the reduced costs of arcs of $G(x)$ are nonnegative, the given pseudoflow x is optimal. See Figure 15 for a pseudocode depiction of the Successive Shortest Paths algorithm.

Most of the algorithm is straightforward, in light of the previous argument. However, we have said very little about the details behind lines (11-13), and nothing about their implementation. Clearly, there are many ways to execute line (10). We have many choices when it comes to finding a shortest path P from a source s to some sink t. Several times we have suggested that Dijkstra's SSSP algorithm is an appropriate choice. However, Dijkstra's algorithm does much more than just find a shortest path between two nodes; we saw in the previous section that the Dijkstra SSSP algorithm finds shortest paths distances d from some

```
procedure SuccessiveShortestPaths(G=(N,A), b)
begin
(1)    flow x(i,j) = 0, for all (i,j) in A
(2)    potential pi[] = 0;
(3)    let G(x) = G      // either implicitly, or explicitly
(4)    imbalances e[] = b[]  // assert mass balance constraints
(5)    let E = {i: e(i)>0, i in N}
(6)    let D = {i: e(i)<0, i in N}
(7)    G(x)->setWeightsToReducedCosts(pi)
          // initially, just the costs

(8)    while (E not empty) do
(9)       choose some s in E and t in D
(10)      d[] = Shortest Path Distances from s
             with respect to the reduced costs
(11)      let P be the shortest path from s to t, by d[]
(12)      let delta = min { u(i,j) in path P, e(s), -e(t) }
(13)      x = augmentAlongPath(G(x), P, delta)
             // and making appropriate updates to G(x)
             // based on new flow
(14)      pi = pi - d
(15)      if (e(s) == 0) then discard s from E endif
(16)      if (e(t) == 0) then discard t from D endif
(17)      G(x)->setWeightsToReducedCosts(pi)
(18)   enddo
end
```

Figure 15: Pseudocode for the Successive Shortest Paths Algorithm, for solving the minimum cost maximum flow algorithm. Requires mass balance constraints b from (12) and the network G as input. We assume that the Shortest Paths Algorithm referred to in lines (7) nd (15) use a *weight* field for each arc in calculating the shortest paths. The pseudocode given here is an adaptation of similar code given in [1] (Chapter 9).

source s to *all* other nodes (sinks) $t \in N$. What does this mean for line (14) which calls for calculating $\pi' = \pi - d$? What exactly is behind choosing an appropriate amount of flow (δ) in line (12) to augment along P, and what exactly does such an augmentation entail?

To answer the first question, the distance vector d returned from Dijkstra's SSSP algorithm presents no trouble at all. In fact, Lemma 3 requires that d represent the shortest paths distances from the source s to all other nodes in the graph. This is just what we get from Dijkstra's SSSP algorithm. Notice that if we were to modify Dijkstra's algorithm to stop finding shortest paths once it finds a path to any node with excess, this will likely boost the run-time performance our overall successive shortest paths algorithm. However, one must be careful in this case not to revise a reduced cost when $d(i) = \infty$ for some node i which the shortest paths algorithm has not reached yet. In our implementation of Dijkstra's SSSP algorithm, if stopped early, it might have not found any path to some nodes, like i, and $d(i) = \infty$. In this case, performing the operation $\pi' = \pi - d$, may not be well-defined, and may indeed produce negative reduced costs! In this case we can adopt the convention the potentials remain unmodified for for arcs adjacent to nodes i where $d(i) = \infty$. Clearly, no flow is augmented on arcs adjacent to nodes unreachable from the source. Therefore, the old potential will suffice for keeping such an arc's reduced cost non-negative.

Determining just how much flow should be augmented along the path P is a simple matter of checking capacity constraints, excesses and deficits. Clearly we cannot augment more flow than can be sourced from the excess, or sinked at the deficit. Hence the last two terms in the *min* of line (12). Then, along the path P we cannot augment more than the remaining capacity for any arc, taking into consideration the amount of flow already on that arc. If the capacity for each forward and reverse arc in the network is included

in our graphical representation (as in our working representation discussed earlier), then determining an appropriate δ is just a matter of traversing the path, all-the-while computing a minimum of the remaining capacity minus the flow on those arcs. Thus we have that augmenting

$$\delta \;=\; \min\{\min_{\forall(i,j)\in P}\{u_{ij}-x_{ij}\}\;, e(s), e(t)\} \tag{38}$$

units of flow along P provides the maximum flow increase which still maintains the feasibility of the pseudoflow.

Once δ is determined, actually augmenting this amount of flow along an arc $(i,j) \in P$ is a matter of determining whether this arc is a forward arc or a reverse arc, and making the appropriate changes to $G(x)$. In the case where the arc $(i,j) \in P$ is a forward arc, this amounts to increasing the flow along x_{ij}, and increasing the capacity of its antagonist $\overline{(i,j)}$. If (i,j) is a reverse arc, we must find its associated forward arc $\overline{(i,j)}$, decrease the flow on this arc, and then recalculate the capacity of our original reverse arc. *If, after augmenting flow along any arc, the flow on this arc or its antagonist equals its capacity, then this arc must be (symbolically) removed, or temporarily decomissioned, from $G(x)$.*

For any path P, the maximum amount of flow that can be augmented on P is sometimes referred to as the *increment* of P, written $i(P)$. The increment of a single arc $a \in A(G)$, alone, is the difference between the amount of flow on the arc and its capacity, written $i(a)$. In this way, we have that $i(P)$ corresponds to the first set of terms in the minimum of (38). Normally, the increment of a reverse arc is written in terms of its antagonist, forward arc. However, since we have defined the capacity of reverse arcs to be equal to the amount of flow on its antagonist, forward arc, and we never allow reverse arcs to have nonzero flow, these two notions of increment are essentially equivalent.

After augmenting flow along the path returned by running Dijkstra's SSSP algorithm on $G(x)$ with respect to the reduced costs, and after the potentials have been updated, all there is left to do is recalculate the reduced costs for the arcs of $G(x)$. However, given π, the pseudocode of lines (17) and (6) take the place of a simple subroutine which can loop over all of the arcs of $G(x)$ and calculate the reduced cost of each arc, and subsequently set its weight w to be this value.

Correctness:

Having built up this algorithm from several lemmas and theorems all regarding optimality, and describing how to move from one optimal solution to another; and then producing an algorithm motivated by these results; we must only show that the pseudocode of Figure 15 does indeed converge to an optimal solution to the MCMF problem.

- Theorem 6 shows that after each iteration the reduced cost optimality conditions are satisfied. Moreover, since the initial flow $x = 0$ is a valid pseudoflow, and since π starts as the zero vector we have that all of the reduced costs are non-negative

$$\begin{aligned} c_{ij}^{\pi} &= c_{ij} - \pi[i] + \pi[j] \\ &= c_{ij} - 0 + 0 \\ &= c_{ij} \geq 0 \qquad \forall (i,j) \in A(G(x)). \end{aligned}$$

  Furthermore, since we have assumed that all of the arc-costs themselves are non-negative, we have that $x = 0$ satisfies the reduced cost optimality conditions. Inductively, assuming that before an iteration of the *while* loop of line (8) x is a valid pseudoflow which satisfies the reduced cost optimality conditions, Theorem 6 shows that x' induced by the flow augmentation along P of line (12) must also satisfy these conditions before the next iteration.

- Each time flow is augmented, an amount of flow described by the increment $i(P)$ is sent into each node, and out of that node, for every node along this path– which is not the source s or sink t. Therefore, the balance of each node $i \in N(P) - \{s,t\}$ is preserved. That is, each $b_x(i) = b_{x'}(i)$ for an initial flow x and the flow x' resulting from the augmentation, along P.

