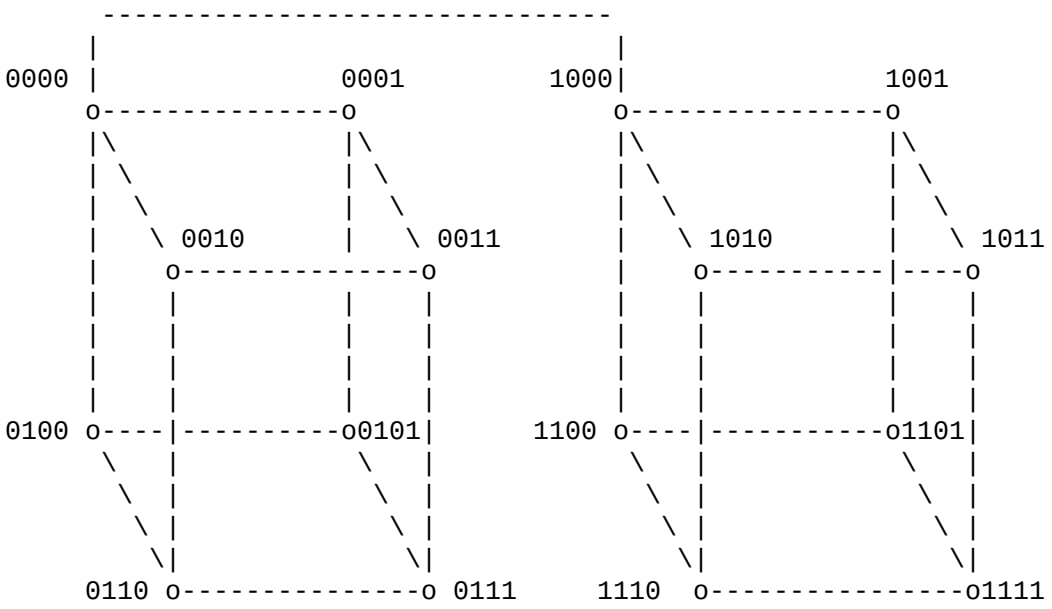
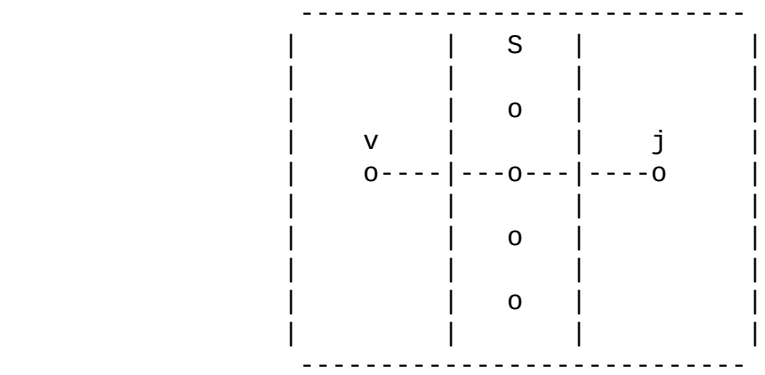


Figure 6-23. Four-Cube with a Single Fourth Dimensional Connection Shown.



S Disconnects G as well as v and j.

Figure 6-21. VC(G) Calculation.

SECTION 6-8

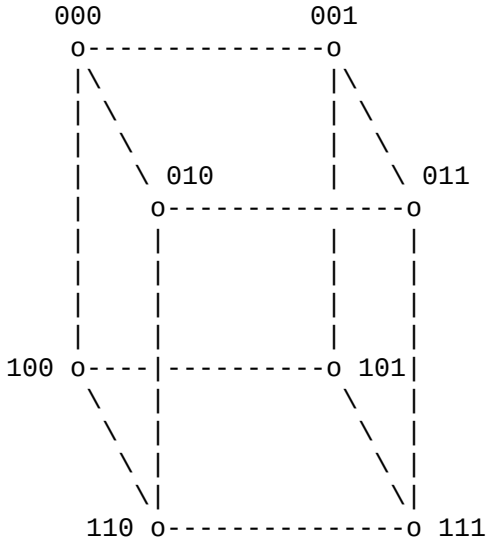
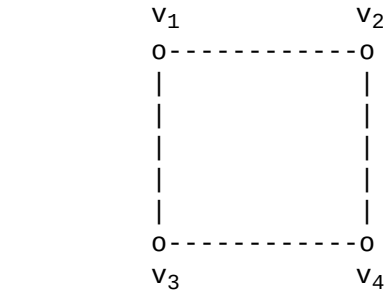
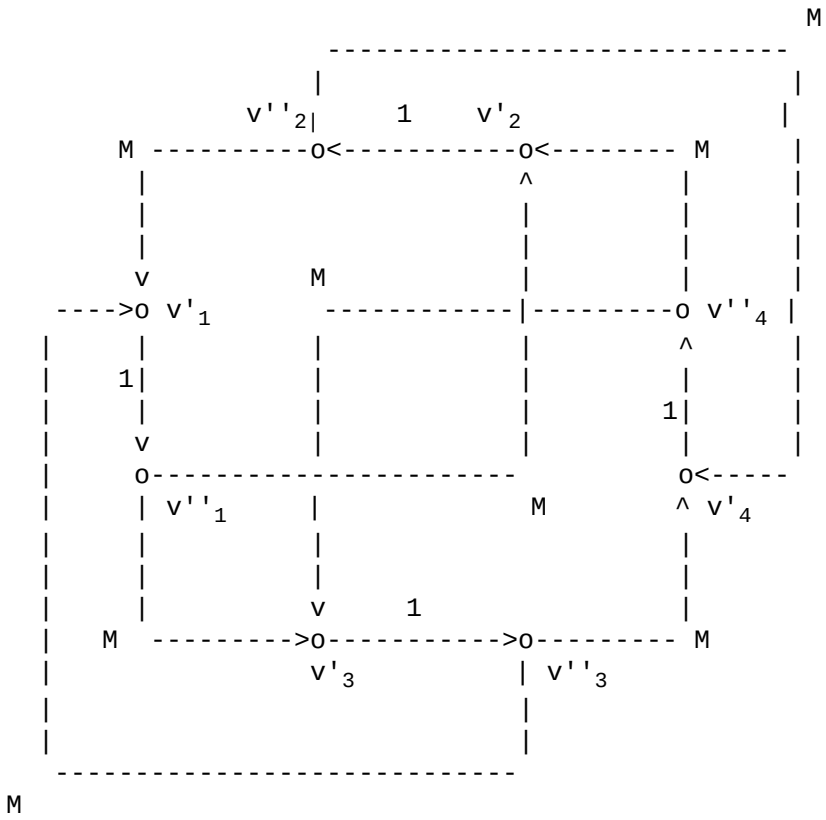


Figure 6-22. Three-Cube on Eight Vertices.

SECTION 6-7

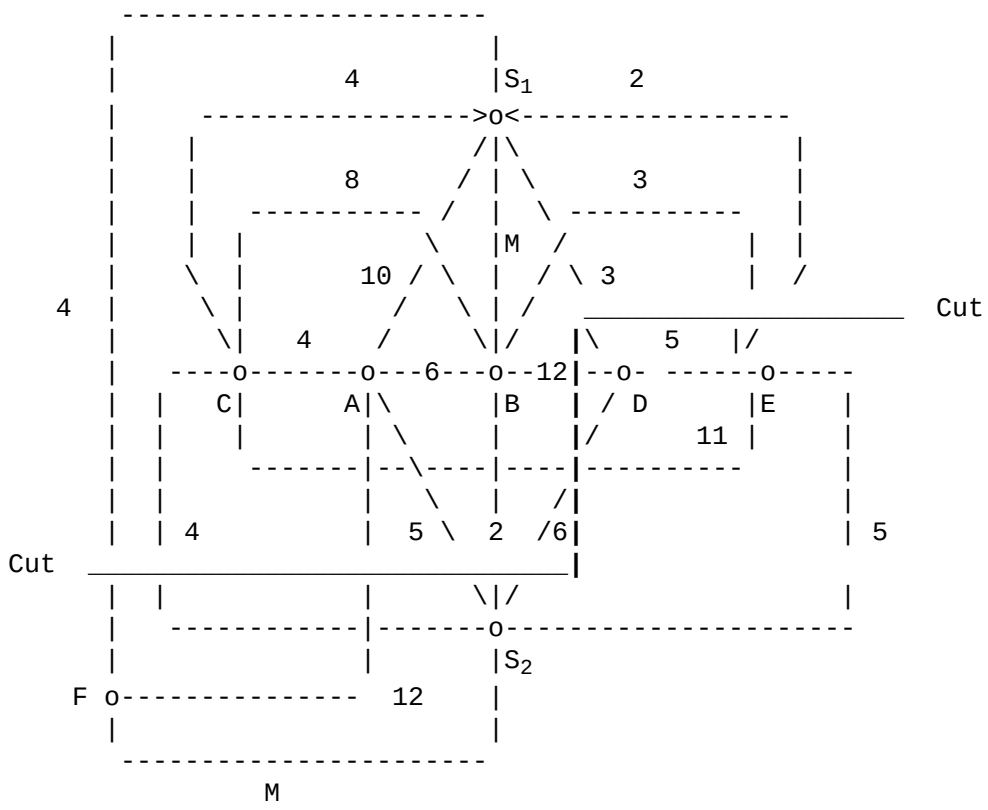


(a) Graph G to be Transformed.



(b) Flow Network G'.

Figure 6-20. Associated Flow Network.

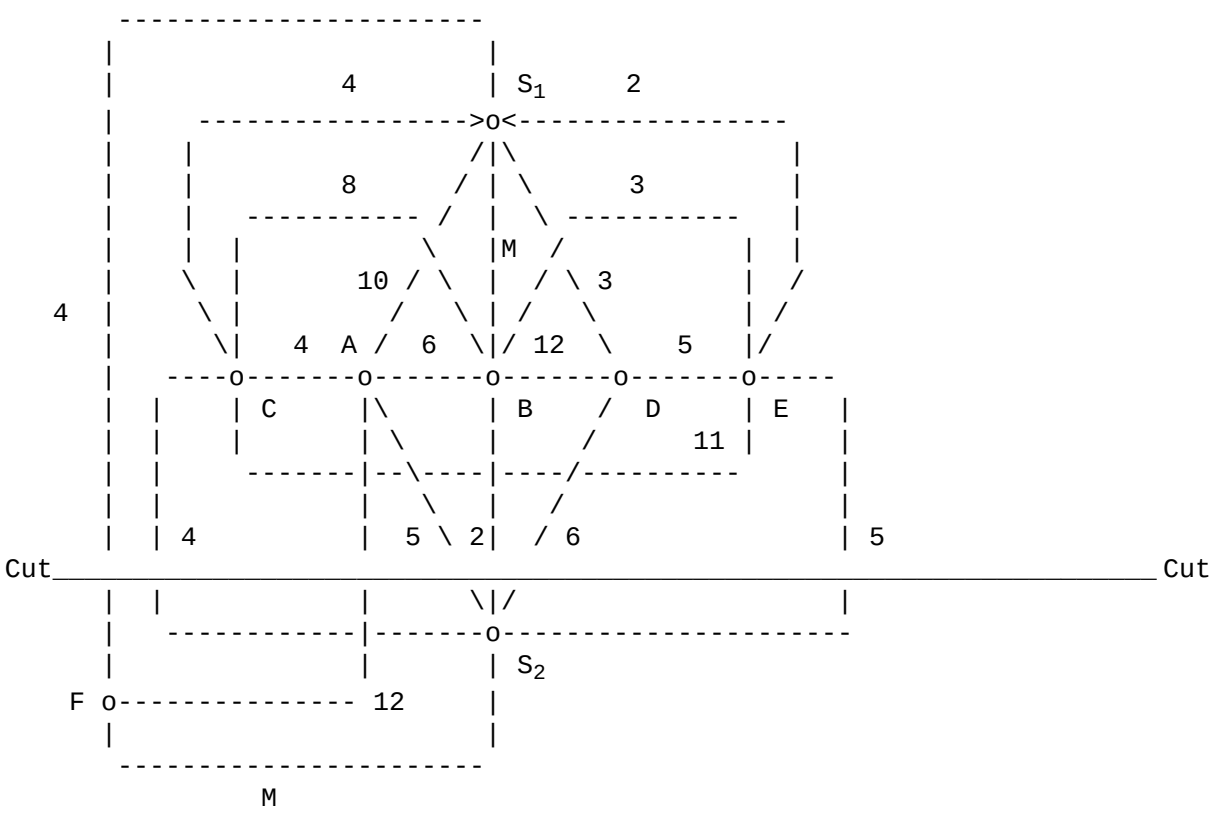


(a) Suboptimal Assignment with Non-minimum Cut.

Processor	Task
P1	A B C
P2	D E F
Execution Cost	20
Communication Cost	38
Assignment Cost	58

(b) Assignment Cost for Suboptimal Assignment.

Figure 6-19. Suboptimal Assignment.



Cut: $X = \{S_1, A, B, C, D, E\}$, $X^c = \{S_2, F\}$

(a) Optimal Assignment with Minimum Cut.

Processor	Task
P1	A B C D E
P2	F
Execution Cost	26
Communication Cost	12
Assignment Cost	38

(b) Assignment Cost of Optimal Assignment.

Figure 6-18. Optimal Assignment.

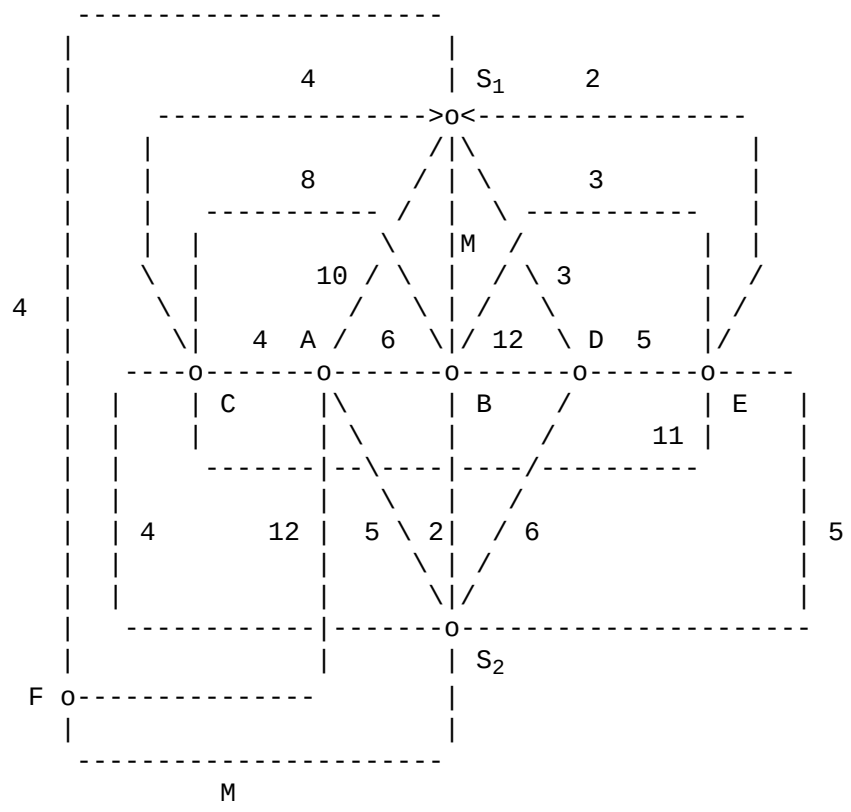
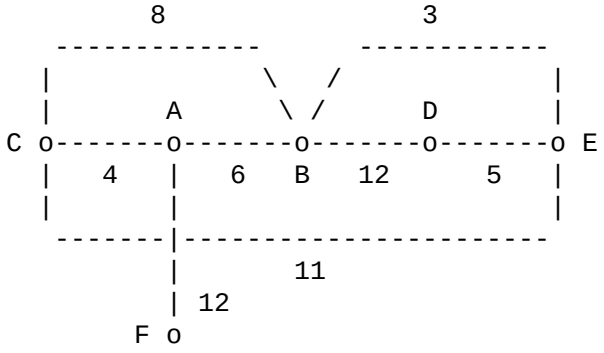


Figure 6-17. Module Assignment Graph.

SECTION 6-6

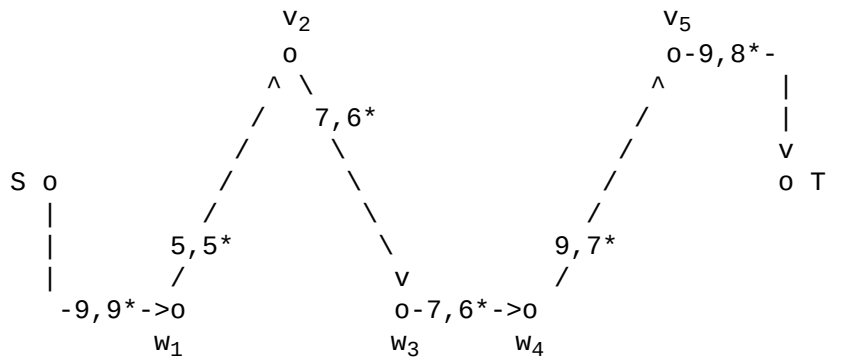


(a) Inter-Module Communication Cost Graph.

