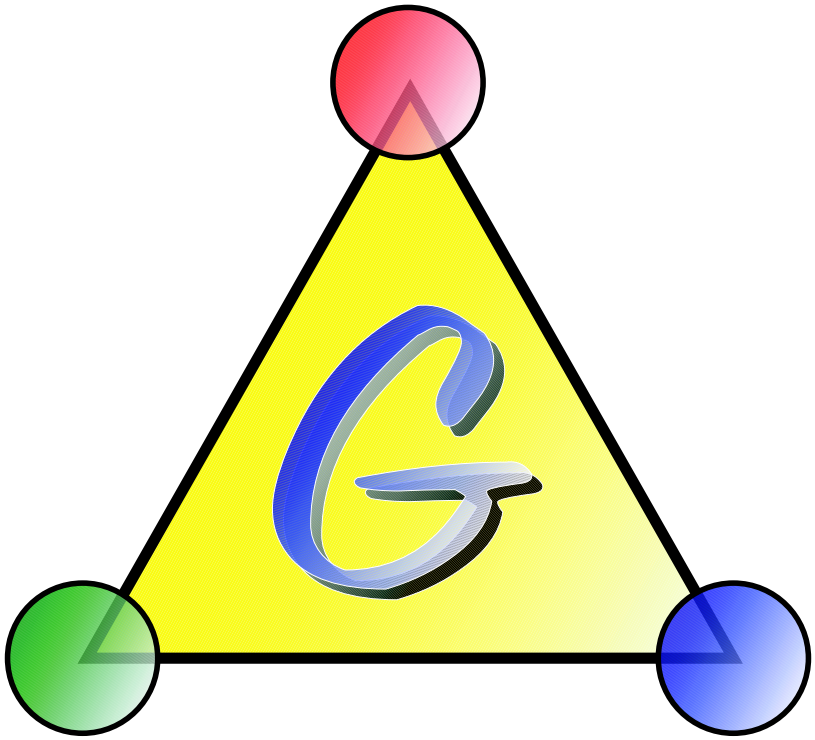
GIDEN

a graphical environment for network optimization



VERSION J–2.0β
JUNE 26, 2003

Collette R. Coullard
David S. Dilworth
Jonathan H. Owen

About the Authors

Collette R. Coullard is an Associate Professor and a Charles Deering McCormick Professor of Teaching Excellence at Northwestern University. Her research focuses on modeling, solution techniques, and the fundamental mathematics of combinatorial optimization. She obtained her Ph.D. in 1985 from Northwestern University; she has taught at Purdue University and the University of Waterloo, and she was a Humboldt Fellow at the University of Bonn.

David S. Dilworth is the principal of Systems Research, a small business that develops advanced computing systems for industry and academia. His research area is electronic holography for imaging through scattering media, and he holds a patent for that work. He obtained his Ph.D. in 1989 from the University of Michigan where he is currently an Adjunct Associate Professor in Electrical Engineering and Computer Science.

Jonathan H. Owen is a research engineer in the Enterprise Systems Lab at General Motors. In 1998 he received his Ph.D. from the Industrial Engineering and Management Sciences Department at Northwestern University, where he maintains a part-time position as an assistant research professor. His primary research focuses on solving mixed-integer linear programs with general-integer variables. Additional research has dealt with solving specialized binary integer programs using branch-and-cut methodology and the development of GIDEN as a teaching and research tool for network optimization.

Resources

The latest information on the GIDEN environment is available through the web site, which is located at the following url:

`http://giden.nwu.edu/`

GIDEN is under ongoing development. If you have feedback on the environment or any of the associated documentation, please contact us via email at `giden@iems.nwu.edu` or through the web site.

Acknowledgments

The developers gratefully acknowledge essential support for the GIDEN project from the following organizations:

Office of Naval Research	`http://www.onr.navy.mil/`
Sun Microsystems, Inc.	`http://www.sun.com/`
National Science Foundation	`http://www.nsf.gov/`
Optimization Technology Center	`http://www.mcs.anl.gov/otc/`
Committee on Institutional Cooperation	`http://NTX2.cso.uiuc.edu/cic/`
Zero G Software, Inc	`http://www.zerog.com/`

The authors would also like to express their appreciation to those individuals who have provided comments and suggestions that have resulted in the improvement of GIDEN, including undergraduate and graduate students at Northwestern University who have taken IEMS courses C-13, D-50, and D-52. We are particularly grateful to the following individuals: Robert Fourer, Peh Ng, Susan Owen, and Donald Wagner.

Special thanks are offered to Susan Owen for her many contributions to the GIDEN project.

Contents

Introduction

Let $G = (N, A)$ be a graph with node set N and edge set A. Given information I about the nodes and edges of G, we call $\mathcal{N} = (G, I)$ a *network*. Networks are useful mathematical modeling constructs for representing a variety of physical and abstract systems. Given some system that can be modeled as a network $\mathcal{N}$, we may ask questions about what properties characterize the system. These questions can be translated into problems that we can solve using the network model $\mathcal{N}$. Network optimization deals with constructing a network to model an underlying system, describing in terms of the network model the properties of interest, solving the network problem using exact or heuristic techniques, and using the model solution to develop a better understanding of the real-world system. Examples of network optimization problems include shortest path problems, minimum spanning tree problems, flow problems, and matching problems. Two recent textbooks on network optimization are [1] and [3].

GIDEN is a visually oriented interactive software environment for network optimization. There are many accounts of situations where visual representations have been used effectively to communicate ideas when verbal and textual representations alone have been less successful. For network optimization, the use of *visualization* is natural because of the convenient graphical representation of nodes as points and edges as lines between pairs of nodes. (For a nice reference on visualization and optimization, see [5].) The fundamental purpose of GIDEN is to facilitate the visualization of network optimization problems, solutions, and solution algorithms. Features of GIDEN include the following:

- a graphical interface for building and modifying networks
- facilities for algorithm animation
- an expandable toolkit of animated solution algorithm implementations ("solvers")
- an extensive library of network-related data structures
- platform independence

The features listed above make GIDEN an attractive tool for a variety of users. The most obvious users in the context of viewing algorithms are in the academic community. Teachers of network optimization can use the graphical animation tool in the classroom to demonstrate network algorithms. Algorithms are learned through various methods, depending on needs and learning styles. Typically at some stage in the learning process the steps of the algorithm are performed on an example. An effective example illustrates the steps of the procedure, including termination, and also provides some convincing evidence of the correctness of the final solution. A graphical animation tool enables users to make this stage of the learning process as extensive as they want. They can try the algorithm on a number of examples or on a specific example a number of times. They can test their knowledge of the algorithm by predicting its behavior. Students learning network optimization can use the tool outside of class to solidify their knowledge, both through independent endeavors and

through specific homework assignments. GIDEN has been well-received when used as a teaching and learning aid in graduate and undergraduate classes at Northwestern University.

GIDEN is also a useful implementation environment for network optimization algorithms. Users can quickly implement network optimization algorithms using the network data structures class library, which provides abstract data types and containers that are appropriate for network programming (e.g., `Node`, `Edge`, `Network`, `List`, `PriorityQueue`). Algorithm animation assists in implementation debugging as well as identification of bottlenecks. A pseudocode window may optionally be created with each algorithm to assist in tracing algorithm execution.

Chapter 1

Preliminaries: Network Optimization and GIDEN

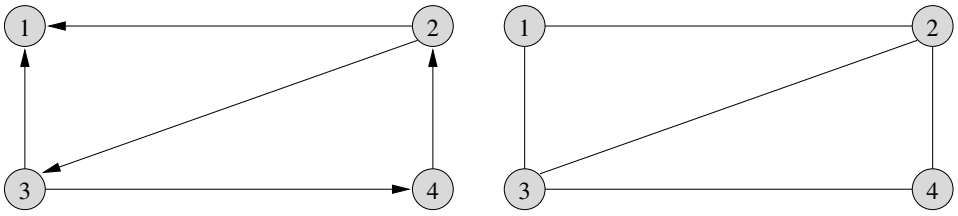
1.1 Notation and Definitions

In this section, we define notation and elementary network concepts that will be used throughout this manual. For a thorough introduction to networks, see [1], [2], [3], and [6].

graph: A *graph* $G = (N, A)$ consists of a set N of *nodes* (or *vertices*) and a set A of *edges* (or *arcs*).

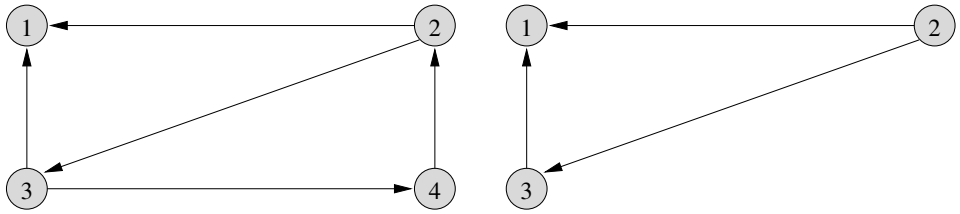
directed and undirected edges: An edge $e = (i, j)$ is a pairing of nodes i and j; we call node i and node j the *ends* of edge e. An edge e is *directed* if it is an ordered pairing of nodes, i.e., $e = (i, j) \neq (j, i)$; if an edge $e = (i, j)$ is a directed edge, we call i the *tail node* (or *tail*) of edge e, and we call j the *head node* (or *head*) of edge e. If an edge e is an unordered pairing of nodes, i.e., $e = (i, j) = (j, i)$, then we call the edge *undirected*. A directed edge is also called an *arc*.

directed and undirected graphs: Graph G is a *directed graph* (or *digraph*) if edges $e \in A$ are directed. G is an *undirected graph* if edges $e \in A$ are undirected. For the sake of simplicity, we assume that edges in a graph are either all directed or all undirected. The illustration below, shows a graph G with node set $N = \{1, 2, 3, 4\}$ and edge set $A = \{(2, 1), (2, 3), (3, 1), (3, 4), (4, 2)\}$.



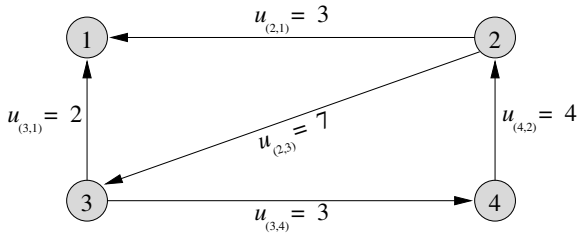
Directed graph $G = (N, A)$. Undirected graph $G = (N, A)$.

subgraph: A graph $\bar{G} = (\bar{N}, \bar{A})$ is a *subgraph* of $G = (N, A)$ if $\bar{N} \subseteq N$ and $\bar{A} \subseteq A$. In the illustration below we show a directed graph $G = (N, A)$ (left) and a subgraph $\bar{G} = (\bar{N}, \bar{A})$, where $\bar{N} = \{1, 2, 3\}$ and $\bar{A} = \{(2, 1), (2, 3), (3, 1)\}$ (right).



Given $E \subseteq A$, the subgraph *induced by* E is denoted $G(E) = (N(E), E)$, where $N(E)$ is the set of end nodes of members of E. Where no confusion arises, we sometimes use E to refer to this subgraph $G(E)$.

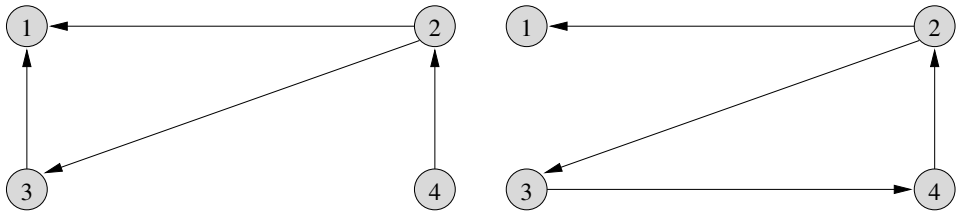
network: Given some information I about the nodes and/or edges of a graph G, we call $\mathcal{N} = (G, I)$ a *network*. If the edge set of G is directed, we call $\mathcal{N}$ a *directed network*; otherwise, the edge set of G is undirected and we call $\mathcal{N}$ an *undirected network*. The following illustration shows a network $\mathcal{N} = (G, u)$ where G is the directed graph from above and $u = \{u_e \in \mathbb{Z} : e \in A\}$ is some information on the edges A.



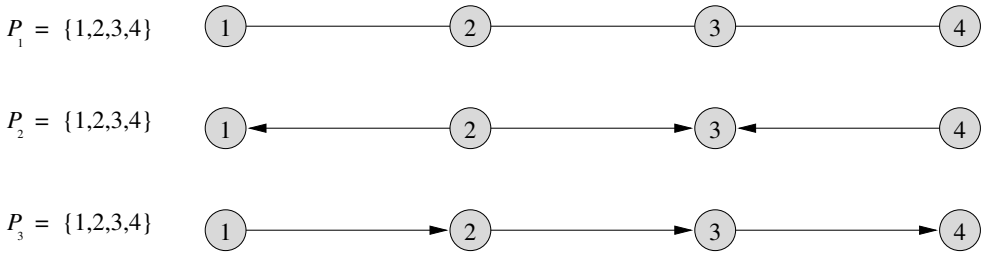
walk: A *walk* W of G is a sequence of nodes and edges

$$W = \{i_1, e_1, i_2, e_2, i_3, \ldots, i_{r-1}, e_{r-1}, i_r\}$$

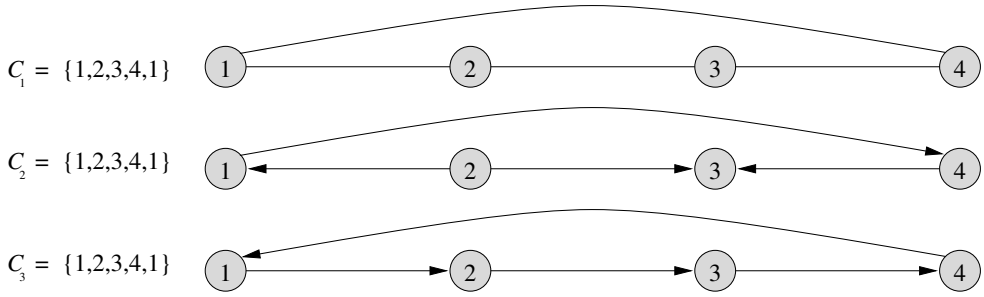
such that $e_k = (i_k, i_{k+1})$ or $e_k = (i_{k+1}, i_k)$ for all $k \in \{1, 2, \ldots, r-1\}$. The term *walk* may also be used to refer to a sequence of either nodes or edges (e.g., the walk W can be also be written as $\{i_1, i_2, \ldots, i_r\}$ or $\{e_1, e_2, \ldots, e_{r-1}\}$). If G is directed, then W is called a *directed walk* if the edge $(i_k, i_{k+1}) \in A$ for any two consecutive nodes in the walk, i_k and i_{k+1}. Pictured below are two walks: walk $W_1 = \{2, 3, 1, 2, 4\}$ (left) and directed walk $W_2 = \{2, 3, 4, 2, 1\}$ (right).



path: A *path* P is a walk with no repeated nodes. A *directed path* (or *dipath*) P is a directed walk with no repeated nodes. We call a path P an *s-t path* (*s-t dipath*) if it begins at node s and ends at node t. The following illustration shows three paths; only the bottom path, P_3, is a directed path.

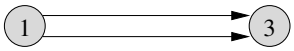


cycle: A *cycle* (or *circuit*) C is a path $\{i_1, e_1, i_2, e_2, \ldots, i_{r-1}, e_{r-1}, i_r\}$ along with the edge (i_1, i_r) or the edge (i_r, i_1). A *directed cycle* (or *directed circuit*) is a directed path followed by the directed edge (i_r, i_1). The illustration below shows three cycles; only the bottom cycle, C_3 is a directed cycle.



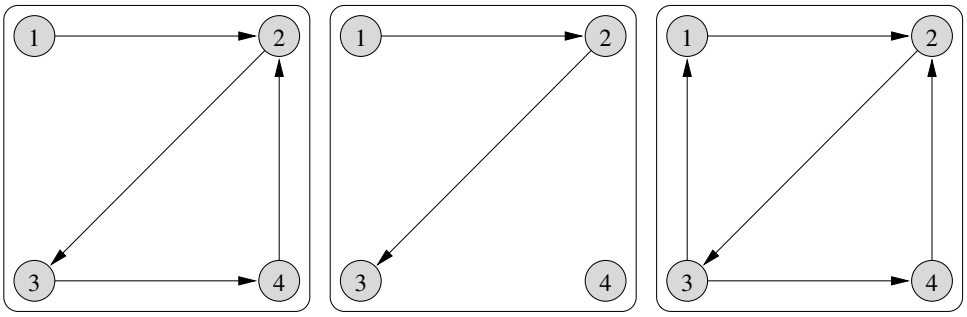
acyclic: We say a graph is *acyclic* if it does not contain a cycle. Similarly, a network is called acyclic if its associated graph is acyclic.

multiedge: Duplicate edges in A are called *multiedges* (or *parallel edges*). The following illustration shows a multiedge between node 1 and node 3.

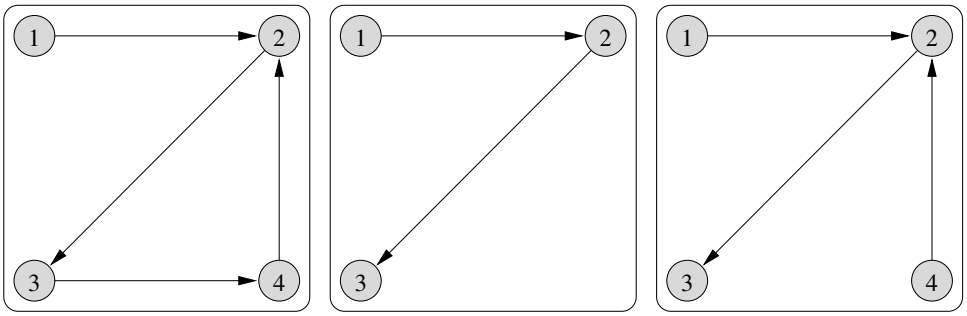


loop: An edge whose endpoints are the same node (i.e., $e = (i, i)$ for some $i \in N$) is called a *loop*. The following illustration shows a loop on node 3.

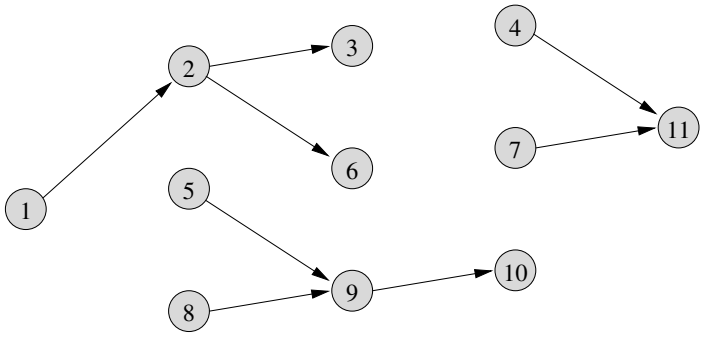
connectedness: In a graph $G = (N, A)$, two nodes $i \in N$ and $j \in N$ are called *connected* if there exists at least one i-j path in G. A *connected graph* is a graph $G = (N, A)$ where every pair of nodes in N is connected. The maximal connected subgraphs of G are called the *components* of G. If there exists a pair of nodes $i, j \in N$ such that there is no path between node i and node j, then we call G a *disconnected graph*. We say that a digraph G is *strongly connected* if for each pair of nodes, $i, j \in N$, there exists both an i-j dipath and a j-i dipath in G. In the following illustration, we show a connected graph (left), a disconnected graph (middle), and a strongly connected graph (right).



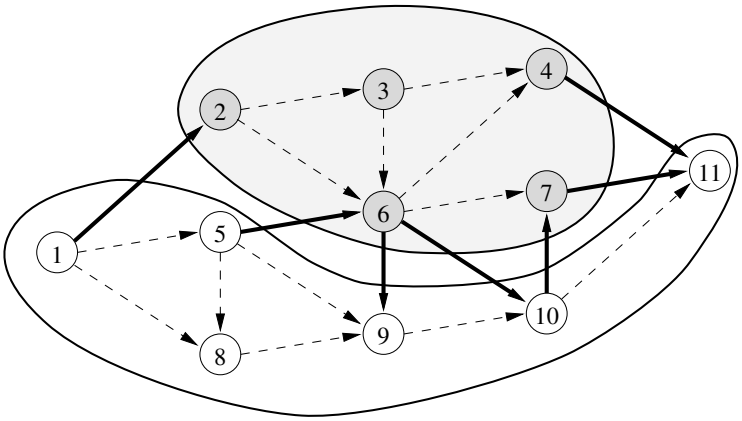
tree: A *tree* is an acyclic connected graph. A *spanning tree* of graph G is a tree T that is a subgraph of G containing all nodes of G. In the following illustration, we show a graph G (left), a tree that is a subgraph of G (middle), and a tree that is a spanning tree of G (right).



forest: A *forest* is an acyclic graph; it follows that each connected component of a forest is a tree. The following illustration shows a forest with three components, $F = \{T_1, T_2, T_3\}$, where $T_1 = \{(1,2),(2,3),(2,6)\}$, $T_2 = \{(4,11),(7,11)\}$, and $T_3 = \{(5,9),(8,9),(9,10)\}$.

