Shortest Paths

MA428, Dr Katerina Papadaki

I. Introduction

In this lecture we will cover the following:

1) Problem definition
2) The search algorithm
3) Topological Ordering
4) Characteristics of Solutions
5) Label Setting Algorithm for Acyclic Networks
6) Dijkstra's algorithm
7) LP formulation
8) Applications

This lecture covers material from the book ”Network Flows” by R.K. Ahuja, T.L. Magnanti, J.B. Orlin (AMO), which can be found in the following pages pp 73-79, 89, 97-102, 107-111, 123-127, 732-734.

Note that solutions to odd numbered exercises of AMO can be found at: http://jorlin.scripts.mit.edu/Solution_Manual.html.

II. Problem definition

A. Preliminaries

Consider a directed graph $G = (N, A)$ made up of $n = |N|$ nodes; $m = |A|$ directed edges or *arcs* denoted by a pair of nodes (i, j) signifying a connection from node i to node j (but not

K. Papadaki is with the Department of Management, Operational Research Group at London School of Economics, United Kingdom
E-mail: k.p.papadaki@lse.ac.uk

from node j to node i). Suppose we are also given a length (cost) c_{ij} for each arc (i, j). Let $s \in N$ be the *source node*. Such (directed) graphs with costs are called *networks*.

Definition Given a network $G = (N, A)$, a *directed walk* from node i_1 to node i_r is a sequence of nodes $i_1 - i_1 - \ldots - i_r$ such that $(i_k, i_{k+1}) \in A$ for $k = 1, \ldots, r - 1$.

Definition A *directed path* from node p to node q is a directed walk from p to q where none of the nodes repeat themselves.

Definition Given a network $G = (N, A)$ and a source node s, the *shortest paths problem* is the problem of finding a minimum length directed paths between the source node s and all the other nodes, $i \in N \setminus \{s\}$.

As the graph is directed, we need to take care of some *awkward cases*.

1) *Connectivity*: When we were dealing with undirected graphs, then connectivity could have been achieved with $n - 1$ edges in the acyclic case. In a network G there may not be directed paths between s and all other nodes irrespective of the number of arcs in G. For example, a network with arcs $(s, 1)$, $(2, 1)$ and $(2, s)$ has no directed path between node s and node 2, even though it has 3 nodes and 3 arcs.
2) *Negative cycles*: The length of the shortest walk from s to t may be $-\infty$, i.e., the problem may be unbounded. This happens if there is a directed cycle in the graph with a negative sum of lengths over the cycle (called a negative cycle). For example, consider G as given in Table I. We would like to avoid this problem of having a directed walk with length of $-\infty$ because it violates the sub-path optimality conditions that we will see later

ARC (i, j)	$(s, 1)$	$(1, 2)$	$(2, 3)$	$(3, 1)$
COST c_{ij}	5	-8	1	1

TABLE I

AN EXAMPLE OF A NETWORK G THAT HAS A NEGATIVE CYCLE:

To solve the above problems we provide the following definitions:

Definition Given a network $G = (N, A)$ and a source node s, we say that the nodes of the network are *reachable* from node s if there exist directed paths from node s to every other node.

Definition In a network $G = (N, A)$ with costs c_{ij}, a *negative cycle* is a directed cycle whose sum of lengths is negative.

To avoid these complications we make the following *Assumptions*:

1) The nodes of G are reachable from node s.
2) The network G does not contain a negative cycle.

Note that if, as in many applications, either there is no cycle in the graph or all lengths are non-negative ($c_{ij} \geq 0$ for all $i, j \in A$), then the second assumption is not needed.

For general networks there are simple algorithms that can be applied to test assumptions 1 and 2. For the 2^{nd} assumption we first test whether the network contains a directed cycle. We describe these search algorithms in sections III and IV.

III. Search Algorithm to find reachable nodes

In this section we define a *search algorithm* on a network G and demonstrate how to use it to find all nodes in the network that are reachable from source node s. I call the problem of finding all nodes in a network that are reachable from the source node, the *reachability problem*.

The search algorithm starts with the source node s and at each intermediate stage classifies all nodes into two groups: *marked* and *unmarked* nodes. In the reachability problem the marked nodes are nodes where a directed path from s has been established and for the unmarked nodes their reachability has not yet been established. Further, in the reachability problem we define an *admissible* arc to be an arc $(i, j) \in A$ such that i is marked and j is unmarked.

Arcs	$(1,2)$	$(1,3)$	$(2,3)$	$(3,5)$	$(4,2)$	$(4,3)$	$(4,5)$	$(4,6)$	$(6,1)$	$(6,5)$

TABLE II

A network with $n = 6$ and $m = 10$

Whenever, the search algorithm marks a new node j by examining an admissible arc (i, j), we say that i is the *predecessor* of node j, and use notation $pred(j) = i$.

The search algorithm is described in Algorithm 1. When it terminates there are marked and unmarked nodes. The nodes that are marked are reachable from source s and the unmarked nodes are not reachable. So if there is a directed path from s to all other nodes, then the algorithm

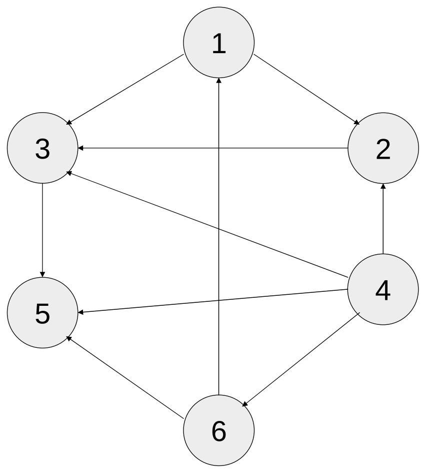


Fig. 1. Diagram of the network from Table II.

will terminate with all nodes marked. Further the algorithm will give the directed paths from node s to all the marked nodes. These can be found by the array $pred$ which traces back the path from node i to the source node.

