Max flow min cut in undirected planar graphs

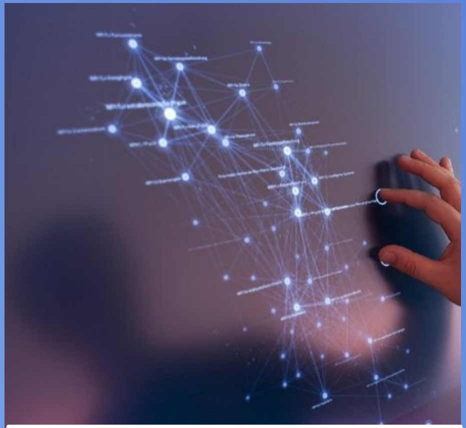
Kiril Mitev

Max flow min cut in undirected planar graphs

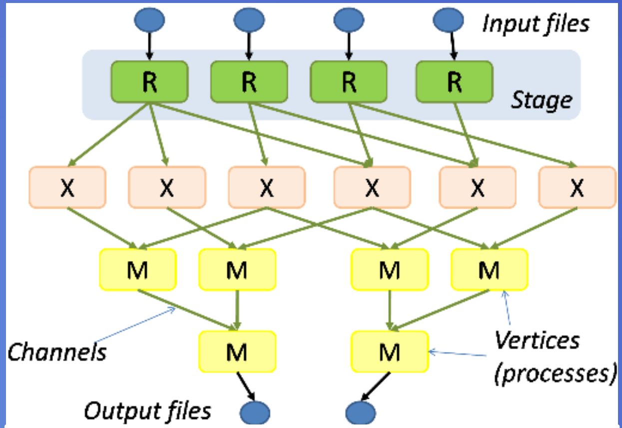
- Introduction and motivation
- Cuts and min cuts
  - Definitions
  - Algorithm
    - Reif's Algorithm
  - Complexity
- Flows
  - Cuts as upper bound
  - Feasible flows
  - St-planar graphs
  - Flows in general undirected graphs
    - From max flow to shortest path problem
- References

Applications

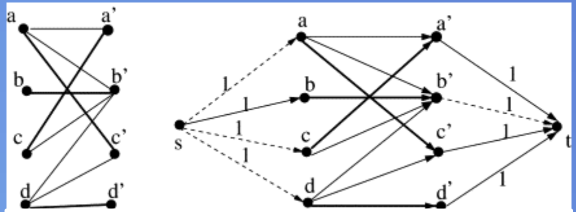
- Max flow and min cut: Two very rich algorithmic problems (cornerstone problems)
- Problems with reductions to flow/cut:
  - Network connectivity
  - Bipartite matching
  - Airline scheduling
  - Image processing
  - Distributed computing
  - Traffic control
  - Design of communication networks
  - Routing of VLSI circuits (very large scale integration)
    Integrating trasistors into a circuit


