Maximum Flow in Planar Networks

Alon Itai*, Yossi Shiloach†

Naman Kumar, Diptaksho Palit

Presented At: Chennai Mathematical Institute

Department of Computer Science, Technion*
Department of Applied Maths, Weizmann†

November 26, 2022

Introduction

- A directed flow network $N = (G, s, t, c)$ is a quadruple where:
  - $G = (V, E)$ is a directed linear graph;
  - s and t are distinct vertices, the source and the terminal respectively;
  - $c : E \to \mathbb{R}^+$ is the capacity function.
- Let n and m denote the number of vertices and edges in the graph respectively.

Introduction

- A function $f : E \to \mathbb{R}^+$ is a *flow* if it satisfies:
  - the capacity rule: $f(e) \leq c(e) \forall e \in E$;
  - the conservation rule: $IN(f, v) = OUT(f, v) \quad \forall v \in V \backslash \{s, t\}$.
- Here
  - $IN(f, v) = \sum_{u:(u,v) \in E} f(u, v)$; and
  - $OUT(f, v) = \sum_{u:(v,u) \in E} f(v, u)$.
- The flow value $|f|$ is defined by

$$|f| = OUT(f, s) - IN(f, s).$$

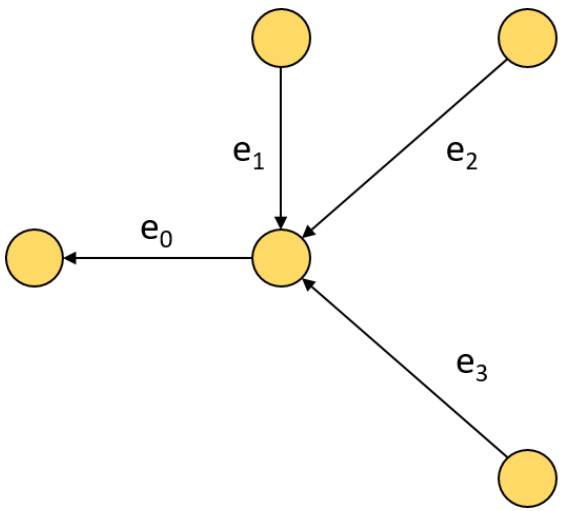
- A flow is a *maximum flow* if $|f| \geq |f'|$ for any other flow $|f'|$.

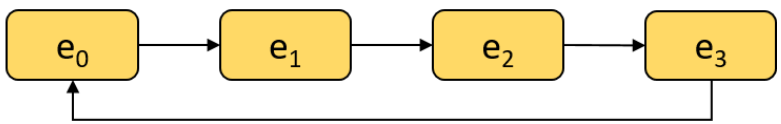
Introduction

- We will discuss the problem of finding a maximum flow in planar networks.
- Berge proposed an algorithm to find a maximum flow, a straightforward implementation of which requires $O(n^2)$ time.
- This paper presents an $O(n \log n)$ time implementation for the algorithm.
- It also gives an $O(n \log n)$ lower bound to any implementation of Berge's algorithm.

Setup

- Assume G has a fixed planar representation.
- The adjacency list is represented by a circular list corresponding to the circular clockwise ordering around edges.