- After each iteration of the *while* loop the excess and deficit of two nodes are reduced, so (as long as we enforce that a certain minimum δ is augmented along each augmenting path) we have that E

and D must eventually empty. Recall that we required previously that $G(x)$ must always contain an uncapacitated path between every pair of nodes in N. In particular, there will be an uncapacitated path for every excess and deficit pair. Once E and D have been emptied, our algorithm has satisfied the mass balance constraints b, and this means it must have established a feasible flow. Furthermore, since this flow satisfies the reduced cost optimality conditions, and the supply/demand vector b, we can conclude that this flow is a solution to the MCMF problem.

The above facts and the supporting arguments given previously verify that the algorithm given in Figure 15 solves the combinatorial optimization problem as given (13) and (14). We have thereby established the correctness of the Successive Shortest Paths algorithm given in Figure 15. □

Runtime:
The runtime of the Successive Shortest Paths algorithm depends heavily on the shortest paths algorithm we choose to use as a subroutine. Having already enumerated a good shortest paths algorithm, namely Dijkstra SSSP which runs in $O(n^2)$ time, we can certainly get an upper bound on the running time of the entire procedure by determining the number of iterations of the *while* loop in line (8) of Figure 15. Each time through the while loop some amount of excess and deficit is relaxed, as computed in line (11). If we enforce that capacities and flow augmentations are integral, then in the worst case this amount flow is 1. This integrality assumption will be made throughout our further discussion. It is from this observation that we can conclude that after each iteration of the *while* loop in line (8) at least one unit of flow is augmented along a path from an excess node to a deficit node. Therefore, in the worst case, if U is the initial sum of the excess over the entire network, i.e.

$$U = \sum_{i \in N} e(i), \tag{39}$$

then U must also be the maximum number of iterations in the while loop. From this we conclude that the Successive Shortest Paths algorithm runs in time $O(Un^2)$. However, since we showed in the previous section that with different, more clever, priority queue implementations we can do better than $O(n^2)$ with Dijkstra's SSSP algorithm on sparse graphs. See Figure 16.

	Trivial Array	**Fibonacci Heap**	**Binary Heap**
SSP Algorithm	$O(Un^2)$	$O(Um + Un\log n)$	$O(Um\log n)$

Figure 16: Addendum to the Complexity Table for three Priority Queue data structures often used in Dijkstra's SSSP algorithm.

We would like to be able to compare these results to other algorithms for solving the MCMF problem. Currently, we have only one other algorithm, namely the Cycle Canceling Algorithm. This algorithm requires two powerful subroutines, namely one which establishes a maximum flow, and another which identifies negative cost cycles. Using such high level subroutines is doomed to fail. As already mentioned, we can use the Floyd-Warshall APSP algorithm to identify negative cycles. We might even be able to come up with an algorithm for identifying negative cost cycles in $O(n^2)$ time, perhaps by modifying the Dijkstra SSSP routine. However, even if this were possible we may need to identify U many of these cycles which means that we are already doing no better than the SSP algorithm. While there are still other algorithms for solving the MCMF problem[24] we leave these descriptions to our references, in particular [1]. Each of these algorithms has some advantage over the SSP algorithm, though not necessarily in terms of worst case complexity.

Note that since the value U does not necessarily need to be polynomial in the number of nodes or arcs, the runtime of any of the algorithms mentioned above is only *pseudo-polynomial*. However, there are

[24]the Primal-Dual algorithm which is similar to the SSP algorithm, but involving sending flow along all shortest paths rather than the one between the excess and deficit; the Out-Of-Kilter algorithm which contrary to the SSP algorithm maintains a feasible flow at every iteration, and strives to attain optimality by augmenting flows on shortest paths; as well as others like the Relaxation Algorithm which relies on some heuristics which make it efficient for most MCMF instances.

scaling techniques which produce polynomial analogues of these procedures. The SSP algorithm receives the largest boost of the pseudo-polynomial algorithms by employing this scaling procedure. We will illustrate the *capacity scaling* technique following the Convex Flow section, and then conclude this discussion with a polynomial time algorithm for solving our ESD Budget Analysis Problem.

4.5 Convex Flows

Finally, there are two fundamental differences between the general MCMF optimization problem and the problem we are trying to solve.

The first difference is that in some sense the MCMF optimization problem as formulated at the beginning of this chapter is too general. We are only interested in the excitation level between bondpad pairs when one of them is "zapped" and the other is grounded. While it is entirely possible that more than one of the bondpads of the integrated circuit I/O be "zapped", and more than one grounded, this generalization is a less extreme case. Certainly, if the ESD protection holds up under the one-source one-sink model, a protection scheme can provide more protection when there are more avenues on which the current can flow. This is just what happens when the current enters and/or exits the device from more than one terminal. Furthermore, having a capacity constraint on an electrical device such as a resistor is in most respects an unrealistic modeling, and therefore not a necessarily useful variable in the problem. If we were to impose a capacity constraint, what would it be? Would this constraint help us model our situation under fixed, or variable currents? Such is not likely if we wish to model an electrical network during an ESD event.

The second difference is that the arcs in our network represent resistors, diodes, etc which carry current. We are essentially faced with the same problem as when we discussed Shortest Paths for overestimating excitation levels, earlier. While we now a have mechanism to augment flow on multiple parallel paths, we have little mechanism to force current flow to act as it would in a real simulation. In fact, what we really want to know is: How much current is flowing on each resistor (arc) in the network. Clearly, if we could obtain this information we will have solved the crux of our problem. Once we have the amount of current flowing on each component in the electrical network during a simulated ESD event, calculating the voltage drop between terminal pairs is a simple matter of applying Ohm's law to each component along any single path, and summing voltage drop contributions. This will be the culminating result of this section.

Our first task is to rephrase the optimization problem. The motivation behind this new formulation is that Ohm's law shows that a resistor with resistance r dissipates rI^2 watts of power, where I is the amount of current flowing along the resistor. Therefore, we wish to find the *equilibrium* current on each arc in our network under this cost model. That is, for each arc $(i,j) \in A(G)$ which collectively depicts our network graphically, we wish to learn the equilibrium current x_{ij} for each arc where the cost of augmenting current along the arc (i,j) varies (convexly as opposed to just linearly) as the square of the amount of flow along that arc, times its resistive component c_{ij}. With this in mind we rewrite the Minimum Cost Maximum Flow problem of (13-15) as

$$\text{minimize} \quad \sum_{j=1}^{n} C(x_{ij}) \tag{40}$$

$$\text{subject to} \quad \sum_{j:(i,j)\in A}^{n} x_{ij} - \sum_{j:(i,j)\in A}^{n} x_{ji} \quad = \quad b(i) \qquad \forall v_i \in N \tag{41}$$

$$0 \leq x_{ij} \leq u_{ij} \qquad \forall (i,j) \in A \tag{42}$$

and take our convex cost C to be $C(x_{ij}) = c_{ij} * x_{ij}^2$. That is, the objective (40) changes, but everything else stays the same. We make the one general assumption that $C(x_{ij}) = 0$ when $x_{ij} = 0$, but this is already taken care of by the quadratic convex cost function we have chosen. A formulation of this type, where the cost of an arc varies convexly in the amount of flow placed along it, is referred to as a Convex Minimum Cost Maximum Flow Problem, Convex Cost Flow, or simply Convex Flow.

As mentioned previously, being able to specify a capacity for each arc in the network isn't very relevant or helpful since we are interested in finding the optimal amount (*equilibrium*) current along each resistor, and don't really care how much current is on any single resistor. Therefore we let $u_{ij} = \infty$ for all $(i,j) \in A(G)$. We may wish, however, to keep the capacity constants around for the reverse arcs in a residue graph since augmenting flow on a reverse arc is essentially reducing flow on the antagonist forward arc, as we can only reduce as much flow as is currently assigned to the forward arc.

Since we haven't changed the overall optimization problem much, we should be able to get away with a similar algorithmic approach to producing the optimal flow as was used for the (linear cost) MCMF problem. In fact, this is just what we will do! The goal of the remainder of the section will be to adapt the Successive Shortest Paths algorithm of Figure 15 to the Convex Flow optimization problem, and further fine tune it to our specific ESD Analysis problem. The biggest difference is that whenever flow on an arc is augmented, the cost of that arc changes! That is, we need to compute the costs of arcs based on the amount of flow we intend to augment along that arc. However, such foresight is not easy to abstract or program. Although it may seem like we can augment any amount of flow we choose along a path from the source to the sink, there are several reasons why it is not to our advantage to do this.

