CHAPTER 3 SOME GRAPH ALGORITHMS

SECTION 3.1 WARSHALL'S ALGORITHM

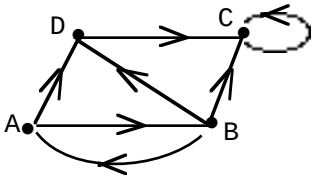
digraphs

A graph with one-way edges is called a *directed graph* or a *digraph*.

For the digraph in Fig 1, the adjacency matrix contains a 1 in row B, col C to indicate edge BC *from* B *to* C. There's a 0 in row C, col B since there is no edge CB *from* C *to* B.

In general, the adjacency matrix of a digraph is not necessarily symmetric.

An undirected graph can always be redrawn as a directed graph. Just replace each (two-way) edge by two one-way edges (Fig 2)



	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	0	1	0
D	0	0	1	0

FIG 1

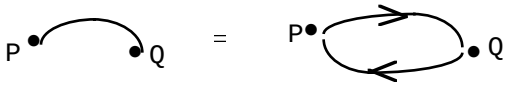


FIG 2

paths and cycles in a digraph

In the *underlying graph* of Fig 1 (where the edges are not directed) there is a path between A and C; each vertex is reachable from the other. But in the *digraph* (where the edges are directed) there is a path from A to C but not from C to A.

Similarly, in the underlying graph of Fig 1 there's a cycle ABDA. But in the digraph there is no cycle ABDA since there is no edge directed from D to A.

the reachability matrix M_∞

If M is the adjacency matrix of a digraph then an entry of 1 in row i, col j indicates an edge $v_i v_j$, i.e., a path from v_i to v_j with just one edge. In this section I'll extract from M a new matrix called the *reachability matrix*, denoted M_∞, in which an entry of 1 in row i, col j indicates a path (with one or more edges) from v_i to v_j, and an entry of 0 means no path at all. In other words, the reachability matrix indicates whether you can get from here to there. (Some books call M_∞ the *transitive closure* of M.)

footnote

It may seem odd but frequently you can't get from here to *here*. In Fig 1 there's a path from A to A, namely ABA, and there's a path from C to C, a loop, but there is no path from D to D.

Maybe it's not so odd after all. If the vertices in a graph represent people and an edge from A to B means that B is the parent of A then there can't be a path like A P Q A, from A to itself, since that would make A her own great grandmother.

Boolean arithmetic

To compute the matrix M_∞ from the adjacency matrix you'll be dealing entirely with 0's and 1's and it will turn out that Boolean arithmetic will be pertinent. It differs from ordinary arithmetic only in that

$$1 + 1 = 1.$$

In fact the new law 1 + 1 = 1 together with the old law 1 + 0 = 1 means that

$$1 + \text{anything} = 1$$

All sums in this section are intended to be Boolean .

Warshall's algorithm for finding the reachability matrix M_∞ for a digraph

Start with a digraph with n vertices.
Here's the idea of the algorithm.
Begin with the adjacency matrix M which indicates which pairs of vertices are directly connected.
The first round will get a matrix M_1 which indicates which pairs of vertices are connected by a path using v_1 as the only possible intermediate point (i.e., connected by an edge from here to there or by a path from here to v_1 to there).
The next round gets matrix M_2 which indicates which pairs of vertices are connected by a path allowing only v_1 and v_2 as possible intermediate points etc.

(1)

In general here is the *loop invariant* (something that's true after every round): At round k, you will get a matrix M_k. Look at the entry say in row 3, col 5. An entry of 1 means that there is a path from v_3 to v_5 with $v_1,\ldots, v_k$ as the only possible intermediates. An entry of 1 means that there is no path from v_3 to v_5 that uses only if only $v_1,\ldots, v_k$ as intermediates,

Here's the algorithm for finding M_∞ starting with a digraph with n vertices and adjacency matrix M.

Set M_0 = M.

To get M_1, for every row in M_0 that has a 1 in col 1, add (Boolean) row 1 to that row; i.e., look down col 1 and if there's a 1 in a row, add row 1 to that row.

To get M_2, for every row in M_1 with a 1 in col 2, add row 2 to that row.

In general, to go from M_{k-1} to M_k, for every row in M_{k-1} that has a 1 in col k, add row k to that row.
Continue until you have M_n.

At the end of the algorithm,

(2) $M_\infty = M_n$

I'll do an example first and then show why the algorithm works, i.e., show why (1) and (2) hold.

example 1

Start with a digraph with this adjacency matrix M.

$$M = \begin{matrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$$

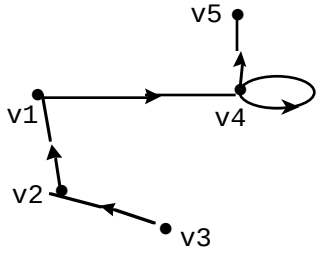


FIG 3

That's M_0.

round 1 To go from M_0 to M_1.
Look at col 1 in M_0. The only row with a 1 in col 1 is row 2. So add row 1 to row 2

and leave the other rows alone. Note that the Boolean addition amounts to replacing 0's in row 2 by 1's if there is a corresponding 1 in row 1.

$$M_1 = \begin{matrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$$

round 2 To go from M_1 to M_2.
Look at col 2 in M_1. The only row with a 1 in col 2 is row 3. So add row 2 to row 3 and leave the other rows alone.

$$M_2 = \begin{matrix} 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$$

round 3 To go from M_2 to M_3.
Look at col 3 of M_2. There are no 1's in col 3 so leave M_2 alone, i.e., $M_3 = M_2$.

round 4 Look at col 4 of M_3. The first four rows contain 1's in col 4 so add row 4 to each of the first four rows (note that adding a row to itself can't change it).

$$M_4 = \begin{matrix} 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{matrix}$$

round 5 Look at col 5 in M_4. The first four rows contain 1's in col 5 so add row 5 to them which doesn't change anything: $M_5 = M_4$. So

$$M_\infty = M_5 = \begin{bmatrix} 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

For instance, column 3 is all 0's so there are no paths to v_3 from anywhere; row 5 is all 0's so there are no paths from v_5 to anywhere.

why (1) holds

I'll use the graph from example 1 to illustrate the idea.
Here's why (1) holds for k = 1, i.e., for M_1.
As you go from M_0 to M_1 there are two ways in which entries of 1 appear in M_1.
(I) All the 1's in M_0 are still there in M_1
For example, the 1 in row 3, col 2 is a holdover. In M_0 (and therefore in M_1) it indicates an edge from v_3 to v_2, which you can call a path from v_3 to v_2 using no intermediates.

(II) Some 0's in M_0 might become 1's in M_1

For example, the 0 in row 2, col 4 of M_0 changed to a 1 because there was a 1 in row 2, col 1 (this told you to add row 1 to row 2) and there was a 1 in row 1, col 4 (which got added to the 0 entry and changed it to a 1) (Fig 4)

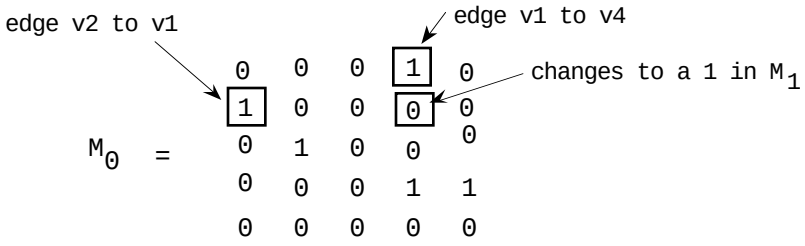


FIG 4 See a 0 in M_0 change to a 1 in M_1

The 1 in row 2, col 1 (in M_0) indicates that there is an edge from v_2 to v_1.
The 1 in row 1, col 4 indicates that there is an edge from v_1 to v_4.
Put those two edges together and you know there is a path from v_2 to v_4 using v_1 as an intermediate, justifying the 1 in row 2, col 4 of M_1.

All in all, a 1 in row i, col j in M_1 signals a path from v_i to v_j either using no intermediate (if it's type (I)) or using v_1 as the only intermediate (if it's type (II)). And every such path is signaled like that in M_1.

Here's why (1) holds for M_2.

As you go from M_1 to M_2 there are two ways in which entries of 1 appear in M_2.

(I) All the 1's in M_1 are still there in M_2.

For example the 1 in row 2, col 4 is a holdover.
In M_1, and now in M_2, it indicates a path from v_2 to v_4 using v_1 as a possible intermediate.

(II) Some 0's in M_1 might become 1's in M_2.

For example, the 0 in row 3, col 4 of M_0 changed to a 1 because there was a 1 in row 3, col 2 (this told you to add row 2 to row 3) and there was a 1 in row 2, col 4 (which got added to the 0 entry and changed it to a 1) (Fig 5)

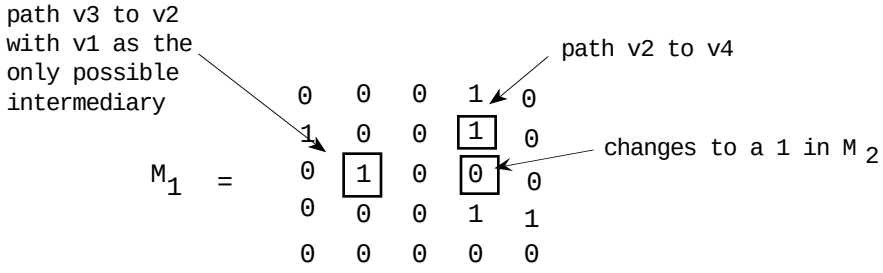


FIG 5 See a 0 in M_1 change to a 1 in M_2

As I just proved, the 1 in row 3, col 2 (in M_1) indicates that there is a path from v_3 to v_2 with v_1 as the only possible intermediate.

And the 1 in row 2, col 4 indicates that there is a path from v_2 to v_4 with v_1 as the only possible intermediate.

Put the two paths together and you know there is a path from v_3 to v_4 using v_2 as an intermediate and maybe v_1 but no other intermediates.

All in all, a 1 in row i, col j in M_2 (whether type(I) or type (II)) signals a path from v_i to v_j with v_1 and v_2 as the only possible intermediates. And every such path is signaled like that in M_2.

So far, I have (1) true for M_1 and M_2. Similarly it's true for M_k in general.

why (2) holds

By (1), an entry of 1 in row i, col j in M_n indicates a path from v_i to v_j using $v_1,\ldots, v_n$ as the only possible intermediates. But the graph only has the n vertices $v_1,\ldots, v_n$. So this is your last chance for paths. So $M_n = M_\infty$.

mathematical catechism (you should know the answer to these questions)

question 1 After round 4 of Warshall's algorithm, what does an entry of 1 in row 3, col 6 in M_4 mean.

answer There is a path from v3 to v6 using only vertices v1, v2, v3, v4 as possible intermediates.

question 2 After round 4 of Warshall's algorithm, what does an entry of 0 in row 3, col 6 in M_4 mean.

answer There is no path from v3 to v6 using only vertices v1, v2, v3, v4 as possible intermediates. (There may be a path to be discovered in later rounds using more than v1, v2, v3, v4 as possible intermediates.)

warning Saying "no path from v3 to v6 *so far*" is not good enough. Too vague.

PROBLEMS FOR SECTION 3.1

1. Let M be the adjacency matrix of a digraph. Find as many entries of M_∞ as possible if

(a) vertex v_2 is an isolated vertex

(b) the digraph contains the (directed) cycle $v_1v_7v_8v_1$

2. If M_∞ contains 1's in row 1, col 2 and in row 2, col 5 what other 1's must it have.

3. Suppose a digraph has 9 vertices. And there is a path $v_7v_1v_5v_2v_3$ (there may be other paths but this is all you know about).
What entries can you determine in M_0, ..., M_9 from this information.