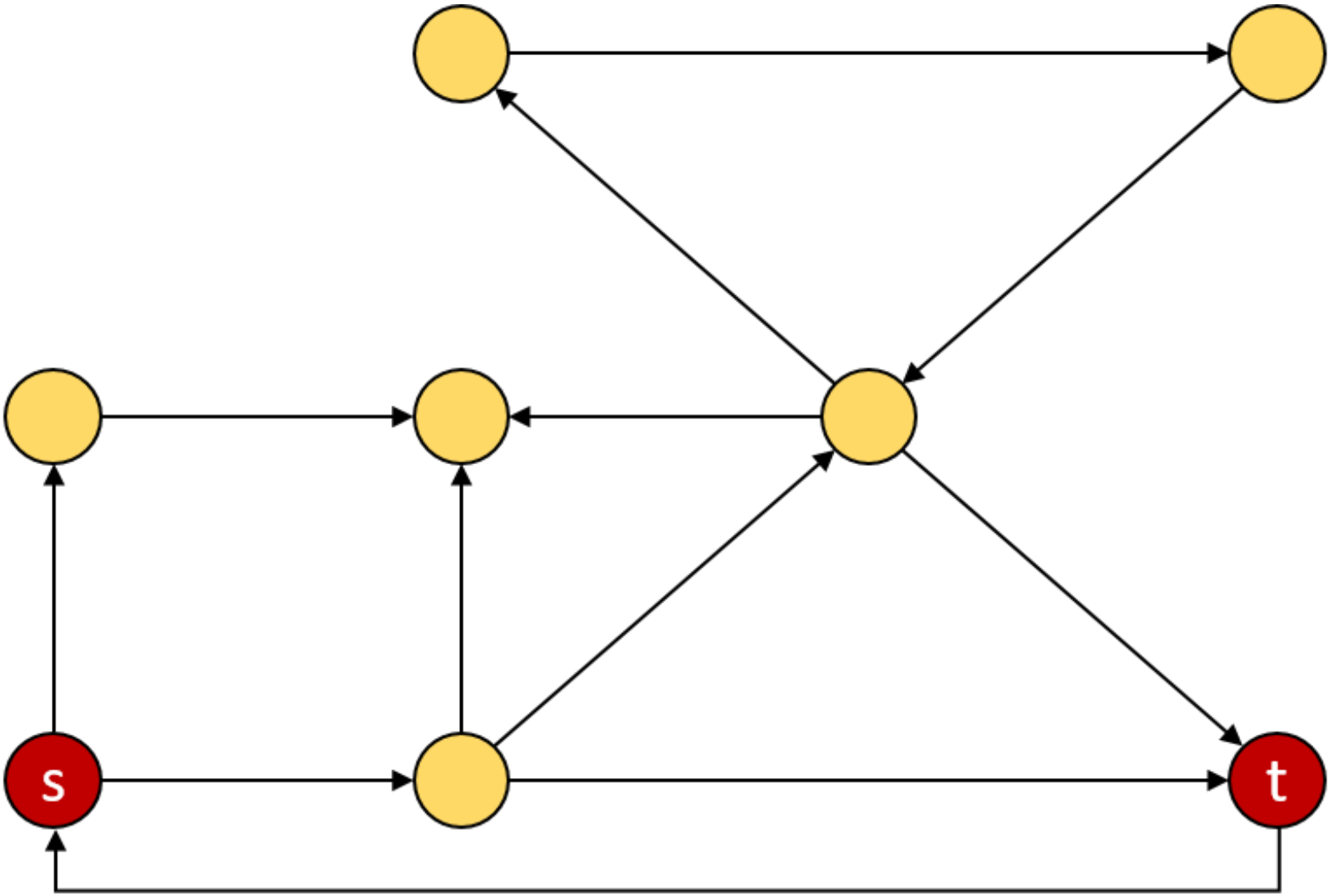
Setup

- We first deal with the case where s and t are on the same face.
- Without loss of generality, assume $(t, s) \in E$ (otherwise it can be added with zero capacity).

