Estratégias Algorítmicas 2020/21

Week 11 – Graph Algorithms (SCC and MaxFlow)

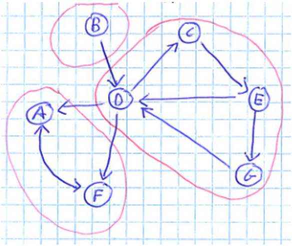
· U C ·

Universidade de Coimbra

Outline

Introduction

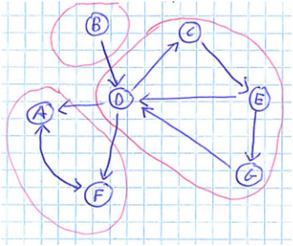
Given a directed graph $G = (V, A)$, a subgraph G' is a strongly connected component if there exists a path between a vertex and every other vertex in G' and this subgraph has maximal size.



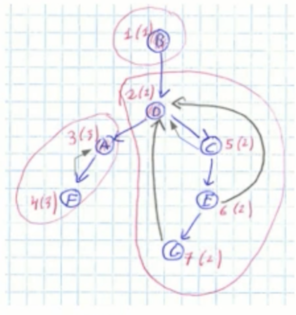
3 strongly connected components

Strongly Connected Components

A vertex v is the root of a connected component if $low[v] = dfs[v]$



A graph



DFS tree

The vertices in each strongly connected component under the root are stored in a stack. It is possible to solve it with Tarjan Algorithm in $O(|V| + |A|)$.

Strongly Connected Components

Function $Tarjan(v)$

```
low[v] = dfs[v] = t
t = t + 1
push(S, v)
for each arc (v, w) ∈ A do
    if dfs[w] has no value then
        Tarjan(w)
        low[v] = min(low[v], low[w])
    else if w ∈ S then
        low[v] = min(low[v], dfs[w])
if low[v] = dfs[v] then
    C = ∅
    repeat
        w = pop(S)
        push(C, w)
    until w = v
    push(Scc, C)
```

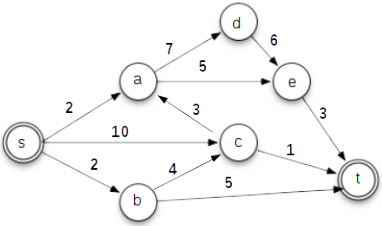
Scc collects all stacks of strongly connected components. This algorithm must be called for every unvisited vertex in the graph.

Given a directed graph $G = (V, A)$, where each arc (u, v) has a capacity $c(u, v) > 0$ (a *network*), and two vertices *source* s and *sink* t, the maximum flow problem consists of maximizing the total amount of flow from s to t subject to two constraints:

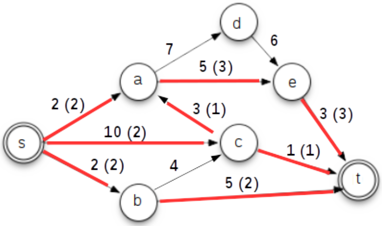
1. Flow on arc (u, v) does not exceed $c(u, v)$
2. For every vertex $v \neq s, t$, incoming flow is equal to outgoing flow.

This problem arises in many real-life situations:

- Routing as many packets as possible on a network, given a bandwidth capacity
- Sending as many trucks as possible, where roads have limits on the number of trucks per unit time



A network



A network with maximum flow of 6

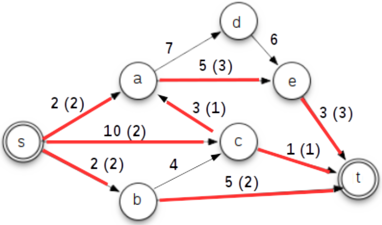
How to compute the maximum flow?

