CS 170 Section 8
Max Flow, Duality, MT2

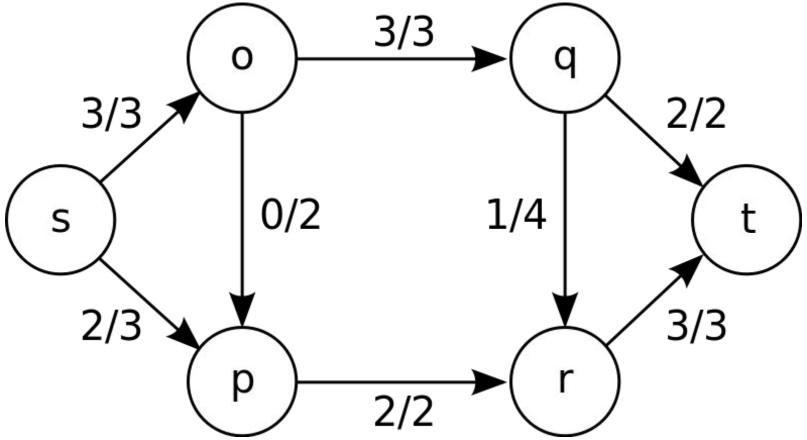
Owen Jow | owenjow@berkeley.edu

Agenda

- Max flow
- MT2
    - Duality
    - Zero-sum games
    - Bipartite matching
    - Dynamic programming

Max Flow

Max Flow



Given:

- directed graph G = (V, E)
- a source node s ∈ V
- a sink node t ∈ V
- capacities c(u, v) > 0 on the edges

We are sending something from s to t, and our task is to determine a flow (i.e. an f(u, v) for every edge).
We want to maximize the flow sent from s, subject to the constraints that

- **capacities are not violated:** 0 ≤ f(u, v) ≤ c(u, v) for each edge
- **"flow in" equals "flow out":** amount of flow entering a node (≠ s, t) equals amount of flow coming out

Solving Max Flow

- Note: this is just a linear program. **Max flow reduces to linear programming.**
- (We can formulate it as a linear program and then just solve that.)
- This means max flow can be solved using simplex (because simplex solves LPs).

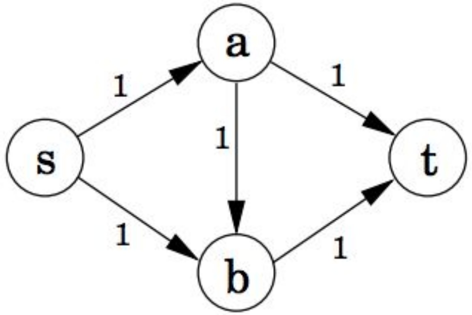
However, simplex is too inefficient to be used in practice.[1] Instead, we typically use an algorithm called **Ford-Fulkerson**.[2] The intuition for Ford-Fulkerson is based on the behavior of simplex.

[1] *Simplex iterates over vertices*. A vertex is the intersection of n constraints (i.e. hyperplanes). There are m regular constraints and n inequality constraints, meaning m + n constraints in total. Since we have to choose n inequalities to form a vertex, there are {m + n \choose n} potential vertices. (Note that many of these are infeasible, so this is only an upper bound.) This is exponential. So we might have an exponential number of iterations which are each O(mn)!

[2] Ford-Fulkerson is the max flow algorithm described in the textbook.

Intuition

- In a greedy, simplex-like fashion, we might think to solve for the max flow by
  - starting with zero flow
  - repeatedly choosing an s-t path and increasing flow along its edges as much as possible



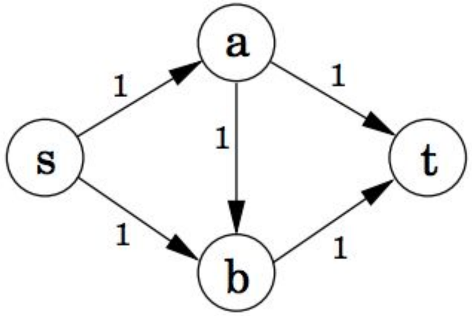
For example, given this graph we might select the paths s-a-t and s-b-t. By our naive algorithm, we fill the capacity along these paths.