Module	P_1 Cost	P_2 Cost
A	5	10
B	2	M (denotes a large value)
C	4	4
D	6	3
E	5	2
F	M	4

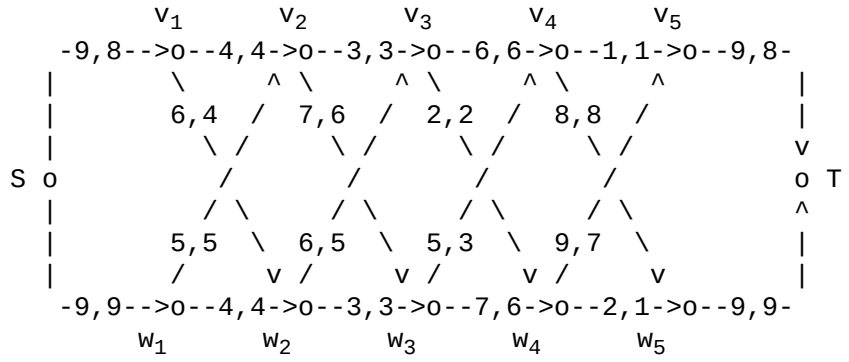
(b) Module-Processor Execution Cost Table.

Figure 6-16. Communication and Execution Costs for Modules.



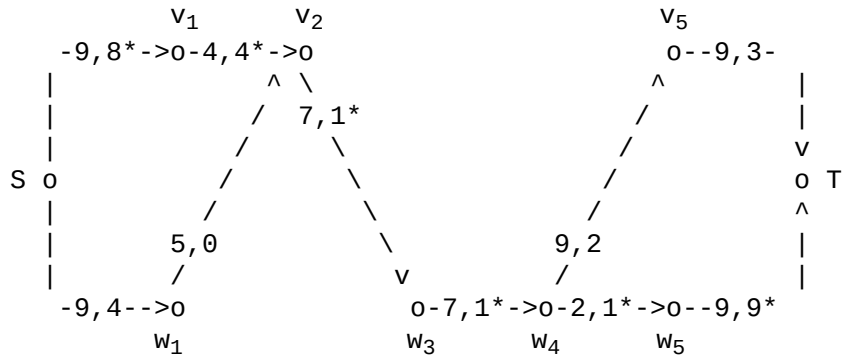
(h) Blind Flow Routing Through w_1.

Figure 6-14. Maximal Flow on Layered Network of (a).

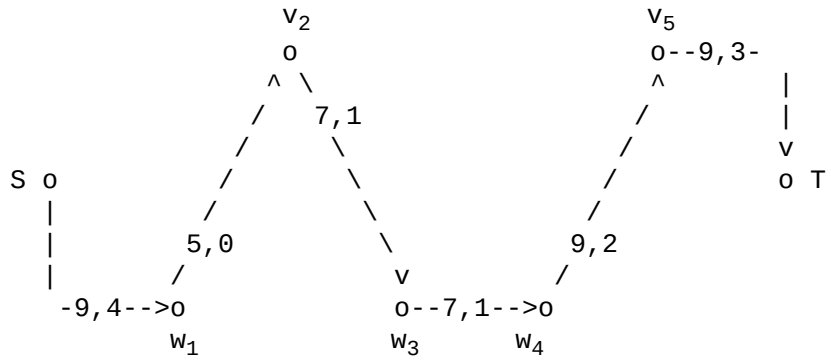


Edge Notation (Capacity, Flow)

Figure 6-15. Reconstituted Maximal Layered Flow Network.



(f) Blind Flow Routing Through w_5.



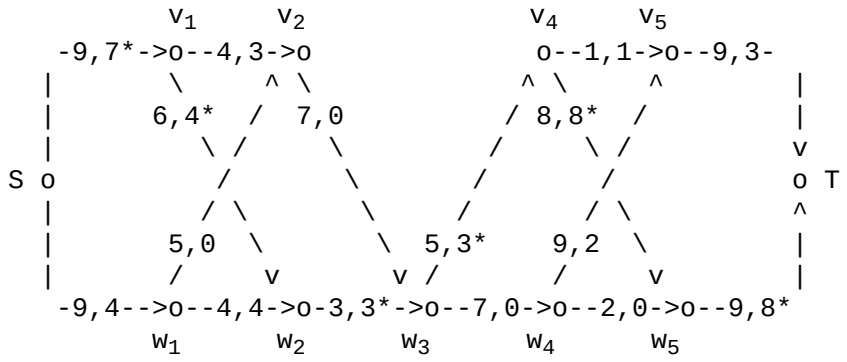
Deleted Vertices

w_5, v_1

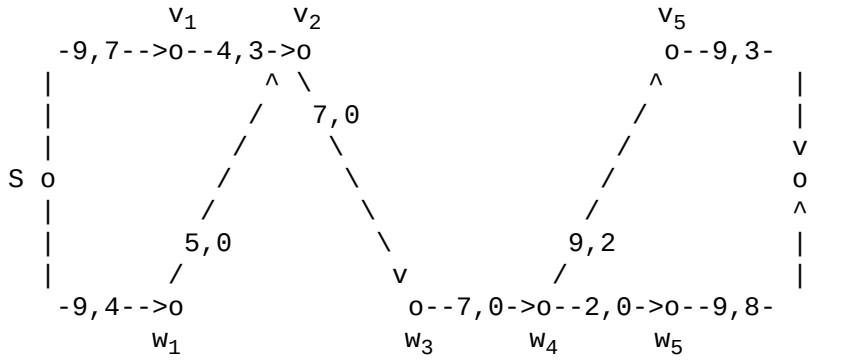
Edge	Flow
(w_4,w_5)	1
(w_5,T)	9
(S,v_1)	8
(v_1,v_2)	4

Minimum Thruput Vertex w_1 of Thruput 5.

(g) Elimination of Zero Thruput Vertices w_2, w_5.



(d) Blind Routing Through v_4.



Store Deleted Elements

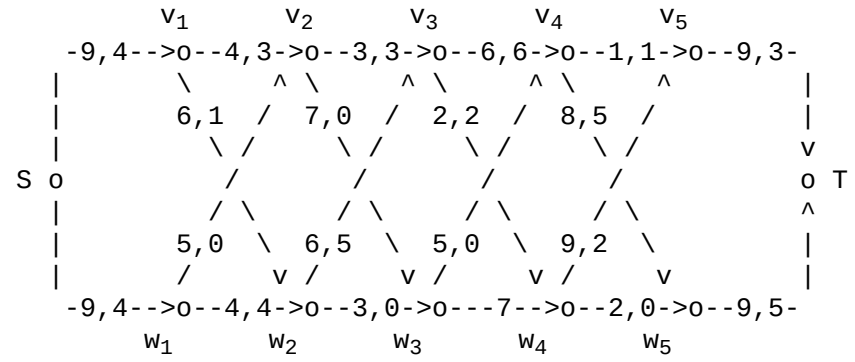
v_4 Level 4

w_2 Level 2

Edge	Flow
(w_3,v_4)	3
(v_4,v_5)	1
(v_4,w_5)	8
(v_1,w_2)	4
(w_1,w_2)	4
(w_2,w_3)	3

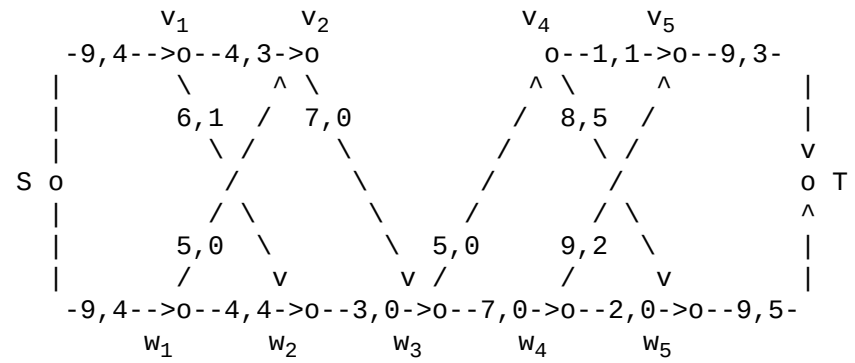
Minimum Thruput Vertex w_5, Thruput 1

(e) Elimination of Vertices of Zero Thruput.



Edge Notation (Capacity, Flow)

(b) Blind Flow Routing Through v_3.



Deleted Elements

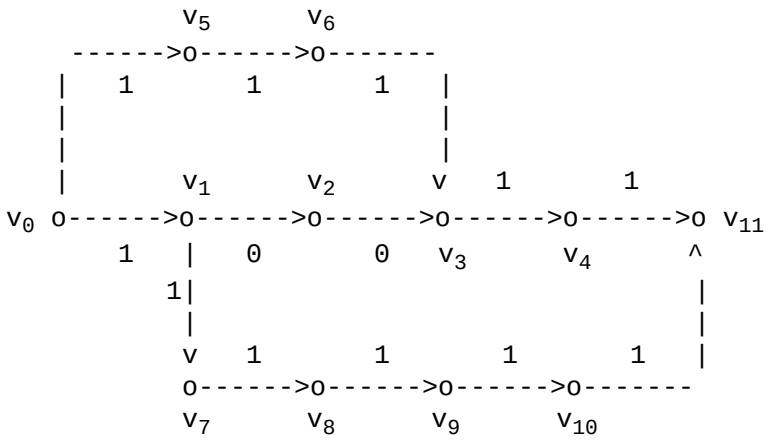
v_3 Level 3

Edge	flow
(v_2,v_3)	3
(w_2,v_3)	5
(v_3,v_4)	6
(v_3,w_4)	2

Minimum Thruput Vertex v_4 of Thruput 3

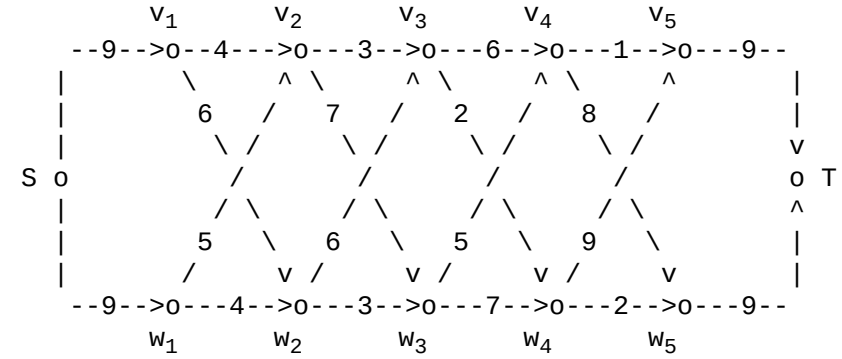
(c) Elimination of Vertices of Zero Thruput.

(f) Second Augmenting Layered Network ALN'.



(g) N After Applying ALN' Flow.

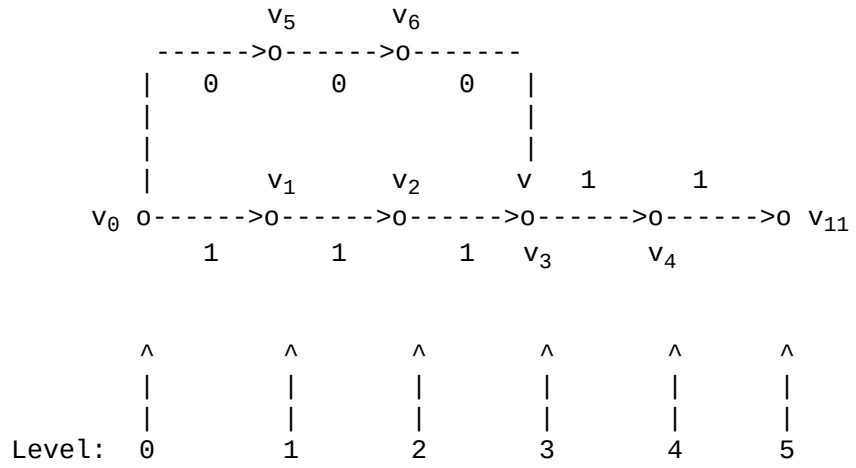
Figure 6-13. Example of Dinic's Algorithm.



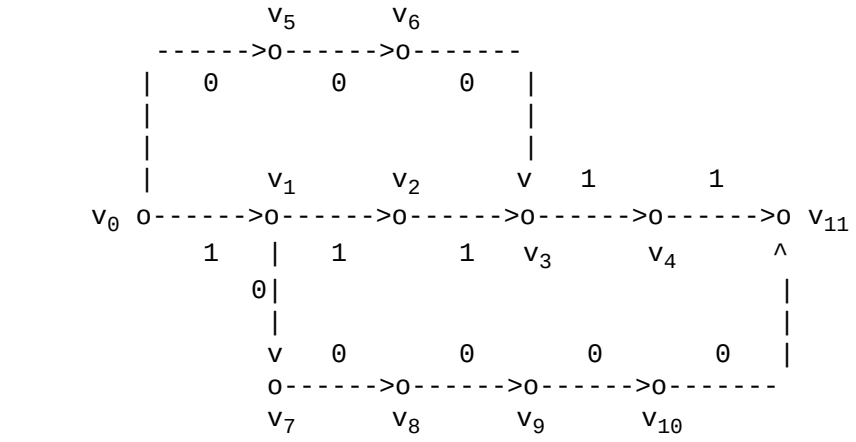
Edges Labelled with Capacities.

Minimum Thruput Vertex v_3 of Thruput 8

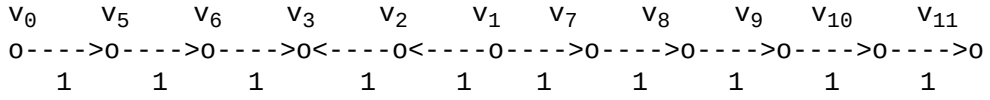
(a) Augmenting Layered Network Showing Capacities.

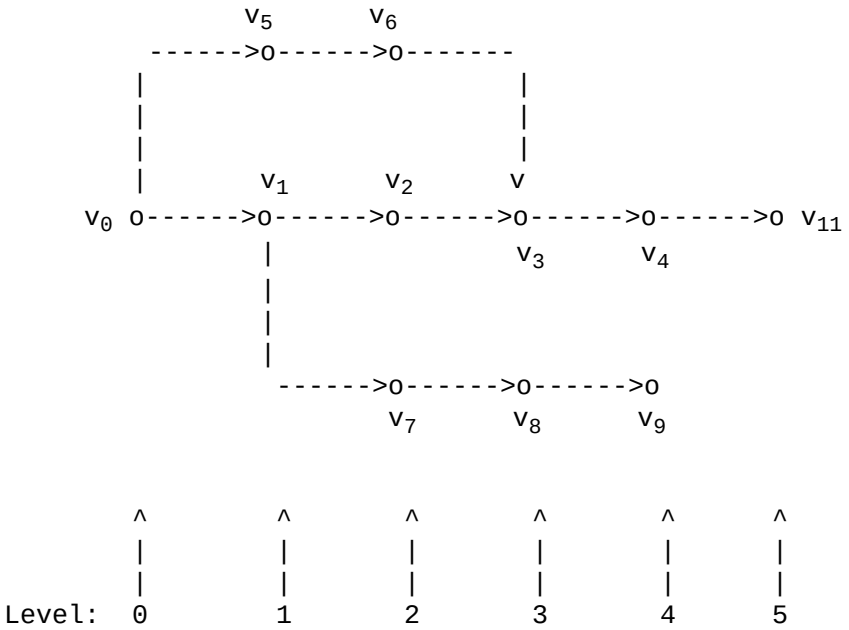


(d) ALN Showing Maximal Flow.

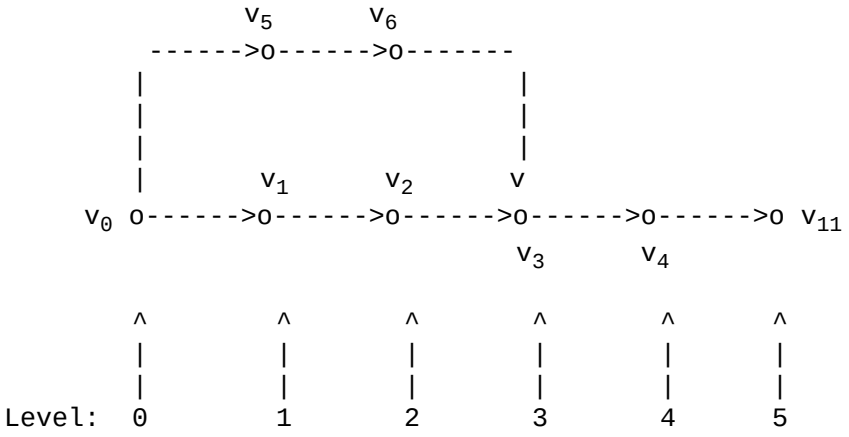


(e) N After Applying ALN Flow.





(b) First ALN Before Elimination of Zero Thruput Vertices.



(c) First ALN After Elimination of Zero Thruput Vertices.