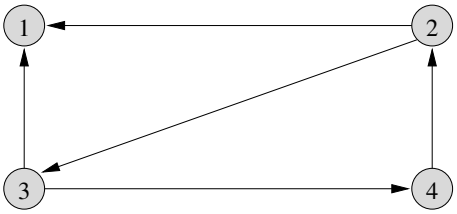


cut: Given a partition of the node set N into two subsets N_1 and $N_2 = N \backslash N_1$, the collection of edges (i,j) such that $i \in N_1$ and $j \in N_2$ is called a *cut* (or *cutset*), denoted $\delta(N_1, N_2)$. For two specified nodes, $s, t \in N$, an *s-t cut* is any cut $\delta(N_1, N_2)$ with the added property that $s \in N_1$ and $t \in N_2$. In the following illustration, shaded nodes belong to N_1 and non-shaded nodes belong to N_2; the edges drawn as solid arrows are in the cut $\delta(N_1, N_2)$.



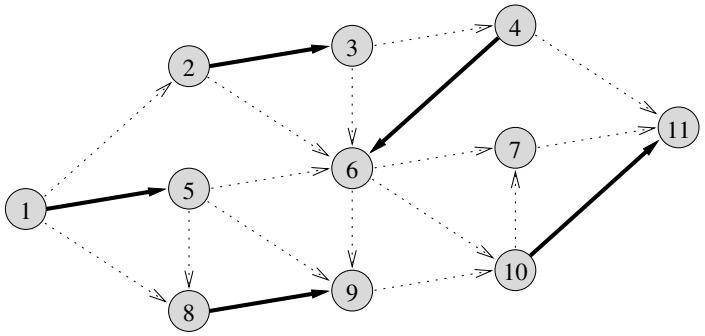
$N_1 = \{2,3,4,6,7\}$, $N_2 = \{1,5,8,9,10,11\}$,
$\delta(N_1, N_2) = \{(1,2),(4,11),(5,6),(6,9),(6,10),(7,11),(10,7)\}$

adjacency: We say that node i and node j are *adjacent nodes* if edge $(i,j) \in A$ or edge $(j,i) \in A$. For a node $i \in N$, we say that any edge $(i,j) \in A$ or $(j,i) \in A$ is an *incident edge* of node i. The collection of edges that are incident to a node $i \in N$ is called the *adjacency list* of node i; we denote the adjacency list of node i as $A(i)$. The following illustration shows a graph and gives the adjacency list for each node.



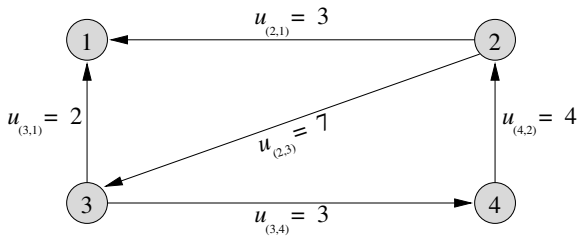
$A(1) = \{(2,1),(3,1)\}$, $A(2) = \{(2,1),(2,3),(4,2)\}$
$A(3) = \{(2,3),(3,1),(3,4)\}$, $A(4) = \{(3,4),(4,2)\}$

matching: A *matching* of a graph G is a subset $M \subseteq A$ such that each node of G has at most one incident edge in M. An example of a matching is shown as the bold edges in the following illustration; in this example, the matching is given as $M = \{(1,5),(2,3),(4,6),(8,9),(10,11)\}$.



Definition: Given a finite set E and a field $\mathbb{F}$, $\mathbb{F}^E$ denotes the $|E|$-dimensional vector space with components indexed on the members of E. For example, $u \in \mathbb{Z}^A$ means $\{u_e \in \mathbb{Z} : e \in A\}$.

capacitated network: A *capacitated directed network* is a directed network $\mathcal{N} = (G,u)$ where information $u \in \mathbb{Z}^A$ consists of integer-valued edge capacities for each edge $e \in A$.

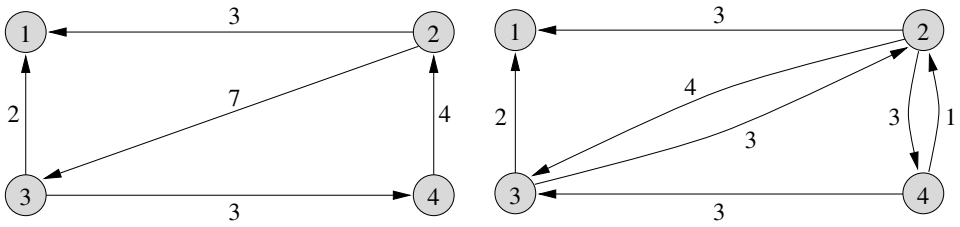


flow: Given a capacitated directed network $\mathcal{N} = (G,u)$, a *flow* is a vector $x \in \mathbb{R}^A$ that satisfies *flow convervation* at each of the nodes of the network, that is $\sum_{(i,j)\in E} x_{(i,j)} = \sum_{(j,i)\in E} x_{(j,i)}$ for each node $j \in N$. A *feasible flow* also satisfies the capacity (and lower bound) restrictions on the edges, that is $0 \le x_e \le u_e$ for each edge $e \in A$.

residual network: Given a capacitated directed network $\mathcal{N} = (G,u)$ and a flow $x \in \mathbb{R}^A$, the associated *residual network* is the network $\mathcal{N}^x = (G^x, u^x)$. The graph $G^x = (N, A^x)$ has node set N and edge set A^x, where for each edge $(i,j) \in A$, the edge $e = (i,j)$ is in A^x if $x_e < u_e$ and the edge $e = (j,i)$ is in A^x if $x_e > 0$. The capacities of the residual network, u^x, are defined as follows:

$$u^x_{(i,j)} = \begin{cases} u_{(i,j)} - x_{(i,j)} & \text{for } (i,j) \in A \\ x_{(j,i)} & \text{for } (j,i) \in A \end{cases}$$

The illustration below shows a network $\mathcal{N}$ (left) and the associated residual network $\mathcal{N}^x$ (right) for flow $x = \{x_{(2,1)}, x_{(2,3)}, x_{(3,1)}, x_{(3,4)}, x_{(4,2)}\} = \{0,3,0,3,3\}$. Edge capacities for each network (u or u^x) are given as numeric labels on the edges.


1
2
3
4
3
2
7
4
3
1
2
3
4
3
4
3
3
3

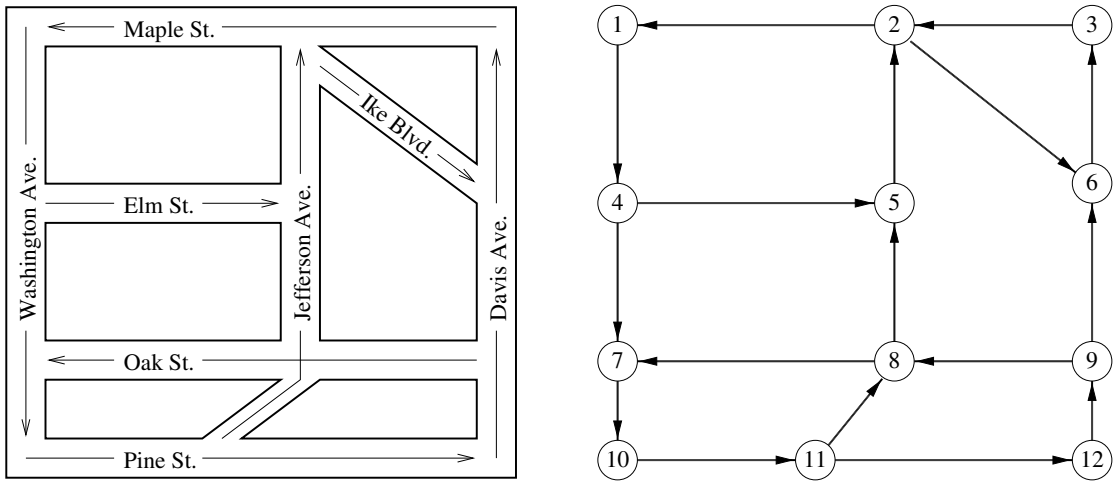
1.2 An Example of Network Optimization

In this section we demonstrate network optimization by describing how it can be used to analyze a system of interconnected one-way streets: We first build a network model of our system. We then pose a question about the system and formulate a corresponding network optimization problem. We then demonstrate a process for solving the problem, using a network optimization algorithm. Finally, we show the solution of the problem on our network and translate the solution into an answer to our original question regarding the underlying system.

1.2.1 Constructing a Network Model

As mentioned in the introduction, networks are often used to represent a physical or logical system. As an example, let's consider a system of interconnected one-way streets. In order to represent this system as a graph, let each intersection of two or more streets be represented by a node $i \in N$. Given a pair of intersections $i, j \in N$, if you can travel along a street from i to j without encountering any other intersection, let the street segment between i and j be represented as a directed edge $e = (i, j) \in A$. The system of roads is then represented by the directed graph $G = (N, A)$.

It is natural to consider visual representations of systems that can be modeled using graphs and networks, with nodes depicted as points and edges as lines between pairs of nodes. Suppose that part (a) of Figure 1.1 shows our example system of streets. In this system we have eight streets



(a) A system of interconnected one-way streets.

(b) Representation of street system as a directed graph.

Figure 1.1: System of streets and corresponding network model.

and twelve intersections. We can represent this system as a directed graph with twelve nodes — each representing an intersection — and seventeen edges that collectively represent the eight streets. In particular, we can represent the system pictured in Figure 1.1a as a directed graph $G = (N, A)$ where

$$N = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$$

and

$$A = \{(1,4), (2,1), (2,6), (3,2), (4,5), (4,7), (5,2), (6,3), (7,10), (8,5), (8,7), (9,6), (9,8), (10,11), (11,8), (11,12), (12,9)\}.$$

Part (b) of Figure 1.1 shows a representation of this directed graph.

Suppose we also have values l associated with each arc in A that represent the length of each street segment. In this case, the system of roads and associated street lengths are represented by the (directed) network $\mathcal{N} = (G, l)$. For the example system illustrated in Figure 1.1a, lengths of the street segments are given in Table 1.1. In the table, street segments are represented as an ordered

Table 1.1: Lengths for street segments between adjacent intersections.

"Tail" Intersection	"Head" Intersection	Length
Maple/Washington	Elm/Washington	30 units
Maple/Jefferson/Ike	Maple/Washington	50 units
Maple/Jefferson/Ike	Davis/Ike	47 units
Maple/Davis	Maple/Jefferson/Ike	40 units
Elm/Washington	Elm/Jefferson	50 units
Elm/Washington	Oak/Washington	20 units
Elm/Jefferson	Maple/Jefferson/Ike	30 units
Davis/Ike	Maple/Davis	25 units
Oak/Washington	Pine/Washington	15 units
Oak/Jefferson	Elm/Jefferson	20 units
Oak/Jefferson	Oak/Washington	50 units
Oak/Davis	Davis/Ike	25 units
Oak/Davis	Oak/Jefferson	40 units
Pine/Washington	Pine/Jefferson	40 units
Pine/Jefferson	Oak/Jefferson	18 units
Pine/Jefferson	Pine/Davis	50 units
Pine/Davis	Oak/Davis	15 units

pair of intersections such that we can drive from the *tail intersection* to the *head intersection* while obeying the one-way restrictions. The *lengths* listed give the distances between the tail and head intersections along the corresponding street segments. Figure 1.2 shows a network that represents our resulting system (complete with lengths l_e for each $e \in A$).

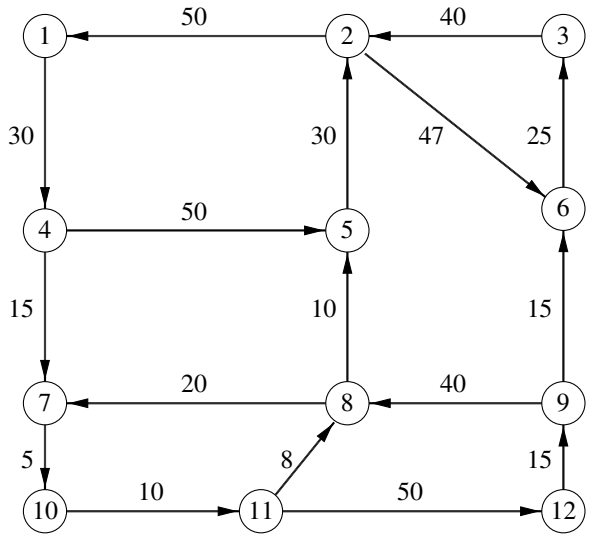


Figure 1.2: A network representing the system; street lengths are associated with each street segment.

1.2.2 Formulating a Network Optimization Problem

Given such a system of interconnected streets, we may be interested in answering questions such as

> *"What is the shortest driving distance between* `intersection-X` *and* `intersection-Y`*?"*

This question translates into a problem that we can solve using our network.[1] Using the earlier definition of a directed path, we can translate the above question in terms of the network $\mathcal{N} = (G, l)$, where `intersection-X` and `intersection-Y` correspond to nodes $x \in N$ and $y \in N$ as follows:

> *"Find a shortest x-y dipath P in G."*

This problem statement leads to the following network optimization problem formulation, where the length of a dipath P is equal to the sum of the edge lengths l_e for all edges $e \in P$:

$$\begin{array}{ll} \min & \sum_{e \in P} \; l_e \\ \text{such that} & P \text{ is an } x\text{-}y \text{ dipath in } G \end{array}$$

We now describe how network optimization can be used to solve this problem formulation, returning a solution or some proof that no solution exists.

1.2.3 Solving a Network Optimization Problem

Methods for solving network optimization problems are known as *network algorithms*. Graph and network-based algorithms consist of a set of instructions. *Execution* of such an algorithm is the process of following these instructions by performing a series of operations on a network instance. Thus, algorithm executions result in a finite sequence of operations on the nodes, edges, and associated information that yields a solution or some proof that no solution exists. The algorithm execution process is illustrated in Figure 1.3.

Return now to our problem formulation for finding the shortest driving distance between two intersections. Consider the situation where this problem formulation does not have a solution (i.e., there is no x-y dipath P in network $\mathcal{N}$). In this case, we say that our problem is *infeasible*. Recall our underlying question:

> *What is the shortest driving distance between* `intersection-X` *and* `intersection-Y`*?*

If our problem formulation is infeasible, the appropriate answer would be

> *You cannot drive from* `intersection-X` *to* `intersection-Y` *via the given system of streets, and thus the shortest driving distance between those two intersections is undefined (infinite).*

When an x-y dipath P does exist (i.e., the problem is *feasible*), the solution to our problem is a shortest dipath with respect to the street segment lengths l. For the example system illustrated in Figure 1.1, our readers should be able to convince themselves through observation that at least one dipath exists between each pair of intersections.

Let's consider a particular instance where we are trying to find the shortest driving distance from our house at the corner of `Maple/Washington` to the movie theater at `Davis/Ike`. One approach would be to enumerate all dipaths between the two intersections, calculate the length of each dipath, and choose a dipath with minimal total length. While this approach seems straightforward for our example system, for larger systems it becomes impractical to find all dipaths. Fortunately for us, there are network optimization algorithms that will solve our problem more efficiently, even for large networks. One such algorithm that we can apply is known as *Dijkstra's Algorithm*.

[1] In fact, what we have described is known as the *shortest-path problem*. See § 3.2 for more information on this problem.

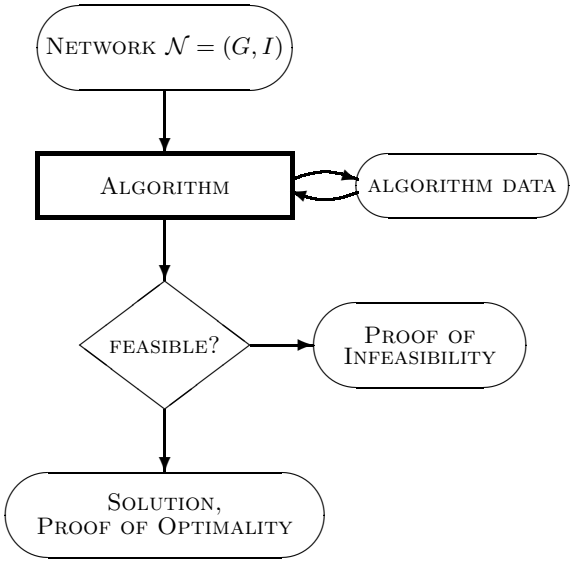


Figure 1.3: Solution process representation.

Dijkstra's Algorithm

Consider a directed network $\mathcal{N} = (G, l)$, where l_e is a non-negative length for each edge $e \in A$. Given such a network and some node $s \in N$, Dijkstra's algorithm finds the shortest dipaths from node s to all other nodes in N.[2] To do so, Dijkstra's algorithm maintains a partition of the node set N into three subsets: a set of *permanently labeled nodes* to which a shortest dipath is known from s, a set of *labeled nodes* to which some dipath is known from s — though it may not be the shortest such dipath available, and a set of *unlabeled nodes* to which no dipath from s is known. Figure 1.4 presents a description of Dijkstra's algorithm where these three sets are referred to as P, L, and U for permanently labeled, labeled, and unlabeled nodes respectively.

Although we do not offer a proof here, it can be shown that

1. Dijkstra's algorithm terminates finitely when given a finite network $\mathcal{N}$ as input.
2. At completion of Dijkstra's algorithm, all nodes belong to set P or set U.
3. For each node $i \in U$ at completion of Dijkstra's algorithm, there is no dipath P in $\mathcal{N}$ from s to i.
4. For each node $i \in P$, the shortest dipath from node s to node i has total length d_i.

Application of Dijkstra's Algorithm to Our Example

Let's now apply Dijkstra's algorithm to our example network from Figure 1.2 to find the shortest dipath from our house (node 1) to the theater (node 6). We will demonstrate the application of Dijkstra's algorithm through a series of illustrations.

[2] If dipaths do not exist from s to some nodes in N, the final "distance" of the shortest dipaths to those nodes will be reported as infinite.

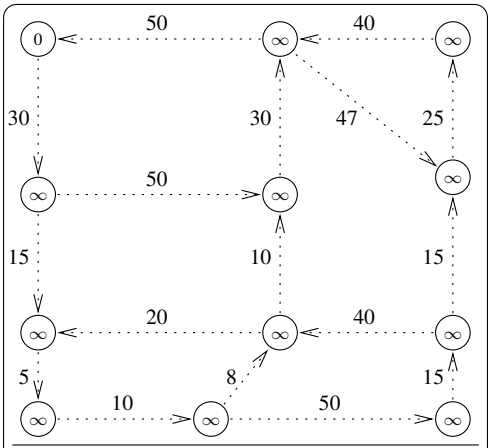
```
algorithm Dijkstra;
{
    // initialize algorithm information
    d_i := ∞ and pred(i) := null for all i ∈ N;
    P := ∅, L := ∅, U := N;

    // setup for starting at the given node s
    d_s := 0;
    move node s from set U to set L;

    while (the set L is non-empty) do
    {
        select node i ∈ L such that d_i ≤ d_j for all j ∈ L;
        move node i from set L to set P;

        for (all nodes j with (i, j) ∈ E) do
        {
            if (d_j > d_i + l_(i,j)) then
            {
                d_j := d_i + l_(i,j);
                pred(j) := i;
                if (j ∈ U) then move j from set U to set L;
            }
        }
    }
}
```

Figure 1.4: Description of Dijkstra's algorithm.

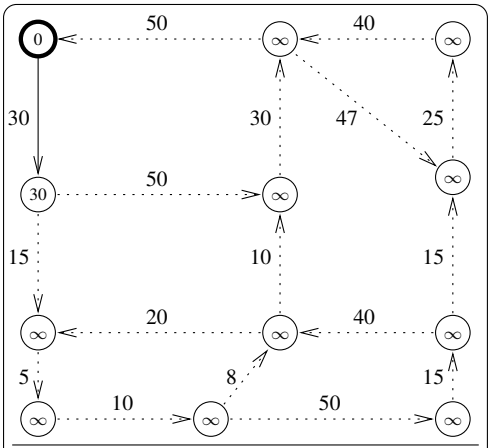
In the illustrations that follow, each node $i \in N$ is drawn with an associated distance label d_i that represents the length of the shortest dipath currently known from s to i; for unlabeled nodes this distance label is "∞." Nodes are drawn as either solid or bold circles. Nodes drawn as solid circles may belong to either L or U, depending on whether the distance label is ∞. Bold circles represent nodes that belong to P, i.e., those that have been permanently labeled; the distance labels for such nodes $i \in P$ gives the length of the shortest dipath from s to i.

Each edge is drawn either as a dotted arrow, a solid arrow, or a bold arrow. An edge that is drawn as a dotted arrow is not currently used in any dipath from node s. An edge $e = (i, j)$ that is drawn as a solid arrow is in the best dipath *currently known* from node s to node $j \in L$. An edge $e = (i, j)$ that is drawn as a bold arrow belongs to the best dipath from node s to node $j \in P$.