Note further that combining the directed paths of the search algorithm will produce a tree. This is because the search algorithm finds only one path to each marked node: the arcs added are admissible only if they go from a marked node to an unmarked node. To produce a second path we would need to add an arc that goes from a marked node to a marked node. This cannot happen and thus the search algorithm produces a tree, which we call the *search tree* or, in the case of the reachability problem, the *reachability tree*.

The search algorithm described in Algorithm 1 does not specify how to select a node i from the $LIST$ of nodes. We consider two ways to do that:

1) Breadth-First Search: In this case the search algorithm selects the marked nodes from the $LIST$ in a First-In, First-Out (FIFO) order.
2) Depth-First Search: In this case the search algorithm selects the marked nodes from the $LIST$ in a Last-In, First-Out (LIFO) order.

We consider examples of both of the above methods.

Further the algorithm does not specify the order in which to select admissible arcs (if there are many admissible arcs) and thus the order in which to mark the nodes and put them to the

Algorithm 1 Search Algorithm

```
begin
    unmark all the nodes in $N$;
    mark node $s$;
    $pred(s) = 0$;
    $LIST = \{s\}$;
    while $LIST \neq \emptyset$ do
        select a node $i$ in $LIST$;
        If node $i$ is incident to an admissible arc $(i, j)$ then
            mark node $j$;
            $pred(j) = i$;
            add node $j$ to $LIST$;
        else delete node $i$ from $LIST$;
    end of while;
end;
```

$LIST$. For convenience we will use the rule that we always select the arc (i, j) for which the index j is the smallest, and thus mark first the nodes with smaller indices.

A. Breadth-First Search

Consider the network shown in Figure 1. Let us first let the source node to be node 1, $s = 1$. We will use the search algorithm to produce a search tree or reachability tree. Before we start the search algorithm it is always useful to create an *adjacency table*, which helps us identify the admissible arcs. The adjacency table is shown in Table III, and for each node in the network it shows nodes that are adjacent to that node. Remember that an arc (i, j) is admissible if a marked node i is adjacent to an unmarked node j. The search algorithm starts with the source node $s = 1$. We start by adding the source node to the $LIST$, $LIST = \{1\}$. From the adjacency table we can see that the admissible arcs are $(1, 2)$ and $(1, 3)$. We first add node 2 since we use the rule that we first add the one with a smaller index. Now we have $LIST = \{1, 2\}$ and the priority is given in this order since we used the breadth-first rule which gives priority to the first

NODE	1	2	3	4	5	6
ADJACENT NODES	$2, 3$	3	5	$2, 3, 5, 6$	$-$	$1, 5$

TABLE III

ADJACENCY TABLE FOR THE NETWORK DESCRIBED IN TABLE II

that came in the list, which was 1. We examine 1 and find admissible arc $(1, 3)$ and add 3 to the list $LIST = \{1, 2, 3\}$. Again we check 1 and see that it has no admissible arcs and thus we remove it to get $LIST = \{2, 3\}$.

Note that in the breadth-first case when we are examining node i for admissible arcs we will add all admissible arcs (and corresponding nodes) before we start examining any of the new entries. Thus, we can save time and add all admissible arcs and corresponding nodes in the order specified by their index.

We continue and pick from $LIST = \{2, 3\}$ node 2 since it was the first to enter. The only arc out of 2 is $(2, 3)$ and this is not admissible since it goes from a marked node to marked node. Thus we take node 2 out of $LIST$ and we have $LIST = \{3\}$. Now we examine node 3 and there is only one admissible arc $(3, 5)$. Thus we add node 5 to $LIST$ and remove node 3: $LIST = \{5\}$. We examine node 5 and note that there are no arcs out of 5. We remove node 5 from $LIST$ and now the list is empty and the algorithm terminates. This gives the reachability tree shown in Figure 2. The steps of the algorithm are shown in table IV: in each step we add all the admissible arcs and corresponding nodes (ordered) and remove the current node.

STEP	NEXT NODE FROM $LIST$	ADMISSIBLE ARCS (ORDERED)	INPUT NODES (ORDERED)	$LIST$ (ORDERED)
0	$-$	$-$	$-$	$\{1\}$
1	$s = 1$	$(1, 2)$,$(1, 3)$	2,3	$\{2, 3\}$
2	2	$-$	$-$	$\{3\}$
3	3	$(3, 5)$	5	$\{5\}$
4	5	$-$	$-$	$\{\}$

TABLE IV

BREADTH-FIRST SEARCH ALGORITHM FOR NETWORK IN FIGURE 1 WITH $s = 1$.

Thus, with source $s = 1$ we cannot reach all the nodes. Now let's look at the example with $s = 4$. We perform a similar procedure as described above and note the nodes that come out

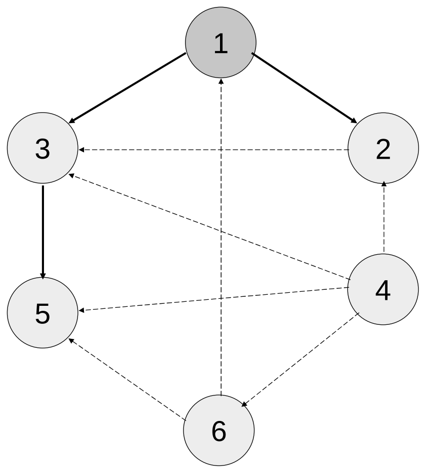


Fig. 2. The reachability tree from source node $s = 1$ using breadth first shown in bold lines: nodes $2, 3, 5$ are reachable from $s = 1$, but nodes $4, 6$ are not reachable from $s = 1$.

and go into $LIST$ in Table V. As we can see from Table V we have reached all the nodes and the resulting reachability tree is shown in Figure 3.

