CSC5160: Combinatorial Optimization and Approximation Algorithms
Topic: Maximum Flow **Date:** 25/01/2008
Lecturer: Lap Chi Lau **Scribe:** Lam Chi Ho

In this lecture, we focus on the Maximum Flow problem, which is to send as much data as possible from a source to a destination through a network. This problem is polynomial time solvable, and is the most well-studied problem in combinatorial optimization. First, we will show the problem formulation. Then, we will discuss a general algorithm using augmenting path, and some polynomial time implementations of this general algorithm to efficiently solve the problem. Also, we will prove the celebrated max-flow min-cut theorem, which shows that the optimal value of the Maximum Flow problem is equal to the optimal value of the Minimum Cut problem. Finally, we will see different applications of the Maximum Flow problem.

6.1 Maximum Flow Problem: Problem Formulation

Given a directed graph $G = (V,E)$ with a *source* node s and a *sink* node t where $s, t \in V$, the goal of the Maximum Flow problem is to send as much information from s to t. The following is a more formal definition of the problem.

6.1.1 Flow Network, Flows, Value of Flow

Definition 6.1.1 *(Flow Network) A directed graph where each edge e is associated with a nonnegative capacity c_e. We also make the following assumptions:*

- *No edge enters the soruce s and no edge leaves the sink t.*
- *All edge capacties are integers.*

Definition 6.1.2 *(Flow) An* s-t flow *is a function f that maps each edge e to a nonnegative rational number, which satisfies the following two properties:*

i. *(Capacity conditions) For each $e \in E$, $0 \leq f(e) \leq c_e$. The amount of flow is a non-negative integer and bounded by the edge capacity.*

ii. *(Conservation conditions) For the node v other than source s and sink t, the sum of the flow value $f(e)$ entering node v is equal to the sum of flow values over all edges leaving node v. i.e. For each $v \in V/\{s,t\}$, we have*

$$\sum_{e \in \delta^{in}(v)} f(e) = \sum_{e \in \delta^{out}(v)} f(e)$$

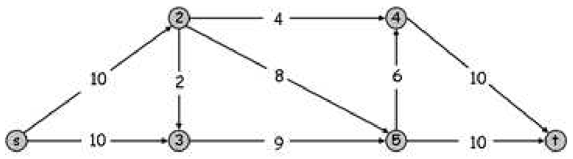
Definition 6.1.3 *(Value of Flow)For every flow f, denoted $f(v)$, is defined to be the amount of flow generated at the source, we have*

$$v(f) = \textstyle\sum_{e \in \delta^{out}(s)} f(e)$$

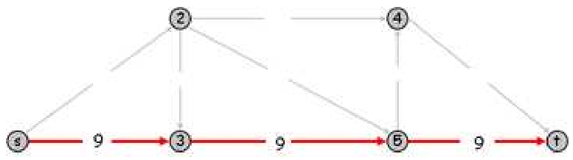
In the Maximum Flow problem, we are going to find the maximum value of $v(f)$, and also the flow f which attains $v(f)$.

6.1.2 Flow Decomposition

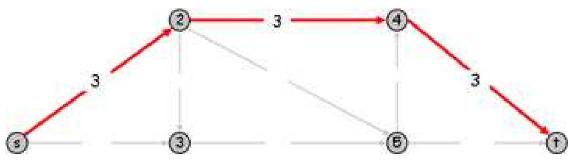
One can also think of flow as a union of flow paths. In fact, given any flow, it can be decomposed into at most m flow paths, where m is the number of edges. See Figure 6.1.1 for an example, G consists of several paths with different flow values. This flow decomposition always exists; we leave the (simple inductive) proof to the reader.



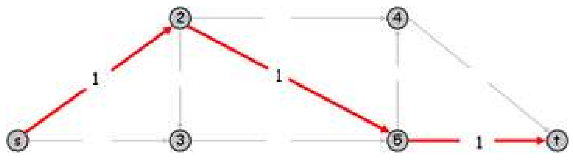
(a) Original Graph G



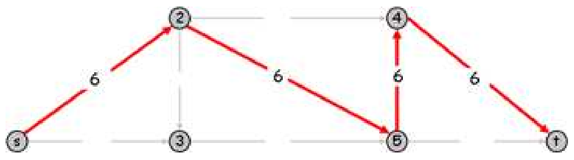
(b) Path with flow value 9



(c) Path with flow value 3



(d) Path with flow value 1



(e) Path with flow value 6

Figure 6.1.1: The flow consists of a set of flow path.

6.1.3 Upper Bounds

Before talking about the algorithm for solving Maximum Flow problems, let us consider an upper bound of the maximum flow value.

6.1.3.1 Source

One upper bound of the maximum s-t flow value is the sum of capactiy of the edges that are connected to source s, i.e, $\sum_{e\in\delta^{out}(s)} c_e$. However, this is not always tight. Next, we develop the notion of an s-t *cut* to give a more general upper bound for the maximum flow value.