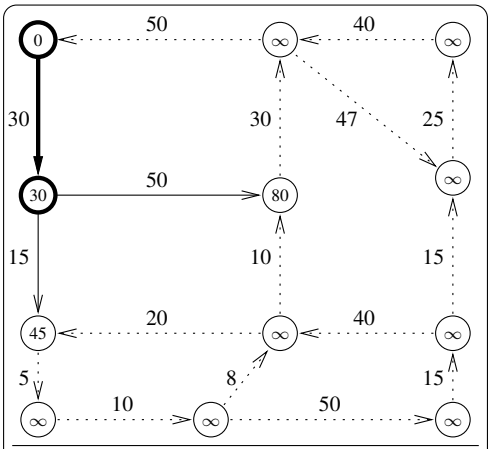
In the illustration for `Step 0` we see the state of the network after algorithm information has been initialized and node s has been labeled. `Step 1` – `Step 12` show the state of the network after each pass through the `while` loop in Dijkstra's algorithm as described in Figure 1.4.



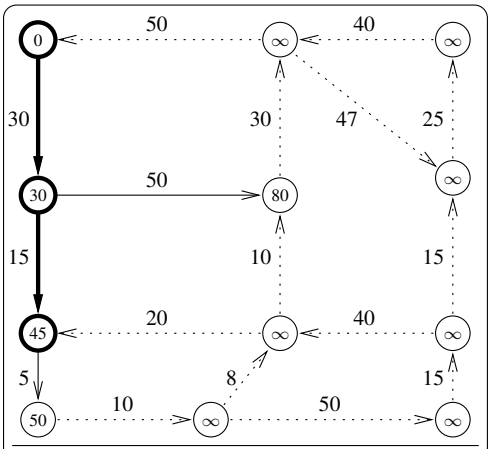
Step 0. $P = \emptyset$
$L = \{1\}$
$U = \{2,3,4,5,6,7,8,9,10,11,12\}$



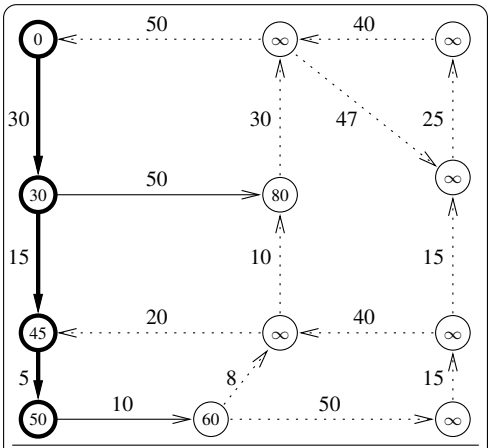
Step 1. $P = \{1\}$
$L = \{4\}$
$U = \{2,3,5,6,7,8,9,10,11,12\}$



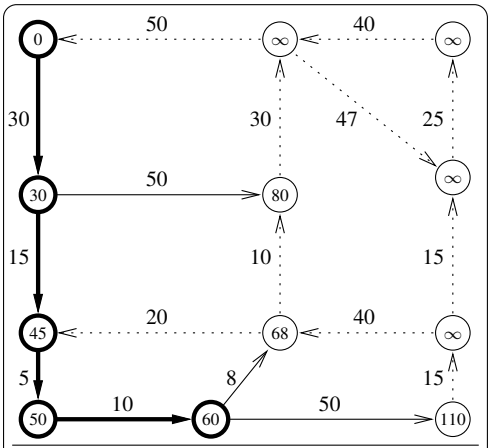
Step 2. $P = \{1,4\}$
$L = \{5,7\}$
$U = \{2,3,6,8,9,10,11,12\}$



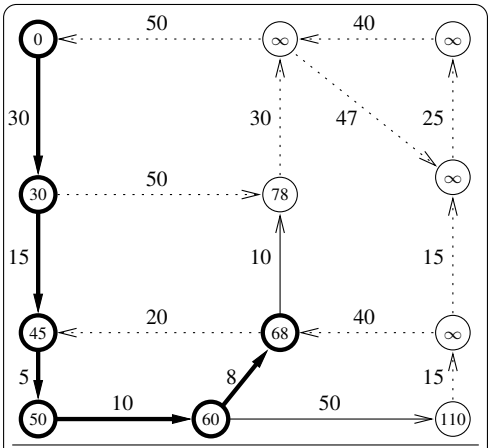
Step 3. $P = \{1,4,7\}$
$L = \{5,10\}$
$U = \{2,3,6,8,9,11,12\}$



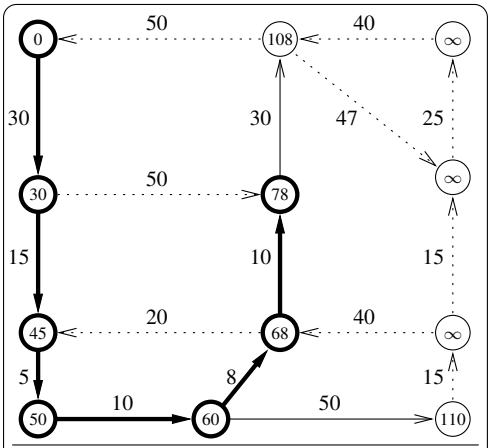
Step 4. $P = \{1,4,7,10\}$
$L = \{5,11\}$
$U = \{2,3,6,8,9,12\}$



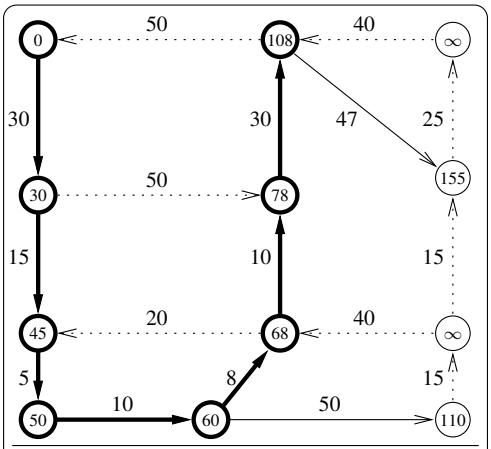
Step 5. $P = \{1,4,7,10,11\}$
$L = \{5,8,12\}$
$U = \{2,3,6,9\}$



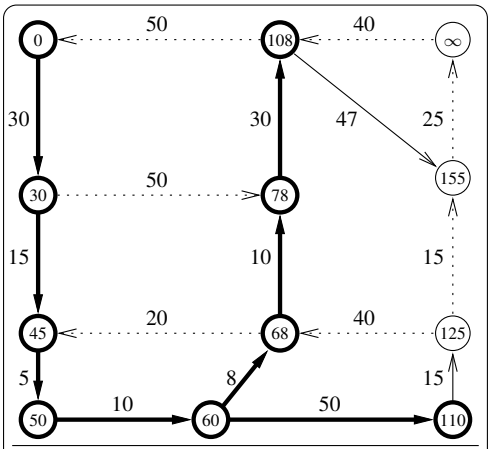
Step 6. P = {1,4,7,8,10,11}
L = {5,12}
U = {2,3,6,9}



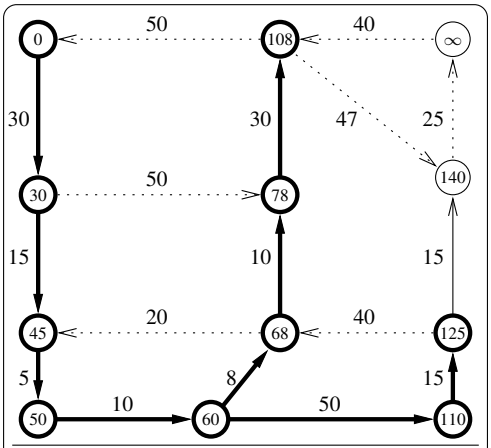
Step 7. P = {1,4,5,7,8,10,11}
L = {2,12}
U = {3,6,9}



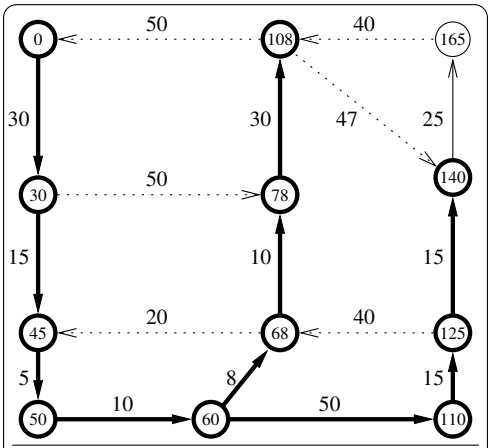
Step 8. P = {1,2,4,5,7,8,10,11}
L = {6,12}
U = {3,9}



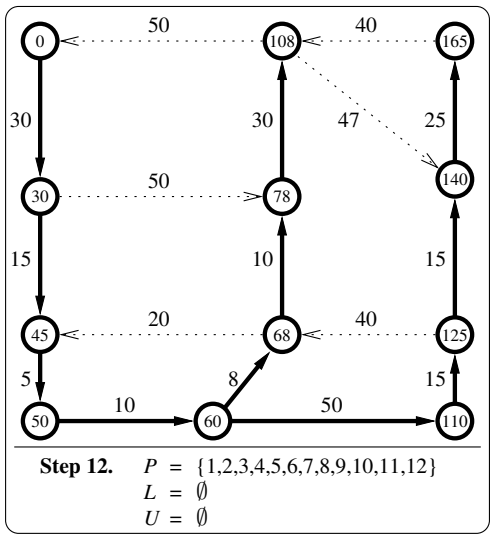
Step 9. P = {1,2,4,5,7,8,10,11,12}
L = {6,9}
U = {3}



Step 10. P = {1,2,4,5,7,8,9,10,11,12}
L = {6}
U = {3}



Step 11. P = {1,2,4,5,6,7,8,9,10,11,12}
L = {3}
U = $\emptyset$



Notice that we permanently labeled node 6 in `Step 11`. For the purposes of our example, we would have liked to have terminated execution after this step since we are only interested in finding the shortest dipath from node s to node 6. We can specialize Dijkstra's algorithm to terminate early when it finds the shortest dipath from the start node s to a specified destination node t by changing the stopping criteria from

> **while** the set L is non-empty **do** ....

to the stopping criteria

> **while** $t \notin P$ and the set L is non-empty **do** ....

With this change Dijkstra's algorithm terminates execution if we permanently label the destination node t, even when L is non-empty. (Observe that by modifying the algorithm in this manner, we no longer maintain properties (2) and (3) given earlier for Dijkstra's algorithm.)

1.2.4 A Solution for a Network Optimization Problem

Throughout this section we have worked towards answering the question

> *What is the shortest driving distance between our house and the theater?*

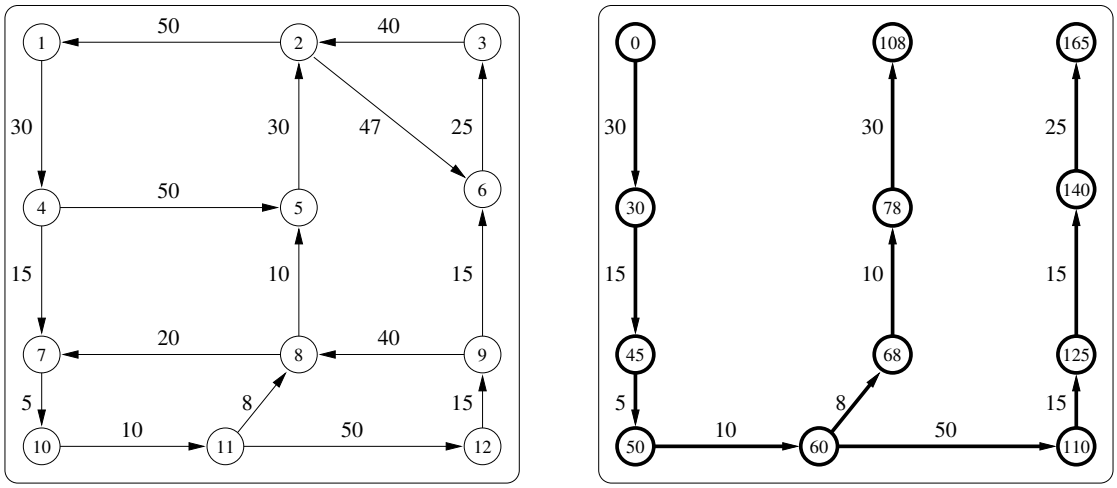
We have modeled the streets in our neighborhood as a network $\mathcal{N} = (G, l)$, posed the question as a network optimization problem on $\mathcal{N}$, and applied Dijkstra's algorithm to solve the network optimization problem. All that is left now is to interpret the solution as an answer to our question.

The solution that is given by Dijkstra's algorithm for our problem is illustrated in Figure 1.5b. For any node $i \in N$, the shortest dipath from s to i in our network is given by the set of edges that connect node s to node i. In particular, the shortest dipath from node 1 (representing our house) to node 6 (representing the movie theater) is the dipath

$$P_{1,6} = \{(1,4),(4,7),(7,10),(10,11),(11,12),(12,9),(9,6)\}.$$

The total length of the dipath from node s to node i for $i \in N$ is given by the distance label d_i represented in the nodes of Figure 1.5b. For our example problem, the total length of the dipath $P_{1,6}$ between node 1 and node 6 is 140 units. Thus, we have an answer to our question:

> *The shortest driving distance from our house to the theater is 140 units: Drive three blocks down Washington Ave. until you reach the intersection of Washington Ave. and Pine St. Turn left onto Pine and drive two blocks until you reach Davis Ave. Turn left onto Davis and drive two blocks to the intersection of Davis and Ike Blvd.*



(a) Our network model $\mathcal{N} = (G, l)$.

(b) The solution to our example problem.

Figure 1.5: Final network and problem solution.

1.3 The GIDEN Environment

GIDEN is a visually-oriented interactive software environment for network optimization whose fundamental purpose is to facilitate the visualization of network optimization problems, solutions, and algorithms. The GIDEN system has three primary components: The first component, "core-GIDEN," controls the graphical user interface and provides the framework for communication between solvers and graphical input/output devices. The second component is a toolkit of animated solvers. The third component is a collection of data structures that are tailored to the needs of a network optimization environment; both core-GIDEN and the solvers use these data structures extensively.

The current release of GIDEN is implemented in Java, an object-oriented programming language with built-in platform-independent graphics routines (see `http://java.sun.com`). In its current form GIDEN can be run as a stand-alone application on on any Java-enabled computing platform (i.e., any machine with an available Java interpreter).[3]

1.3.1 Core-GIDEN

At the heart of the GIDEN environment is a framework that provides the interface between graphical input/output devices and algorithm implementations — we call this framework "core-GIDEN". The idea behind core-GIDEN is to provide communication channels between users and solvers that are convenient for each. For users this translates into a graphical interface that is intuitive, flexible, and easy-to-use. For solvers, core-GIDEN provides convenient mechanisms for retrieving and reporting information through the graphical devices. After describing the core-GIDEN mechanism for animation, we will discuss the core-GIDEN user and solver interfaces.

Animation Sets

Solvers in GIDEN provide animation through the use of *animation sets*. Animation sets are based on the idea that the progress of a solver, given the current state of the network, can be represented by classifying nodes and edges into mutually exclusive sets. By representing each set with a distinct color, users can follow the progress of the solver by observing changes in the colors of nodes and edges. (See Chapter 3 for information on the animation sets that are used in specific solvers.)

Core Environment User Interface

Users communicate with the GIDEN environment through the graphical interface, using a mouse to select actions and the keyboard to input textual information. The graphical interface is designed to be similar to that of popular applications for the Macintosh and Windows operating systems, making the basic functionality of GIDEN easy to access and use.

GIDEN users can create and modify networks using the mouse and keyboard. Once a network is built (or opened), the user selects a solver from the solver toolkit to apply to the network. At that time, the network is analyzed and the user is prompted for any undefined information needed by the solver. The user then executes the solver, controlling the level of animation detail and the animation speed. Pseudo-code of the solution algorithm may be provided in a separate window with a graphical tracer that shows the progress of the executing algorithm.

The reader is referred to Chapter 2 for more complete information on the GIDEN user interface.

[3] GIDEN is being developed under Windows9x/NT, Linux, and Solaris operating systems.

Core Environment Solver Interface

Core-GIDEN is an object-oriented environment (see [7] for a general description of object-oriented programming). Solvers are implemented as derived classes of a generic solver base class named `ExecBase` that is included in core-GIDEN. The constructor for the derived solver class is passed a `UserBase` object that provides several methods for communicating with the user interface; stable functions defined in the `UserBase` object are used by the solver class to report important changes (e.g., when items have moved between animation sets). Virtual methods in `ExecBase` provide for communication from core-GIDEN to the solver.[4] In particular, through calls to these methods, core-GIDEN instructs the solver when to prepare for execution, when to execute, and when to terminate execution.

1.3.2 Solver Toolkit

Solvers contain the algorithm logic to solve network optimization problem instances. Solvers perform data manipulations, update the associated animation mechanisms, and inform core-GIDEN of the animation changes. As mentioned previously, solvers are implemented as derived classes of a generic solver base class named `ExecBase`. In this section we describe how GIDEN's solver toolkit is organized; descriptions of the solvers included in the toolkit are given in Chapter 3.

While each network optimization problem has a variety of solution algorithms, the input and output of each algorithm for a given problem are typically the same. For this reason, algorithm implementations belonging to the solver toolkit are implemented as derived classes of an associated problem class, rather than as classes derived directly from `ExecBase`. For example, rather than having the solver for Dijkstra's algorithm as a derived class of `ExecBase`, the solver class for Dijkstra's algorithm is a derived class of the `ShortestPath` problem class which is derived directly from `ExecBase`. Figure 1.6 illustrates the relationship between individual solver classes and `ExecBase`. For a given problem all solvers will have a collection of common input and output requirements (e.g.,

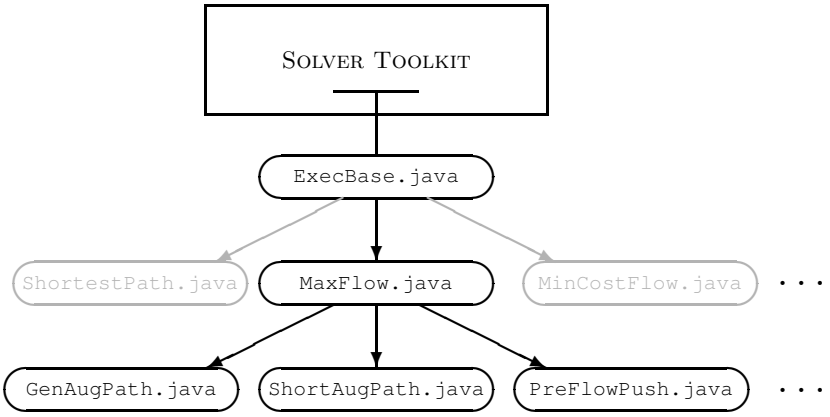


Figure 1.6: Class hierarchy for solver implementations.

all maximum-flow problem solvers require a source node, a sink node, and edge capacities as input, and provide edge flows and a minimum cut as output). Similarly, all solvers for a given problem will also have a collection of common animation sets, namely those associated with displaying problem instance solutions. The base class for each problem (e.g., `ShortestPath`) is designed to include all common characteristics for related solvers, allowing individual solver classes to benefit from code

[4] See Section 4.1.2 for a description of these methods.

reusability at the problem level. In addition to having the actual solver logic, individual solver classes are only required to have data checking routines and animation sets that differ from those of other solvers for the same problem.

Although the solver toolkit included with GIDEN covers a range of problems and is continually expanding, many users will want to add their own solvers to the toolkit. The following features of GIDEN facilitate solver development:

- prototypes for new solver implementations
- direct access to network data structures
- access to animation set creation and manipulation
- visual debugging through the user interface

The necessary tools for adding solvers to the solver toolkit are a text editor and the Java Development Kit (JDK), or some Java-based integrated development environment (IDE). Additionally, you must have access to the implementor's distribution of GIDEN.[5] The process of developing new solvers and adding them to the toolkit is described in Chapter 4.

1.3.3 Network Data Structures

The network data structures library is based on standard graph data structures (see, for example, [9]). Node and edge arrays are provided to associate values with nodes and edges. Additional data structures include single and double linked lists, priority queues, queues, and other network related data structures. Both the single and double linked lists were developed from a common base class that supports all linked list methods. A common base that supports both types of linked lists provides a simple mechanism to switch between single and double linked lists without needing to edit or recompile the source code. Such libraries allow users to quickly build efficient network algorithm implementations. Our prototype version of GIDEN was implemented in the C++ programming language and used LEDA (Library of Efficient Data types and Algorithms) developed at the Max-Planck-Institut für Informatik (see [8]). The current data structures library used by GIDEN is based on a similar environment written by us in C++ (see [4]).

[5]Currently, only limited documentation is available to support GIDEN solver developers. For this reason, the implementor version of GIDEN is available only by special request. Please contact the authors for more information.

Chapter 2

Using the GIDEN Environment

2.1 The GIDEN User Interface

The GIDEN graphical user interface (GUI) features a single *controller window* and multiple *network windows*. In addition, each network window may have an auxiliary *pseudocode window* when it is operating in solver execution mode.