4. (a) Suppose there is a 1 in row 1, col 6 in M_3. What does that signify (quote the loop invariant).

(b) Suppose there is a 0 in row 6, col 1 in M_3. What does that signify.

5. Find M_∞ if M is

(a)
$$\begin{matrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{matrix}$$

(b)
$$\begin{matrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{matrix}$$

6. Find the reachability matrix R_∞ if the adjacency matrix R is

(a)
$$\begin{matrix} 1 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 0 \end{matrix}$$

(b)
$$\begin{matrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \end{matrix}$$

(c)
$$\begin{matrix} 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \end{matrix}$$

7. Let

$$R = \begin{matrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

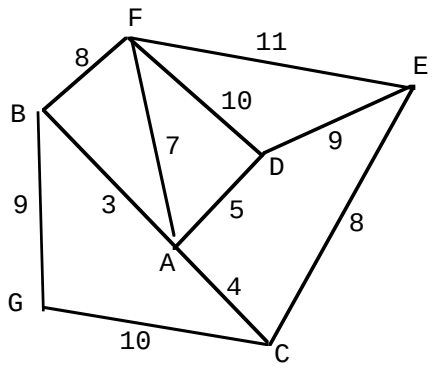
be the adjacency matrix of a digraph. Find R_∞ and then use it to decide if there's a path from (a) v_2 to v_4 (b) v_4 to v_2

SECTION 3.2 PRIM'S ALGORITHM

weighted graphs and weight matrices

Start with a graph with no loops and no multiple edges. Suppose each edge has a non-negative number associated with it called the edge *weight* or *cost*. (The weight of edge AB might be the cost of building a telephone line between A and B or the distance between A and B etc.)

Fig 1 shows a weighted graph with the corresponding *weight matrix*. The entry 4 in row A, col C (and row C, col A) is the weight of the edge AC. The entries on the diagonal are chosen to be 0, i.e., choose the "distance" from a vertex to itself to be 0. If two different vertices are not connected by an edge, put ∞ in the weight matrix, indicating no edge at any cost.



	A	B	C	D	E	F	G
A	0	3	4	5	∞	7	∞
B	3	0	∞	∞	∞	8	9
C	4	∞	0	∞	8	∞	10
D	5	∞	∞	0	9	10	∞
E	∞	∞	8	9	0	11	∞
F	7	8	∞	10	11	0	∞
G	∞	9	10	∞	∞	∞	0

FIG 1

minimal-weight spanning trees

A connected weighted graph has many spanning trees (each representing say a system of telephone lines or a system of roads allowing cities to be linked without wasting money on cyclic links). Each spanning tree has a weight, the sum of all the edge weights (the total cost of the telephone system, total miles of roadway). A *minimal spanning tree* for a connected weighted graph is a spanning tree with minimum weight (links the cities with the cheapest phone system, with the least asphalt).

Prim's (greedy) algorithm for a minimal spanning tree in a weighted connected graph

Given a weighted connected graph.
Here's the idea of the algorithm.

> Pick any initial vertex and start to grow a tree.
> Add vertices and edges one at a time as follows.
> At each step find all vertices not yet in the growing tree but connected by edges to some vertex already in the tree. Pick the one linked by the cheapest edge (i.e., be greedy) and grow the tree by adding that vertex and edge.

example 1

For a small graph you can run Prim's algorithm by inspection. Look at the graph in Fig 1 again. I'll find a minimal spanning tree starting with vertex G.

Vertices B and C are connected to G by edges. The cheaper connection is with B. So add vertex B and edge BG (Fig 2)

The next candidates are vertex C and edge CG, vertex F and edge BF, vertex A and edge AB. Cheapest is A and edge AB (Fig 3).

Now choose either vertex F and edge FB or vertex F and edge FA or vertex D and edge DA or vertex C and edge CA or vertex C and edge CG. Cheapest is vertex C and edge AC (Fig 4);

Then add vertex D and edge AD (Fig 5); vertex F and edge AF (Fig 6); vertex E and edge EC (Fig 7).

The total cost of the minimal spanning tree in Fig 7 is 36 (add the weights).

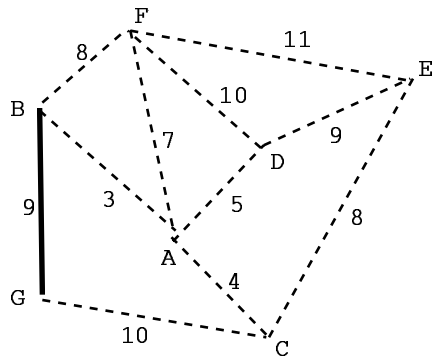


FIG 2

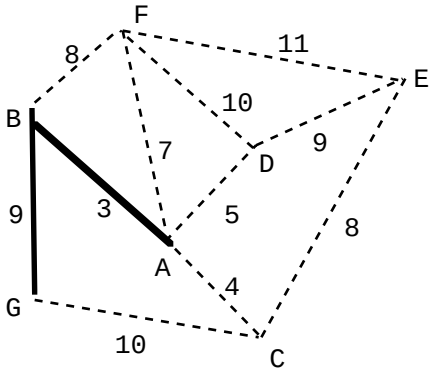


FIG 3

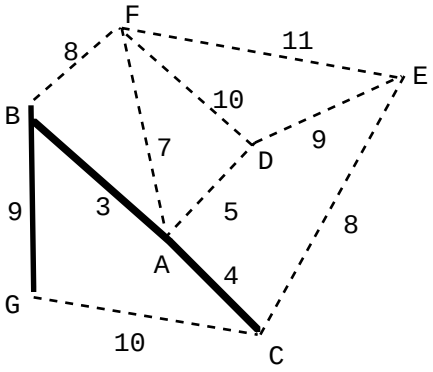


FIG 4

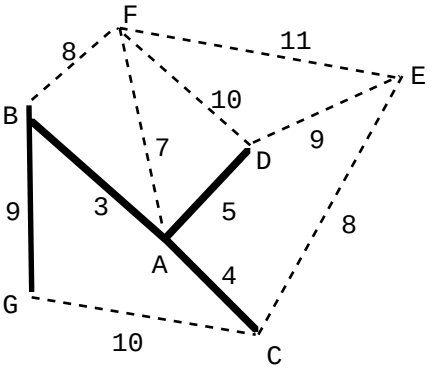


FIG 5

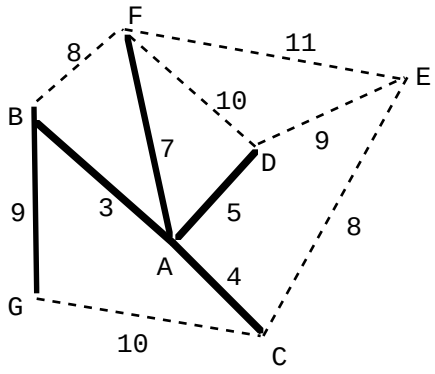


FIG 6

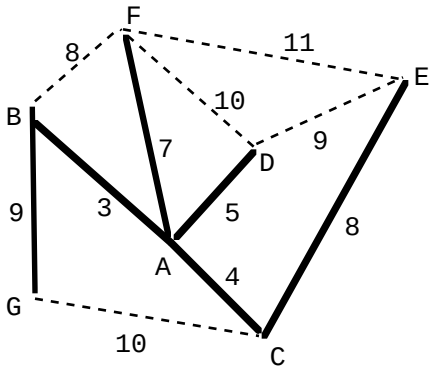


FIG 7

Prim programmed (how to do it in a large graph)

I'll repeat example 1, again starting with vertex G. Here's the weight matrix repeated.

	A	B	C	D	E	F	G
A	0	3	4	5	∞	7	∞
B	3	0	∞	∞	∞	8	9
C	4	∞	0	∞	8	∞	10
D	5	∞	∞	0	9	10	∞
E	∞	∞	8	9	0	11	∞
F	7	8	∞	10	11	0	∞
G	∞	9	10	∞	∞	∞	0

And at every round, for each vertex (except the one I started from) I'll keep track of these *loop invariants*.

status — Tells you if the vertex been picked for the tree yet
I've typed √ for picked and x for not picked.

link — For any *unpicked* vertex v, link v is the vertex in the growing tree joined to v by the cheapest edge. In other words, if unpicked vertex v is going to join the growing tree *now*, the cheapest way to go would be to add the edge from v to link v.

cost — For any *unpicked* vertex v, cost v is the weight of that cheapest connection. It might improve as more vertices are picked.

round 0

Here's the initial table.
I put all the vertices except G in the table.
Each vertex has G has its link because G is the only picked vertex so far.
The costs come from col G (or row G) in the weight matrix

	A	B	C	D	E	F
status	x	x	x	x	x	x
link	G	G	G	G	G	G
cost	∞	9	10	∞	∞	∞

round 1

Of all the unpicked vertices, B is the one with the smallest cost (i.e., would be the cheapest to add right now). Pick B. And update the table like this.

Update B
Change its status to picked.

Update A
At the last round, A had link G with cost ∞. This means that at that stage if we added A to the tree by adding edge AG, the cost would be ∞ and that's the best A could have done.
But now B is available to be a link. Let's see if A can do better by linking to B.
cost A = ∞
AB = 3, better than ∞.
So change link A from G to B and change cost A from ∞ to 3.

Update C:
cost C = 10
CB = ∞, not an improvement.
Leave link C and cost C unchanged.

Update D:
DB = ∞, can't be an improvement
Leave link D and cost D unchanged.

Update E
EB = ∞, can't be an improvement.
No change

Update F
cost F = ∞
FB = 8, an improvement
Set link F = B, cost F = 8

	A	B	C	D	E	F
status	x	√	x	x	x	x
link	B	G	G	G	G	B
cost	3	9	10	∞	∞	8

round 2

Of all the unpicked vertices, A is the one with the smallest cost. Pick A. And update.

Update A
Change its status to picked.

Update C
At the last round, C had link G with cost ∞. This means that at that stage if we added C to the tree by adding edge CG, the cost would be ∞ and that's the best C could have done.

But now A is available. Can C do better by linking to A.
CA = 4, an improvement.
Set link C = A, cost C = 4

Update D
Cost D = ∞
DA = 5, better
Set link D = A, cost D = 5

Update E
EA = ∞, can't be better
No change

Update F
cost F = 8
FA = 7, better
Set link F = A, cost B = 7

	A	B	C	D	E	F
status	√	√	x	x	x	x
link	B	G	A	A	G	A
cost	3	9	4	5	∞	7

round 3

Pick C (because it has the smallest cost of all the unpicked vertices))

Update D
DC = ∞, DC will not be < cost C. No change.

Update E
EC = 8, cost E = ∞, EC < cost E
Set link E = C, cost E = 8

Update F
FC = ∞, FC will not be < cost F. No change.

	A	B	C	D	E	F
status	√	√	√	x	x	x
link	B	G	A	A	C	A
cost	3	9	4	5	8	7

round 4

Pick D

Update E
ED = 9, cost E = 8, ED is not < cost E, no change

Update F
FD = 10, cost F = 7, FD is not < cost F, no change

	A	B	C	D	E	F
status	√	√	√	√	x	x
link	B	G	A	A	C	A
cost	3	9	4	5	8	7

round 5

Pick F.
Update E
EF = 11, cost E = 8, no change

	A	B	C	D	E	F
status	√	√	√	√	x	√
link	B	G	A	A	C	A
cost	3	9	4	5	8	7

round 6

Pick E and stop. All the vertices have now been picked.