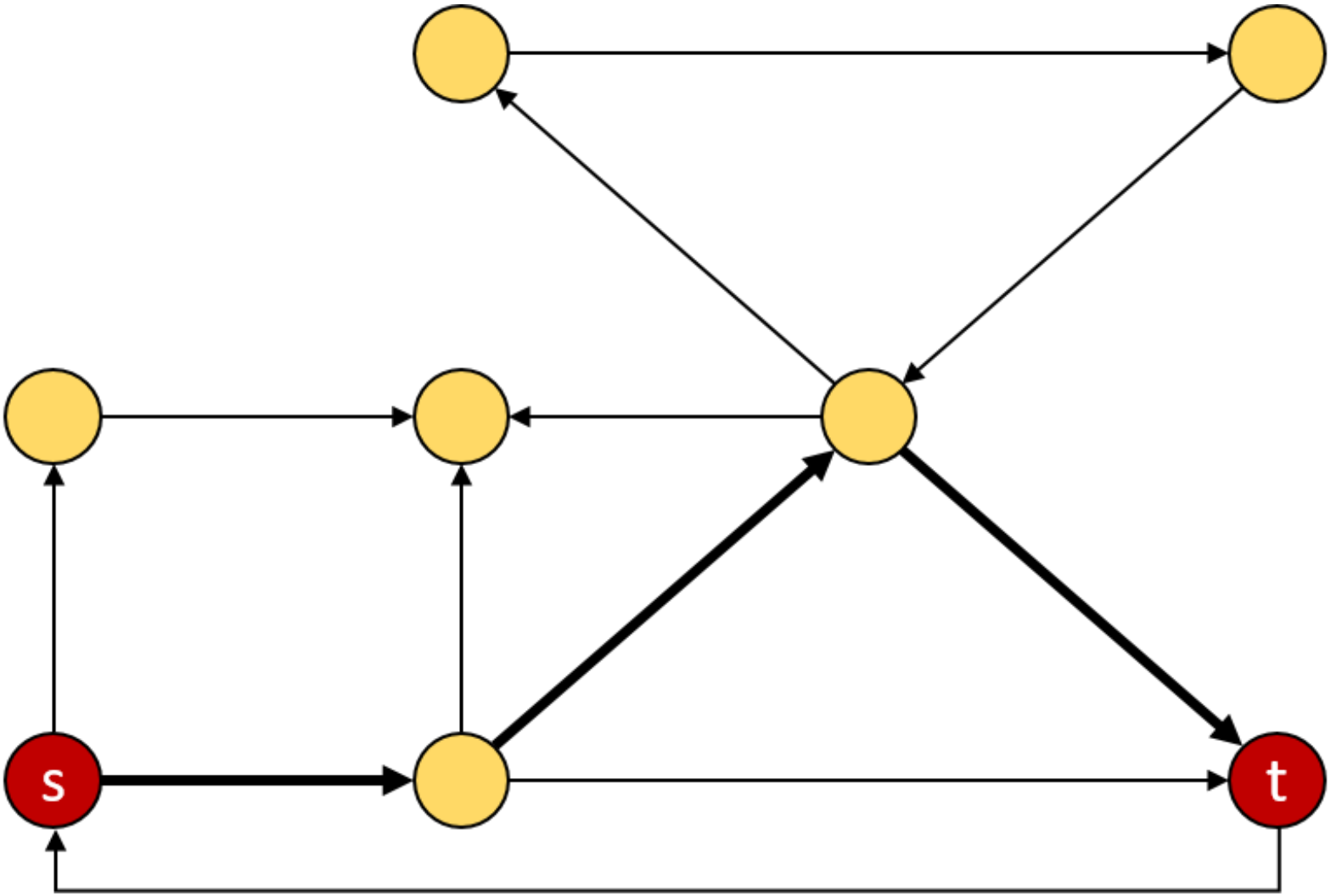
Setup

- We define an ordering on paths from s to t.
- $P_1 = (s, v_1, \cdots, v_k, t)$ *lies above* $P_2 = (s, u_1, \cdots, u_l, t)$ if
  - $v_i = u_i$ for all $1 \leq i \leq r$ and $v_{r+1} \neq u_{r+1}$;
  - (v_r, v_{r+1}) appears before (v_r, u_{r+1}) in the clockwise order starting at (v_{r-1}, v_r).
- This induces a total ordering on the paths, and gives us a unique *uppermost path*.

Example: Uppermost Path



Example: Uppermost Path



Berge's Algorithm

1. Initialize: set $i = 1$; start with zero flow: set $f_0(e) = 0$ and $\text{res}(e) = c(e) \quad \forall e \in E$.
2. Find the uppermost path P_i, if none exists then stop.
3. Let e_i be the bottleneck of P_i (edge with lowest residual capacity).
4. Increase the flow by $\text{res}(e_i)$ units along P_i:
   - $f_i(e) = \begin{cases} f_{i-1}(e) + res(e_i) & \text{if } e \in P_i \\ f_{i-1}(e) & \text{otherwise} \end{cases}$

   $\text{res}(e) = c(e) - f_i(e)$.
5. Delete the bottleneck e_i from G.
6. Set $i = i + 1$ and go to step 2.

Berge's Algorithm: Analysis

- A straightforward implementation of Berge's algorithm requires $O(n^2)$ time.
- Let $I(e)$ and $L(e)$ denote the index of the first and last *uppermost paths* in which the edge e participates.

Lemma

If e participates in any uppermost path then e participates in all the paths between $P_{I(e)}$ and $P_{L(e)}$.

- The following corollary follows immediately: Let $e \in E$ and $I(e) \leq i \leq L(e)$ then $f_i(e) = |f_i| - |f_{I(e)-1}|$.

Modified Capacity Algorithm

1. Initialize: set $i = 1$, $|f_0| = 0$, $|P_0| = \varnothing$.
2. Find the uppermost path P_i, if none exists then go to step 7.
3. For $e \in P_i \backslash P_{i-1}$, set $M(e) = c(e) + |f_{i-1}|$
4. Find a bottleneck $e_i \in P_i$ (edge with lowest modified capacity); set $|f_i| = M(e_i)$.
5. Delete e_i from E.
6. Set $i = i + 1$ and go to step 2.
7. Find the flow of each edge; set

$$f(e) = \begin{cases} 0, & \text{if } e \text{ is not in any } \textit{uppermost path}, \\ |f_{L(e)|} - |f_{l(e)}| & \text{otherwise.} \end{cases}$$

Modified Capacity Algorithm: Analysis

- It can be shown that the P_i and e_i, and subsequently $|f_i|$ are equal in both algorithms. This is because

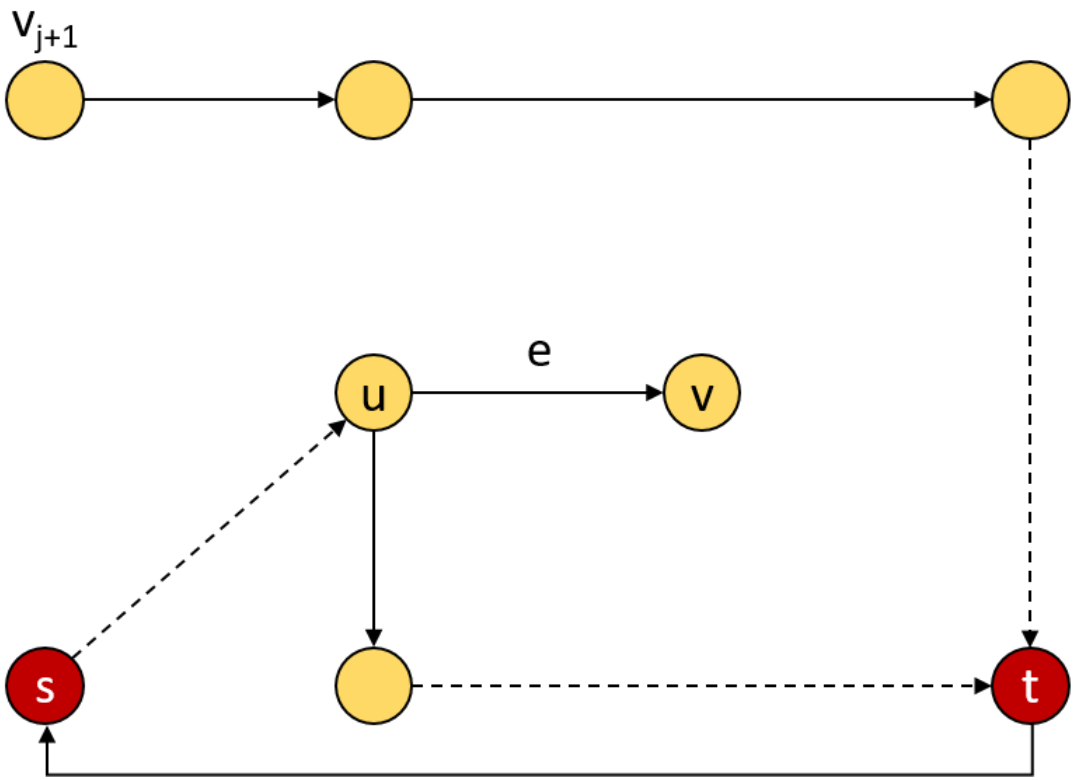
$$\begin{aligned}\text{res}(e) &= c(e) - f_{i-1}(e) \\ &= c(e) - (|f_{i-1}| - |f_{I(e)-1}|) \\ &= c(e) + |f_{I(e)-1}| - |f_{i-1}| \\ &= M(e) - |f_{i-1}|\end{aligned}$$