1. Recall that reverse arcs in $A(G(x))$ cannot have infinite capacities, and shortest paths often contain reverse arcs.
2. The more flow we augment along each path the faster we will come to a maximal flow, (that is, we may be able to find a feasible flow faster, one which satisfies the mass balance constraints), but the less accurate our calculation of the equilibrium currents along each arc in the final flow will be.

Therefore, in order to get an appropriate minimum $\hat{\delta}$ amount of flow to augment along a path, we insist that in order for an arc to be (symbolically) included in the residual network $G(x)$, that the capacity of the arc is at least $\hat{\delta}$ so that the increment $i(a) \geq \hat{\delta}$ for all $a \in A(G(x))$. This way we can shortcut the computation of $i(P)$ when preparing to augment flow. We would like to further insist only a certain minimum $(\hat{\delta})$ amount of flow is augmented along a path at any one time, so that future arc-costs are predictable. Essentially, the effect we are aiming for is a like a piecewise linearization of the convex cost of an arc into segments of a fixed length $(\hat{\delta})$. This can accomplished simply by setting the increment $\delta = \hat{\delta} \leq i(P)$ in line (11) of Figure 15, and be sure to subsequently recompute the cost of an arc after flow has been augmented on it, knowing that the next augmentation will also attempt to augment this same amount of flow. The most sensible value for $\hat{\delta}$ is 1. There are at least two reasons for this. Firstly, we would like the linearization to provide a nice approximation to the true convex cost function, and secondly because working with integers is nice. In other words, we now require that flow be incremented one unit at a time; no more, no less.

Implementationally, we define the convex cost of each edge $(i,j) \in A(G(x))$ to be cc_{ij}. That is, cc_{ij} is the convex cost of augmenting one unit of flow along arc $(i,j) \in A(G(x))$.

$$cc_{ij} = C(x_{ij}+1) - C(x_{ij}), \quad \text{for forward arcs } (i,j) \in A(G(x)) \tag{43}$$

and

$$cc_{ji} = C(x_{ij}-1) - C(x_{ij}), \quad \text{for reverse arcs } (j,i) \in A(G(x)). \tag{44}$$

In this way we have that the cost of augmenting flow along any arc can vary convexly in the amount of flow already on that arc– for a fixed amount of flow which can be augmented each time through the while-loop of Figure 15. In the case of forward arcs this means that the cost will be greater than before the previous augmentation. For reverse arcs this means that the cost will be less (yielding a larger refund) than before the previous augmentation. Computing the reduced cost of an arc under these revisions must use the convex costs

$$c^{\pi}_{ij} = cc_{ij} + \pi[i] - \pi[j]. \tag{45}$$

Having modified the successive shortest paths algorithm it is important to discuss the information gained in terms of equilibrium currents from the resulting minimum cost convex flow. If the current I incident on the circuit is a simulated ESD event (either through one source, or many sources) and U is the sum of the excess (31), then we can interpret the resulting equilibrium current as a percentage of the total current on any arc in the graph G. In this way, the convex flow simulation has essentially done the hard work of indirectly

applying Kirchoff's Current and Voltage Laws, assigning currents to arcs using Ohm's Law. Returning to the single source and single sink model, all that is left to do is find a path $P = \langle s = v_1, v_2, \ldots, v_k = t \rangle$ from the source to the sink (e.g. using a *breadth first* search). Then, if $f = U$ represents 100% of the total amount of current sourced on the network, we can determine the exact voltage drop along a path from the source to the sink by computing the sum

$$\text{aVd} = \frac{I}{f}\left(\sum_{i=2}^{n} c(v_{i-1}, v_i) \cdot x^*(v_{i-1}, v_i)\right) \tag{46}$$

where I represents the current of the ESD event based on some Human Body Model (HBM). We call this quantity the *actual voltage drop* (or aVd), which is a near perfect *approximation* to the true excitation level between terminal pairs. To clarify, we say *near perfect* because we are only augmenting flow in integral amounts. Note that this is no great limitation since by scaling rational input we can get results to any desired accuracy. Here, we use f rather than U to distinguish this quantity it as a variable. It will be a parameter to the Convex SSP analysis. We will discuss the accuracy of this simulation, the wisdom behind choosing f, and some results in the following section. Note that while we can perform a simulation where there are multiple sources and multiple sinks and obtain useful current information, the computation of an excitation level between pairs of terminals isn't as straight-forward. For each pair of source and sink we can get ESD Budget information, but only all of the paths together can accurately represent the results of a simulated ESD event. Perhaps taking the maximum of the voltage drop for each pair would yield the desired result, but this is no more efficient than running a separate simulation for each pair of sources and sinks.

At this point we rewrite the Successive Shortest Paths algorithm for our new quadratic cost setting, single source, single sink, and finally the computation of the actual voltage drop. See Figure 17.

Note that Figure 17 makes no explicit use of the *imbalances* e or the sets E and D, describing them earlier was not in vain. We will use them again when we discuss the capacity scaling technique for finding a polynomial time algorithm. It should be obvious that the worst case complexity of this modified Successive Shortest Paths algorithm is the same as the original one discussed earlier. In fact, since we only augment $\delta = 1$ units of flow at each iteration of the while loop the worst case time complexity is the same as the best case. The summary of the complexities like Figure 16 is included for completeness in Figure 18.

After computing the optimal (equilibrium) current on each arc, we wish to compute actual voltage drop (aVd) between the source and the sink. This is done just as we have already described, and is illustrated in Figure 19. We might choose to use Dijkstra's SSSP algorithm using the flow computed in Figure 17 as our weights, or simply set all of the arc weights to one– thus simulating a breadth first search. Nonetheless, we illustrate the breadth first search explicitly in Figure 19, and other details involved in computing the actual voltage drop.

Shortest Paths Overestimation (Theorem 1) Revisited:

In the shortest paths section of this paper we showed in Theorem 1 that the resistance along any path P from some source s to any sink t (not considering parallel components) is an overestimate of the effective resistance between s and t. In the proof of this theorem we promised to show an alternative method which uses Network Flows. We now have the appropriate artillery to do this, namely Convex Flows.

The fraction of the total current assigned to each arc $(i, j) \in P$ is x_{ij}. When we compute the adjusted voltage drop as in Figure 19 and (46) the result is a scaling down of a similar flow where instead we could have

$$x_{ij} = \frac{f}{n-1} = \frac{U}{n-1} \quad \forall (i,j) \in P. \tag{47}$$

This basically means that each arc in the path is carrying 100% of the flow among its $n-1$ components (between n terminals). But then this is just the same as the resistance along any single path (without parallel components). In Figure 19 we chose to take the path to be that with the least number of arcs, but this was only for efficiency of the routine– as the computed excitation level should be the same for any $s \to t$ path.

```
let C(x) = x^2

procedure ESDbudgetSSP(G=(N,A), source, sink, flow)
begin
(1)   flow x(i,j) = 0, for all (i,j) in A
(2)   potential pi[] = 0;
(3)   let G(x) = G      // either implicitly, or explicitly
(4)   rem_flow = flow
(5)   s = source
(6)   t = sink
(7)   G(x)->setWeightsToReducedCosts(pi)
         // initially, just the costs

(8)   while (rem_flow > 0) do
(9)      d[] = Shortest Path Distances from s to t
            with respect to the reduced costs
(10)     let P be the shortest path from s to t, by d[]
(11)     let delta = 1
(12)     x = augmentAlongPath(G(x), P, delta)
            // and making appropriate updates to G(x)
            // based on new flow
(13)     pi = pi - d
(14)     rem_flow = rem_flow - delta
(15)     recomputeConvexCostsAlongPath(G(x), P)
(16)     G(x)->setWeightsToReducedConvexCosts(pi, C(x))
(17)  enddo
end
```

Figure 17: Pseudocode for the ESD Budget Analysis Convex Cost Successive Shortest Paths Algorithm. Requires only a single source and a single sink node, and the desired amount of current "flow" f representing 100% of the current between them as input. We assume that the Shortest Paths Algorithm referred to in line (9) uses a *weight* field for each arc in calculating the shortest paths. Note that, implicitly, $E = \{s\}$ and $D = \{t\}$.

