CS38
Introduction to Algorithms

Lecture 12

May 8, 2014

Outline

- Network flow
  - finishing capacity-scaling analysis
  - Edmonds-Karp, blocking-flow implementation
  - unit-capacity simple graphs
  - bipartite matching
  - edge-disjoint paths
  - assignment problem

* slides from Kevin Wayne

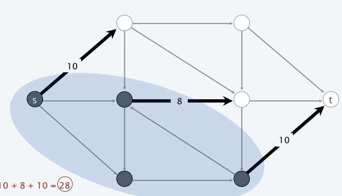
Minimum cut problem

Def. A *st*-cut (cut) is a partition (A, B) of the vertices with $s \in A$ and $t \in B$.

Def. Its capacity is the sum of the capacities of the edges from A to B.

$$cap(A, B) = \sum_{e \text{ out of } A} c(e)$$

Min-cut problem. Find a cut of minimum capacity.



capacity = 10 + 8 + 10 = 28

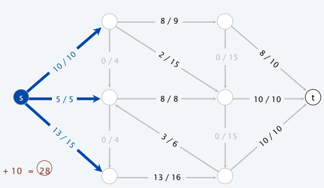
Maximum flow problem

Def. An *st*-flow (flow) f is a function that satisfies:

- For each $e \in E$: $0 \le f(e) \le c(e)$ [capacity]
- For each $v \in V - \{s, t\}$: $\sum_{e \text{ in to } v} f(e) = \sum_{e \text{ out of } v} f(e)$ [flow conservation]

Def. The value of a flow f is: $val(f) = \sum_{e \text{ out of } s} f(e)$.

Max-flow problem. Find a flow of maximum value.

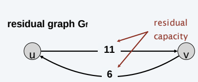


value = 8 + 10 + 10 = 28

Residual graph

Original edge: $e = (u, v) \in E$.
- Flow $f(e)$.
- Capacity $c(e)$.

Residual edge.
- "Undo" flow sent.
- $e = (u, v)$ and $e^R = (v, u)$.
- Residual capacity:



$$c_f(e) = \begin{cases} c(e) - f(e) & \text{if } e \in E \\ f(e) & \text{if } e^R \in E \end{cases}$$

Residual graph: $G_f = (V, E_f)$.
- Residual edges with positive residual capacity.
- $E_f = \{e : f(e) < c(e)\} \cup \{e^R : f(e) > 0\}$.
- Key property: f' is a flow in G_f iff $f + f'$ is a flow in G.

where flow on a reverse edge negates flow on a forward edge

Augmenting path

Def. An augmenting path is a simple $s \rightsquigarrow t$ path P in the residual graph G_f .

Def. The bottleneck capacity of an augmenting P is the minimum residual capacity of any edge in P.

Key property. Let f be a flow and let P be an augmenting path in G_f .
Then f' is a flow and $val(f') = val(f) + bottleneck(G_f, P)$.

```
AUGMENT (f, c, P)
b ← bottleneck capacity of path P.
FOREACH edge e ∈ P
   IF (e ∈ E ) f(e) ← f(e) + b.
   ELSE        f(e^R) ← f(e^R) − b.
RETURN f.
```

Ford-Fulkerson algorithm

Ford-Fulkerson augmenting path algorithm.

- Start with $f(e) = 0$ for all edge $e \in E$.
- Find an augmenting path P in the residual graph G_f.
- Augment flow along path P.
- Repeat until you get stuck.

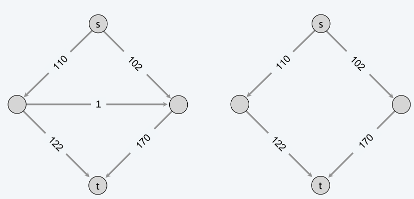
```
FORD-FULKERSON (G, s, t, c)

FOREACH edge e ∈ E : f(e) ← 0.
   G_f ← residual graph.
   WHILE (there exists an augmenting path P in G_f)
      f ← AUGMENT (f, c, P).
      Update G_f.
   RETURN f.
}
```


Capacity-scaling algorithm

Capacity-scaling algorithm

Intuition. Choose augmenting path with highest bottleneck capacity: it increases flow by max possible amount in given iteration.

- Don't worry about finding exact highest bottleneck path.
- Maintain scaling parameter Δ.
- Let $G_f(\Delta)$ be the subgraph of the residual graph consisting only of arcs with capacity $\geq \Delta$.