This is the optimal flow!
But does the algorithm *always* work?
One example does not a proof make...

Intuition

A flow is suboptimal if

- there is an s-t path for which all of the edges are used at less than full capacity
- *we chose an incorrect s-t path which blocks off flow from some optimal path(s)*



The optimal flow is f(s, a) = 1, f(a, t) = 1, f(s, b) = 1, f(b, t) = 1, f(a, b) = 0.

But if we first choose the path s-a-b-t according to our naive algorithm, we will not find this flow.

Now, if we could cancel existing flow, we could avoid this problem.

Residual Network

- We can “cancel” existing flow by *sending flow back*
  - given an edge (u,v) for which f(u, v) > 0, we can add an edge that allows us to send f(u, v) units from v to u
  - this provides the option of negating the flow we originally sent from u to v
  - and by extension the ability to deal with problematic path choices

- This behavior is encapsulated by a **residual network**, which depends on both G and a flow f.
- The residual network of G *with respect to f* is one for which the edge (u, v) has capacity

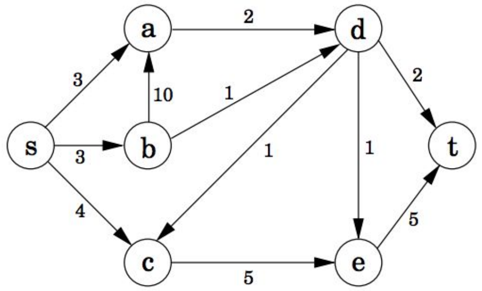
$$c^f(u, v) = c(u, v) - f(u, v) + f(v, u)$$

i.e. how much more flow can be pushed along the edge

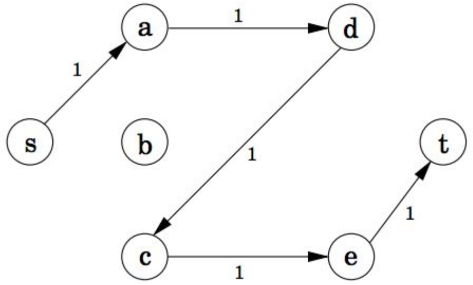
f(u, v) units of the initial capacity c(u, v) are taken by the current flow, and we get f(v, u) units of extra capacity from the ability to reverse flow from v to u

Residual Network

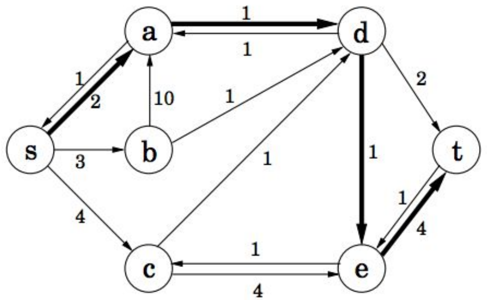
- Here is an example of a residual network. Recall that $c^f(u, v) = c(u, v) - f(u, v) + f(v, u)$.



ORIGINAL (G)



FLOW (f)



RESIDUAL (G^f)

Ford-Fulkerson

- Ford-Fulkerson is a greedy algorithm which, at each step,
  - constructs the residual graph for the current flow [1]
  - finds an s-t path with positive capacity on all edges *in the residual graph*
  - increases flow along this path by the minimum of these residual capacities
- This repeats until there is no s-t path in the residual graph with positive capacity on all edges.

We can use, e.g., BFS or DFS to find s-t paths in the residual network.[2]
If using BFS (i.e. the Edmonds-Karp variant), the runtime is $O(|V| \cdot |E| \cdot |E|)$.