	Trivial Array	**Fibonacci Heap**	**Binary Heap**
ESDbudgetSSP	$\Theta(Un^2)$	$\Theta(Um + Un\log n)$	$\Theta(Um\log n)$

Figure 18: Addendum to the Complexity Table for three Priority Queue data structures often used in Dijkstra's SSSP algorithm.

In other words, we have that for any $s \to t$ path P with $n-1$ arcs, from (46) and the fact that $x_{ij} \leq f = U$ as a result of any Convex Flow

$$\begin{aligned} \text{aVd}\ (s \to t) &= \frac{I}{f}\left(\sum_{i=2}^{n} c(v_{i-1}, v_i) \cdot x^*(v_{i-1}, v_i)\right) \\ &\leq \frac{I}{f}\left(\sum_{i=2}^{n} c(v_{i-1}, v_i) \cdot \frac{f}{n-1}\right) \end{aligned}$$

```
procedure aVd(G=(N,A), source, sink, I, flow) -> real
begin
(1)   G->setAllArcWeights(1)
(2)   PriorityQueue pq
(3)   d(v) = infinity, forall v in N
(4)   d(source) = 0
(5)   pq->insert(source)
(9)   while (pq not empty) do
(10)    u = pq->min()
(11)    foreach v adjacent to u
(12)      if d(v) == infinity then
(13)        d(v) = d(u) + 1
(14)        p(v) = u
(15)        parc(u) = (u,v)
(16)        pq->insert(u)
(17)      endif
(18)      if v == sink then break endif
(19)    enddo
(20)    pq->remove(u)
(21)  enddo

(22)  cost = 0
(23)  u = sink
(24)  while (u != source) do
(25)    c(u,v) = parc(u)->cost
(26)    x(u,v) = parc(u)->flow
(27)    cost = c(u,v) * x(u,v) / flow
(28)    u = p(u)
(29)  enddo

(30)  aVd = I * cost  // I is current
(31)  return aVd
end
```

Figure 19: Given a graph G, with optimal flow which corresponds to equilibrium currents along its arcs, we wish to return the excitation level information. Lines (1-21) correspond to the breadth first search. If we find the sink before the search is finished, we cut it short in line (18). In lines (22-29) the adjusted voltage drop is computed.

$$
\begin{aligned}
&= \frac{f \cdot I}{f \cdot (n-1)} \left(\sum_{i=2}^{n} c(v_{i-1}, v_i) \right) \\
&= \frac{I}{n-1} \left(\sum_{i=2}^{n} c(v_{i-1}, v_i) \right) \\
&= \frac{I}{n-1} ((n-1) \cdot R(P)) \\
&= I \cdot R(P).
\end{aligned}
$$

From this we conclude that the actual resistance $A_{st} = \text{aVd } (s \to t)/I \leq R(P_{(s,t)})$, which is our desired result. As in the proof to Theorem 1, we have shown that the sum of the resistance along any path between a pair of terminals is an overestimate of the actual resistance, or corresponding excitation level. □

Concluding this section, we turn now to some appropriate uses of a simulation tool built around the algorithm of Figure 17, a few details and remarks, and some experimental test cases and results.

4.6 Some Results & Integrality of Data

So far we have said little about how to choose $f = U$, the total amount of flow to source. In fact, perhaps it is not even obvious yet why f is a "variable" in our ESD Budget Simulation technique. This variable is necessitated by the model mismatch which has arisen due to the fact that we have an integer-valued vector x representing a *fraction* of the total rational-valued current flowing on an electrical device. Since the flow on any arc in the network, as determined by our convex flow algorithm, represents a percentage of the total flow across the components of the entire network, and since we augment flow in integral amounts (namely unit amounts), this percentage is based on a fraction whose denominator is the total flow. Therefore, the larger the total amount flow the greater the precision of the resulting fraction. Note that when we speak of flow in this sense ($f = U$, the amount we source in a simulation) we are not speaking of current flow based on some HBM. This value I will be used only in converting to units of voltage from resistance for the final output.

In general, when dealing with parameters like voltage, current, and resistance which are typically represented as rationals in engineering or scientific notation, it is important to remember that computers don't do perfect rational arithmetic. That is, while computing a value like aVd, or most importantly the reduced costs, there may be truncation, roundoff, or overflow error resulting from floating-point arithmetic. Inaccurate computer arithmetic of this sort could cause catastrophic results. For example, arithmetic overflow might cause the reduced cost of some arc to become negative. This would void all of those nice results we showed about non-negative reduced costs, which the correctness of the algorithms of this section hinge on. In retrospect, the motivation behind our previous choice to augment flow in integral amounts was twofold. As already noted, this puts no real restriction on the types of data we can represent or use in our calculations. This is because it is always possible to scale rational input in order to coerce it to be integral, yet while retaining the measurement represented by the rational. However, one might need a "long unsigned" integer to represent such a rational as an integer to the desired precision. Initially, we could have made the assumption that all input data were integral. In fact, many Network Flow texts do just this. However, from a general and implementation standpoint, this is not usually necessary. All that is really necessary for our purposes is that the amount of flow which we augment (δ) is always bigger than some constant ϵ, throughout the entire algorithm, in order to guarantee that we eventually converge to an feasible solution.

Returning matter of choosing f: In terms of accuracy, the more we scale our rational input, and the more integral flow source in our simulation, the more accurate our aVd excitation level output will be. In principle, the source flow f should be no less than an integer multiple of the number of arcs in the resulting flow. Since it is not possible to know how many such arcs there will be in advance we might naively try increasing powers of 10. Following are several examples of this. In the first two examples the effective resistance between the source s and the sink t is a factor of more than two and three (respectively) less than the shortest paths result. In these small examples the effective resistance can easily be computed by hand. The third example serves as a demonstration of how the shortest path and convex flows compare on a larger, more expository example– the simulation of an ESD event on a real ESD protection scheme. These experiments were run on 280MHz HPUX workstations.

Example 1 (Parallel Resistors). *Two resistors of equal resistance in parallel. See Figure 20.*

The effective resistance of the parallel resistor in Figure 20 can be easily computed to be $\frac{1}{2}R$ by using the parallel resistor formula of (9). It is also quite obvious that any shortest paths result is R. In this simple example, any choice of *even* $f \geq 2$ yields the expected equilibrium currents of 50% for each resistor. These results are tabulated in Figure 21. This is because all of the (two) resistors will be used when current flows from s to t.

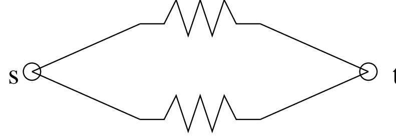


Figure 20: Parallel resistors $R = R_1 = R_2$.

	eff	SP	aVd $f = 2$	$f = 10$	$f = 11$	$f = 100$	$f = 1000$
Resistance	$R/2$	R	$R * 0.5$	$R * 0.50$	$R * 0.45\overline{45}$	$R * 0.500$	$R * 0.5000$
Time (sec)		$<< 0.5$	$<< 0.5$	< 0.5	< 0.5	0.5	$0.5 - 1.0$

Figure 21: Parallel Resistor ESD Budget Table

As this is a particularly simple example we get the exact result for all (even) positive values of f. Note that if we employ the resistor combination described for reducing parallel arcs at the end of the shortest paths section, we would get the exact result for the overestimation. Without it we are off by a factor of two. In our next example we will be able to better demonstrate the choosing of an appropriate total flow f.

