1. Dynamic Programming

**Weighted Interval Scheduling**

- Input: set of requests {1,...,n} with start time s(i), end time f(i) and weight vi

- Objective:

Select a subset S{1,...,n} of mutually compatible intervals so as to maximize

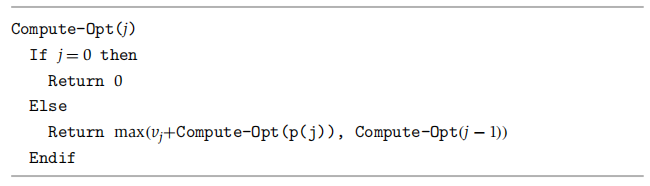
- Solution (find the optimal value):

Sort requests in order of non-decreasing finish time f1≤f2≤...≤fn →O(nlogn)

*P(j):* the largest index i<j such that interval i & j are disjoint →O(nlogn)

*Oj*: the optimal solution to the problem consisting of requests {1...j}

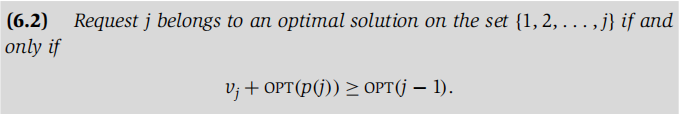
*OPT(j)*: the value of Oj

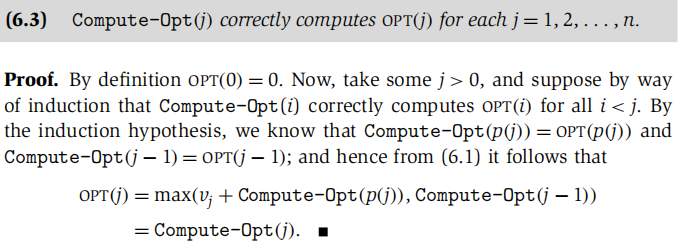


**Analysis**

- take exponential time to run in the worst case and cannot take only polynomial-time

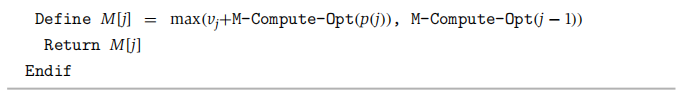
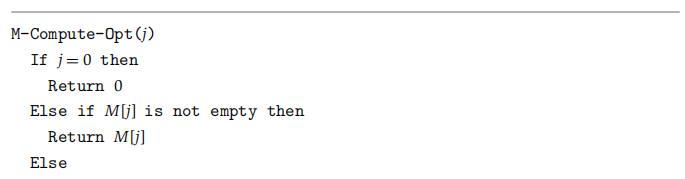






**Memoization (polynomial-time)**

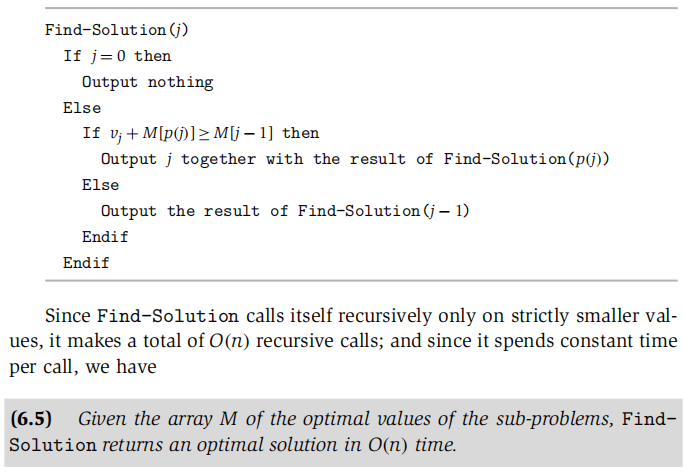
- store the value of Compute-Opt in a globally accessible place the first time we compute it and use this precomputed value in place of all future recursive calls.



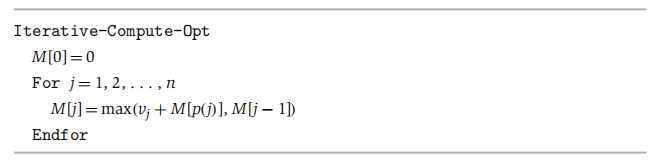
- Time Complexity: Θ(n)

- Initial Sorting = Build P(j) = Overall Time complexity: Θ(nlogn)

**Find optimal set of intervals**



**Iteration over Sub-problems**



**Video-game Problem**

- Objective: go home with the least lost energy.