The main purpose of the GIDEN *controller window* (see Figure 2.1) is to create and manage all network windows, and to provide central access to the network editing tools. Additionally there are a few miscellaneous "global" operations that are available through the controller window.

Figure 2.1: GIDEN-2.0 controller window with "Edit Values" tool selected.

The second type of window is a *network window*. The controller window can create and manage multiple network windows simultaneously, each running in its own thread. Each network window can be used in two modes. In *edit mode*, a network window can be used to create and modify networks, as illustrated in Figure 2.2. A network window is in *solver mode* when executing an algorithm implementation from the GIDEN solver toolkit. An example of a network window in solver mode is shown in Figure 2.3, along with an auxiliary *pseudocode window*.

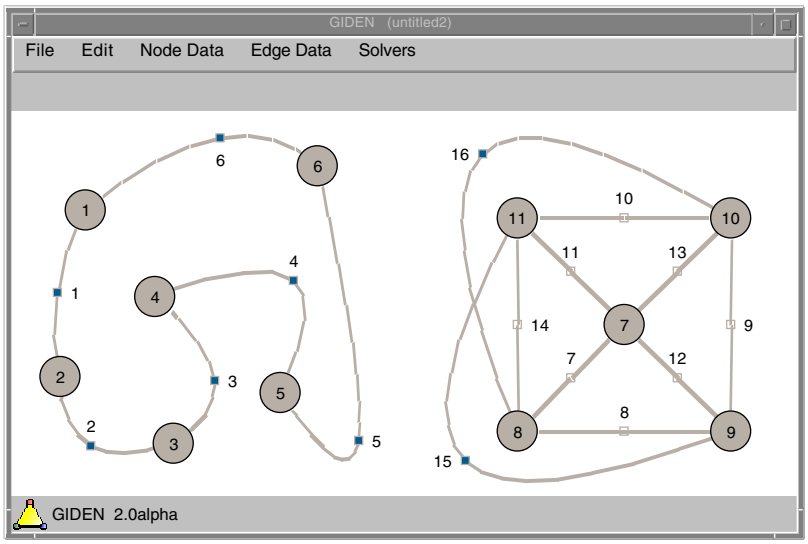


Figure 2.2: GIDEN-2.0 network window in edit mode.

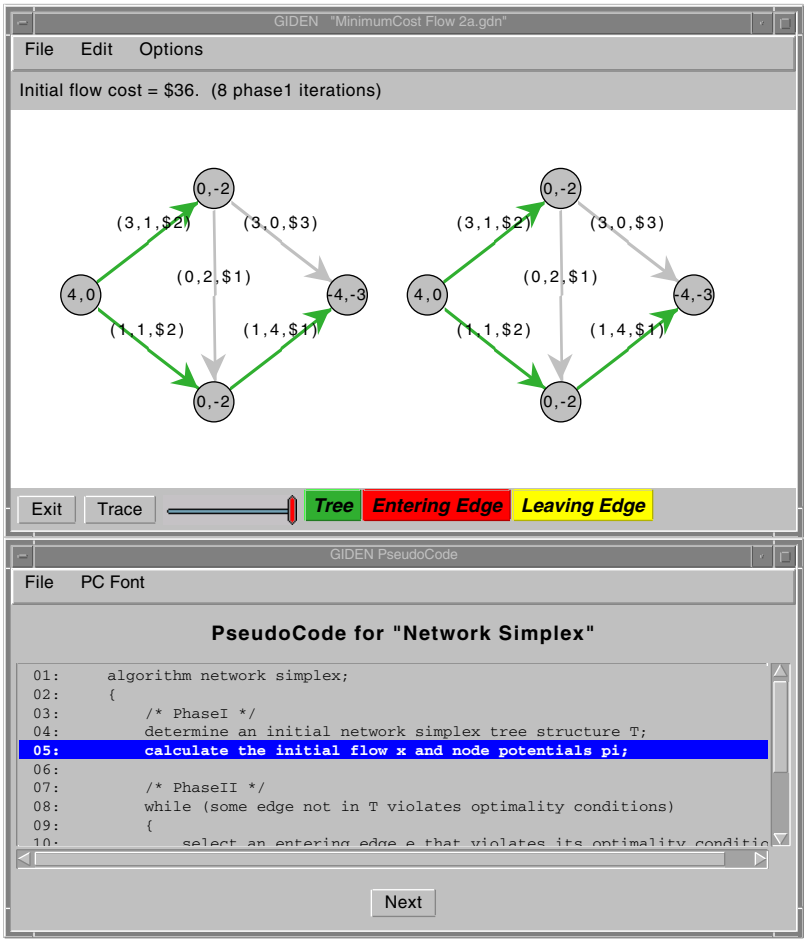


Figure 2.3: GIDEN-2.0 network window during solver execution with pseudocode displayed.

2.2 Building and Editing Networks

The GIDEN environment includes many features for building network instances. In this section we describe the facilities for drawing networks and for managing information associated with a network's nodes and edges. Throughout this section we assume that the network window of interest is in edit mode.

It is important to note that the current version of GIDEN *does not* include facilities to "undo" any editing action.

2.2.1 Drawing Networks

Network instances in GIDEN are created by placing nodes and edges on the "drawing canvas" of a network window that is in edit mode. There are several tools and operations available in GIDEN for drawing networks from scratch or for modifying existing networks. Many of the drawing operations use the "edit tools" that are available through the main controller window, while other operations and settings are accessible through the menu system of a network window.

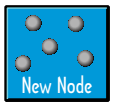
Edit Tools

A prominent feature of the controller window is the collection of icons representing available "edit tools." Each of these tools is used to control certain editing operations in network windows. An edit tool may be selected by clicking on the associated tool icon or by choosing the appropriate entry from the Tools menu in the controller window. Only one tool may be selected at any given time, and this selected tool is available to all network windows that are running in edit mode.

Most of the "edit tools" available in the controller window are used for drawing operations, such as creating, deleting, and placing nodes and edges on the canvas. Other tools are used to set the direction of edges, to control the placement of edge labels, and to edit information associated with nodes and edges. This section describes the purpose and usage of each of the edit tools.

The Trail Edge tool is used to draw a series of nodes and edges. After selecting the tool, the first click on the drawing canvas will either create a new node at the mouse position or, if a node already exists at that position, it will select that pre-existing node. Subsequent clicks on the canvas will create edges in series from the most recently selected node to the current mouse position, creating a new node at the current position if none exists.

The Star Edge tool is used to draw several edges emanating from a single node. After selecting the tool, the first click on the drawing canvas will either create a new node at the mouse position or, if a node already exists at that position, it will select that pre-existing node as the anchor node. Subsequent clicks on the canvas will create an edge between the anchor node and the current mouse position, creating a new node at the current position if none exists.



The New Node tool is used to create a new node on the drawing canvas. After selecting the tool, each click on the canvas will create a new node at the current mouse position. If a node already exists at the selected location, then no action is performed.

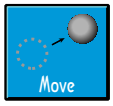
The New Edge tool is used to create a new edge on the drawing canvas. In order to create an edge $e = (i, j)$ between existing nodes i and j, first click on node i, and then click on node j.

The Reverse Edge tool is used to reverse the direction of an edge. When using this tool, selecting an edge $e = (i, j)$ will swap the head and tail nodes such that $e = (j, i)$. (*Note: This tool will still operate even when a network is being drawn as undirected. To see the direction of an edge, turn on the* Directed Edges *option in the network window's* Edit *menu.*)

The Edit Values tool is used to edit information that is associated with the nodes and edges. To edit the currently displayed information, use the mouse to select the label of the target node or edge, and then type the new information in the resulting text input area. See Section 2.2.2 for more information on editing data fields.

The Select Item tool is used to select multiple nodes and edges that will either be deleted from or re-positioned on the drawing canvas. With this tool there are two methods for selecting nodes and edges. The first method is to use single mouse clicks to select individual nodes and edges. The second method is to "click and drag" the mouse over a region of the canvas, which results in selecting all items in the target region. Holding the `<shift>` key when using the first method will toggle the selection status of the item. Holding the `<shift>` key when specifying a region of the canvas will prevent previously-selected items from being automatically de-selected.

Once selected, nodes and edges can be repositioned by dragging them with the mouse while holding the `<control>` key, or by dragging with the middle mouse button on platforms that support a 3-button mouse. Fine tune positioning for selected items is available by using the Nudge Selected operations in the Edit menu. For more information on editing operations available for selected items, see Section 2.2.1 below.



The Move Item tool is used to re-position items on the drawing canvas. When this tool is selected, the cursor changes shape to indicate when the mouse is positioned over a movable item. Use the mouse to select and drag the target item from its current position to the desired location on the canvas. Items that can be moved using this tool include nodes, edge labels, and edge anchor points. For each of these items, details differ on the boundaries accepted for selecting an item, and the rules used to govern item positioning:

Selecting and Positioning Nodes. Nodes may be selected by pressing the mouse button anywhere in the boundary of the drawn node. The selected node can then be dragged to any position on the drawing canvas, without restriction.

Selecting and Positioning Edge Labels. The position of an edge label is tied to the location of the anchor point for the associated edge. An edge label is selected by pressing the mouse button anywhere in the boundary of the currently displayed label.[1] The edge label can then be positioned to the north (centered), south (centered), east (left aligned), or west (right aligned) of the edge anchor point.

[1] If the edge data field currently displayed is empty for a particular edge, then the associated edge label cannot be re-positioned. In this case, temporarily change the displayed data to Edge ID (or some non-empty data field) before attempting to move the edge label.

Selecting and Positioning Edge Anchor Points. Edge anchor points are used to control the shape of the drawn edge and the general location of the associated edge label. An edge anchor point is selected by pressing the mouse button anywhere in the boundaries of the box region drawn at the current anchor point location. The selected edge anchor point can then be dragged to any position on the drawing canvas, without restriction. To allow edge anchor point to be automatically calculated, click the edge anchor point while holding the `<shift>` key.

The Delete Item tool is used to delete nodes and edges from the drawing canvas. To delete a node, click anywhere in the boundary of the drawn node. To delete an edge, click on the associated edge anchor point or edge label. Note that deleting a node results in all adjacent edges being deleted as well.

Edit Menu Operations and Settings

This section briefly describes the drawing operations and settings that are available through the Edit menu of a network window in edit mode. The scope of these menu selections is limited to the single target network window. Several of the menu choices perform operations that are available only when the Select Item tool is enabled; see the Select Item tool description above for more information.

Cut	Delete the selected nodes and edges from the drawing canvas. This choice is available only if the Select Item tool is enabled and there are nodes and/or edges that are currently selected.
Clear	Delete all nodes and edges on the drawing canvas. This choice is always available in edit mode.
Select All	Select all nodes and edges on the drawing canvas. This choice is available only if the Select Item tool is enabled.
Select All Nodes	Select all nodes on the drawing canvas. This choice is available only if the Select Item tool is enabled.
Nudge Selected	Move all selected nodes and edges by one pixel in the chosen direction.[2] This choice is available only if the Select Item tool is enabled.
Enable Grid	Turn on a drawing grid for positioning nodes. Subsequent operations for adding or moving a node will align the node's center with a grid intersection point.[2] This choice is always available in edit mode.
Directed Edges	This setting determines if an edge is drawn as directed or undirected. This choice is always available in edit mode, but may be overridden by an executing solver.

2.2.2 Managing Node and Edge Information

Associated with each network window in GIDEN is an underlying network object. This network object is composed of the following elements: a node set, an edge set, node data fields, and edge data fields. The member elements of the (possibly empty) node and edge sets correspond to the nodes and edges displayed on the network window's drawing canvas, and can thus be managed through the drawing operations described in Section 2.2.1. In this section we describe the facilities within GIDEN for managing the node and edge data fields.

The node and edge data fields (or "arrays") are internally stored as `NodeArray` and `EdgeArray` objects (see [4]). Each node and edge array has a default *data type* associated with it, which

[2]The drawing grid is ignored when using the Nudge Selected operations to move nodes.

determines how the values of the array are parsed. Currently two data types are supported: "text" type and "integer" type. Values in a text array are represented using Java `String` objects, and values in an integer array are represented using Java `Integer` objects. Data fields are handled differently within GIDEN based on their associated type; for an example of this, see the topic "Editing Data Fields" in this section.

Default Data Fields

When a network object is constructed, it is created with three default data fields that are internally "owned" and managed by GIDEN. Because these fields are managed internally by the environment, the values associated with these fields cannot be edited directly by the user. These default fields are named Node ID, Edge ID, and Euclidean Length. The following text describes these fields and their intended usage:

Node ID — This is a node data field of text type. As the name suggests, the node array contains a unique identifier for each node in the network object. The primary purpose of the Node ID array is to distinguish nodes internally within GIDEN.

Edge ID — This is an edge data field of text type. Analogous to the Node ID array for nodes, this data field is used as a container for unique edge identifiers.

Euclidean Length — This is an edge data field of integer type. The array is used to store integer values that represent the Euclidean distance between an edge's two endpoint nodes. Values are calculated using the center coordinates of the endpoint nodes. Note that the value associated with an edge gives the approximate length of that edge on the canvas if it is drawn as a straight line (e.g., if the edge's anchor point is not manually positioned). The value associated with a given edge is automatically recalculated when either of its endpoint nodes is moved (e.g., with the Move Item tool). Although managed internally, the Euclidean Length array is provided as a convenience for end-users. The data array may be used as an input to any of the GIDEN solvers that require an integer type edge array as input (see Section 2.3.2).

User-Defined Data Fields

In addition to the default arrays that are associated with every network, users can create new node and edge data fields. These user-defined arrays are created through the Add Data Field... operation in a network window's Node Data or Edge Data menu (see Section 2.2.2). Selecting this operation will evoke a dialog for specifying the following data field properties:

Name — This is a text identifier for the data field. This is the name that will be listed in the appropriate data field menu (see Section 2.2.2) and in the solver input dialogs (see Section 2.3.2). Valid names can include numbers, letters, spaces, and dashes. To avoid confusion, you cannot assign a name to a new node (edge) array that conflicts with an existing node (edge) array name.[3] Leading, trailing, and consecutive spaces will be removed from the name.

Default Value — This is the default value of an item in the array. When an array is created, this value is assigned to all existing nodes (edges). It will also be the value assigned to all newly created nodes (edges).

[3] For the purpose of comparison, names conflict if they are equal when ignoring case. For example, the name "length" conflicts with the name "Length".

Type — This is the data type associated with the array. Currently two data types are supported: "text" type and "integer" type. Values in a text array are represented using Java `String` objects, and values in an integer array are represented using Java `Integer` objects. The underlying data type of an array determines the range of valid member values (as according to Java specifications).

The Data Field Menus

This section briefly describes the operations and settings that are available through the Node Data and Edge Data menus of a network window. Through these menus the user can create new arrays, delete existing arrays, and select the data fields that are displayed in the node and edge labels on the drawing canvas:

Creating New Arrays. New node and edge data fields are created using the Add Data Field... operation in the appropriate data menu. Selecting this operation will result in a dialog for specifying the properties of the new array. See the topic "User-Defined Data Fields" in this section for more information.

Deleting Existing Arrays. Existing data fields can be deleted from the network by selecting the array name through the Delete Data Field submenu. Only user-defined data fields may be deleted. In particular, the default data fields Node ID, Edge ID, and Euclidean Length are managed internally and can not be deleted (see Section 2.2.2).

Selecting Display Arrays. When a network window is in edit mode, each node and edge has an associated label that is displayed on the drawing canvas. To set which data field values are displayed in the node (edge) labels, select the appropriate array name from the top portion of the Node Data (Edge Data) menu.

Editing Data Fields

Node and edge data fields are edited using the Edit Values tool (see Section 2.2.1). Follow the steps below to edit a user-defined array:

1. Select the target array as the display array under the appropriate data menu in the network window.
2. Select the Edit Values tool from the controller window.
3. Use the mouse to select the label of the node or edge whose value you want to change. This should replace the label with a text input box that contains the current value.
4. Type the new data value in the text input box.
5. Press `<enter>` to accept the new value.

After entering a data value in the text input box, the new value is checked to ensure that it is valid for the underlying array data type. For example, integer type arrays will not accept values that are out of range for a Java `Integer`, or values that contain non-digit characters. If the specified value is invalid, then a warning will appear and the edit will be ignored.

2.3 Running Solvers

GIDEN users can create and modify networks using the mouse and keyboard as described in the previous section. Once a network is built (or opened), the user may select a solver implementation to apply to the displayed network. The process of running a GIDEN solver is outlined in the following steps:

1. Select the desired solver from the Solvers menu.
2. Specify the input data fields required by the solver (if any).
3. Execute the selected solver.
4. Interpret the solver results.

In this section we describe these steps more carefully.

2.3.1 Selecting a Solver

The first step in executing a solver is to select the solver name from the Solvers menu of the network window (only available in edit mode). The solvers currently available in GIDEN are categorized according to the underlying problem type. Currently, the solver toolkit provides solvers from four problem domains: minimum spanning tree problems, shortest path problems, maximum flow problems, and minimum-cost flow problems.[4]

After being chosen from the Solvers menu, the solver will analyze the current network object. If the solver is unable to cast the displayed network into an appropriate problem instance, then it returns immediately (leaving the network window in edit mode). Otherwise, the network window is switched to solver mode, and the user may be prompted for required input information, as described below.

2.3.2 Specifying Inputs

When a solver is selected from the toolkit, the target network window switches to solver mode. At this point, most GIDEN solvers will generate a dialog window to prompt the user to specify the node and edge input arrays required by the solver. This *solver input dialog* has two columns of information. The first column (left) lists the *logical names* of the needed solver inputs. These logical names are text identifiers that the solver uses to request and retrieve the input arrays. The second column (left) includes choice menus that list available candidate arrays (if any) that can be associated with the corresponding input logical name. The candidate arrays included in the choice menus are selected based on the kind of array (i.e., node or edge) and the underlying data type. (For example, the Prim solver will list all edge arrays of integer type as candidates for its "Length" input array.) After selecting the desired input arrays, click the Accept button. To exit the dialog without making selections, click the Cancel button. In this case, the solver will exit and the network window will return to edit mode. Note that some networks may not contain node or edge arrays that match the requirements of a given solver. In this case the dialog will indicate that the appropriate data is not available, and the Accept button will not be able to validate the input selections. To exit the solver, click the Cancel button.

Aside from the initial input dialog, some solvers will ask for user input during solver execution. In most cases, directions for specifying the needed input will appear in the status line of the network

[4]Some distributions also include heuristic solvers for the symmetric traveling salesman problem.

window. Typical examples would require the user to use the mouse to select a particular node, or use the keyboard to specify an option setting.

2.3.3 Controlling Execution

After selecting the needed input arrays (if any), the solver is ready to be run by the user. You will notice several features of a network window in solver mode that distinguish it from when it is running in edit mode. Most notably, the bottom area of the network window will contain several execution-related controls: an Exit button, an "action" button, an animation slider, and possibly several animation-set buttons. Each of these controls is described below:

- *The* Exit *Button:* Selecting this button will cause the solver to terminate at the earliest opportunity and will result in the network window returning to edit mode.
- *The Action Button:* The appearance and functionality of the action button depends on the state of the solver and several related settings. Depending on these settings, the action button will display one of the following labels: Trace, Step, Go, Pause, Final, or Reset.

  When the action button is labeled Reset, the solver has completed execution. Selecting the action button when it displays this label causes the solver to begin execution again, as if the solver was selected from the Solvers menu. The topics "Final Animation Mode", "Trace Animation Mode", "Step Animation Mode", and "Continuous Animation Mode" that appear later in this section describe the functionality of the action button when it displays other labels.
- *The Animation Slider:* The animation slider is used to determine the current animation mode ("final", "trace", or "continuous"). It is also used to control the speed of execution when operating in continuous animation mode. For more details, see the topics "Final Animation Mode", "Trace Animation Mode", and "Continuous Animation Mode" later in this section.
- *Animation-Set Buttons:* Animation-set buttons are used to specify which information about the current solver will be visually displayed during "trace" or "continuous" animation mode (see below). Each of the buttons corresponds to a solver-specific action or state, and visually represents that action or state using a pre-determined color. Under usual circumstances, each animation-set buttons can be either "enabled" or "disabled" to signify whether or not the underlying action or state is to be displayed. The state of an animation-set button is toggled by clicking on the button when in "trace" mode or "continuous" mode. For details on the use of animation-set buttons, see the topic "Trace Animation Mode" later in this section. See Chapter 3 for information on the animation sets provided with each of the GIDEN solvers.

There are several different ways to execute a GIDEN solver, depending on the level of visualization that is desired. We will describe the basics here, but the best way to learn about the algorithm animation capabilities within GIDEN is to experiment with the various solvers and settings. We will cast our discussion of solver execution by describing five different "modes" of animation: final animation mode, trace animation mode, step animation mode, continuous animation mode, and pseudocode animation mode.

Final Animation Mode

Final animation mode is used to execute a given solver without any animation. To run in final mode, position the animation slider to the far left (so that the action button is named Final), and select the action button. This will execute the solver until it terminates — usually with an optimal solution or with a certificate of infeasibility. At conclusion of solver execution, the action button name will change to Reset.