A residual network $G_R = (V, A_R)$ has the same vertices as the original network G and two different types of arcs:

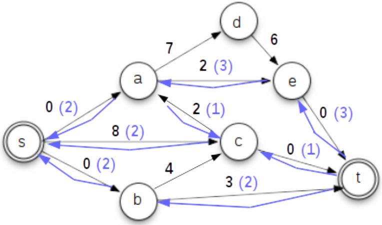
1. For each arc (u, v) with capacity $c(u, v)$ and flow $f(u, v) < c(u, v)$, there exists an arc (u, v) in the residual network with capacity $c(u, v) - f(u, v)$
2. For each arc (u, v) with capacity $c(u, v)$ and flow $f(u, v) > 0$, there exists an arc (v, u) in the residual network with capacity $f(u, v)$

An augmenting path is an path from source to sink in the residual network G^* that should send as much flow as possible.

Ford-Fulkerson Algorithm



Network flow



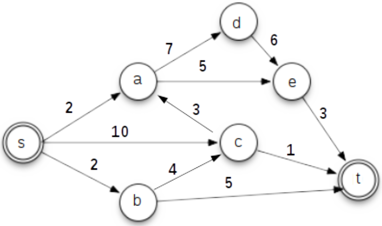
Residual network

Ford-Fulkerson Algorithm

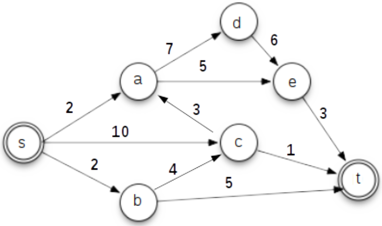
Function $FF(G_R)$

$mflow = 0$
while there is a s-t path P in G_R **do**
  $f_p = \min\{c(u,v) \mid (u,v) \in P\}$
  for each arc (u,v) in P **do**
    $c(u,v) = c(u,v) - f_p$
    $c(v,u) = c(v,u) + f_p$
    $mflow = mflow + f_p$
return $mflow$

For each arc (u,v) in G, G_R contains an arc (u,v) with capacity $c(u,v)$ and an arc (v,u) with capacity 0.