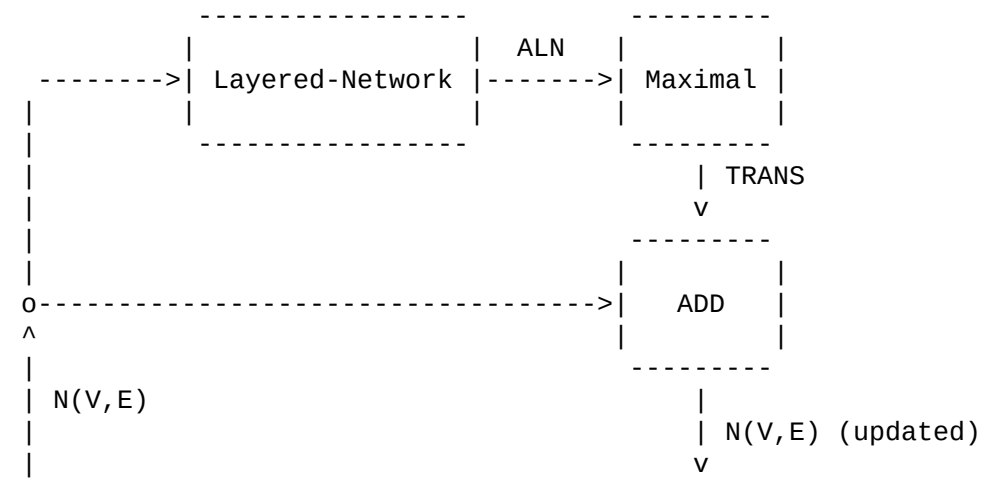
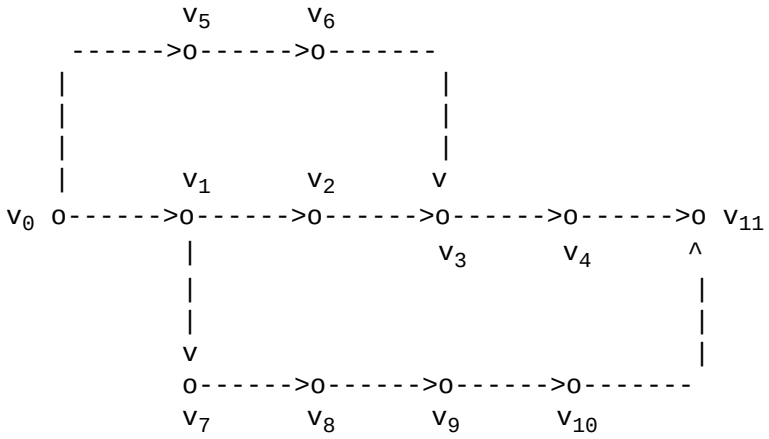


Figure 6-12. Data Flow Diagram for Dinic's Algorithm.



(a) Capacitated Network N with Unit Capacities and Zero Flows.

v_3	v_1,v_2,v_3	Scan (v_3,v_2)		v_2 in Tree
v_3	v_1,v_2,v_3	Scan (v_3,v_4) Enqueue (v_4) Add v_4 to Tree	v_4: 2	Breakthrough

Increase Flow on v_1-v_3-v_4 by 2.

Flow_Augmenting_Path (Third Invocation)

QUEUE	TREE	ACTION	AMOUNT	REMARKS
v_1	v_1	Initialize	v_1: M	
v_1	v_1	Scan (v_1,v_2) Enqueue (v_2) Add v_2 to Tree	v_2: 1	
v_1,v_2	v_1,v_2	Scan (v_1,v_3) Enqueue (v_3) Add v_3 to Tree	v_3: 1	
v_1,v_2,v_3	v_1,v_2,v_3	Dequeue(Q)		
v_2,v_3	v_1,v_2,v_3	Scan (v_2,v_3)		v_3 in Tree
v_2,v_3	v_1,v_2,v_3	Scan (v_2,v_4)		Edge saturated.
v_2,v_3	v_1,v_2,v_3	Dequeue(Q)		
v_3	v_1,v_2,v_3	Scan (v_3,v_2)		v_2 in Tree
v_3	v_1,v_2,v_3	Scan (v_3,v_4)		Edge saturated.
v_3	v_1,v_2,v_3	Dequeue(Q)		
Empty				Search Blocked

Figure 6-11. Trace of Successive Calls to Flow_Augmenting_Path.

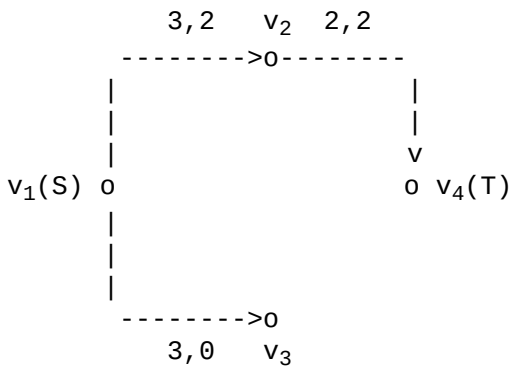
Flow_Augmenting_Path (First Invocation)

QUEUE	TREE	ACTION	AMOUNT	REMARKS
v_1	v_1	Initialize	v_1: M	
v_1	v_1	Scan (v_1,v_2) Enqueue(v_2) Add v_2 to Tree	v_2: 3	
v_1,v_2	v_1,v_2	Scan (v_1,v_3) Enqueue(v_3) Add v_3 to Tree	v_3: 3	
v_1,v_2,v_3	v_1,v_2,v_3	Dequeue(Q)		
v_2,v_3	v_1,v_2,v_3	Scan (v_2,v_3)		v_3 in Tree
v_2,v_3	v_1,v_2,v_3	Scan (v_2,v_4) Enqueue(v_4) Add v_4 to Tree	v_4: 2	Breakthrough

Increase Flow on v_1-v_2-v_4 by 2.

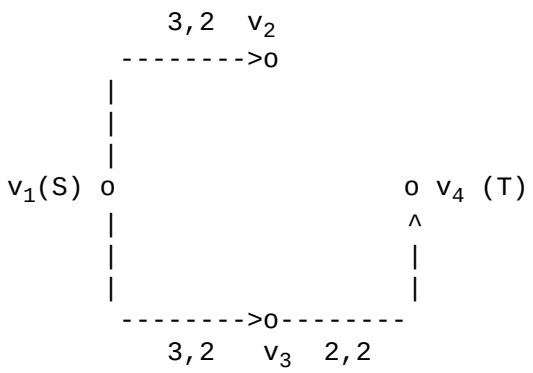
Flow_Augmenting_Path (Second Invocation)

QUEUE	TREE	ACTION	AMOUNT	REMARKS
v_1	v_1	Initialize	v_1: M	
v_1	v_1	Scan (v_1,v_2) Enqueue (v_2) Add v_2 to Tree	v_2: 1	
v_1,v_2	v_1,v_2	Scan (v_1,v_3) Enqueue (v_3) Add v_3 to Tree	v_3: 3	
v_1,v_2,v_3	v_1,v_2,v_3	Dequeue(Q)		
v_2,v_3	v_1,v_2,v_3	Scan (v_2,v_3)		v_3 in Tree
v_2,v_3	v_1,v_2,v_3	Scan (v_2,v_4)		Edge saturated
v_2,v_3	v_1,v_2,v_3	Dequeue(Q)		

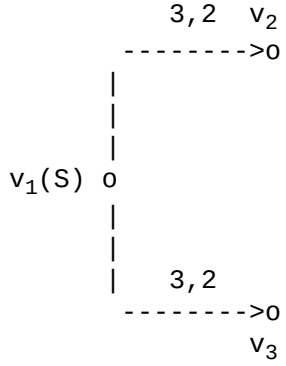


Edge x: (Cap(x), f(x)) - after increase.

(b) First Flow Augmenting Tree.



(c) Second Flow Augmenting Tree.



(d) Blocked Flow Augmenting Tree.

Figure 6-10. Ford Fulkerson Example.

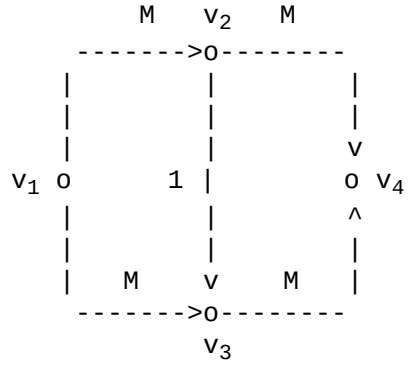
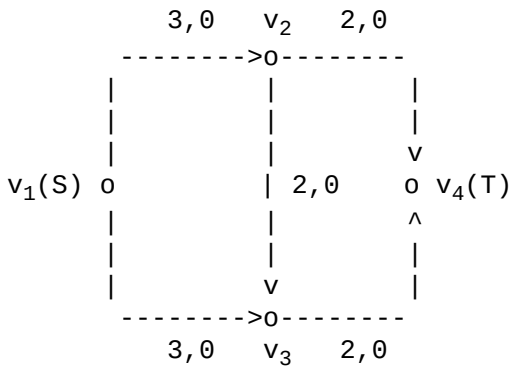


Figure 6-9. Dependency of Performance on Search Technique.



Adjacency list:

v_1: v_2, v_3

v_2: v_3, v_4

v_3: v_4

v_4: nil

(a) Initial Flow.

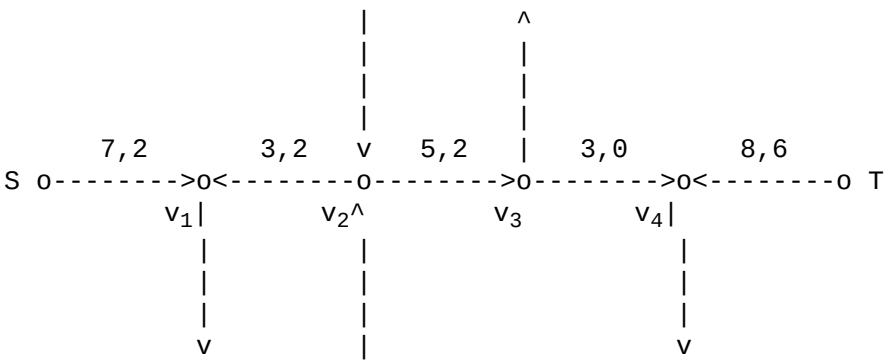
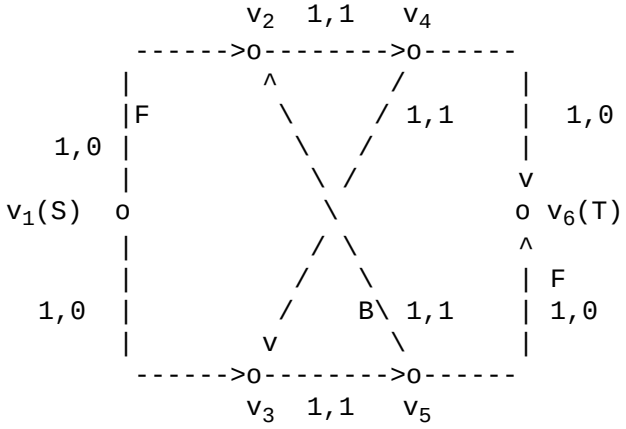


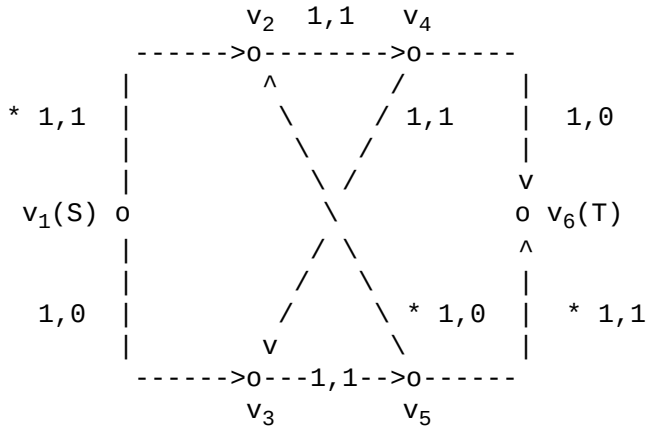
Figure 6-7. A Flow Augmenting Path.



F (Forward) B (Backward)

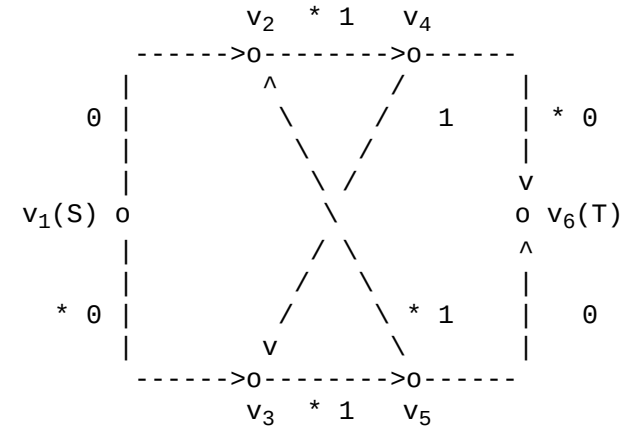
FAP: v_1-v_2-v_5-v_6 , Inc = 1

(a) General FAP for Network of Figure 6-6.



(b) Flow After Using FAP of (a).

Figure 6-8. Flow Augmentation Using FAP.



P_2: v_1-v_3-v_5-v_2-v_4-v_6, $f(P_2) = 1$

(c) Conservative Flow after Removal of Second Flow Path P_2.

Figure 6-5. Application of Get_Flow_Path.

SECTION 6-4

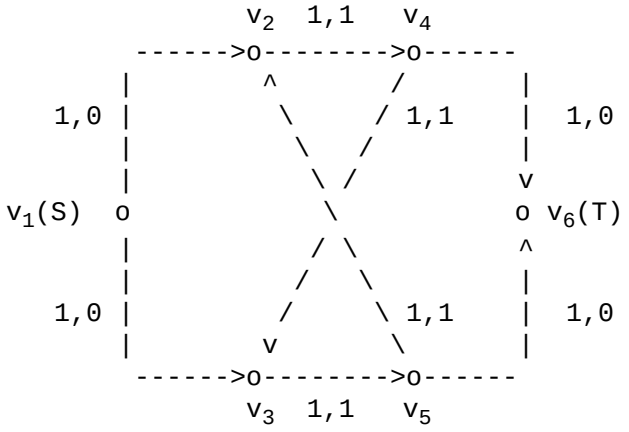
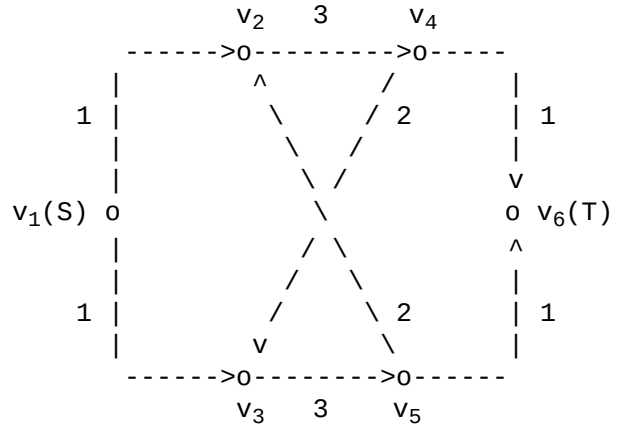


Figure 6-6. Network with No Simple Flow Augmenting Path.



v_1: v_2, v_3

v_2: v_4

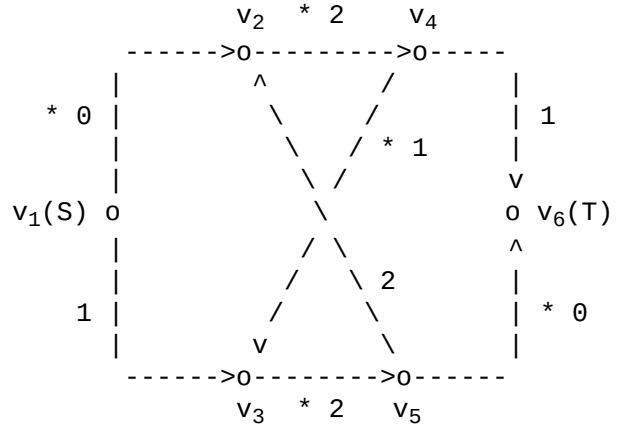
v_3: v_5

v_4: v_3, v_6

v_5: v_2, v_6

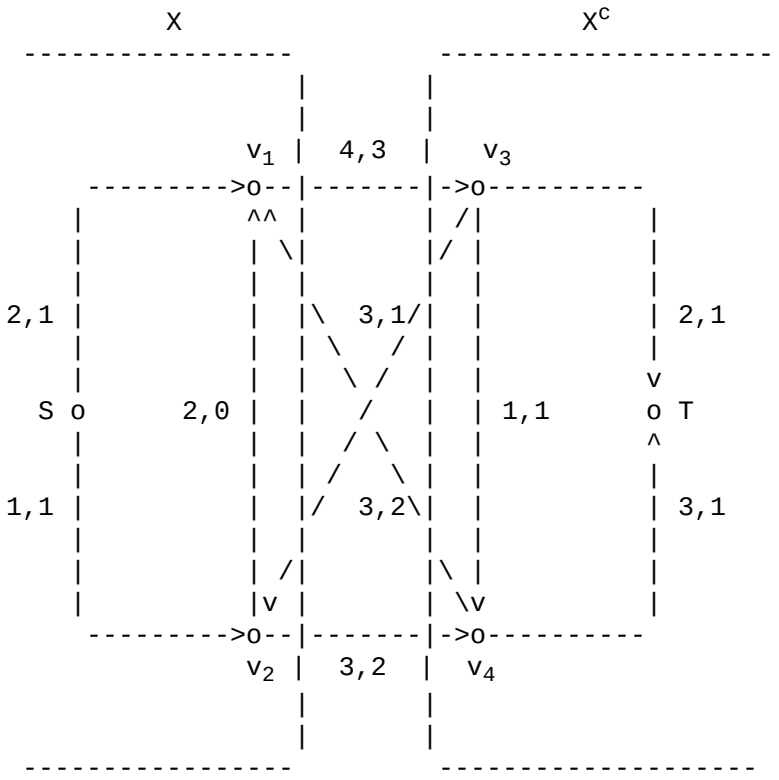
v_6: nil

(a) Conservative Flow with Edge Flows Shown.



P_1: v_1-v_2-v_4-v_3-v_5-v_6, $f(P_1) = 1$

(b) Conservative Flow after Removal of Flow Path P_1.