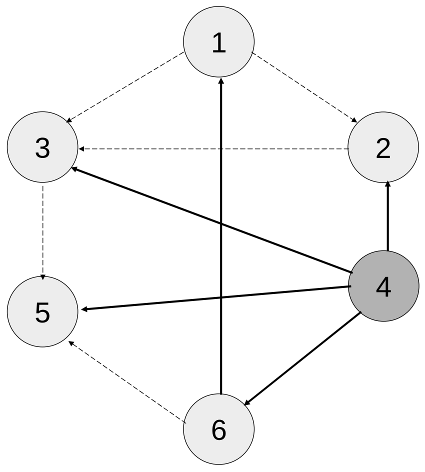


Fig. 3. The reachability tree from source node $s = 4$ using breadth first shown in bold lines: all nodes are reachable from $s = 4$.

STEP	NEXT NODE FROM $LIST$	ADMISSIBLE ARCS (ORDERED)	INPUT NODES (ORDERED)	$LIST$ (ORDERED)
0	–	–	–	$\{4\}$
1	$s = 4$	$(4,2)$,$(4,3)$,$(4,5)$,$(4,6)$	2,3,5,6	$\{2,3,5,6\}$
2	2	–	–	$\{3,5,6\}$
3	3	–	–	$\{5,6\}$
4	5	–	–	$\{6\}$
5	6	$(6,1)$	1	$\{1\}$
6	1	–	–	$\{\}$

TABLE V

BREADTH-FIRST SEARCH ALGORITHM FOR NETWORK IN FIGURE 1 WITH $s = 4$.

B. Depth-First Search

On the same network from Figure 1 with $s = 4$ we perform a depth-first search. Here we start with $LIST = \{4\}$ and pick the admissible arc $(4, 2)$ where the adjacent node has the lowest index. Then we add node 2 to the list. The ordered list is as follows: $LIST = \{2, 4\}$. Now node 2 has priority over node 4 since it was the last to go in the list. The admissible arc from node 2 is $(2, 3)$ and we add 3 to the list. Again, node 3 has priority. As this continues it forms a path consisting of arcs: $(4, 2)$, $(2, 3)$, $(3, 5)$ as shown in Table VI. In the depth-first search we go as *deep* as possible continuing the path that we started until we can go no further. Then we examine nodes from the depth of the path coming back to the source node.

The resulting reachability tree is shown in Figure 4. Note that this tree is quite different from the tree in Figure 3, where we used breadth-first search.

IV. ALGORITHM TO DETECT DIRECTED CYCLES: TOPOLOGICAL ORDERING

Given that negative cycles pose a problem in solving the shortest path problem, it is sometimes desirable to check whether a network contains any directed cycles. Even if a network has positive lengths/costs on each arc, establishing that a network does not have any directed cycles is very useful since more efficient algorithms can be used to find the shortest paths (see section VI).

We proceed with the following definitions.

Definition A network $G = (N, A)$ is said to be *acyclic* if it contains no directed cycle.

Note that the definition of acyclic is quite different for undirected graphs.

STEP	NEXT NODE FROM $LIST$	ADMISSIBLE ARCS (ORDERED)	INPUT NODES (ORDERED)	$LIST$ (ORDERED)
0	–	–	–	$\{4\}$
1	$s = 4$	$(4, 2)$	2	$\{2, 4\}$
2	2	$(2, 3)$	3	$\{3, 2, 4\}$
3	3	$(3, 5)$	5	$\{5, 3, 2, 4\}$
4	5	–	–	$\{3, 2, 4\}$
5	3	–	–	$\{2, 4\}$
6	2	–	–	$\{4\}$
7	4	$(4, 6)$	6	$\{6, 4\}$
8	6	$(6, 1)$	1	$\{1, 6, 4\}$
9	1	–	–	$\{6, 4\}$
10	6	–	–	$\{4\}$
11	4	–	–	$\{\}$

TABLE VI

DEPTH-FIRST SEARCH ALGORITHM FOR NETWORK IN FIGURE 1 WITH $s = 4$.

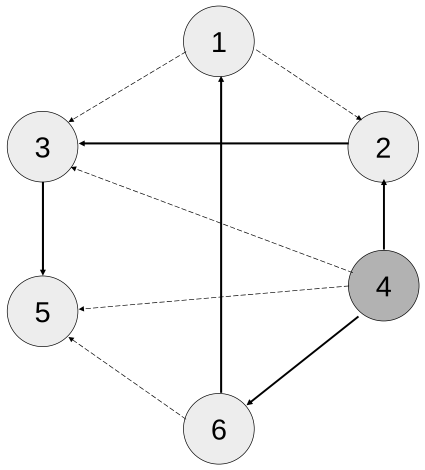


Fig. 4. The reachability tree from source node $s = 4$ using depth first shown in bold lines: all nodes are reachable from $s = 4$. This might result in a different reachability tree than breadth first (see Figure 3) but the same set of reachable nodes.

Definition In a network $G = (N, A)$, a *topological ordering* of the nodes in N is an ordering that satisfies:

$$\text{if } \ (i, j) \in A \ \text{ then } \ order(i) < order(j)$$

The following lemma links the above two definitions:

Lemma 4.1: A network $G = (N, A)$ is acyclic if and only if it has a topological ordering.

We will not formally prove the above lemma but consider the following: if a network has a topological ordering, then the condition $order(i) < order(j)$ cannot hold for all the arcs (i, j) that lie on a directed cycle. Thus, if we have a topological ordering in G, then the network is acyclic. The other direction of the lemma will be proved later.

We now introduce an algorithm that either finds a topological ordering if the network is acyclic or terminates with the directed cycle. The algorithm is described in Algorithm 2. At each step the algorithm looks for nodes with zero ***indegree*** ($indegree(i)$ is the number of arcs whose head node is i, $outdegree(i)$ is the number of arcs whose tail node is i). It labels these nodes with integers starting from 1 and removes them along with all the arcs that emanate from that node. It repeats that until there is no node with zero indegree or all the nodes have been labeled (ordered).