	A	B	C	D	E	F
status	√	√	√	√	√	√
link	B	G	A	A	C	A
cost	3	9	4	5	8	7

Join each vertex to its final link to get edges AB, BG, CA, DA, EC, FA. They are the edges in a minimal spanning tree (Fig 7). The weight of the tree is the sum of the final costs, namely 3 + 9 + 4 + 5 + 8 + 7 = 36

Here's a summary of the algorithm.

To start, pick an initial vertex. Say it's v_0.

For the other vertices v set
status v = x (not picked yet)
link v = v_0
cost v = edge weight vv_0

At each round, among the unpicked vertices find the one with the smallest cost. Say it's v*. Change its status to √ (picked).
For each unpicked vertex v update like this.

If vv* < cost(v) then set link v = v* and cost v = vv*.

Repeat until all vertices are picked.

At the end of the algorithm there are the following conclusions.

The set of edges joining each vertex to its link is a minimal spanning tree and the sum of the cost v 's is its total weight

It can be shown that if the edge weights are all different then there is only one minimal spanning tree. If two edges have the same weight then there might be more than one minimal spanning tree but all minimal spanning trees have the same *weight*.

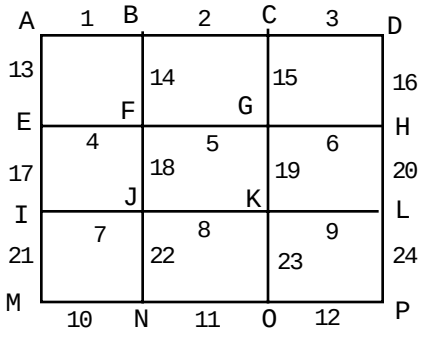
why the algorithm works

Prim's algorithm produces a spanning tree because it has the same form as the plain spanning tree algorithm from Section 2.3. But it takes a lot of proof to show that the tree is minimal just because you were greedy at each step. I'm leaving that out.

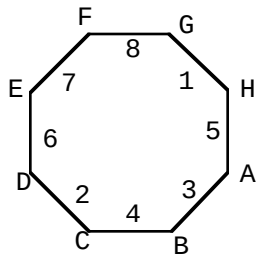
PROBLEMS FOR SECTION 3.2

1. Use Prim's algorithm to find a min spanning tree (but without formally keeping track of status, link, cost). For the initial vertex pick the alphabetically or numerically first vertex. List the edges in order as you find them and draw the tree.

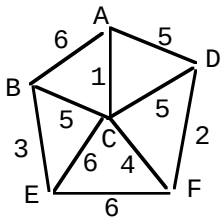
(a)



(b)



(c)



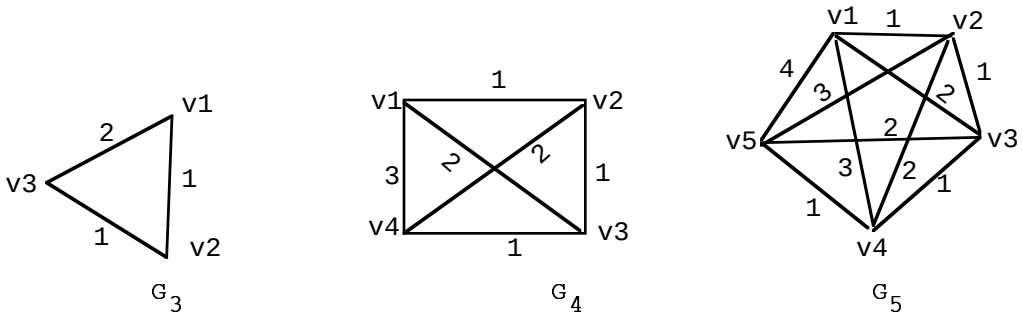
2. (a) Find the weight matrix for the graph in problem 1(c).
(b) Find a min spanning tree again, starting from vertex A, this time keeping track of status, link, cost at every step.

3. For the graph with the weight matrix below, find a min spanning tree, starting from vertex A, and its weight. Record status, link, cost at every step.

	A	B	C	D	E
A	0	2	∞	∞	3
B	2	0	1	∞	∞
C	∞	1	0	4	∞
D	∞	∞	4	0	5
E	3	∞	∞	5	0

4. Make up a graph that has more than one minimal spanning tree.

5. Start with the complete graph K_n and name the vertices successively $v_1, \ldots, v_n$. Suppose the cost of the edge $v_i v_j$ is $|i - j|$ (for example, the weight of $v_7 v_{11}$ is 4). Call the weighted graph G_n. The diagram shows the graphs G_3, G_4, G_5.



(a) Find a minimal spanning tree for G_5 and then in general for G_n. And find the weight of the min spanning tree.
(b) Repeat part (a) if the cost of the edge $v_i v_j$ is $i + j$ instead of $|i-j|$

6. (a) Look at the graph in problem 1(a). Of all the spanning trees which include the edge KO, find the one with minimum weight.
(b) Is the tree you just got in part (a) a minimal spanning tree for the graph.

SECTION 3.3 DIJKSTRA'S ALGORITHM

Dijkstra's algorithm for shortest distances and shortest paths from a given vertex

Start with a weighted conected graph.
Suppose you want to find a shortest path from vertex A to every other vertex.
Here's the idea of the algorithm.

> Dijkstra's algorithm will grow a tree of shortest paths.
> Start the tree with A. Add vertices one at a time as follows.
> At each step look at all vertices not yet in the growing tree but connected by an edge to some vertex already in the tree. Pick the one that is closest to A and add it and its connecting edge to the tree.

example 1

For a small graph you can run Dijkstra's algorithm by inspection. Look at the graph in Fig 1. I'll find a tree of shortest paths from A to every other vertex.

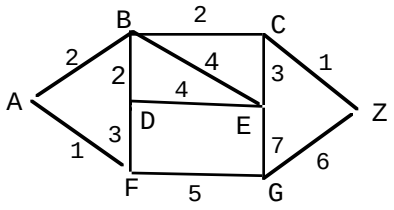


FIG 1

Start the tree with A.

Of all the other vertices, F is closest to A, via edge AF.
Add F and edge AF to the tree (Fig 2).

Consider vertices which can link to A or to F, namely B, D, G. Find their distances to A, namely AB = 2, AFD = 4, AFG = 6. Vertex B is closest to A.

Add B and edge AB to the tree (Fig 3).

Of all vertices which can link to A, B, or F (namely C, D, E, F) there's a tie between C (see path ABC) and D (see paths AFD, ABD) for which is closest to A. So I could add C and edge BC or I could add D and edge FD or BD. Toss a coin.

I'll add C and edge BC (Fig 4)

Of all vertices which can link to the growing tree, the one closest to A is D (see paths ABD or AFD, a tie).

I'll add D and edge FD (Fig 5).

Of all vertices which can link to the growing tree, the one closest to A is Z (see path ABCZ).

Pick Z. Add edge CZ (Fig 6).
(If the problem asked only for the shortest path from A to Z you would stop here.)

There's a tie between E and G.

Pick E (see path ABE). Add edge BE (Fig 7).
Pick G (see path AFG). Add edge FG (Fig 8.)

All vertices are picked. The end! You can read shortest paths from A to any other vertex from the tree in Fig 8.

If you want the shortest path from D to E, this tree doesn't help.

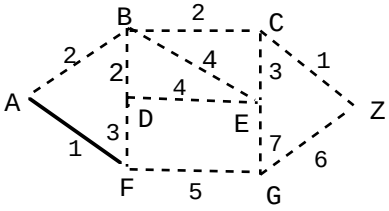


FIG 2

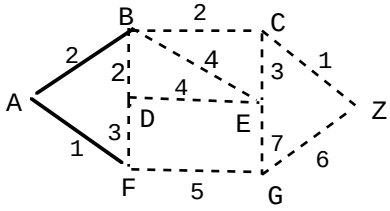


FIG 3

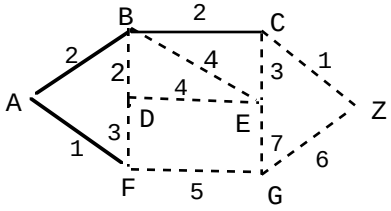


FIG 4

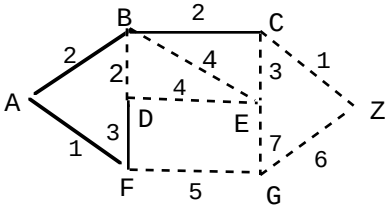


FIG 5

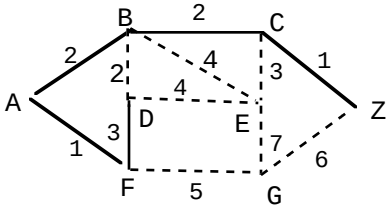


FIG 6

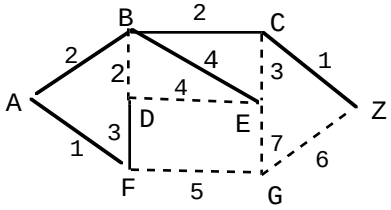
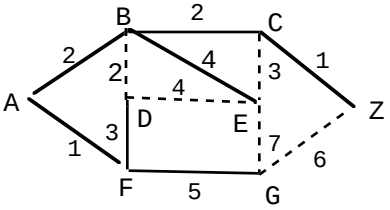


FIG 7



Dijkstra's tree of shortest paths from A
FIG 8

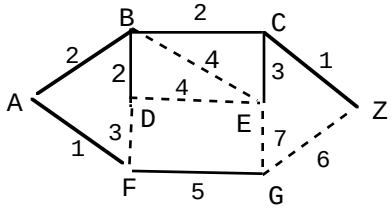
Dijkstra's algorithm vs. Prim's algorithm

The two algorithms both grow a spanning tree in a similar fashion, adding one vertex and edge at a time.

One difference is that Prim's algorithm can start with any initial vertex while Dijkstra's algorithm for shortest paths *from Q* must start the tree with initial vertex Q. The second difference is that Prim's algorithm grows the tree by picking the vertex linked by the cheapest edge while Dijkstra's algorithm for shortest paths *from Q* picks the vertex closest to Q.

To see the difference, run Prim's algorithm in Fig 1, beginning with initial vertex A. As the minimal spanning tree grows, you get Figs 1,2,3,4 as in Dijkstra's algorithm for shortest distances from A. But Prim's algorithm now chooses vertex Z and edge ZC because ZC is the cheapest edge which can be added on while Dijkstra's algorithm picks D and edge DF (or edge DB) because D is closest to A (i.e., because path AFD is the cheapest way to continue growing paths from A).

Fig 9 shows Prim's minimal spanning tree. It's of interest to the township which wants to link its towns with minimal asphalt. On the other hand, Dijkstra's tree of shortest paths from A in Fig 8 is of interest to people in town A who commute to surrounding towns.



Prim's minimal spanning tree
FIG 9

Dijkstra's algorithm programmed

Look at a new graph whose weight matrix is in Fig 10.

	A	B	C	D	E	F	G
A							
B	3						
C	6	2					
D	∞	4	1				
E	∞	∞	4	2			
F	∞	∞	2	∞	2		
G	∞	∞	∞	4	1	4	

FIG 10

I'll find shortest paths from A to every other vertex.

For each vertex other than A, keep track of status (picked or not picked yet), distance and link.

During the algorithm there are the following *loop invariants*.

(1) At the end of each round, for any vertex V, dist V is the shortest distance between A and V allowing only *picked* vertices as possible intermediates, and link V is the last vertex (just before V itself) on such a shortest path, i.e., it's where V attached itself to the tree.

(2) At the end of each round, if V is a *picked* vertex then dist V is *the* length of the shortest path between A and V (can't get a shorter path later in the algorithm as more vertices get picked).

round 0 Here's the initial table

I put all the vertices except A in the table.
Each vertex has A as its link because A is the only picked vertex so far.
The dists are just the weights of the edges to A.