- Choices:

- walk into next stage costs 50 units.

- jump over one stage costs 150 units.

- jump over two stages costs 350 units.

- In general, we lose ei units of energy when landing on stage i.

*OPT(i)*: max amount of energy we could have when landing on stage i

= Max(OPT(i-1)-50-ei, OPT(i-2)-150-ei, OPT(i-3)-350-ei)

- Time complexity: Θ(n)

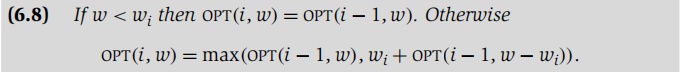
**0-1 knapsack & subset sum**

- a single resource and requests {1...n} each takes time wi to process

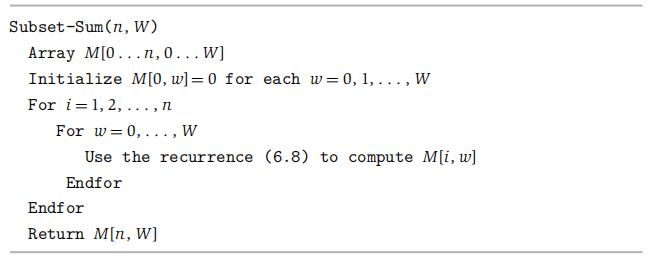
- we could schedule jobs at any time between 0 to W

- Objective: to schedule jobs such that maximize the machine’s utilization

*OPT(i,w)*: value of the opt-sol using a subset of items {1..i} with max-allowed w

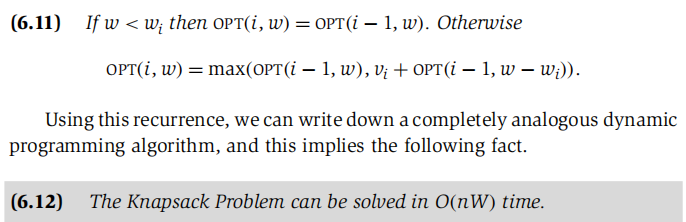


- Solution:



- Time Complexity (pseudo-polynomial) : O(nW)

- Extension (with different value vi)

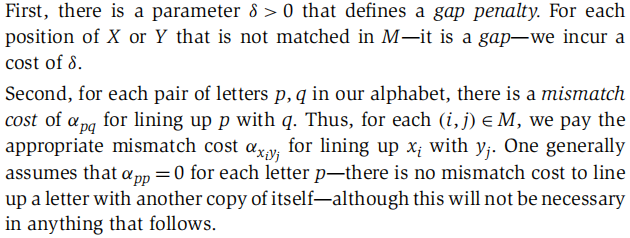


**Sequence Alignment**

- *Matching*: a set of ordered pairs with property that each item occurs at most once.

- *Alignment*: a matching without crossing pairs, (i, j), (i`, j`)∈M & i < i`, then j < j`.

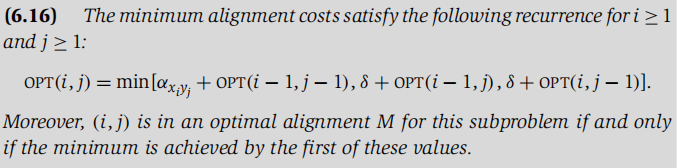
- Suppose M is a given alignment between X and Y.



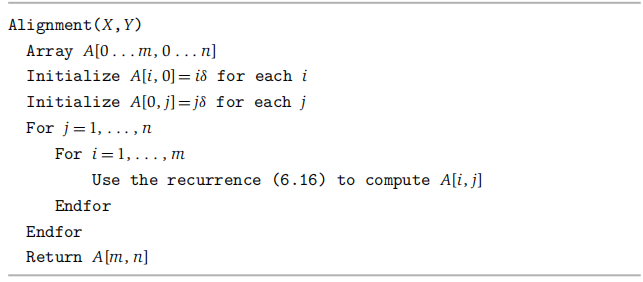
- Objective: Seeking an alignment of minimum cost (gap + mismatch costs)