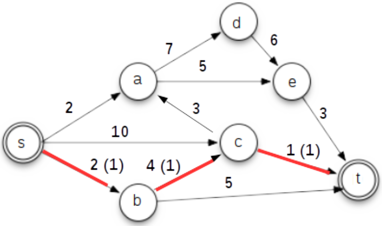


Network flow

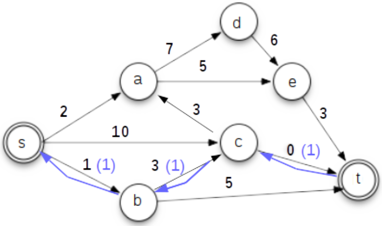


Residual network

Ford-Fulkerson Algorithm



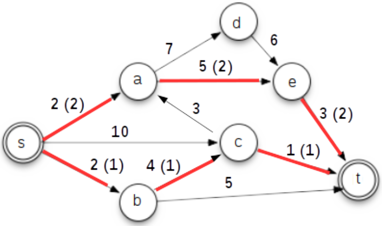
Network flow



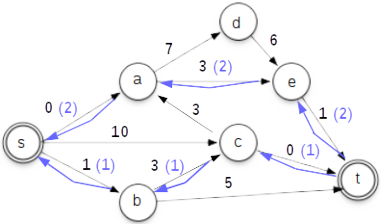
Residual network

Send 1 unit of flow through (s, b, c, t) in the residual network

Ford-Fulkerson Algorithm



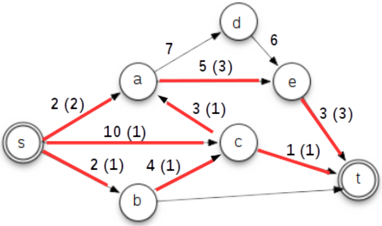
Network flow



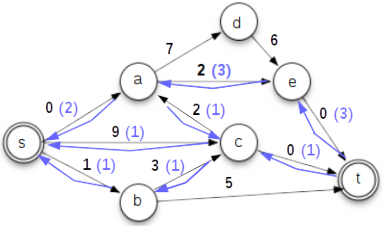
Residual network

Send 2 units of flow through (s, a, e, t) in the residual network

Ford-Fulkerson Algorithm



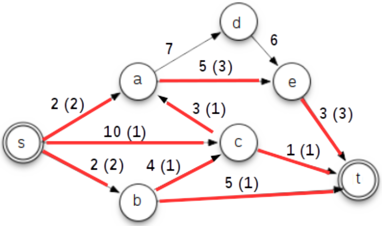
Network flow



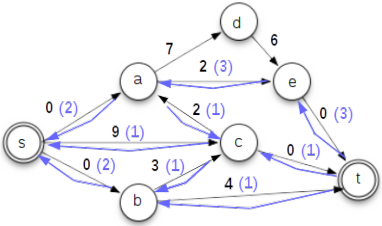
Residual network

Send 1 unit of flow through (s, c, a, e, t) in the residual network

Ford-Fulkerson Algorithm



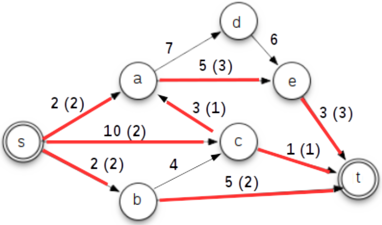
Network flow



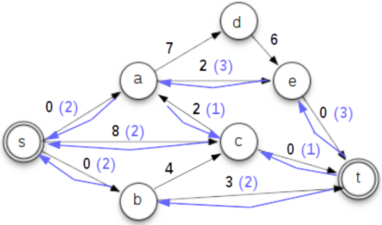
Residual network

Send 1 unit of flow through (s, b, t) in the residual network

Ford-Fulkerson Algorithm



Network flow



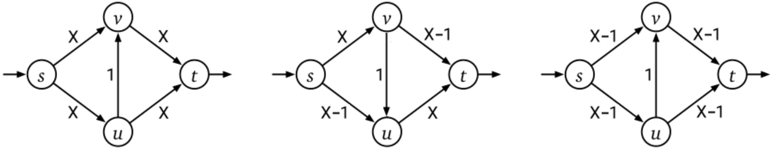
Residual network

Send 1 unit of flow through (s, c, b, t) in the residual network, which cancels the flow from c to b in the network

Edmond-Karp Algorithm

Ford-Fulkerson works under any choice of a *s*-*t* path. However, an arbitrary choice may give a bad worst case time complexity.

Consider that the *s*-*t* path passes always through vertices *u* and *v*:

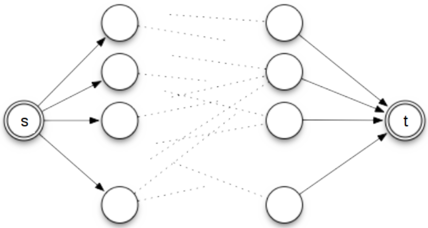


The number of iterations until it stops is twice the value of the maximum flow.

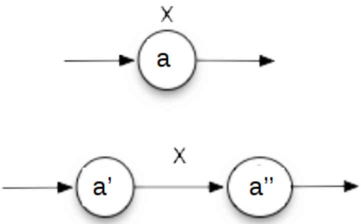
But if the *s*-*t* path with the least number of arcs is chosen (with BFS), the time complexity reduces to $O(|V| \cdot |E|^2)$ for any network (Edmond-Karp Algorithm).

Variants

Many sources and/or many sinks: Add a source vertex that conects to all sources and/or add a sink vertex to which all sinks are connected. The new arcs must have an "infinite" capacity.