Network connectivity

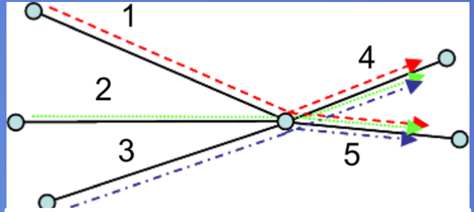

Airline scheduling



Distributed computing

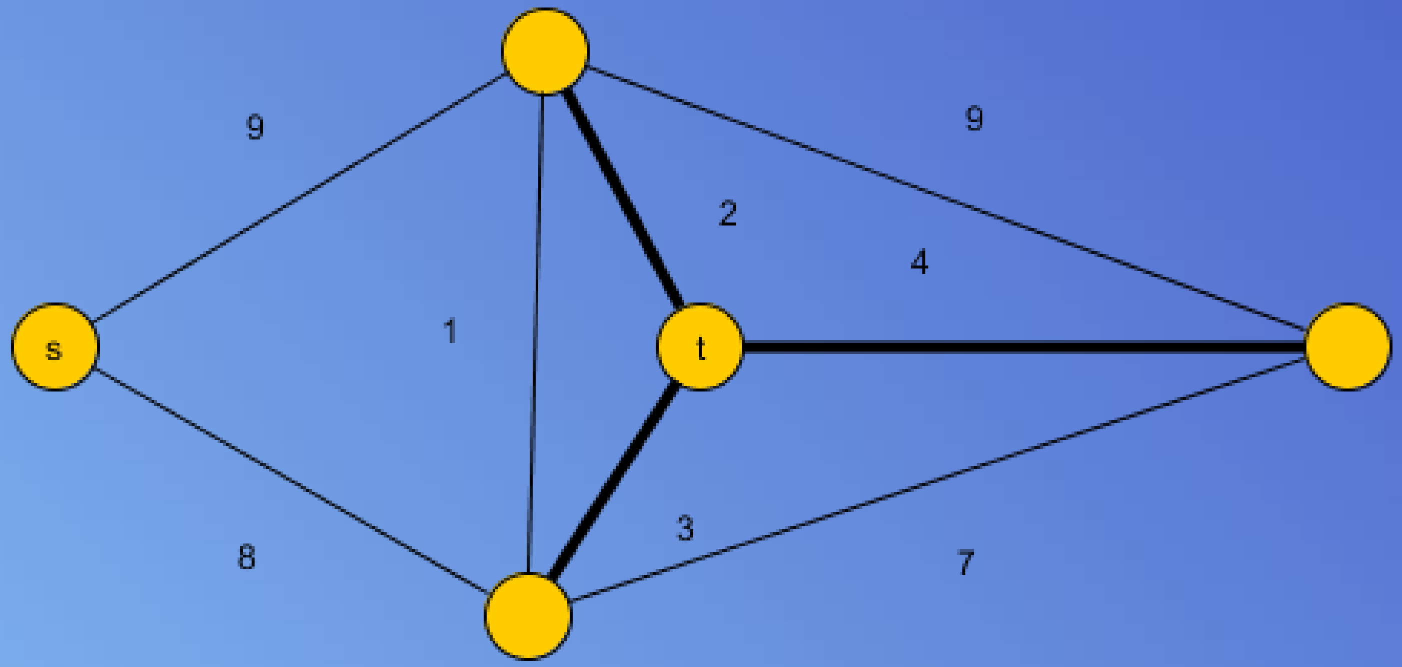


Bipartite matching



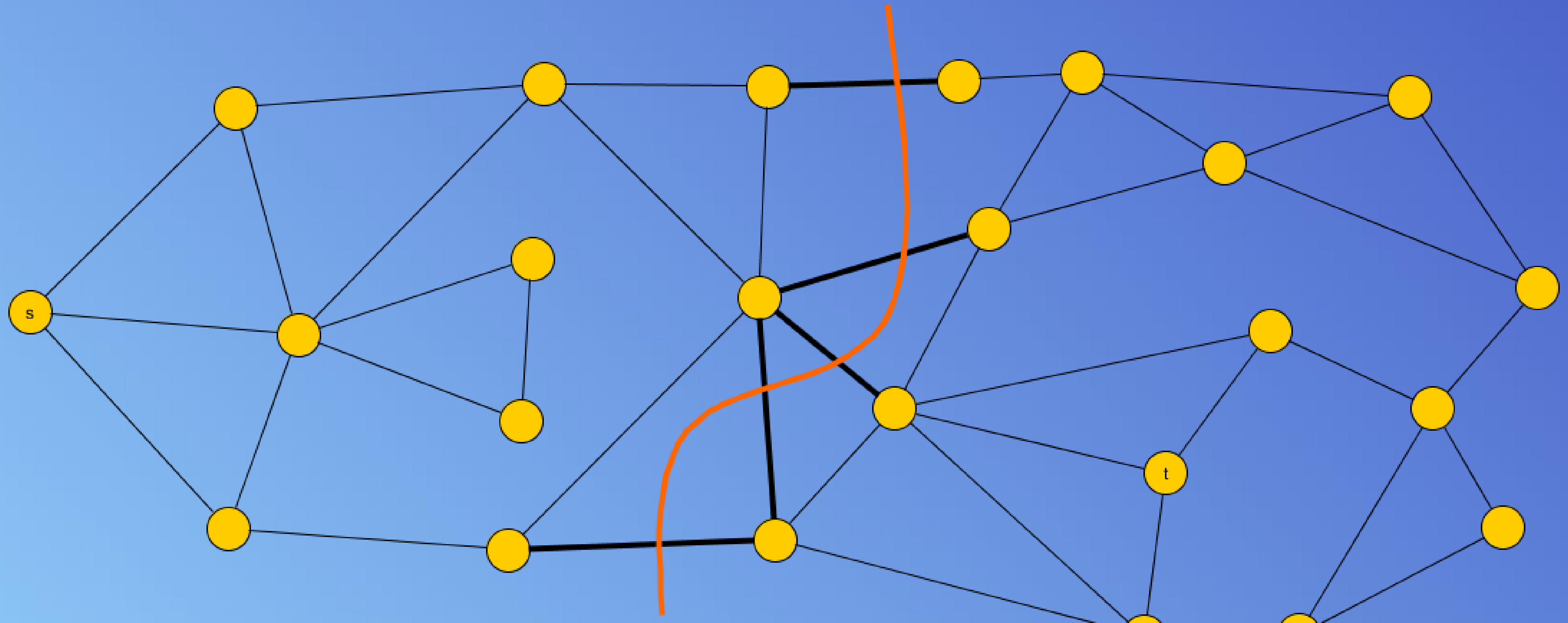
Traffic control

Cuts



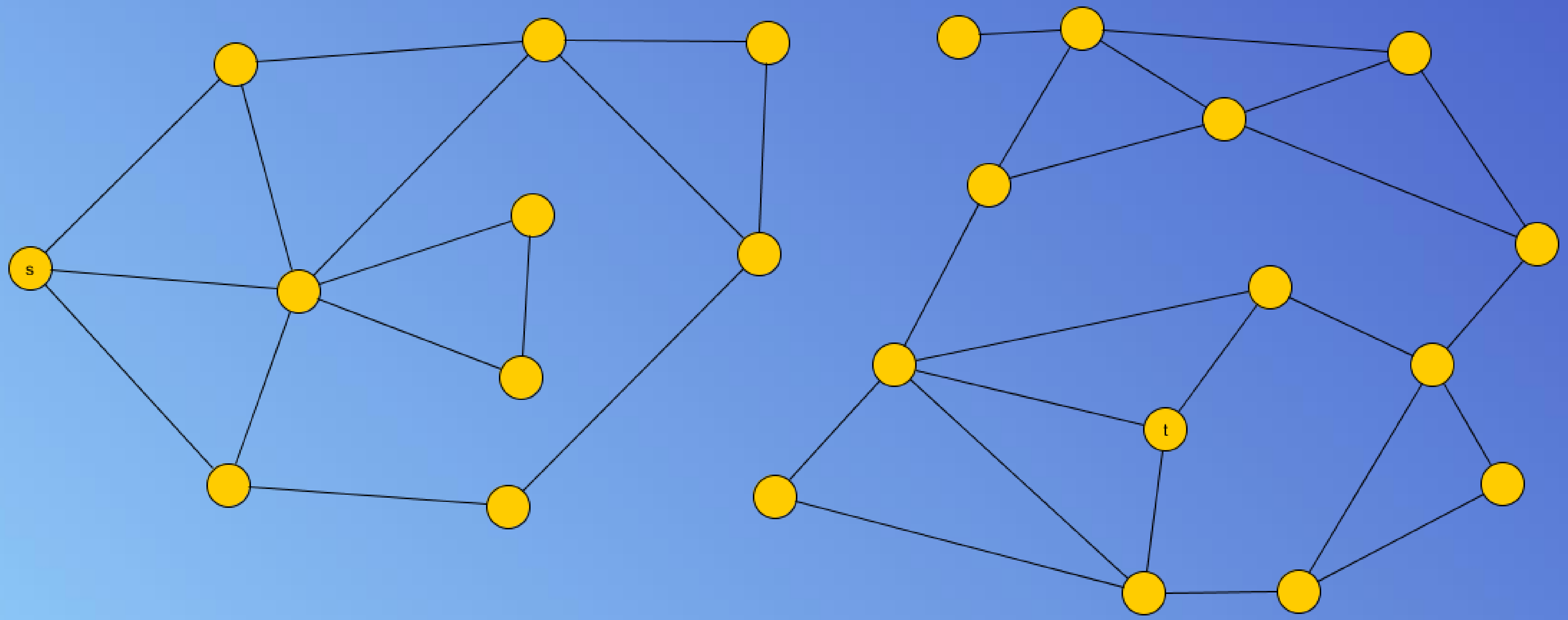
- Edge set $OUT(E(A))$ separating G into two connected components $A, B \in V, s \in A, t \in B$
- Each st path uses one of these edges
- Min st cut = min capacity