G_f $\quad$ $G_f(\Delta)$, $\Delta = 100$

Capacity-scaling algorithm

```
CAPACITY-SCALING(G, s, t, c)

FOREACH edge e ∈ E : f(e) ← 0.
Δ ← largest power of 2 ≤ C.

WHILE (Δ ≥ 1)
   G_f(Δ) ← Δ-residual graph.
   WHILE (there exists an augmenting path P in G_f(Δ))
      f ← AUGMENT (f, c, P).
      Update G_f(Δ).
   Δ ← Δ / 2.

RETURN f.
```


Capacity-scaling algorithm: proof of correctness

Assumption. All edge capacities are integers between 1 and C.

Integrality invariant. All flow and residual capacity values are integral.

Theorem. If capacity-scaling algorithm terminates, then f is a max-flow.
Pf.

- By integrality invariant, when $\Delta = 1 \Rightarrow G_f(\Delta) = G_f$.
- Upon termination of $\Delta = 1$ phase, there are no augmenting paths. ▪

Capacity-scaling algorithm: analysis of running time

Lemma 1. The outer while loop repeats $1 + \lceil \log_2 C \rceil$ times.
Pf. Initially $C/2 < \Delta \leq C$; Δ decreases by a factor of 2 in each iteration. ▪

Lemma 2. Let f be the flow at the end of a Δ-scaling phase. Then, the value of the max-flow $\leq$ $val(f) + m\,\Delta$. ⟵ proof on next slide

Lemma 3. There are at most $2m$ augmentations per scaling phase.
Pf.

- Let f be the flow at the end of the previous scaling phase.
- LEMMA 2 $\Rightarrow$ $val(f^*) \leq val(f) + 2\,m\,\Delta$.
- Each augmentation in a Δ-phase increases $val(f)$ by at least Δ. ▪

Theorem. The scaling max-flow algorithm finds a max flow in $O(m \log C)$ augmentations. It can be implemented to run in $O(m^2 \log C)$ time.
Pf. Follows from LEMMA 1 and LEMMA 3. ▪

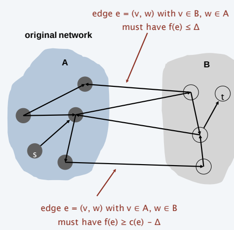
Capacity-scaling algorithm: analysis of running time

Lemma 2. Let f be the flow at the end of a Δ-scaling phase. Then, the value of the max-flow $\leq$ $val(f) + m\,\Delta$.

Pf.

- We show there exists a cut (A, B) such that $cap(A, B) \leq val(f) + m\,\Delta$.
- Choose A to be the set of nodes reachable from s in $G_f(\Delta)$.
- By definition of cut A, $s \in A$.
- By definition of flow f, $t \notin A$.

$$\begin{aligned} val(f) &= \sum_{e \text{ out of } A} f(e) - \sum_{e \text{ in to } A} f(e) \\ &\geq \sum_{e \text{ out of } A} (c(e) - \Delta) - \sum_{e \text{ in to } A} \Delta \\ &= \sum_{e \text{ out of } A} c(e) - \sum_{e \text{ out of } A} \Delta - \sum_{e \text{ in to } A} \Delta \\ &\geq cap(A, B) - m\Delta \;\blacksquare \end{aligned}$$



Shortest augmenting paths

Shortest augmenting path

Q. Which augmenting path?

A. The one with the fewest number of edges.

can find via BFS

SHORTEST-AUGMENTING-PATH(G, s, t, c)

```
FOREACH e ∈ E : f(e) ← 0.
   Gf ← residual graph.
   WHILE (there exists an augmenting path in Gf)
      P ← BREADTH-FIRST-SEARCH (Gf, s, t).
      f ← AUGMENT (f, c, P).
      Update Gf.
RETURN f.
```


Shortest augmenting path: overview of analysis

L1. Throughout the algorithm, length of the shortest path never decreases.

L2. After at most m shortest path augmentations, the length of the shortest augmenting path strictly increases.

Theorem. The shortest augmenting path algorithm runs in $O(m^2 n)$ time.

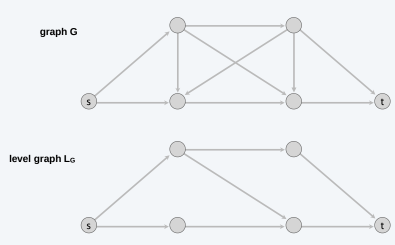
Pf.