($X.X^c$) {(v_1,v_3), (v_2,v_4)}

(X^c,X) {(v_3,v_2), (v_4,v_1)}

flow (X, X^c) 5

flow (X^c,X) 3

Netflow (X,X^c) 2

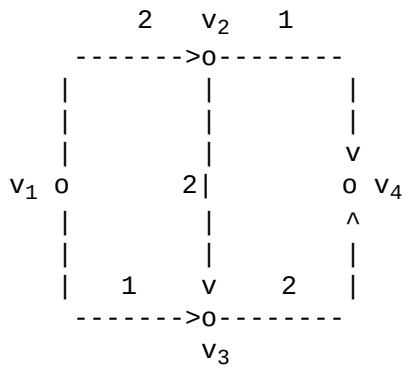
Conval (f) $f(v_3,T) + f(v_4,T) = 2$

minimum capacity
cut (S, V - {S}):{(S,v_1),(S,v_2)}

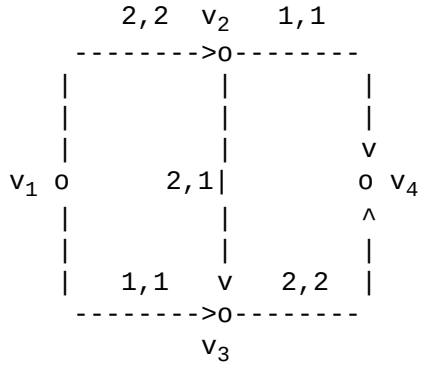
Cap(S, V - {S}) 3

Figure 6-4. (N,S,T,f) Cut Example.

SECTION 6-3



(a) Capacitated Network.



P_1: v_1-v_2-v_4 $f(P_1)$: 1

P_2: v_1-v_3-v_4 $f(P_2)$: 1

P_3: v_1-v_2-v_3-v_4 $f(P_3)$: 1

Edge x: Cap(x), Sum(f,x)

(b) Path Flow.

Figure 6-3. Path Flow Example.

Alternate Pair of Edge Disjoint Paths v_1 to v_5:

v_1-v_2-v_3-v_5

v_1-v_3-v_4-v_5

Maximum Cardinality Set of Vertex Disjoint Paths from v_1 to v_5:

Any path from v_1 to v_5

Figure 6-1. Diverse Routing Example.

SECTION 6-2

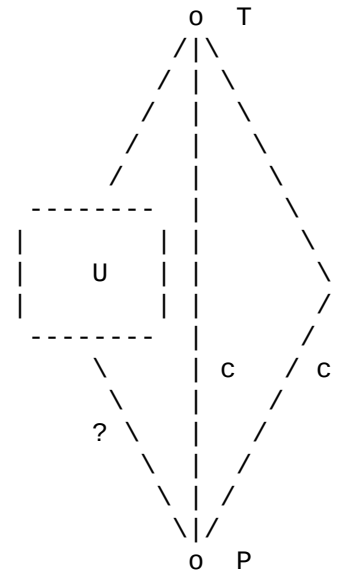


Figure 6-2. Messages Not Using U are Identical.

disconnected, due to some combination of edge failures. For fixed order, and for a given number of edges, can you determine how to allocate the edges among the vertices so the resulting network has minimum probability of failure? Consider, as an elementary example, the case where |E| = |V| + 1. Obtain a theoretical estimate of the probability of failure in the case of such graphs, and compare this with an estimate derived from simulation.

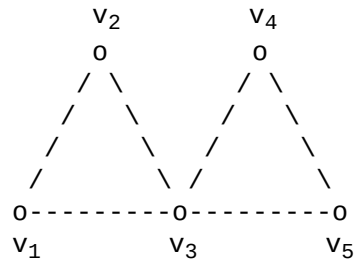
(18) Implement the path extraction algorithm for flow networks.

(19) Simulate the probabilistic routing algorithm on a hypercube.

(20) Write a program that simulates the operation of a "Byzantine" network, and that uses reliable routing methods to route messages through the network. Verify the routing can be done reliably even in the presence of simulated unreliable processors.

(21) Give an example of a maximal flow on a layered network which contains an ordinary flow augmenting path, despite the absence of any advancing flow augmenting paths.

SECTION 6-1



VC = 1

EC = 2

All Paths between v_1 and v_5:

v_1-v_2-v_3-v_4-v_5

v_1-v_3-v_5

v_1-v_3-v_4-v_5

v_1-v_2-v_3-v_5

Pair of Edge Disjoint Paths from v_1 to v_5:

v_1-v_2-v_3-v_4-v_5

v_1-v_3-v_5

minimum value of VC(G,u,v) over all pairs of nonadjacent vertices u and v.

(4) Prove the following special case of the Connectivity Lower Bound Theorem: If $\min(G) \geq |V|/2$, then EC(G) = min(G). Does an analogous result hold for VC(G)?

(5) Prove a graph G(V,E) of vertex connectivity k satisfies that $|E| \geq k|V|2$.

(6) Prove that VC(G) = EC(G) if G(V,E) is a cubic graph.

(7) Prove that the Petersen graph is the smallest three- connected three-regular graph.

(8) Prove that the complement of a disconnected graph is spanned by a complete bipartite graph.

(9) Prove that if a connected graph G(V,E) is r-regular and contains an articulation point, EC(G) is bounded by r/2.

(10) Does the Flow_Augmenting_Path function used for the Ford-Fulkerson maximum flow algorithm need predecessor pointers separate from the search stack if depth first search is used? What graphical object does depth first search naturally maintain? What graphical object does breadth first search naturally maintain? Can each search discipline maintain the other's natural graphical object conveniently?

(11) Adapt the Ford-Fulkerson algorithm to the case where both the vertices and the edges of the network have capacities. Hint: remodel the input network appropriately so that it has the format expected by the Ford-Fulkerson algorithm.

(12) Let G(V,E) be a digraph containing a unique vertex x of indegree 0 and a unique vertex y of outdegree 0. Try to design an algorithm to find a minimum cardinality cutset between x and y.

(13) Suppose that in a network N(V,E), there is not only a capacity associated with each edge, but also a cost, representing the unit cost of transmitting flow through that edge. Show how to model the minimum cost, maximum flow problem on such a network using linear programming.

(14) Implement a flow-based algorithm for finding the edge connectivity of a graph.

(15) Implement a flow-based algorithm for finding the vertex connectivity of a graph.

(16) Implement the probabilistic algorithm for finding the vertex connectivity of a graph, and compare its performance with the corresponding deterministic algorithm for a reasonable population of test graphs.

(17) Suppose every edge of a graph has the same fixed probability of failure. Use simulation to obtain an estimate that a given such graph is

destination Target(m) using the deterministic greedy method.

The performance of the algorithm is summarized in the following theorem.

THEOREM (VALIANT-BREBNER HYPERCUBE ROUTING) Let H(V,E) be an n-dimensional hypercube and let p be any partial permutation on V(H). Then, the probability that Valiant-Brebner random routing takes more than 8n steps is less than $O(0.74^n)$.

We refer to Valiant and Brebner (1981) for the nontrivial proof. One can show that using this method, the chance that more than i(log n) processors will simultaneously try to transmit a message through a given processor decreases exponentially with i. This makes serious bottlenecks rare. Nonetheless, a subset of processors can become embroiled in a deadlock, or the performance may happen to be poor. In each case, we can restart the algorithm with a statistically high chance of improvement on the next attempt.

_

CHAPTER 6: REFERENCES AND FURTHER READING

For a discussion of theoretical results on connectivity, see Harary (1971) and Behzad, et al. (1979). Boesch (1976) and (1982) describe network applications. Even (1979) gives a very thorough treatment both of flow algorithms, their applications to connectivity and planarity algorithms; it was the basis of part of our discussion of how to compute connectivity using flows and for Dinic's algorithm. See Ford and Fulkerson (1962) for the fundamentals of network flow theory and its variations. Lawler (1976) includes a good discussion of multicommodity flows. The enhancement of Dinic's algorithm presented uses a technique of Malhotra, Pramodh Kumar, and Maheshwari (1978). Edmonds and Karp (1972) gives an improved version of the Ford and Fulkerson maximum flow algorithm. For a detailed discussion of other flow maximization techniques, such as the conceptually more complicated but more efficient wave method, see Tarjan (1983) and Sleator and Tarjan (1983) where an O(|E| log |V|) method for finding maximal flows is described. The interesting Byzantine routing problem is described in Dolev (1982). Stone (1977) applies flow techniques to scheduling systems of processors. Becker, et al. (1982) give the probabilistic modification to connectivity calculation. The probabilistic routing algorithm for hypercube routing is analyzed in Valiant and Brebner (1981).

CHAPTER 6: EXERCISES

(1) Prove that if v is an articulation point of G(V,E), then v is not an articulation point of G'.

(2) Construct a graph G with VC(G) = 2, EC(G) = 3, and min(G) = 4. In general, try to construct a graph with VC(G) = n - 2, EC(G) = n - 1, and min(G) = n?

(3) Show the definitions of vertex and edge connectivity given in this chapter are equivalent to those given in Chapter 1. In particular, show that the minimum value of VC(G,u,v) over all pairs of vertices u and v equals the

```
(* Partial permutation routing on hypercube *)

var v, w:  Hypercube vertex
    m, m': message
    Dequeue, Receive: Boolean function

repeat

   if    Dequeue(Message_queue(v),m)
   then  Set w to next neighbor of v such that Address(w) differs
         from Address(v) in bit Leftmost(v, m)
         Send(m ,w)

   if    Receive (m')
   then  Enqueue (m', Message_queue(v))

until  partial permutation is completed

End_Procedure_Greedy_Routing
```

The problem with the greedy algorithm is that messages can be greatly delayed due to access contention at vertices along their routes, as illustrated by the following example. Suppose H is a ten-dimensional hypercube, with 1,024 vertex processors with addresses $(a_0 \ldots a_9)$. Suppose 2^5 messages have home addresses at $(0,0,0,0,0,a_5,a_6,a_7,a_8,a_9)$, and target addresses at $(a_5,a_6,a_7,a_8,a_9,0,0,0,0,0)$. Thus, each message is targeted for a processor whose address is the reverse of its home address. The greedy algorithm forces every message through the processor at (0,0,0,0,0,0,0,0,0,0) at step 5, causing a bottleneck. This leads to a delay of $O(|V(H)|^{.5})$, which is substantially greater than the theoretical logarithmic lower bound of n.

VALIANT-BREBNER RANDOM ROUTING ALGORITHM

A randomized version of the greedy algorithm attains the O(n) lower bound on transmission with probability arbitrarily close to 1 (as a function of n). The idea is to randomize the initial distribution of messages, which spreads the risk of the kind of source-to-target mapping that produces a bottleneck. Thereafter, we route the messages to their destinations using the previous deterministic greedy procedure. The technique is as follows.

(1) Each processor containing a message m to be routed, randomly generates an intermediate destination address r(m) for m.

(2) The greedy algorithm is used to route each message to its random intermediate target. Until a message m reaches r(m), it has priority access to network resources over messages that have already reached their randomized point of departure.

(3) Once a message m arrives at r(m), we route m to its original

A *hypercube* H(V,E) is an n-dimensional cube with 2^n vertices, each with an n-bit address, and containing an edge between any pair of vertices whose addresses differ in a single bit. Each vertex has degree n and the hypercube has diameter n, both logarithmic in the order |V(H)| of the hypercube. We can visualize processors with addresses differing in exactly the i-th bit as geometrically adjacent along the i^{th} dimension of the hypercube. A three-dimensional hypercube is shown in Figure 6-22, while Figure 6-23 shows part of a four-dimensional hypercube. There is a processor at each vertex, which communicates with the other processors by messages routed via the hypercube network. Each message contains the address of its target processor, and it takes unit time to transmit a message along an edge.

Figures 6-22 and 6-23 here

A basic routing problem for hypercubes is to concurrently transmit messages from one subset of its processors to another subset of its processors, so-called *partial permutation routing*. Each message contains a distinct target address, and the objective is to route the messages to their destinations as quickly as possible. Since only one message can be transmitted along an edge at a time, messages may have to be queued at processors lying on their route if there is contention for access to an edge.

We will describe two (distributed parallel) algorithms for implementing partial permutation: a greedy algorithm whose worst case performance is poor, and a randomized version of the greedy algorithm with optimal expected time performance.

GREEDY PARTIAL PERMUTATION ALGORITHM

Permutation routing has an obvious greedy algorithm. We denote a message by m, the address of the destination of m by Target(m), the address of a vertex processor v by Address(v), and the index of the leftmost bit of Target(m) that differs from the corresponding bit in Address(v) by Leftmost(v,m). We can permute the messages by simply moving each message m lying at a vertex v to that neighbor of v whose address matches Target(m) in one more bit than Address(v) does. Each such step brings each message one bit closer to its destination, and along a shortest path of length at most n, though resource contention by competing messages may delay the transmission.

The following high-level procedure describes the algorithm. Messages routed through a processor v are queued on a Message_queue(v) until transmitted to the next vertex on their route. A utility Dequeue(Message_queue(v),m) returns the head of the queue in m, or fails if the queue is empty; while Enqueue(m, Message_queue(v)) queues m on the message queue at v. We use a command Send(m,w) to indicate transmission of m to w. If w is busy, m is delayed at v on a transmission queue which is managed in an interrupt-driven fashion, and is otherwise transparent. A primitive Receive(m) returns a transmitted message in m, or fails if there is no incoming message. The messages reside initially at their home locations. The same procedure is executed at each processor. The data types are only suggestive.

Procedure Greedy_Routing (v)

PROBABILISTIC ALGORITHM FOR VC(G)

A probabilistic algorithm for a graphical invariant calculates the exact value of the invariant with a high degree of probability, though not with certitude. We can calculate VC(G) probabilistically by observing that VC(G) equals VC(G, u, v) for any pair of vertices u and v in G which are separated by a minimum cardinality vertex disconnecting set. The idea is as follows. Let $\{v_1,..,v_k\}$ be a random set of k vertices of G. If any of these vertices is not in some minimum vertex disconnecting set S, say vertex v_j, then VC(G, v_j, v) equals VC(G) for some vertex v which is separated from v_j by S. We can estimate the probability of having at least one such vertex v_j in a set of k random such vertices as follows. The probability that a given random vertex v lies in a particular minimum vertex disconnecting set S of cardinality VC(G) is VC(G)/|V|. Therefore, the probability that at least one of k random vertices lies outside S is at least $1 - (VC(G)/|V|)^k$. For example, if |V| is 100 and VC(G) is 10, then the probability that at least one of three randomly selected vertices lies outside S is at least 99.9%. The probabilistic procedure follows.

```
Function Random_VC (G,k)

(* Returns the value of VC(G) in Random_VC with probability
   1 - (VC(G)/|V(G)|)^k of being correct *)

var  G: Graph
     k: Integer constant
     i, j, R(k): 1..|V(G)|
     Random_VC: Integer function

Set Random_VC to |V(G)| - 1

if G is complete then  return

Select k distinct random vertices {R(1) ,..., R(k)} from V(G)

for i = 1 to k do

  for j = 1 to |V(G)| do

    if    R(i) and j are distinct and nonadjacent
    then  Set Random_VC to min { Random_VC, VC(G, R(i), j) }

End_function_Random_VC
```

6-8 PARTIAL PERMUTATION ROUTING ON A HYPERCUBE

There are a variety of routing problems of practical interest not covered by the classical theory we have described so far. For example, consider the problem of routing on the class of networks called hypercubes, a type of network used to interconnect parallel computers.

```
for i = 1 to |V(G)| do

   for j = i + 1 to |V(G)| do

      if    i and j not adjacent
      then  Set VC to min {VC, VC(G, i, j)}

   if  i > VC  then  return

End_Function_VC
```

THEOREM (CORRECTNESS OF VC) Let G(V,E) be a graph. Then, the function VC(G) correctly calculates the vertex connectivity of G.

The proof is as follows. If G is complete, the procedure is trivially correct. Otherwise, if G is not complete, the first calculation of VC(G, i, j) with i and j not adjacent already forces VC to at most |V| - 2. Therefore, the procedure must terminate with an explicit **return** at least by the time the outer loop is executed with i = |V| - 1. Consequently, upon termination,

$$VC < i.$$

Since VC(G) is the minimum of the nonadjacent pairwise connectivities, VC is always at least as large as VC(G); therefore

$$VC(G) + 1 \leq VC + 1 \leq i.$$

Since VC(G) is the cardinality of a minimum cardinality vertex disconnecting set S for G and since i exceeds VC on termination, at least one vertex j in 1..i cannot lie in S. Since j is not in S, j is separated from some other vertex v by the disconnecting set S. Refer to Figure 6-21. Necessarily, $VC(G, j, v) \leq |S|$. Since S is a disconnecting set of minimum cardinality, VC(G, j, v) must equal |S|, that is, VC(G). Therefore, VC will receive its correct value when the algorithm calculates VC(G, j, v). This completes the proof.

Figure 6-21 here

PROBLEMS OF EDGE CONNECTIVITY

We can calculate EC(G) by finding a maximum flow on a network G' obtained from G by replacing each undirected edge {u,v} in G by a pair of directed edges (u,v) and (v,u) in G' and assigning a unit capacity to each edge of G'. The pairwise edge connectivity, pairwise edge disjoint path, and pairwise edge disconnecting set problems can then be solved using flow methods. The flow bearing paths of a flow on G' correspond directly to a set of edge disjoint paths in G. The edges of a minimum cut between a pair of vertices in G' determine a minimum cardinality edge disconnecting set in G between those vertices. We can calculate EC(G) by selecting an arbitrary vertex v in G and calculating the minimum of VC(G, u, v) over all distinct vertices u in G, and the given v (why?).