6.1.3.2 Cut

We divide the directed graph G into 2 sets A and B, so that $s \in A$ and $t \in B$. Therefore, any flow from s to t must cross some edges from A to B and the capacity of this cut (C) is defined as the sum of the capacity of the edges out of A (i.e. $C = \sum_{e \in \delta^{out}(A)} c_e$) and this provides an upper bound of the flow travelling from A to B

Lemma 6.1.4 *The maximum s-t flow is at most the capacity of an s-t cut.*

Proof: Consider a set A such that $s \in A$ and $t \notin A$. By definition $v(f) = f(\delta^{out}(s))$, and by assumption $f(\delta^{in}(s)) = 0$ (there is no entering edges to s), so we can write $v(f) = f(\delta^{out}(s)) - f(\delta^{in}(s))$. Also, by the flow conservation conditions, we have $f(\delta^{out}(v)) - f(\delta^{in}(v)) = 0$ for every $v \in A/\{s\}$. Then we can write $v(f) = \sum_{v \in A}(f(\delta^{out}(v)) - f(\delta^{in}(v)))$. Thus,

$$\begin{aligned} v(f) &= f(\delta^{out}(s))) \\ &= \sum_{v \in A}(f(\delta^{out}(v)) - f(\delta^{in}(v))) \\ &= f(\delta^{out}(A) - f(\delta^{in}(A))) \\ &\leq f(\delta^{out}(A)) \\ &\leq c(\delta^{out}(A)) \end{aligned}$$

The last equality holds because every edge with both endpoints in A is cancelled out. ∎

6.2 Maximum Flow

We are now interested in finding an algorithm to solve the maximum flow problem. First we will study Greedy method.

6.2.1 Greedy Method

The first natural approach for the maximum flow problem that can think of is Greedy Method. Suppose the flow value $f(e)$ start with 0 for all edge $e \in E$, we find an s-t path P where each edge has $f(e) < c(e)$. We repeat the procedure until we get stuck.

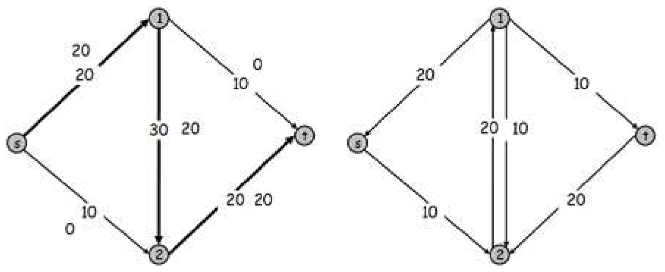
Does the Greedy method provides the maximum flow value? The answer is NO. In Figure 2(a), we augment flow along path P e.g.$[(s,1),(1,2),(2,t)]$ and the flow value increases to 20. However, it is possible to construct a flow of value 30. The problem is that there is no s-t path which we can push flow without exceeding the edge capactiy.

To find a maximum flow, we want to push a flow of value 10 along $(s,2)$, however, there will be too much flow going into 2. So we undo 10 units of flow on $(1,2)$ to restore the conservation condition. As show in Figure 2(b), we now have a valid flow and the flow value is 30 which is the maximum flow value of this graph. To have a general way to provide searching for forward-backward operation, we construct the so called "residual graph".

6.2.2 Residual Graph

Definition 6.2.1 *Given a directed graph G with flow $f(e)$, edge $e = (u, v) \in E$ and capactiy $c(e)$, as shown in Figure 2(b). The* Residual Graph *$G_f = (V, E_f)$ is defined as follows:*

- *The node set of G and G^f are the same.*
- *Forward edges $e = (u, v)$ with residual capacity $c_f(e) = c(e) - f(e)$.*
- *Backward edges $e^R = (v, u)$ with residual capacity $c_f(e) = f(e)$.*



(a) Greedy Method only provides flows of 20

(b) Residual Graph

Figure 6.2.2: Greedy Method and Residual Graph

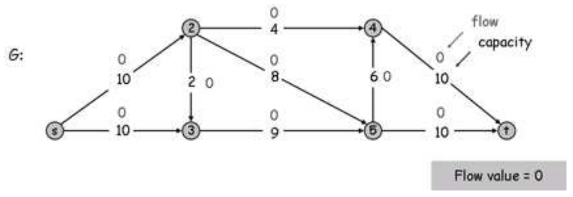
6.2.3 Augmenting path

With the residual graph, we can now search for an augmenting path that increases the flow. To find an augmenting path, we first select an *s*-*t* path P and find the *bottleneck capacity* b of P. For each $e \in P$, we increase the flow $f(e)$ of G by b if e is forward edge; otherwise, we decrease the flow $f(e)$ by b.

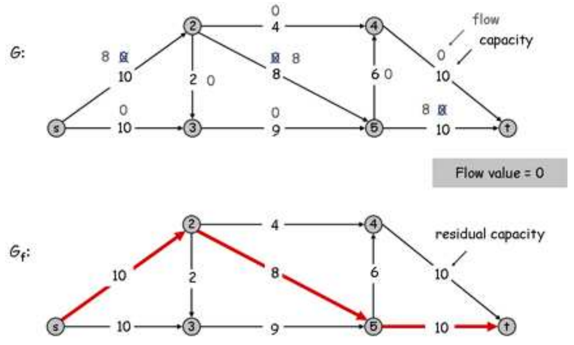
6.3 Ford-Fulkerson Algorithm

Suppose we have a directed graph $G = (V, E)$, we create the residual graph $G_f = (V, E')$ and start from an empty flow f. We find an *s*-*t* path P in the residual graph G_f, and then we update f along P and update the residual graph G_f until there is no *s*-*t* path in residual grpah G_f.