Example 2 (Resistor Cube). *A classic example of multiple parallel paths is the resistor cube with twelve resistors of equal resistance. See Figure 22.*

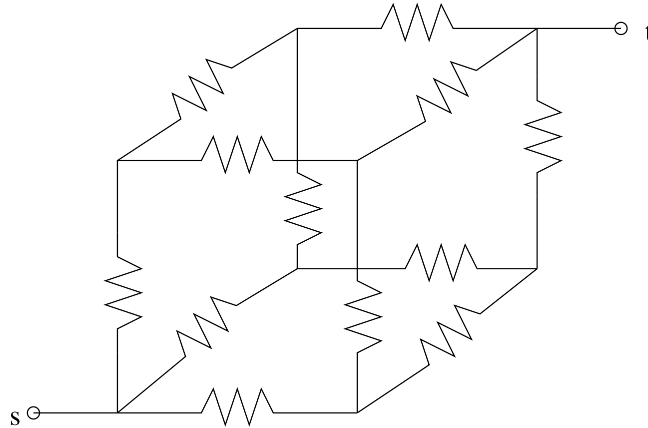


Figure 22: Equal R resistor cube.

In this example, since all twelve resistors would be involved in current flowing from s to t it makes no sense to source less than $f = 12$ units of flow. In fact, we will get the exact effective resistance for all $f = 12k$ for $k \in \{1, 2, \ldots\}$, and we do "good enough" for all 10^k for $k \in \{2, 3, \ldots\}$ thereby obtaining greater precision, but we must wait exponentially longer, and we require a larger integer representation, for increasing k. Note also that in this case the shortest paths resistance is very pessimistic, and that attempting to combine parallel arcs gives no improvement (since there are none).

The same results are obtained for any polygon of resistors. The aVd will agree with the effective resistance when the total flow f is a multiple of the number of resistors (or arcs) in the network as long as the

	eff	SP	$f = 12$	$f = 100$	$f = 1000$	$f = 10000$	$f = 100000$
Resistance	$5R/6$	$3R$	$R * 0.83\overline{3}$	$R * 0.82$	$R * 0.832$	$R * 0.8332$	$R * 0.83332$
Time (sec)		$<< 0.5$	$<< 0.5$	< 0.5	$0.5 - 1$	$1 - 2$	$2 - 3$

Figure 23: Resistor Cube ESD Budget Table

resistances on all of the arcs are the same. Unfortunately (or fortunately) ESD protection schemes are far more complicated than these simple examples. In general their graphical representations have components numbering in the thousands, and there is no easy way to decide how much flow f to source on the network. This is because there is no easy way (other than running a preliminary simulation) to tell how many arcs will carry flow between the source and sink. Not only that, but the resistance of the components an arbitrary network are not necessarily equal. Therefore, our last example will demonstrate the use of a tool implemented using the Convex Flow SSP algorithm on an actual I/O ring ESD protection scheme[25].

Example 3 (QUBiC3 D_Cell_Peripheral).

In this example our netlist had approximately 1800 components, which was reduced down to approximately 1500 by combining (600) immediately parallel components. At least three quarters of these arcs represented resistors extracted from parallel metal layer busses. In order to provide the most dramatic example possible with this design we first use the shortest paths tool to create an ordering on the pairs of bondpads from most to least problematic (from an ESD Protection prospective). Then we use the convex flow tool for increasing f-values in powers of 10 to get a more realistic assessment of the integrity of the protection scheme. In order to enforce that all input data are integral, the resistive costs (the majority of which were between 0 and 1 ohms) were scaled by 10^4 and then coerced to be integers. See Figure 24.

from	to	SP	$f = 100$	$f = 10^3$	$f = 10^4$
BPESDSH_3	BPESDSH_2	13.6137v	9.07485v	9.10738v	9.10572v
BPESDSH_2	BP_19	13.5447v	10.9408v	10.9546v	10.9581v
BPESDSH_2	BP_20	13.4844v	10.9239v	10.9434v	10.9471v
BPESDSH_4	BPESDSH_2	13.3537v	8.95845v	8.9853v	8.98386v
BPESD_10	BPESDSH_2	13.1206v	8.96288v	8.99333v	8.992v
BPESD_3	BPESDSH_2	12.894v	8.81708v	8.80018v	8.80309v
BPESD_9	BPESDSH_2	12.8844v	8.78777v	8.8183v	8.81684v
BPESDSH_2	BPESD_3	12.8671v	11.2804v	11.2768v	11.2788v
BPESD_8	BPESDSH_2	12.8428v	8.75754v	8.75781v	8.75771v
BPESD_7	BPESDSH_2	12.735v	8.77394v	8.75591v	8.75783v
Time (sec)		< 1	≈ 15	≈ 120	≈ 800

Figure 24: QUBiC D_Cell_Peripheral Excitation Level Table. Resistances were converted to integers by a scale factor of 10,000 (for convex flow). It took approximately 10 seconds to generate and sort the entire automated shortest paths portion of the table. $1.3A$ current was used, and all diodes were taken to have 0.7v drop.

Clearly, as f increases, the precision of the aVd increases by the same factor. However, as f increases, so does the amount of time we must wait for our results (by the same factor). We are noticing the effects of a pseudo-polynomial time algorithm as we require more and more precision. Of course, since we only scaled the resistances to 10^4 we can expect to get less accurate results for flows $f > 10^4$. If nothing else, from the

[25]The netlist(s) for this device is copywrited by *Philips Semiconductors*

shortest paths information in this table alone, it should be apparent that *BPESDSH_2* is a weak link in the protection scheme. In fact, it is in every pair!

4.7 Capacity Scaling

In this section we modify the Convex Cost Successive Shortest Paths algorithm to get a polynomial time algorithm. As noted in the table of Figure 18, the algorithm with Dijkstra's SSSP algorithm as a subroutine runs in some polynomial of the number of nodes and arcs, times the total flow U. As shown in the Examples (1-3) we get a more accurate ESD Budget when U is large, receiving greater precision for U values of greater powers (of ten). Therefore, as we require greater precision, the running time of our Convex SSP algorithm will take exponentially longer to finish.

The main reason for this is that the algorithm augments too little flow at each iteration. In particular, when we set $\delta = 1$ (in line 11 of Figure 17) the result is that only one unit of flow is augmented between the source and sink, thereby relaxing only one unit of U. Clearly, augmenting more flow each time through the *while* loop (of line 9) will cause the algorithm to satisfy U faster. However, this will lead to a less accurate result. Therefore, what we want is a way to linearize the convex cost function into segments of unit length, yet still be able to augment more than one unit of flow at each iteration. This can be accomplished by linearizing the convex cost function is several phases. That is, let δ start as a large integer. When there are no more δ-augmenting paths (paths whose arcs have capacity at least δ), decrease $\delta = \delta/2$, and repeat. Of course, since none of the forward arcs in our simulation of an ESD event have any capacity constraint, this procedure will never terminate. Since we want to eventually get down to $\delta = 1$, in order to linearize to convex cost by segments of unit length, we must impose some capacity constraint, even if it must be artificial. Therefore, let $u_{ij} = U$ for each forward arc $(i,j) \in A$. Note that if we were still augmenting only unit flow along paths this would impose no constraint on each arc since we never augment more than the remaining flow at the source, which is itself never more than U. This capacity constraint will guarantee that eventually $\delta = 1$, at which point we have essentially reduced to the original Convex SSP algorithm. Only now much of the original flow U has been satisfied. All that is left to do is augment the remaining flow, while still satisfying the reduced cost optimality conditions.