- The algorithm thus correctly finds a maximum flow.
- Next we describe how to efficiently generate *uppermost paths*.

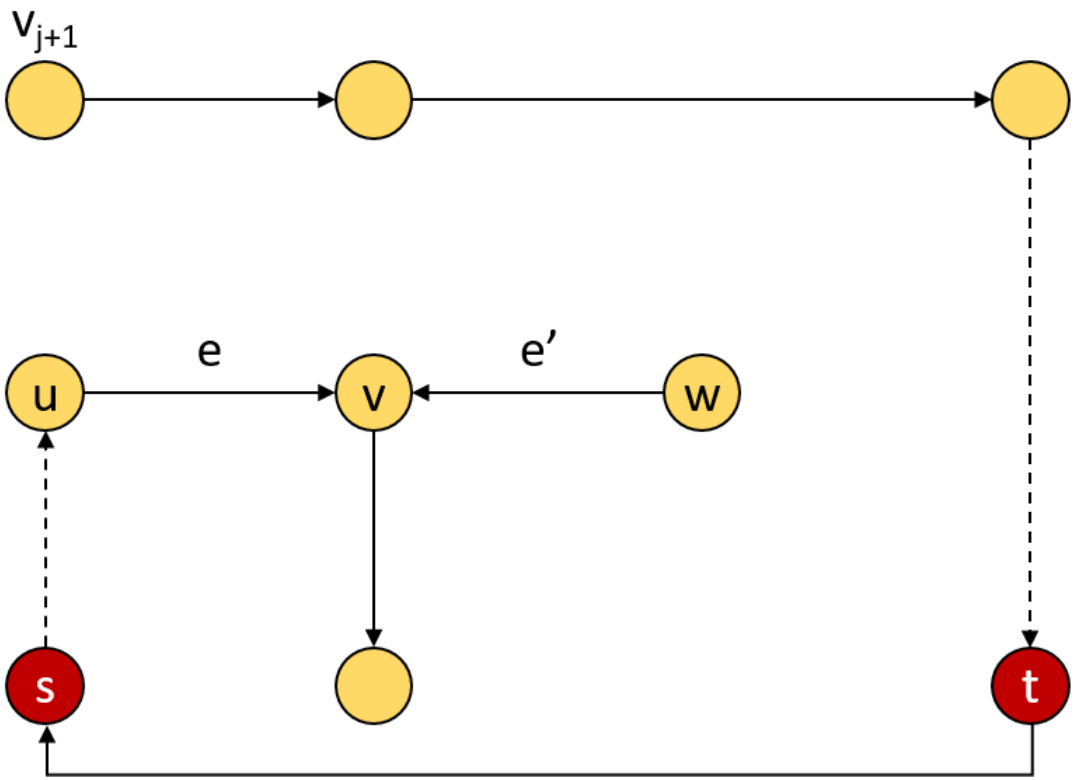
Modified Capacity Algorithm: Analysis

- Given P_{i-1}, we remove e_{i-1} and proceed with DFS on the subpath starting from s.
- It can be shown that, going by the clockwise ordering from the previous edge in the path, edges need to be traversed only once, and can be removed if they do not end in a path to t.
- This gives an $O(m)$ runtime bound on the overall execution of *uppermost paths*.
- We can use this to get the desired $O(m \log m)$ bound by using a priority queue to store the edges in the path (by modified capacity value).