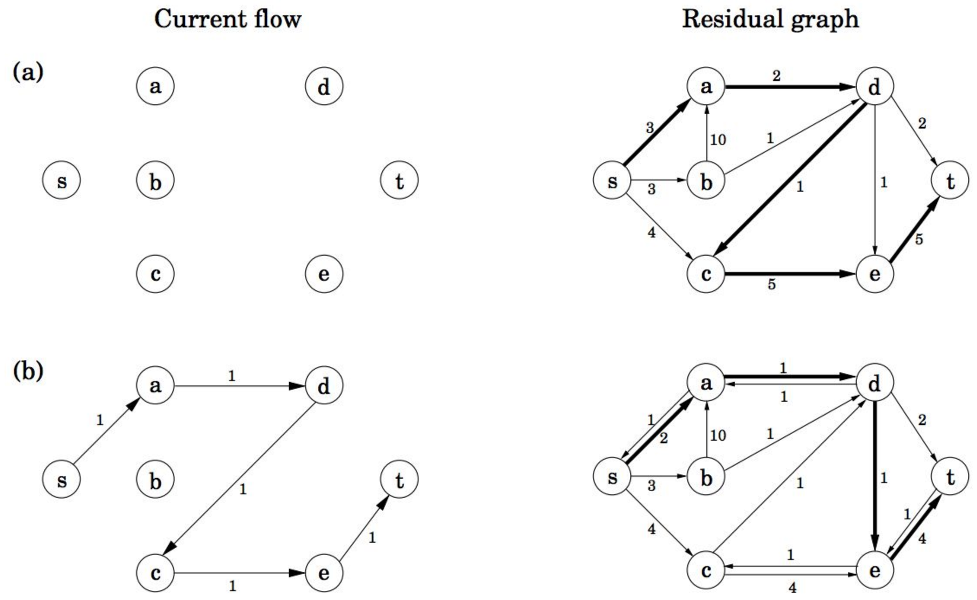
[1] Initially there is no flow, and the residual graph is the same as the original graph.
[2] If BFS is used, Ford-Fulkerson becomes the “Edmonds-Karp” algorithm.

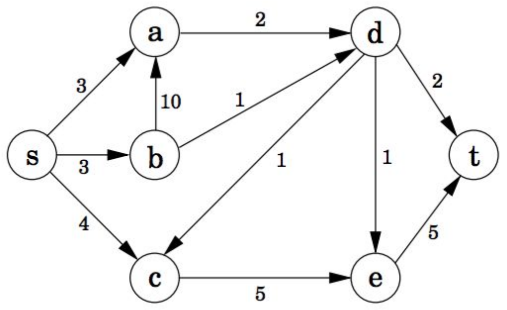
Ford-Fulkerson

- Ford-Fulkerson is just our original greedy algorithm, except we find s-t paths in *residual graphs*
- Residual graphs resolve the 2nd suboptimality possibility by allowing current flow to be canceled

Note that *every* new path out of s will increase flow! (Again, flow is defined by whatever comes out of s.)

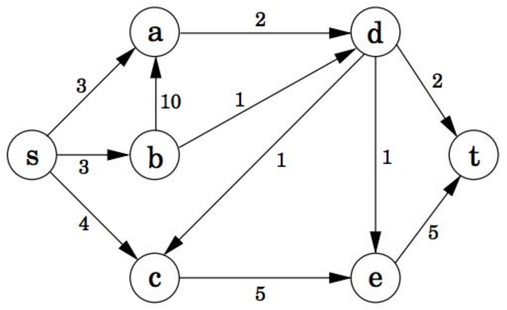
FF Walkthrough





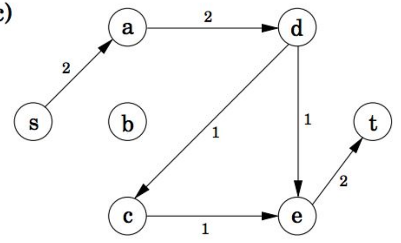
ORIGINAL (G)