- $O(m + n)$ time to find shortest augmenting path via BFS.
- $O(m)$ augmentations for paths of length k.
- If there is an augmenting path, there is a simple one.
  - $\Rightarrow$ $1 \leq k < n$
  - $\Rightarrow$ $O(m\,n)$ augmentations. ▪

Shortest augmenting path: analysis

Def. Given a digraph $G = (V, E)$ with source s, its level graph is defined by:

- $\ell(v)$ = number of edges in shortest path from s to v.
- $L_G = (V, E_G)$ is the subgraph of G that contains only those edges $(v,w) \in E$ with $\ell(w) = \ell(v) + 1$.



Shortest augmenting path: analysis

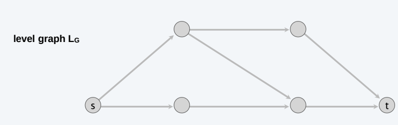
Def. Given a digraph $G = (V, E)$ with source s, its level graph is defined by:

- $\ell(v)$ = number of edges in shortest path from s to v.
- $L_G = (V, E_G)$ is the subgraph of G that contains only those edges $(v,w) \in E$ with $\ell(w) = \ell(v) + 1$.

Property. Can compute level graph in $O(m + n)$ time.

Pf. Run BFS; delete back and side edges.

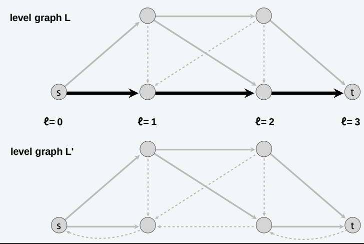
Key property. P is a shortest $s \rightsquigarrow v$ path in G iff P is an $s \rightsquigarrow v$ path L_G.



Shortest augmenting path: analysis

L1. Throughout the algorithm, length of the shortest path never decreases.

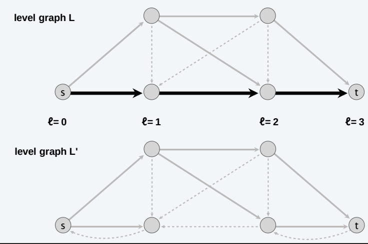
- Let f and f' be flow before and after a shortest path augmentation.
- Let L and L' be level graphs of G_f and $G_{f'}$.
- Only back edges added to $G_{f'}$
  (any path with a back edge is longer than previous length) ▪



Shortest augmenting path: analysis

L2. After at most m shortest path augmentations, the length of the shortest augmenting path strictly increases.

- The bottleneck edge(s) is deleted from L after each augmentation.
- No new edge added to L until length of shortest path strictly increases. ▪



Shortest augmenting path: review of analysis

L1. Throughout the algorithm, length of the shortest path never decreases.

L2. After at most m shortest path augmentations, the length of the shortest augmenting path strictly increases.

Theorem. The shortest augmenting path algorithm runs in $O(m^2 n)$ time.

Pf.

- $O(m + n)$ time to find shortest augmenting path via BFS.
- $O(m)$ augmentations for paths of exactly k edges.
- $O(m n)$ augmentations. ▪

Shortest augmenting path: improving the running time

Note. $\Theta(m n)$ augmentations necessary on some networks.

- Try to decrease time per augmentation instead.
- Simple idea ⇒ $O(m n^2)$ [Dinic 1970]
- Dynamic trees ⇒ $O(m n \log n)$ [Sleator-Tarjan 1983]


A Data Structure for Dynamic Trees

Daniel D. Sleator and Robert Endre Tarjan

Bell Laboratories, Murray Hill, New Jersey 07974

Received May 8, 1982; revised October 18, 1982

Blocking flow

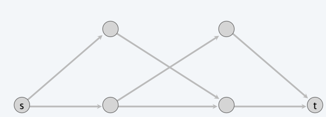
Blocking-flow algorithm

Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.