More precisely, let the first scaling phase begin at $\delta = 2^{\lfloor \log U \rfloor}$. Initialize the flow $x = 0$, and the potentials $\pi = 0$. At the beginning of each δ-scaling phase the algorithm must now maintain what we will call a δ-residual network $G(x,\delta)$. The forward arcs of $G(x,\delta)$ will be those arcs of $(i,j) \in A(G)$ such that $x_{ij} + \delta \leq u_{ij}$ where in which case we let

$$(i,j) \in A(G(x,\delta)) \qquad \text{be a forward arc such that}$$
$$cc_{ij} = (C(x_{ij} + \delta) - C(x_{ij}))/\delta$$
$$u_{ij} = \delta.$$

The reverse arcs will be those arcs (j,i) such that $(i,j) \in A(G)$ and that $x_{ij} \geq \delta$ where in which case we let

$$(j,i) \in A(G(x,\delta)) \qquad \text{be a reverse arc such that}$$
$$cc_{ji} = (C(x_{ij} - \delta) - C(x_{ij}))/\delta$$
$$u_{ji} = \delta.$$

Next, we must verify that each of the arcs of $G(x,\delta)$ satisfy the reduced cost optimality conditions. At first, this is trivial since $x = 0$ and $\pi = 0$, and $G(x,\delta) = G$. However, after the first δ-scaling phase, none of this is true. Furthermore, even though we have that after the previous (2δ)-scaling phase the reduced cost optimality conditions are satisfied, changing from 2δ to δ changes the convex costs cc, and therefore the reduced costs. In particular, the reduced cost of an arc might become negative. Still, after a simple increase or decrease of δ units of flow along such an incriminating arc (or its antagonist), both the arc and its antagonist will again satisfy the reduced cost optimality conditions. This can be done according to the following three cases for a forward arc (i,j) where either it, or its antagonist, violates the reduced cost

optimality conditions. The proof of this is left to the Appendix. For succinctness, let us say that an arc not in the δ-residue graph always has non-negative reduced cost.

1. when $c^{\pi}_{ij} < 0$ and $c^{\pi}_{ji} \geq 0$ increase x_{ij} by δ units of flow, and update the antagonist.
2. when $c^{\pi}_{ij} \geq 0$ and $c^{\pi}_{ji} < 0$ decrease x_{ij} by δ units of flow, and update the antagonist.
3. and $c^{\pi}_{ij} < 0$ and $c^{\pi}_{ji} < 0$ cannot happen.

Unfortunately, such an operation will cause the imbalances (b) of some nodes to violate the mass balance constraints. Therefore, we re-introduce the sets E and D as in (32) and (33) and fill them appropriately. We proceed as in the algorithm of Figure 15 in choosing s and t from E and D, terminating the *while* loop when one of E and D are empty. At this point we enter the next $(\delta/2)$ scaling phase. The optimality of the rest of this δ-scaling phase follows from Theorem 6. Inductively, since we end up eventually with $\delta = 1$, we have an optimal convex flow from linear segments of unit lengths, just as was produced by the algorithm in Figure 17.

The implementation of such a scaling scheme can typically be complicated, and messy. A good programmer will attempt to abstract the original procedure as much as possible, as well as the pre and post-processing to each of the scaling phases. While this is usually a good practice, there are many pitfalls involved in this approach. Some of these include

- Although we assumed that there is always an uncapacitated path between every source and sink node for the original SSP algorithm, we certainly wouldn't want this to be the case in any δ-scaling phase, except possibly the last one ($\delta = 1$).
- Iterating though pairs of elements of E and D can be complicated. There may be paths in the δ-residual network for some pairs and not for others. Therefore, there are actually two criteria for removing a node from either set: (1) its excess/deficit is less than δ, or (2) the excess/deficit is greater than δ, but there exist no augmenting paths between it and any element of the other set in $G(x, \delta)$.
- Satisfying the reduced cost optimality conditions at the beginning of each δ-scaling phase. This typically not only involves checking arcs against the cases enumerated above, but asserting that the resulting augmentations have actually satisfied them.
- There are now three graphs which we must maintain, either explicitly or implicitly: G, $G(x)$, $G(x, \delta)$.
- Any other assertions or pitfalls which are necessitated by the fact that we are now dealing with the implementation of a rather large program.

Therefore, it might be easier to altogether come up with a new procedure, and data structures to best accommodate these modifications.

Capacity Scaling SSP Complexity: As before, the complexity of this routine can be measured by how many augmentations are made along shortest paths overall. This requires that we know about how many augmentations are made at each scaling phase. This can be computed by noting that a previous (2δ)-scaling phase must have ended when one of E or D (both of which can contain at most n nodes) was emptied– after which the sum of the excesses can be at most $2n\delta$. At the beginning of the δ scaling phase, in order to satisfy the reduced cost optimality conditions, the flow on any arc (of which there are m) may require a positive augmentation of at most δ units of flow. Together, we conclude that at the beginning of any δ-scaling phase the total excess is at most $2(n+m)\cdot\delta$. Therefore, at each δ-scaling phase, there can be at most $O(\max\{n, m\})$ path-augmentations of δ units of flow until there no longer exist nodes with imbalance whose magnitude is greater than δ. Altogether there are $O(\log U)$ δ-scaling phases, which means that the algorithm performs $O(\max\{n, m\}\log U) \subseteq O(n^2 \log U)$ path-augmentations. This yields a running time of $O(SP(n, m, C)\cdot\max\{n, m\}\log U)$ depending on the running time SP of the shortest paths algorithm (and priority queue in the case of Dijkstra) we choose. Altogether, we have now shown that the ESD Budget Problem can be solved exactly in time $O(n^4 \log U)$ if we choose Dijkstra's SSSP algorithm with a Trivial Array priority queue, which is polynomial!

4.8 Further Considerations

Earlier we alluded to the possibility of implementing a more careful representation of electrical components. This topic can be addressed from several angles. Representing a multi-terminal device graphically is not difficult. However, enforcing that such a representation act like a transistor during a simulation is not trivial. There are extensions to the capacity constrains of MCMF Networks which allow lower bounds on flow to be placed on arcs (as well as upper bounds.) See [1]. Clearly, this is useful for modeling simple transistors. However, it is less clear how the equilibrium current provided by a solution the convex flow problem can be used here, or what appropriate weights for the arcs should be. Using any of the techniques discussed in this paper requires the use of an auxiliary network when arcs are given non-zero lower bounds.

Also worth noting (again mentioned previously) is that MCMF problems make nice linear programs (see [10] of [4]). [4] gives a nice description of how many network flow problems can be phrased as a linear program. The convex cost network flow problem described for finding equilibrium currents requires quadratic programming, which is in general much more difficult (see [10]). Nonetheless, the existence a tool which solves quadratic problems greatly reduces the number on implementation details required to solve the ESD Budget problem. It would be interesting to study the subset of quadratic programming problems which represent ESD Budget calculations. In addition, the shortest paths problem can also be phrased as a linear program. However, this is more complicated. This involves first reducing the SSSP problem to as Optimal Matching problem on an auxiliary bipartite network, which in turn has an associated Integer Linear Program. The general Integer Linear Programming problem is NP-hard (see [10]). Nonetheless, it turns out in this case that basic solutions from the simplex methods for such reductions from matchings problems are integral if any integral solutions exist.

A Appendix

A.1 Dijkstra SSSP Correctness

The correctness of Dijkstra's algorithm can be proved by induction[26] : Since at any point in the algorithm the node set N is partitioned by S and $\overline{S}$ representing permanently labeled and temporarily labeled nodes respectively, we should like to prove both of the following:

i. the distance labels of nodes in S are optimal

ii. for each $t \in S$, the cost of the shortest path $P_t = \langle s = v_1, v_2, \ldots, v_j = t \rangle$, from the original source s to t, is equal to $d(t)$ provided that $v_i \in S$, for $i \in \{1, \ldots, j\}$. That is, each internal node in the path lies in S.

Proof. by induction on $|S|$.

Base: $|S| = 1$. This means that $S = \{s\}$, therefore we want that $d(s) = 0$, and that the internal nodes P_s are all in S. The shortest path to s, since it is the source, is obviously zero and s is clearly the only node on the path.

Inductive Step: Assume $|S| > 1$ and that the distance label of nodes in S' are optimal, where $|S'| = |S| - 1$. Furthermore, assume that each internal node on the corresponding path is in S'. We want to show that:

i. The distance labels of nodes in S, created when Dijkstra's SSSP algorithm adds another node from $\overline{S}'$ to S', are optimal.