*OPT(i,j)*: the minimum cost of an alignment between x{1..i} and y{1..j}

= min((m,n)∈M, the mth position of X is not matched, the nth position of Y is not matched)



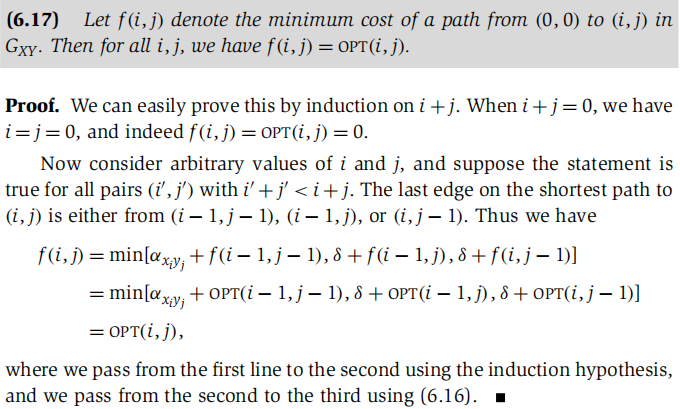
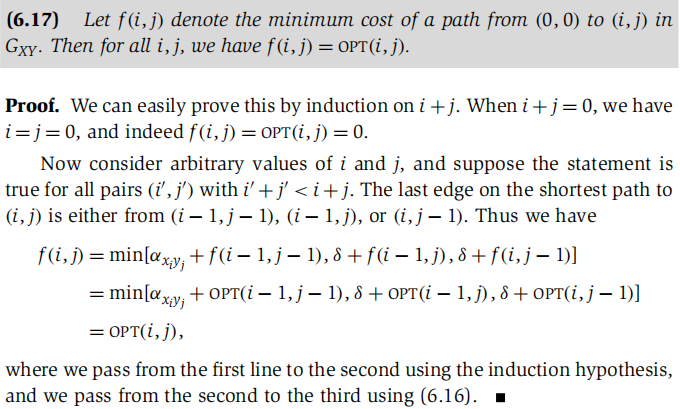
- Solution: O(mn)



**Analysis**

- Suppose we build a two-dimensional m×n grid graph *GXY*, with the rows labeled by symbols in the string X, the columns labeled by symbols in Y

- the cost of each horizontal and vertical edge is gap penalty, and the cost of the diagonal edge from (i-1, j-1) to (i, j) is the mismatch cost.



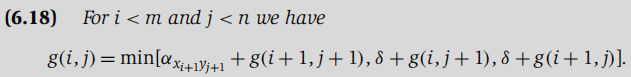
**Sequence Alignment in Linear Space via Divide and conquer**

- decrease the space requirement to O(m+n)

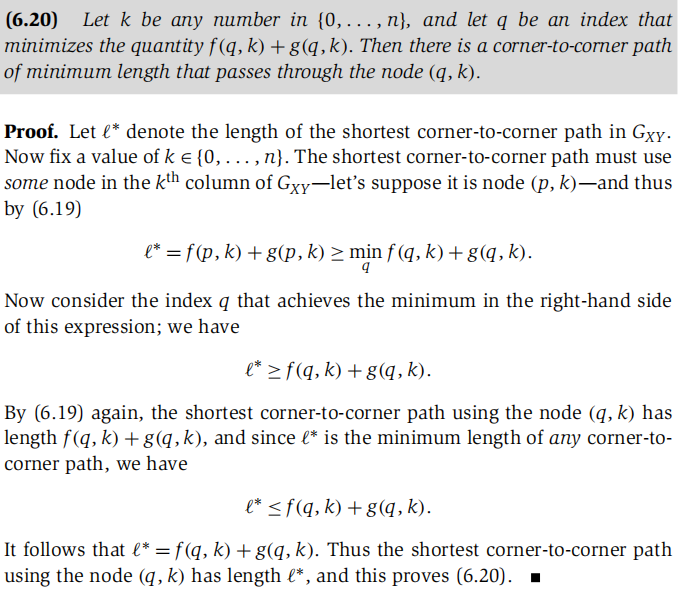
- only care the value of the optimal alignment and not the alignment itself

- Solution:

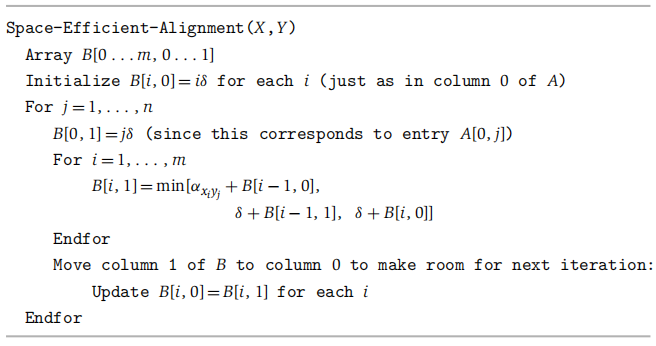
1. Divide GXY along its center column and compute the value of f(i, n/2) and g(i, n/2) for each value of i, using two space-efficient algorithms.
2. A backward formulation of the Dynamic Program

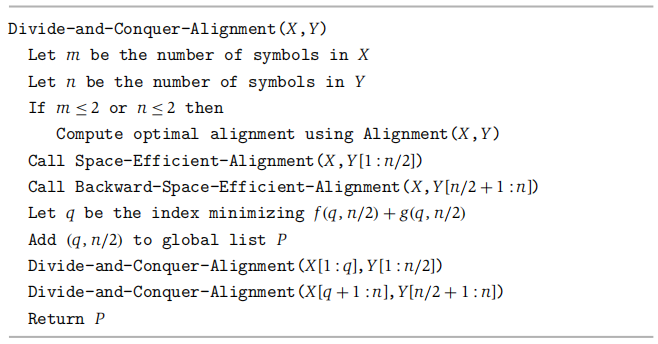


1. Search for the shortest path recursively in the portion of GXY
2. Combining the forward and backward formulations

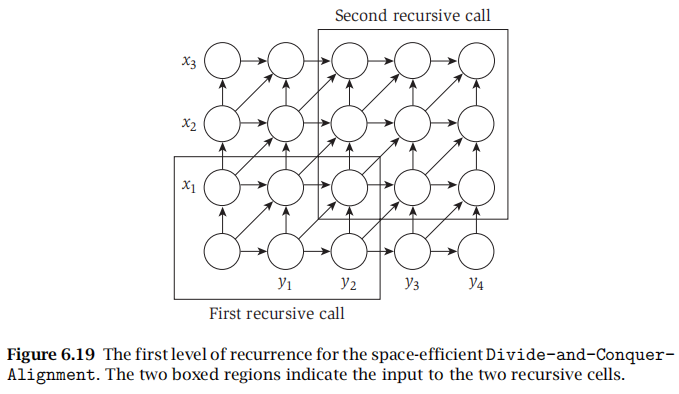


- Algorithm





- Time Complexity: O(mn)



**Matrix Chain Multiplication**

- Input: matrix sequence {A1, ..., An}, Dimension of Ai is Ri×Ci

- Objective: Seeking minimum total number of operations

*OPT(i,j)*: the optimal cost of multiplying matrix {Ai, ..., Aj} (i≤k＜j)

= min(OPT(i,k)+OPT(k+1,j)+RiCkCJ)

- Solution:

For i = 1 to n, OPT(i,i) = 0;

For j = 2 to n O(n)

For i = j-1 to 1 by -1 O(n)

min(OPT(i,k)+OPT(k+1,j)+RiCkCJ); O(n)

Endfor

Endfor

Time Complexity: O(n³)

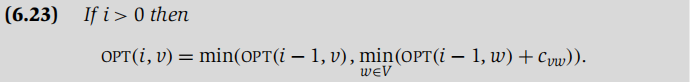
**Shortest Paths in a Graph**

- there are negative edge costs but no negative cycles.

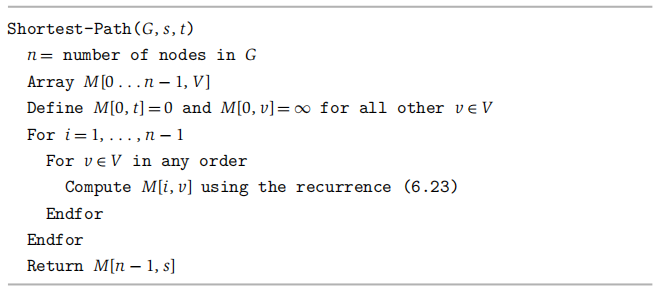
- Base: If G has no negative cycles, then there is a shortest path from s to t that is simple (does not repeat nodes), and hence has at most n-1 edges.