(1) A maximum cardinality set of vertex disjoint paths between a given pair of vertices u and v in a graph G corresponds to a set of flow paths realizing a maximum flow between u'' and v' in G';

(2) The pairwise connectivity VC(G, u, v) between a pair of vertices u and v in G equals the maximum flow value between the vertices u'' and v' in G'; and

(3) A minimum size vertex disconnecting sets between a pair of vertices u and v in G corresponds to a minimum capacity u'' - v' cut in G'.

Statement (3) follows by observing that a finite capacity cut in G' must contain only edges of the form (x',x''). Therefore, the vertices x in G that correspond to the edges (x',x'') of a cut in G' comprise a vertex disconnecting set in G of cardinality equal to the capacity of the cut.

The correspondences indicated by these statements allow us to calculate connectivities on G using flow algorithms on G', as intended. We refer to Figure 6-20 for an illustration. VC(G, v_1, v_4) equals 2, so there are a maximum of two vertex disjoint paths between v_1 and v_4. The vertices {v_2, v_3} constitute a minimum size vertex disconnecting set. As expected, the maximum value flow from v_1'' to v_4' in G' is also 2. The paths in G' realizing the maximum flow are v_1''-v_2'-v_2''-v_4' and v_1''-v_3'-v_3''-v_4'. If we contract edges on the flow paths of the form (x',x''), we obtain a maximum set of vertex disjoint paths in G, namely: v_1-v_2-v_4 and v_1-v_3-v_4. A minimum capacity cut between v_1'' and v_4' in G' is given by ({v_1'', v_3', v_2'}, {v_1', v_2'', v_3'', v_4', v_4''}). The edges of the cut are (v_2',v_2'') and (v_3',v_3''). Therefore, {v_2,v_3} is the corresponding minimum size vertex disconnecting set.

VERTEX CONNECTIVITY OF A GRAPH

The usual formula for the vertex connectivity of a graph G(V,E) in terms of its pairwise vertex connectivities is

$$VC(G) = \min_{(u,v) \text{ not in } E(G)} \{ VC(G, u, v) \} .$$

The following procedure improves on this slightly. For convenience, we assume that if G is not a complete graph; then its vertices are ordered so that v_1 is not adjacent to some vertex v. We invoke functions VC(G, u, v) to calculate the pairwise vertex connectivities.

```
Function VC(G)

(* Returns in VC the vertex connectivity of G *)

var  G: Graph
     i, j: 1..|V(G)|
     VC: Integer function

Set VC to |V(G)| - 1
if G is complete then return
```

6-7 CONNECTIVITY ALGORITHMS

We can compute the vertex and edge connectivity of a graph by modelling connectivity as a network flow problem, and then applying maximum flow techniques. There are a variety of connectivity problems to consider. The *pairwise connectivity* problem asks for the vertex (or edge) connectivity between a given pair of vertices in a graph. The *pairwise disjoint paths* problem requires finding a maximum cardinality set of vertex (or edge) disjoint paths between a given pair of vertices. The *pairwise disconnecting sets* problem seeks a minimum size vertex (or edge) disconnecting set between a given pair of vertices; while the *graph connectivity* problem seeks the vertex (or edge) connectivity of a graph. We begin by considering the vertex variations of these problems.

PROBLEMS OF VERTEX CONNECTIVITY

Let G(V,E) be a graph. We shall construct a flow network called the *associated flow network* G' for G as follows:

(1) For each vertex v in V(G), create a pair of vertices v' and v'' and an edge (v', v'') in V(G');

(2) For each edge (u, v) in E(G), create a pair of edges (u'',v') and (v'',u') in E(G');

(3) Assign a unit capacity to each edge in E(G') of the form (v',v''), and some large capacity M (that plays the role of plus infinity) to each edge created in (2).

This construction allows us to translate problems about paths in G into problems about flows in G'. The construction is illustrated in Figure 6-20.

Figure 6-20 here

The associated flow network satisfies the following important property.

Claim: Let u'' and v' be vertices in G' and let F be a set of paths from u'' to v', each of unit capacity, which realize a flow from u'' to v'. Then, the paths in F are vertex disjoint, except for their endpoints u'' and v'.

The claim follows because any internal vertex on any of the paths in F has either a unique incoming edge of unit capacity or a unique outgoing edge of unit capacity. Therefore, at most one unit capacity path can pass through any internal vertex of the paths in F.

We can map a set of u'' to v' flow paths in G' to a set of u to v paths in G by the simple expedient of contracting flow edges of the form (x',x'') to a vertex x and replacing flow edges of the form (x'',y') by an edge (x,y).

Conversely, we can map a u to v path in G to a u'' to v' flow path in G' as indicated by the following example: The path u-a-b-v in G becomes u''-a'-a''-b'-b''-v' in G'.

It follows readily from these observations that

assignment is static. That is, each module remains with its assigned processor for the duration of the problem. In order to make an intelligent choice of module-to-processor allocation, we also assume that we know the frequency with which the different modules reference each other. If a pair of modules reside on the same processor, their intermodule communication cost is assumed to be zero. If a pair of modules M_i and M_j reside on different processors, the communication cost between them is denoted by M_{ij}. The intermodule communication costs are specified by a *communication cost graph* and the module-processor execution costs by a table. The cost of a schedule (that is, a module-to-processor assignment for every module) is the sum of the intermodule communication costs M_{ij} and the module execution costs T_{ij}. The objective is to find a schedule of minimimum cost.

Figure 6-16 here

We model the assignment of modules to processors by a *module assignment graph* defined as follows.

(1) Add two vertices S_1 and S_2 to the communication cost graph, corresponding to the pair of processors P_1 and P_2.

(2) Add edges (M_i,S_2) of weight M_{i1} corresponding to the possible assignment of module M_i to P_1.

(3) Add edges (M_i,S_1) of weight M_{i2}, corresponding to the possible the assignment of module M_i to P_2.

Figure 6-17 shows the module assignment graph for the configuration shown in Figure 6-16. With this definition of the module assignment graph, there is a 1-1 correspondence between processor-to-module assignments and (S_1,S_2) cuts. Namely, we assign the module whose vertices lie in the S_1 part of the cut to P_1, and we assign the modules in the S_2 part of the cut are to P_2. The weight of the resulting cut is equal to the cost of the assignment.

(1) The modules M_i and M_j will incur the communication cost M_{ij} if and only M_i and M_j are on opposite sides of the cut; so the edge (M_i,M_j) is in the cut, and

(2) Module M_i incurs the execution cost T_{i1} if and only if (M_i,S_2) is an edge of the cut (similarly for T_{i2}).

Thus, the cost of the cut equals the cost of the module-to-processor assignment. It follows from the theory of network flow that the optimal module-to-processor assignment is obtained by finding the maximum value flow from S_1 to S_2. Figure 6-18 shows an optimal solution for the example of Figure 6-17. Figure 6-19 illustrates a suboptimal assignment.

Figure 6-17 here

Figures 6-18 and 6-19 here

not in Layered(k). But, since (u,w) is augmenting in Layered(k + 1) and is not in Layered(k), then (u,w) must also be augmenting at stage k. Since u is in Layered(k) and (u,w) is augmenting but not in Layered(k), Level(w,k) $\geq$ Level(T, k). By the same argument as in Case 1, Level(w, k + 1) $\geq$ Level(w, k), which implies Level(w, k + 1) $\geq$ Level(T, k). Consequently, Level(T, k + 1) $\geq$ Level(T, k) + 1, was to be shown. This completes the proof of the theorem.

The performance of the algorithm can be summarized as follows.

THEOREM (DINIC PERFORMANCE) Let N(V,E) be a capacitated network and let S and T be a pair of distinct vertices in N. Then, Dinic's algorithm, enhanced by the maximal flow technique of Malhotra, Pramodh Kumar, and Maheshwari, takes $O(|V|^3)$ time.

The proof of this theorem is as follows. By the previous theorem, the algorithm constructs at most |V(N)| layered networks. Thus, the performance estimate depends on the time taken by the maximal flow algorithm on the layered network. The key point is that at each flow routing phase in Maximal we use at most one edge without saturating it, so that the number of edges used is at most $O(|E(N)| + |V(N)|^2)$. Thus, the overall complexity of Dinic is $O(|V(N)|^3)$. This completes the proof of the theorem.

THEOREM (CORRECTNESS OF DINIC ALGORITHM) Let N(V,E) be a capacitated network and let S and T be a pair of distinct vertices in N. Then, the flow constructed by Dinic's algorithm is a maximum flow.

The proof of this theorem is as follows. The bound of the Monotonicity theorem ensures that the algorithm terminates. Since the termination condition is that the layered search network ALN does not reach the sink T, just as in the case of the Ford and Fulkerson algorithm, (V(ALN), V(N) - V(ALN)) determines a cut whose forward edges are saturated and whose backward edge flows are all zero. It follows from the earlier theorems on net and maximum flow that the flow must be maximum at this point. This completes the proof of the theorem.

6-6 FLOW MODELS: MULTIPROCESSOR SCHEDULING

We can optimally schedule the execution of programs on a pair of processors using flow techniques (Stone [1977]). Let us assume that the system to be scheduled consists of separate modules which may be either executable or data modules. The execution time of an executable module depends on which processor it is scheduled to execute on; while, on the other hand, if a data module is accessed by an execution module on a different processor, then a communication cost is incurred. The objective is to assign the modules to the processors in such a way as to minimize the combined execution and communication costs.

Denote the processors by P_1 and P_2 and the modules by M_1 ,..., M_k. The execution time required by an executable module M_i executing on processor P_j is denoted by T_{ij}. We assume that the modules scheduled to execute on a given processor execute sequentially. We assume further that the scheduling

monotonically increasing in i.

The proof of this theorem is as follows. Let Len(Layered(k + 1)) be denoted by n and let P be an advancing flow augmenting path from S to T in Layered(k + 1). Denote the successive vertices of P by w_i, i = 0..n. We will distinguish two cases according to whether or not every vertex in P lies in Layered(k).

Case 1: Every vertex in P lies in Layered(k)

We first establish the following claim:

Claim: Level(w_i,k) $\leq$ i, for i = 0 ,..., n.

The proof of the claim is by induction on i. The claim is trivial for i = 0. Suppose the claim is true for i = j. We shall prove it for i = j + 1. That is, we shall show that Level(w_{j+1},k) $\leq$ j + 1. Otherwise, w_{j+1} would lie in level L(x, k) where x $\geq$ j + 2. But, by induction, Level(w_j, k) $\leq$ j. Therefore, the edge (w_j,w_{j+1}) would be from L(y,k), y $\leq$ j to L(x,k), x $\geq$ j + 2. Therefore, (w_j,w_{j+1}) would not be in Layered(k), since by construction all the edges in a layered network are between adjacent levels. Therefore, (w_j,w_{j+1}) would not have been useful at stage k. Since (w_j,w_{j+1}) is an edge of Layered(k+1) by definition, it must be useful at stage k + 1. But, since it was not changed in stage k, it must also have been useful at stage k. Since w_j and w_{j+1} are both in Layered(k), (w_j,w_{j+1}) must lie in Layered(k), which is a contradiction. Thus, w_{j+1} cannot belong to L(x,k), x $\geq$ j + 2. Therefore, it must belong to L(r, k), r $\leq$ j + 1, as was to be shown for the claim.

Returning to the proof in case 1, we observe that it follows from the claim that Level(T, k + 1) $\geq$ Level(T, k), and so Len(Layered(k + 1)) $\geq$ Len(Layered(k)). Furthermore, the inequality must be strict, for otherwise we can prove that P is an advancing flow augmenting path in Layered(k) after the maximal flow for Layered(k) has been established, which is a contradiction. To prove this, observe that w_i must be in L(i, k). Otherwise, we can use the same argument as used in the proof of the claim to identify an edge (w_i,w_{i+1}) that skips a level in Layered(k) which would again lead to a contradiction, forcing us to include (w_i,w_{i+1}) in Layered(k) and to conclude that w_i and w_{i+1} must be in successive levels of Layered(k), showing w_i would lie in L(i,k). A similar argument shows that the edges (w_i,w_{i+1}) of P must also lie in Layered(k). This proves that P is a flow augmenting path in Layered(k). But, since P is augmenting at the beginning of the phase that constructs Layered(k + 1), P must also be augmenting at the end of the maximal flow phase that constructs Layered(k), contradicting the correctness of the maximal flow algorithm for Layered(k). It follows that Len(Layered(k)) < Len(Layered(k + 1)), as was to be shown.

Case 2: Some vertices in P are not in Layered(k)

Let w be the first vertex in P which is not in Layered(k) and let u be predecessor of w on P. The edge (u,w) cannot be in Layered(k), since w is

```
var  ALN: Layered Network
     i: 1..|V(N)| - 1
     x: (+1, -1)
     u,w: 1..|V|
     uw-edge-ptr: ALN-Edge pointer
     f: Nonnegative Integer

for  u in L(i)  do

   while  Flow(u) > 0  do

     Next-level(ALN, i, u, x, w, uw-edge-ptr)

     if    Useful(ALN, x, u, w, uw-edge-ptr)

     then  Set f to min {Flow(u), useful capacity of uw-edge-ptr}
           Change the flow on uw-edge-ptr (using f) and Thruputs
           Set Flow(u) to Flow(u) - f
           Set Flow(w) to Flow(w) + f

End_Procedure_Route
```

Add(TRANS,N) updates the flow on N(V,E) using TRANS to locate the edges whose flows have to be altered and to indicate the magnitude and direction of the adjustments to be made. The value in the amount field on an edge x listed in TRANS is added to the flow field of the corresponding edge in N, if the direction field of x in TRANS is forward, and is subtracted if the direction equals backward.

illustrative examples are given in Figures 6-13 through 6-15.

Figures 6-13, 6-14, and 6-15 here

CORRECTNESS AND PERFORMANCE

The correctness of Dinic's algorithm follows easily from a proof of its termination, just as in the case of the Ford and Fulkerson maximum flow algorithm. Therefore, we shall consider the performance of the algorithm first and its correctness afterwards. First, we will introduce some notation. Let us denote the j^{th} augmenting layered network constructed by Dinic's algorithm from a given network N by Layered(j) and the i^{th} level of Layered(j) by L(i,j), or by L(i) if j is clear from the context. If a vertex v lies in L(i,j), then Level(v,j) equals i, the index of the level in Layered(j) that v lies in. If the parameter j is clear from the context, we use the shorthand notation Level(v). We denote the index of the level the sink T lies in, Level(T,j), by Len(Layered(j)). The performance of the algorithm relies on the fact that Len(Layered(j)) is monotonically increasing in j.

THEOREM (MONOTONICITY OF DINIC'S LAYERED NETWORKS) Let N(V,E) be a capacitated network and let S and T be a pair of distinct vertices in N. Let (N,S,T,0) denote a trivial (zero-valued) conservative flow. If we denote the i^{th} layered search network constructed by Dinic's algorithm by Layered(i), starting from the trivial flow (N,S,T,0), Len(Layered(i)) is strictly

version of Useful to test the utility of the edge for augmentation. The adjustments made to the edge flows (by Route) are positive or negative depending on whether the edges are forward or backward in ALN.

```
Procedure Maximal (ALN,S,T,TRANS)

(* Construct maximal flow in the layered network ALN from S to T,
   which is returned in TRANS *)

var  ALN: Layered Network
     TRANS: Translation Table
     S,T,v: 1..|V|
     i, Lev: 1..|V(N)| - 1
     Save_Thruput: Integer
     ZTQ: Queue pointer
     Disconnected: Boolean

Reset (TRANS)
for v in V(ALN) do Set Flow(v(ALN)) to 0
Set Disconnected to False

while  not Disconnected  and*   Select(ALN,v)  do

   (* Move Thruput(v) units of flow through v *)

   Set Save_Thruput to Thruput(v)
   Set Flow(v(ALN)) to Save_Thruput
   for i = Level(v) to Len(ALN) - 1 do Route(ALN, i, +1)
   Set Flow(v(ALN)) to Save_Thruput
   for i = Level(v) to 1 do Route(ALN, i, -1)

   (* Remove vertices of zero Thruput *)

   Create (ZTQ)

   for Lev = Level(T) - 1  to  1  do

      for v in L(Lev) do if Thruput(v) = 0  then  Enqueue(ZTQ,v)

      while  Dequeue(ZTQ,v)  do  Store edges at v in TRANS
                                 Remove(v, ALN)

      if Empty(L(Lev))  then  Set Disconnected to True

for v in V(ALN) do Store edges at v in TRANS

End_Procedure_Maximal


Procedure Route(ALN, i, x)

(* Route the flow accumulated at level i of ALN to level i + 1
   or level i - 1 as determined by i and x *)
```

```
    if   Level(v) > Level(Head(Q))
                and*  Useful(N, Head(Q), v, v-edge-ptr)

    then if Level(v) = M then Add (v, Level(Head(Q)) + 1)
                              Enqueue (Q, v)
         Put (v-edge-ptr, ALN)

until  Head (Q) = T  or*  Empty (Dequeue (Q))

(* Removal of ALN vertices not on flow augmenting paths in ALN
   from S to T *)

if    Head(Q) = T

then  Set Layered_Network  to True

      Set ZTQ to L(Level(T)) - {T}

      for  i = Level(T) - 1 to 0  do

         while  Dequeue(ZTQ,v)  do  Remove(v, ALN)

         for v in L(i) do if Thruput(v) = 0 then Enqueue(ZTQ,v)

else  Set Layered_Network  to False

End_Function_Layered_Network
```

CONSTRUCTION OF THE MAXIMAL FLOW ON ALN

Maximal(ALN,S,T,TRANS) constructs a maximal flow on the layered network ALN, that is, a conservative flow (ALN,S,T,f') for which there are no advancing flow augmenting paths from S to T. The idea is to find a vertex v in ALN of minimum thruput capacity Thruput(v) and then route Thruput(v) units of flow from S to T through v. We first push Thruput(v) units of flow upwards from v towards T via useful edges and then pull Thruput(v) units through v from the direction of S. The flow can never be blocked at intermediate vertices regardless of how the routing is done because Thruput(v) is the minimum thruput capacity of any vertex currently in ALN, whence a blockage is impossible. If the routed flow reduces the thruput capacities of any vertices to zero, we remove them in the same manner as was done in Layered-Network. The procedure stops when ALN becomes disconnected. Until then, we repeatedly select successive vertices of minimum thruput and repeat the routing process.