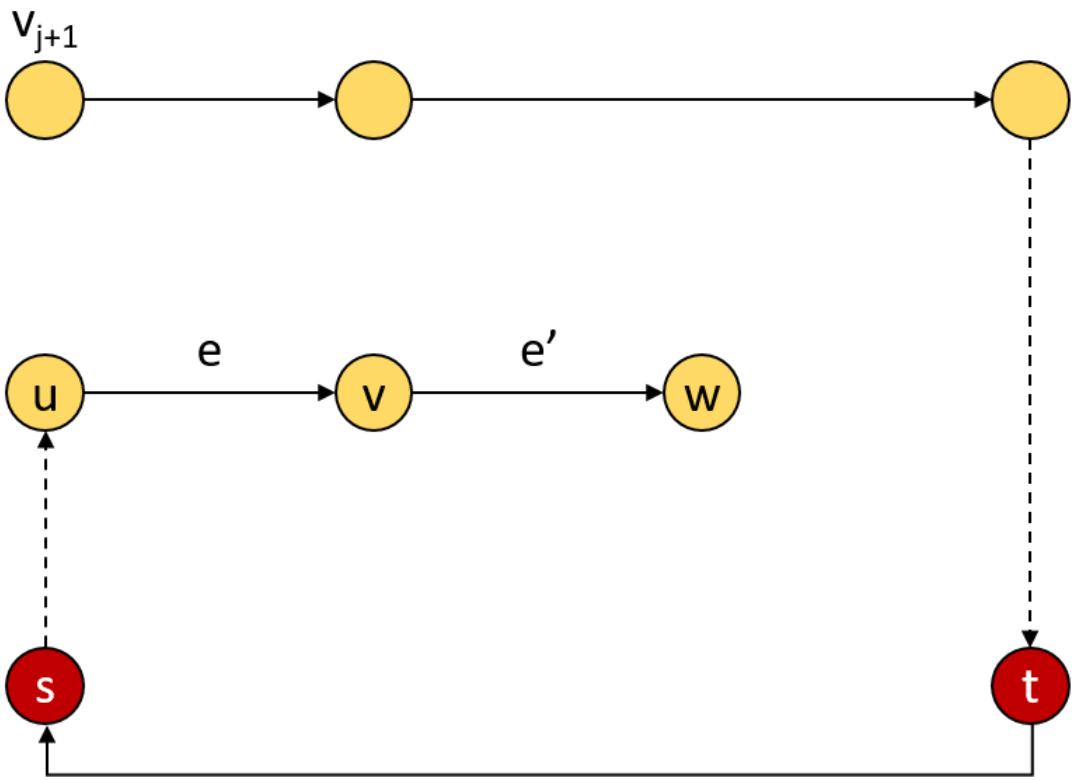
Finding the uppermost path: DFS



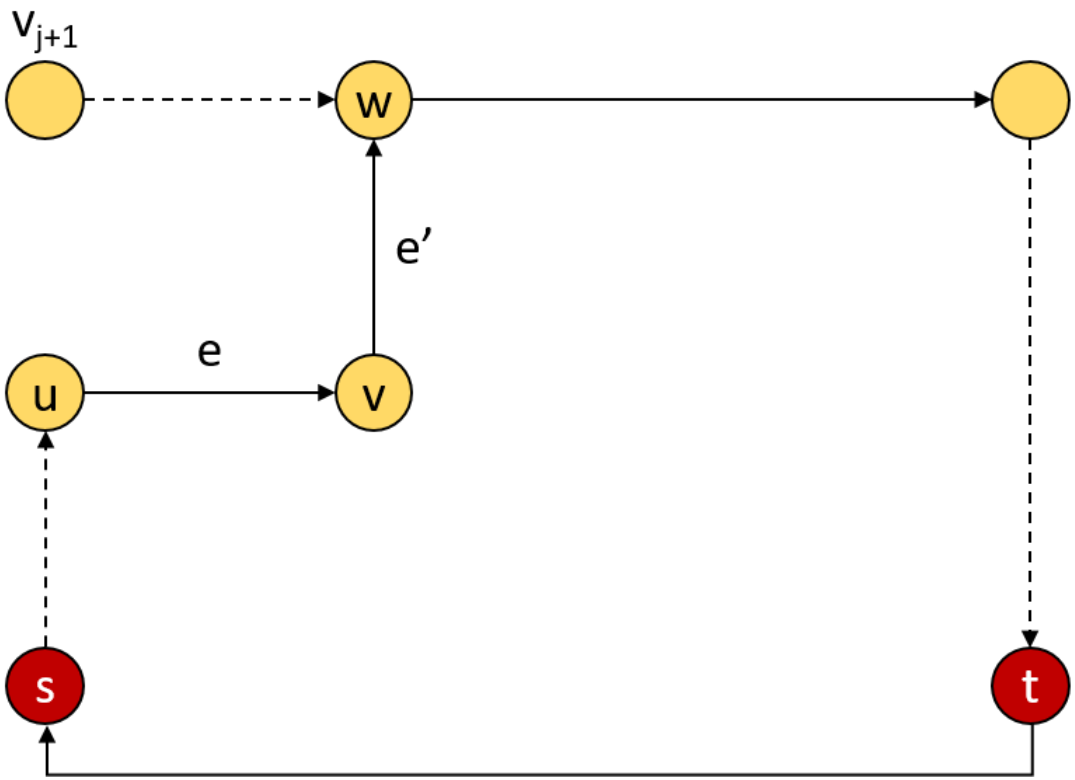
Finding the uppermost path: DFS



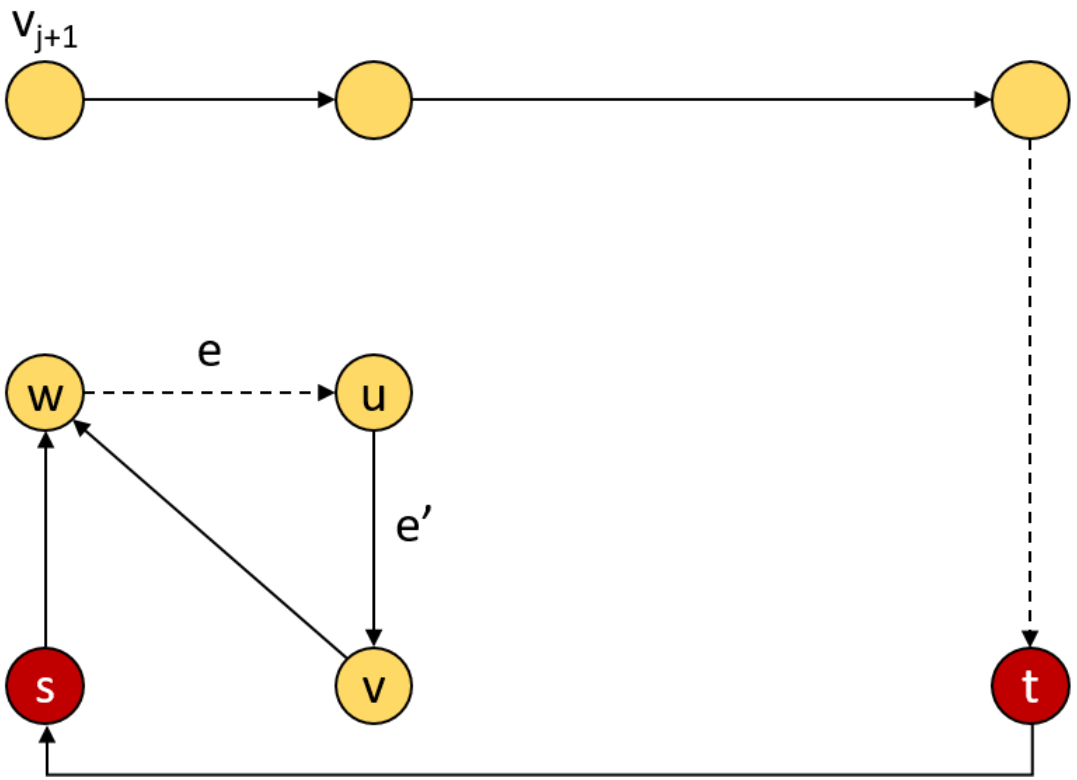
Finding the uppermost path: DFS



Finding the uppermost path: DFS

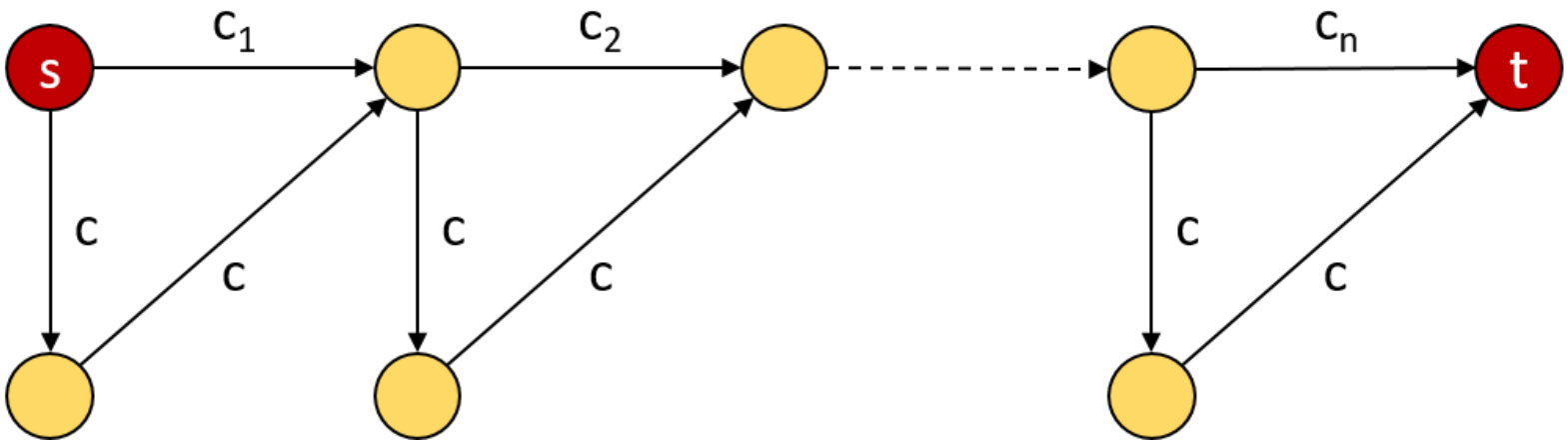


Finding the uppermost path: DFS



Modified Capacity Algorithm: Conclusion

- We have thus shown an $O(m \log m)$ implementation for Berge's Algorithm.
- From Euler's Equality, in planar graphs $m = O(n)$.
- To show that this is tight for all implementations of Berge's Algorithm, we use a reduction from sorting.



The Problem

- Let N be a general planar network
- Take $D \in \mathbb{R}^+$, where a flow of amount D exists if $D \leq C$ where C is the size of the mincut

Problem Statement

Does a flow network N have a flow of value D?