	B	C	D	E	F	G
status	x	x	x	x	x	x
link	A	A	A	A	A	A
dist	3	6	∞	∞	∞	∞

round 1 The unpicked vertex with the smallest dist value is B so pick B. And update the table like this.

Update B

Change its status to picked.

Update C

At the last round, C had link A with dist 6. This means that at that stage if we added C to the tree by adding edge CA, the dist from A to C would be 6 and that's the best C could have done.

But now B is available to be a link.

Dist B = 3 so there is a path from A to B with length 3 (the best A to B path) and then B to C would be 2 more. That's a total distance of 5. So C can get a better path to A by linking to B.

Set link C = B

Set dist C = 5

warning

dist C is not 2, the single edge weight BC. Rather, dist C = dist B + BC = 3 + 2 = 5, the distance on a shortest-so-far path between A and C.

Update D
dist D = ∞ (i.e., there was no good path from A to D)
dist B + BD = 3 + 4 = 7
Set link D = B
Set dist D = 7.

Update E
dist E = ∞ (i.e. there was no good path from A to E).
But BE = ∞ also. So there can't be any improvement in a path from A to B to E.
No change in link E or dist E

Update F
BF = ∞. No change in link F or dist F

Update G
BG = ∞. No change in link G or dist G.

	B	C	D	E	F	G
status	√	x	x	x	x	x
link	A	B	B	A	A	A
dist	3	5	7	∞	∞	∞

Here's a summary.

Initially, pick A and for the other vertices V set

status V = x (not picked yet)
link V = A
dist V = weight of edge AV.

Among the unpicked vertices, pick the one with the smallest dist value. Call it V* and change its status to picked.

For each remaining unpicked vertex V, update as follows:

(3) If dist V* + V*V < dist V
then set link V = V* and set dist V = dist V* + VV*

(The test in (3) is designed to see if the distance from the last pick, A, to an unpicked vertex V can be improved using the latest pick, V*, as a last intermediate. If so, switch to an improved dist for V and to V* as its new link.)

Repeat until all the vertices are picked.

round 2 Of all the unpicked vertices, C has the smallest dist value so pick C and update.
dist C + CD = 5 + 1 = 6, dist D = 7. Set link D = C, dist D = 6
dist C + CE = 5 + 4 = 9, dist E = ∞. Set link E = C, dist E = 9
dist C + CF = 5 + 2 = 7, dist F = ∞. Set link F = C, dist F = 7
dist C + CG = 5 + ∞ = ∞ No change

	B	C	D	E	F	G
status	√	√	x	x	x	x
link	A	B	C	C	C	A
dist	3	5	6	9	7	∞

round 3 Pick D and update

If you're interested only in the shortest distance and shortest path between A and D then you can stop here; as soon as a vertex is picked, its dist and link give final results.

dist D + DE = 6 + 2 = 8, dist E = 9 Set link E = D, dist E = 8
dist D + DF = 6 + ∞ = ∞ No Change
dist D + DG = 6 + 4 = 10, dist G = ∞ Set link G = D, dist G = 10

	B	C	D	E	F	G
status	√	√	√	x	x	x
link	A	B	C	D	C	D
dist	3	5	6	8	7	10

round 4 Pick F and update

dist F + FE = 7 + 2 = 9, dist E = 8 No Change
dist F + FG = 7 + 4 = 11, dist G = 10 No Change

	B	C	D	E	F	G
status	√	√	√	x	√	x
link	A	B	C	D	C	D
dist	3	5	6	8	7	10

example of the loop invariants

Dist E = 8 and E has not been picked yet. This means that of all paths between A and E going through B or C or D or F (the picked vertices), the shortest is 8 (that shortest path is EDCBA). But there may be a shorter path going through the as yet unpicked vertex G.

Dist F = 7 and F *has* been picked. This means that of all paths between A and F going through B or C or D (the picked vertices), the shortest is 7; the path itself is FCBA. And furthermore this is *the* overall shortest path; there is no shorter path even if you go through the unpicked vertices E or G.

round 5 Pick E and update

dist E + EG = 8 + 1 = 9, dist G = 10 Change

	B	C	D	E	F	G
status	√	√	√	√	√	x
link	A	B	C	D	C	E
dist	3	5	6	8	7	9

round 6 Pick G and you're finished

At the end of the algorithm here are the conclusions.

(4) For each vertex V, dist V is the shortest distance from A to V.

The last table gives final results for all vertices.
For example, dist G = 9 so the shortest distance from A to G is 9.

(5) And you can use the links to find the shortest path itself:
link G = E, link E = D, link D = C, link C = B, link B = A.

A shortest path between A and G is A B C D E G.

warning

No adding is necessary at the end of Dijkstra's algorithm (it's *different* from Prim).
In example 2, the shortest path between A and G is ABCDEG and the shortest distance between A and G is 9, *not*
9 + 8 + 6 + 5 + 3.

mathematical catechism (you should know the answer to these questions)
question 1 After any round of Dijkstra's algorithm for shortest paths from A, what does dist Q represent if Q is an unpicked vertex.

answer Dist Q is the length of the shortest path between A and Q using picked vertices as the only possible intermediates. (Dist Q may improve as more vertices get picked.)

warning The answer is not shortest path "so far" or "up to now" or "at this time" because that's too vague.

question 2 After any round of Dijkstra's algorithm for shortest paths from A, what does dist Q represent if Q is a picked vertex.

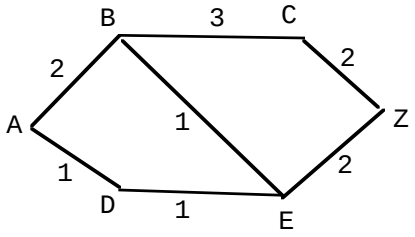
answer It is still the length of the shortest path between A and Q using picked vertices as the only possible intermediates. But more important, it is the length of *the* shortest path between A and Q (shortest of all possible paths); there is no shorter path no matter what intermediates you use.

PROBLEMS FOR SECTION 3.3

1. Make up an example to show that there can be *two* (equally) shortest paths from A to Z even if the edge weights are all different.

2. A graph has vertices A, B, C, D, E, P, Q, X, Z.
After round 4 of Dijkstra's algorithm for finding shortest paths from Q to the other vertices, vertices D, F, X, Z are picked. The other vertices are unpicked.
(a) If dist P = 8, what does that signify (quote the loop invariant).
(b) If dist X = 7, what does that signify

3. (a) Use Dijkstra's algorithm (but without formally keeping track of status, link, dist) to find the shortest paths from B to the other vertices in the diagram below. List all the shortest paths and their lengths and draw the tree of shortest paths
(b) For comparison, use Prim's algorithm starting with B to find a minimal spanning tree.



4. For the graph in problem 3, find shortest paths from A to the other vertices, keeping track of status, link, dist at each step.

5. For the given weight matrix, keep track of status, dist, link to find
(a) a shortest path from C to E
(b) shortest paths from B to all the other vertices

	A	B	C	D	E	F	G
A							
B	3						
C	6	2					
D	∞	4	1				
E	∞	∞	4	2			
F	∞	∞	2	∞	2		
G	∞	∞	∞	4	1	4	

6. For the given weight matrix, keep track of status, dist, link to find shortest paths from A to D.

	A	B	C	D	E	F	G	Z
A								
B	2							
C	∞	2						
D	∞	2	∞					
E	∞	4	3	4				
F	1	∞	∞	3	∞			
G	∞	∞	∞	∞	7	5		
Z	∞	∞	1	∞	∞	∞	6	

7. Look at Dijkstra's algorithm for shortest paths from A.
(a) True or False (and explain).
If vertex Z is picked right after vertex B then in the final table dist B ≤ dist Z.

(b) True or False (and explain). If Z is picked after B but not right after; say B then Q then Z are picked then in the final table dist B ≤ dist Z.

8. Here is the result of four rounds of Dijkstra's algorithm for finding shortest paths from vertex B to other vertices in a weighted graph.

vertex	A	C	D	E	F	G
status	√	√	√	x	√	x
link	B	B	C	D	C	D
cost	3	2	3	5	4	7

(a) What conclusions can you draw about the shortest distance between G and B and about the shortest path itself.
(b) What conclusions can you draw about the shortest distance between F and B and about the shortest path itself.

SECTION 3.4 THE RUNNING TIME (TIME COMPLEXITY) OF AN ALGORITHM

example 1 (running time of an algorithm which finds the largest [smallest] of n numbers)

Start with the n numbers $x_1, \ldots, x_n$. Here's an algorithm to find the maximum.

Compare x_1 with x_2 and choose the largest.
Compare the winner of the preceding round with x_3 and choose the largest.
Compare the winner of the preceding round with x_4 and choose the largest.
⋮
The final step is a comparison of the latest winner with x_n.

With this algorithm it takes n−1 comparisons to find the largest (smallest) of n numbers it can be shown that no algorithm can do better.

The algorithm must perform other operations besides comparisons (e.g., it has to store the winner at the end of each round) but it performs comparisons more than any other operation so it's an appropriate unit of measure. We say that the algorithm has *running time* or *time complexity* n−1. The functions n−1 and n have the same order of magnitude (coming next) so for simplicity we say that the running time is n.

order of magnitude

Suppose f(n) and g(n) both approach ∞ as $n \to \infty$ so that

(1) $$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \frac{\infty}{\infty}$$

If the limit in (1) turns out to be ∞ then f(n) is said to be of a *higher order of magnitude* than g(n); i.e., f grows faster than g.
If the limit is 0 then f has a *lower order of magnitude* than g.
If the limit is a positive number then f and g have the *same order of magnitude.*

The pecking order in (2) shows some well-known functions which approach ∞ as $n \to \infty$, and lists them in increasing order of magnitude, from lower to higher.

(2)

$$\ell n\, n, \quad \sqrt{n}, \quad n, \quad n^2, \quad n^3, \quad \ldots, \quad (1.5)^n, \quad 2^n, \quad 3^n, \ldots, \quad n!$$

The list is not intended to be complete. There are functions with lower order of mag then $\ell n\, n$, higher than n!, in between $\sqrt{n}$ and n, etc.

Here's the check that n^4 has a higher order of mag then n^3:

$$\lim_{n \to \infty} \frac{n^4}{n^3} = \frac{\infty}{\infty} = \lim_{n \to \infty} n \text{ (cancel)} = \infty$$

Here's the check that e^x has a higher order of mag then x^3:

$$\lim_{x \to \infty} \frac{e^x}{x^3} = \frac{\infty}{\infty} = \lim_{x \to \infty} \frac{e^x}{3x^2} \text{ (L'Hopital's rule)} = \frac{\infty}{\infty}$$

$$= \lim_{x \to \infty} \frac{e^x}{6x} \text{ (L'Hopital)} = \frac{\infty}{\infty}$$

$$= \lim_{x \to \infty} \frac{e^x}{6} \text{ (L'Hopital)} = \frac{\infty}{6} = \infty$$

On the other hand look at n^3 versus $5n^3$:

$$\lim_{n \to \infty} \frac{5n^3}{n^3} = \frac{\infty}{\infty} = \lim_{n \to \infty} 5 \text{ (cancel)} = 5$$

So $5n^3$ and n^3 have the same order of magnitude.

Intuitively, n^4 has a higher order of mag than n^3 because for *large* n, n^4 is *much* larger than n^3. In particular it is larger by a factor of n, a very big factor when n is large. On the other hand, $5n^3$ and n^3 have the same order of mag because for large n, $5n^3$ is not *much* larger than n^3. Of course it is larger by a factor of 5 but that factor stays constant as $n \to \infty$ so $5n^3$ does not decisively outdistance n^3.