Blocking-flow algorithm

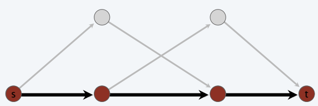
Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.

advance



level graph L_G

Blocking-flow algorithm

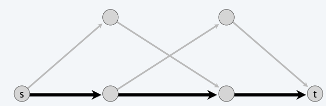
Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.

augment



level graph L_G

Blocking-flow algorithm

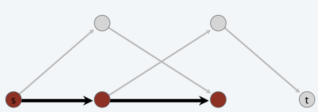
Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.

advance



level graph L_G

Blocking-flow algorithm

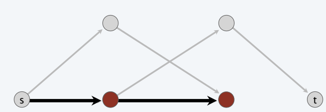
Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.

retreat



level graph L_G

Blocking-flow algorithm

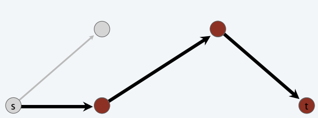
Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.

advance



level graph L_G

Blocking-flow algorithm

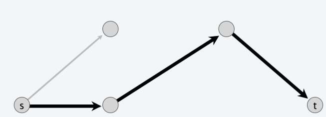
Two types of augmentations.

- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.

augment

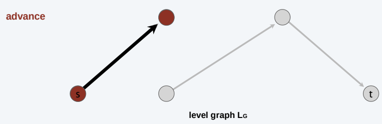


level graph L_G

Blocking-flow algorithm

Two types of augmentations.
- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

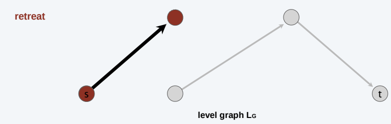
Phase of normal augmentations.
- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.



Blocking-flow algorithm

Two types of augmentations.
- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

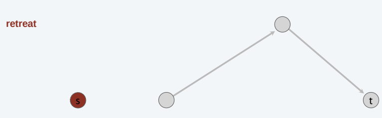
Phase of normal augmentations.
- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.



Blocking-flow algorithm

Two types of augmentations.
- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

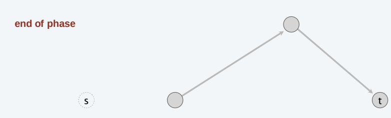
Phase of normal augmentations.
- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.



Blocking-flow algorithm

Two types of augmentations.
- Normal: length of shortest path does not change.
- Special: length of shortest path strictly increases.

Phase of normal augmentations.
- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.



Blocking-flow algorithm

```
INITIALIZE(G, s, t, f, c)
  L_G ← level-graph of G_f.
  P ← ∅.
  GOTO ADVANCE(s).

RETREAT(v)
  IF (v = s) STOP.
  ELSE
    Delete v (and all incident edges) from L_G.
    Remove last edge (u, v) from P.
    GOTO ADVANCE(u).

ADVANCE(v)
  IF (v = t)
    AUGMENT(P).
    Remove saturated edges from L_G.
    P ← ∅.
    GOTO ADVANCE(s).

  IF (there exists edge (v, w) ∈ L_G)
    Add edge (v, w) to P.
    GOTO ADVANCE(w).

  ELSE GOTO RETREAT(v).
```


Blocking-flow algorithm: analysis

Lemma. A phase can be implemented in $O(mn)$ time.

Pf.
- Initialization happens once per phase. ⟵ $O(m)$ using BFS
- At most m augmentations per phase. ⟵ $O(mn)$ per phase
  (because an augmentation deletes at least one edge from L_G)
- At most n retreats per phase. ⟵ $O(m + n)$ per phase
  (because a retreat deletes one node from L_G)
- At most mn advances per phase. ⟵ $O(mn)$ per phase
  (because at most n advances before retreat or augmentation) ▪

Theorem. [Dinic 1970] The blocking-flow algorithm runs in $O(mn^2)$ time.

Pf.
- By lemma, $O(mn)$ time per phase.
- At most n phases (as in shortest augment path analysis). ▪

Choosing good augmenting paths: summary

Assumption. Integer capacities between 1 and C.

method	# augmentations	running time
augmenting path	$n\,C$	$O(m\,n\,C)$
fattest augmenting path	$m \log (mC)$	$O(m^2 \log n \log (mC))$
capacity scaling	$m \log C$	$O(m^2 \log C)$
improved capacity scaling	$m \log C$	$O(m\,n \log C)$
shortest augmenting path	$m\,n$	$O(m^2\,n)$
improved shortest augmenting path	$m\,n$	$O(m\,n^2)$
dynamic trees	$m\,n$	$O(m\,n \log n)$