Why is the result a topological ordering? Suppose the algorithm labels node i with $order(i) = k$. Then any previously labeled node j will have order $order(j) = l < k$. If there was an arc $(i, j) \in A$ then we would not have removed node j in an earlier step because it would not have had zero indegree. Thus we cannot have an arc (i, j) with $order(i) > order(j)$.

To complete the proof of lemma 4.1 consider the following: If the network is acyclic, then the algorithm will terminate with zero nodes and arcs. If the algorithm terminates with zero nodes and arcs that means that the algorithm has labeled each node with its order and this is a topological ordering. Thus if the network is acyclic there exists a topological ordering.

Note that if in a connected network there is no node of zero indegree, then the network must contain a directed cycle. Thus if the algorithm does not terminate in a topological ordering, some of the remaining unlabeled nodes will produce a directed cycle.

Figure 5 shows a network where each node is labeled with its topological order. Node 2 has zero indegree and thus it is numbered $order(2) = 1$. Removing 2 and its emanating arcs leaves nodes 1 and 5 with zero indegree. Using a tie-breaking rule of choosing the node with the lowest index, we label node 1 with $order(1) = 2$. Then we label 5 with $order(5) = 3$ and continue. The resulting order of the nodes is: 2-1-5-4-3.

Figure 6 shows a network where it has a directed cycle. We begin with node 4 that has zero indegree. After we remove 4 and its emanating arcs, none of the remaining nodes has zero indegree. We stop here and say that the network contains a directed cycle.

Algorithm 2 Topological Ordering Algorithm

```
begin
    $k = 0$;
    $G' = G$;
    while there exists a node $i \in G'$ with $indegree(i) = 0$ do
        $k = k + 1$;
        $order(i) = k$;
        delete from $G'$ node $i$ and all arcs emanating from $i$;
    end;
    if $G'$ is empty (or equivalently if $k = n$) then
        there exists a topological ordering and the graph is acyclic;
    else
        the network contains a directed cycle;
end;
```

V. Characteristics of solutions

There are some properties of shortest paths that we explore in this section. For the following discussion we assume that the network has no negative cycles.

Lemma 5.1: Sub-path optimality: If the path $P = s - i_1 - i_2 \ldots - i_t$ is a shortest path from s to i_t, then for intermediate nodes $i_p \in \{i_1, i_2, \ldots, i_{t-1}\}$, the sub-path $s - i_1 - i_2 \ldots - i_p$ is a shortest path from s to i_p.

Proof: Suppose there was a better path $s - i'_1 - i'_2 \ldots - i_p$ from s to i_p. Then we know that $s - i'_1 - i'_2 \ldots - i_p - i_{p+1} \ldots - i_t$ is a directed walk from s to i_t with lesser cost. But a walk is not a path when it repeats nodes in which case it contains directed cycles. If we remove the directed cycles of this walk then we end up with a directed path from s to i_t with even lesser cost (since we assume no negative cycles). This contradicts the optimality assumption of P. Thus there cannot be a better path from s to i_p. ■

In Figure 7 the shortest path from node s to t is $P_1 - P_3$, which passes through node p. The Figure shows another path P_2 from s to p. By the sub-path optimality the length of P_2 must be

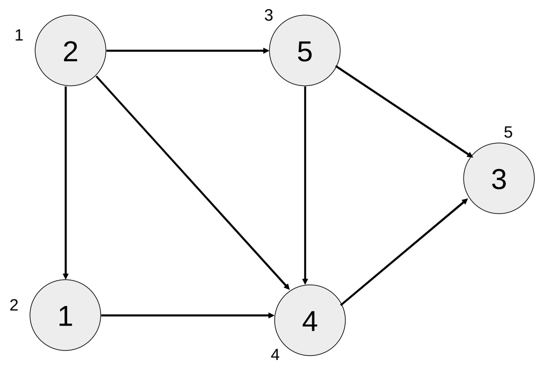


Fig. 5. Topological ordering of the network shown, where each node is labeled with its topological order. Note that we used a tie breaking rule of selecting the node with the lowest index.

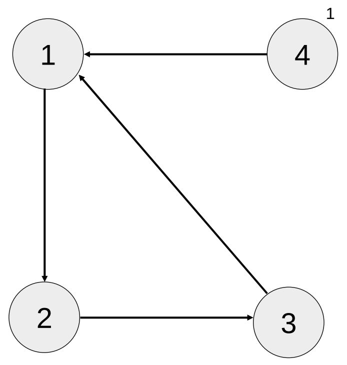


Fig. 6. This network has no topological ordering. After removing node 4 and all arcs emanating from it, we cannot find a node with zero in-degree. The remaining nodes $1, 2, 3$ form a directed cycle.

greater than the length of P_1, thus the shortest path from s to p is P_1.

For our next result we need some notation. Let $d(s,t)$ be the length/cost of the shortest path P from s to t. When source node s is given we can also denote it with $d(t)$.

Lemma 5.2: Sub-path cost optimality: A directed path P from s to t is a shortest path if and only if $d(s,j) = d(s,i) + c_{ij}$ for all $(i,j) \in P$.

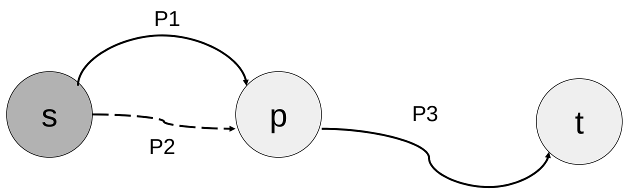


Fig. 7. The shortest path $P_1 - P_3$ from node s to node t passes through node p and it is shown by the bold line. By subpath optimality, the shortest path from node s to node p is P_1: $c(P_1) \leq c(P_2)$.