*OPT(i,v)*: the min cost of a v-t path using at most i edges.

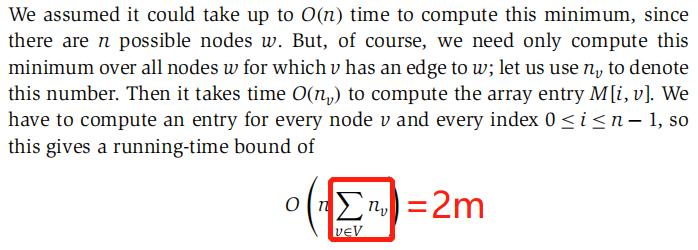
- Objective: find OPT(n-1,s)



- *Bellman Ford Solution*: O(n3)



- Improvement to the algorithm: O(mn)



1. Network flow

**The Maximum-Flow Problem**

- Problem statement: Given a flow network, find a flow of maximum possible value.

*Flow network:* a directed graph G∈(V, E)

- Features

- Each edge e has a non-negative capacity ce

- Has a single source node s∈V.

- Has a single sink node t∈V.

- Nodes other than s and t will be called internal nodes.

- Assumptions:

- no edges enter s or leave t.

- at least one edge in connected to each node.

- all capacities are integers.

*Flow:* s-t flow is a function f, f(e) represents the amount of flow carried by edge e.

- Features:

- capacity constraint: for each e∈E, 0≤f(e)≤ce.

- conservation of flow: [f in (v)=f out (v)].

- the value of a flow f: the amount of flow generated at the source

*Resident graph Gf* :

- Features

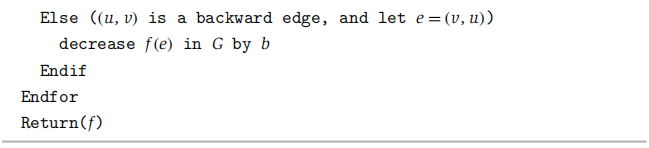
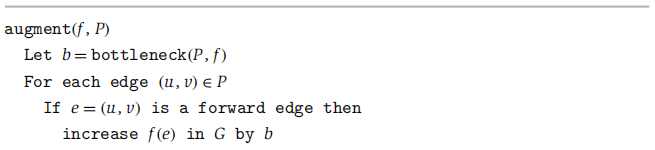
- Gf has the same set of nodes as G

- for each edge e with f(e)<ce., we include e in Gf with capacity ce.- f(e).

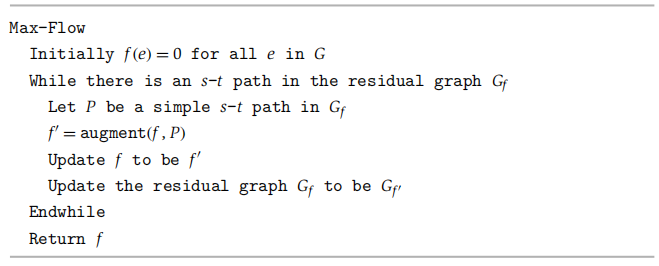
- for each edge e with f(e)>0., we include e`(opposite direction to e) in Gf with capacity f(e).

- Solution:

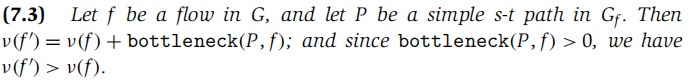
*Bottleneck(p)*: the minimum residual capacity of any edge on the path from s to t.