Furthermore, not only do $5n^3$ and n^3 have the same order of mag but so do all of $5n^3$, n^3, $2n^3$, $n^3 + 2n - 3$, $8n^3 + n^2$ etc.

A polynomial has the same order of magnitude as its highest power. For example,

$n^4 + 3n^2$,
$6n^4$
$2n^4 - 7n^3$
etc

all have the same order of magnitude as n^4.

tractable versus intractable problems

Start with a problem of "size" n (e.g., find the smallest of n numbers, find an Euler cycle in a graph with n vertices, invert an $n \times n$ matrix). Suppose the problem can be solved in polynomial time, i.e., there is an algorithm to solve the problem which has a running time of n or n^2 or n^3 or n^4 etc. Then the problem is called *tractable*. On the other hand if all algorithms to solve the problem have running times higher than polynomial (e.g., exponential running time) then the problem is called *intractable*; for large n it would take centuries to solve the problem

big oh notation

If the running time of an algorithm is *less than or equal to* say n^4 then we say that the running time is $O(n^4)$ (big oh of n^4)

Often you can't pin down the precise running time of an algorithm because it might run faster for an easy problem of size n (e.g., a graph with n vertices and very few edges) as opposed to a messy problem of size n (a graph with n vertices and many edges). In that case you can find the running time in the worst situation and claim that the running time is O(worst)

example 2 (running time for Warshall's algorithm)

Let the adjacency matrix M be $n \times n$.
Use comparisons (is an entry a 1?) and additions as the units of measure.

Going from M_0 to M_1 takes one comparison for each of the n rows (to see if the entry in col 1 is a 1) for a total of n comparisons. And *at worst*, row 1 is added to each row which takes n additions for each of n rows, for a total of n^2 additions. So the total number of operations is $n^2 + n$.

To work your way up to M_n from M_0 you have to do these $n^2 + n$ operations n times for a final total of $n(n^2 + n) = n^3 + n^2$ ops. So, *at worst*, the running time is n^3, i.e., it's $O(n^3)$

warning

Don't say the running time of Warshall's algorithm is $O(n^3 + n^2)$. Use the simplest function with the same order of magnitude as $n^3 + n^2$ and call the running time $O(n^3)$. And use the big oh notation because n^3 was a worst case scenario.

example 3 (running time for Prim's algorithm)

Let M be an n × n weight matrix. Use the comparison as the unit of measure.

First count the comparisons you need in each round to find the unpicked vertex with the smallest cost. In round 1 there are n−1 unpicked vertices, and with the algorithm in example 1, it takes n−2 comparisons. In round 2 there are n−2 unpicked vertices and it takes n−3 comparisons to pick the smallest etc. In the (n−2)th round there are 2 unpicked vertices and it takes 1 comparison.

The total number of comparisons is

$$(n-2) + (n-3) + \ldots + 1$$

for reference (to be given out on exams if needed)

(A) $1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$

(B) $1^2 + 2^2 + 3^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$

To find the sum, use (A) but with n−2 playing the role of n:

$$(n-2) + (n-3) + \ldots + 1 = \frac{(n-2)(n-1)}{2}$$

Now count the comparisons necessary to update in each round (is vv* < cost v?) In round 1, after picking a vertex there are n−2 vertices to update so there are n−2 comparisons. In the next round there are n−3 comparisons etc. so the number of comparisons for updating is (n−2) + (n−3) + ... + 1 again.

The final total is $2 \cdot \frac{(n-2)(n-1)}{2} = (n-2)(n-1)$ so the order of mag of the running time is n^2. I don't call it $O(n^2)$ because the conclusion was that the order of mag *is* n^2, not just $\leq n^2$.

$O(n^2)$ and $O(n^3)$ versus plain n^2 and plain n^3

When I counted operations for Prim's algorithm I found that for *any* n × n matrix the running time is n^2. For Warshall's algorithm I found that in a *worst* case the running time is n^3; for *some* matrices it could be $< n^3$.

So we say that Warshall's algorithm is $O(n^3)$ and Prim's algorithm is n^2 (without the big oh).

example 3 (running time for Dijkstra's algorithm)

Let M be an n × n weight matrix. Use the comparison as the unit of measure.

Dijkstra's algorithm must have the same running time as Prim's algorithm, namely n^2. The only difference between the algorithms is in *what* is being compared in the updating step (vv* vs. cost v in Prim, dist v* + v*v vs. dist v in Dijkstra). There is no difference in the *number* of comparisons in updating or picking.

PROBLEMS FOR SECTION 3.4

1. Here's the Bubblesort algorithm which arranges n numbers $x_1,\ldots, x_n$ in increasing order.

Find the larger of x_1 and x_2 and store the larger in x_2, the smaller in x_1.

Find the larger of x_2 and x_3 and store the larger in x_3, the smaller in x_2.

Continue until you find the larger of x_{n-1} and x_n and store the larger in x_n, the smaller in x_{n-1}.

By the end of this round, the largest of $x_1,\ldots, x_n$ has bubbled up to register x_n.

Now repeat the process with $x_1,\ldots, x_{n-1}$.

And repeat with $x_1,\ldots, x_{n-2}$ etc.

Find the running time of the algorithm using the comparison as the unit of measure.

2. (a) Find the running time of the following algorithm using the comparison as the unit of measure

```
Let k = 1,..., n
    Let i = 1,..., n
        Let j = 1,..., n
        If W(i,k) + W(k,j) < W(i,j), set W(i,j) = W(i,k) + W(k,j)
```

footnote
For the purposes of this problem it doesn't matter what the algorithm is for but in case you're curious, this is Warshall's algorithm for shortest distances (not the Warshall from Section 3.1 which finds a reachability matrix). If W is the weight matrix of a graph then at the end of the algorithm, W(i,j) is the shortest distance from v_i to v_j.

(b) Does it change the answer to part (a) if you use comparisons *and* additions as the unit of measure?

(c) Here's a totally meaningless algorithm. Find its running time using the comparison as the unit of measure.

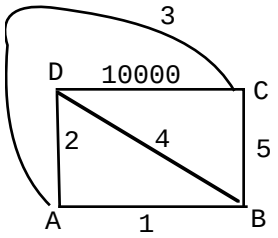
```
Let k = 1,..., n
    If W(1,k) + W(2,k) ≤ W(3,k) set W(1,1) = 4

Let i = 1,..., n
    If W(i,1) + W(i,2) ≤ W(i,3) set W(1,1,) = 5

Let j = 1,..., n
    If W(j,j) ≤ W(1,1) set W(1,1) = 6
```

3. A traveling saleswoman who has to visit cities $v_1,\ldots, v_n$ in her district wants a cycle, called a *Hamilton* cycle, which passes through every vertex exactly once (as opposed to an Euler cycle which passes through every edge exactly once). Furthermore she wants a Hamilton cycle with minimum distance so that she does a minimum of driving.

For example, for the weighted graph in the diagram, one of several Hamilton cycles is ABCDA with a total weight of 10008. The minimal Hamilton cycle for the graph in the diagram happens to be ADBCA, with weight 14.



Here's a (greedy) algorithm called the nearest neighbor algorithm which is reputed to usually produce a reasonable although not necessarily minimal Hamilton cycle for a weighted graph.

> Start with any vertex, say x.
> Pick the vertex, call it y, linked to x by the smallest edge and start a path with edge xy.
> Look at all the vertices not picked yet and find the one, call it z, linked to y by the smallest edge. Grow the path by adding edge yz.
> The path keeps growing in this fashion until it passes through all the vertices.
> Then make a cycle by adding the edge from the last vertex picked back to x.

In other words, the path grows (greedily) one edge at a time from x to its nearest neighbor y, then to y's nearest neighbor z, then to z's nearest neighbor etc. Once the last vertex, say q is picked, close up the cycle with edge qx.

For the graph in the diagram, starting at vertex A, the nearest neighbor algorithm produces the Hamilton cycle ABDCA; its weight is 10008, not very close to the minimum so being greedy didn't get a good result this time.

Find the running time of the nearest neighbor algorithm for a graph with n vertices, assuming there are no multiple edges. (Two vertices which aren't connected can be thought of as joined by an edge with weight ∞, so you can consider that the graph is the complete graph K_n, with weights.)

4. Dijkstra's algorithm finds shortest paths from a fixed vertex to every other vertex. Suppose you want to use Dijkstra's algorithm over and over to find paths from *every* vertex to every other vertex. What's the running time of the over-and-over-again Dijkstra.

5. John claims that a certain algorithm is $O(n^2)$ and Mary says it's $O(n^3)$

(a) Could both be right.
(b) Could John be wrong and Mary right.
(c) Could John be right and Mary wrong.

6. In an n × n matrix, call row i and col i a *good* pair if all their entries are positive. For example, if

$$M = \begin{bmatrix} 1 & 2 & 3 & 5 \\ 4 & 2 & \pi & 6 \\ 1 & 6 & 8 & 1 \\ -2 & 9 & 2 & 7 \end{bmatrix}$$

then there are two good pairs, row 2 & col 2 and row 3 & col 3.
Here's an algorithm which finds all the good pairs.

> Go across row 1 and ask "is the entry positive".
> If all yeses, then go down col 1 and ask "is the entry positive".
> If all yeses then you have found a good pair.
> Repeat for each of the n rows of the matrix.

Choose a unit of measure and find the running time of my algorithm.

7. Consider an algorithm to solve a task of size n (e.g., a graph problem with n vertices).

(a) If the algorithm has running time n^2 [operations] and each operation takes 10^{-6} seconds what's the largest size task that can be done in one minute.

(b) If the algorithm has running time n^3 [operations] and each operation takes 10^{-6} seconds what's the largest size task that can be one in one minute.

8.(a) Suppose you want to solve a problem of size n. You have a computer and an algorithm with running time n^3. If you had a choice of getting a computer which is 10 times as fast or an algorithm with running time n^2 which would you prefer if

(a) n = 1000
(b) n = 10
(c) n = 9

SECTION 3.5 NETWORK FLOWS

transport networks

Fig 1 shows a transport network with a flow. The network is a weighted directed graph with no loops. The underlying undirected graph is connected. There is one source (vertex with exits but no entrances) named A, and one sink (entrances but no exits) named Z. The first weight on an edge, always positive, is the carrying capacity of the edge. The second weight is non-negative and is the actual flow on the edge (if there is no second weight it's assumed to be 0). In Fig 1, edge CZ has capacity 4 and flow 2 (say in gallons).

The flow through an edge can't exceed the edge capacity and there must be conservation of flow at a vertex in the following sense.

(1) For any vertex other than A and Z, the flow in equals the flow out

For example, at B in Fig 1 the flow in (from A and E) is 1 + 1 = 2 and the flow out (to C) is 2.

value of the network flow

(2) It can be shown that in the flow out of A equals the flow into Z.

This quantity is called the *value of the network flow*.
For example, in Fig 1 the flow out of A (to B and D) is 1 + 3 = 4 and the flow into Z (from C and E) is 2 + 2 = 4; the value of the network flow is 4.

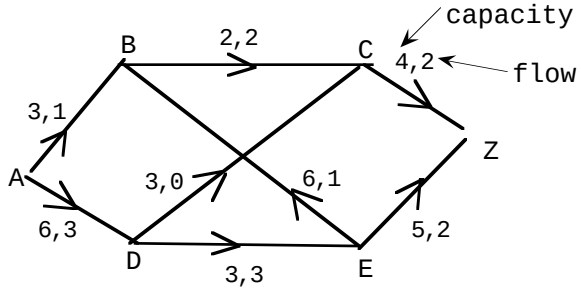


FIG 1

cuts

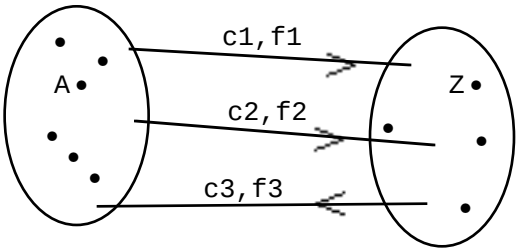
If the vertices of the network are divided into two groups, one containing A and the other containing Z, then the set of all edges from one group to the other is called a cut (Fig 2).