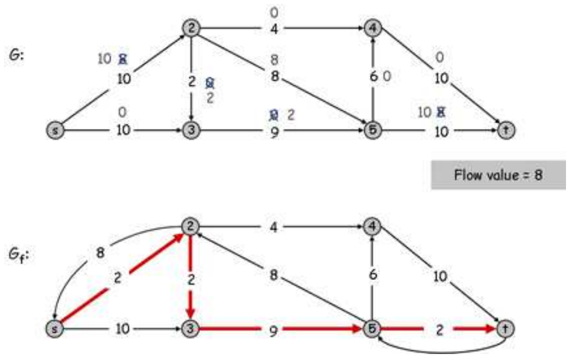
Example: Suppose their is a graph G, as Figure 3(a). In Figure 3(b), we create a residual graph G_f and find an augmenting path $P_1 = [(s, 2), (2, 5), (5, t)]$, the flow value now is 8 and then update the flow in G. In Figure 3(c), the residual graph G_f is reconstructed and we find an augmenting path $P_2 = [(s, 2), (2, 3), (3, 5), (5, t)]$ with flow value 2. Finding other augmenting path (P_3-P_5) continuely (Figure 3(d)-3(f)) until there is no augmenting path left(Figure 3(g)). Finally, we get the maximum flow value 19. ■



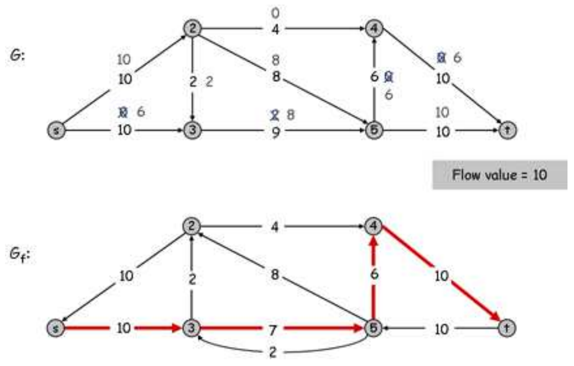
(a) Original Graph



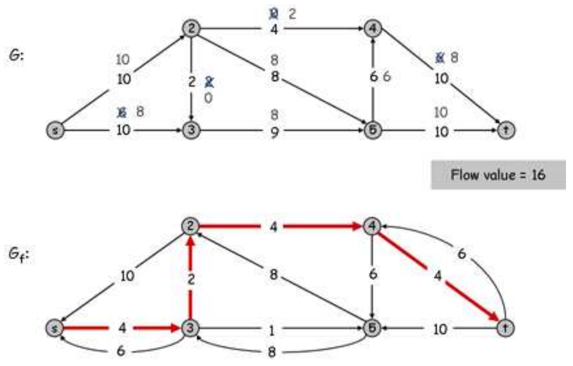
(b) Choose a path P_1 with value 8



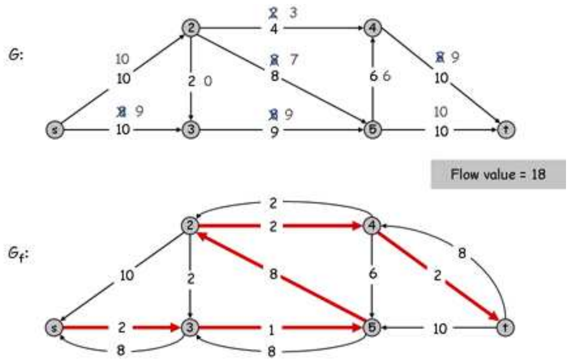
(c) Choose a path P_2 with value 2



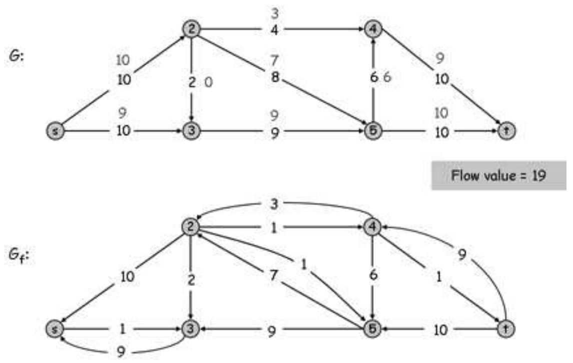
(d) Choose a path P_3 with value 6



(e) Choose a path P_4 with value 2



(f) Choose a path P_5 with value 1



(g) No augmenting path left

Figure 6.3.3: Ford-Fulkerson Algorithm

Theorem 6.3.1 *(Max-Flow Min-Cut Theorem) The value of the max flow is equal to the value of the min cut.*

Proof: Let the current flow be f and G_f be the residual graph. Suppose there is no augmenting path in G, which means that there is no directed path in G_f from s to t. Consider the set X of vertices reachable from s. So $t \notin X$. For an edge $e = (u, v)$ with $u \notin X$ and $v \in X$, we must have $f(e) = 0$ since the edge $e^R = (v, u)$ does not exist in the residual graph G_f. For an edge $e = (u, v)$ with $u \in X$ and $v \notin X$, we must have $f(e) = c(e)$ since the edge e does not exist in the residual graph G_f. Hence,

$$\begin{aligned} v(f) &= \sum_{e \in \delta^{out}(X)} f(e) - \sum_{e \in \delta^{in}(X)} f(e) \\ &= \sum_{e \in \delta^{out}(X)} c(e) \\ &= c(\delta^{out}(X)) \end{aligned}$$

So we find a flow f which achieves the full capacity of the edges going out X. Thus the flow f is maximum and the cut $\delta^{out}(X)$ is minimum. ■

Fact 6.3.2 *Every flow value $f(e)$ and every residual capacities $c_f(e)$ remains an integer throughout the algorithm.*

(Running Time) Suppose there is a network flow $G = (V, E)$, with flow capactiy $c(e)$ on each edge $e \in E$ and $1 \le c(e) \le C$. Since the flow value will be increased by 1 for each augmentation, there will be at most nC iterations to find the maximum flow value. Also, to find an s-t path takes $O(m)$ time, so the total running time would be $O(nmC)$.