Trace Animation Mode

Trace animation is intended as the default animation mode. When running a solver in trace mode, the detail of solver animation depends on the state of the animation-set buttons. In particular, the solver will suspend execution at certain epochs that correspond to the action or underlying state of an enabled animation-set button. Each time execution is suspended, the network is redrawn to reflect the current state of the solver.

To run a solver in trace mode, first position the animation slider to the far right, and make sure that the Detail Animation setting is turned off in the Options menu. The action button should now display the name Trace. To execute the solver, use the mouse to repeatedly select the Trace button.

You can think of trace animation mode as follows: Each click on the Trace button says to the solver *"start running until something interesting happens"*, where the solver decides if something "interesting" happened based on which animation-set buttons are enabled.

Step Animation Mode

Note: Step animation mode is intended to be used primarily for solver debugging purposes. We recommend against end users executing solvers in step mode.

Step mode is similar to trace mode with two major differences: (1) All animation-set buttons are treated as enabled, and (2) solver execution suspends whenever any item is placed in an animation set. Running a solver in step mode is similar to trace mode, except that the Detail Animation setting must be turned on in the Options menu.

Continuous Animation Mode

When executing in trace mode, a solver will suspend execution at certain epochs that correspond to "interesting" events. Then to resume solver execution, the user must manually select the Trace button. Continuous animation mode is similar to trace mode except that there is a built-in time delay between when a solver suspends, and when it resumes execution. The amount of this delay is determined by the position of the animation slider.

To execute a solver in continuous animation mode, first position the animation slider strictly between the far left and the far right positions. (The action button name changes to Go when the slider is positioned between the two extremes.) The built-in delay will decrease as the slider is moved to the left and increase as it is moved to the right.[5] After enabling the animation-set buttons of interest, press the Go button. This will start execution and change the action button name to Pause. The solver will continue to execute until it terminates or until you select the Pause button.

Continuous animation mode can be used with the Detail Animation setting either enabled or disabled. When detail animation is enabled, the behavior of continuous animation is similar to executing in step mode with a built-in delay. If the setting is not enabled (as by default), then continuous animation is similar to trace mode with an automatic delay.

[5] It may help to think of "final mode" as the far left extreme, where there is no delay, and of "trace mode" as the far right extreme, where the delay is infinite. However the continuous animation delay is actually a linear function between two finite values; the maximum automatic delay is approximately 5 seconds.

Pseudocode Animation Mode

As shown in Figure 2.3, certain GIDEN solvers will display an auxiliary window with a text-based pseudocode description of the algorithm. This *pseudocode window* contains a Next button, that is used to control solver execution by stepping through the algorithm logic. Each time the user selects the Next button, the solver resumes execution until it encounters the next algorithmic step that is represented in the pseudocode. The solver then suspends execution, highlights the current line of pseudocode, and redraws the network to reflect the state of the solver. Using this approach to control solver execution is what we mean by running in pseudocode mode.

Trace, step, and continuous animation modes all depend on the animation sets that are present (and enabled) in a solver. In contrast, the pseudocode animation mode is entirely independent of the available animation sets, and instead is driven by the text description of the underlying algorithm.

To execute a solver in pseudocode mode, the auxiliary pseudocode window must be visible. If this window is not visible by default (it will depend on the position of the network window and other factors), you may evoke the window by enabling the Show Pseudo-Code Window setting from the Options menu. Then click repeatedly on the Next button to step through solver execution.

Please note that not all solvers provide a pseudocode description of the algorithm. For solvers that do not support pseudocode animation mode, the Show Pseudo-Code Window setting will be automatically disabled.

2.3.4 Interpreting Results

Once a solver has found an optimal solution,[6] or determined that no such solution exists, it will terminate execution. Upon termination, most GIDEN solvers will display information about the final solution (possibly including a "certificate" of optimality), or they will provide a certificate of infeasibility. In either case, usually the animation-set buttons will be replaced with a a "result key" that can be used to interpret the solver's final result. Additional information about the solution may be provided in the network window's status line. The user is referred to Chapter 3 for information on the particular result information provided by each of the default GIDEN solvers.

[6] Some solvers may not yield an optimal solution, but may yield a "good" solution be some measure. These solvers are known as "heuristics."

Chapter 3

Solver Reference

In this chapter, we describe the network optimization problems that can be solved using the GIDEN solver toolkit, and we give information on available solvers. The information we present for each solver includes algorithm pseudocode that closely follows the algorithm descriptions presented in the text *Network Flows: Theory, Algorithms, and Application*, by Ahuja, Magnanit, and Orlin [1]; we have made it a priority to include the pseudocode exactly as it appears in [1] whenever possible. The following assumptions are made for all solvers in the GIDEN solver toolkit:

1. The network does not contain any parallel edges.
2. The network does not contain any loops.
3. Lower bounds on edge flows are zero-valued.

At present, GIDEN's solver toolkit includes implementations for the spanning tree, shortest path, maximum flow, and minimum-cost flow problems:

Minimum Spanning Tree

- Kruskal's algorithm
- Prim's algorithm

Shortest Path

- Dijkstra's algorithm
- FIFO label correcting algorithm

Maximum Flow

- Generic augmenting path algorithm
- Shortest augmenting path algorithm
- Preflow-push algorithm

Minimum-Cost Flow

- Capacity scaling algorithm
- Cycle canceling algorithm
- Out-of-kilter algorithm

- Network simplex algorithm
- Successive shortest path algorithm

Additional solver implementations are under development and testing.

3.1 Minimum Spanning Tree Problem

Given an undirected network $\mathcal{N} = (G, l)$, the *minimum spanning tree problem* is to find a spanning tree of G with minimal length, where the length of a tree T is calculated as $\sum_{e \in T} l_e$.

`Input:` An undirected network $\mathcal{N} = (G, l)$, where $l \in \mathbb{R}^A$.

`Formulation:`

$$\begin{array}{ll} \min & \sum_{e \in T} l_e \\ \text{s.t.} & T \text{ is a spanning tree of } G. \end{array}$$

`Feasible Output:` A minimum spanning tree T^* and the tree's total length $\sum_{e \in T^*} l_e$.

`Infeasible Output:` A proper subset R of N, such that no edges (i, j) exist with $i \in R$ and $j \in N \backslash R$. We also give either a forest F^* composed of a minimum spanning tree T^* for each component of G, or a minimum spanning tree for the component of G with node set R.

Required Problem Input

GIDEN solvers derived from the `MinSpan` problem base class will request the following data field in the input dialog:

"Length" — edge array of integer data type;
provides edge lengths l_e for each $e \in A$.

3.1.1 Kruskal's Algorithm

```
algorithm kruskal;
{
    sort edges in nondecreasing order of length;
    LIST := ∅;
    while (|LIST| < |N| − 1 and unexamined edges exist) do
    {
        e := unexamined edge with minimal length;
        if (adding e to LIST does not create a cycle) then
            add e to LIST;
        else
            discard e;
    }
} (END of 'kruskal')
```

Solver-Specific Input: (none)

Solver Animation Sets:

Trial **(red):** An edge being considered for inclusion on LIST.

Acquired **(green):** Edges included on LIST and nodes with an incident edge on LIST.

Discarded **(yellow):** Edges $e = (i, j)$ that were considered for inclusion in LIST but were rejected because adding e to LIST would have created a cycle in the component of G containing node i and node j.

Label Information:

`Edge labels:` Edge length l_e for each edge $e \in A$.

`Node labels:` None.

Feasible Result Information:

Minimum Spanning Tree **(orange):** Used to color the edges in a minimum spanning tree of G. The tree length is reported in the network window status line.

Infeasible Result Information:

Forest of Component-wise Minimum Spanning Trees **(orange):** A minimum spanning tree is found for each component; this is used to color the edges in each such spanning tree. The total forest length is reported in the network window status line.

3.1.2 Prim's Algorithm

```
algorithm prim;
{
    /* initialize algorithm information */
    d_i := ∞ and pred(i) := null for all i ∈ N;
    P := ∅, L := ∅, U := N;

    /* setup for starting at node s */
    d_s := 0;
    move node s from set U to set L;

    while (the set L is non-empty) do
    {
        select node i ∈ L such that d_i ≤ d_j for all j ∈ L;
        move node i from set L to set P;

        for (all nodes j with (i,j) ∈ A) do
        {
            if (d_j > l_(i,j)) then
            {
                d_j := l_(i,j);
                pred(j) := i;
                if (j ∈ U) then move j from set U to set L;
            }
        }
    }
} (END of 'prim')
```

Solver-Specific Input:

s — start node for searching the network

Solver Animation Sets:

Trial **(red):** Nodes $i \in L$ and edges $(i, \text{pred}(i))$ for nodes $i \in L$ such that $\text{pred}(i) \neq$ `null`.

Acquired **(green):** Nodes $i \in P$ and edges $(i, \text{pred}(i))$ for nodes $i \in P$ such that $\text{pred}(i) \neq$ `null`.

Discarded **(yellow):** An edge $(i, j) \in A$ such that $i \in P$ and $j \in L$ (or vice versa) is "discarded" if it is considered as a candidate predecessor edge for node j but is rejected because either

(i) at the time (i, j) is considered, node j already has a predecessor edge of length no greater than the length of (i, j), i.e., $l_{(\text{pred}(j),j)} \leq l_{(i,j)}$), or

(ii) a predecessor edge of shorter length is later found, before node j is permanently labeled.

Label Information:

`Edge labels:` Edge length l_e for each edge $e \in A$.

`Node labels:` The length $d_i \equiv l_{(\text{pred}(i),i)}$ of the predecessor edge $(\text{pred}(i), i)$ from each node $i \in N$ to the connected component R. Nodes with $d_i = \infty$ are labeled "-".

Feasible Result Information:

Minimum Spanning Tree **(orange):** Used to color the edges in a minimum spanning tree of G. The tree length is reported in the network window status line.

Infeasible Result Information:

Minimum Spanning Tree on R **(orange):** A minimum spanning tree for the component R of the network that contains the specified starting node s. The length of the minimum spanning tree on component R is reported in the network window status line.

3.2 Shortest Path Problem

Given a directed network $\mathcal{N} = (G, l, s)$, the *shortest path problem* is to find a set of dipaths from node s to each node $i \in N \backslash \{s\}$ with minimal length, where the length of an s-i dipath P_{si} for $i \in N \backslash \{s\}$ is calculated as $\sum_{e \in P_{si}} l_e$.

`Input:` A directed network $\mathcal{N} = (G, l, s)$, where $l \in \mathbb{R}^A$ and $s \in N$.

`Formulation:`

For each node $i \in N \backslash \{s\}$,
min $\sum_{e \in P_{si}} l_e$
s.t. P_{si} is an s-i dipath in G.

`Feasible Output:` A shortest path tree T^* that includes a shortest path P^*_{si} for each $i \in N \backslash \{s\}$, the tree's total length $\sum_{e \in T^*} l_e$, the shortest path lengths to each node $i \in N$, and the sum of the shortest path lengths, $\sum_{i \in N} \sum_{e \in P^*_{s,i}} l_e$.

`Infeasible Output:` We return one of the following: (1) A proper subset R of N, such that $s \in R$ and there are no edges $(i, j) \in A$ such that $i \in R$ and $j \notin R$; in this case, we also give the shortest path tree from s to all nodes in R. (2) A negative length cycle C in the component of G containing s; in this case we also give the length of the cycle, $\sum_{e \in C} l_e < 0$.

Required Problem Input

GIDEN solvers derived from the `ShortestPath` problem base class will request the following information:

"Length" — edge array of integer data type;
provides edge lengths l_e for each $e \in A$.

s — start node for searching the network

The "Length" data field is requested in the solver input dialog, and the source node s is requested at the start of solver execution.

3.2.1 Dijkstra's Algorithm

```
algorithm dijkstra;
{
    /* initialize algorithm information */
    d_i := ∞ and pred(i) := null for all i ∈ N;
    P := ∅, L := ∅, U := N;

    /* setup for starting at node s */
    d_s := 0;
    move node s from set U to set L;

    while (the set L is non-empty) do
    {
        select node i ∈ L such that d_i ≤ d_j for all j ∈ L;
        move node i from set L to set P;

        for (all nodes j with (i,j) ∈ A) do
        {
            if (d_j > d_i + l_(i,j)) then
            {
                d_j := d_i + l_(i,j);
                pred(j) := i;
                if (j ∈ U) then move j from set U to set L;
            }
        }
    }
} (END of 'dijkstra')
```

Solver-Specific Input:

Additional requirement that the selected "Length" input array can contain only non-negative values.

Solver Animation Sets:

Trial **(red):** Nodes $i \in L$ and edges $(i, \text{pred}(i))$ for nodes $i \in L$ such that $\text{pred}(i) \neq$ `null`.

Acquired **(green):** Nodes $i \in P$ and edges $(i, \text{pred}(i))$ for nodes $i \in P$ such that $\text{pred}(i) \neq$ `null`.

Discarded **(yellow):** An edge $(i, j) \in A$ such that $i \in P$ and $j \in L$ is "discarded" if it is considered as a candidate predecessor edge for node j but is either

(i) rejected immediately, because an $s - j$ dipath has already been found that is at least as short as the $s - j$ dipath through (i, j), i.e., $d_j \leq d_i + l_{(i,j)}$, or

(ii) rejected later, because a shorter $s - j$ dipath is found before node j is permanently labeled.

Label Information:

`Edge labels:` Edge length l_e for each edge $e \in A$.

`Node labels:` Current "node distance" d_i from the starting node s. Nodes with $d_i = \infty$ are labeled "-". Node s is labeled "s" initially and then with its "distance label," $d_s = 0$.

Feasible Result Information:

Shortest Path Tree **(orange):** Used to color a shortest path tree rooted at s and spanning the nodes of G. The total tree length, along with the sum of (finite) path lengths is reported in the network window status line. The node labels report the final distances d_i for each $i \in N$.

Infeasible Result Information:

Shortest Path Tree **(orange):** Used to color a shortest path tree rooted at s for the component R of nodes and edges that are di-path reachable from s. This tree length, along with the sum of (finite) path lengths is reported in the network window status line.

3.2.2 FIFO Label Correcting Algorithm

```
algorithm fifo label-correcting;[a]
{
    d_s := 0 and pred(s) := null;
    d_j := ∞ for each node j ∈ N\{s};
    QUEUE := {s};
    while (QUEUE not empty) do
    {
        remove a node i from QUEUE;
        for (each edge (i,j) ∈ A(i)) do
        {
            if (d_j > d_i + l_(i,j)) then
            {
                d_j := d_i + l_(i,j);
                pred(j) := i;
                if (j ∉ QUEUE) then add j to QUEUE;
            }
        }
    }
} (END of 'fifo label-correcting')
```

[a] See Chapter 5, Sections 3-5 of [1] for a description of the FIFO label correcting algorithm as presented here.

Solver-Specific Input: (none)

Solver Animation Sets:

Accepted **(green):** Nodes $i \in N$ such that $d_i < \infty$, and edges $(\text{pred}(i), i)$ for each i such that $i \neq s$ (whose predecessor node is undefinied).

Current **(red):** The node i taken from the QUEUE whose outgoing edges are being examined, and the edge that is currently being examined.

Discarded **(yellow):** An edge $(i, j) \in A$ such that $i \in P$ and $j \in L$ (or vice versa) is "discarded" if it is considered as a candidate predecessor edge for node j but is rejected because either

(i) node j already has a predecessor edge of length no greater than the length of (i, j), i.e., $l_{(\text{pred}(j),j)} \leq l_{(i,j)}$), or

(ii) a predecessor edge of shorter length is later found before node j is permanently labeled.

Label Information:

`Edge labels:` Edge length l_e for each edge $e \in A$.

`Node labels:` Current "node distance" from the acquired component, d_i. Nodes with $d_i = \infty$ are labeled "-". Node s is labeled "s"; its "distance" is equal to zero.

Feasible Result Information:

Shortest Path Tree **(orange):** Used to color a shortest path tree rooted at s and spanning the nodes of G. The total tree length, along with the sum of (finite) path lengths is reported in the network window status line. The node labels report the final distances d_i for each $i \in N$.

Infeasible Result Information:

Shortest Path Tree **(orange):** Used to color a shortest path tree rooted at s for the component R of nodes and edges that are di-path reachable from s. This tree length, along with the sum of (finite) path lengths is reported in the network window status line.

Negative-Cost Cycle **(blue):** Used to color a directed cycle in the network with negative total length (this is known as a "negative cost cycle"). The length (cost) of the cycle is reported in the network window status line.

Note that the infeasible result information that is reported depends on the component R of nodes and edges that are di-path reachable from s. If this component contains a negative cost cycle, then this cycle will be reported. Otherwise, only the shortest path tree on component R will be reported — even if there is a negative cost cycle present elsewhere in the network.

3.3 Maximum Flow Problem

Given a directed network $\mathcal{N} = (G, u, s, t)$, the *maximum flow problem* is to find a feasible s-t flow of maximal value, where the value of a feasible flow x is calculated as $\sum_{\{i:(i,t)\in A\}} x_{(i,t)}$. We say a flow x is *feasible* for the maximum flow problem if it satisfies the bound constraints, $0 \leq x \leq u$, and the balance constraints:

$$\sum_{\{j:(i,j)\in A\}} x_{(i,j)} = \sum_{\{j:(j,i)\in A\}} x_{(j,i)} \qquad \text{for each node } i \in N\backslash\{s,t\}.$$

`Input:` A network $\mathcal{N} = (G, u, s, t)$, where $u \in \mathbb{Z}_+^A$ and $s, t \in N$.

`Formulation:`

$$\begin{array}{lll} \max & \sum_{\{i:(i,t)\in A\}} x_{(i,t)} & \\ \text{s.t.} & \sum_{\{j:(i,j)\in A\}} x_{(i,j)} = \sum_{\{j:(j,i)\in A\}} x_{(j,i)} & \text{for each node } i \in N\backslash\{s,t\}. \\ & 0 \leq x \leq u & \end{array}$$

`Feasible Output:` A maximum flow x^*, the maximum flow value $\sum_{\{i:(i,t)\in A\}} x^*_{(i,t)}$, and an s-t cut $\delta(S, \overline{S})$ such that $s \in S$, $t \in \overline{S}$ and $\sum_{\{(i,j)\in A: i\in S, j\in\overline{S}\}} u_{(i,j)} = \sum_{\{i:(i,t)\in A\}} x^*_{(i,t)}$.

`Infeasible Output:` *The maximum-flow problem, as we have described it, is never infeasible since the flow $x = 0$ satisfies the bound constraints and the balance constraints for all problem instances.*

Required Problem Input

GIDEN solvers derived from the `MaxFlow` problem base class will request the following information:

"Capacity"	—	edge array of integer data type; provides edge capacities u_e for each $e \in A$.
s	—	source node
t	—	sink node

The "Capacity" data field is requested in the solver input dialog. Source node s and sink node t are requested at the start of solver execution.

3.3.1 Generic Augmenting Path Algorithm

```
algorithm generic augmenting path;[a]
{
    x := 0;
    while (G^x contains an s–t dipath) do
    {
        identify an s–t dipath P in G^x;
        calculate δ := min_{(i,j)∈P} u^x_(i,j);
        augment δ units of flow along P and update G^x;
    }
} (END of 'generic augmenting path')
```

[a] See Chapter 6, Section 4 of [1] for a description of the generic augmenting path algorithm as presented here.

Solver-Specific Input:

The user is asked to specify a search method for finding an augmenting path. The current implementation supports only two choices: "breath-first search" (BFS) and "depth-first search (DFS). This input is requested through the network window status line.

Solver Animation Sets:

Trial **(red):** Nodes and edges being considered for inclusion in an augmenting path of G.

Acquired **(green):** Nodes and edges in the current augmenting path of G.

Discarded **(yellow):** Edges that are encountered during the search for an augmenting path but are disregarded because their ends have both already been "seen" during the course of the search, or because no residual capacity is available.

Label Information:

`Edge labels:` Labels on the edges $e \in A$ give the current flow and the remaining capacity: $(x_e, u_e - x_e)$

`Node labels:` Source node s is labeled "s", and sink node t is labeled "t"; other nodes have empty labels.

Feasible Result Information:

Reachable Nodes **(orange):** Used to color the set R of nodes that are reachable from s in G^x at the end of solver execution.

Minimum Capacity Cut **(black):** Used to color the edges in the minimum cut $\delta(R, N \backslash R)$ at the end of solver execution.

The final flow value and the minimum cut capacity are displayed in the status line.

3.3.2 Shortest Augmenting Path Algorithm

```
algorithm shortest augmenting path;[a]
{
    x := 0;
    obtain the exact distance labels d_i for each node i ∈ N;[b]
    i := s;
    while (d_s < |N|) do
    {
        if (i has an admissible edge[c]) then
        {
            advance(i);
            if (i = t) then
            {
                augment;
                i := s;
            }
        }
        else retreat(i);
    }
} (END of 'shortest augmenting path')

procedure advance(i);
{
    let (i, j) be an admissible edge in A^x(i);
    pred(j) := i and i := j;
}

procedure retreat(i);
{
    d_i := min_{{(i,j)∈A^x(i):u^x_(i,j)>0}} d_j + 1;
    if (i ≠ s) then i := pred(i);
}

procedure augment;
{
    using the pred indices, identify the augmenting s–t path P;
    δ := min_{{(i,j)∈P}} u^x_(i,j);
    augment δ units of flow along path P;
}
```

[a] See Chapter 7, Section 4 of [1] for a description of the shortest augmenting path algorithm as presented here.