The *capacity* of the cut in Fig 2 is $c_1 + c_2$, *the amount that can be carried from the A side to the Z side*.

The *cut flow* in Fig 2 is $f_1 + f_2 - f_3$; *add the flow going from the A side to the Z side and subtract anything going the other way*.

> **warning**
> To get the cut *capacity*, IGNORE the edges going the "wrong" way (i.e., from the Z-side to the A-side)
> To get the cut *flow,* SUBTRACT the edges going the wrong way.

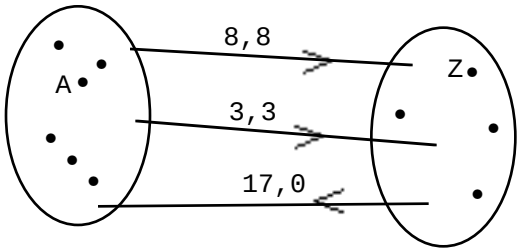
A cut is *saturated* if the cut flow equals the cut capacity (Fig 3), i.e., if the edges from the A side to the Z side carry max flow and the edges from the Z side to the A side carry zero flow.



cut capacity $c_1 + c_2$

cut flow $f_1 + f_2 - f_3$

FIG 2



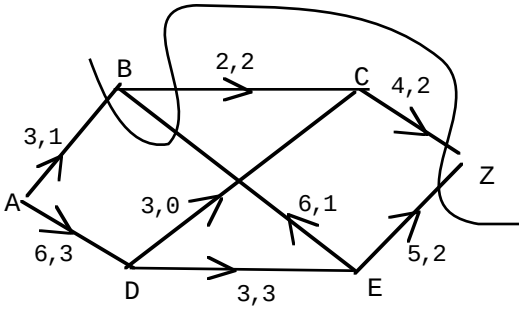
saturated cut

FIG 3

(3) *It can be shown that all cuts carry the same flow which in turn is the same as the value of the network flow itself.* (This is a generalization of (2).)

Intuitively, the network flow must be carried across every cut since however you choose to divide up the vertices to form a cut, the network flow must get from the A-side to the Z-side.

Fig 4A shows Fig 1 again along with the cut separating A, D, E, C from B, Z.



cap = AB + EB + CZ + EZ
= 3 + 6 + 4 + 5
= 18

flow = AB + EB + CZ + EZ - BC
= 1 + 1 + 2 + 2 - 2
= 4

FIG 4A

Fig 4B shows some more cuts in the network of Fig 1.
All the cuts have flow 4. And the network flow is 4.

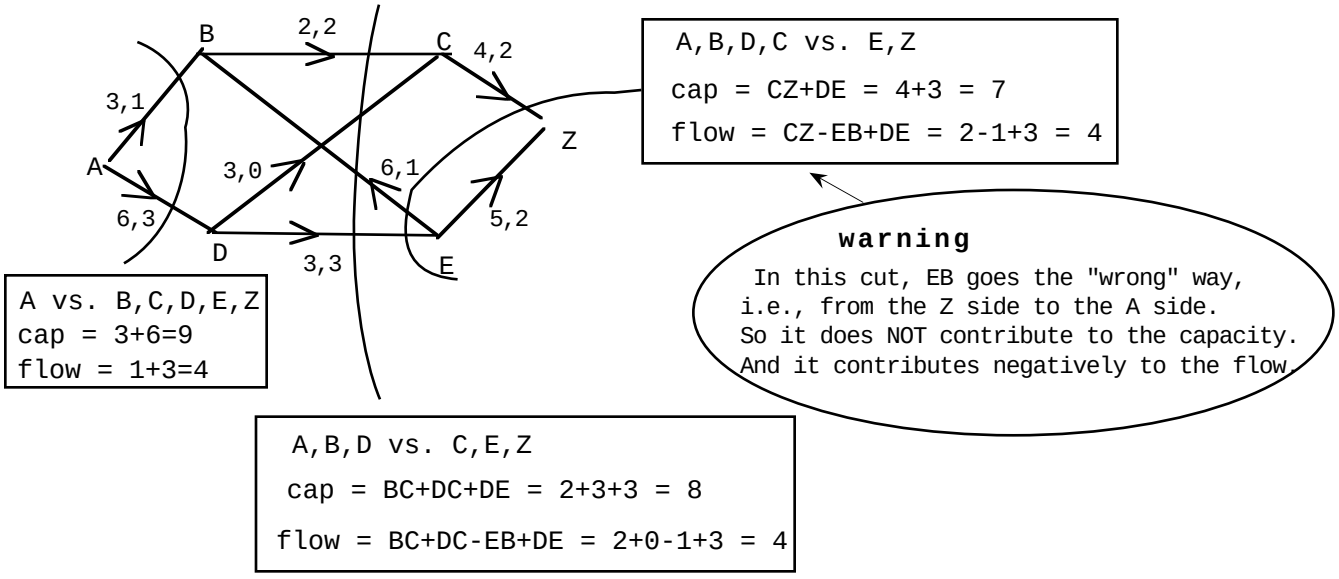


FIG 4B

max flow

Look at a transport network and all the possible cuts. By (3), the network flow must go across every cut, so *once some cut is saturated, the max network flow is achieved*.

(Furthermore *the maximum flow possible through the network equals the minimum of all the cut capacities*---can't do better than the weakest link.)

Fig 5 shows one possible max flow---you know it's max because of the saturated cut. The purpose of this section is to show you how to get it.

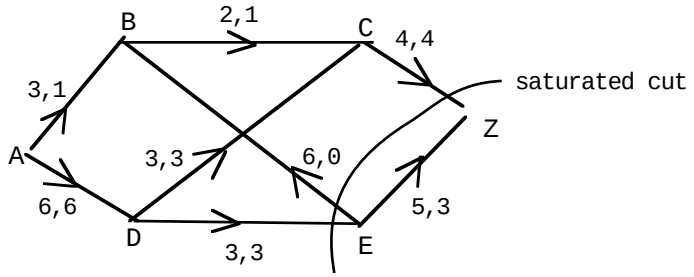


FIG 5 max flow

preview of the labeling algorithm for finding a max flow

Suppose you have an initial flow (maybe all zero). To get a max flow, aim to increase the flow until some cut is saturated. In a small diagram you can do it by inspection. The labeling procedure is designed to do it mechanically.

Look at the network in Fig 6.

By inspection you can increase the flow by sending 4 gallons on path ADZ (can't send more on this particular path since edge DZ only has room for 4 more gallons).

Here's how to get the corresponding result by a labeling procedure instead of inspection.

Give A the label $(\cdot,\infty)$ to indicate that A is a bottomless source (Fig 7).

Once A is labeled, you can label D by thinking as follows: A can supply any amount, edge AD can carry 6 more to D, so give D the label $(A^+,6)$ to indicate that D can get 6 gallons from A (Fig 7).

Once D is labeled you can label Z: The label on D indicates that D can supply 6 more, edge DZ can carry 4 more to Z, so give Z the label $(D^+,4)$.

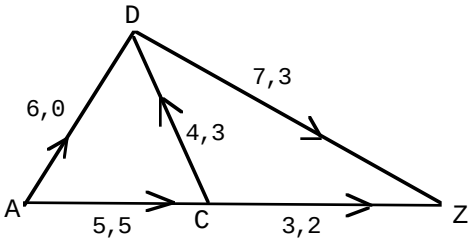


FIG 6 Initial flow

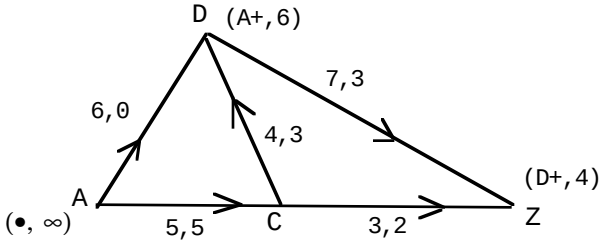


FIG 7 Round 1 Labels

Once Z is labeled, trace the labels from Z back to A and augment the flow on the path ZDA by 4, the amount on the Z-label (Fig 7A).

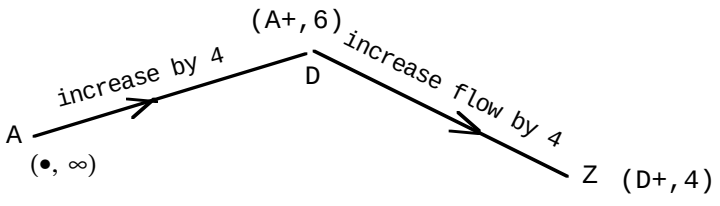


FIG 7A

This produces the new flow in Fig 8, a result you saw earlier by inspection. But now you have a method that works on a large, not easily-inspected network.

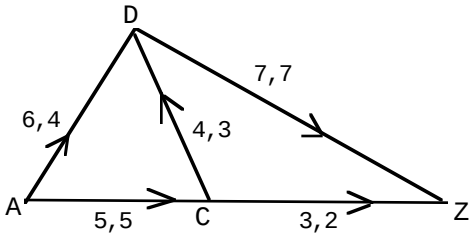


FIG 8 Round 1 Augmented flow

Let's try to increase the flow again
By inspection of Fig 8 you can't send more to Z conventionally on paths ACZ, ADZ and ACDZ since edges AC and DZ are saturated. But we can send 1 gallon *less* from C to D and keep the flow at D conserved by taking an extra gallon from A to replace it. Then C will have an extra gallon to send to Z, increasing the network flow.

Here is the labeling process corresponding to this idea.
Begin with the A-label $(\cdot,\infty)$ again (Fig 9).
Give D the label $(A^+,2)$ since D can get 2 extra gallons from A.
Once D is labeled, you can give C a label as follows: Right now, C is sending 3 gallons to D but 2 of the gallons can be *retracted* since the D label shows that D can get 2 more from A to make up the loss. So give C the *negative* label $(D^-,2)$.
Once C is labeled you can label Z as follows: The C label indicates that C can get 2 extra gallons, edge CZ can carry 1 more to Z, so give Z the label $(C^+,1)$.
Now that Z is labeled, trace the labels back to A: Change the flow on each edge of the path ZCDA by 1, the amount on the Z label. The negative label $(D^-,2)$ on vertex C means that the flow on edge CD should be *decreased* by 1. The other edge flows are *increased* by 1. Fig 10 shows the augmented flow. (Note that we still have conservation of flow at every vertex.)

Since this network is small, you can probably spot a saturated cut in Fig 10 by inspection and realize that the flow in Fig 10 is maximum. But you can get a saturated cut automatically by continuing the algorithm. In Fig 11, I found labels for A, D, C but got stuck trying to label Z. When you get stuck in the labeling process the cut separating the labeled vertices from the unlabeled (just Z here) will be saturated, indicating that the flow is maximum and the algorithm is over. The max flow is in Fig 11. The value of the max flow is 10 (the flow across the saturated cut -- also the flow out of A and also the flow into Z).