Maximum flow algorithms: theory

year	method	worst case	discovered by
1951	simplex	$O(m^3\,C)$	Dantzig
1955	augmenting path	$O(m^2\,C)$	Ford-Fulkerson
1970	shortest augmenting path	$O(m^3)$	Dinic, Edmonds-Karp
1970	fattest augmenting path	$O(m^2 \log m \log(\,m\,C\,))$	Dinic, Edmonds-Karp
1977	blocking flow	$O(m^{5/2})$	Cherkasky
1978	blocking flow	$O(m^{7/3})$	Galil
1983	dynamic trees	$O(m^2 \log m)$	Sleator-Tarjan
1985	capacity scaling	$O(m^2 \log C)$	Gabow
1997	length function	$O(m^{3/2} \log m \log C)$	Goldberg-Rao
2012	compact network	$O(m^2 / \log m)$	Orlin
?	?	$O(m)$	?

max-flow algorithms for sparse digraphs with m edges, integer capacities between 1 and C

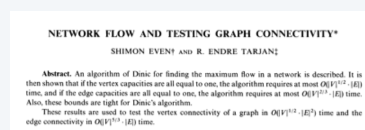
Unit capacity simple graphs

Bipartite matching

Q. Which max-flow algorithm to use for bipartite matching?
- Generic augmenting path: $O(\,m\,|f^*|\,) = O(m\,n)$.
- Capacity scaling: $O(m^2 \log U) = O(m^2)$.
- Shortest augmenting path: $O(m\,n^2)$.

Q. Suggests "more clever" algorithms are not as good as we first thought?
A. No, just need more clever analysis!

Next.


NETWORK FLOW AND TESTING GRAPH CONNECTIVITY*

SHIMON EVEN† AND R. ENDRE TARJAN‡

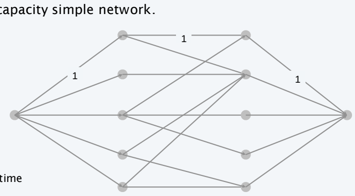
Abstract. An algorithm of Dinic for finding the maximum flow in a network is described. It is then shown that if the vertex capacities are all equal to one, the algorithm requires at most $O(|V|^{1/2} \cdot |E|)$ time, and if the edge capacities are all equal to one, the algorithm requires at most $O(|V|^{2/3} \cdot |E|)$ time. Also, these bounds are tight for Dinic's algorithm.

These results are used to test the vertex connectivity of a graph in $O(|V|^{1/2} \cdot |E|^2)$ time and the edge connectivity in $O(|V|^{5/3} \cdot |E|)$ time.

Unit-capacity simple networks

Def. A network is a unit-capacity simple network if:
- Every edge capacity is 1.
- Every node (other than s or t) has either (i) at most one entering edge or (ii) at most one leaving edge.

Property. Let G be a simple unit-capacity network and let f be a 0-1 flow, then G_f is a unit-capacity simple network.



We will prove a running time of $O(mn^{1/2})$

Unit-capacity simple networks

Shortest augmenting path algorithm.
- Normal augmentation: length of shortest path does not change.
- Special augmentation: length of shortest path strictly increases.

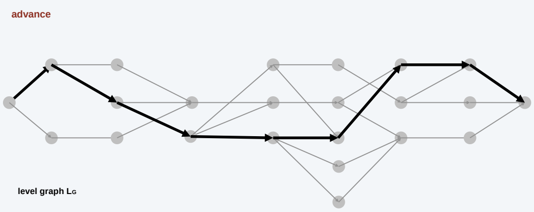
Theorem. [Even-Tarjan 1975] In unit-capacity simple networks, the shortest augmenting path algorithm computes a maximum flow in $O(m\,n^{1/2})$ time.

Pf.
- L1. Each phase of normal augmentations takes $O(m)$ time.
- L2. After at most $n^{1/2}$ phases, $|\,f\,| \geq |\,f^*| - n^{1/2}$.
- L3. After at most $n^{1/2}$ additional augmentations, flow is optimal. ▪

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ⟵ delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

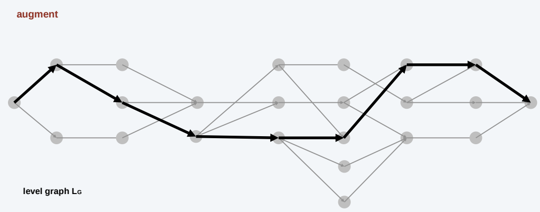


44

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ⟵ delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

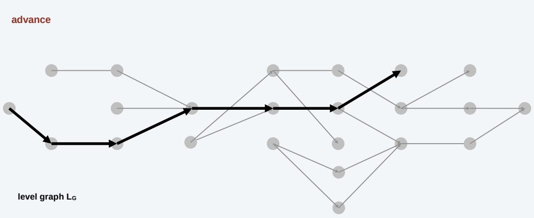


45

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ⟵ delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