Theorem 6.3.3 *(Integrality theorem)If every edge has integer capacity, then there is a flow of integer value.*

6.4 Efficient Implementations

As the running time of Ford-Fulkerson Algorithm depends on the capacity, the bound of running time may be very large when C is large. So we have the following ways to help us to choose a better augmenting path for speeding up the algorithm.

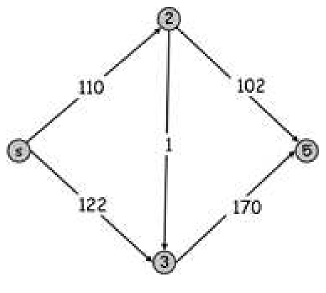
6.4.1 Capacity Scaling

First, we target to choose the path with the largest bottleneck capacity. We define a *scaling parameter* Δ, and find the paths whose capacity is at least 2^{Δ}.

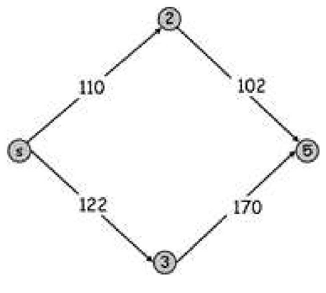
Let $G_f(\Delta)$ be the subgraph of the residual graph consisting of edges with capacity at least 2^{Δ}. As shown in Figure 6.4.4, 4(a) is the original graph G and 4(b) is the updated graph G_f consisting of edges with capacity at least 2^{Δ}. When the procedure is stuck, we divide Δ by 2 and repeat the algorithm until the algorithm terminates and we will get the maximum $v(f)$.

Proof: (Correctness) By Fact 6.3.2, $G_f(\Delta) = G_f$ when $\Delta = 0$ and there are no augmenting paths. ■

(Running Time) The *scaling parameter* Δ is updated for $1+\lceil log_2 C\rceil$ times; this bounds the number of iterations. After each iteration, there are no more augmenting paths with capacity 2^Δ. Consider the set X of vertices reachable from s when there is no more augmenting path in $G_f(\Delta)$. So each edge can send at most 2^Δ units of flow out of X. Since there are at most m edges, the capacity of this cut is at most $m2^\Delta$. In the next iteration, each augmenting path is of capacity at least $2^{\Delta-1}$, and so there will be at most $2m$ augmentations in each iteration. Hence, the number of augmentations in the *Scaling Max-Flow Algorithm* is at most $O(m^2 log_2 C)$.



(a) G_f



(b) $G_f(\Delta)$ and capacity $\geq 2^\Delta$, $\Delta = 6$

Figure 6.4.4: Capacity Scaling

6.4.2 Finding a shortest s-t path

Another method is to augment the path by the shortest path algorithm. In each iteration, we construct a layered network by BFS and then find a maximal set of shortest augmenting paths by DFS. So, each shortest augmenting path has at least one edge with its direction reversed. Hence the length of a shorest augmenting path increases after each iteration. So there will be at most n iteration, and the total number of augmentations is at most $O(mn)$.

6.4.3 Faster Algorithms

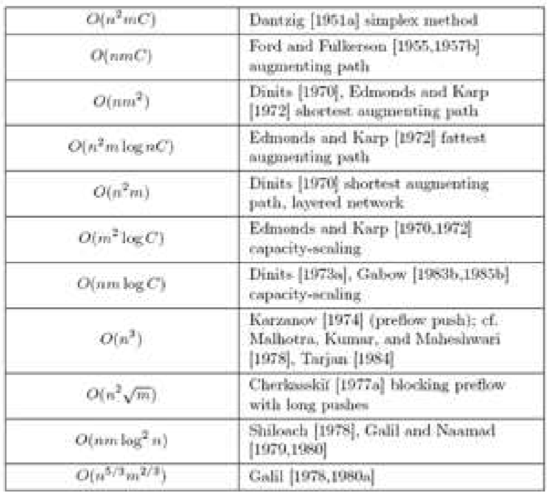
Faster algorihtm have been developed over the last century. The table below lists these results.

6.5 Applications

The Max-Flow problem can be applied to the following problems.

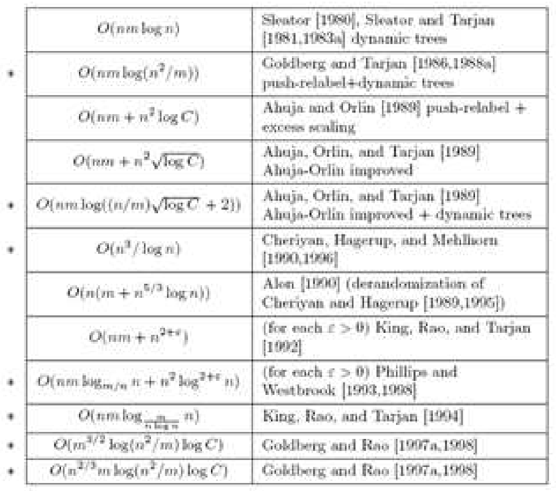
6.5.1 Multi-source Multi-sink

Suppose there are a set of sources $S = \{s_1, ..., s_k\}$ and a set of sinks $T = \{t_1, ..., t_m\}$, as shown in Figure 6.5.6, we now want to find the maximum flow from S to T. We can simply transform the problem to Max-Flow problem by adding an extra node s_e with infinity capacity edges connecting to S and a common node t_e with infinity capacity edges connecting from T.