Cuts



- Cuts and min cuts -> Definitions

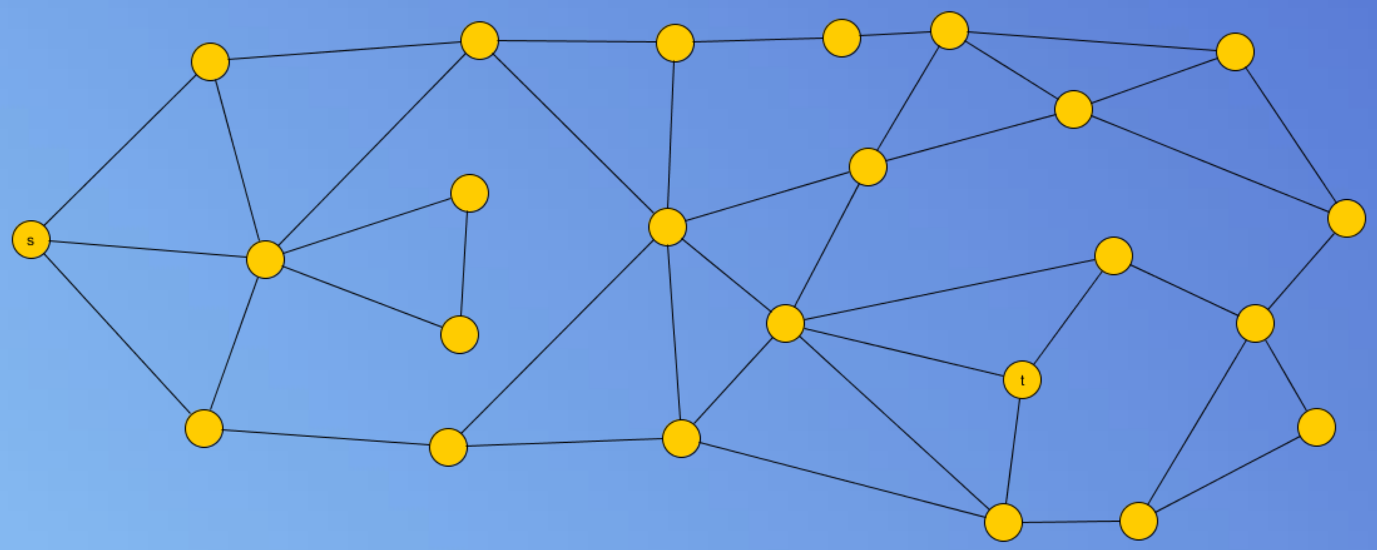
Cuts



- Cuts and min cuts -> Definitions

Min st cut

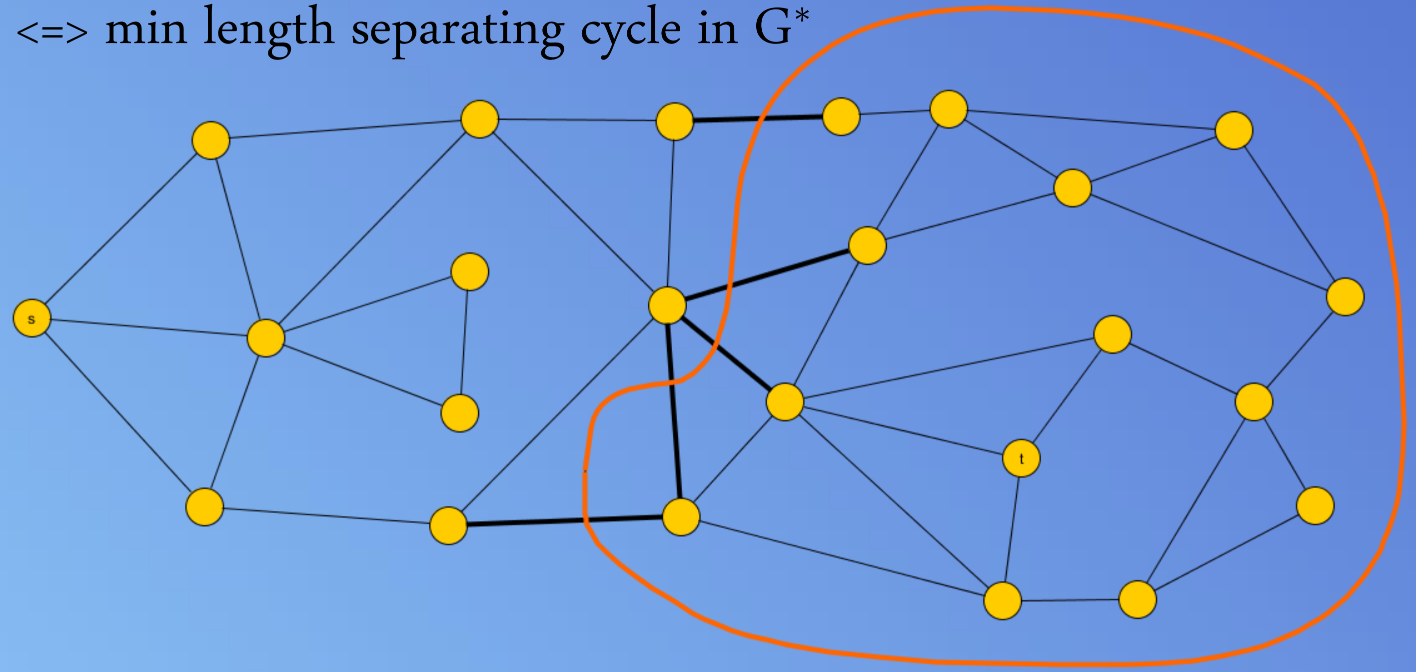
- Best known algorithm for planar graphs: $O(n * \log\log n)$
- Idea: use dual graph G^*, search for separating cycle



Min st cut

Separating cycle

- Dual of st cut is a cycle separating s and t
- Min st cut in G <=> min length separating cycle in G*