- We show an algorithm that works in time $O(n^2 \log n)$
- The difference between the previous algorithm and this algorithm is that s and t do not necessarily need to be on the same face

Pseudo-flows

- We call a function $f : E \to \mathbb{R}^+$ a **pseudo-flow** if it satisfies the conservation rule (flow in = flow out) but not the capacity rule (flow through an edge must be less than capacity)
- An edge is **over-flowed** if $f(e) < c(e)$
- We follow the following conventions:
  - If $e \in E$ then also $\overleftarrow{e} \in E$
  - If a flow or pseudo-flow passes through e, no flow passes through $\overleftarrow{e}$
  - We define

$$(f_1 \pm f_2)(e) = \max\{0, f_1(e) - f_1(\overleftarrow{e}) \pm (f_2(e) - f_2(\overleftarrow{e}))\}$$

The Algorithm

1. Find a shortest (s, t)-path, P.
2. Let f be the pseudo-flow obtained by pushing D units of flow through P
3. Choose an overflowed edge $e_0 : x \to y$. If none exists, stop.
4. Define the residual graph N', where

$$c'(e) = \begin{cases} 0 & \text{if } f(e) > c(e) \\ c(e) - f(e) & \text{if } c(e) \geq f(e) > 0 \\ c(e) + f(\overline{e}) & \text{if } c(e) + f(\overline{e}) \end{cases}$$

   Find a flow f' in N such that $|f'| = f(e_0) - c(e_0)$. If none exists, stop.
5. Set $f'(\overline{e}) = |f'|$, $f = f + f'$, go to 3.

Analysis

- First, why does this algorithm terminate?
- The maximum number of executions is $|P|$, since after each iteration the selected edge is 'fixed'
- Now we show that in step 4, we can find such a flow if N has a flow with D units.
  - Suppose there is a flow of value D, then let $f^* = f_D - f$
  - We take

$$|f^*|_E| = f_D(\overline{e}) - f(\overline{e}) - f_D(e) + f(e) \geq f(e) - c(e) > 0$$

  - It follows that such a flow exists

Analysis

- We now check **correctness**
- If $D \leq C$, there is a flow of value D in N. Then we never terminate in step 4, and thus terminate in step 3 when we can no longer find overflowed edges, hence we have found a flow
- If $D > C$, then we can't find such a flow, and we terminate at step 4, correctly identifying when we have no flow

Analysis

- We now check **complexity**
- Step 1 is just reachability, which can be done in $O(n)$.
- Step 2 takes $O(P) \leq O(n)$ time.
- Step 3 takes $O(1)$ time.
- Step 4 takes $O(n \log n)$ time, since x and y are on the same face and we can use the previous algorithm to get a flow.
- Step 5 takes $O(m) = O(n)$ time.

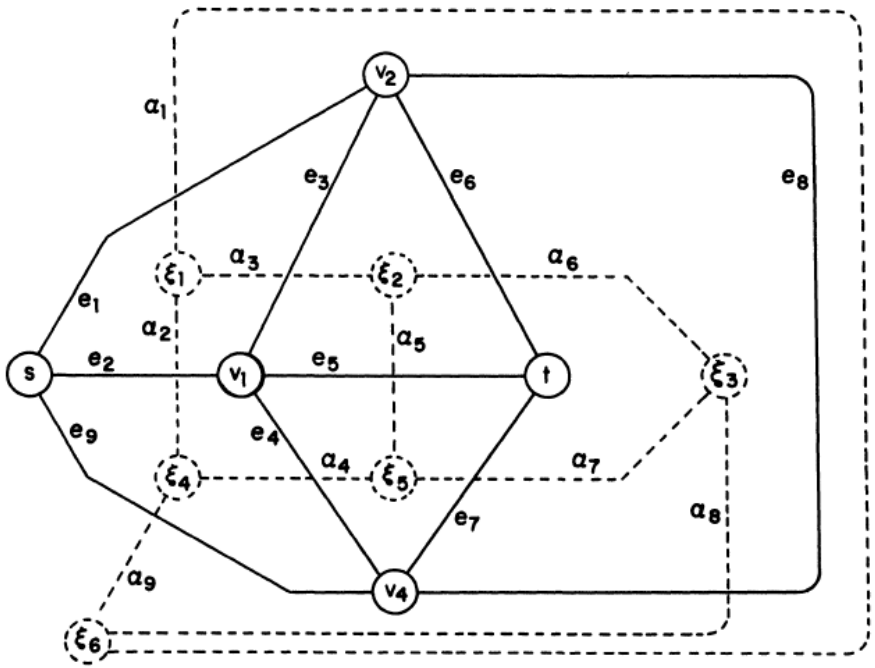
We conclude that the algorithm takes a maximum of $O(pn \log n) = O(n^2 \log n)$ time.

Min-Cut

- We now turn to the problem of finding a min (s, t)-cut in a planar network
- The upcoming algorithm runs in time $O(n^2 \log n)$
- The album works only for undirected graphs, the authors were unable to get a good runtime algorithm for directed graphs

Dual of a Planar Graph

- We briefly consider the dual of a planar graph, the graph formed by replacing faces with vertices and vice versa
- There is a one-to-one correspondence between faces and vertices and edges and edges between a graph and a dual
- We let $\varphi_v \leftrightarrow v$, and $\xi_i \leftrightarrow \text{face}_i$



Basic Concept

We use the following lemma without proof.