$O(n^2mC)$	Dantzig [1951a] simplex method
$O(nmC)$	Ford and Fulkerson [1955,1957b] augmenting path
$O(nm^2)$	Dinits [1970], Edmonds and Karp [1972] shortest augmenting path
$O(n^2m\log nC)$	Edmonds and Karp [1972] fattest augmenting path
$O(n^2m)$	Dinits [1970] shortest augmenting path, layered network
$O(m^2\log C)$	Edmonds and Karp [1970,1972] capacity-scaling
$O(nm\log C)$	Dinits [1973a], Gabow [1983b,1985b] capacity-scaling
$O(n^3)$	Karzanov [1974] (preflow push); cf. Malhotra, Kumar, and Maheshwari [1978], Tarjan [1984]
$O(n^2\sqrt{m})$	Cherkasskiĭ [1977a] blocking preflow with long pushes
$O(nm\log^2 n)$	Shiloach [1978], Galil and Naamad [1979,1980]
$O(n^{5/3}m^{2/3})$	Galil [1978,1980a]

(a) Faster Algorithms



	$O(nm\log n)$	Sleator [1980], Sleator and Tarjan [1981,1983a] dynamic trees
*	$O(nm\log(n^2/m))$	Goldberg and Tarjan [1986,1988a] push-relabel+dynamic trees
	$O(nm + n^2\log C)$	Ahuja and Orlin [1989] push-relabel + excess scaling
	$O(nm + n^2\sqrt{\log C})$	Ahuja, Orlin, and Tarjan [1989] Ahuja-Orlin improved
*	$O(nm\log((n/m)\sqrt{\log C} + 2))$	Ahuja, Orlin, and Tarjan [1989] Ahuja-Orlin improved + dynamic trees
*	$O(n^3/\log n)$	Cheriyan, Hagerup, and Mehlhorn [1990,1996]
	$O(n(m + n^{5/3}\log n))$	Alon [1990] (derandomization of Cheriyan and Hagerup [1989,1995])
	$O(nm + n^{2+\epsilon})$	(for each $\epsilon > 0$) King, Rao, and Tarjan [1992]
*	$O(nm\log_{m/n} n + n^2\log^{2+\epsilon} n)$	(for each $\epsilon > 0$) Phillips and Westbrook [1993,1998]
*	$O(nm\log_{\frac{m}{n\log n}} n)$	King, Rao, and Tarjan [1994]
*	$O(m^{3/2}\log(n^2/m)\log C)$	Goldberg and Rao [1997a,1998]
*	$O(n^{2/3}m\log(n^2/m)\log C)$	Goldberg and Rao [1997a,1998]

(b) Even Faster Algorithms

Figure 6.4.5: Other Algorithms

6.5.2 Bipartite Matching

Given a bipartite graph $G = (S \cup T, E)$, we would like to find a maximum cardinality matching. Except using the algorithm studied in lecture 3, we can tranform the problem to a max flow problem.

(Max flow formulation) We can create a directed graph $G' = (S \cup T \cup \{s,t\}, E')$, and then direct all edges from S to T and assign unit capacity for each edge. After that we add source s with unit capacity edges from s to each node in S and add sink t with with unit capacity edges from each node in T to t.

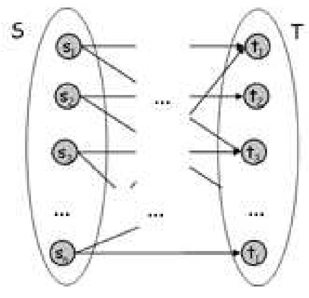
Theorem 6.5.1 *Max cardinality matching in G is equivalent to the value of max flow in G'*

Proof: ($\leq$) Given a max matching M of cardinality k in G, as shown in Figure 7(a). Consider a flow f that sends 1 unit along each of k paths in G', as shown in Figure 7(b). We can conclude that f is a flow and has cradinality k.

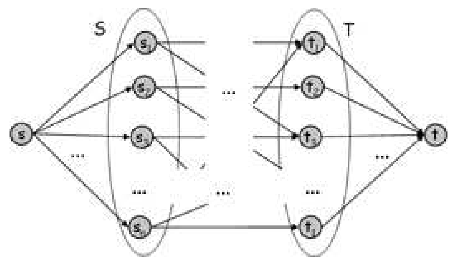
($\geq$) Let f be a max flow in G' of value k. By *Integrality theorem*, since k is integral, we can assume f is 0 or 1. We can also consider a set M consisting of edges from S to T with $f(e) = 1$, for each node in S and T participates in at most one edge in M and $|M| = k$, we can think that it is a cut $(S \cup s, T \cup t)$ ■

6.5.3 Disjoint Paths

There are 4 kinds of Disjoint Paths problems, they are *Directed Edge-Disjoint Paths problem, Directed Vertex-Disjoint Paths problem, Undirected Edge-Disjoint Paths problem and Undirected Vertex-Disjoint Paths problem.* These 4 problems actually can be transformed to Max-Flow problem



(a) Original Graph



(b) New Graph

Figure 6.5.6: Multi-source multi-sink problem

and we will discuss them one by one in the following sections.