[b] For each node $i \in N \backslash \{t\}$, $d_i :=$ the minimum number of edges in an i-t dipath in G.

[c] An edge (i, j) in G^x is *admissible* if $d_j = d_i + 1$ and $u^x_{(i,j)} > 0$.

Solver-Specific Input:

There is a solver option to use "smart label updates"; this option, if selected, incorporates into the solver an improved approach for updating distance labels as suggested on pp. 219–220 in [1].

Solver Animation Sets:

Trial **(red):** Nodes and edges being considered for inclusion in an augmenting s-t dipath of G^x.

Acquired **(green):** Nodes and edges in the current augmenting s-t dipath of G^x.

Discarded **(yellow):** Inadmissible edges that are encountered during the search for an augmenting s-t dipath. Nodes whose distance labels have changed during the current search for an augmenting s-t dipath.

Label Information:

`Edge labels`: Labels on the edges $e \in A$ give the current flow and the remaining capacity: $(x_e, u_e - x_e)$

`Node labels`: Current distance labels d_i for nodes $i \in N \backslash \{t\}$. Sink node t is labeled "t" $(d_t = 0)$.

Feasible Result Information:

Reachable Nodes **(orange):** Used to color the set R of nodes that are reachable from s in G^x at the end of solver execution. set

Minimum Capacity Cut **(black):** Used to color the edges in the minimum cut $\delta(R, N \backslash R)$ at the end of solver execution.

The final flow value and the minimum cut capacity are displayed in the status line.

3.3.3 Preflow-Push Algorithm

```
algorithm preflow-push;[a]
{
    preprocess;
    while (network contains an active node[b]) do
    {
        select an active node i;
        push/relabel(i);
    }
} (END of 'preflow-push')

procedure push/relabel(i);
{
    if (network contains and admissible edge (i,j)[c]) then
    {
        δ := min{e_i, u^x_(i,j)};
        push δ units of flow from node i to node j;
    }
    else
        replace d_i by min_{j:(i,j)∈A^x(i), u^x_(i,j)>0} d_j + 1;
}

procedure preprocess;
{
    x := 0;
    compute exact distance labels d_i;[d]
    x_(s,j) := u_(s,j) for each edge (s,j) ∈ A(s);
    d_s := n;
}
```

[a] See Chapter 7, Sections 6-8 of [1] for a description of the preflow-push algorithm as presented here.
[b] Node $i \in N$ is *active* if $e_i \equiv \sum_{\{j:(j,i)\in A\}} x_{(j,i)} > \sum_{\{j:(i,j)inA\}} x_{(i,j)}$.
[c] An edge (i,j) in G^x is *admissible* if $d_j = d_i + 1$ and $u^x_{(i,j)} > 0$.
[d] For each node $i \in N\backslash\{t\}$, $d_i :=$ the minimum number of edges in an i-t dipath in G.

Solver-Specific Input: (none)

Solver Animation Sets:

Current **(red):** The selected active node i and the current admissible edge incident to i.

Active **(blue):** All nodes $j \in N\backslash\{s,t\}$ with $e_j > 0$. (note: if advanced termination is selected, this is all nodes with $e_j > 0$ and $d_j < |N|$.)

Label Information:

`Edge labels`: Labels on the edges $e \in A$ give the current flow and the remaining capacity: $(x_e, u_e - x_e)$

`Node labels`: Current distance labels d_i for nodes $i \in N\backslash\{t\}$. Sink node t is labeled "t" $(d_t = 0)$.

Feasible Result Information:

Reachable Nodes **(orange):** Used to color the nodes in the set S at the end of solver execution.

Minimum Capacity Cut **(black):** Used to color the edges in the cut $\delta(S, \bar{S})$ at the end of solver execution.

The final flow value and the minimum cut capacity are displayed in the status line.

Implementation Notes: The solver implementation follows the "highest-label" rule described in [1] for selecting an active node i. An advanced termination option is available; this option relabels some nodes $i \in N$ with positive excess e_i by setting $d_i := |N|$ until an optimal preflow is found, then the preflow is resolved as described in [1].

3.4 Minimum Cost Flow Problem

Given a directed network $\mathcal{N} = (G, c, u, b)$, the *minimum-cost flow problem* is to find a feasible flow of minimal cost, where the cost of a flow x is calculated as $\sum_{e \in A} c_e x_e$. We say a flow x is feasible for the minimum-cost flow problem if it satisfies the bound constraints, $0 \leq x \leq u$, and the balance constraints:

$$\sum_{\{j:(i,j)\in A\}} x_{(i,j)} - \sum_{\{j:(j,i)\in A\}} x_{(j,i)} = b_i \qquad \text{for each node } i \in N.$$

A node $i \in N$ is called a *supply node* if $b_i > 0$, a *demand node* if $b_i < 0$, or a *transshipment node* if $b_i = 0$. For a supply node i, we say that the amount of supply available at i is b_i units. For a demand node i, we say that the amount of demand at i is $|b_i|$ units.

`Input:` A network $\mathcal{N} = (G, c, u, b)$, where $c \in \mathbb{Z}^A$, $u \in \mathbb{Z}_+^A$ and $b = \mathbb{Z}^N$.[1]

`Formulation:`

$$\begin{array}{ll} \min & \sum_{e \in A} c_e x_e \\ \text{s.t.} & \sum_{\{j:(i,j)\in A\}} x_{(i,j)} - \sum_{\{j:(j,i)\in A\}} x_{(j,i)} = b_i \qquad \text{for each node } i \in N \\ & 0 \leq x \leq u \end{array}$$

`Feasible Output:` A minimum cost flow x^*, the flow cost $\sum_{e \in A} c_e x_e^*$, and node potentials $\pi^* \in \mathbb{R}^N$ such that $c_{(i,j)} - \pi_i^* + \pi_j^* \geq 0$ for all edges $(i, j) \in A$.

`Infeasible Output:` A partition of the nodes into two sets, S and D, such that the capacity of the cut $\delta(S, D)$ is less than the net supply of nodes in S:

$$\sum_{\{(i,j)\in A: i\in S, j\in D\}} u_{(i,j)} \; < \; \sum_{i \in S} b_i$$

`Assumptions:` $\sum_{i \in N} b_i = 0$

Required Problem Input

GIDEN solvers derived from the `MinCostFlow` problem base class will request the following information:

"Capacity" — edge array of integer data type;
provides edge capacities u_e for each $e \in A$.

"Cost" — edge array of integer data type;
provides edge costs c_e for each $e \in A$.

"Supply" — node array of integer data type;
provides node supply b_i for each $i \in N$.

The "Capacity," "Cost," and "Supply" data fields are requested in the solver input dialog.

[1] Note that the solvers currently included in GIDEN require finite and integer-valued costs as well capacities. For the minimum cost flow problem in general, the cost vector c does not need to be integer-valued (i.e., $c \in \mathbb{R}^A$), and the capacity vector u may include unbounded values (i.e., $u_e = \infty$).

3.4.1 Capacity Scaling

```
algorithm capacity scaling;
{
    x := 0 and π := 0;
    δ := 2^⌊log2 U⌋;
    while (δ ≥ 1) do
    {
        for (every arc (i,j) in G^x) do
        {
            if (r_ij ≥ δ and c^π_ij < 0) then
                send r_ij units of flow along (i,j), updating x_ij, e_i, and e_j;
        }

        E := {i ∈ N | e_i ≥ δ};
        D := {i ∈ N | e_i ≤ −δ};
        while (E ≠ ∅) do
        {
            select a node k ∈ E;
            if (a δ-capacity path exists in G^x(δ) from k to any node in D) then
            {
                let d(l) be the shortest path distance in G^x(δ) from
                    k to a node l ∈ D with respect to reduced costs c^π; [a]
                let P denote a shortest path from node k to node l in G^x(δ);
                for (each node i that was permanently labeled[b]) do
                    π_i := π_i − d_i + d_l;
                augment δ units of flow along the path P;
                update x, E, D, and G^x(δ);
            }
            else remove k from E;
        }
        δ := δ/2;
    }
} (END of 'capacity scaling')
```

[a] Dijkstra's algorithm used to find shortest path distances.
[b] Nodes are permanently labeled while computing node distances d_i.

Solver-Specific Input: (none)

Solver Animation Sets:

Path **(blue):** Nodes and edges in path from an excess node k to a demand node l.

Label Information:

`Edge labels:` Labels on the edges $e \in A$ give the current flow, remaining capacity, and cost: $(x_e, u_e - x_e, c_e)$

`Node labels:` Labels on the nodes $i \in N$ give the supply and current potential: b_i, π_i

Feasible Result Information:

Flow between Bounds **(orange):** Used to color the edges $e \in A$ such that $0 < x_e < u_e$ in the final flow.

Flow at Capacity **(red):** Used to color the edges that are at capacity in the final flow. These are the edges $e \in A$ such that $x_e = u_e$.

The edges not included in either of these final sets have zero flow in the final solution (i.e., $x_e = 0$); these unused edges are drawn in the default color.

An optimal minimum cost flow is displayed, and the total cost of the flow is reported in the network window status line. The potentials reported in the node labels can be used as a certificate of optimality.

Infeasible Result Information:

Excess Reachable Nodes **(orange):** Used to color a set S of nodes with positive net supply. The net supply of this set of nodes exceeds the capacity of the implied cut $\delta(S, D)$, where $D \equiv N \backslash S$.

Infeasibility Cut **(black):** Used to color the edges in the cut $\delta(S, D)$.

When a problem instance is reported as infeasible, node labels display the supply available at each node.

Implementation Notes: During the δ-phase preprocessing step, edges with sufficient residual capacity and appropriate reduced costs will be temporarily colored red when executing the solver in step mode or pseudocode mode.

3.4.2 Cycle Canceling

```
algorithm cycle canceling;
{
    establish a feasible flow x, if one exists;
    while (the residual network contains a negative-cost cycle) do
    {
        identify a negative-cost cycle;
        calculate the residual capacity of the cycle;
        augment flow along the cycle;
    }
} (END of 'cycle canceling')
```

Solver-Specific Input: (none)

Solver Animation Sets:

Cycle **(blue):** Used to color nodes and edges in the current negative-cost cycle. When this animation set is being traced, the cycle will be drawn, and the cycle capacity and cost will be reported in the network window status line. Selecting the Trace button again will then highlight the bottleneck arc in the cycle (using the color red), and report the new flow cost in the status line.

Label Information:

`Edge labels:` Labels on the edges $e \in A$ give the current flow, remaining capacity, and cost: $(x_e, u_e - x_e, c_e)$

`Node labels:` Labels on the nodes $i \in N$ give the supply: b_i

Feasible Result Information:

Flow between Bounds **(orange):** Used to color the edges $e \in A$ such that $0 < x_e < u_e$ in the final flow.

Flow at Capacity **(red):** Used to color the edges that are at capacity in the final flow. These are the edges $e \in A$ such that $x_e = u_e$.

The edges not included in either of these final sets have zero flow in the final solution (i.e., $x_e = 0$); these unused edges are drawn in the default color.

An optimal minimum cost flow is displayed, and the total cost of the flow is reported in the network window status line.

Infeasible Result Information:

Excess Reachable Nodes **(orange):** Used to color a set S of nodes with positive net supply. The net supply of this set of nodes exceeds the capacity of the implied cut $\delta(S, D)$, where $D \equiv N \backslash S$.

Infeasibility Cut **(black):** Used to color the edges in the cut $\delta(S, D)$.

When a problem instance is reported as infeasible, node labels display the supply available at each node.

3.4.3 Out-of-Kilter

```
algorithm out-of-kilter;
{
    π := 0;
    establish a feasible flow x, if one exists;
    while (the residual network contains an out-of-kilter edge[a]) do
    {
        select an out-of-kilter edge (p,q) in G^x;
        define the length of each edge (i,j) in G^x as max{0, c^π_ij};
        let d be the shortest path distances for q to all other nodes
            in G^x\{(q,p)}, and let P be the shortest path from q to p;
        if (d_p == ∞) then
            update π_i := π_i − c^π_pq for all nodes reachable from q;
        else
            update π_i := π_i − d_i for all nodes reachable from q;
        if (c^π_pq < 0) then
        {
            let W be the cycle obtained by adding edge (p,q) to path P;
            calculate the residual capacity of cycle W;
            augment flow along W;
            update x, G^x, and the reduced costs;
        }
    }
} (END of 'out-of-kilter')
```

[a] An edge e is said to be "out-of-kilter" if it does not satisfy the complementary slackness conditions. These conditions require that $c_e^\pi \geq 0$ when $x_e = 0$, $c_e^\pi = 0$ when $x_e \in (0, u_e)$, and $c_e^\pi \leq 0$ when $x_e = u_e$.

Solver-Specific Input: (none)

Solver Animation Sets:

Kilter Edge **(red):** Currently selected out-of-kilter edge, (p, q).

Cycle **(blue):** The cycle W obtained by adding the out-of-kilter edge (p, q) to the path P.

Label Information:

`Edge labels:` Labels on the edges $e \in A$ give the current flow, remaining capacity, and cost: $(x_e, u_e - x_e, c_e)$

`Node labels:` Labels on the nodes $i \in N$ give the supply and current potential: b_i, π_i

Feasible Result Information:

Flow between Bounds **(orange):** Used to color the edges $e \in A$ such that $0 < x_e < u_e$ in the final flow.

Flow at Capacity **(red):** Used to color the edges that are at capacity in the final flow. These are the edges $e \in A$ such that $x_e = u_e$.

The edges not included in either of these final sets have zero flow in the final solution (i.e., $x_e = 0$); these unused edges are drawn in the default color.

An optimal minimum cost flow is displayed, and the total cost of the flow is reported in the network window status line. The potentials reported in the node labels can be used as a certificate of optimality.

Infeasible Result Information:

Excess Reachable Nodes **(orange):** Used to color a set S of nodes with positive net supply. The net supply of this set of nodes exceeds the capacity of the implied cut $\delta(S, D)$, where $D \equiv N \backslash S$.

Infeasibility Cut **(black):** Used to color the edges in the cut $\delta(S, D)$.

When a problem instance is reported as infeasible, node labels display the supply available at each node.

3.4.4 Network Simplex Algorithm

```
algorithm network simplex;
{
    determine an initial network simplex tree structure T;
    calculate the initial flow x and node potentials π;

    while (some edge not in T violates optimality conditions) do
    {
        select an entering edge e that violates its optimality conditions;
        determine a leaving edge that results from adding e to T;
        update flow x and tree structure T;
        if (entering edge ≠ leaving edge) then
            update tree indices and node potentials π;
    }
} (END of 'network simplex')
```

Solver-Specific Input: (none)

Solver Animation Sets:

Tree **(green):** Used to color the nodes in the current tree T.

Entering Edge **(red):** Used to color the selected entering edge.

Leaving Edge **(yellow):** Used to color the selected leaving edge;

Label Information:

`Edge labels:` Labels on the edges $e \in A$ give the current flow, remaining capacity, and cost: $(x_e, u_e - x_e, c_e)$

`Node labels:` Labels on the nodes $i \in N$ give the supply and current potential: b_i, π_i

Feasible Result Information:

Flow between Bounds **(orange):** Used to color the edges $e \in A$ such that $0 < x_e < u_e$ in the final flow.

Flow at Capacity **(red):** Used to color the edges that are at capacity in the final flow. These are the edges $e \in A$ such that $x_e = u_e$.

The edges not included in either of these final sets have zero flow in the final solution (i.e., $x_e = 0$); these unused edges are drawn in the default color.

An optimal minimum cost flow is displayed, and the total cost of the flow is reported in the network window status line. The potentials reported in the node labels can be used as a certificate of optimality.

Infeasible Result Information:

Excess Reachable Nodes **(orange):** Used to color a set S of nodes with positive net supply. The net supply of this set of nodes exceeds the capacity of the implied cut $\delta(S, D)$, where $D \equiv N \backslash S$.

Infeasibility Cut **(black):** Used to color the edges in the cut $\delta(S, D)$.

When a problem instance is reported as infeasible, node labels display the supply available at each node.

Implementation Notes:

1. The algorithm implementation adds an artificial root node and auxilary edges, which are not drawn on the network window canvas. Since these nodes and edges are not drawn, it is possible that the tree structure T will appear to have too few edges.
2. The tree structure maintained throughout execution is a strongly feasible tree structure. The leaving edge is selected according to the description in [1].

3.4.5 Successive Shortest Paths Algorithm

algorithm successive shortest path;[a]
{
  $x := 0$ and $\pi := 0$;
  $e_i := b_i$ for each $i \in N$;
  initialize the sets $E := \{i \in N : e_i > 0\}$ and $D := \{i \in N : e_i < 0\}$
  while $(|E| > 0)$ **do**
  {
    select a node $k \in E$;
    determine the shortest dipath distance d_l from k to
      a node $l \in D$ in G^x with respect to reduced costs $c^x(\pi)$;[b]
    let P denote a shortest path from node k to node l;
    for each node i that was permanently labeled, $\pi_i := \pi_i - d_i + d_l$;
    $\delta := \min\{e_k, -e_l, \min_{\{(i,j)\in P\}} u^x_{(i,j)}\}$;
    augment δ units of flow along path P;
    update x, G^x, E, D, and the reduced costs $c(\pi)$;
  }
} (END of 'successive shortest path')

[a] See Chapter 9, Section 7 of [1] for a description of the successive shortest path algorithm as presented here.
[b] For $e = (i,j) \in A$, the reduced cost $c^x_{(i,j)}(\pi) = c_{(i,j)} - \pi_i + \pi_j$ if $(i,j) \in A^x$, and $c^x_{(j,i)}(\pi) = -c_{(i,j)} - \pi_j + \pi_i$ if $(j,i) \in A^x$.

Solver-Specific Input: (none)

Solver Animation Sets:

Path **(blue):** Nodes and edges in path from an excess node k to a demand node l.

Path Bottleneck **(red):** Edges in the k-l path are *bottleneck edges* if the amount of flow sent from k to l has been limited by the residual capacity of these edges.

Label Information:

`Edge labels:` Labels on the edges $e \in A$ give the current flow, remaining capacity, and cost: $(x_e, u_e - x_e, c_e)$

`Node labels:` Labels on the nodes $i \in N$ give the supply and current potential: b_i, π_i

Feasible Result Information:

Flow between Bounds **(orange):** Used to color the edges $e \in A$ such that $0 < x_e < u_e$ in the final flow.

Flow at Capacity **(red):** Used to color the edges that are at capacity in the final flow. These are the edges $e \in A$ such that $x_e = u_e$.

The edges not included in either of these final sets have zero flow in the final solution (i.e., $x_e = 0$); these unused edges are drawn in the default color.

An optimal minimum cost flow is displayed, and the total cost of the flow is reported in the network window status line. The potentials reported in the node labels can be used as a certificate of optimality.

Infeasible Result Information:

Excess Reachable Nodes **(orange):** Used to color a set S of nodes with positive net supply. The net supply of this set of nodes exceeds the capacity of the implied cut $\delta(S, D)$, where $D \equiv N \backslash S$.

Infeasibility Cut **(black):** Used to color the edges in the cut $\delta(S, D)$.

When a problem instance is reported as infeasible, node labels display the supply available at each node.

Chapter 4

Developing Solvers

In this chapter we give an overview of the process for developing GIDEN solvers. The first section covers basic topics, such as how to obtain the implementor distribution, the anatomy of a solver, and the process of linking a solver through the `Solvers` menu. The second section is a tutorial for developing a solver implementation of Kruskal's algorithm for the minimum spanning tree problem. This section starts with a pseudocode description of the algorithm, and gradually steps through the process of developing a functional solver. The final section of this chapter builds on the tutorial example, and introduces a few "advanced" features that can be added to your solvers. The topics covered in this last section include visually displaying solver result information, using a solver input dialog, and executing solvers as subroutines.

4.1 GIDEN Solver Basics

4.1.1 The Implementor Distribution

The implementor version is distributed in a ZIP file that is downloaded from:

`http://users.iems.nwu.edu/~dilworth/giden-faq.html`

The distribution contains the files needed to develop GIDEN solvers. Installation instructions are included in the ZIP file. Note this guide is not included in the implementor version, but it can also be downloaded from the same web site. Java tools need to be installed on the computer prior to developing solvers.

4.1.2 Anatomy of a Solver

Every GIDEN solver is derived from the `ExecBase` base class, which provides the fundamental framework for interacting with the core environment. Each solver is expected to provide four public methods, which are used by the core environment to create and execute the solver. To describe the basic anatomy of a GIDEN solver, we will refer to the `EmptySolver` class presented in Figure 4.1.

The first line of `EmptySolver.java` is the class declaration. This line declares that this is a public class named "EmptySolver", which is derived from the `ExecBase` base class. Note that in Java, the

```
package giden.userSOLVERS;
import giden.CORE.*;
import giden.GRDS.*;

public class EmptySolver extends ExecBase
{
    EmptySolver(UserBase userinfo) {
        super(userinfo);
    }

    public boolean Setup(Network net) {
        return true;
    }

    public boolean Execute() {
        return true;
    }

    public boolean Shutdown() {
        return true;
    }
}
```

Figure 4.1: Code listing for `EmptySolver.java`

class name determines what the filename should be (including upper and lower case). In particular, the code for class "`EmptySolver`" should be saved in a file named "`EmptySolver.java`".