Proof: Let P be a shortest path from s to t. Then for any $(i, j) \in P$ we let P_i, P_j be the sub-paths of P up to i, j respectively. Then P_i, P_j are shortest paths to i, j respectively by sub-path optimality and $cost(P_i) = d(s, i)$ and $cost(P_j) = d(s, j)$. By decomposing P_j we get:

$$d(s, j) = \text{cost}(P_j) = \text{cost}(P_i) + c_{ij} = d(s, i) + c_{ij}$$

for each $(i, j) \in P$ which is the desired result.

Now suppose that $P = s - 1 - 2 - \ldots - t$ is a path that satisfies $d(s, j) = d(s, i) + c_{ij}$ for all $(i, j) \in P$. Then we have:

$$\begin{aligned} d(s,t) &= d(s, t-1) + c_{t-1,t} \\ &= d(s, t-2) + c_{t-1,t-2} + c_{t-1,t} \\ &\vdots \\ &= c_{s1} + c_{1,2} + \ldots + c_{t-1,t} = \text{cost of } P \end{aligned} \tag{1}$$

Since the cost of P is equal to the shortest path cost $d(s, t)$, then P must be a shortest path. ■

Lemma 5.3: Tree of Shortest paths: Let G be an s-connected (meaning there is a path from s to every other node) network without negative cycles. In such a network there is always a spanning tree that contains shortest paths from s to all other nodes.

Proof: Given that we have the shortest path costs $d(s,i)$ for each $i \in N$. Using the search algorithm, we can perform a breadth-first search of the network using as admissible arcs the arcs satisfying the equality $d(s,j) = d(s,i) + c_{ij}$, then we should be able to reach every node (G is s-connected). Also the result is a tree (since the search algorithm gives trees), it is a spanning tree (reaches all nodes) and the unique path from s to node i is a shortest path by construction. ■

Note that every arc in Figure 8 satisfies the sub-path optimality condition. Figure 9 shows a shortest path tree for the network in Figure 8.

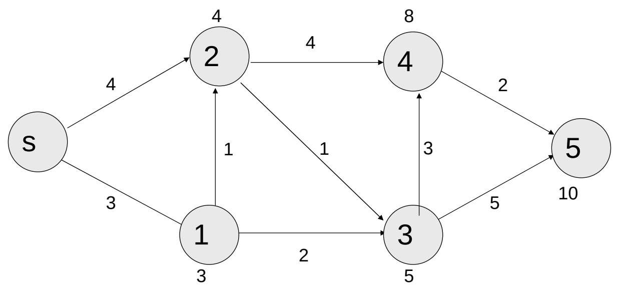


Fig. 8. A network with multiple shortest paths.

VI. Shortest Path Algorithm for Acyclic Networks

Recall that a network is acyclic if it contains no directed cycles. We consider acyclic networks separately for two reasons. The first is that in acyclic networks we can find shortest paths even when we have negative lengths/costs. The second reason is that acyclic networks can be topologically ordered and the topological ordering provides us a with fast algorithm to find shortest paths.

The algorithm described in this section is a label setting algorithm just like Dijkstra described in the next section.

The algorithm for finding shortest paths is described in Algorithm 3. The algorithm is quite straightforward and works as follows. It starts by assigning temporary distance labels $d(i)$ of

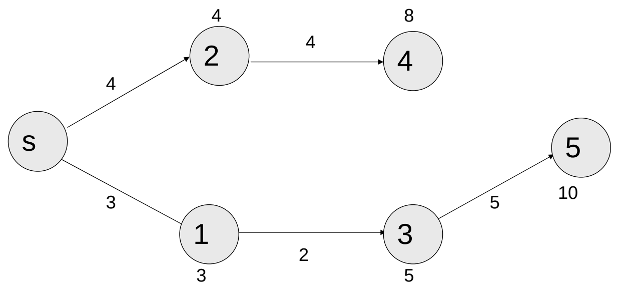


Fig. 9. A shortest path tree rooted at s for the network in Figure 8

the best known shortest path so far (upper bounds on the shortest paths). At the beginning it assigns $d(s) = 0$, which we know is optimal, and for all other nodes $d(i) = \infty$ since we do not know anything about their shortest paths. Then using the topological ordering of the nodes it examines each node i in that order. When examining node i, the algorithm checks all arcs (i, j) that emanate from i and if the distance label at j, $d(j)$, is greater than the distance label at i plus the arc cost c_{ij}, then it updates $d(j)$ to the new lower cost distance label: $d(j) = d(i) + c_{ij}$. It continues until all nodes have been examined.

The algorithm has $O(m)$ time since it examines every arc and there can be no faster algorithm since every arc needs to be examined.

A. *Example of the Label Setting Algorithm for Acyclic Networks*

An example of this algorithm is performed in the network shown in Figure 10 with source node $s = 1$. Note that the nodes are topologically ordered in the order $1-2-3-5-4-6$ (note 5 is before 4). We assign $d(1) = 0$, and $d(i) = \infty$ for all other nodes. We start with node 1 and check all arcs emanating from 1. These are arcs $(1, 2)$ and $(1, 3)$. So we update the distance labels $d(2) = d(1) + c_{12} = 6$ and $d(3) = d(1) + c_{13} = 4$. Then we go to the next node according to the topological ordering and do the same. The steps of the algorithm for this example are shown in Table VII. Note that in the table we underline the cost of a node when it is optimal, which is when the node is selected.