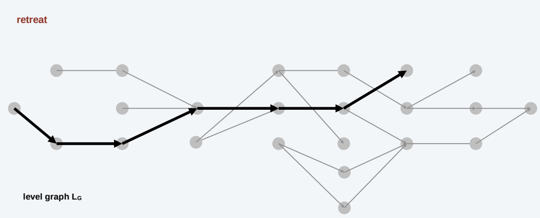


46

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ⟵ delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

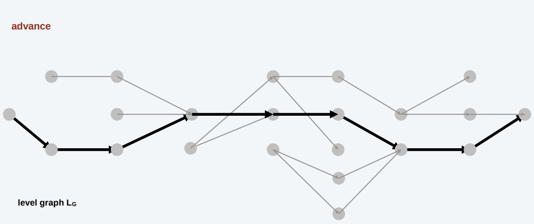


47

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ⟵ delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

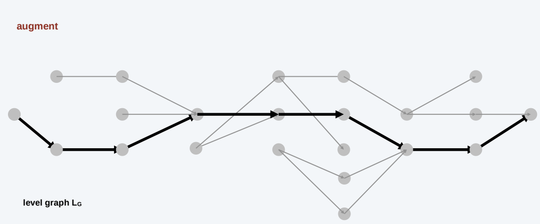


48

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ⟵ delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.

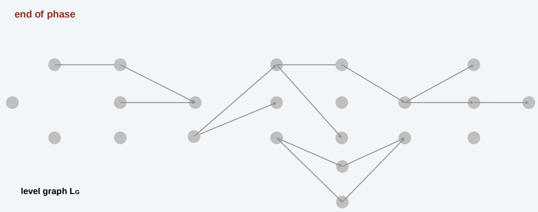


49

Unit-capacity simple networks

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G. ← delete all edges in augmenting path from L_G
- If get stuck, delete node from L_G and go to previous node.



Unit-capacity simple networks: analysis

Phase of normal augmentations.

- Explicitly maintain level graph L_G.
- Start at s, advance along an edge in L_G until reach t or get stuck.
- If reach t, augment and and update L_G.
- If get stuck, delete node from L_G and go to previous node.

LEMMA 1. A phase of normal augmentations takes $O(m)$ time.

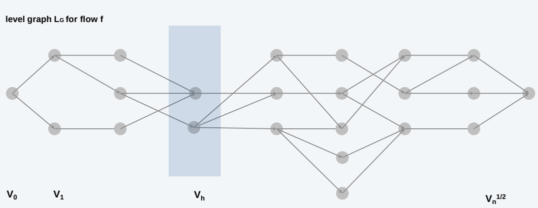
Pf.

- $O(m)$ to create level graph L_G.
- $O(1)$ per edge since each edge traversed and deleted at most once.
- $O(1)$ per node since each node deleted at most once. ▪

Unit-capacity simple networks: analysis

LEMMA 2. After at most $n^{1/2}$ phases, $|f| \geq |f^*| - n^{1/2}$.

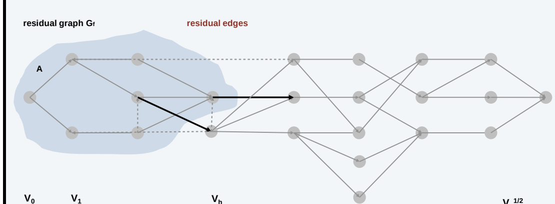
- After $n^{1/2}$ phases, length of shortest augmenting path is $> n^{1/2}$.
- Level graph has more than $n^{1/2}$ levels.
- Let $1 \leq h \leq n^{1/2}$ be layer with min number of nodes: $|V_h| \leq n^{1/2}$.



Unit-capacity simple networks: analysis

LEMMA 2. After at most $n^{1/2}$ phases, $|f| \geq |f^*| - n^{1/2}$.