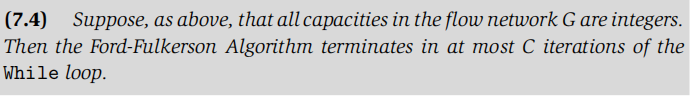


**Ford-Fulkerson Algorithm**



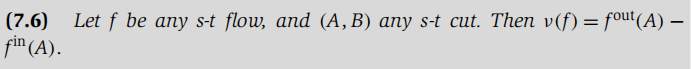
- Proof of termination:



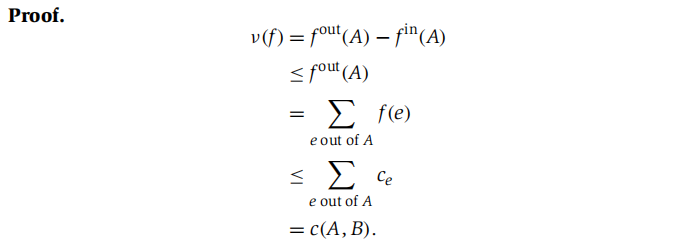


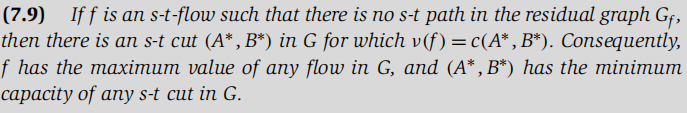
- Proof that f is a max flow

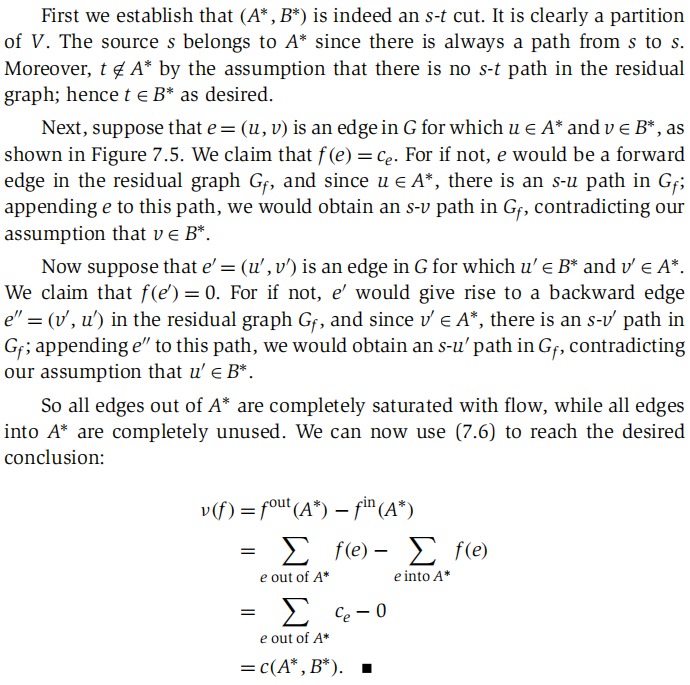
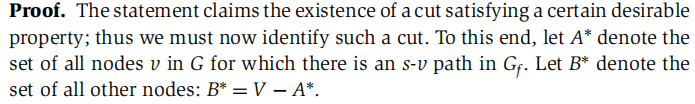
*cut:* dividing nodes in the graph into 2 sets A&B such that s∈A and t∈B.



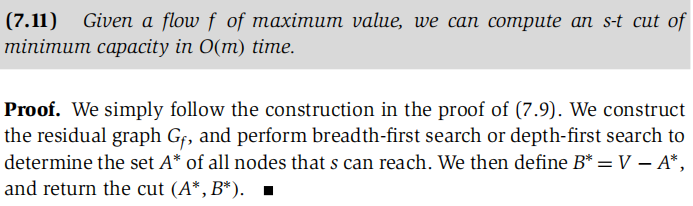
the value of every flow is upper-bounded by the capacity of every cut:

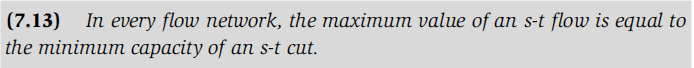






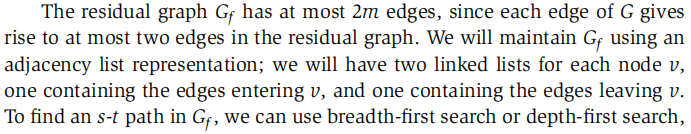
- Time complexity of compute a minimum s-t cut:

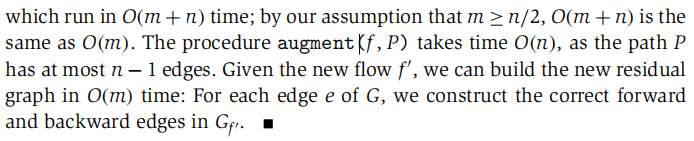




- minimum cut is not unique even if all the capacity of edges are unique.