Maximal uses a function Select(ALN,v) to return a vertex v in ALN (other than S or T) of minimum thruput capacity. Select fails if the minimum thruput capacity is zero, and succeeds otherwise. The procedure Reset reinitializes the translation table TRANS. Another procedure Route is used to move flow accumulated at one level of ALN to an adjacent level of ALN. Route(ALN,i,x) routes the accumulated flow at level i to level i + x, where x is restricted to be plus or minus one. Route calls Next-level(ALN, i + x, u, w, uw-edge-ptr) to return the next neighbor w of u on level i + x, and a new

this case.

Layered_Network calls several subordinate functions and procedures. We use a (lexically overloaded) procedure Reset to perform the obvious resetting functions for N and ALN. We use a variant of the familiar function Next(N, Head(Q), v, v-edge-ptr) to return not only the index of the next neighbor of Head(Q) in N(V,E), but also a pointer v-edge-ptr to the corresponding entry on the adjacency list of Head(Q). The procedure Useful(N, Head(Q), v, v-edge-ptr) succeeds if the referenced edge is useful from Head(Q) to v, and fails otherwise. Add (v, Level(Head(Q)) + 1) adds v to Level(Head(Q)) + 1 of ALN. Put(v-edge-ptr, ALN) inserts the edge referenced by v-edge-ptr in the appropriate level of ALN, assigns its capacity and flow correctly, and updates any affected fields in ALN such as Input, Output, and Thruput.

After the initial breadth first draft of ALN is constructed, we then proceed to remove vertices not lying on flow augmenting S to T paths in this draft of ALN. These vertices correspond to vertices lying at the same level as T, as well as vertices of zero Thruput at lower levels. As we remove these vertices, we may introduce additional such vertices, since the removal process may affect the Thruput of the neighbors of removed vertices. We start the process at Level(T) and proceed downward through the lower levels. The zero Thruput vertices at each level are stored on a queue ZTQ. An enhanced queue operation Dequeue(ZTQ,v) returns the head of the queue in v, if it exists, and fails otherwise. The procedure Remove(ALN, v) removes v from ALN. This is a complex operation which includes deleting edges to the neighbors of v (which are necessarily on Level(v) - 1), updating affected Inputs, Thruputs, and the like.

```
Function Layered_Network (N,S,T,ALN)

(* Constructs an augmenting layered network ALN for (N,S,T,f),
   or fails *)

var  N: Flow Network
     ALN: Augmenting Layered Network
     S,T,v: 1..|V(N)|
     v-edge-ptr: N-Edge pointer
     M, Inc: Nonnegative Integer
     Dir: (Forward, Backward)
     Next, Empty, Dequeue, Layered_Network: Boolean function
     Head: Integer function
     Q, ZTQ: Queue pointer

Reset (N); Reset (ALN, M)
Create (Q); Enqueue (Q,S)
Set M to |V(G)|

(* Breadth First Advance to T *)

repeat

  while  Next(N,Head(Q),v,v-edge-ptr)
```

field in Shared-ALN-edge serves a similar role to the identically named field in the representation for N(V,E).

The breadth first search in procedure Layered-Network uses a queue with entries of type

```
type  Queue = record
                Identifier: 1..|V(N)|
                Queue-successor: Queue pointer
              end
```

The purpose of the table TRANS is to store the maximal flow values determined by Maximal for ALN in a form that allows them to be readily applied to update the flow in N(V,E). The type definition is as follows.

```
type  Translation Table =
             record
               T(|E(N)|): Table entry
             end

      Table entry =
             record
               E(1,2): 1..|V|
               N-edge-entry: N-Shared-edge pointer
               Amount: Integer
               Direction: (Backward,Forward,Undefined)
             end
```

The fields have the following meanings. The array T has |E(N)| entries since there are |E(N)| edges in N(V,E). Each edge in N is represented by the E field of one of the table entries. The pointer N-edge-entry allows direct access to the network representation of the edge. Recall that both N and ALN contain indices Edge-index which allow direct access into the array T. Layered-Network copies these indices from N to ALN when it constructs ALN. Maximal then uses these indices to store the maximal flow values on ALN directly in TRANS. Add then uses TRANS to directly update the flows on N(V,E). The fields Amount and Direction are determined by Maximal and define the magnitude and sign of the flow changes to be made by Add to N. Prior to each operation of Maximal, the values of Amount and Direction are considered Undefined, and remain so if the edge is not involved in a flow change.

CONSTRUCTION OF THE AUGMENTING LAYERED NETWORK

The algorithm for constructing an augmenting layered subnetwork of N(V,E) has two phases. First, it performs a modified breadth first search which uses only useful edges to extend the search network it constructs. Then, it removes any excess vertices and edges which do not lie on flow augmenting paths from S to T. The capacities and flows of the edges in ALN are assigned as follows. Let (c,f) be the (cap,flow) for an edge x in N which lies in ALN and let (c', f') denote the (cap,flow) x is to be assigned in ALN. If x is a forward edge of ALN, then we set c' to c - f and f' to 0 (corresponding to the useful capacity of x). If x is a backward edge of ALN, we set both c' and f' to f, which is again appropriate considering the useful capacity of x in

indicate a dummy level. Subsequently, when we actually have occasion to scan the level lists, we can check the Level fields of the vertices at that point, and delete a vertex from the level list if a dummy value in the entry indicates that that is appropriate. The Layered Network type definition follows.

```
type  Layered Network =
            record
              ALNH(|V(N)|): ALN-Vertex
              L(0..|V(N)| - 1): Level-Vertex
            end

      ALN-vertex =
            record
              Level: 0..|V(N)|
              Input, Output, Thruput, Flow: Nonnegative Integer
              Positional-Pointer, Successor: ALN-edge pointer
            end

      ALN-edge =
             record
               ALN-Shared-edge: Shared-ALN-edge pointer
               Edge-successor, Edge-predecessor: ALN-edge pointer
             end

      Shared-ALN-edge =
             record
               E(1,2): 0..|V|
               cap, flow: Nonnegative Integer
               Edge-index: 1..|E(N)|
               Back-ref(2):  ALN-edge pointer
             end

      Level-vertex =
             record
               Positional-Pointer, Successor: Level-entry pointer
             end

      Level-entry =
             record
               Identifier: 1..|V(N)|
               Successor: Level-entry pointer
             end
```

Once again, most of the fields are either familiar or self-explanatory. The fields for ALN-vertex are defined as follows. Level is on 0..|V|, with the value |V| used to indicate a dummy level. The *input capacity* of a vertex v (denoted Input(v)) in level i of ALN is the total of the useful capacities on edges useful from level i - 1 of ALN to v. The *output capacity* of a vertex v is the total of the useful capacities on edges useful from v to vertices on level i + 1, denoted Output(v). The *thruput capacity* of v equals min(Input(v), Output(v)) and is denoted by Thruput(v). The pair of pointers in Back-ref(2) in Shared-ALN-edge point to the adjacency list entries for the endpoints of the represented edge and facilitate deletion. The Edge-index

DATA STRUCTURES

We will describe the data structures first. Dinic uses three non-elementary types:

N: Flow Network
ALN: Augmenting Layered Network
TRANS: Translation Table

We will represent N and ALN as linear arrays. TRANS is an array which lets us directly access the edge representatives in N.

The type definition of N (Flow Network) is just a slightly enhanced capacitated network and follows. Its edges are represented in a shared manner on the adjacency lists of its endpoints.

```
type  Flow Network =
                  record
                    NH(|V(N)|): N-Vertex
                  end

      N-Vertex = record
                    Positional-Pointer, Successor: N-Edge pointer
                  end

      N-Edge  =  record
                    Shared-rep: N-Shared-edge pointer
                    Edge-successor,
                    Edge-predecessor: N-Edge pointer
                  end

      N-Shared-edge =
                  record
                    E(1,2): 0..|V|
                    cap, flow: Nonnegative Integer
                    Edge-index: 1..|E(N)|
                  end
```

Most of the fields are either familiar or self-explanatory. The N-Shared-edge record contains a field Edge-index which gives the index in the array TRANS of the edge given by the 1 x 2 array E(1,2). Its use will be explained when we define the type of TRANS.

The type definition of ALN is designed with several objectives in mind. It must facilitate fast access to the vertices of the augmenting layered network on a given level, which we do by packaging within the representation of ALN a linear array L of the vertices by level. Since both the procedures Layered-Network and Maximal delete vertices and edges, we design the representation to facilitate this by including pointers in each shared edge representative in ALN that point back to the adjacency list entries for its endpoints, and we use doubly-linked adjacency lists for the same reason. *Lazy deletion* can be used when deleting vertices from the level lists. That is, at the point at which a vertex is to be deleted, we set its Level field to

will say an edge y in E(N) is *useful from u to v* if either y equals (u,v) and cap(u,v) - f(u,v) > 0 or y equals (v,u) and f(v,u) > 0. The *useful capacity* of an edge y useful from u to v is cap(u,v) - f(u,v) if y equals (u,v) and f(v,u) if y equals (v,u).

An *augmenting layered network* ALN with respect to (N,S,T,f) is then defined as follows. The set of vertices V(ALN) is a subset of V(N) which is partitioned into disjoint parts L(i), i = 0 ,..., t where L(0) equals {S} and L(t) equals {T} and where the partition has the property that every vertex v in the partition lies on a flow augmenting path in N from S to T whose successive vertices come from successive parts L(i) and L(i + 1) of the partition. If u and v are vertices of ALN from parts L(i) and L(i + 1) respectively, the edge (u,v) or (v,u) is in E(ALN) if and only if (u,v) or (v,u) is in E(N) and is useful from u to v.

We refer to Figure 6-13 for an example. The parts L(i) are called the *layers* or *levels* of ALN. The index t of the level that vertex T lies in is called the *length* of the layered network and is denoted by Len(ALN). We define an *advancing flow augmenting path* from S to T on a layered network as a flow augmenting path whose vertices are from successive levels of ALN. We say a flow (ALN,S,T,f) on a layered network ALN is *maximal* if there is no advancing flow augmenting path in ALN from S to T.

The key to Dinic's algorithm lies in the construction of maximal flows on successive augmenting layered subnetworks of N(V,E). The procedure Dinic gives a high level view of this algorithm. We naturally assume that the original flow on N is identically zero, and that at each iteration the function Layered_Network(N,S,T,ALN) finds an augmenting layered network ALN with respect to the current conservative flow (N,S,T,f). Maximal(ALN,S,T,TRANS) constructs a maximal flow on ALN which it outputs as a table TRANS, which the procedure Add(TRANS,N) uses to augment the flow on N. The process is repeated as long as Layered_Network succeeds in finding an augmenting layered network with respect to the updated flows. Initial (N,TRANS) merely initializes TRANS using N(V,E). See Figure 6-12 for a data flow diagram of the algorithm.

Figure 6-12 here

```
Procedure Dinic(N,S,T)

var  N: Flow Network
     ALN: Augmenting Layered Network
     TRANS: Translation Table
     S,T: 1..|V(N)|
     Layered_Network: Boolean function

Initial (N,TRANS)

while Layered_Network (N,S,T,ALN)  do  Maximal (ALN,S,T,TRANS)
                                       Add (TRANS,N)

End_Procedure_Dinic
```

THEOREM (CORRECTNESS OF FORD AND FULKERSON ALGORITHM) Let N(V,E) be a capacitated network with distinct vertices S and T and let (N,S,T,f) be the conservative flow determined by the Ford and Fulkerson algorithm. Then, the flow value Conval(f) has a maximum possible value among all conservative flows on N.

The proof of this theorem is as follows. We first show that the algorithm terminates, and then that on termination the flow must have maximum value. To establish termination, observe that each invocation of Flow_Augmenting_Path increases the flow value by an integral amount Path-Val(T). Since maxflow(N) is bounded above by mincap(N), the algorithm must terminate within mincap(N) calls to Flow_Augmenting_Path.

We show next that the value of the final flow equals mincap(N) and so must equal maxflow(N) by the Maximum Flow / Minimum Capacity Cut Theorem. Let us denote the blocked search tree by BT. Then, (V(BT), V(N) - V(BT)) is an S-T cut. Every edge of the cut is saturated since otherwise some edge would be returned by Try_to_Augment. Similarly, every edge of (V(N) - V(BT), V(BT)) must have zero flow since otherwise some edge would be returned by Try_to_Augment. It follows from the Net-Flow Theorem that

$$\begin{aligned} \text{Conval}(f) &= \text{flow}(X, X^C) - \text{flow}(X^C, X) \\ &= \text{Cap}(X, X^C) - 0. \end{aligned}$$

Thus, the flow value equals the capacity of an S-T cut. Since maxflow(N,S,T) $\leq$ mincap(N,S,T), (N,S,T,f) must be a maximum flow and the cut must be a minimum capacity S-T cut. This completes the proof of the theorem.

he following theorem follows directly from the previous proof and summarizes the relation between maximum flows and minimum cuts.

THEOREM (MAXIMUM FLOW EQUALS MINIMUM CUT) Let N(V,E) be a capacitated network with distinct vertices S and T. Then,

$$\text{maxflow}(N,S,T) = \text{mincap}(N,S,T).$$

6-5 MAXIMUM FLOW ALGORITHM: DINIC

The maximum flow algorithm of Dinic finds a maximum flow in a capacitated network N(V,E) by repeatedly finding maximal flows in a layered subnetwork of N. The algorithm takes |V(N)| search phases in contrast to the O(|V||E|) phases required by the Edmonds-Karp version of the Ford and Fulkerson algorithm. We will present an enhanced version of Dinic's algorithm which uses an efficient procedure for finding maximal flows in layered networks due to Malhotra, Pramodh Kumar, and Maheshwari (1978). The algorithm has performance $O(|V|^3)$ which represents a significant improvement over the $O(|V|^2 |E|)$ performance of the Edmonds-Karp version of Ford and Fulkerson. (For faster, but less simple algorithms, see the references.)

Let us introduce some terminology. Let (N,S,T,f) be a conservative flow. We

v_2,v_3	v_1,v_2,v_3	Scan (v_2,v_3)		v_3 in Tree
v_2,v_3	v_1,v_2,v_3	Scan (v_2,v_4)		Edge saturated
v_2,v_3	v_1,v_2,v_3	Dequeue(Q)		
v_3	v_1,v_2,v_3	Scan (v_3,v_2)		v_2 in Tree
v_3	v_1,v_2,v_3	Scan (v_3,v_4) Enqueue (v_4) Add v_4 to Tree	v_4: 2	Breakthrough

Increase Flow on v_1-v_3-v_4 by 2.

Flow_Augmenting_Path (Third Invocation)

QUEUE	TREE	ACTION	AMOUNT	REMARKS
v_1	v_1	Initialize	v_1: M	
v_1	v_1	Scan (v_1,v_2) Enqueue (v_2) Add v_2 to Tree	v_2: 1	
v_1,v_2	v_1,v_2	Scan (v_1,v_3) Enqueue (v_3) Add v_3 to Tree	v_3: 1	
v_1,v_2,v_3	v_1,v_2,v_3	Dequeue(Q)		
v_2,v_3	v_1,v_2,v_3	Scan (v_2,v_3)		v_3 in Tree
v_2,v_3	v_1,v_2,v_3	Scan (v_2,v_4)		Edge saturated.
v_2,v_3	v_1,v_2,v_3	Dequeue(Q)		
v_3	v_1,v_2,v_3	Scan (v_3,v_2)		v_2 in Tree
v_3	v_1,v_2,v_3	Scan (v_3,v_4)		Edge saturated.
v_3	v_1,v_2,v_3	Dequeue(Q)		
Empty				Search Blocked

Figure 6-11. Trace of Successive Calls to Flow_Augmenting_Path.

Figure 6-10 here

Flow_Augmenting_Path (First Invocation)

QUEUE	TREE	ACTION	AMOUNT	REMARKS
v_1	v_1	Initialize	v_1: M	
v_1	v_1	Scan (v_1,v_2) Enqueue(v_2) Add v_2 to Tree	v_2: 3	
v_1,v_2	v_1,v_2	Scan (v_1,v_3) Enqueue(v_3) Add v_3 to Tree	v_3: 3	
v_1,v_2,v_3	v_1,v_2,v_3	Dequeue(Q)		
v_2,v_3	v_1,v_2,v_3	Scan (v_2,v_3)		v_3 in Tree
v_2,v_3	v_1,v_2,v_3	Scan (v_2,v_4) Enqueue(v_4) Add v_4 to Tree	v_4: 2	Breakthrough

Increase Flow on v_1-v_2-v_4 by 2.