- After $n^{1/2}$ phases, length of shortest augmenting path is $> n^{1/2}$.
- Level graph has more than $n^{1/2}$ levels.
- Let $1 \leq h \leq n^{1/2}$ be layer with min number of nodes: $|V_h| \leq n^{1/2}$.
- Let $A = \{v : \ell(v) < h\} \cup \{v : \ell(v) = h$ and v has ≤ 1 outgoing residual edge$\}$.
- $cap_f(A, B) \leq |V_h| \leq n^{1/2} \Rightarrow |f| \geq |f^*| - n^{1/2}$. ▪

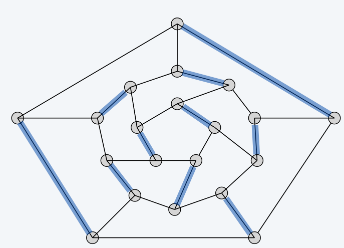


Bipartite matching

Matching

Def. Given an undirected graph $G = (V, E)$ a subset of edges $M \subseteq E$ is a matching if each node appears in at most one edge in M.

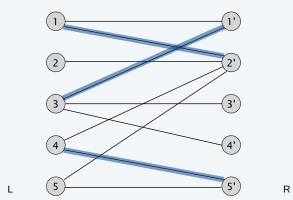
Max matching. Given a graph, find a max cardinality matching.



Bipartite matching

Def. A graph G is bipartite if the nodes can be partitioned into two subsets L and R such that every edge connects a node in L to one in R.

Bipartite matching. Given a bipartite graph $G = (L \cup R, E)$, find a max cardinality matching.



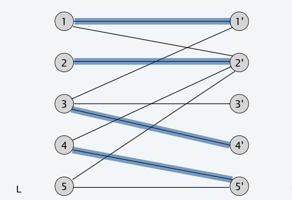
matching: 1-2', 3-1', 4-5'

56

Bipartite matching

Def. A graph G is bipartite if the nodes can be partitioned into two subsets L and R such that every edge connects a node in L to one in R.

Bipartite matching. Given a bipartite graph $G = (L \cup R, E)$, find a max cardinality matching.

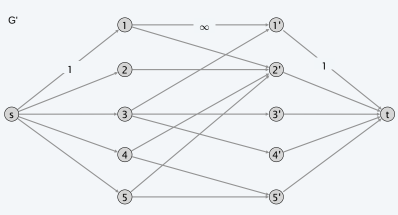


matching: 1-1', 2-2', 3-4', 4-5'

57

Bipartite matching: max flow formulation

- Create digraph $G' = (L \cup R \cup \{s, t\}, E')$.
- Direct all edges from L to R, and assign infinite (or unit) capacity.
- Add source s, and unit capacity edges from s to each node in L.
- Add sink t, and unit capacity edges from each node in R to t.



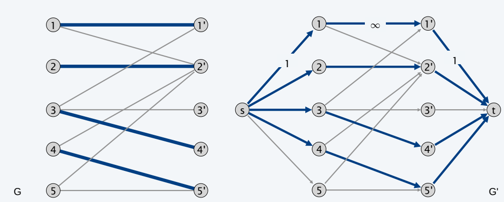
58

Max flow formulation: proof of correctness

Theorem. Max cardinality of a matching in G = value of max flow in G'.

Pf. δ

- Given a max matching M of cardinality k.
- Consider flow f that sends 1 unit along each of k paths.
- f is a flow, and has value k. ▪



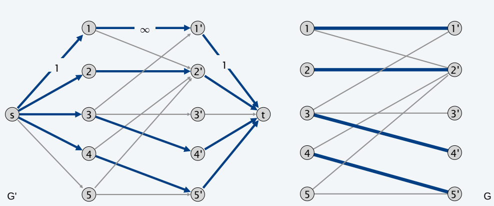
59

Max flow formulation: proof of correctness

Theorem. Max cardinality of a matching in G = value of max flow in G'.

Pf.

- Let f be a max flow in G' of value k.
- Integrality theorem implies k is integral and can assume f is 0-1.
- Consider M = set of edges from L to R with $f(e) = 1$.
  - each node in L and R participates in at most one edge in M
  - $|M| = k$: consider cut $(L \cup s, R \cup t)$ ▪



60

Perfect matching in a bipartite graph

Def. Given a graph $G = (V, E)$ a subset of edges $M \subseteq E$ is a perfect matching if each node appears in exactly one edge in M.

Q. When does a bipartite graph have a perfect matching?

A. Hall's Theorem. Let $G = (L \cup R, E)$ be a bipartite graph with $|L| = |R|$. G has a perfect matching iff $|N(S)| \geq |S|$ for all subsets $S \subseteq L$.

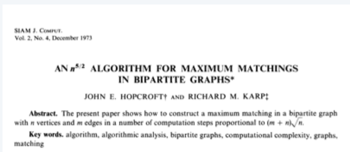
On problem set!