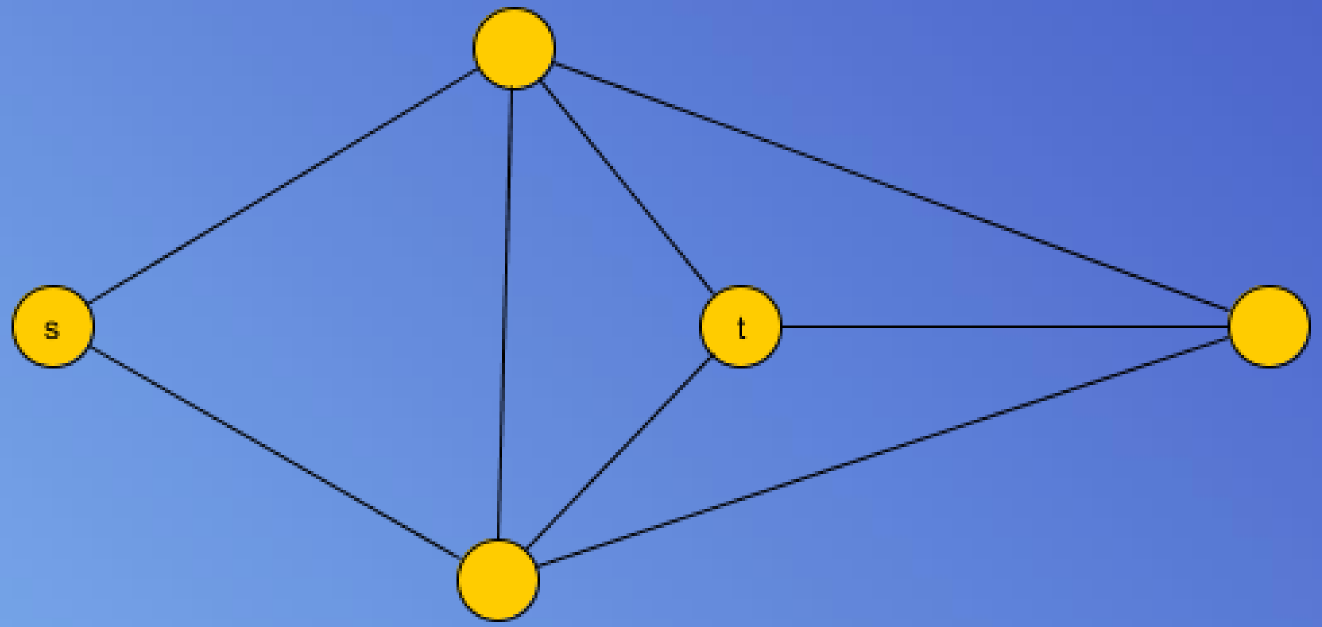


- Cuts and min cuts -> Separating cycle

Min st cut

Dual graph G*

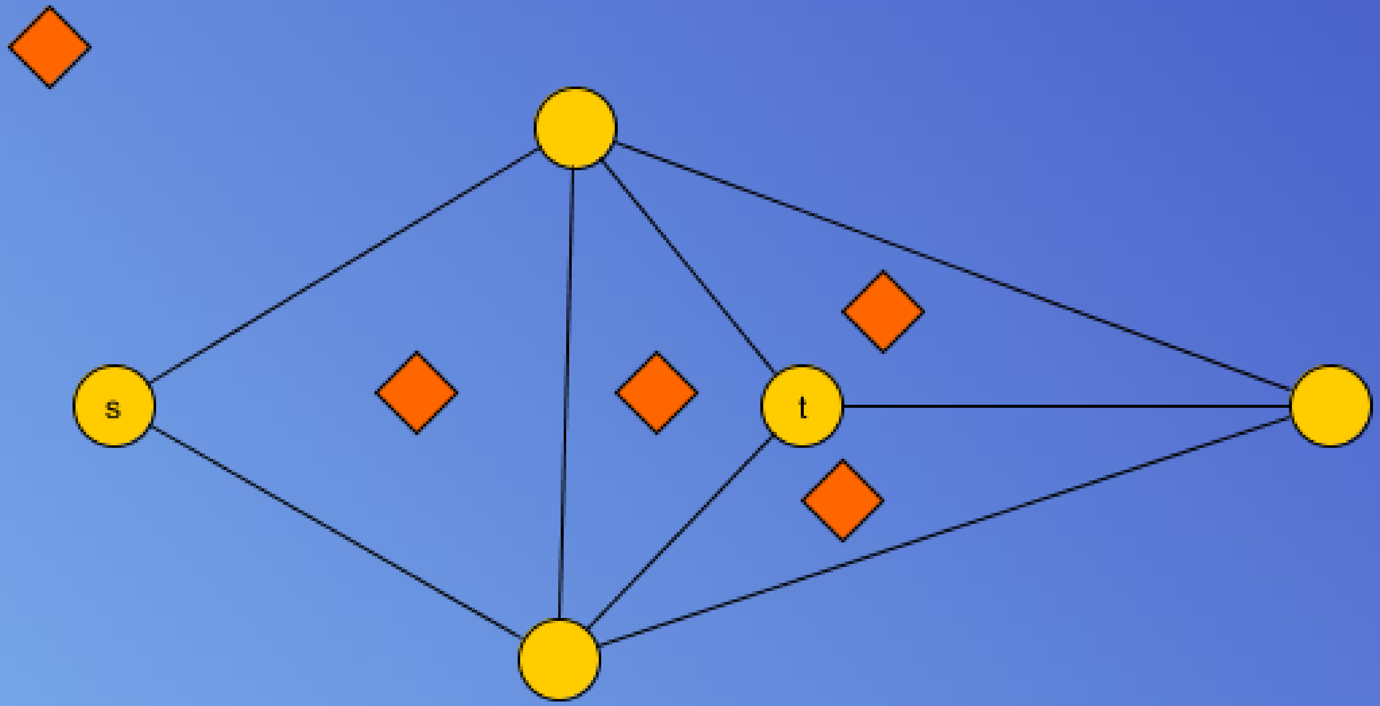
- Each face becomes a vertex



Min st cut

Dual graph G*

- Each face becomes a vertex

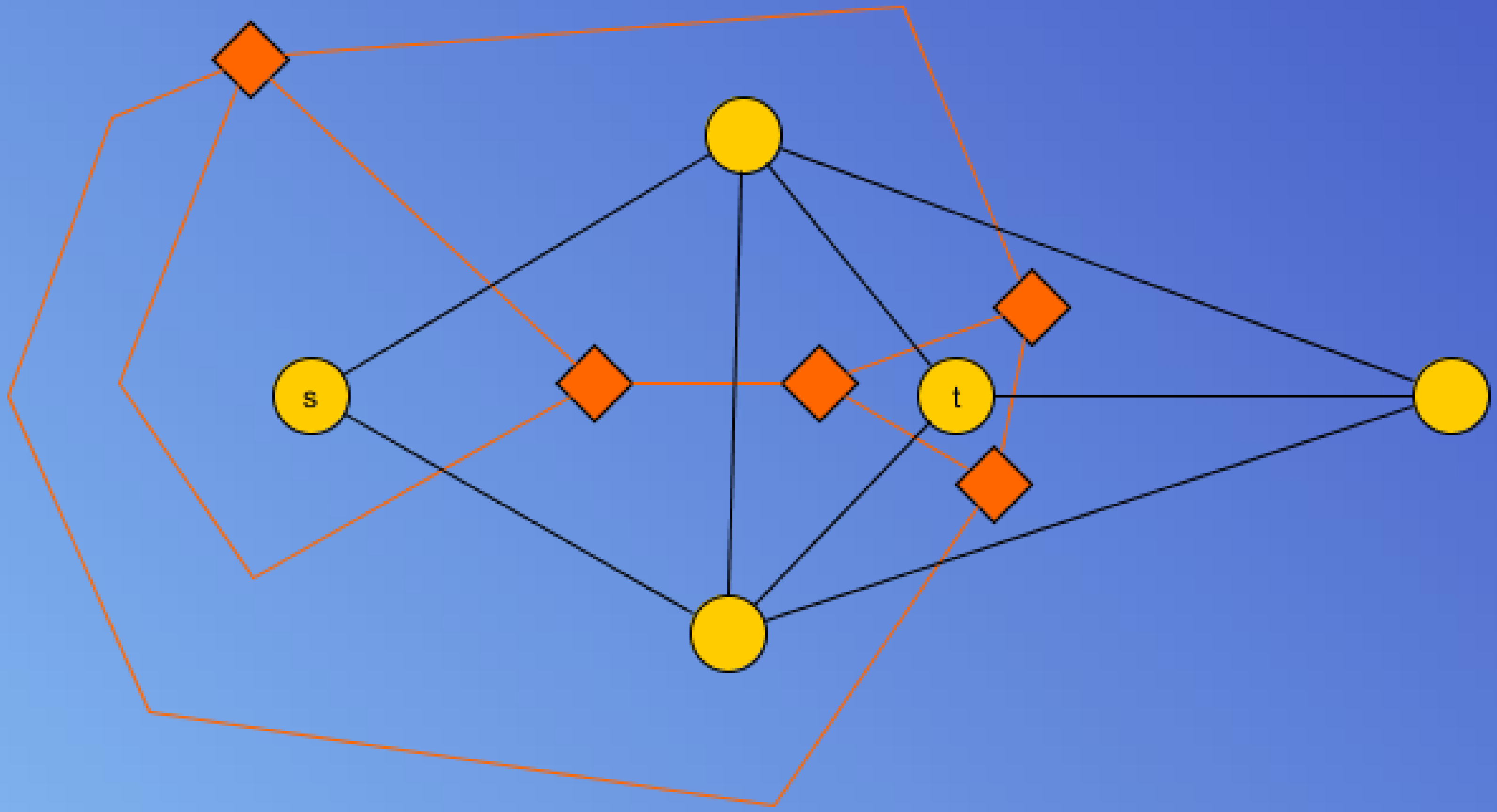


- Cuts and min cuts -> Dual graph

Min st cut

Dual graph G*

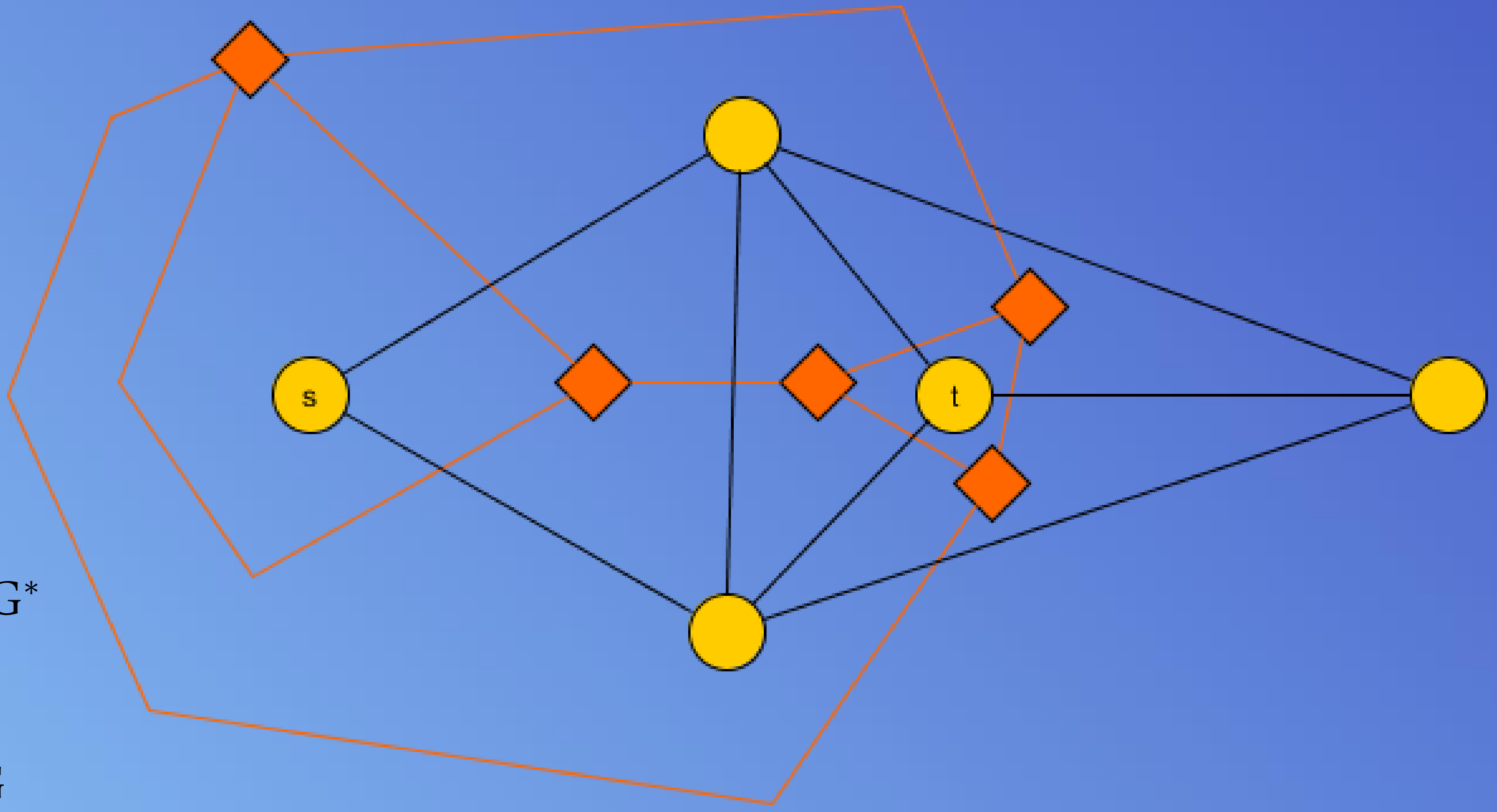
- Each face becomes a vertex
- Dual e* of e
  connects faces adjacent to e
  Length $l(e^*) = c(e)$