Flow_Augmenting_Path (Second Invocation)

QUEUE	TREE	ACTION	AMOUNT	REMARKS
v_1	v_1	Initialize	v_1: M	
v_1	v_1	Scan (v_1,v_2) Enqueue (v_2) Add v_2 to Tree	v_2: 1	
v_1,v_2	v_1,v_2	Scan (v_1,v_3) Enqueue (v_3) Add v_3 to Tree	v_3: 3	
v_1,v_2,v_3	v_1,v_2,v_3	Dequeue(Q)		

```
repeat


while  Head(Q) <> T  and*  Next (Head(Q), u, u-edge-ptr) do

      if   Bfs(u) = 0  and*
                        Try_to_Augment (Head(Q),u,u-edge-ptr)
      then Set Bfs(u) to 1
           Enqueue(Q,u)

until   Head(Q) = T   or*   Empty (Dequeue(Q))

(* Augment edge flow along FAP, if one was found *)

Set   Flow_Augmenting_Path to not Empty (Q)

if    Flow_Augmenting_Path

then  Set x to Head(Q)
      repeat
         Set   y to Pred(x)
         if    cap(x,y) - f(x,y) > f(y,x) (using Temp(y))
         then  Set f(x,y) to f(x,y) + Path-Val(T)
         else  Set f(y,x) to f(y,x) - Path-Val(T)
         Set x to y
      until x = S

End_Function_Flow_Augmenting_Path

Function Try_to_Augment (x, y, y-edge-ptr)

(*This tries to augment the flow on the forward or backward edges
   at x which are referenced via y-edge-ptr *)

var  N: Network
     Inc: Nonnegative Integer
     x: 1..|V|
     y-edge-ptr: Edge pointer
     Q: Queue pointer

Set Inc to max (cap(x,y) - f(x,y), f(y,x))
Set Try_to_Augment to (Inc > 0)

if Inc > 0  then  Set Pred(y) to x
                  Set Temp(x) to y-edge-ptr
                  Set Path-Val(y) to min (Path-Val(x), Inc)

End_Function_Try_to_Augment
```

We refer to Figures 6-10 and 6-11 for an example. Observe how the forward edges of the cut determined by the final blocked search tree in Figure 6-10d are saturated. The vertices in the blocked search tree define a cut $\{(v_2,v_4),(v_3,v_4)\}$ of capacity 4 equal to the maximum flow value.

```
     Edge   =  record
                 Shared-rep: Shared-edge pointer
                 Edge-successor: Edge pointer
               end
     Shared-edge = record
                     E(2,2): 0..|V|
                     cap(2), flow(2): Nonnegative Integer
                   end
```

Path-Val(v) stores the increment attainable on the partial flow augmenting path from S to v. We use Bfs to indicate whether a vertex has been scanned yet during the current invocation of Flow_Augmenting_Path. Bfs should be cleared prior to invoking Flow_Augmenting_Path. Pred(v) points to the search path predecessor of v. If the successor of v on the FAP is u, Temp(v) points to the entry for u on the edge list of v. This facilitates updating the edge flows on the FAP when one is found. Shared-rep points to the descriptor record for the edge. The vector (E(i,1),E(i,2)), i = 1..2, gives the index of the initial and terminal vertex of edge i and is zero if the edge does not exist. Thus, if (u,v) and (v,u) are both edges, (E(1,1),E(1,2)) equals (u,v) and (E(2,1),E(2,2)) equals (v,u). The fields cap(i) and flow(i) give the capacity and flow of the i^{th} edge.

We denote the breadth first queue by Q. We use a function Next(x, y, y-edge-ptr) to return in y the next neighbor of x together with a pointer in y-edge-ptr to the edge entry for y on the adjacency sublist of x, or fail. We consider the vertex y a neighbor of x if either (x,y) or (y,x) (corresponding to a forward or backward edge, respectively) is an edge. We also use a function Try_to_Augment(x, y, y-edge-ptr) which determines whether the edge (x,y) or (y,x) is augmentable. The function Nextcount is autoincrementing, starts at zero, and should be cleared prior to invoking Flow_Augmenting_Path.

```
Function Flow_Augmenting_Path (N,S,T)

(* This tries to find and apply a FAP from S to T, and fails if
   there is none *)

var  N: Network
     S,T,u,x,y: 1..|V|
     u-edge-ptr: Edge pointer
     k: Nonnegative Integer
     M: Integer Constant
     Next, Empty: Boolean function
     Flow_Augmenting_Path, Try_to_Augment: Boolean function
     Head: Integer function
     Q: Queue pointer
     Dequeue: Queue pointer function

Set Path-Val(S) to M
Enqueue (Q, S)

(* Try to find a FAP from S to T *)
```

Figure 6-8 here

We can characterize the maximum value flow in terms of flow augmenting paths as follows.

THEOREM (FLOW AUGMENTING PATH CHARACTERIZATION OF MAXIMUM FLOW) Let (N,S,T,f) be a conservative flow. Then, Conval(f) equals maxflow(f) if and only if there is no flow augmenting path with respect to (N,S,T,f).

The proof of the theorem will follow from the proof of correctness of the maximum flow algorithm.

MAXIMUM FLOW ALGORITHM

he Ford-Fulkerson algorithm uses procedure Flow_Augmenting_Path to find FAPs. The procedure constructs a search tree consisting of partial flow augmenting paths. If the search tree reaches T, the algorithm has found a flow augmenting path and the edge flows along the path are then changed, resulting in a greater flow value. On the other hand, if the search tree is blocked before reaching T, the blocked tree determines a cut of minimum capacity, and we can conclude that the flow value is a maximum. The maximum flow algorithm merely consists in calling Flow_Augmenting_Path repeatedly until it fails to find a further flow augmenting path. |E|

We use breadth first search in Flow_Augmenting_Path because it can be shown (Edmonds and Karp (1972)) that if the search algorithm uses breadth first search and a shortest augmenting path, then in that case we never need more than O(|V||E|) FAPs to obtain the maximum flow. Flow_Augmenting_Path is easily seen to be O(|E|), so the overall performance of this algorithm is then $O(|V||E|^2)$. Figure 6-9 illustrates a problem that can arise if other search techniques are used. Thus, if the search alternates between the FAPs. S-v_2-v_3-T and S-v_3-v_2-T, it takes 2M calls to Flow_Augmenting_Path before the maximum value flow is found; thus making the performance of the algorithm dependent on the (minimum) capacity of the network. This phenomenon is avoided by breadth first search.

Figure 6-9 here

We will represent the capacitated network N(V,E) as a linear array. Since we need to access both forward and backward edges, we use a shared representation for the edges. Thus, if directed edges (u,v) and (v,u) are incident with the vertices u and v, then the shared representation, which is pointed to by the edge entries on the adjacency sublists for both u and v, will contain the capacity and flow information for both edges. The vertex and edge types are as follows.

```
type Vertex =  record
                 Path-Val: Nonnegative Integer
                 Bfs, Pred: 0..|V(N)|
                 Positional-Pointer,
                         Successor, Temp: Edge pointer
               end
```

We will require some preliminary definitions. Let (N,S,T,f) be a conservative flow. We say an edge x in E(N) is *saturated* with respect to (N,S,T,f) if f(x) equals cap(x). We define the *spare capacity* of an edge x as the difference cap(x) - flow(x). The simplest kind of flow augmenting path is a directed path from S to T that contains no saturated edges. If we denote the minimum of the spare capacities taken over all the edges of such a path by Inc and increment the edge flow on every edge of the path by Inc, then the resulting flow has flow value Conval(f) + Inc.

It may happen that the flow value Conval(f) is less than maxflow(N) even when there are no such directed paths along which we can increase the flow. Figure 6-6 gives an example. However, we can define flow augmenting paths of such generality that their absence guarantees the flow value is maximum.

Figure 6-6 here

We define a *flow augmenting path* (FAP) with respect to a conservative flow (N,S,T,f) as an alternating sequence of vertices and edges of N(V,E):

$$v_1, e_1, \ldots, v_i, e_i, \ldots, v_n$$

such that

(1) v_1 equals S and v_n equals T.

(2) Each edge e_i ($i = 1, \ldots, n - 1$) is either of the form (v_i, v_{i+1}) or of the form (v_{i+1}, v_i) and is called respectively a *forward* or a *backward* edge of the sequence.

(3) Every forward edge e_i ($i = 1, \ldots, n - 1$) of the sequence has positive spare capacity, that is

$$cap(e_i) - flow(e_i) > 0.$$

(4) Every backward edge e_i ($i = 1, \ldots, n - 1$) has positive flow, that is

$$flow(e_i) > 0.$$

We refer to Figure 6-7 for an example. The Ford and Fulkerson maximum flow algorithm is essentially a search tree technique for finding flow augmenting paths.

Figure 6-7 here

In order to increase the flow value using flow augmenting paths, we let M denote the minimum of the spare capacities on the forward edges of the FAP and let m denote the minimum of the edge flows on the backward edges of the FAP. Denote min(M,m) by Inc. We then change the edge flows along the FAP by increasing the edge flow on each forward edge by Inc and by decreasing the edge flow on each backward edge of the FAP by Inc. The resulting conservative flow has flow value Conval(f) + Inc. Refer to Figure 6-7 for an example where Inc equal to min(3,2) and also to Figure 6-8.

```
     Next, Empty, Get_Flow_Path: Boolean function
     Nextcount,Top: Integer function
     ST: Stack pointer
     Pop: Stack pointer function

Set k to Nextcount
Set Dfs(S) to k
Set Path-Val(S) to M
Create(ST); Push (ST,S)

(* Try to find an S to T flow path *)

repeat



while  TOP(ST) <> T  and*  Next (TOP(ST), u) DO

    if    Dfs (u) < k  and  f(TOP(ST),u) > 0
    then  Set Dfs (u)  to  k
          Set Pred(u) to Top(ST)
          Push(ST,u)
          Set Path-Val(u) to min{Path-Val(Top(ST)),f(TOP(ST),u)}

until   TOP(ST) = T  or*  EMPTY (POP(ST))

(* If non-zero flow path found, reduce affected edge flows *)

Set Get_Flow_Path to not EMPTY(ST)

if    Get_Flow_Path

then  (* Reduce flow along path *)
      Set Val to Path-Val(T)
      Set x to Top(ST)
      while x <> S  DO
        Set p  to Pred(x)
        Set f(p,TOP(ST))  to f(p,TOP(ST)) - Path-Val(T)
        Set x to p

End_Function_Get_Flow_Path
```

Figure 6-5 here

6-4 MAXIMUM FLOW ALGORITHM: FORD AND FULKERSON

The maximum flow algorithm of Ford and Fulkerson uses generalized flow paths to modify the values of the edge flows of a conservative flow in a way that both preserves the conservative nature of the flow and increases the value of the flow. The "paths" are called flow augmenting paths and not only provide a means of increasing the value of a flow, but also allow us to characterize maximum flows algorithmically.

identically zero because "circulatory" flows not contributing to any S-T flow may remain.

We represent the capacitated network N(V,E) as a linear array with vertices and edges of type

```
type  Vertex  =  record
                   Cap, Path-Val: Nonnegative Integer
                   Dfs, Pred: 0..|V(N)|
                   Positional-Pointer, Successor: Edge pointer
                 end

      Edge    =  record
                   Nghb: 1..|V(N)|
                   f (flow): Nonnegative Integer
                   Edge-successor: Edge pointer
                 end
```

The meaning of most of the fields is clear from the field names. Path-Val(v) stores the value of the flow on the depth first flow path to v. Pred(v) points to the predecessor of v in the depth first tree. Dfs is the field that would usually contain the depth first number of the vertex; however, we use it as a flag which indicates whether the vertex has been visited yet during the current invocation of Get_Flow_Path. The source and sink are distinct vertices S and T in V(N). The depth first stack is denoted by ST. The flow path (or at least its successive vertices in reverse order) is returned in ST, while the value of the flow path is returned in Val. M denotes some large positive integer which plays the role of plus infinity.

We denote the conditional and (or) operator by and* (or*). The second condition in A and* B (A or* B) is not tested if the first condition fails (succeeds). The auto-increment function Nextcount is initially zero, as are the Dfs fields. A global counter k is incremented by one at each invocation of Get_Flow_Path. When we consider the maximum flow algorithm in the next section, we will need to enhance the network representation so as to facilitate access to both adjacent to and adjacent from vertices, since the maximum flow algorithm considers both incoming and outgoing edges at each vertex. But, for our present purposes, we only need adjacent to vertices, so the function Next(x,y) may be defined as usual.

We refer to Figure 6-5 for an example of the procedure. The final conservative flow in the example has value zero, though it is not identically zero.

```
Function Get_Flow_Path (N, S, T, ST, Val)

(* This returns an S to T flow path in ST with value Val,
   reducing the conservative flow (N,S,T,f) accordingly, and
   fails if there is none. *)

var  N: Network
     S,T,u,x,p: 1..|V|
     k, Val: Nonnegative Integer
     M: Integer Constant
```

$\leq$ Cap(X,X^C),

by the Net Flow theorem and the nonnegativity of flow(X^C,X). Consequently, the value of any conservative flow (N,S,T,f) is bounded above by the capacity of every S-T cut. Hence, the maximum flow value is bounded by the minimum cut capacity. This completes the proof of the theorem.

EQUIVALENCE OF PATH AND CONSERVATION MODELS

We will now show that the path and the conservative flow models are equivalent in the sense that given any path flow there is a conservative flow of equal value, and conversely.

THEOREM (CONSERVATIVE AND PATH FLOW EQUIVALENCE) If (N,S,T,P,f') is a path flow, there exists a conservative flow (N,S,T,f) such that Value(f') equals Conval(f). Conversely, if (N,S,T,f) is a conservative flow, there exists a path flow (N,S,T,P,f') such that Conval(f) equals Value(f').

The proof is as follows. To convert a path flow (N,S,T,P,f') into a conservative flow (N,S,T,f) of the same value, we define for each edge x in E(N) a function f(x) equal to Sum(f',x). The function f is nonnegative and satisfies both the capacity and conservation of flow constraints at vertices other than S or T which is the definition of an edge flow. The quadruple (N,T,S,f) then defines a conservative flow such that Conval(f) equals Inflow(T) which equals Value(f) as required.

We can convert a conservative flow (N,S,T,f) into a path flow (N,S,T,P,f') of the same value by repeatedly identifying and "removing" S to T paths in the conservative flow network all of whose edges have positive flow. Whenever we find a path, we assign it a path flow value equal to the minimum of the edge flows on the path, and we reduce the edge flows along the path by the value of the path flow. We repeat the procedure until we can no longer find a nonzero flow path at which point the total value of the path flows removed equals the value of the original conservative flow.

To prove that we can repeat this process until the value of the conservative flow has been reduced to zero, we argue as follows. If the conservative flow has nonzero flow value, then we can prove by the Net-Flow theorem that there exists a flow path of nonzero value. We let X denote the set of vertices in V(N) reachable from S by paths containing only edges with nonzero edge flows. If T is in X, we can certainly find a nonzero path flow from S to T. On the other hand, if T is not in X, then (X,X^C) is an S-T cut and all the edges in the cut have zero edge flows. Therefore, flow(X,X^C) is zero whence by the Net-Flow Theorem and the nonnegativity of the edge capacities, flow(X^C,X) is also zero. Thus, the conservative flow already has value zero, which completes the proof of the theorem.

A depth first search procedure for extracting flow paths Get_Flow_Path is given below. The procedure trys to find a path from S to T containing only edges with nonzero edge flows which thus defines a nonzero flow path. We can call Get_Flow_Path repeatedly until Outflow(S) is reduced to zero. The final reduced flow network has zero flow value, although the flow mapping may not be

$$\sum_{x} f(x,v) - \sum_{y} f(v,y) = 0$$

where the first sum is taken over all vertices x in V(N) adjacent to v, while the second sum is taken over all vertices y in V(N) adjacent from v. At the sink T, we have

$$\sum_{x} f(x,T) - \sum_{y} f(T,y) = Conval(f),$$

where x and y are defined similarly. The sum of the right hand sides taken over all these equations is Conval(f), while we can show the sum of the left hand sides taken over all the equations equals $flow(X,X^c) - flow(X^c,X)$. For, any edge (x,y) with x and y both in X^c, contributes opposite terms to the sum of the left hand sides. The edge (x,y) generates a positive term +f(x,y) considered as an incoming edge at y and an opposite negative term -f(x,y) when considered as an outgoing edge at x; so that the terms of this form cancel. The only remaining terms arise from flows on edges in (X,X^c) (positive terms) and flows on edges in (X^c,X) (negative terms); so the equations overall sum to

$$flow(X,X^c) - flow(X^c,X) = Conval(f),$$

which completes the proof of the theorem.

The next theorem follows readily from the Net-Flow Theorem. It shows that the value of the maximum possible S to T flow realizable on a given capacitated network is never greater than the capacity of the smallest cut S-T on the network. We will use this theorem later in establishing both the termination and correctness of the maximum flow algorithms. We will also see that the upper bound in the theorem is always attainable, that is, the maximum flow value equals the minimum cut capacity.

THEOREM (MAXIMUM FLOW / MINIMUM CAPACITY CUT BOUND) Let N(V,E) be a capacitated network and let S and T be distinct vertices in N. Then,

$$maxflow(N,S,T) \leq mincap(N,S,T).$$

The proof of this theorem is as follows. Observe that for any conservative flow (N,S,T,f) and any S-T cut (X,X^c)

$$flow(X,X^c) \leq Cap(X,X^c),$$

by the capacity constraints on a flow. Consequently,

$$\begin{aligned} Conval(f) &= Netflow(X,X^c) \\ &= flow(X,X^c) - flow(X^c,X) \end{aligned}$$

N(V,E) be a capacitated network and let S and T be a pair of vertices in N. Let f be a nonnegative integral function on E(N). We denote the sum of the values of f on incoming edges at a vertex v in V(N) by Inflow(v). Outflow(v) is defined similarly. We call f an *edge flow on N* if

(1) For every edge x in E(N), $f(e) \leq Cap(x)$.

(2) For every vertex v <> S or T, Inflow(v) = Outflow(v).