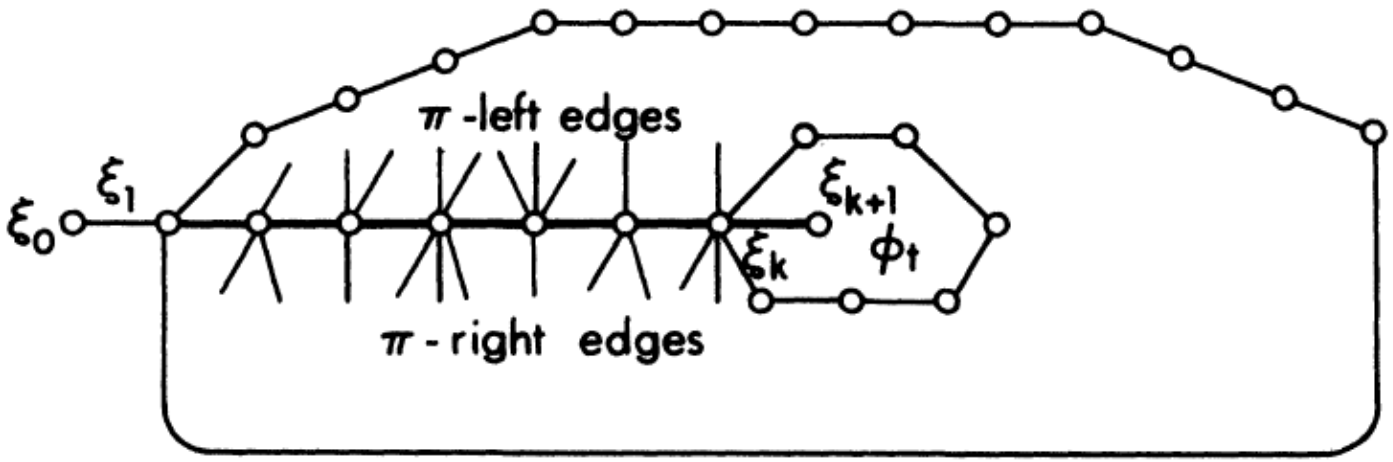
Minimum-length Cycle

Let C be a minimum length (s,t)-cut. Then $C^d = \{\alpha^d \mid \alpha \in C\}$ is a cycle of minimum length around φ_t.

- If we find this cycle, we are done
- We choose $\xi_s \in \phi_s$ and $\xi_t \in \phi_t$
- Let $\Pi = (\xi_1 = \xi_s, \xi_2, \ldots, \xi_k = \xi_t)$ be a shortest path from ξ_s to ξ_t
- Every edge in the dual has an associated weight, the capacity of the edge it 'cuts'; we can order the edges linearly based on wieght

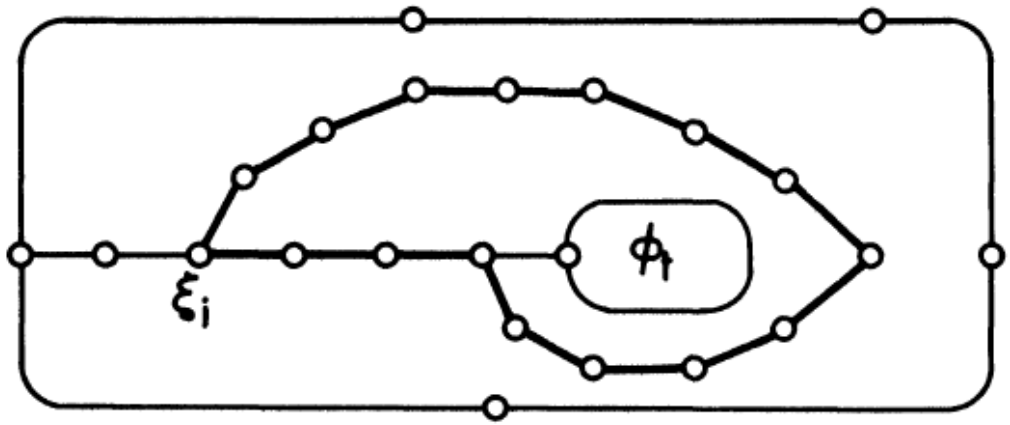
Min-length Cycle

- We define left-edges and right-edges that lead into a ξ_i based on linear ordering of α (the weights of edges)
- Add additional ξ_0 and ξ_{k+1} to ensure that the definition holds for start and end nodes



Min-length Cycle

- A ξ_i-cycle is a cycle such that one left and one right edge is used, every ξ_i-cycle encloses φ_t
- Every min-length cycle must intersect Π, and shortest distance between points is on Π (this argument does not work in directed graphs)
- Thus, every min-length cycle is a ξ_i-cycle



Finding a Min-Length ξ_i-cycle

- To find such a cycle, direct left edges outward and right edges inward; for other edges, add both directed edges
- Now, find every shortest (ξ_i, ξ_i)-path
- This path is a ξ_i-cycle, since it can only leave via a left edge and come in via a right edge (or a Π-edge)
- We can find such a path in $O(n \log n)$ time
- The algorithm requires at most $O(n^2 \log n)$ time

Conclusion

- The authors implement the algorithm and find that it works better than Dinic's algorithm and equal to Bergen's algorithm for graphs which require the full $O(n^3)$ of Dinic's algorithm
- For random graphs, the runtime is about the same
- They find maximum flow and minimum cut in $O(n^2 \log n)$
- The min-cut for directed graphs is not found