Definition 6.5.2 *Two paths are* edge-disjoint *if they have no edge in common.*

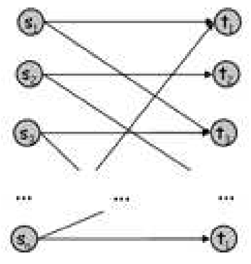
Definition 6.5.3 *Two paths are* vertex-disjoint *if they have no vertex in common.*

6.5.3.1 Directed Edge-Disjoint Paths problem

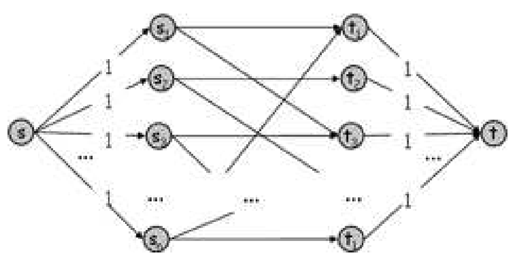
Given a graph $G = (V, E)$ and two nodes s and t, we are going to find the maximum number of edge-disjoint s-t paths.

Theorem 6.5.4 (Menger 1927) *The maximum number of edge-disjoint s-t paths is equal to the minimum number of edges whose removal disconnects t from s.*

Proof: Suppose the maximum flow is k, and hence the number of edge-disjoint s-t is k by flow decomposition. By Theorem 6.3.1, it states that there is an s-t cut (A, B) with capacity k. Let F be the set of edges that go from A to B and the capacity of each edge is 1, so $|F| =$ k. By definition of cut, removing F from G separates s and t. ■



(a) G

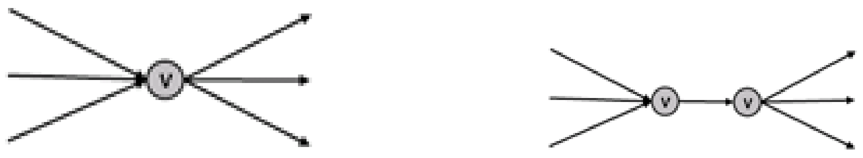


(b) G'

Figure 6.5.7: Bipartite Matching

6.5.3.2 Directed Vertex-Disjoint Paths problem

Given a directed graph $G = (V, E)$, we are going to find the maximum number of vertex-disjoint s-t paths. To transform the problem into max flow problem, we choose the vertex $v \in V$ with multiple input and multiple output and split it into 2 vertices v_1 and v_2 and add a unit capacity edge to connect (v_1, v_2). Therefore, the vertices v_1 and v_2 can only used by one edge-disjoint path. This is called the *node-splitting* technique.



(a) Vertex with multiple input and multiple output

(b) Vertex is split into 2 vertices and a unit capacity edge is added

Figure 6.5.8: Directed Vertex-Disjoint Paths

6.5.3.3 Undirected Vertex-Disjoint Paths problem

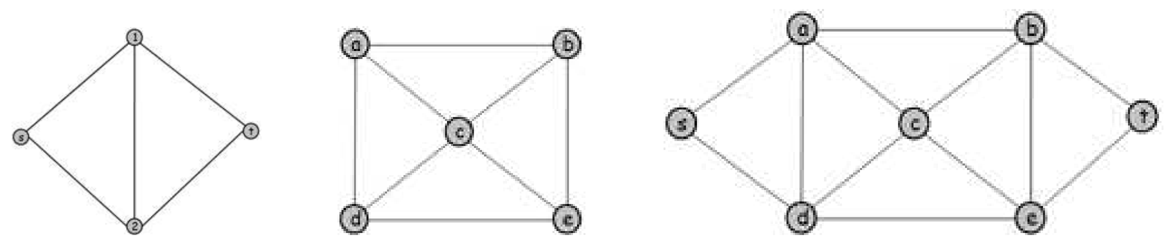
Given an undirected graph $G = (V, E)$, we are going to find the maximum number of vertex-disjoint paths. To transform the problem into directed vertex-disjoint path problem, we replace each undirected edge by two directed edges (u, v) and (v, u) (We can also delete the edges into s and out of t) and the undirected graph G is now being a directed graph $G' = (V, E')$ where $|E'| = 2|E| - 2$. Since the paths are vertex-disjoint, we choose either (u, v) or (v, u) where $u, v \in V$. This is called the *bidirecting* technique.

6.5.3.4 Undirected Edge-Disjoint Paths problem

Given an undirected graph $G = (V, E)$, we are going to find the maximum number of edge-disjoint paths. To solve his problem, we construct the line graph $L(G) = (E, F)$ where each vertex in $L(G)$ corresponds to an edge in G, and two vertices in $L(G)$ are adjacent if the corresponding edges share an endpoint. This is called the *line graph* of G; see Figure 9(b) for an example (We may add a common source and a common sink if there are muiltiple sources and mutltiple sinks). As a result, we trasnformed the problem into a *Undirected Vertex-Disjoint Path problem* and we can apply the previous algorithm to solve the problem.