Algorithm 3 Shortest Path Algorithm on Acyclic Networks

```
begin
    $d(s) = 0$; $d(i) = \infty$ for all $i \in N \setminus \{s\}$;
    $k = 1$;
    while $k \leq n$ do
        pick $i$ such that $order(i) = k$;
        for all arcs $(i, j)$ that emanate from $i$ do
            if $d(j) > d(i) + c_{ij}$ then set $d(j) = d(i) + c_{ij}$ and $pred(j) = i$;
        end;
        set $k = k + 1$;
    end;
end;
```

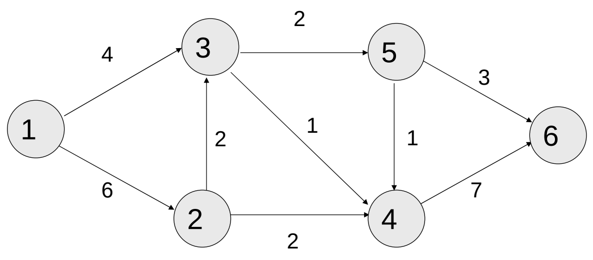


Fig. 10. An acyclic network where both the label setting algorithm for acyclic graphs and the Dijkstra algorithm have been tested.

B. Correctness of the Label Setting Algorithm for Acyclic Networks

Why does the Shortest Path Algorithm for Acyclic networks provide optimal paths? We will argue that when the algorithm examines a node its distance label is optimal. We will show this by induction. Suppose that the algorithm examined nodes $1, 2, \ldots, k$ and their distance labels

	Node i:	**1**	**2**	**3**	**4**	**5**	**6**	**Next Selected Node**
	Step							
$d(i)$	**1**	$\underline{0}$	∞	∞	∞	∞	∞	1
$pred(i)$		0						
	2		$\underline{6}$	4	∞	∞	∞	2
			1	1				
	3			$\underline{4}$	8	∞	∞	3
				1	2			
	4				5	$\underline{6}$	∞	5
					3	3		
	5				$\underline{5}$		9	4
					3		5	
	6						$\underline{9}$	6
							5	

TABLE VII

TABLE INDICATING THE STEPS OF THE LABEL SETTING ALGORITHM FOR ACYCLIC GRAPHS THAT USES TOPOLOGICAL ORDERING FOR THE NETWORK OF FIGURE 10.

are optimal. Suppose that the algorithm is currently examining node $k+1$. Let $s - i_1 - i_2 - \dots i_r - (k+1)$ be a shortest path of node $k+1$. Then $s - i_1 - i_2 - \dots i_r$ must be a shortest path to node i_r by sub-path optimality conditions. Since the nodes are topologically ordered and there exists an arc $(i_r, k+1)$, it means that $order(i_r) < order(k+1)$. Thus $i_r \in \{1, 2, \dots k\}$. By the inductive hypothesis the label $d(i_r)$ is optimal. When the algorithm examined node i_r it updated the label of node $k+1$ as follows $d(k+1) = d(i_r) + c_{i_r,k+1}$, thus the distance label $d(k+1)$ is optimal (i.e. equals to the distance of the shortest path to $k+1$).

VII. DIJKSTRA'S ALGORITHM

In this section we solve the shortest path problem in general networks (not necessarily acyclic) that have non-negative costs. We describe a label setting algorithm (Dijkstra) that solves this problem. For general networks with negative costs label correcting algorithms are needed which we will not cover in this lecture (you can read about these in chapter 5 of AMO).

A. *The Description of the Algorithm*

We define $A(i)$ to be the set of arcs $(i,j) \in A$ emanating from node i. Algorithm 4 gives a description of Dijkstra's algorithm on a network $G = (N, A)$ where $N = \{1, 2, \dots n\}$ with source node $s = 1$. The algorithm assigns distance labels $d(i)$ to each node i and these are updated throughout the algorithm. At an intermediate stage of the algorithm some of these labels are *permanent* (they are the optimal distances) and some are *temporary* (they are upper bounds on the optimal distances). To keep track of which ones are permanent and which are temporary we partition the node set N into two sets $PERM$ and $TEMP$, which contain the current permanent and temporary nodes respectively.

The algorithm starts by assigning distance label 0 to the source node 1 and ∞ to all other nodes. At the beginning all nodes are temporary. Then it picks a node from $TEMP$ that has the minimum distance label. In the first iteration this will be the source node 1 and it will remove it from $TEMP$ and put it in $PERM$. Then it scans all the arcs from 1 to all the other temporary nodes j and updates all the distance labels. Note that all the nodes adjacent to 1 will be updated since their distance labels were ∞. In the second iteration it will pick the node with the minimum distance label from set $TEMP$ and make it permanent. Note that this is the node that is closest to the source node which justifies why its distance label is optimal. This continues until all the nodes are permanently labeled.

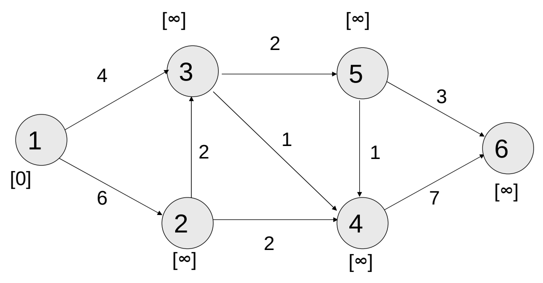
Algorithm 4 Dijkstra's Algorithm

```
begin
  1) Set $PERM = \{\}$; $TEMP = \{1, 2, \ldots\}$; $d(1) = 0$; $d(i) = \infty$ for $i \in N \setminus \{1\}$.
  2) Pick node $i$ from $TEMP$ with minimum $d(i)$ and remove it from $TEMP$ and put it into
     $PERM$.
  3) Scan arcs $(i, j) \in A(i)$ such that $j \in TEMP$,
     if $d(j) > d(i) + c_{ij}$ then set $d(j) = d(i) + c_{ij}$ and $pred(j) = i$.
  4) If $TEMP = \{\}$ terminate, otherwise go to step 2.
end
```

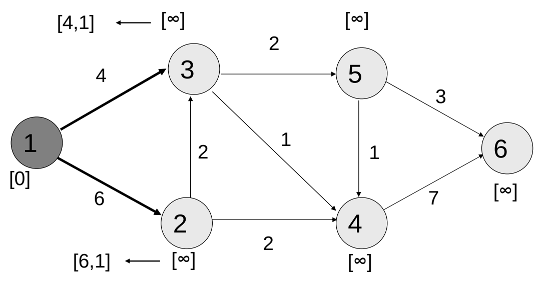
Note that the worst case complexity of Dijkstra is that for each selected node it checks/updates all the temporary nodes. This gives complexity $O(n^2)$.