61

Bipartite matching running time

Theorem. The Ford-Fulkerson algorithm solves the bipartite matching problem in $O(m\,n)$ time.

Theorem. [Hopcroft-Karp 1973] The bipartite matching problem can be solved in $O(m\,n^{1/2})$ time.


SIAM J. COMPUT.
Vol. 2, No. 4, December 1973

AN $n^{5/2}$ ALGORITHM FOR MAXIMUM MATCHINGS IN BIPARTITE GRAPHS*

JOHN E. HOPCROFT† AND RICHARD M. KARP‡

Abstract. The present paper shows how to construct a maximum matching in a bipartite graph with n vertices and m edges in a number of computation steps proportional to $(m + n)\sqrt{n}$.

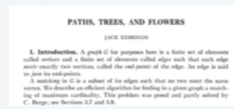
Key words. algorithm, algorithmic analysis, bipartite graphs, computational complexity, graphs, matching

62

Nonbipartite matching

Nonbipartite matching. Given an undirected graph (not necessarily bipartite), find a matching of maximum cardinality.

- Structure of nonbipartite graphs is more complicated.
- But well-understood. [Tutte-Berge, Edmonds-Galai]
- Blossom algorithm: $O(n^4)$. [Edmonds 1965]
- Best known: $O(m\,n^{1/2})$. [Micali-Vazirani 1980, Vazirani 1994]


PATHS, TREES, AND FLOWERS

JACK EDMONDS


COMBINATORICA

A THEORY OF ALTERNATING PATHS AND BLOSSOMS FOR PROVING CORRECTNESS OF THE $O(\sqrt{V}E)$ GENERAL GRAPH MAXIMUM MATCHING ALGORITHM

VIJAY V. VAZIRANI

63

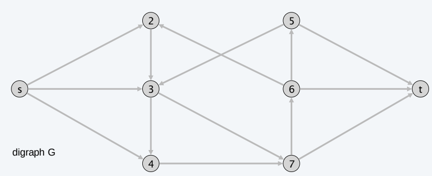
Edge-disjoint paths

May 8, 2014 CS38 Lecture 12 64

Edge-disjoint paths

Def. Two paths are edge-disjoint if they have no edge in common.

Disjoint path problem. Given a digraph $G = (V, E)$ and two nodes s and t, find the max number of edge-disjoint $s \rightsquigarrow t$ paths.



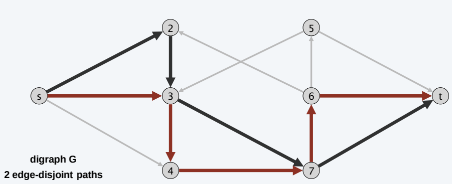
65

Edge-disjoint paths

Def. Two paths are edge-disjoint if they have no edge in common.

Disjoint path problem. Given a digraph $G = (V, E)$ and two nodes s and t, find the max number of edge-disjoint $s \rightsquigarrow t$ paths.

Ex. Communication networks.



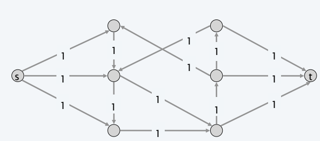
66

Edge-disjoint paths

Max flow formulation. Assign unit capacity to every edge.

Theorem. Max number edge-disjoint $s \rightsquigarrow t$ paths equals value of max flow.
Pf. ≤

- Suppose there are k edge-disjoint $s \rightsquigarrow t$ paths $P_1, \ldots, P_k$.
- Set $f(e) = 1$ if e participates in some path P_j ; else set $f(e) = 0$.
- Since paths are edge-disjoint, f is a flow of value k. ▪



67

Edge-disjoint paths

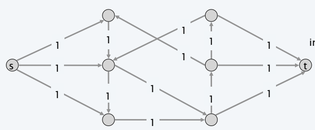
Max flow formulation. Assign unit capacity to every edge.

Theorem. Max number edge-disjoint $s \leadsto t$ paths equals value of max flow.

Pf. ε

- Suppose max flow value is k.
- Integrality theorem implies there exists 0-1 flow f of value k.
- Consider edge (s, u) with $f(s, u) = 1$.
  - by conservation, there exists an edge (u, v) with $f(u, v) = 1$
  - continue until reach t, always choosing a new edge
- Produces k (not necessarily simple) edge-disjoint paths. ▪

can eliminate cycles
to get simple paths
in O(mn) time if desired
(flow decomposition)



68