The remainder of the code listing consists of four methods, which are required in every GIDEN solver.[1] The first method is the class constructor, which is used to create an instance of the solver. The remaining three methods are used by the core environment to control the solver execution. Each of these methods are described below.

Note that the `EmptySolver` example doesn't actually do anything. The only value of `EmptySolver` is that it demonstrates the bare minimum structure of a GIDEN solver. The constructor method is an empty shell that executes the base class constructor; the `Setup()`, `Execute()`, and `Shutdown()` methods return immediately without performing any operations. In Section 4.2 we will develop a working example of a solver whose methods perform additional operations.

Constructor Method

The constructor method has the same name as the solver class, and takes a single parameter of type `UserBase`. The `UserBase` object that is passed to the constructor provides several methods for communicating with the user interface (see the html documentation in the `doc` folder). The constructor method typically performs the following operations:

1. Invokes the base class constructor. (Required)
2. Saves a copy of the `UserBase` object for local use.
3. Initializes the solver's animation sets.

[1] Some of these methods may be omitted, in which case the corresponding base class methods will be executed.

The first operation in the solver constructor is to invoke the base class constructor, by executing the `super()` method and passing the `UserBase` object. This operation, which is required, registers the `UserBase` object with the base class so that it can be used throughout solver execution.

In addition to invoking the base class constructor, the solver constructor may perform a variety of solver-specific operations. Items 2 and 3 above are only examples of solver-specific operations that may be included in the constructor method. The kind of operations that should usually be included in the constructor are those operations that initialize solver variables or other data items that do not change between consecutive executions of a solver.

`Setup()` Method

The `Setup()` method is executed by the core environment each time the user makes a request to run the solver. In particular, this method is executed when the solver is selected from the network window's Solvers menu, and each time the user selects the Reset action button when the solver is active. The `Setup()` method typically performs the following operations:

1. Saves a copy of the `Network` object for local use.
2. Sets the "directed state" of the `Network` object.
3. Creates a solver input dialog.
4. Initializes the node and edge labels.

The `Setup()` method is passed a single parameter of type `Network` (see [4]). The passed `Network` instance corresponds to the network currently displayed in the window. This is the base network that is used as input for the solver algorithm.

The return value of `Setup()` is a boolean that indicates whether or not the operations of the method were performed successfully. If this method returns "false", then the core environment will abort solver execution and return the network window to edit mode.

`Execute()` Method

The `Execute()` method is invoked by the core environment each time the solver is run. It is executed immediately following a successful return from the `Setup()` method. The `Execute()` method typically performs the following operations:

1. Creates and initializes any solver-specific data structures. (E.g., node and edge arrays that the solver maintains for transient use during solver execution.)
2. Executes the algorithm logic of the solver.

The return value of `Execute()` is a boolean that indicates whether or not the solver was able to complete execution. This value is ignored in the current version of GIDEN, but it is reserved for future use.

`Shutdown()` Method

The `Shutdown()` method is invoked by the core environment after a solver has completed execution. It is executed immediately following the `Execute()` method. The `Shutdown()` method typically performs the following operations:

1. Saves any result information from the solver.[2]

2. Resets any transient data values that should not carry over to subsequent solver executions.

The return value of `Shutdown()` is a boolean that indicates whether or not operations of the method were performed successfully. This value is ignored in the current version of GIDEN, but it is reserved for future use.

4.1.3 The Obligatory `HelloWorld` Example

The `EmptySolver` example in the previous section is utterly useless except for illustrating the basic components of a GIDEN solver. In the name of tradition, we will now proceed with the customary "hello world" example, which is only slightly more interesting. The code listing for our `HelloWorld` solver is given in Figure 4.2.

```
package giden.userSOLVERS;
import giden.CORE.*;
import giden.GRDS.*;

public class HelloWorld extends ExecBase
{
    private UserBase UI;

    public HelloWorld(UserBase userinfo) {
        super(userinfo);
        UI = userinfo;
    }

    public boolean Execute() {
        UI.StatusLine("Hello world!");

        return true;
    }
}
```

Figure 4.2: Code listing for `HelloWorld.java`

The `HelloWorld` solver is probably the simplest GIDEN solver possible that actually interacts with the user. `HelloWorld` does not request any user input, perform any operations on the network, create any animation sets, or display a pseudocode window. It only writes the text string "Hello world!" in the network window status line.

There are a few notable differences between the structure of the `HelloWorld` and `EmptySolver` solvers:

1. `HelloWorld` includes a private class variable named `UI`. This variable is used to save a local copy of the `UserBase` object that is passed to the constructor method.

2. In the `Execute()` method, the `UI` variable is used to write the text string "Hello world!" in the network window status line.

[2]The current GIDEN release does not provide infrastructure support for saving solver results.

3. Observe that the `Setup()` and `Shutdown()` methods are missing from `HelloWorld`, even though we said just a couple of pages ago that these methods were required in all GIDEN solvers. (If you go back and check, you'll notice that we included a footnote that hedged a bit on this point.) In practice, since `HelloWorld` doesn't need to perform any operations in these two methods, you can simply omit them from the solver. The end result, is that `Setup()` and `Shutdown()` are still executed by the core environment, but in this case the actual methods that are executed are empty methods provided by the `ExecBase` base class.

The next topic in this section will describe how to link this `HelloWorld` solver into the GIDEN solver toolkit.

4.1.4 Linking a Solver to GIDEN

Before a solver is accessible through GIDEN, an entry must be created in the network window Solvers menu. The `SolverMenu` class is responsible for managing solvers that are not part of the GIDEN solver toolkit. There are two methods in the `SolverMenu` class that must be modified in order to link your solver through the Solvers menu: the `CreateMenu()` method and the `CreateSolver()` method.

The following steps describe how to modify the `SolverMenu` class file in order to add an entry for the `HelloWorld` solver:

1. Open the `SolverMenu.java` file for editing.
2. In the `SolverMenu.java` file, create a global `String` variable that gives the name of your new solver. For example, after the similar line for the `FeatureExample` solver, add the following text:

```
final String HELLO_WORLD = "My Hello World Solver";
```

3. In the `CreateMenu()` method, add the following text in order to set up a problem submenu and solver selection for the new solver:

```
AddProblem("My New Solvers");
AddSolver(HELLO_WORLD);
```

4. In the `CreateSolver()` method, you will need to check for the selection of your `HelloWorld` solver. To do this, add the following lines immediately after the similar lines that are given for the `FeatureExample` solver (and before the `return` call):

```
if (name.equals(HELLO_WORLD))
    exec = new HelloWorld(user);
```

5. Save the changes that you have made to the `SolverMenu.java` file, then recompile GIDEN.

After you recompile, you can execute GIDEN and run your new `HelloWorld` solver.

4.2 An Example Solver

So you've made it through the trivial `EmptySolver` and `HelloWorld` examples, and now you're ready to tackle a more complete (and useful) solver. Well, you've come to the right place. In this section, we describe a straightforward implementation of Kruskal's algorithm for the minimum spanning tree problem.[3] We develop the solver in several stages, starting with a pseudocode description

[3]The solver we develop here will differ somewhat from the solver included in the standard GIDEN distribution.

of the algorithm. While this tiered approach to solver development may not be ideal for your own coding style, it is useful for understanding the different components of the solver.

4.2.1 Kruskal's Algorithm

Consider a network $\mathcal{N} = (G, l)$, where $G = (N, A)$ is a connected undirected graph and l_e is the edge length for each edge $e \in A$. Given such a network $\mathcal{N}$, the minimum spanning tree problem is to find a minimal length subset of the edges $A' \subseteq A$ such that the graph $G' = (N, A')$ is connected and acyclic. One straightforward approach for solving the minimum spanning tree is known as Kruskal's algorithm. A pseudocode description of the algorithm is given below:

```
algorithm Kruskal (network 𝒩)
{
    mark all edges in A as unexamined;
    LIST := ∅;
    while (|LIST| < |N| − 1 and unexamined edges exist) do
    {
        e := a minimal length unexamined edge;
        mark edge e as examined;
        if (adding e to LIST does not create a cycle)
            add e to LIST;
    }
}   /* A′ == LIST */
```

4.2.2 Creating the Solver File and Linking to GIDEN

The first step in developing a GIDEN solver is to create a solver class file, and to link that file through the Solvers menu. Name your file `MyKruskal.java` and include the following text:

```
package giden.userSOLVERS;
import giden.CORE.*;
import giden.GRDS.*;

public class MyKruskal extends ExecBase
{
    MyKruskal(UserBase userinfo) {
        super(userinfo);
    }

    public boolean Setup(Network net) {
        return true;
    }

    public boolean Execute() {
        return true;
    }

    public boolean Shutdown() {
        return true;
    }
}
```

Now follow the directions in Section 4.1.4 to create an entry in the Solvers menu for your new Kruskal solver. For example, you could add an entry for the `MyKruskal` solver by editing the `SolverMenu.java` file as shown here (changes in black):

```
package giden.userSOLVERS;
import giden.CORE.*;
import giden.GRDS.*;

public class SolverMenu extends SolverMenuBase
{
    final String KRUSKAL_EXAMPLE="The Kruskal Example";
    ...

    SolverMenu(){ ... }

    public void CreateMenu() {
        ...

        AddProblem("Example Solvers");
        AddSolver(KRUSKAL_EXAMPLE);
    }

    public ExecBase CreateSolver(String name, SolverServices solvserv) {
        ...

        if (name.equals(KRUSKAL_EXAMPLE))
            exec = new MyKruskal(user);

        return exec;
    }
}
```

These changes will result in the creation of a new submenu in the network window's Solvers menu. This submenu will be named Example Solvers, and it will contain an entry named The Kruskal Example that will execute the `MyKruskal` solver. You may choose the name of the submenu and solver entry to suit your personal taste, but please note that *the name of the pseudocode file for the* `MyKruskal` *solver (see § 4.2.6) must exactly match the name assigned to the solver in this submenu with "txt" appended (e.g., "The Kruskal Example.txt").*

Recompile after making the above changes to `SolverMenu.java`. The next time you execute GIDEN, you should be able to select the new solver from the Solvers menu. At this point, nothing will happen when you try to execute the solver, because we have not yet included any algorithm logic in the `MyKruskal` class.

4.2.3 Adding the Algorithm Logic

Now that the basic solver framework is in place, the next step to update our solver is to add the algorithmic logic. A modified version of the file `MyKruskal.java` is given by the listing in Figure 4.3, where the method `ExecuteMyKruskalSolver` is replaced by the listing in Figure 4.4.

We now describe the changes in `MyKruskal.java` (shown in black type in the figures):

- We define several global variables for the solver: `UI`, `Net`, `Length`, `LIST`, and `ForestLength`. The variables `UI` and `Net` are used to store local copies of the `UserBase` and `Network` instances that are passed through the solver constructor and `Setup()` methods. The `Length` array is used to hold the network information that is required as an input for the minimum spanning tree problem. The `LIST` and `ForestLength` variables are used to store the result

```
package giden.userSOLVERS;
import giden.CORE.*;
import giden.GRDS.*;

public class MyKruskal extends ExecBase
{
    private UserBase UI;
    private Network Net;

    private EdgeArray Length;

    private LinkList LIST;
    private int ForestLength;

    MyKruskal(UserBase userinfo) {
        super(userinfo);
        UI = userinfo;
    }

    public boolean Setup(Network net) {
        Net = net;
        UI.PutDirected(false);

        Length = Net.GetEdgeArray("Euclidean Length");
        if (Length == null)
            return false;

        LIST = null;
        ForestLength = 0;

        return true;
    }

    public boolean Execute() {
        ExecuteMyKruskalSolver();

        return true;
    }

    public boolean Shutdown() {
        if (LIST != null)
            LIST.DeleteContents();
        ForestLength = 0;

        return true;
    }

    private void ExecuteMyKruskalSolver() { ... }
}
```

Figure 4.3: Changes to `MyKruskal.java` to accommodate algorithm logic.

information for the solver (e.g., the edges in a minimum spanning tree, and the total length of that tree).

- The global class variable `UI` is initialized in the solver constructor method. This variable is used throughout the solver to interact with the user interface.
- In the `Setup()` method we perform the following operations: (1) Initialize the global class variable `Net`. (2) Set a boolean value that tells GIDEN to draw the current network as an undirected graph. (3) Initialize the `Length` array to use the Euclidean length data field.[4] (4)

[4]In this example, we are "hard coding" the name of the data field to use for the solver's `Length` array input. In

```
private void ExecuteMyKruskalSolver() {
    Edge e;
    EdgeListIter eit = Net.GetEdgeListIter();
    Node n;
    NodeListIter nit = Net.GetNodeListIter();

    PQList unexamined = new PQList();
    NodeArray component = new NodeArray(Net.GetNodeIndex(),"component");

    /* set each node in its own component */
    int num_components = 0;
    for (n=nit.GetFirstNode() ; n != null ; n=nit.GetNextNode())
        component.Put(n,num_components++);

    /* mark all edges as unexamined */
    for (e=eit.GetFirstEdge() ; e != null ; e=eit.GetNextEdge())
        unexamined.Insert(e, Length.GetInt(e));

    LIST = new LinkList();

    while (LIST.GetSize() < Net.GetNodeCount()-1 && unexamined.NotEmpty()) {
        e = unexamined.GetMinEdge();
        unexamined.DeleteMinEdge();

        /* check to see if edge e creates a cycle */
        int source_component = component.GetInt(e.GetSource());
        int target_component = component.GetInt(e.GetTarget());
        if (source_component != target_component) {
            /* can add edge to LIST without creating a cycle */

            /* reset component for target to be that of source */
            for (n=nit.GetFirstNode() ; n != null ; n=nit.GetNextNode())
                if (component.GetInt(n) == target_component)
                    component.Put(n,source_component);

            /* decrement count of components */
            num_components--;

            /* "acquire" edge e by adding to LIST */
            LIST.AddItem(e);
            ForestLength += Length.GetInt(e);
        }
    }
}
```

Figure 4.4: Method `ExecuteMyKruskalSolver` containing algorithm logic.

Initialize the solver's result information to be undefined.

- The `Execute()` method is modified to execute the actual algorithm logic by calling a private class method named `ExecuteMyKruskalSolver`. (We separate the logic from the `Execute()` method for style reasons that we will not discuss here.) Figure 4.4 shows a complete listing of the `ExecuteMyKruskalSolver` method.
- The `Shutdown()` has been modified to reset the solver's result information. In particular, the contents of the tree/forest `LIST` are removed, and the value of `ForestLength` is set to zero.

Section 4.3.2 we will describe how to use an input dialog to allow the user to select the input array.

4.2.4 Setting the Network Labels

In order to add the algorithm logic to our solver, it was necessary to select an input array to use as the edge lengths. Although we selected the "Euclidean Length" array in the `Setup()` method, we neglected to update the edge labels to display the values associated with this array.

Figure 4.5 shows a modified version of our solver that incorporates two changes. First, a private solver method `InitializeNetworkLabels()` is now included. This method updates the edge labels to display the appropriate length values, and it sets the node labels to be empty (since we choose not to display any node information in this solver). The second change is to execute this new method from `Setup()`. Note that the new method is executed only after the `Length` array has been initialized.

```
package giden.userSOLVERS;
import giden.CORE.*;
import giden.GRDS.*;

public class MyKruskal extends ExecBase
{
    ...

    MyKruskal(UserBase userinfo) { ... }

    public boolean Setup(Network net) {
        Net = net;
        UI.PutDirected(false);

        Length = Net.GetEdgeArray("Euclidean Length");
        if (Length == null)
            return false;

        InitializeNetworkLabels();

        LIST = null;
        ForestLength = 0;

        return true;
    }

    public boolean Execute() { ... }

    public boolean Shutdown() { ... }

    private void InitializeNetworkLabels() {
        Edge e;
        EdgeListIter eit = Net.GetEdgeListIter();
        for (e=eit.GetFirstEdge() ; e != null ; e=eit.GetNextEdge())
            UI.PutLabel(e,Length.GetInt(e));

        Node n;
        NodeListIter nit = Net.GetNodeListIter();
        for (n=nit.GetFirstNode() ; n != null ; n=nit.GetNextNode())
            UI.PutLabel(n,"");
    }

    private void ExecuteMyKruskalSolver() { ... }
}
```

Figure 4.5: Changes to `MyKruskal.java` to set the network labels.

4.2.5 Adding Animation

Although we have now implemented the logic of Kruskal's algorithm, if we recompile and execute the modified solver we do not have any way of observing the steps of the algorithm or the final result. As a next step to a complete GIDEN solver, we will modify our code further to add some basic animation.

The notion we describe here of animation is motivated by an idea that the state of an algorithm operating on a network can be represented using a partition of the nodes and edges. Solver animation in GIDEN is then based on putting nodes and edges into implementor-defined subsets. Each subset in a GIDEN solver is represented by a `AnimationSet` object, and a distinct color is associated with each `AnimationSet`.

Figure 4.6 and Figure 4.7 show changes to `MyKruskal.java` needed to add basic solver animation. For our implementation of Kruskal's algorithm we define three `AnimationSet` variables:

1. `Current` — the unexamined edge that is selected as a candidate for being added to LIST. We associate the color "red" with this set.
2. `Acquired` — the set of edges that have been added to LIST, along with the set of nodes that are adjacent to any edge in LIST. The greenish color that we associate with this set is specified using RGB values.
3. `Discarded` — the set of edges that were selected as candidates for being added to LIST but were excluded because they would have created a cycle. We associate the color "yellow" with this set.

We now describe the changes in `MyKruskal.java`:

- The solver imports the Java class that is needed in order to work with colors, `java.awt.Color`.
- Global variables are defined for the `AnimationSet` objects.
- The constructor method creates the `AnimationSet` objects. Creating these objects through the `UI` variable registers the animation sets with the core environment.
- The `ExecuteMyKruskalSolver` method puts nodes and edges into the appropriate animation sets, using the `SingleStep()`, `Trace()`, and `Update()` methods of `UserBase`. The `SingleStep()` call is used to put a node or edge object into a particular animation set. Then the `Trace()` method is executed to explicitly mark the animation set as having changed. Finally, the `Update()` method is executed to signal the core environment that the solver is at a "good" stopping point if the user is executing in either trace animation mode or continuous animation mode (see Section 2.3.3).

```
package giden.userSOLVERS;
import giden.CORE.*;
import giden.GRDS.*;
import java.awt.Color;

public class MyKruskal extends ExecBase
{
    private UserBase UI;
    private Network Net;

    private EdgeArray Length;

    private LinkList LIST;
    private int ForestLength;

    private AnimationSet Current;
    private AnimationSet Acquired;
    private AnimationSet Discarded;

    MyKruskal(UserBase userinfo) {
        super(userinfo);
        UI = userinfo;

        Current = UI.CreateAnimationSet("Current",Color.red);
        Acquired = UI.CreateAnimationSet("Acquired",new Color(48,175,48));
        Discarded = UI.CreateAnimationSet("Discarded",Color.yellow);
    }

    public boolean Setup(Network net) { ... }

    public boolean Execute() { ... }

    public boolean Shutdown() { ... }

    private void InitializeNetworkLabels() { ... }

    private void ExecuteMyKruskalSolver() { ... }
}
```

Figure 4.6: Changes to `MyKruskal.java` to accommodate GIDEN animation.

```
private void ExecuteMyKruskalSolver() {
    Edge e;
    EdgeListIter eit = Net.GetEdgeListIter();
    Node n;
    NodeListIter nit = Net.GetNodeListIter();

    PQList unexamined = new PQList();
    NodeArray component = new NodeArray(Net.GetNodeIndex(),"component");

    /* set each node in its own component */
    int num_components = 0;
    for (n=nit.GetFirstNode() ; n != null ; n=nit.GetNextNode())
        component.Put(n,num_components++);

    /* mark all edges as unexamined */
    for (e=eit.GetFirstEdge() ; e != null ; e=eit.GetNextEdge())
        unexamined.Insert(e, Length.GetInt(e));

    LIST = new LinkList();

    while (LIST.GetSize() < Net.GetNodeCount()-1 && unexamined.NotEmpty()) {
        e = unexamined.GetMinEdge();
        UI.SingleStep(e, Current);
        UI.Trace(Current);
        UI.Update();

        unexamined.DeleteMinEdge();

        /* check to see if edge e creates a cycle */
        int source_component = component.GetInt(e.GetSource());
        int target_component = component.GetInt(e.GetTarget());
        if (source_component != target_component) {
            /* can add edge to LIST without creating a cycle */

            /* reset component for target to be that of source */
            for (n=nit.GetFirstNode() ; n != null ; n=nit.GetNextNode())
                if (component.GetInt(n) == target_component)
                    component.Put(n,source_component);

            /* decrement count of components */
            num_components--;

            /* "acquire" edge e by adding to LIST */
            LIST.AddItem(e);
            ForestLength += Length.GetInt(e);

            UI.SingleStep(e, Acquired);
            UI.SingleStep(e.GetSource(), Acquired);
            UI.SingleStep(e.GetTarget(), Acquired);
            UI.Trace(Acquired);
            UI.Update();
        }
        else {
            /* adding e would have created a cycle; "discard" edge e */
            UI.SingleStep(e, Discarded);
            UI.Trace(Discarded);
            UI.Update();
        }
    }
}
```

Figure 4.7: Changes to `ExecuteMyKruskalSolver` method for GIDEN animation.