- Cuts and min cuts -> Dual graph

Min st cut

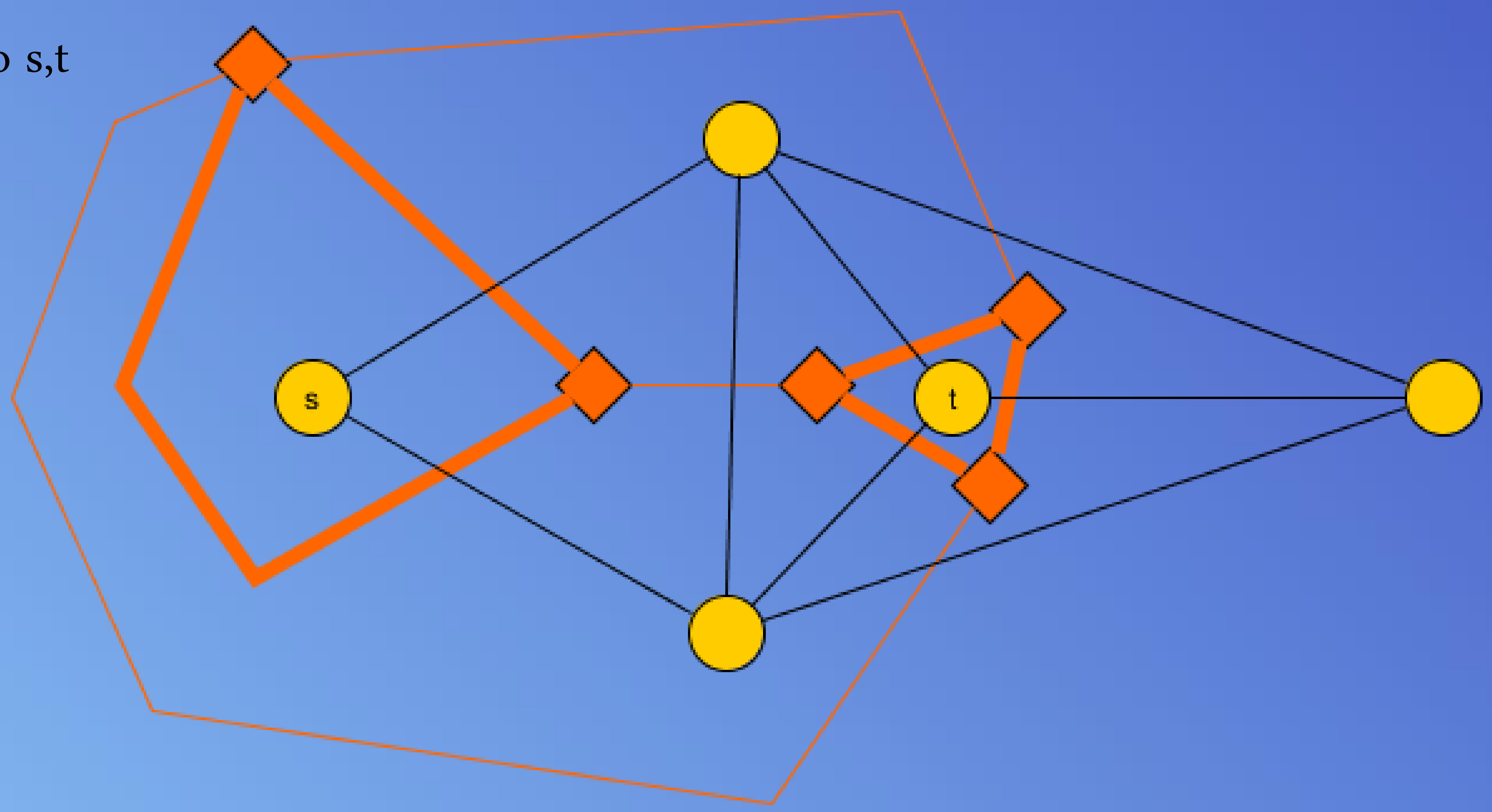
Dual graph G*



- One-to-one correspondence:
  - V and Φ Φ: Faces of G*
  - E and E*
  - V* and F   F: Faces of G