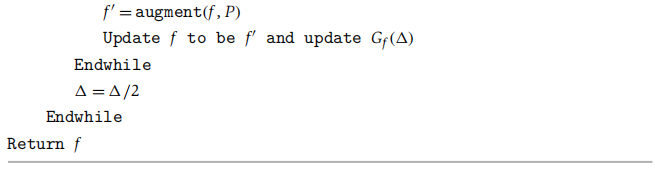
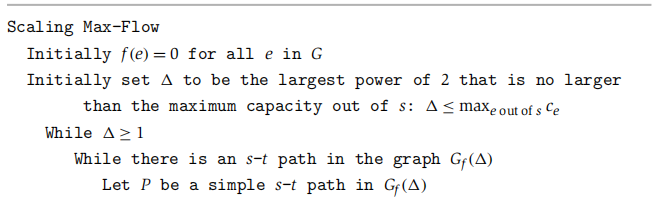
- Time complexity: O(Cm), m denote the number of edges in G.



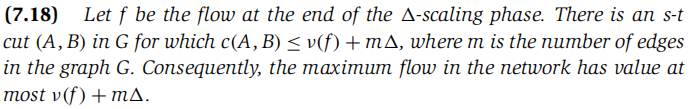
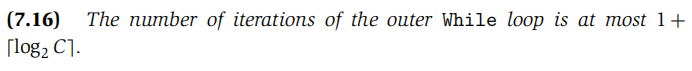


**Scaled version of Ford-Fulkerson Algorithm**

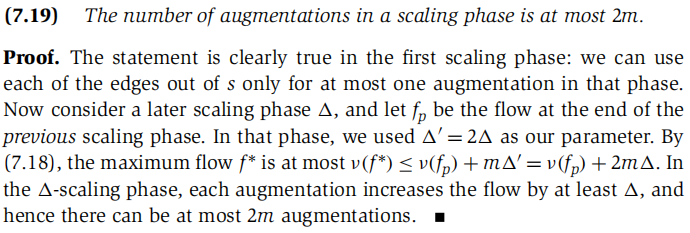
- Gf (△) : the subset of the residual graph consisting only of edges with residual capacity of at least △.

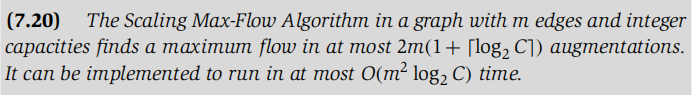


- Background



- Claim





**Strongly Polynomial Algorithms**

- an algorithm runs in strongly polynomial time if the number of operations is bounded by a polynomial in the number of integers in the input.

*Edmonds-Karp algorithm (BFS)*: same as Ford-Fulkerson (DFS), except that each augmenting path must be a shortest path with available capacity.

- Time Complexity: O(nm2)

**Bipartite Matching Problem**

*Bipartite graph G=(V, E)*: an undirected graph whose node set can be partitioned as V=X∪Y with property that every edge e∈E has one end in X and the other in Y.

*Matching M in G*: a subset of the edges M⊆E such that each node appears in at most one edge in M.

- Problem statement: Find a matching M of largest possible size in G.

- General Plan:

Design a flow network G` that will have a flow value v(f)=k if there is a matching of size k in G. Moreover, flow f in G` should identify the matching M in G.

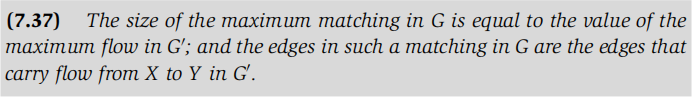
- Solution:

Run Max Flow on G`. Edges carrying flow between sets X & Y will correspond to our max size matching in G.

- Proof: If we have a matching of size k in G, we can find an s-t flow f of value k in G` and its converse statement.

Consider the set M` of edges of the form (x,y) on which the flow value is 1:





- Time Complexity : O(mn), n = |X| = |Y|, m is the number of edges of G

**Edge-disjoint Path Problem**

*edge-disjoint*: a set of paths if their edge sets are disjoint.

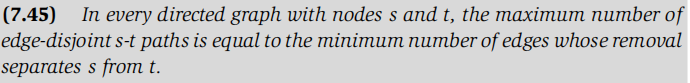
- Problem Statement:

Find max number of edge-disjoint s-t paths in a directed graph G with s&t∈V.