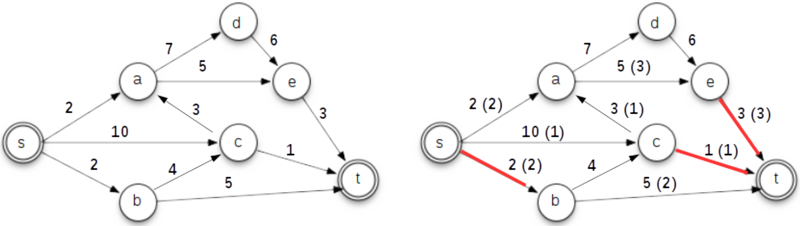


A vertex a with capacity constraint x: Split the vertex into two, a' and a'', and connect them by an arc with capacity x.



Minimum Cut Problem

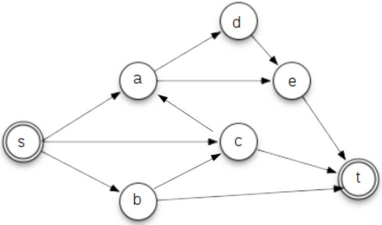
Given a directed graph $G = (V, A)$, where each arc (u, v) has a weight $w(u, v)$, and two vertices s and t, the minimum cut problem consists of finding the set of arcs to be removed (a *cut*) with the least total weight such that s and t become unreachable.

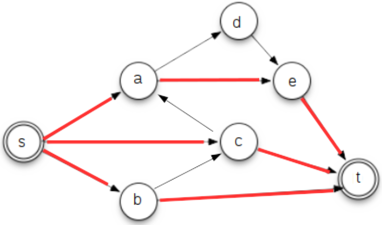


Solve a maximum flow problem where the arc weight is the arc capacity. The least total weight is equal to the maximum flow (*max-flow min-cut theorem*).

Arc-Disjoint Paths Problem

Given a directed graph $G = (V, A)$, and two vertices s and t, the arc-disjoint paths problem consists of finding the maximum number of s-t paths that do not have arcs in common.

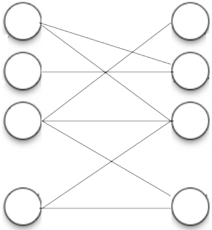




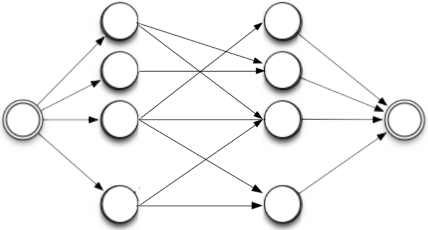
Add capacity one to each arc and solve the corresponding maximum flow problem. The number of arc-disjoint paths is equal to the maximum flow.

Maximum Bipartite Matching

Given a directed graph $G = (U, V, E)$, where U and V are two disjoint sets of vertices, and E is a set of edges, each of which connects an element of U to an element of V, the maximum bipartite matching problem consists of finding the maximum number of edges such that no two edges share a vertex (a *matching*).



A bipartite graph

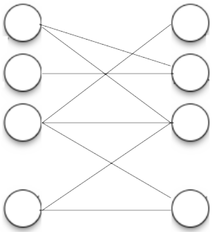


Transformed graph

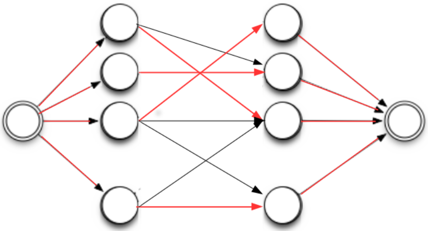
Transformed graph: add a source that connects to all vertices in U and a sink to which all vertices in V are connected. Arc capacity is equal to 1.

Maximum Bipartite Matching

Given a directed graph $G = (U, V, E)$, where U and V are two disjoint sets of vertices, and E is a set of edges, each of which connects an element of U to an element of V, the maximum bipartite matching problem consists of finding the maximum number of edges such that no two edges share a vertex (a *matching*).



A bipartite graph



Transformed graph

The number of edges in a maximal bipartite matching is equal to the maximum flow in the transformed graph