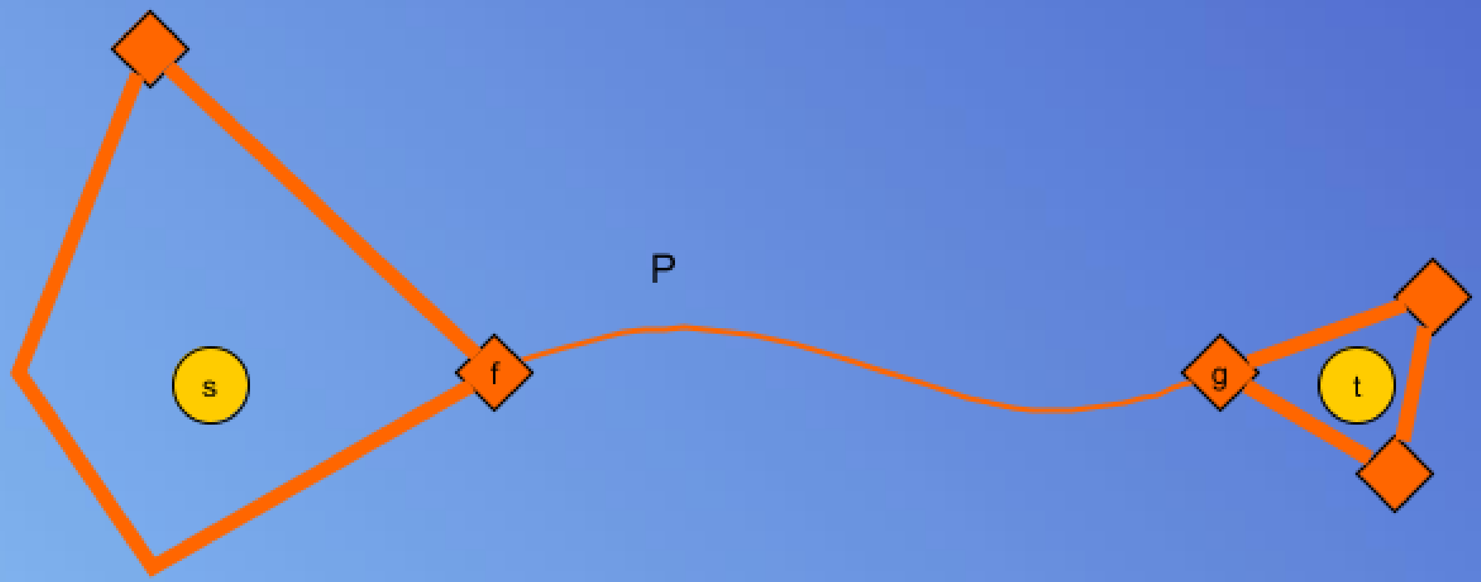
Min st cut - Algorithm

1. Let f,g be faces incident to s,t



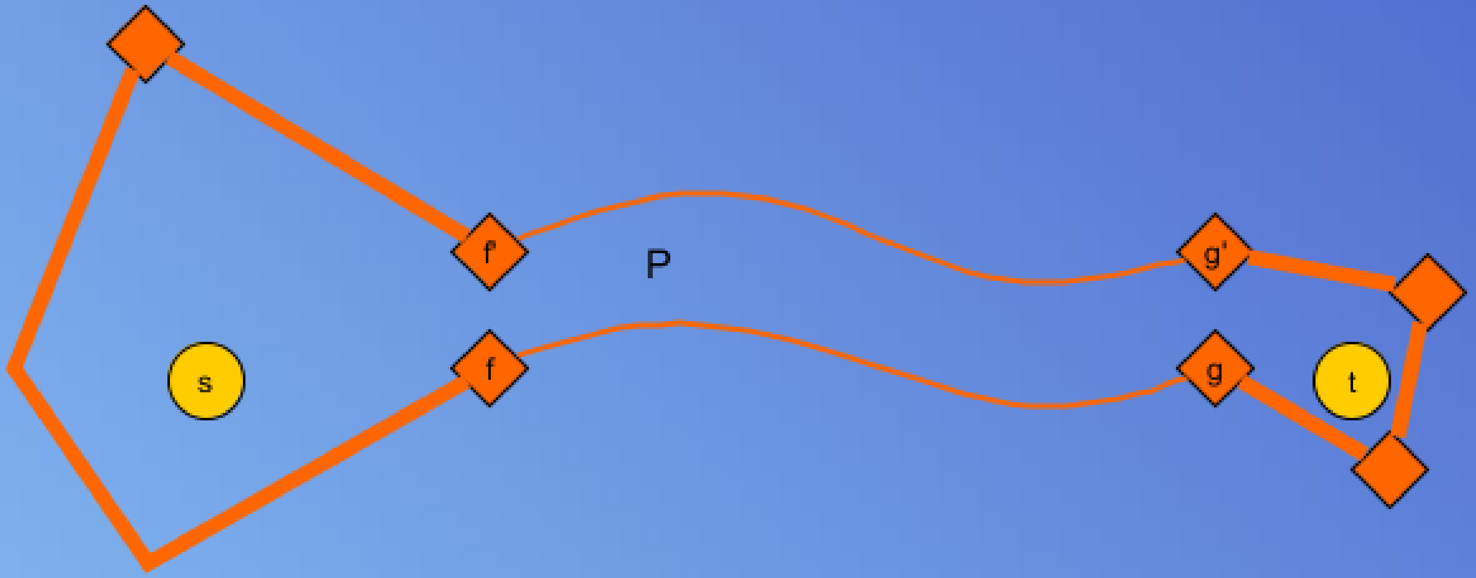
Min st cut - Algorithm

1. Let f,g be faces incident to s,t
2. Compute SP P in G^* from f to g



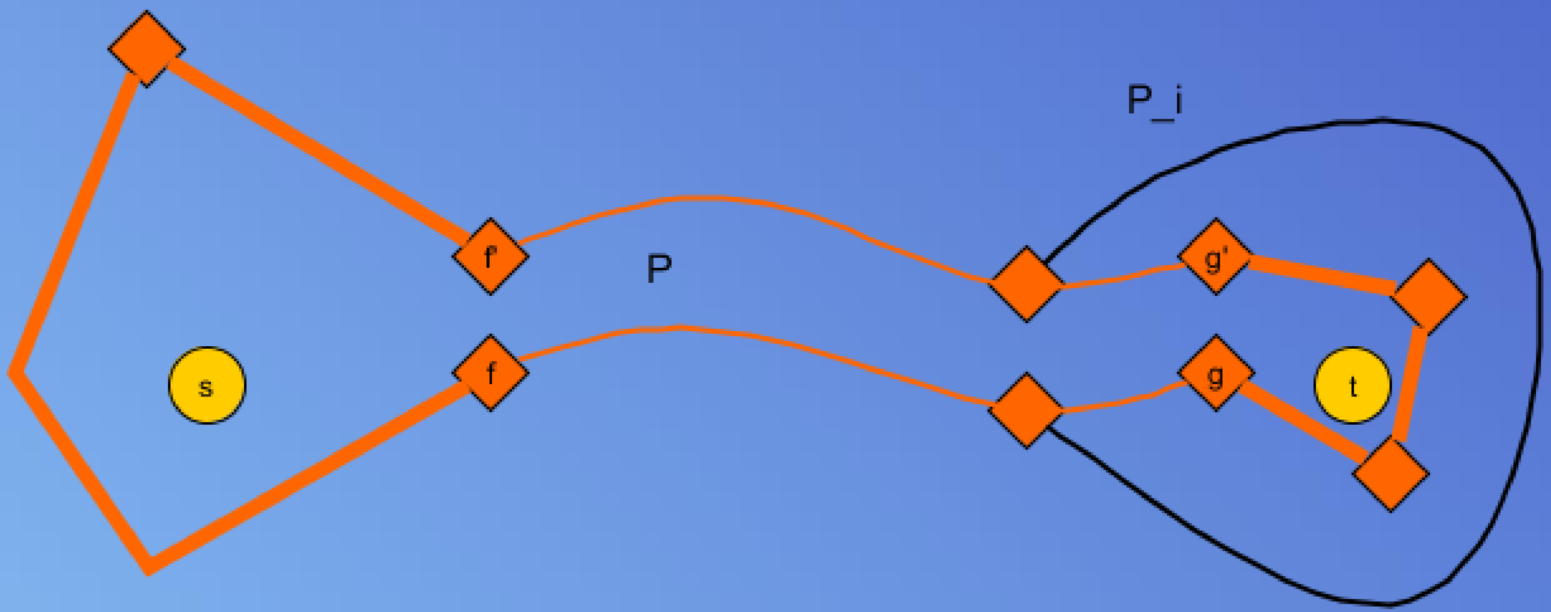
Min st cut - Algorithm

1. Let f,g be faces incident to s,t
2. Compute SP P in G^* from f to g
3. Cut G^* open along P



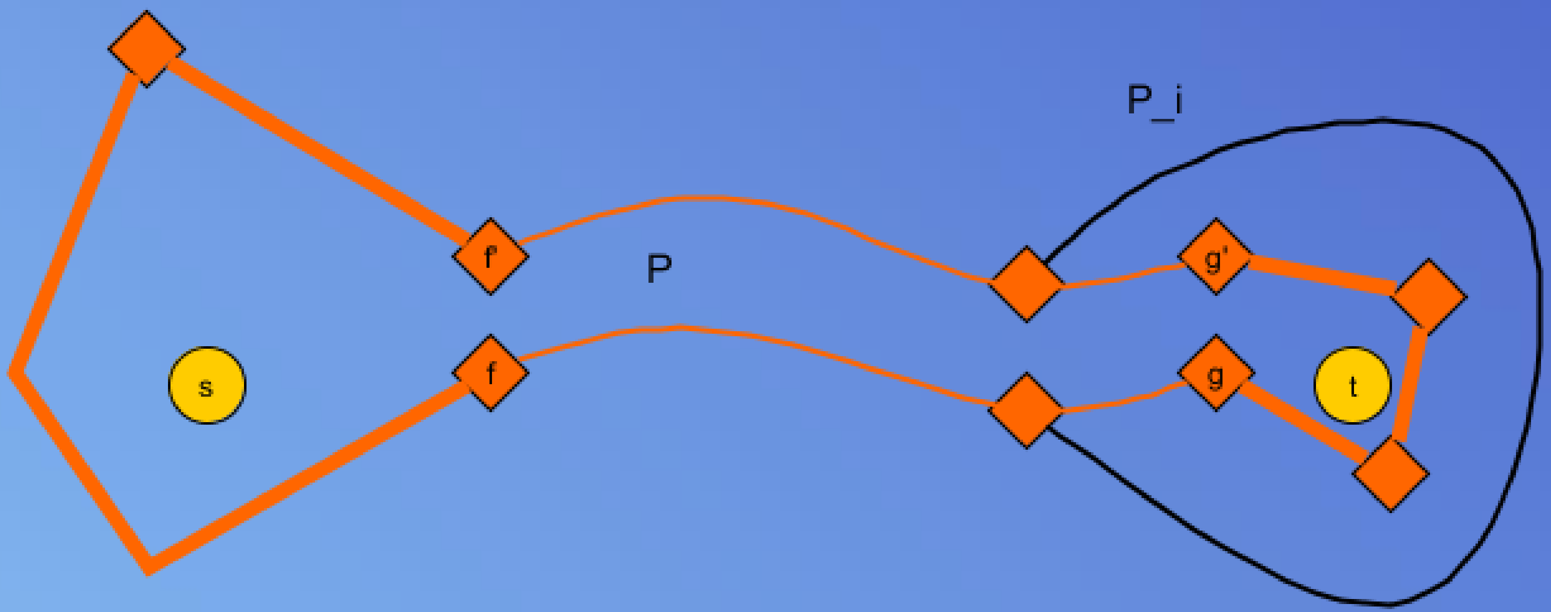
Min st cut - Algorithm

1. Let f,g be faces incident to s,t
2. Compute SP P in G^* from f to g
3. Cut G^* open along P
4. Compute SP P_i for every pair of copies of nodes of P in resulting graph