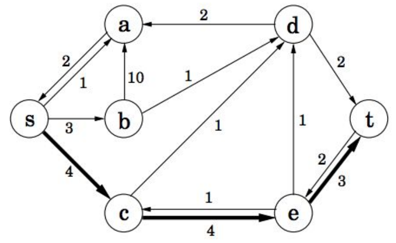
FF Walkthrough



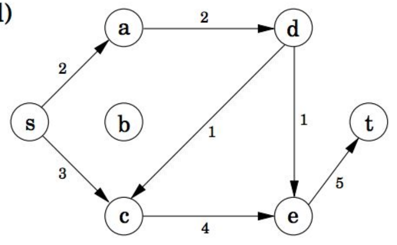
ORIGINAL (G)

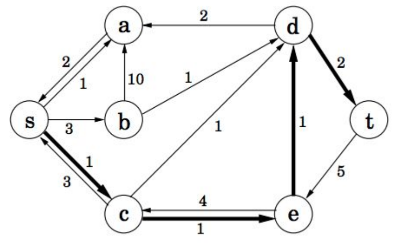
(c)





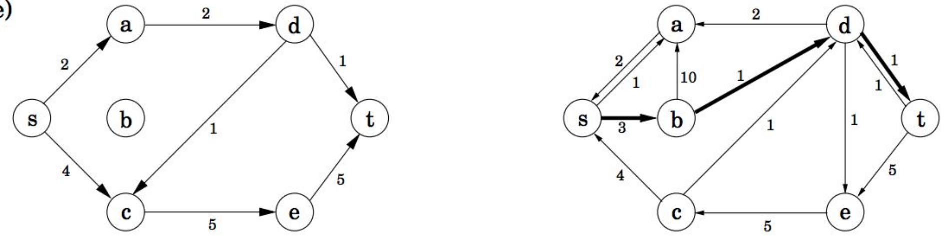
(d)



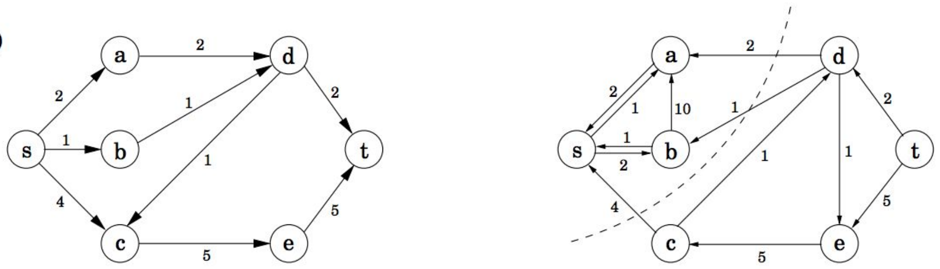


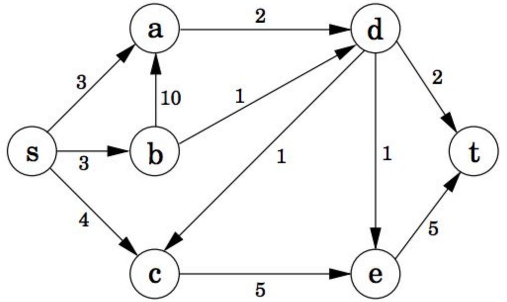
FF Walkthrough

(e)



(f)





ORIGINAL (G)

Max Flow / Min Cut Theorem

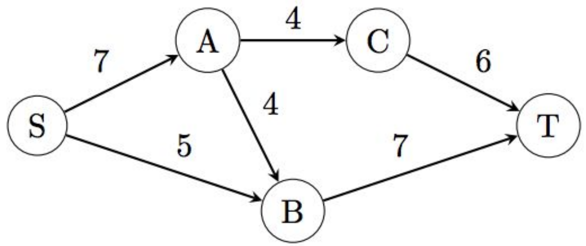
- An (s, t)-cut partitions the vertices into two disjoint groups L and R, s.t. $s \in L$ and $t \in R$
- Its **capacity** is the total capacity of all edges from L to R
    - This is an upper bound on any flow, since we must always cross this cut
- Max-flow min-cut theorem: the size of the maximum flow in a network equals the capacity of the smallest (s, t)-cut