6.5.4 Minimum Path Cover

Given a directed acyclic graph $G = (V, E)$, we are going to find the minimum number of paths to cover all $v \in V$. To solve this problem, we create a bipartite graph $G' = (V', E')$, as shown in Figure 10(b) where $V' = V_i^{in} \cup V_i^{out}$ and V_i^{in} represents the input of the vertex $v \in V$ and V_i^{out} represents the output of the vertex $v \in V$, $E' = (v_i^{in}, v_j^{out}) : (i, j) \in E$. We can show that this bipartite graph has a matching of size $n - k$ if and only if there are k directed paths covering all



(a) Original undriected graph G

(b) Inverted graph G'

(c) Adding a source s and a sink t to form max flow problem

Figure 6.5.9: Undirected Vertex-Disjoint Paths

the vertices in G. This transformation from a directed graph to a bipartite graph is often useful.

6.5.5 Matrix Rounding

For this application we use the following generalization of the integrality theorem, which also allows "lower bounds" on the flows.

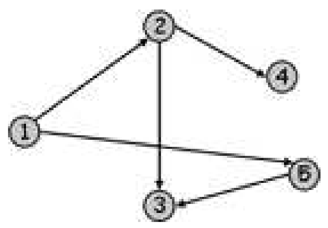
Theorem 6.5.5 *If there is a feasible flow bounded by integeral capacity ($l(e) \leq f(e) \leq c(e)$), there is an integeral flow with $l(e) \leq f(e) \leq c(e)$.*

Given a $p \times q$ matrix $D = \{d_{ij}\}$ of *rational* numbers, where the sum of row i elements $= r_i$ and the sum of column j elements $= c_j$. We would like to find a feasible rounding for each d_{ij}, r_i and c_j so that the sum of the rounded elements in each row (column) equals the rounding of the row (column) sum. (Figure 6.5.11)

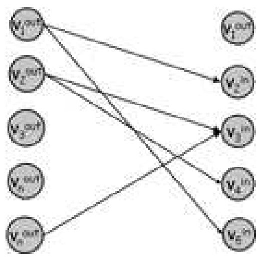
(Solution) We create a directed graph $G = (s, t, R, D, C, E)$, where s is the source, t is the sink, R is a set of vertices representing the rows, C is a set of vertices representing the columns, and D is a set of vertices representing the elements of matrix d_{ij}. The capacity of the input edge of each vertex is bounded by the floor and ceiling value of the corresponding elements. As shown in Figure 6.5.12, we set $\lfloor r_i \rfloor \leq c((s, r_i)) \leq \lceil r_i \rceil$, $\lfloor d_{ij} \rfloor \leq c(a_i, d_{ij}) \leq \lceil d_{ij} \rceil$ and $\lfloor c_i \rfloor \leq c(c_i, t) \leq \lceil c_i \rceil$ for $i, j \in [1, 3]$, e.g. $17 \leq c((s, r_i)) \leq 18$. It can be easily checked that with the initial rational entries and rational sums, there is a feasible "fractional" flow for this flow problem, i.e. the capacity constraints and the conservation constraints are satisfied. By the integrality theorem, there is also an integral solution, as desired. This is an application of the integrality theorem.

6.5.6 League Winner

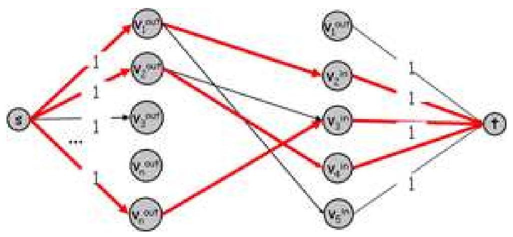
Given a Baseball match report, Figure 13(a), we want to find which teams have a chance of finishing the season with most wins. Assume we are going to check if team 3 finish with most win? This problem is actually a maximum flow problem. We create a directed graph $G = (s, t, U, V, R, E, W)$, as shown in Figure 13(b) where s, t are the source and sink of the graph, $u_{ij} \in U, v_i \in V$ are the vertices of the graph representing the game nodes and the team nodes respectively and R, E, W are



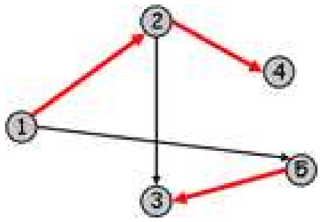
(a) Original acyclic graph G



(b) Converted to bipartite graph



(c) Max flow formulation



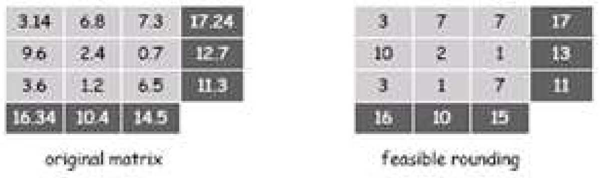
(d) Minimum path cover

Figure 6.5.10: Minimum Path Cover

the edges of the graph connected (s, U), (U, V) and (V, t) respectively. The meaning are as follows.

- $f(r_{ij})$, $r_{ij} \in R$, represents the number of games going to be played by team i and team j and the capacity $c(r_{ij})$ is the number of game remains between team i and team j;
- $f(e_{ij})$, $e_{ij} \in E$, represents the number of wins of team i for the remaining games between team i and team j, and $c(e_{ij})$ can be set to be the number of game remains between team i and team j (or can be set to be ∞);
- $f(w_i)$, $w_i \in W$, represents the total number of wins of team i and the capacity $c(w_i)$ is the maximum number of winning of v_i which will not affect our favorite team to be the champion. (First, we assume our favourite team wins all the remaining games say 80 games, then $c(w_i)$ is set to be 80 minus the current number of wins of team i.)