Min st cut - Algorithm

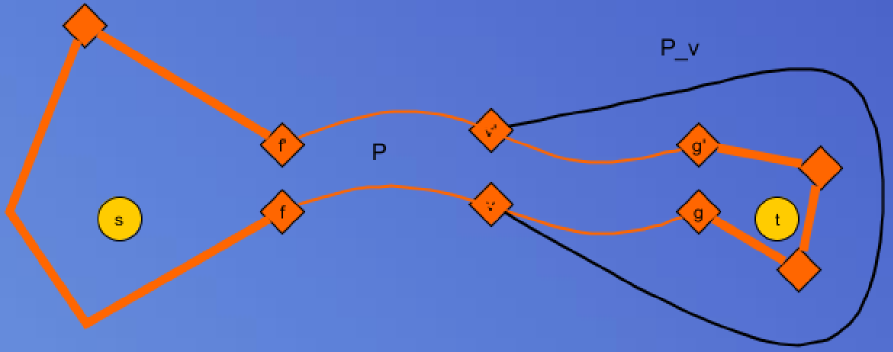
1. Let f,g be faces incident to s,t
2. Compute SP P in G^* from f to g
3. Cut G^* open along P
4. Compute SP P_i for every pair of copies of nodes of P in resulting graph
5. Return $\min P_i$



Min st cut – Reif's Algorithm

Reif [1983]

- Start with the middle vertex v of P
- Divide and Conquer

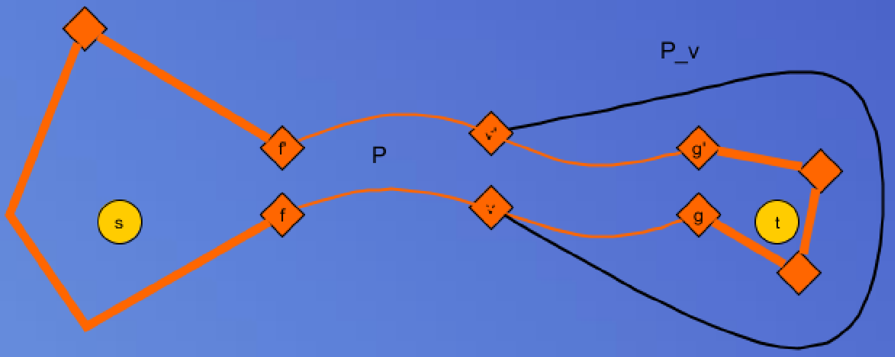


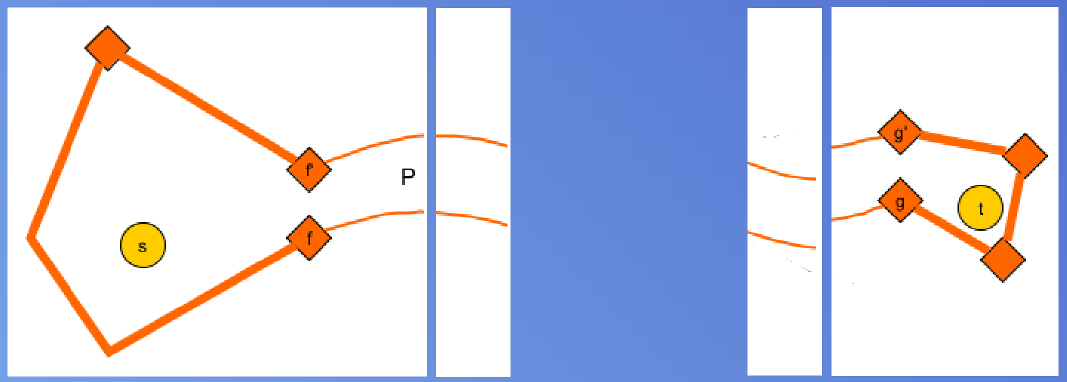
Min st cut – Reif‘s Algorithm

Reif [1983]

- Start with the middle vertex v of P
- Divide and Conquer

- Total time: $O(n * \log n)$
  - Recursion depth: $O(\log n)$
  - SP Algorithm for planar graphs:
    $O(n)$ or $O(n * \log n)$





Min st cut - Complexity

- Min P_i = min separating cycle = min cut = max flow
- Complexity
  1. Time for computing SP P: $O(n)$
  2. Time for computing SP P_i:
     - 1983 Reif's recursive algorithm – divide and conquer: $O(n * \log n)$
     - 2005 MSSP – modified successive shortest path: $O(n * \log n)$
     - Best known uses r-decompositions and FR-Dijkstra: $O(n * \log\log n)$ by Italiano, Nussbaum, Sankowski and Wulff Nilsen

Flows

- Single- and mulicommodity flows
- Best single-commodity algorithm for planar graphs: Sleator and Tarjan $O(n * \log n)$

Input: Flow network N = (G, P, c)

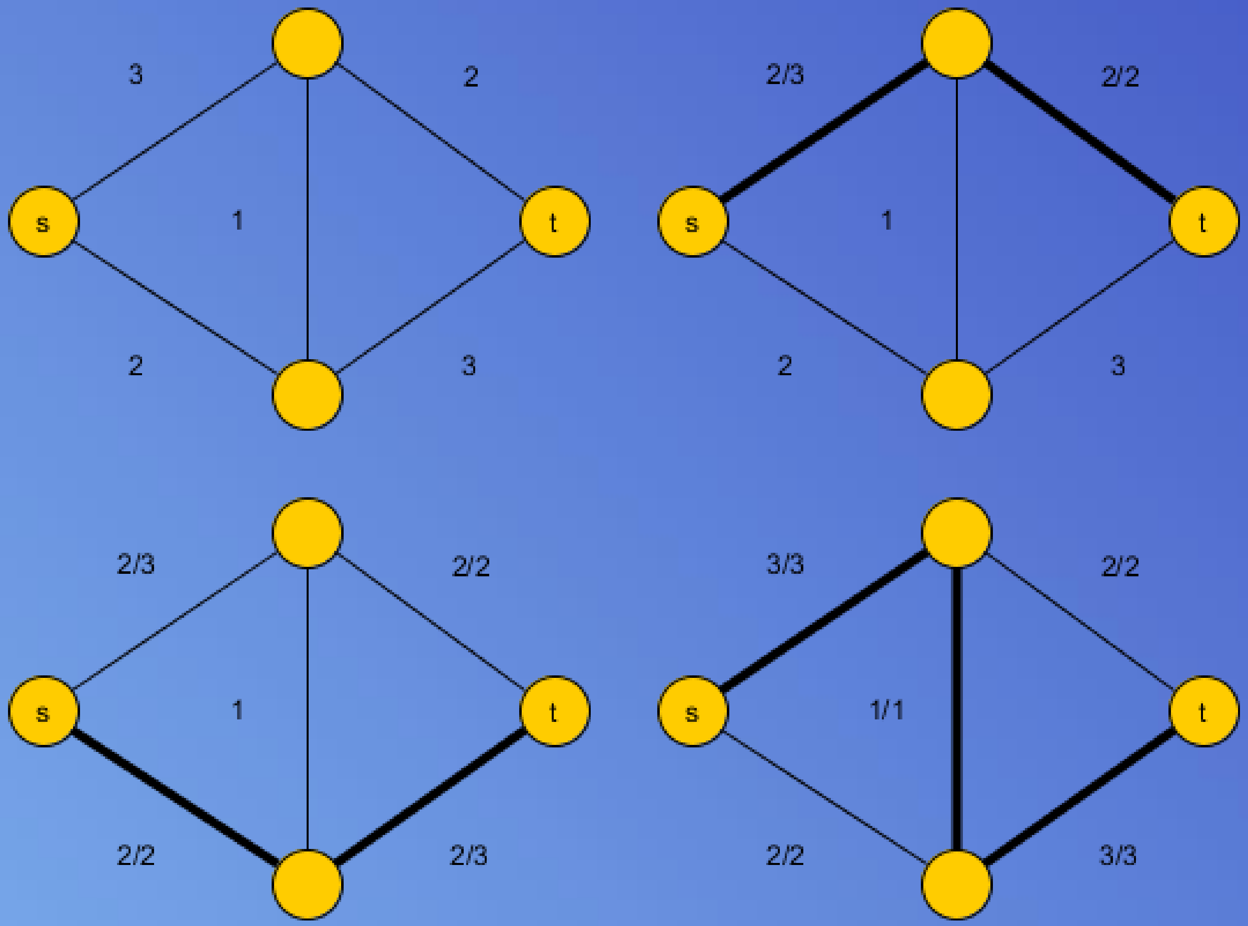
- G = (V, E)
- P: set of source-sink pairs (s_i, p_i)
- c: capacity function