- General Plan as bipartite matching problem

- Solution: Run max flow in G` and v(f) = the max number of edge-disjoint s-t path, s will identify edges on their paths.

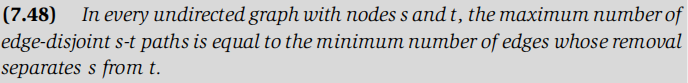
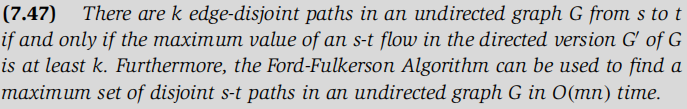
- Proof: If there are k edge-disjoint paths in a directed graph G from s to t, then the value of the maximum s-t flow in G is at least k.



- Time Complexity : O(mn)

**Extension: In undirected graph**

Replace each undirected edge in G by two directed edges, and there always exists a maximum flow in any network that uses at most one out of each pair of oppositely directed edges.



**Extension: Node-disjoint Path Problem**

Turn each node into an edge so that we can apply a capacity limit (c=1) to it.

**Circulation**

*A circulation with demand{dv}:* a function f,that assigns non-negative real number to each edge.

- Features:

- sink: if dv > 0, node v has demand of dv for flow.

- source: if dv < 0, node v has a supply of |dv| for flow.

- capacity constraint: for each e∈E, 0≤f(e)≤ce.

- Demand condition: for each node v∈V，dv = f in (v)-f out (v).



- Solution:

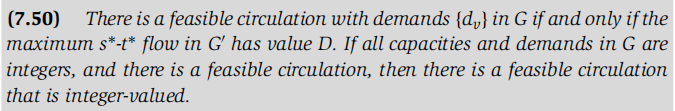
Construct G`: There are a set S of sources generating flow and a set T of sinks that can absorb flow. Attach a s\* to each node in S and a t\* to each node in T. Then, add edges with capacity |dv|. Finally, run max flow on G` and v(f) = X.

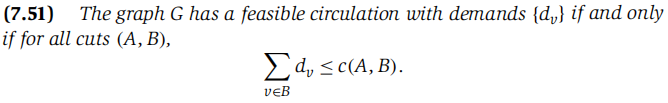
- Fact: D is the total demand value

- X<D: no feasible circulation in G

- X>D: not possible

- X<D: feasible circulation in G





**Extension: Circulation with Lower Bounds**

*Lower bound le* : the flow value on e must be at least le

- Solution: Find feasible circulation (if it exists) in two passes:

1. find f0 to satisfy all le ’s
2. Use remaining capacity of the network to find a feasible circulation f1(if it exists)

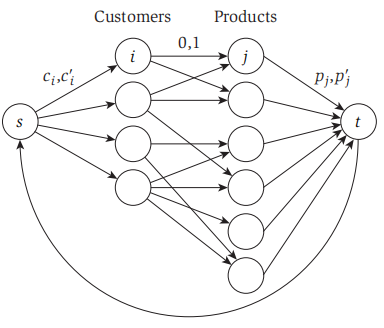
- Construct G`: the capacity of edge e is ce-le, the demand of node v is

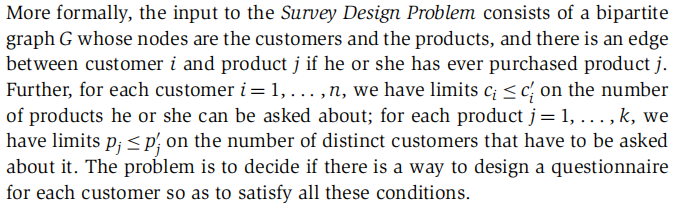
1. Combine the two flows: f = f0 +f1

**Survey Design Problem**

- Input:

1. Information on who purchased which products
2. Maximum and minimum number of questions to send to customer i
3. Maximum and minimum number of questions to ask about product j





**Min Flow Problem with lower bounds**

- Input:

A directed graph G with a source and a sink, each edge has no capacity but le

- Solution:

1. Assign the sum of le capacities to all edges and find a feasible flow f.
2. Construct G` where all the edges have capacity = fe - le
3. Find maximum flow from s to t
4. Min flow = f - f`