B. An example using Dijkstra's Algorithm

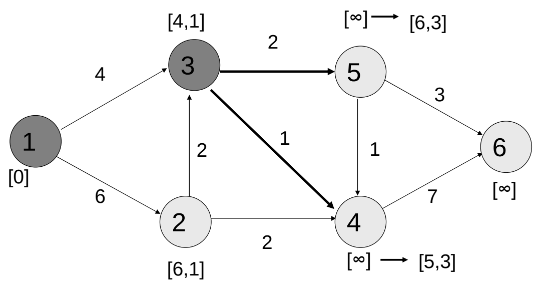
We use the network of Figure 10 to demonstrate how Dijkstra works. The six steps shown in Table VIII are also represented in the six diagrams of Figure 11.



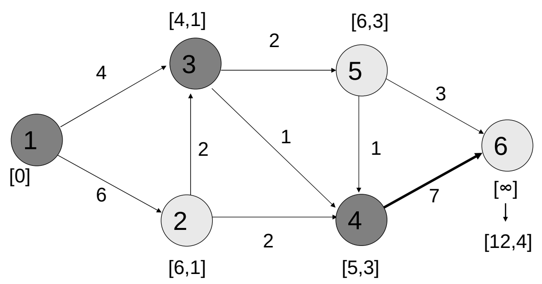
(a)



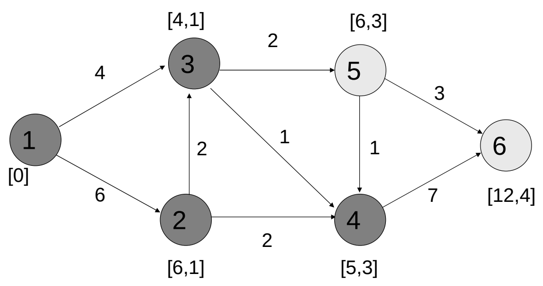
(b)



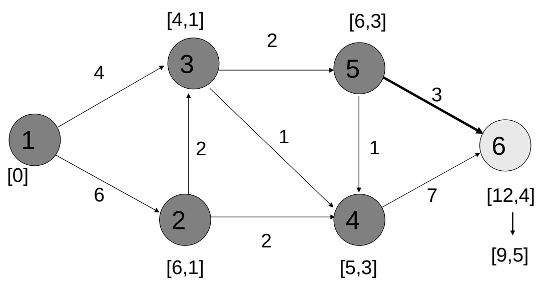
(c)



(d)



(e)



(f)

Fig. 11. Dijkstra Algorithm on network of Figure 10, with $s = 1$. Steps (a) - (f) are the steps 1-6 shown in Table VIII.

	Node i:	**1**	**2**	**3**	**4**	**5**	**6**	**Next Selected Node**
	Step							
$d(i)$ $pred(i)$	**1**	$\underline{0}$ 0	∞	∞	∞	∞	∞	1
	2		6 1	$\underline{4}$ 1	∞	∞	∞	3
	3		6 1		$\underline{5}$ 3	6 3	∞	4
	4		$\underline{6}$ 1			6 3	12 4	2
	5					$\underline{6}$ 3	12 4	5
	6						$\underline{9}$ 5	6

TABLE VIII

TABLE INDICATING THE STEPS OF DIJKSTRA FOR THE NETWORK OF FIGURE 10.

C. *Correctness of Dijkstra's Algorithm*

We will not provide a rigorous proof for why Dijkstra finds the shortest paths but instead argue the case. We would like to show that when a node becomes permanent then its distance label is indeed optimal. Suppose we have a partition of the nodes $PERM$, $TEMP$ and the nodes in $PERM$ have indeed optimal distance labels. In the next step we take a node i from $TEMP$ that has the minimum distance label $d(i)$. We would like to show that the shortest path from s to i has length $d(i)$. Out of all paths from s to i that only use nodes in $PERM$, the shortest one should have length $d(i)$ since $d(i)$ was derived by examining all the arcs of nodes in $PERM$.

However, could the shortest path from s to i pass through a node $k \in TEMP$? Suppose path P from s to i contains at least one node from $TEMP$. We decompose P into two segments P_1 and P_2: where P_1 starts at s and terminates at $k \in TEMP$ but all in between nodes are in $PERM$ (i.e. k is the first node from $TEMP$). We have the following:

$d(P_1) \geq d(k)$ Since $d(k)$ was derived by examining all the arcs of nodes in $PERM$, it should be the shortest path cost to k when using only nodes in $PERM$.

$d(k) \geq d(i)$ Since i was picked because it had the smallest distance label in $TEMP$

From the above we get $d(P_1) \geq d(i)$ and thus $d(P) = d(P_1) + d(P_2) \geq d(i)$. Thus, we cannot have a shorter path from s to i that goes through $TEMP$. Thus, $d(i)$ is the optimal distance.

D. Finding Shortest Paths for all Pairs of Nodes

It is easy to extend Dijkstra's algorithm to find all pairs of shortest paths. Consider a network with non-negative costs. A simple minded extension would be to apply it n times for $s = 1, 2, \ldots, n$. If there is no directed path from node p to node q the algorithm will show that when we run it for $s = p$. One way to implement it would be to select a node - say node 1 - from which a new arc is drawn to and from every other node where it does not exist and put a cost of infinity on all the new arcs. Now apply Dijkstra's algorithm n times. Where there is no path from p to q the algorithm will throw up a *shortest* directed path with a cost of infinity indicating that there is no path from p to q. A cleverer algorithm would use the sub-path optimality characteristic and not calculate distances to all paths. See AMO for details.