The algorithm proceeds by choosing the node v in the priority queue (in $\overline{S}'$) whose temporary distance label is smallest (line 5), and then removes it from the priority queue (thus removing it from $\overline{S}'$ and placing it in S' (which will then become S)). By the inductive hypothesis, the temporary distance label $d(v)$ corresponds to the shortest path P_v in which all of the internal nodes on the path belong to S'. Now suppose that $d(v)$ were not optimal. Then, there must exist a path $P'_v = \langle s = v'_1, v'_2, ..., v'_n = v \rangle$, such that $\sum_{i=1}^{n-1} c(v'_i, v'_{i+1}) < d(v)$, where at least one of v_i for $i \in \{1, 2..., n\}$ is not in S'. Therefore in $v_i \in \overline{S}'$. Let $u \in \overline{S}'$ be the first such node on this shortest path. The path P'_v can be decomposed into P_u, the current path to u found by the algorithm, and $P_{(u,v)}$, some short path from u to v. But then the temporary label $d(u) \geq d(v)$ since it was not the minimum from the priority queue *min* operation. That is, both u and v were in $\overline{S}'$, but v won the *min* operation, and therefore must have priority less than or equal to u. From this we have that the total cost of P_u is at best $d(v)$. Since we assumed that there were no negative arc-costs, the total cost of $P_{(u,v)}$ can be at best zero. But this contradicts that P'_v is optimal, since clearly P_v is better. We conclude that since the shortest path cannot contain any nodes from $\overline{S}'$, the distance label $d(v)$ must be optimal. Therefore the distance labels of nodes in $S = S' \cup \{v\}$ are optimal, as desired.

ii. The optimal distance labels of nodes $t \in S$, created when Dijkstra's Algorithm adds another node from $\overline{S}'$ to S', is equal to $d(t)$ provided that each internal node lies in S'.

We first observe that a parent $p(v)$ of a node $v \in N$ (line 11) is always an element of S' (line 6). Therefore, when Dijkstra's algorithm removes a node v from the priority queue, thus permanently labeling it (placing it in S), this may cause a decrease of the temporary labels of vertices $t \in \overline{S}$. After this update, the paths from the nodes t back to the source s, which are recovered by recursively traversing the parents $(p(t))$, satisfy $d(t) = d(p(t)) + c(p(t), t)$, by the inductive hypothesis. Therefore we have that that $d(t)$ is the distance of the shortest path from the original source s to $t \in \overline{S}$ provided that each internal node on the path is in S.

Therefore, by the principle of mathematical induction, we conclude that Dijkstra's SSSP algorithm is correct. □

[26]hypothesis from [1] page 110. The proof of these hypothesis are based on [1] and [6]

A.2 Dijkstra Complexity for Arbitrary Priority Queues

Now that we have discussed all of the details of Dijkstra's SSSP algorithm, and the abstract data types which it uses, we can talk generally about its time and space complexity. It is then straightforward to make an analysis with respect to a specific priority queue data structure.

The key to the runtime analysis of Dijkstra's SSSP algorithm, which is in general true of many *greedy* algorithms, is the number of times the contents of a loop are executed. The first key observation, in our case, is that the *foreach* loop (line 9) is executed at most once for every arc in the graph. Within this loop there is a *decrease key* operation. In a dense graph, this operation might potentially be executed each time through the loop. The second key observation is that each node in the graph is inserted into the priority queue data structure (line 5) before anything interesting happens. Before the each execution of the *while* expression (line 8) a node is removed from the priority queue (lines 6-7 and 16-17) and we choose the next node by a *min* priority queue operation.

Before choosing a priority queue implementation we can formulate the runtime of Dijkstra's SSSP algorithm by abstracting the runtime of the priority queue operations. If we let:

$$\begin{aligned} PQ_{init} &= \text{pq } \textit{initialization} \text{ running time} \\ PQ_{ins} &= \text{pq } \textit{insert} \text{ running time} \\ PQ_{min} &= \text{pq } \textit{extract min} \text{ running time} \\ PQ_{dec} &= \text{pq } \textit{decrease key} \text{ running time} \end{aligned}$$

we can make the following general analysis. Up to the choice of the priority queue, Dijkstra's SSSP algorithm runs in:

$$\Theta(PQ_{init} + n * PQ_{ins} + n * PQ_{min} + m * PQ_{dec}) \tag{48}$$

on graph $G = (N, A)$, where $n = |N|$ (number of nodes), and $m = |A|$ (number of arcs).

If we use a *Trivial Array* as our priority queue data structure we get the following (infamous) short-answer to how fast Dijkstra's algorithm runs. It turns out that this is the best that we can do, using any of the priority queues mentioned, for completely dense networks. The *decrease key* and *initialize* operations are in time $O(1)$ for this implementation, and initialization is in time $O(n)$, while all of the *extract min* operations run in no better than time

$$\sum_{i=1}^{n} i = n + (n-1) + \ldots + 1 = \frac{n(n+1)}{2} \in O(n^2). \tag{49}$$

which accounts for the size of the legal elements in the priority queue decreasing each time through the *while* loop (line 17). Since $m = |A(G)|$ (number of arcs in G) is less than or equal to the number of arcs in a *complete directed graph* with n nodes ($n(n-1)/2$), we have $m \leq n(n-1)/2 \in O(n^2)$, and we conclude that Dijkstra's Algorithm using the *Trivial Array* priority queue data structure runs in $O(n^2)$.

A.3 Capacity Scaling: δ-Scaling Phase Arc Augmentations

We continue here, where we left off in the Capacity Scaling section, and offer a proof that we can ensure that the reduced cost optimality conditions on all of the arcs of any $G(\delta, x)$ are satisfied before any path augmentations are made during the δ-scaling phase. As already noted, this is true for our base case, $\delta = 2^{\lfloor \log U \rfloor}$, and so we are left only with showing that as long as the reduced cost optimality conditions are satisfied after the (2δ)-scaling phase, they are also satisfied before the path augmentations of the δ-scaling phase– although we may have to augment, or decrease, the flow on some arcs in order for this to be true. Recall that for any arc (i, j) or $\overline{(i, j)}$ which did not satisfy the reduced cost optimality conditions, there were three cases with respect to the forward arc (i, j).

1. when $c^\pi_{ij} < 0$ and $c^\pi_{ji} \geq 0$

   Here, we want to show that increasing the flow on (i, j) by δ that

   A. (i, j) satisfies the reduced cost optimality conditions, namely that $c^\pi_{ij} = cc_{ij} - \pi[i] + \pi[j] = (C(x_{ij} + 2\delta) - C(x_{ij} + \delta))/\delta - \pi[i] + \pi[j] \geq 0$, and

   B. $\overline{(i, j)} = (j, i)$ satisfies the reduced cost optimality conditions, namely that $c^\pi_{ji} = cc_{ji} - \pi[j] + \pi[i] = (C(x_{ij}) - C(x_{ij} + \delta))/\delta - \pi[j] + \pi[i] \geq 0$.