With this setup, the total number of wins of team i ($\sum_{ij \in U} f(e_{ij})$) is bounded by $c(w_i)$, i.e. $\sum_{ij \in U} f(e_{ij}) \le c(w_i)$ where $w_i \in W$ and $r_{ij} \in R$. So in the flow graph, we can ensure that there is no team with more wins than our favorite team. And, if all games can be scheduled to play (i.e. $f(r_{ij}) = c(r_{ij})$ for all i, j) which satisfy the capacity constraints, then there is a feasible schedule with our favourite team be the champion. To check whether all the games can be scheduled, we just need to whether the maximum flow from the source to the sink is equal to the total number of games remains.



3.14	6.8	7.3	17.24
9.6	2.4	0.7	12.7
3.6	1.2	6.5	11.3
16.34	10.4	14.5	

original matrix

3	7	7	17
10	2	1	13
3	1	7	11
16	10	15	

feasible rounding

Figure 6.5.11: Matrix D

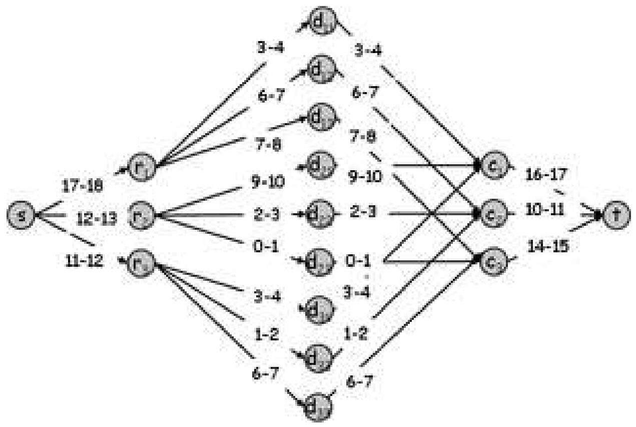
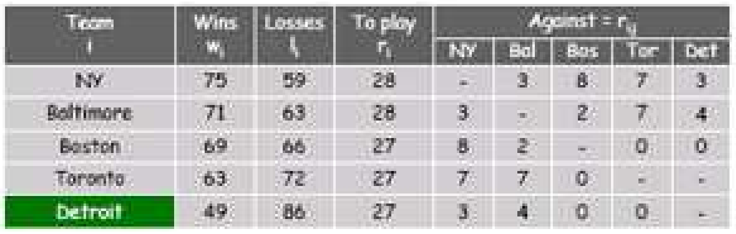


Figure 6.5.12: Max flow formulated graph

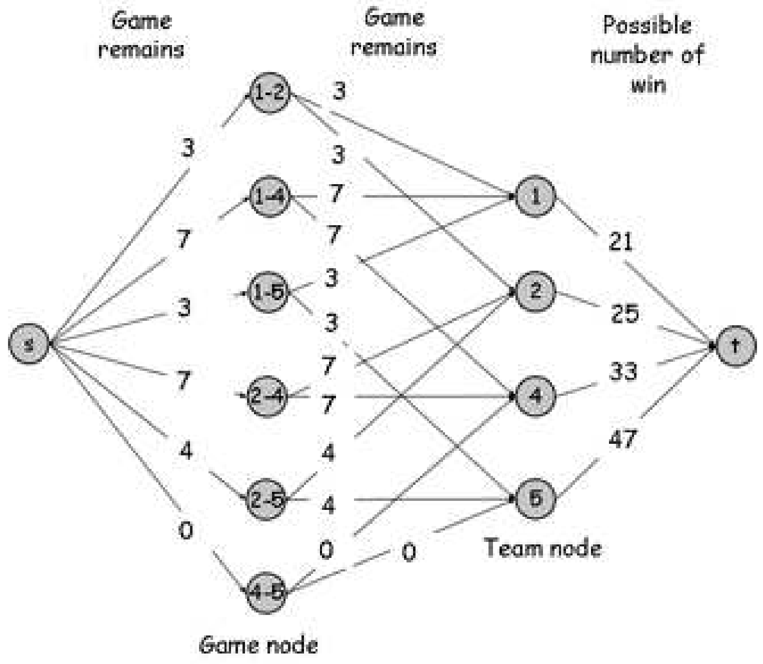
References

[1] J. Kleinberg, Eva Tardos, *Algorithm Design*, Addison Wesley, 2006.



Team i	Wins w_i	Losses l_i	To play r_i	Against = r_{ij}				
				NY	Bal	Bos	Tor	Det
NY	75	59	28	-	3	8	7	3
Baltimore	71	63	28	3	-	2	7	4
Boston	69	66	27	8	2	-	0	0
Toronto	63	72	27	7	7	0	-	-
Detroit	49	86	27	3	4	0	0	-

(a) Baseball Match Report; note that this is only a partial report.



(b) Flow formulation

Figure 6.5.13: League Winner Problem. Suppose we want to know if Boston can still win the leaque. First assume Boston wins all the remaining games, for a total of 96 games. For Boston to be champion, New York can not win more than 96-75=21 games, and this is written as the capacity of the edge from the vertex representing New York to the sink, and similarly for other teams. To determine if Boston can still win the league, we run the maximum flow algorithm and see if all the games (the total capacity going out from the source) can be sent to the sink; a feasible flow gives us feasible results for the remaining games for Boston to win the league. Note again that this is only part of the graph, since the report is only a partial report.