(Finding the min cut is the dual of finding the max flow.)
Ford-Fulkerson automatically finds the minimum cut:

- L = all nodes reachable from s in G^f, and R = the rest of the nodes

Residual in Graphs

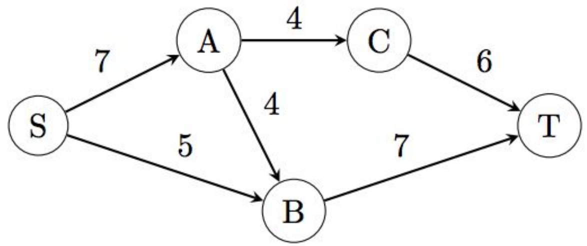
- Consider the following graph with edge capacities as shown:

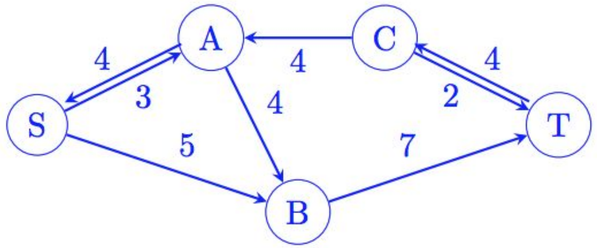


Say we push 4 units through S-A-C-T. What is the residual graph with respect to this flow?

Residual in Graphs

- Consider the following graph with edge capacities as shown:



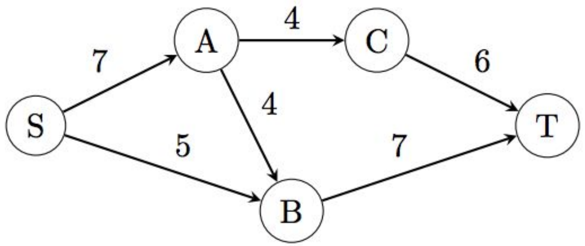


Say we push 4 units through S-A-C-T. What is the residual graph with respect to this flow?

See above.

Residual in Graphs

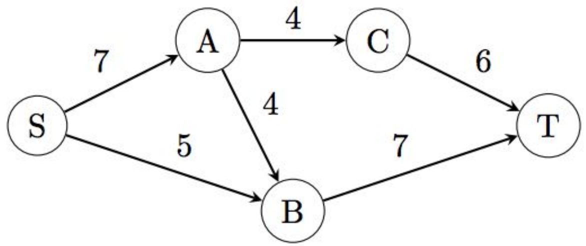
- Consider the following graph with edge capacities as shown:

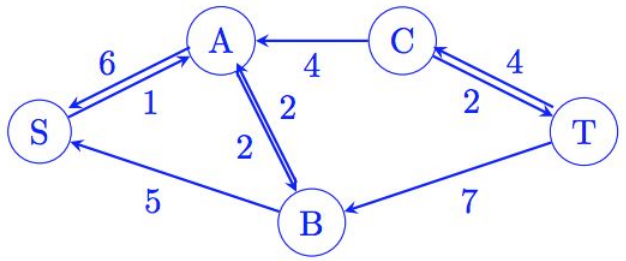


Compute a max flow of the graph. Also find a min cut. Draw the residual graph of the max flow.

Residual in Graphs

- Consider the following graph with edge capacities as shown:





Compute a max flow of the graph. Also find a min cut. Draw the residual graph of the max flow.

One max flow involves pushing 4 units through S-A-C-T, 5 units through S-B-T, and 2 units through S-A-B-T. The value is 11. The min cut is between {S, A, B} and {C, T}. There are other max flows with the same value.

A Cohort of Spies

- There are k spies traveling on a railway system G = (V, E)
- Each spy i starts at city s_i ∈ V and needs to get to a city in T ⊆ V
- At most c spies should ever pass through any given city

Find a path for each spy, or determine that the requirements cannot be met.

A Cohort of Spies

Find a path for each spy, or determine that the requirements cannot be met.