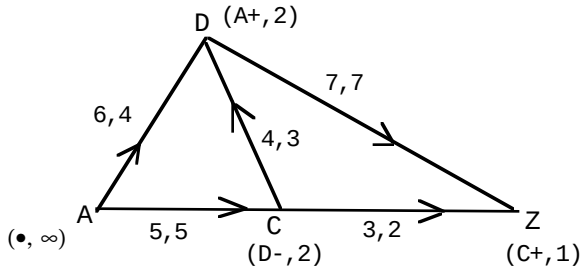


FIG 9 Round 2 Labels

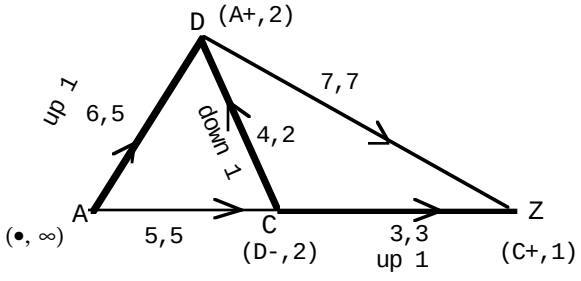


FIG 10 Round 2 Augmented flow

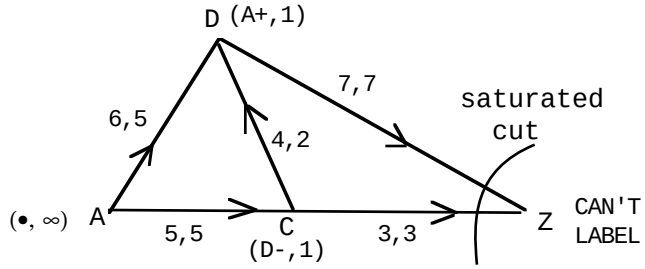


FIG 11 Round 3 max flow

the labeling algorithm in general

Here are the general steps in the algorithm.

Begin with the A-label (·,∞)
Then continue labeling using the rules illustrated in Figs 12 and 13. The aim is to label a path from A to Z. The labeling process is not unique; some networks can be labeled in more than one way.

Fig 12 illustrates the rule for assigning positive labels.

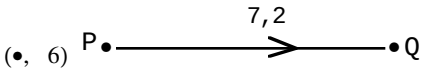


FIG 12 Q gets label (P+, 5)

In Fig 12, P's label indicates that P can get 6 more gallons (doesn't matter from where).

Edge PQ has room to carry 5 more to Q.

Take the smaller of 5 and 6 and give Q the label (P^+, 5) to indicate that Q can get 5 more from P.

In general, a vertex Q can get a positive label when an edge leads from a labeled vertex to Q and that edge is not saturated.

Fig 13 illustrates the rule for assigning negative labels.

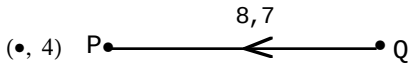


FIG 13 Q gets label (P^-, 4)

In Fig 13, Q can take back 4 of the 7 it is sending to P because P's label indicates it can get 4 extra gallons to make up for a loss.

So give Q the label (P^-, 4)

(The amount on the Q label is the smaller of the flow 7 on edge QP and 4 on the P label but it's better if you understand it rather than memorize something.)

In general, a vertex Q can get a negative label when an edge leads from Q to a labeled vertex and that edge carries nonzero flow (which can be retracted)

Here are the two typical situations where Q can't get a label.

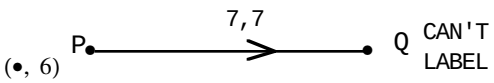


FIG 14

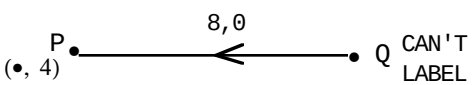


FIG 15

In Fig 14, Q can't get a positive label because edge PQ is filled.
In Fig 15, Q can't get a negative label because Q is sending zero gallons to P (there is nothing to retract).

Here's how to use the labels to augment the flow.
Once Z is labeled, trace the labels back to A and augment the flow (either up or down) by the amount on the Z-label as illustrated in Fig 16.

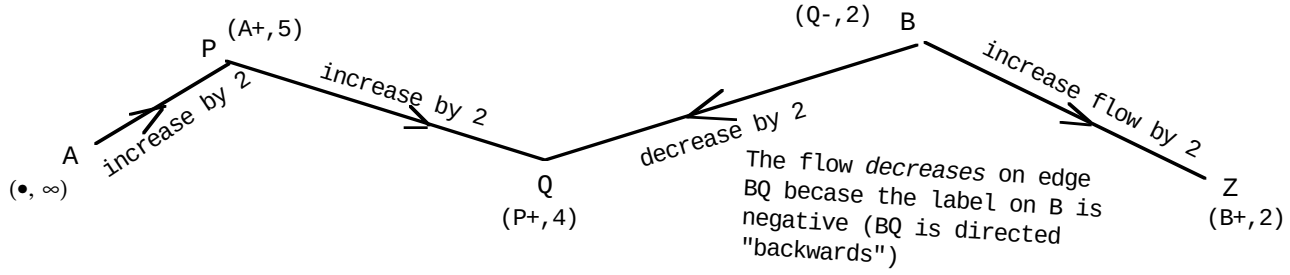


FIG 16 Augmenting the flow

warning

Vertex Q in Fig 16 has label $(P^+,4)$ but the flow is augmented on edge PQ by 2, not 4. The amount of the change is determined by the Z-label, $(B^+,2)$, and all edges are augmented by the same amount (some up, some down).

The algorithm continues until the labeling process gets stuck before reaching Z. In that case the cut separating the labeled vertices from the can't-be-labeled vertices is saturated and the flow is max.

why the algorithm works

Here's why the cut between the labeled and the can't-be-labeled vertices must be saturated.

In Fig 17, if say pipeline XR weren't filled to capacity then R could get a positive label (contradiction) and if say edge TY weren't carrying zero flow then T could get a negative label (contradiction). So XR carries as much as it can and TY carries zero flow.

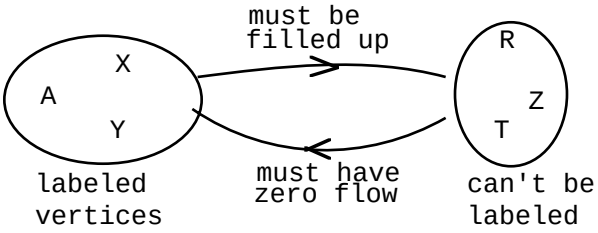


FIG 17

example 1

I'll continue from Fig 18 and find a max flow, a saturated cut to verify that it's a max flow, and the actual value of the max flow.

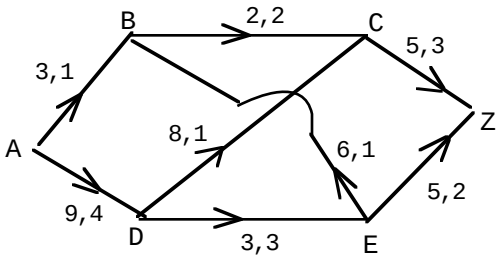


FIG 18

round 1

After A gets the label (·, ∞) here's one way to continue.

There is room on edge AD for 5 more gallons and A has infinitely many gallons available so D can get label (A+, 5) (Fig 19a)

There is room on edge DC for 7 more gallons but D can supply only 5 so C gets label (D+, 5).

There is room on edge CZ for 2 gallons and C can supply 5 so Z gets label (C+, 2).

footnote

Can also give B the label (A+,2), E the label (B−,1) and Z the label (E+,1). But I'll just use the labels I have in Fig 19a.

Follow the labels back from Z to A (Fig 19b). On the path ZCDA, augment each flow by 2, the amount on the Z label (Fig 19b). All flows *increase* because all the labels are positive

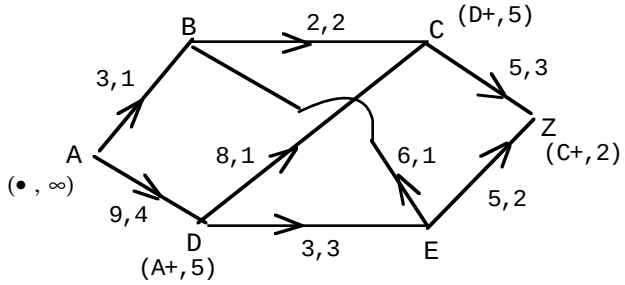


FIG 19a Label a path to Z

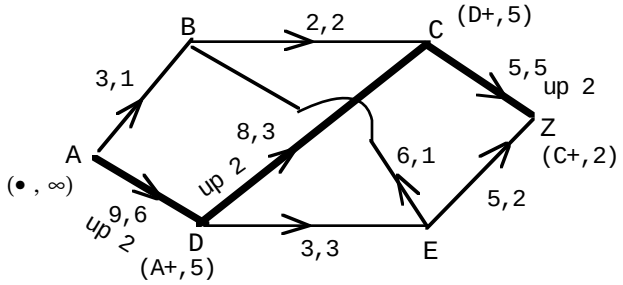


FIG 19b Augment along that path

round 2

Continue from Fig 19b.

A gets label (·, ∞).

There is room on edge AB for 2 gallons so B gets label (A+, 2)

Similarly D gets the label (A+, 3)

Even though B has a label, C can't get a B label since edge BC is full but C can get a D label. Doesn't help since edge CZ is saturated.

Since B can get 2 gallons from A, E can take back the gallon it is sending to B so E can get the negative label (B−, 1)

There is room on edge EZ for 3 gallons but E can supply only 1 so Z gets label (E+, 1) (Fig 20a).

On the path ZEBA (Fig 20b) augment the flow by 1, the amount on the Z label (Fig 20b). The flow on edge EB goes *down* by 1 because E has a negative label (this *increases* the flow from A to Z since edge EB goes the "wrong" way.)

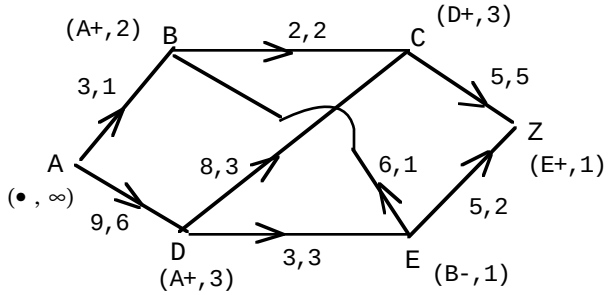


FIG 20a Label a path to Z

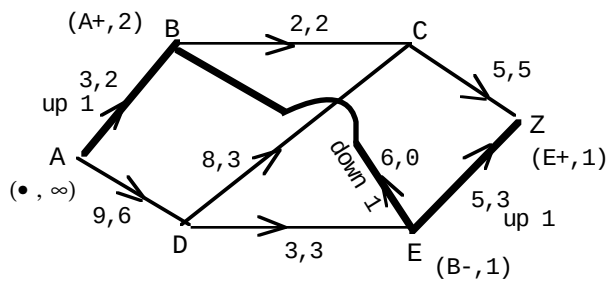


FIG 20b Augment along that path

round 3

I labeled everything I could but couldn't give E and Z labels. The cut separating E and Z from the other vertices is saturated and the algorithm is over.

Fig 21 shows a max flow. Value of the max flow is 8, the flow on the saturated cut (also the flow out of A, also the flow into Z).