VIII. LP/IP FORMULATION

We let:

$$x_{ij} = \text{number of shortest paths in which arc } (i,j) \text{ is used}$$

. Then the shortest path problem can be formulated as follows:

$$\min_{x_{ij}} \sum_{(i,j)\in A} c_{ij}x_{ij}$$

$$\text{s.t.} \sum_{j:(i,j)\in A} x_{ij} - \sum_{k:(k,i)\in A} x_{ki} = \begin{cases} n-1 & \text{for } i = s \\ -1 & \text{for all } i \in N \setminus \{s\} \end{cases} \tag{2}$$

$$x_{ij} \geq 0 \text{ for all } (i,j) \in A$$

The left hand side of the constraints adds the number of shortest paths that leave node i and subtracts the number of shortest paths that come into i. Thus, the constraints state that for every node $i \in N\setminus\{s\}$ the number of shortest paths that come into node i is one more than the number of shortest paths that leave node i. This means that a shortest path will have to terminate at node i. For the source node s the condition basically says that in total $n-1$ paths should leave node s (it would be suboptimal to have a shortest path that comes into node s after leaving node s). Then the minimum cost solution is the same as the Shortest path tree.

We can also think about the problem as a flow problem. Suppose that each node apart from s demand one unit of flow. Then the constraints are flow balance constraints that ensure that a net of $n-1$ units of flow must go out of node s and a net flow of one unit should terminate at each other node.

For example, suppose the shortest path tree for a problem with nodes $s, 1, 2, 3, 4$ is given by the paths: $s-1-2-4$ and $s-3$. Then the LP solution will be: $x_{s1} = 3, x_{12} = 2, x_{24} = 1; x_{s3} = 1$ and all other $x_{ij} = 0$.

Note that even though the x_{ij} variable is an integer variable, we did not impose any integrality constraints. This is because the constraint matrix A given by 2 has two non-zeroes in every column, one $+1$ and one -1. Such A matrices are called totally unimodular and guarantee integer extreme points for the LP (we will not cover total unimodularity in this course). This is the reason why we can solve this integer program using linear programming.

IX. Applications

There are hundreds of applications. Typically these deal with large networks with hundreds of thousands of arcs. The variety of problem types is well covered by AMO. Scattered through AMO's book there are 22 different applications. We illustrate one of these and cover some others in the exercises.

A. Reallocation of Housing

A housing authority has a number of houses at its disposal that it lets to tenants. Each house has its own particular attributes. For example, a house might or might not have a garage, it has a certain number of bedrooms, and its rent falls within a particular range. These variable attributes permit us to group the houses into several categories, which we index by $i = 1, 2, \ldots, n$.

Over a period of time a number of tenants will surrender their tenancies as they move or choose to live in alternative accommodation. Furthermore, the requirements of the tenants will change with time (because new families arrive, children leave home, incomes and jobs change etc.). As these changes occur, the housing authority would like to relocate each tenant to a house of his or her choice category. While the authority can often accomplish this objective by simple exchanges, it will sometimes encounter situations requiring multiple moves: moving one tenant would replace another tenant from a house in a different category, who, in turn, would replace a tenant from a house in another category, and so on, thus creating a cycle of changes. We call such a change a *cyclic change*. The decision problem is to identify a cyclic change, if it exists, or to show that no such change exists.

To solve this problem as a network problem, we first create a *relocation graph* G whose nodes represent various categories of houses. We include arc (i, j) in the graph whenever a person living in a house of category i wishes to move to a house of category j. A directed cycle in G specifies a cycle of changes that will satisfy the requirements of one person in each of the categories contained in the cycle. Applying this method iteratively, we can satisfy the requirements of an increasing number of persons.

This application requires a method for identifying directed cycles in a network, if they exist. Topological sorting would identify such cycles. In general, many tenant reassignments might be possible, so the relocation graph G might contain several cycles. In that case the authority's management would typically want to find a cycle containing as few arcs as possible, since fewer moves are easier to handle administratively. This requires an algorithm for finding the shortest directed cycle in a network. The labeling procedure can be adapted to find this. Details can be found in the solution to exercise 5.43 on Ahuja's solutions website.

An interesting aspect of the application is the number of different objectives that could arise in reallocating houses. To see this consider a case where there are four categories of houses and the following adjacency table shows what transfers are required. Suppose each arrow represents

NODE	1	2	3	4
ADJACENT NODES	$2, 4$	$3, 4$	$2, 1$	3

TABLE IX

A HOUSING REALLOCATION PROBLEM.

demand from two households for the implied transfer. How would you devise a fair system?

B. *Critical Path Analysis*

See pages 732-734 of AMO. Note that the network is drawn differently from the way CPA notes in OR401 does. We have activities on nodes rather than on arcs. See Hillier and Liberman, *Operations Research* chapter (5 in my edition) on Network Analysis for a short coverage of CO1 to CO3 as well as CPA (called CPM there).

C. Allocating Inspection effort on a production line

See pages 99 - 100 of AMO.

D. Knapsack problem

Special case of Dynamic Programming see pages 100-102 of AMO.

X. Exercises

1) Solve 3.24 (a) and (b) of AMO.
2) Solve 4.4 of AMO. (Hint: you can use the approach in application 4.2, pp 99-100 or the dynamic programming approach.)
3) Add arc $(4, 1)$ with $c_{41} = 5$ to Figure 10 and solve the all pairs shortest path problem.
4) Solve 4.14 of AMO. Use the fact that the nodes in diagram (b) have a topological ordering.
5) For the housing reallocation problem in Table IX find a fair reallocation system under two different but reasonable definitions of fairness.
6) Formulate the problem of finding the shortest route from s to t as an LP and write its dual.