   Since the reduced cost optimality conditions were satisfied at the end of the (2δ)-scaling phase we have that

$$0 \leq (C(x_{ij} + 2\delta) - C(x_{ij}))/(2\delta) - \pi[i] + \pi[j] \quad (50)$$
$$0 = C(x_{ij} + 2\delta) - C(x_{ij}) - 2\delta\pi[i] + 2\delta\pi[j] \quad (51)$$

   and from our assumption this produces

$$0 > c^\pi_{ij} \quad (52)$$
$$= (C(x_{ij} + \delta) - C(x_{ij}))/\delta - \pi[i] + \pi[j] \quad (53)$$
$$0 = C(x_{ij} + \delta) - C(x_{ij}) - \delta\pi[i] + \delta\pi[j]. \quad (54)$$

   Therefore, from (51) and (54) we can write

$$C(x_{ij} + 2\delta) - C(x_{ij}) - 2\delta\pi[i] + 2\delta\pi[j] \geq C(x_{ij} + \delta) - C(x_{ij}) - \delta\pi[i] + \delta\pi[j]$$

   which implies that (moving everything onto one side of the inequality)

$$\begin{aligned} 0 &\leq C(x_{ij} + 2\delta) - C(x_{ij} + \delta) - \delta\pi[i] + \delta\pi[j] \\ &= (C(x_{ij} + 2\delta) - C(x_{ij} + \delta))/\delta - \pi[i] + \pi[j] \end{aligned}$$

   which satisfies A. Now, note that at (54) we essentially have that

$$\begin{aligned} -(C(x_{ij} + \delta) - C(x_{ij}) - \delta\pi[i] + \delta\pi[j]) &> 0 \\ C(x_{ij}) - C(x_{ij} + \delta) - \delta\pi[i] + \delta\pi[j] &\geq 0 \\ (C(x_{ij}) - C(x_{ij} + \delta))/\delta - \pi[j] + \pi[i] &\geq 0 \end{aligned}$$

   thus showing B. □

2. when $c^\pi_{ij} \geq 0$ and $c^\pi_{ji} < 0$ decrease x_{ij} by δ units of flow, and update the antagonist.

   This follows practically verbatim from Case 1, dealing instead with the antagonist arc $\overline{(i, j)} = (j, i)$, and decreasing the flow by δ rather than increasing it. □

3. both $c^\pi_{ij} < 0$ and $c^\pi_{ji} < 0$ cannot happen for convex cost functions.

   To see this, we need a formal definition of a *convex* function[27], namely: A real-valued function f defined in some interval (a, b) is said to be convex if

$$f(\lambda x + (1 - \lambda)y) \leq \lambda f(x) + (1 - \lambda)f(y) \quad (55)$$

   whenever $a < x < b$, $a < y < b$, and $0 < \lambda < 1$. In particular, for any flow x_{ij}, and our convex function $C(x_{ij})$, we can take $a = 0$ and $b = u_{ij}$ (which are of little significance here). We have then, by our convexity assumption

$$C((x_{ij} + \delta)\lambda + (1 - \lambda)(x_{ij} - \delta) \leq \lambda C(x_{ij} + \delta) + (1 - \lambda)C(x_{ij} - \delta).$$

[27] taken from p.101 of [11]

Suppose that we let $\lambda = 1/2$, and then rewrite the above equation:

$$\begin{aligned} C(\frac{1}{2}(x_{ij}+\delta)+\frac{1}{2}(x_{ij}-\delta) &\leq \frac{1}{2}C(x_{ij}+\delta)+\frac{1}{2}C(x_{ij}-\delta) && (56) \\ C(x_{ij}) &\leq \frac{1}{2}C(x_{ij}+\delta)+\frac{1}{2}C(x_{ij}-\delta) && (57) \\ 2\cdot C(x_{ij}) &\leq C(x_{ij}+\delta)+C(x_{ij}-\delta). && (58) \end{aligned}$$

Now let us further suppose (for the sake of contradiction) that it is possible that both $c_{ij}^{\pi} < 0$ and $c_{ji}^{\pi} < 0$. As in Case 1 we have that

$$\begin{aligned} C(x_{ij}+\delta)-C(x_{ij})-\delta\pi[i]+\delta\pi[j] &< 0 \\ C(x_{ij}-\delta)-C(x_{ij})-\delta\pi[i]+\delta\pi[j] &< 0 \end{aligned}$$

Taking the sum of the above two equations we get

$$\begin{aligned} C(x_{ij}+\delta)+C(x_{ij}-\delta)-2\cdot C(x_{ij}) &< 0 \\ C(x_{ij}+\delta)+C(x_{ij}-\delta) &< 2\cdot C(x_{ij}) \end{aligned}$$

which contradicts (58). Therefore, or assumption that both c_{ij}^{π} and c_{ji}^{π} could both be less than zero must have been false. □

References

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Addison Wesley, 1993.

[2] Sara Baase and Allen van Gelder. *Computer Algorithms.* Addison Wesley, 3rd edition, 2000.

[3] M. Baird and R. Ida. Verify ESD: A tool for efficient circuit level ESD simulations of mixed signal ICs. *EOS/ESD ESD Symposium*, 2000.

[4] Vašek Chvátal. *Linear Programming.* W.H. Freeman and Company, 15th edition, 2000.

[5] John Clark and Derek Alan Holton. *Graph Theory.* World Scientific, 2nd edition, 1998.

[6] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* McGraw Hill, 21st edition, 1998.

[7] Johnson D.B. A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Mathematical Systems Theory*, 1982.

[8] Bruce M. Fleisher. *Electrical Circuits Review.* Some Company, 3000.

[9] Jeffrey H. Kingston. *Algorithms and Data Structures: Design, Correctness, Analysis.* Addison Wesley, 1st edition, 1990.

[10] Christos H. Papadimitriou and Kenneth Steiglitz. *Combinatroial Optimization: Algorithms and Complexity.* Dover, Dover edition, 1998.

[11] Walter Rudin. *Principles of Mathematical Analysis.* McGraw Hill, 3rd edition, 1964.

List of Figures

1 Graph Example 6
2 Graphical Resistor 6
3 Graphical Diode 7
4 Graphical Connection 7
5 Back to Back Diodes 8
6 Dijkstra's SSSP Algorithm 12
7 Shortest Paths Example 13
8 Priority Queue Complexities 14
9 Floyd-Warshall APSP Algorithm 17
10 Parents for Floyd-Warshall APSP 17
11 Network Example 22
12 Flow Example 23
13 Residue Graph Example 24
14 Reduced Costs Example 28
15 SSP Algorithm 31
16 SSP Runtime 33
17 Convex Flow ESD Budget SSP Algorithm 37
18 Convex ESD Budget SSP Runtime 37
19 Adjusted Voltage Drop Algorithm 38
20 Parallel Resistors 40
21 Parallel Resistor ESD Budget Table 40
22 Resistor Cube 40
23 Resistor Cube ESD Budget Table 41
24 QUBiC D_Cell_Peripheral Excitation Level Table 41

Contents

1 Overview **2**

2 Preprocessing & Electrical Components **4**
2.1 Preprocessing 4
2.2 Graphing, Representation of Electrical Components 5

3 Overestimating & Shortest Paths **9**
3.1 Overestimating 9
3.2 Single Source Shortest Paths (Dijkstra) 11
3.3 All Pairs Shortest Paths (Floyd-Warshall) 15
3.4 Graph-size Reductions, Optimizations & Improving Overestimates 18

4 Convex Flows **20**
4.1 Calling All Paths 20
4.2 Networks, Minimum Cost Maximum Flow & Residue Graphs 20
4.3 Principles and Conditions of Optimality 24
4.4 Successive Shortest Paths Algorithm 28
4.5 Convex Flows 34
4.6 Some Results & Integrality of Data 39
4.7 Capacity Scaling 42
4.8 Further Considerations 44

A Appendix **45**
A.1 Dijkstra SSSP Correctness 45
A.2 Dijkstra Complexity for Arbitrary Priority Queues 46
A.3 Capacity Scaling: δ-Scaling Phase Arc Augmentations 46

Thanks: Foremost, I would like to thank my mentor and advisor at UCSC, Dave Helmbold. I continue to appreciate your patience, constructive criticism, confidence in my work, and especially all of your coaching on this project. I would also like to thank UCSC in general for being a fantastic institution, and the Jack Baskin School of Engineering faculty and staff for their encouragement and hospitality. I would also like to thank Philips Semiconductors for providing me with an interesting problem, an introduction to electrical engineering which made this work more tangible, for the time, and especially for money. This project would not have been possible without Dave Perasso and Ching Kwok Wong at Philips. Thanks for having faith in me! In addition I would like to give thanks to Paul Ngan, Christine Dufour, Axel Schurr, Menno Clerk, and Theo Smedes at Philips for all of the helpful technical and implementational feedback they have given. Although this research has a major focus of my last few years, much of my progress would not have been possible without the above people's contributions.