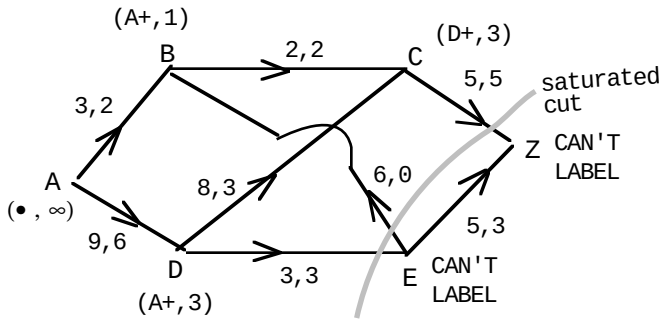


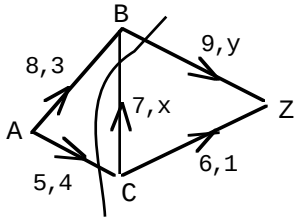
FIG 21 Try to label a path to Z

warning

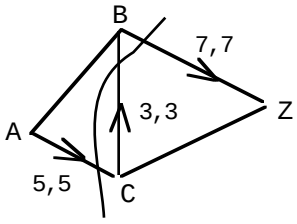
The algorithm isn't finished *until you label as many vertices as you can but eventually get stuck* and can't label Z, in which case the saturated cut appears automatically as the *cut between the set of labeled vertices and the set of can't-label vertices*. If you find this or any other saturated cut by inspection then you are bypassing the algorithm. You can get away with this in small networks but not in large networks and not on exams.

PROBLEMS FOR SECTION 3.5

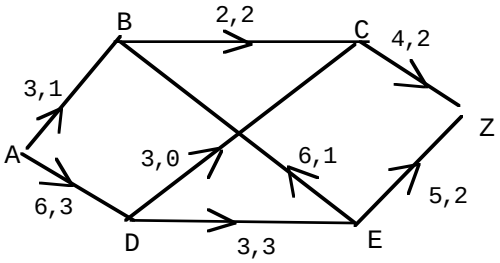
1. (a) Find flows x and y.
   (b) Find the capacity and flow across the indicated cut.
   (c) Find (effortlessly) the flow across every other cut.



2. Is the cut in the diagram saturated.

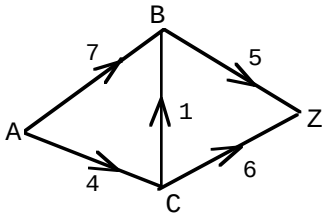


3. Look at the following network flow.

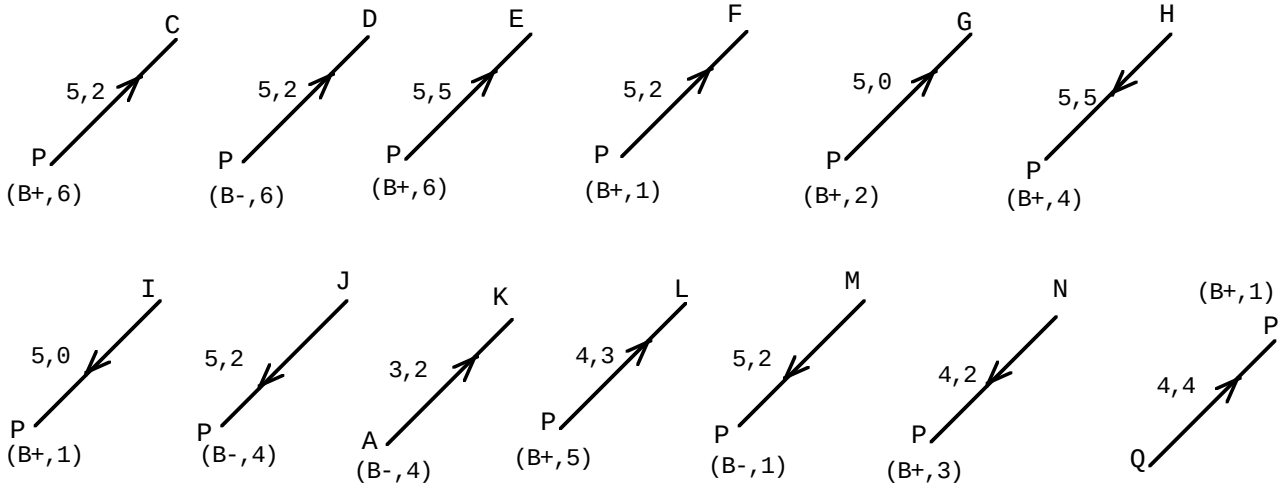


(a) Find the capacity and flow for the following two cuts.
(i) A,B,D versus the other vertices
(ii) A,E versus the others
(b) (a counting problem) How many cuts are there.

4. By inspection, find all cut capacities in the diagram below (the weight on each edge is the edge capacity), find a min cut and hence a max flow value for the network. And by inspection find an actual max flow.



5. Put down labels whenever possible.

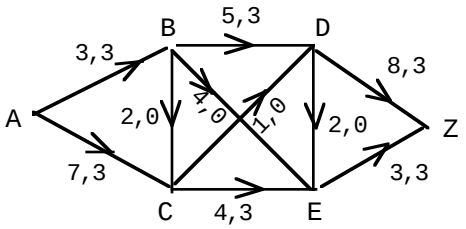


6. Can Z ever get a negative label.

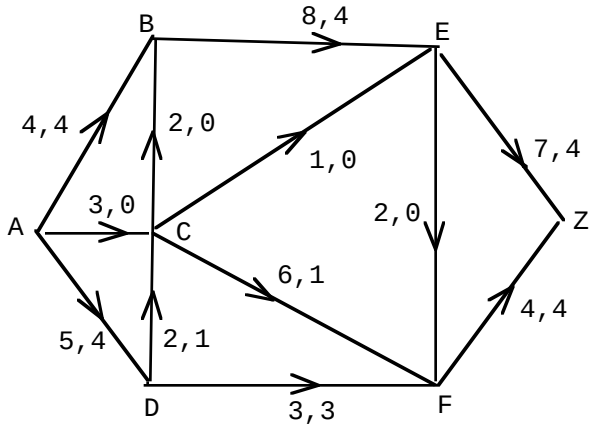
7. Go back to Fig 21 which says Z can't be labeled. There is room on edge EZ for 2 more gallons so why can't Z get the label $(E^+, 2)$?

8. Use the labeling algorithm to find a max flow and a saturated cut.
(Your work may differ from mine because labels are not unique but we should at least agree at the end on the *value* of the max flow.)

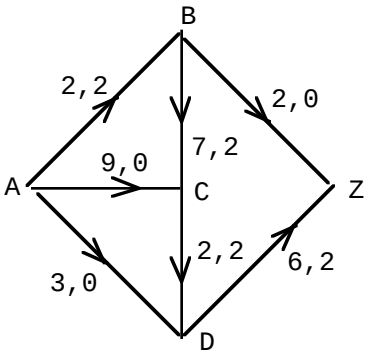
(a)



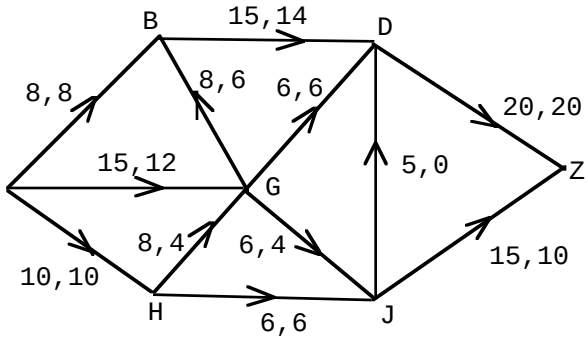
(b)



(c)



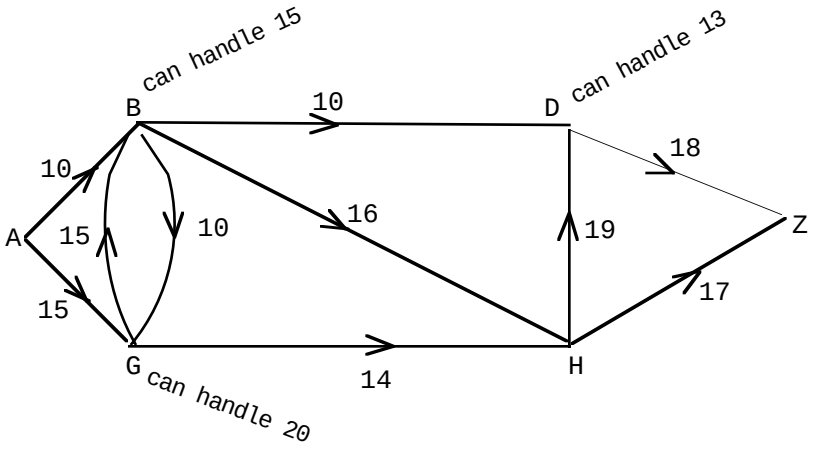
(d)



9. In the diagram below, in addition to *edges* with carrying capacities, several *vertices* have capacities as well; B can handle at most 15 gallons, G at most 20 and D at most 13 gallons.

So the diagram is not a transport network. But see if you can be clever and turn it into a transport network. (And then stop.The purpose of this problem is to illustrate that a problem that does not start out as a traditional network flow problem can be turned into one.)

Suggestion. Think of vertex B as an industrial complex with a receiving dock B_1 and a shipping dock B_2 where the road carrying items from B_1 to B_2 has capacity 15.



REVIEW PROBLEMS FOR CHAPTER 3

1. Given the distances in the chart below between cities B, E, F, G, I, S, T.
(a) Use the appropriate algorithm, to find a highway system that connects them minimally. How may miles in the minimal system.

(b) Suppose E and I must be directly connected. Find a minimal highway system with this restriction.

	B	E	F	G	I	S
E	119					
F	174	290				
G	198	277	132			
I	51	168	121	153		
S	198	303	79	58	140	
T	58	113	201	164	71	196

2. Here's the result of several rounds of Dijkstra's algorithm for shortest paths from A.

	B	C	D	E	F
status	√	√	x	x	√
link	A	B	F	C	A
dist	2	4	4	6	1

Fill in the blanks.
The shortest distance between A and E _____ is _____.

3. Let M be the adjacency matrix of a digraph with 10 vertices v1,..., v10.
I've run Warshall's algorithm.

(a) (i) What can you conclude if there is a 1 in row 3, col 5 of M_4.
(ii) What can you conclude if there is a 0 in row 3, col 5 of M_4.

(b) Suppose there's a 1 in row 2, col 3 of M_6. Are these conclusions correct.
(i) There's a path from v_2 to v_3.
(ii) You can get from v_2 to v_3 without stepping on v_8.
(iii) You can get from v_2 to v_3 but you'll have to step on v_6 along the way.

(c) Suppose there's a 0 in row 2, col 3 of M_6. Are these conclusions correct.
(i) There's no path from v_2 to v_3.
(ii) There might be a path from v_2 to v_3 but if so you'll have to step on at least one of v_7, v_8, v_9, v_{10} along the way.

4. Start with a directed graph with n vertices. A vertex v in a digraph is called a *hub* if there is an edge from v to every other vertex and an edge from every other vertex to v.
(a) Make up an algorithm which examines the adjacency matrix of a digraph (without loops) with n vertices and finds all hubs. Describe your algorithm in simple English.
(b) Find the running time of your algorithm (after choosing an appropriate unit of measure).

5. Find M_∞ if

$$M = \begin{matrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{matrix}$$

6. Here's a weight matrix for a digraph and a table from round 1 of Dijkstra's algorithm for finding shortest distances from P. Go one round further and state whatever conclusions you can at that stage.

	A	B	C	D	E	F	G	P
A	0	∞	∞	∞	∞	∞	∞	∞
B	2	0	∞	∞	∞	∞	∞	∞
C	∞	1	0	∞	∞	∞	4	∞
D	∞	∞	3	0	∞	∞	8	∞
E	∞	∞	∞	∞	0	3	∞	∞
F	2	∞	∞	8	∞	0	∞	∞
G	∞	6	∞	∞	∞	∞	0	∞
P	10	∞	∞	∞	4	∞	∞	0

A	B	C	D	E	F	G
x	x	x	x	√	x	x
P	P	P	P	P	E	P
10	∞	∞	∞	4	7	∞

7. Use the appropriate algorithm to find a max flow and a saturated cut.