We will think of each spy as a unit of flow that we want to move from s_i to any vertex in T. We convert G into a flow network by setting the capacity of each edge to ∞, adding a sink t with an edge (t', t) for every t' ∈ T, and adding a source s with edges (s, s_i) of capacity 1. We can incorporate vertex capacities of c by splitting each vertex into two and adding an edge with capacity c between the two sub-vertices.

If the requirements can be met, Ford-Fulkerson will output a flow which meets our constraints. We can then have each spy i follow a path in the flow from s_i to any vertex in T.

If the requirements cannot be met, Ford-Fulkerson will fail to output a flow of value k.

Repairing a Flow

- Say we have found a max flow from s to t in the graph G = (V, E) with integer capacities c(· , ·).
- For a given edge (u, v), we discover that c(u, v) was supposed to be c(u, v) - 1.
- Unfortunately, our max flow uses the full capacity of that edge.

How can we recompute the max flow in O(|V| + |E|) time?

Repairing a Flow

How can we recompute the max flow in $O(|V| + |E|)$ time?

Consider the residual graph. Since there is a flow of at least 1 unit going from v to t, the residual graph must have a path from t to v (the residual graph always gives the option to "cancel" existing flows). Likewise, there must be a path from u to s.

We can find the t-v path via a DFS from t, and the u-s path via a DFS from u. We will send back 1 unit of flow through the path $t \to v$-$u \to s$. This will cause the flow through (u, v) to become $c(u, v) - 1$. Note: the flow through the entire graph has been decreased by 1, and is valid even after we decrease $c(u, v)$ by 1.

Finally, we check for an s-t path in the new residual graph to see if the flow can be increased back to what it was.
At most it can be increased by 1; this was the max flow value to begin with and the min cut capacity did not change by > 1.
Hence this is just one more DFS (or BFS), & overall the runtime is that of three depth-first searches: $O(|V| + |E|)$.

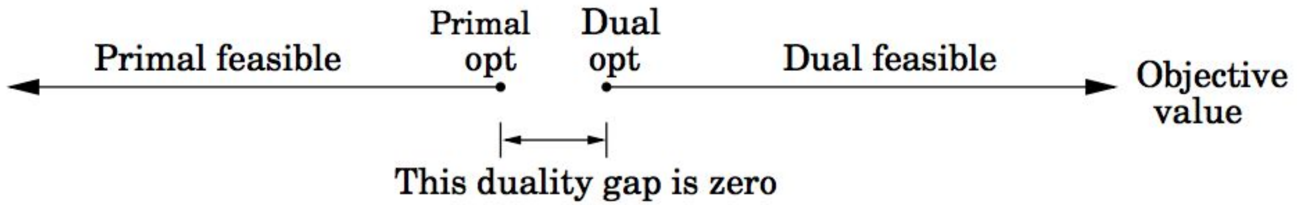
MT2

MT2

- Midterm 2 is next week
- Focus on *Algorithms* chapters 5.1 - 7.5
  - MSTs, greedy algorithms, DP, LP, max flow, duality, zero-sum games, reductions

Duality

- Every linear maximization problem has a **dual** minimization problem, where *any feasible value* of the dual LP is an upper bound on *any feasible value* of the original (**primal**) LP
- If we find a pair of primal and dual feasible values that are equal, they must both be optimal
- Duality theorem: if an LP has a bounded optimum, so does the dual, and the optimum values coincide



Zero-Sum Games

- Zero-sum game: one player’s gain is the other player’s loss
- If we formulate the linear programs, then each player’s LP is the dual of the other player’s LP
- Duality equalizes the two players’ strategies such that their expected outcomes are equal
- This allows us to uniquely define “optimal play”

Essentially, zero-sum games are an example of duality IRL.

Bipartite Matching

- Now and beyond, reductions are a big theme of the class
- Bipartite matching is an example of a problem that *reduces to max flow*
    - i.e. you can reformulate it so it can be solved as a max flow problem
    - hence if you can solve max flow, you can solve bipartite matching

Dynamic Programming

- HW6 #4 (Road Trip)
- Solution: http://cs170.org/assets/hw/hw06-sol.pdf