4.2.6 Adding Pseudocode

After adding animation to our solver, we can visually observe the progress of Kruskal's algorithm applied to an appropriate network. As a supplement to this visualization, we now consider adding pseudocode for the algorithm that will show the progress of our `MyKruskal` solver in a textual form.

In the current release of GIDEN, solver pseudocode is stored in an external file. This pseudocode file is a plain ASCII text file, where each line has two items that are separated by a colon (":") character. The first item on a line is that line's token identifier. This token consists of the characters up to (but not including) the first colon character on the line, excluding leading and trailing whitespace. The second item on a line is the actual pseudocode text for that line. This item includes all characters that follow the colon separator, through the end of the line. Figure 4.8 shows a pseudocode text file for the `MyKruskal` solver.

```
start        : algorithm kruskal
             : {
mark edges   :     mark all edges as unexamined;
init LIST    :     LIST := emptyset;
main loop    :     while (|LIST| < |N|-1 and unexamined edges exist) do
             :     {
select edge  :         e := a minimal length unexamined edge;
mark edge    :         mark edge e as examined;
cycle check  :         if (adding e to LIST does not create a cycle) then
acquire edge :             add e to LIST;
cycle else   :         else
discard edge :             discard e;
             :     }
end          : }  (END of 'kruskal')
```

Figure 4.8: Pseudocode file for `MyKruskal` solver.

The pseudocode for a user-developed solver must be stored in the "`userPC`" subdirectory of the GIDEN installation directory. The name of the pseudocode file must be the the `String` name of the solver that you defined when linking the solver through the `SolverMenu` class with ".txt" appended (see Step 2 in Section 4.1.4). For example, if the changes to the `SolverMenu.java` file were made as presented in Section 4.2.2, then the name of the pseudocode file should be "The Kruskal Example.txt".

Changes to the `ExecuteMyKruskalSolver` method of the solver are shown in Figure 4.9. The method `ShowPseudoCodeText()` is used to specify the line of pseudocode that is displayed during solver execution. If a user is executing in pseudocode animation mode (see Section 2.3.3), then the solver will suspend execution each time a call to `ShowPseudoCodeText()` is made. Note that many of the pseudocode lines are displayed independently of any visual animation. When the pseudocode should instead be synchronized with changes in the animation sets, the call to `ShowPseudoCodeText()` should follow the appropriate calls to `SingleStep()` and should precede the subsequent call to `Update()`.

```
private void ExecuteMyKruskalSolver() {
    UI.ShowPseudoCodeText("start");

    ...

    /* mark all edges as unexamined */
    for (e=eit.GetFirstEdge() ; e != null ; e=eit.GetNextEdge())
        unexamined.Insert(e, Length.GetInt(e));
    UI.ShowPseudoCodeText("mark edges");

    LIST = new LinkList();
    UI.ShowPseudoCodeText("init LIST");

    while (LIST.GetSize() < Net.GetNodeCount()-1 && unexamined.NotEmpty()) {
        UI.ShowPseudoCodeText("main loop");

        e = unexamined.GetMinEdge();
        UI.SingleStep(e, Current);
        UI.ShowPseudoCodeText("select edge");
        UI.Trace(Current);
        UI.Update();

        unexamined.DeleteMinEdge();
        UI.ShowPseudoCodeText("mark edge");

        /* check to see if edge e creates a cycle */
        int source_component = component.GetInt(e.GetSource());
        int target_component = component.GetInt(e.GetTarget());
        UI.ShowPseudoCodeText("cycle check");
        if (source_component != target_component) {

            ...

            /* "acquire" edge e by adding to LIST */
            LIST.AddItem(e);
            ForestLength += Length.GetInt(e);

            UI.SingleStep(e, Acquired);
            UI.SingleStep(e.GetSource(), Acquired);
            UI.SingleStep(e.GetTarget(), Acquired);
            UI.ShowPseudoCodeText("acquire edge");
            UI.Trace(Acquired);
            UI.Update();
        }
        else {
            UI.ShowPseudoCodeText("cycle else");

            /* adding e would have created a cycle; "discard" edge e */
            UI.SingleStep(e, Discarded);
            UI.ShowPseudoCodeText("discard edge");
            UI.Trace(Discarded);
            UI.Update();
        }
    }
    UI.ShowPseudoCodeText("main loop");

    UI.ShowPseudoCodeText("end");
}
```

Figure 4.9: Changes to `ExecuteMyKruskalSolver` to accommodate GIDEN pseudocode.

4.2.7 Using the Status Line

With the algorithm logic, animation, and pseudocode in place, our solver is essentially complete. One oversight is that we might want to give additional messages to the user during solver execution — to report the length of our spanning tree, for example. The top line of the GIDEN drawing canvas, known as the *status line*, is reserved for writing such messages to the user[5].

In Figure 4.10 we make a few final changes to the `ExecuteMyKruskalSolver` method of our solver. These changes use the status line to report additional information about the progress of our solver. The methods used to change the status line information are `StatusLine()` and `ClearStatusLine()`. The `StatusLine()` method takes a Java `String` object as an argument, and displays the string text in the status line. As the name implies, the `ClearStatusLine()` method clears any text that is currently displayed in the status line.

Information that is presented in the status line is usually intended to supplement either the visual information presented through the solver's animation sets, or the textual information presented through the solver's pseudocode window. For this reason, calls to `StatusLine()` and `ClearStatusLine()` must be placed carefully in the solver code. In particular, if the status line information should be displayed when the solver is running in pseudocode animation mode, then the call to `StatusLine()` should appear before the related call to `ShowPseudoCodeText()`. Similarly, if the status line information should be synchronized with trace animation mode or continuous animation mode, then the call to `StatusLine()` should appear before the call to `Update()`. After setting the status line information, a subsequent call to `ClearStatusLine()` should be made to manually clear the message; this is necessary to avoid leaving any outdated information in the status line.

In our example solver, at the end of the `ExecuteMyKruskalSolver` method, we use the status line to report the aggregate length of edges in LIST. If the edges in LIST form a spanning tree of the underlying graph, then we report a message indicating the length of that tree in the status line. If the edges do not form a spanning tree, then they form a "minimal spanning forest" of the graph, and we report the total length of that forest. Note that this final status line message is not cleared with a call to `ClearStatusLine()`. In this case, we want the message to persist until the user either exits the solver, or re-starts solver execution using the Reset action button. In both of these cases, the core environment will automatically clear the status line.

[5]The status line can also be used to get user input on decisions, as demonstrated in `SimpleExample.java`.

```
private void ExecuteMyKruskalSolver() {
    UI.StatusLine("My Kruskal Solver begins...");
    UI.ShowPseudoCodeText("start");
    UI.ClearStatusLine();

    ...

    while (LIST.GetSize() < Net.GetNodeCount()-1 && unexamined.NotEmpty()) {

        ...

        if (source_component != target_component) {

            ...

            /* "acquire" edge e by adding to LIST */
            LIST.AddItem(e);
            ForestLength += Length.GetInt(e);

            UI.SingleStep(e, Acquired);
            UI.SingleStep(e.GetSource(), Acquired);
            UI.SingleStep(e.GetTarget(), Acquired);
            UI.StatusLine("Adding edge does not create a cycle; edge \"acquired\"");
            UI.ShowPseudoCodeText("acquire edge");
            UI.Trace(Acquired);
            UI.Update();
            UI.ClearStatusLine();
        }
        else {
            UI.ShowPseudoCodeText("cycle else");

            /* adding e would have created a cycle; "discard" edge e */
            UI.SingleStep(e, Discarded);
            UI.StatusLine("Adding edge would create a cycle; edge \"discarded\"");
            UI.ShowPseudoCodeText("discard edge");
            UI.Trace(Discarded);
            UI.Update();
            UI.ClearStatusLine();
        }
    }
    UI.ShowPseudoCodeText("main loop");


    if (num_components > 1)
        UI.StatusLine("Forest length = "+ForestLength+" units.");
    else
        UI.StatusLine("Minimum spanning tree length = "+ForestLength+" units.");

    UI.ShowPseudoCodeText("end");
}
```

Figure 4.10: Changes to `ExecuteMyKruskalSolver` to use the status line.

4.3 Advanced Topics

The previous section presents an example solver for the minimum spanning tree problem. The final example solver includes the main components of any functioning GIDEN solver, including support for pseudocode and algorithm animation. In this section, we describe additional features that can be incorporated into your solvers. In particular, we discuss the following topics:

- visually displaying result information for a solver
- using a solver input dialog
- executing solvers as subroutines
- modifying networks within solvers

Where appropriate, we show how the `MyKruskal` solver may be modified to incorporate a given feature.

4.3.1 Visually Displaying Results

One useful feature of GIDEN is the ability to visually display the progress of a solver during execution. To this point, however, we have not described how a solver can visually represent any result information.[6] To do this, we will introduce the notion of "final sets" and "color sets."

Final Sets

Final sets are similar to animation sets in several respects. First, final sets maintain a "color" property that is used when drawing items that are associated with the set. Secondly, final sets maintain a text "label" that describe the set. Finally, node and edge items are placed into final sets using the `SingleStep()` method of the `UserBase` class.

There are also several differences between final sets and animation sets. Unlike animation sets, final sets are not represented in the network window until a solver has completed execution. At this point, any animation-set buttons will be replaced with "swatches" that represent the final sets. Each of these swatches displays the color and label that are associated with the final set.[7] Also, since these final set swatches are only displayed after a solver has completed execution, it is possible to change the text in a final set's label at any point during solver execution. This is useful if some final set may be used to represent different result conditions. (An example of this is included in the modified `MyKruskal` solver code.)

Color Sets

Color sets in GIDEN are similar to final sets and animation sets in that they maintain a a "color" property that is used when drawing member items. The main difference that distinguishes color sets is that they do not maintain a "label" property, nor are they ever represented with a "button" or "swatch" in the network window. Instead, color sets are an internal mechanism for setting the color property of nodes and edges.

[6] So far, the status line is the only mechanism that we have described for reporting solver result information. For example, in the `MyKruskal` solver, the status line is used to report the final tree length.

[7] The difference between these final set "swatches" and the animation-set "buttons" is that the swatches are not selectable by the user; instead they only serve as a key to the displayed result information.

In practice, the use of color sets should be limited to setting the color of nodes and edges to some default value. This is useful, for example, to reset the color of an item after an operation has been performed, or to visually contrast items that are currently in an animation or final set with those items that are not in the set.

Updating the `MyKruskal` Solver

A modified version of `MyKruskal.java`, shown in Figure 4.11, illustrates the use of color sets and final sets to visually display solver results. In the case of the `MyKruskal` solver, the final result is a spanning tree or forest. The changes are described below:

- We have added a final set variable named `Forest`. This variable is initialized in the constructor to use the color "blue" and to display the label "Minimum Spanning Tree."
- We have added a color set variable named `Default`. This variable is initialized in the constructor to use the color "light gray."
- At the end of the `ExecuteMyKruskalSolver()` method, we have added a call to `MoveAll()`. The purpose of this call is to associate all node and edge items with the `Default` color set.
- Following the call to `MoveAll()`, the `LIST` is traversed. Each edge in the `LIST`, along with its endpoint nodes, are then placed into the `Forest` final set.
- If the input network has exactly one component, then the result of `MyKruskal` is a minimum spanning tree on the network. Otherwise, the solver returns a spanning forest that is composed of minimum spanning trees for each of the input network's components. In the first case, the initial label assigned to the `Final` set is correct, and remains unchanged. In the latter case, the label associated with the `Forest` final set is updated with a call to `PutLabel()` to reflect the solver result.

```
package giden.userSOLVERS;
import giden.CORE.*;
import giden.GRDS.*;
import java.awt.Color;

public class MyKruskal extends ExecBase
{
    ...

    private ColorSet Default;
    private FinalSet Forest;

    MyKruskal(UserBase userinfo) {
        ...

        Default = UI.CreateColorSet(Color.lightGray);
        Forest = UI.CreateFinalSet("Minimum Spanning Tree", Color.blue);
    }

    public boolean Setup(Network net) { ... }

    public boolean Execute() { ... }

    public boolean Shutdown() { ... }

    private void InitializeNetworkLabels() { ... }

    private void ExecuteMyKruskalSolver() {
        ...

        while (LIST.GetSize() < Net.GetNodeCount()-1 && unexamined.NotEmpty()) {
            ...
        }
        UI.ShowPseudoCodeText("main loop");

        UI.MoveAll(Default);
        ListEntry lit;
        for (lit=LIST.GetFirstEntry(); lit != null ;lit=LIST.GetNextEntry(lit)) {
            e = (Edge) lit.GetItem();
            UI.SingleStep(e, Forest);
            UI.SingleStep(e.GetSource(), Forest);
            UI.SingleStep(e.GetTarget(), Forest);
        }
        if (num_components > 1) {
            Forest.PutLabel("Forest of Component-Wise Spanning Trees");
            UI.StatusLine("Forest length = "+ForestLength+" units.");
        }
        else
            UI.StatusLine("Minimum spanning tree length = "+ForestLength+" units.");

        UI.ShowPseudoCodeText("end");
    }
}
```

Figure 4.11: Changes to `MyKruskal.java` to display results.

4.3.2 Using a Solver Input Dialog

Most GIDEN solvers will require node and edge data fields (arrays) as input. For example, minimum spanning tree solvers require that edge lengths be specified as part of the problem (see § 3.1). Although it is possible to "hard code" the names of data fields that are needed as input, this is an inflexible approach that can lead to unexpected results (e.g., if the named arrays don't exist in the input network). A preferable approach is to allow the user to specify which node and edge arrays should be used as input. The mechanism in GIDEN that supports this is the *solver input dialog*.

A solver input dialog works as follows: For each required input array the solver assigns a logical name for the input and specifies a data type. The solver then creates the input dialog through the user interface (via the `UserBase` object). The solver input dialog window appears as described in § 2.3.2, and the user selects the data fields that should be associated with each input. The solver can then retrieve the input arrays using the assigned logical names.

Input Dialog Data Types

In the current version of GIDEN, only the following node and edge data types are supported for solver input dialogs:[8]

`SolverInputItem.EDGE_ARRAY` — an edge array of unspecified data type. Member elements should be treated as Java `Object` items for retrieval from the selected `EdgeArray`. For example, a member element would be retrieved with the `GetObject()` method. (Note: This is the input type that should be used to access edge arrays of "text" type.)

`SolverInputItem.NODE_ARRAY` — a node array of unspecified data type. Member elements must be treated as Java `Object` items for retrieval from the selected `NodeArray`. For example, a member element would be retrieved with the `GetObject()` method. (Note: This is the input type that should be used to access node arrays of "text" type.)

`SolverInputItem.INTEGER_EDGE_ARRAY` — an edge array of integer data type. Member elements are of Java type `Integer`, and can be either be retrieved as `int` values using the `GetInt()` method, or they can be retrieved as Java `Integer` objects using the `GetObject()` method.

`SolverInputItem.INTEGER_NODE_ARRAY` — a node array of integer data type. Member elements are of Java type `Integer`, and can be either be retrieved as `int` values using the `GetInt()` method, or they can be retrieved as Java `Integer` objects using the `GetObject()` method.

Validating Input Data

Aside from being a convenient way for the user to specify the input arrays for a solver, the input dialog also provides the solver with a built-in mechanism for checking the validity of the specified input.

The input validation mechanism works as follows: When the user selects the Accept button in the input dialog, the dialog executes a solver method named `ValidateInput()`.[9] This method can retrieve the specified input arrays and perform any desired checks (e.g., the `Dijkstra` solver checks

[8] Currently, the solver input dialog can only be used to specify input arrays. If nodes or edges are required as input to a solver, these items must be requested through the `UserBase` object (see the html documentation).

[9] If this method is not included in your solver code, an empty method from the `ExecBase` base class will be executed.

that the specified "Length" array does not contain any negative values). If the inputs pass the validity checks, then the `ValidateInput()` should return "true"; otherwise the method should return "false," after optionally displaying an appropriate error message. The dialog window will not "accept" the specified input unless the `ValidateInput()` method returns "true." The input dialog will still return if the Cancel button is selected, in which case the solver should exit immediately.

Updating the `MyKruskal` Solver

A modified version of `MyKruskal.java`, shown in Figure 4.12, demonstrates how a solver might use an input dialog. The changes are described below:

- Before we begin the class declaration, we import the classes `giden.GUI.GidenSolverInput` and `giden.GUI.SolverInputItem`; these classes provide the input dialog window object as well as the input dialog data type descriptors.
- A global class variable `input` has been added. This variable is used to manage the solver's input data.
- The `input` variable is initialized in the solver's constructor method. After initialization, a single input item is registered. The requested input is an edge array of integer data type, with the assigned logical name "Length."
- A call to `CreateSolverInputDialog()` is added to the `Setup()` method. There are four required arguments: (1) the name of the solver, (2) the `this` reference to the solver, (3) the network object of interest, and (4) the variable that contains the input specifications.
- After the `CreateSolverInputDialog()` method returns, the solver checks to see if the `input` object has been validated (i.e., if the dialog was exited by selecting the Accept button, rather than the Cancel button). If the input has not been validated, then `Setup()` exits with a return value of "false," which will cause the network window to return to edit mode.
- The user-specified "Length" input array is retrieved from the `input` object.
- A private `ValidateInput()` method has been added. This method would normally be used by the solver to validate the user-specified inputs for any special requirements (for example, if the solver required that all "length" values be non-zero).

Finally, we note that the new segment of code in the `Setup()` method is conditionally executed if `UI.NullBase` is "false." This check is required when the solver may be executed as a subroutine of another solver. We will defer any explanation of this for now, though the topic will eventually be addressed in § 4.3.3.

```
package giden.userSOLVERS;
import giden.CORE.*;
import giden.GRDS.*;
import java.awt.Color;

import giden.GUI.GidenSolverInput;
import giden.GUI.SolverInputItem;

public class MyKruskal extends ExecBase
{
    private UserBase UI;
    private Network Net;

    private GidenSolverInput input;
    private EdgeArray Length;

    ...

    MyKruskal(UserBase userinfo) {

        ...

        input = new GidenSolverInput();
        input.addItem(SolverInputItem.INTEGER_EDGE_ARRAY,"Length");
    }

    public boolean Setup(Network net) {
        Net = net;
        UI.PutDirected(false);

        if (!UI.NullBase) {
            UI.CreateSolverInputDialog("My Kruskal Solver", this, Net, input);
            if (!input.isValidated())
                return false;

            /* extract input info */
            Length = (EdgeArray) input.getItemData("Length");
        }
        if (Length == null)
            Length = Net.GetEdgeArray("Euclidean Length");

        if (Length == null)
            return false;

        InitializeNetworkLabels();

        LIST = null;
        ForestLength = 0;

        return true;
    }

    public boolean ValidateInput() {return true;}

    public boolean Execute() { ... }

    public boolean Shutdown() { ... }

    private void InitializeNetworkLabels() { ... }

    private void ExecuteMyKruskalSolver() { ... }
}
```

Figure 4.12: Changes to `MyKruskal.java` to include a solver input dialog.

4.3.3 Executing Solvers as Subroutines

One of the advanced features of GIDEN is the ability for solvers to communicate with each other. In particular, a GIDEN solver can execute other solvers in order to obtain a solution to some subproblem. For example, a minimum-cost-flow solver might call a maximum-flow solver to obtain an initial feasible flow (see § 3.4 in this guide and Section 9.6 in [1]).

⋮

4.3.4 Modifying Networks within Solvers

As mentioned earlier, the solver is passed a network object when its `Setup()` method is executed. Throughout the course of the solver execution, it may be desirable to modify this network in some way. An example might be to add temporary nodes and edges to a minimum-cost-flow problem network in order to obtain an initial solution for "phase-1" of the network simplex algorithm. The current version of GIDEN provides several mechanisms for making temporary modifications to a given network object.

⋮

Bibliography

[1] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[2] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications*. Elsevier North-Holland, 1976.

[3] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. John Wiley & Sons, Inc., New York, 1998.

[4] David Dilworth, Collette Coullard, and Jonathan H. Owen. Graph & related data structures user's guide. Technical Report TR-96.09, Department of Industrial Engineering and Management Sciences, Northwestern University, 1996.

[5] Christopher V. Jones. *Visualization and Optimization*. Operations Research / Computer Science Interfaces Series. Kluwer Academic Publishers, Boston, 1996.

[6] Eugene Lawler. *Combinatorial Optimization: Networks and Matroids*. Saunders College Publishing, Fort Worth, 1976.

[7] Bertrand Meyer. *Object Oriented Software Construction*. Prentice Hall, second edition, 1996.

[8] Stefan Näher and Christian Uhrig. *The LEDA User Manual, Version R-3.3*, 1996. Available via anonymous ftp from `ftp.mpi-sb.mpg.de` in `/pub/LEDA`.

[9] Robret Endre Tarjan. *Data Structures and Network Algorithms*, volume 44 of *CBMS-NFS Regional Conference Series in Applied Mathematics*. Society for Industrial and Applied Mathematics, Philadelphia, Pennsylvania, 1983.