Condition (1) is called a *capacity constraint*, while (2) is called a *conservation constraint*. We define a conservative flow on N from S to T as a quadruple (N,S,T,f) where f is an edge flow on N. If x is an edge in E(N); then f(x) is called the *edge flow on x*. The value of a conservative flow (N,S,T,f) is the net flow into T, Inflow(T) - Outflow(T), and is denoted by Conval(f). We denote the value of a maximum possible value conservative flow from S to T on a capacitated network N by maxflow(N,S,T).

We will prove that any conservative flow can be interpreted as a superposition of path flows of equivalent flow value, and conversely. But, first we need to establish some basic bounds on the value of a flow.

MAXIMUM NETWORK FLOWS AND MINIMUM CAPACITY CUTS

The fundamental bound on the maximum value of a path flow or a conservative flow on a network N with source S and sink T can be informally stated as follows: The maximum flow value is bounded above by the tightest "bottleneck" between S and T. We now formalize this intuitive notion.

Let (N,S,T,f) be a conservative flow. If X is a set of vertices in V(N), then X^C denotes the complement of X with respect to V(N), that is, V(N) - X. We define a *cut* (X,X^C) as the set of edges { (x,x') | x in X and x' in X^C}. If S is in X and T is in X^C, then (X,X^C) is called an S-T cut. If (X,X^C) is an S-T cut, there is no path from S to T in $N(V, E - (X,X^C))$. We define the *capacity of a cut*, denoted by $Cap(X,X^C)$, as the sum of the capacities of the edges in (X,X^C). We denote the capacity of a minimum capacity S-T cut in N by mincap(N,S,T). The flow on (X,X^C), denoted $flow(X,X^C)$, is the sum of the edge flows on edges in (X,X^C). The net flow across (X,Xc), denoted by $Netflow(X,X^C)$, is $flow(X,X^C) - flow(X^C,X)$. Refer to Figure 6-4 for an illustration.

Figure 6-4 here

THEOREM (NET FLOW AND FLOW VALUE) Let (N,S,T,f) be a conservative flow and let (X,X^C) be an S-T cut in N. Then,

$$flow(X,X^C) - flow(X^C,X) = Conval(f).$$

The proof of the theorem is as follows. For every vertex v in X^C other than T, the following conservation equations hold:

we denote the set of paths in P that include the edge x by P(x). Let f be a nonnegative integral function defined on P. We denote the sum of the values of f(p) taken over all paths p in P(x) by Sum(f,x). We call f a *flow on P* if for every edge x in E(N), Sum(f,x) is at most Cap(x). If f is a flow on P and p is a path in P, we call f(p) the *flow on p*. If x is an edge in E(N), we call Sum(f,x) the *flow on x*. The set of paths P together with a flow f on P is called a *path flow on N* and is specified by the quintuple (N,S,T,P,f). The value of a path flow (N,S,T,P,f) is the sum of the values of f(p) taken over all paths p in P and is denoted by Value(P,f), the other parameters being left implicit. The vertices S and T are called the *source* and *sink* of the path flow, respectively.

Figure 6-3 gives an example. S and T are v_1 and v_4 respectively, the paths P are given by $\{v_1\text{-}v_2\text{-}v_4,\ v_1\text{-}v_3\text{-}v_4,\ v_1\text{-}v_2\text{-}v_3\text{-}v_4\}$, and f(p) is 1 for each path. Sum(f,x) is given for each edge in Figure 6-3b and is within capacity constraints. Value(P,f) equals 3. We obtain a different path flow if we set P equal to $\{v_1\text{-}v_2\text{-}v_3\text{-}v_4\}$ and f(p) equal to 2. Value(P,f) equals 2, which we know from the previous example is less than the maximum path flow value on N. Interestingly, we cannot augment the value of this flow by adding additional flow paths to P because the existing path prevents this. A path flow with this characteristic is called unaugmentable or *maximal*. The existence of maximal but not maximum value path flows demonstrates how the resources of a network can be squandered by routing traffic carelessly.

Figure 6-3 here

CONSERVATION MODEL OF NETWORK FLOW

The *conservation model* interprets the traffic flow on a network as composed of edge flows which are subject to conservation constraints at the vertices of the network, rather than as a superposition of flow paths. We will see this is mathematically and algorithmically more convenient to work with than the path flow model.

We introduce the conservation model with an example. Consider the path flow in Figure 6-3b. If we denote the sum of the flows on the flow paths entering a vertex v by Path-inflow(v), and the sum of the flows on the flow paths exiting a vertex v by Path-outflow(v), the following relations hold.

(1) Path-inflow(S) = 0, Path-outflow(S) = 3.

(2) Path-inflow(T) = 3, Path-outflow(T) = 0.

(3) Path-inflow(v_1) = Path-outflow(v_1) = 2, and
Path-inflow(v_2) = Path-outflow(v_2) = 2.

Thus, Path-Inflow and Path-Outflow are balanced at every every vertex other than the source S and the sink T. Conversely, the *net flow out of S* (defined as Path-outflow(S) - Path-inflow(S)) equals the *net flow into T* (defined as Path-inflow(T) - Path-outflow(T)) and each equals 3, the value of the path flow.

These relations are typical and suggest the following definitions. Let

route and so can delete the message. It follows similarly that the routing description delivered with an incorrect message must be for a bad path, though not necessarily the bad path that was actually taken.

There are at most F bad paths from T to P since there are no more than F faulty processors. Since T uses 2F + 1 disjoint paths from T to P, at least F + 1 valid copies of the message transmitted must reach P. An even larger number of invalid messages can also arrive at P, but P can filter out the invalid messages by the following technique of identifying a suspicious set of paths U from T to P.

Let us first observe that some suspicious sets of paths certainly exist. For example, if we set U to the set of disjoint paths that actually contain faulty processors, U is a suspicious set since there are at most F such paths and any messages not passing through U will have the same (correct) value.

We can show that the common message value transmitted not using a suspicious set U is always the correct message value. To prove this, observe that U contains at most F good paths. Since there are at least F + 1 good paths, there is at least one good path not in U. Therefore, at least one correct copy of the message is routed outside U. By the definition of a suspicious set, only a single message value is routed outside U. Therefore, that value must always be the correct one. This completes the proof of the theorem.

6-3 NETWORK FLOWS: BASIC CONCEPTS

A *network* is a digraph G(V,E) with nonnegative, real-valued weights assigned to each edge. Alternatively, a network is an ordered triple (V,E,w), where V is a set of vertices, E a set of directed edges on V, and w is a function mapping the elements of E to the nonnegative reals. If we interpret the edge weights as limits on the capacities of the edges to transmit a commodity of some type (such as, vehicular traffic, data, fluids), the network is called a *capacitated network*. We will show how to find the maximum amount of traffic that can be transmitted between a given pair of vertices in a capacitated network. This is called the maximum network flow problem. We present two algorithms for finding the maximum flow: the classical one of Ford and Fulkerson, and a more efficient but more complicated algorithm based on the work of Dinic. The Ford-Fulkerson algorithm uses a search tree to iteratively build up the flow in the network; while the Dinic algorithm uses a "layered network" for the same purpose.

Before proceeding with a description of the maximum flow algorithms, we will need to introduce some terminology, beginning with a precise definition of what we mean by a traffic flow. There are two ways to define this, either by a path model of flow, or using conservation equations. While the two methods are mathematically equivalent, the path model is more intuitive, while the conservation model is mathematically more convenient. We give each definition, and then prove their equivalence.

PATH MODEL OF NETWORK FLOW

Let N(V,E) be a capacitated network. Let S and T be a pair of distinct vertices in V(N), and let P be a set of paths from S to T. If x is an edge in E(N), we denote the (nonnegative integral) capacity of x by Cap(x), and

multiple incorrect copies of messages. It is not at all obvious whether reliable transmission can be achieved in the presence of such potential chaos. But, surprisingly, we can show that if the interconnection graph for the network has sufficiently high vertex connectivity, then the reliable processors in the network can communicate reliably even in the presence of severely unreliable components (Dolev (1982)).

It will be convenient to introduce the following terminology. Let S be a set of vertex disjoint paths in G(V,E) from a reliable processor T to a reliable processor P. We will define a path in S to be *good* if every processor on the path is reliable. We define a path in S to be *bad* if there is at least one faulty processor on the path. Let F be an upper bound on the number of faulty processors in G. A suspicious sets of paths from T to P is a set U of at most F vertex disjoint paths in G with the property that all messages from T to P which are not routed through U arrive at P with the same message value. Refer to Figure 6-2 for an illustration.

Figure 6-2 here

Before describing the reliable routing procedure, let us specify some assumptions.

(1) Each reliable processor can always correctly identify that adjacent processor from which it receives a given message even when the adjacent processor is unreliable.

(2) Each reliable processor knows the global definition of the communication network graph G and the sets of disjoint paths used by each reliable processor for processor to processor communication.

Under these assumptions we can prove the following theorem.

THEOREM (RELIABLE TRANSMISSION CONDITION) Let G(V,E) be a graph model of a network with at most F faulty processors. Let the vertex connectivity of G be at least 2F + 1. Suppose T is a reliable transmitter in G which sends 2F + 1 copies of a message m to a reliable processor P through 2F + 1 known vertex disjoint paths. Then, P can correctly receive m by finding a set of suspicious paths U from T to P and selecting the unique message value not routed through U.

The proof of the theorem is as follows. Let M denote the set of messages received at P. Some of these are correct copies of the original message m and some are invalid copies. P must try to determine which represent(s) the correct message. Each message contains a description of the path it took from T to P. Of course, the description may have been perverted by a faulty processor along that route, and the message itself could be spurious. Nonetheless, since the 2F + 1 paths T uses to transmit to P are known (by assumption (2)) both to P and to all the other reliable processors in the network, the reliable processors can ensure in the first place that incorrect messages are restricted to bad paths. For example, if an unreliable processor attempts to move an incorrect message from a bad path to a good path, the first (reliable) processor on the good path to receive the message will recognize (by assumption (1)) that the route is an invalid T to P

of vertices lie on a cycle. The following theorem generalizes this.

THEOREM (CONNECTIVITY GUARANTEED CYCLES) If G(V,E) is an n vertex connected graph, $n \geq 2$, every n vertices in G lie on some cycle.

It is nontrivial to calculate the connectivity of a graph or to identify the disjoint paths between vertices which are guaranteed by a given level of connectivity. For graphs of connectivity one, we can use depth first search to calculate both the connectivity and to identify the connecting paths. The orientation and block algorithms essentially determine if a graph has connectivity at least 2. For higher levels of connectivity, the evaluation of connectivity and the identification of disjoint paths is generally done using maximum flow algorithms, as we shall describe in a later section.

6-2 CONNECTIVITY MODELS: VULNERABILITY AND RELIABLE TRANSMISSION

VULNERABILITY OF NETWORKS

The number of vertices or edges that must fail before a network becomes disconnected is a common measure of the *reliability* or *vulnerability* of a communication network. The corresponding graphical invariants are the vertex or edge connectivity of the network. It is natural to ask how to design a network with a given number of vertices, which attains a prescribed level of reliability, as cheaply as possible, where we measure the cost of constructing the network in terms of the number of its edges. In graph-theoretic terms, the question is how do we construct a graph G(V,E) of given order and prescribed vertex connectivity, but of minimimum size?

There is an extensive literature on such extremal problems where some graphical invariants are held fixed, while others are optimized. One class of graphs that solve the minimum cost reliable network design problem just posed are the so-called *circulants*, defined as follows. Let C(n) be a cycle of order n. For even values of k, the graph C(n)(k/2), has connectivity k, and the minimum number of edges (nk/2) among all graphs of connectivity k and order n.

RELIABLE TRANSMISSION (BYZANTINE GENERALS PROBLEM)

Consider a distributed system of processors whose interconnection network is given by an undirected graph G(V,E). Assume there is a processor at each vertex of G which can communicate directly with the neighboring processors at adjacent vertices. To transmit a message to a distant processor, a processor includes the intended route with its message and then routes the message through the network, by transmitting it to the first processor/ vertex on the route, which proceeds in a similar manner, and so on, until the message eventually reaches its destination.

We are interested in the kind of problems that can arise when some of the processors in the network may be faulty. We will assume that faulty processors can exhibit a bizarre variety of deleterious effects. They may lose messages, route messages incorrectly, garble messages, or even generate

THEOREM (PATH DIVERSITY CHARACTERIZATION OF CONNECTIVITY) Let G(V,E) be a graph and let k be an integer. Then,

(1) If the removal of k (but not less than k) vertices (edges) disconnects a pair of vertices u and v in G, there are at least k vertex (edge) disjoint paths in G from u to v;

(2) If G is k vertex (edge) connected, there are at least k vertex (edge) disjoint paths between every pair of vertices in G.

Figure 6-1 gives an example. Variations on this theorem guarantee vertex disjoint paths between a given vertex and a set of other vertices or between sets of vertices, as follows.

Figure 6-1 here

THEOREM (DIVERSE VERTEX-TO-SET ROUTING) If G(V,E) is an n vertex connected (edge connected) graph and v, v_1 ,..., v_n are n + 1 disjoint vertices of G, then there are n vertex disjoint (edge disjoint) paths P_i from v to v_i, i = 1..n.

For example since the graph in Figure 6-1 has edge connectivity 2, there must be two edge disjoint paths between v_1 and $\{v_4, v_5\}$, such as the paths v_1-v_3-v_4 and v_1-v_2-v_3-v_5. More generally, we have the following.

THEOREM (DIVERSE SET-TO-SET ROUTING) Let G(V,E) have order at least 2n. Then, G is n vertex connected if and only if for every pair of disjoint sets of vertices X and Y which are each of cardinality n, there exist n vertex disjoint paths between X and Y.

The proof of this theorem is as follows. First, observe that it is easy to show that if G is n vertex connected, there must be n vertex disjoint paths between X and Y. We merely add artificial vertices x and y to G which we make incident with every vertex in X and Y, respectively. The resulting graph G' is still n vertex connected. Therefore by the Path Diversity Theorem there are n vertex disjoint paths between x and y in G', which in turn determine n vertex disjoint paths between X and Y in G.

Conversely, we can show that the existence of n disjoint paths between every disjoint pair of sets of cardinality n implies G is n vertex connected. Thus, let S be a vertex disconnecting set in G which disconnects G into two parts which we denote by S_1 and S_2. $|S \cup S_1 \cup S_2|$ is at least 2n, so we can partition S into two disjoint parts S'_1 and S'_2 such that $|S_1 \cup S'_1|$ and $|S_2 \cup S'_2|$ are both at least n. Define X and Y to be any subsets of $S_1 \cup S'_1$ and $S_2 \cup S'_2$, respectively, of cardinality n. By supposition, there are n vertex disjoint paths from $S_1 \cup S'_1$ to $S_2 \cup S'_2$. Since S separates S_1 from S_2, each of these paths must contain a vertex of S; hence S must contain at least n vertices. This completes the proof.

Our earlier characterization of orientable graphs implied that every pair of vertices in a (connected) bridgeless (two-edge connected) graph lie on a circuit. One can easily show for a two- vertex connected graph that every pair

Chapter 6. CONNECTIVITY AND ROUTING

We have already defined the *vertex connectivity* VC(G) and *edge connectivity* EC(G) of a graph G. These invariants measure the susceptibility of a graph to disconnection under vertex or edge deletion. From one viewpoint, they calibrate the vulnerability of the graph considered as a communication network to disruption. It is remarkable that precisely these measures of strength of connection (or vulnerability to disconnection) also determine the diversity of routing in the graph, that is, the number of disjoint paths between pairs of vertices. After introducing some of the basic theoretical results for these invariants, we will illustrate their use in modelling and then describe certain network flow algorithms and how they can be used to calculate the vertex and edge connectivity.

6-1 CONNECTIVITY: BASIC CONCEPTS

We can define VC(G) and EC(G) for a graph G(V,E) in terms of parametrized versions of these invariants which will be useful later in developing algorithms for VC and EC. If u and v are nonadjacent vertices in G, we define a *vertex disconnecting set (or vertex separator) between u and v* in G as a set of vertices S in V(G) such that u and v are in different components of G - S. S is said to *disconnect* (or *separate*) the pair of vertices u and v. An *edge disconnecting set between a pair of vertices u and v* in G is similarly defined as a set of edges T in E(G) such that u and v are in different components of G - T. For nonadjacent vertices u and v, we define the *vertex connectivity between u and v*, denoted VC(G,u,v), as the cardinality of the smallest vertex disconnecting set between u and v. On the other hand, EC(G,u,v) denotes the cardinality of the smallest edge disconnecting set between the pair of vertices u and v. We can then define VC(G) as the minimum of VC(G,u,v) taken over all distinct, nonadjacent vertices u and v in G (or |V| - 1 if G is a complete graph), while EC(G) is the minimum of EC(G,u,v) taken over all distinct vertices u and v in G. It is easy to prove:

THEOREM (BOUNDS ON CONNECTIVITY) For every graph G(V,E)

$$VC(G) \leq EC(G) \leq \min(G).$$

There are a variety of conditions on the degree sequence of a graph that guarantee lower bounds on connectivity, such as the following.

THEOREM (CONNECTIVITY LOWER BOUND) Let G(V,E) be a graph of order greater than one, and let n satisfy $1 \leq n \leq |V| - 1$. If $\min(G) \geq (|V(G)| + n - 2) / 2$, then $VC(G) \geq n$.

By the previous theorem, this also establishes a lower bound on edge connectivity.

The precise relation between the connectivity of a graph and the number of disjoint paths that exist between pairs of vertices in the graph is the subject of the following classical theorem. Let us say a pair of paths are *vertex disjoint* if they have at most their endpoints in common; while a pair of paths are *edge disjoint*, if they have no edges in common.