Output: An st flow of max value

Flows

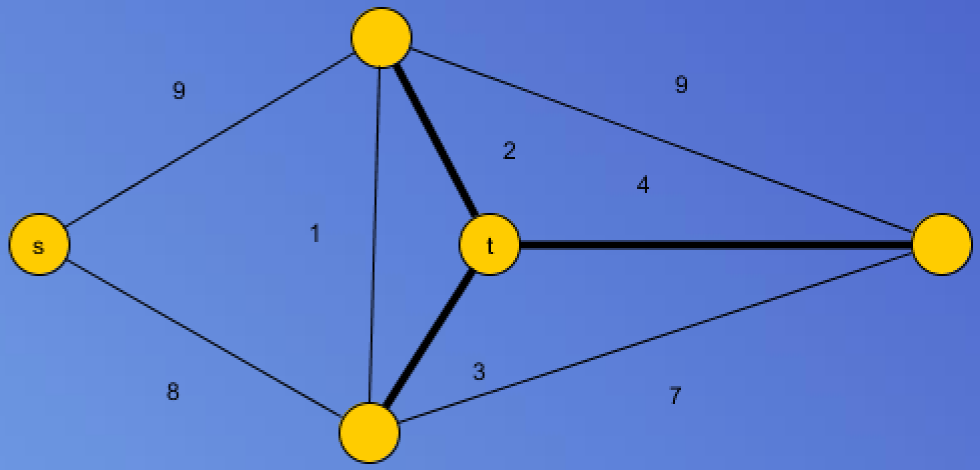
Ford-Fulkerson Algorithm

1. Initialize zero flow
   Initialize residual graph G‘
2. While (Augmenting path P in G‘)
   1. Determine bottleneck b of P
   2. Increase flow along P by b
   3. Update residual graph G‘



Flows

- Max st flow uses (at most) all edges of s-t cut
- Max st flow bounded by min s-t cut
- 1956 Ford and Fulkerson proof equality



- Flows -> Cuts as upper bound

Feasible flows

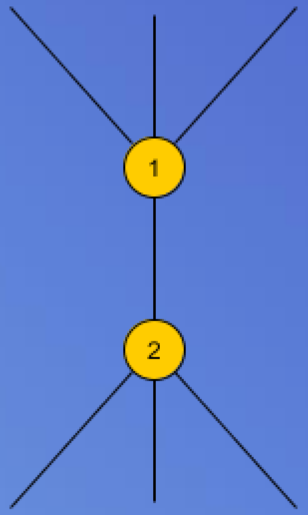
- Respect capacities:

  $f(e) \leq c(e) \quad \forall e \in E$

- Satisfy the flow conservation rule:

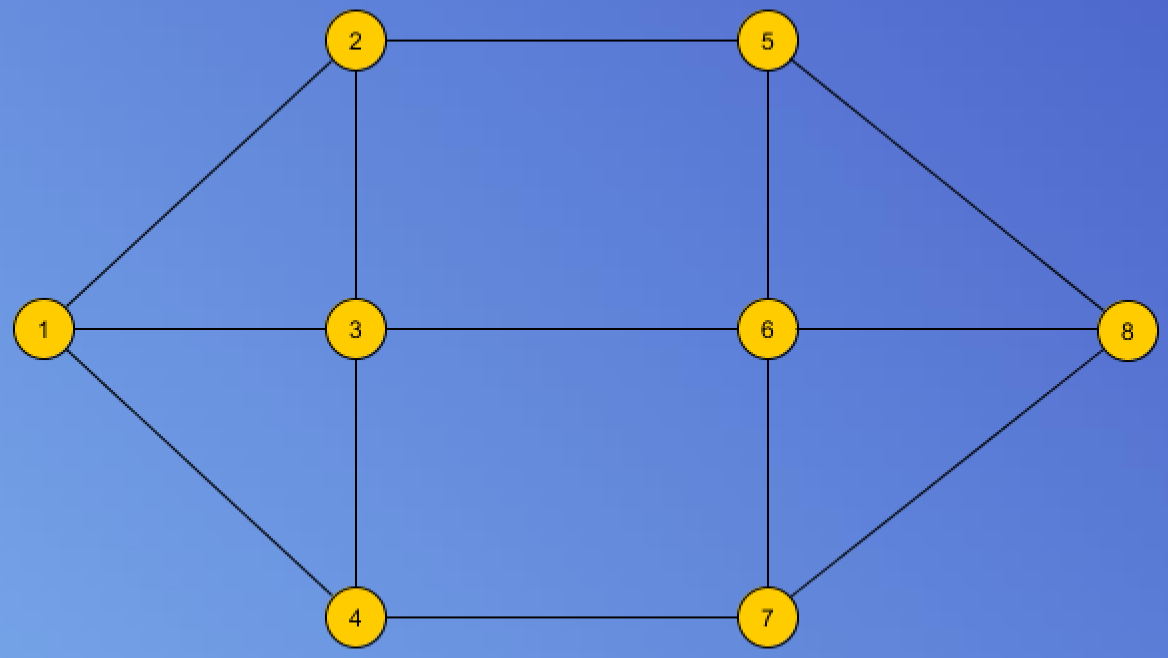
$$\sum_{e \in \delta^+} f(e) - \sum_{e \in \delta^-} f(e) = \begin{cases} -v, & i = s \\ 0, & i \neq s, t \\ v, & i = t \end{cases}$$

- Can be tested in $O(n^2 * \log n)$



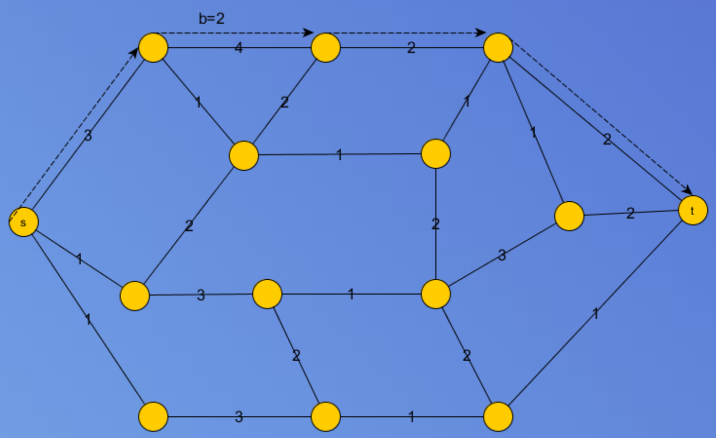
st-planar graphs

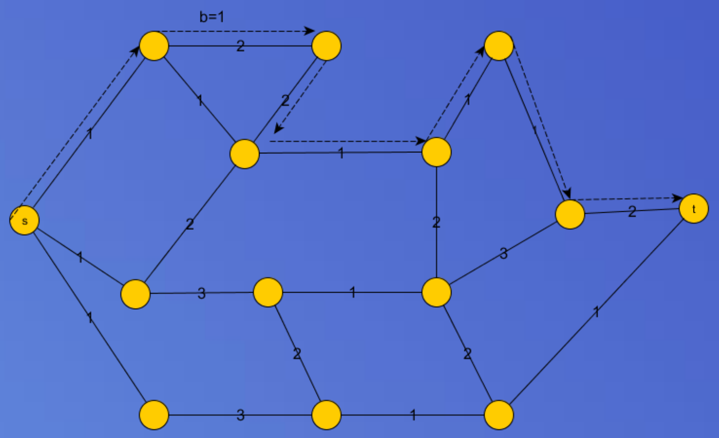
- Graph is st-planar if s and t both lie on the outer (unbounded) face
- St-planar for s=1 and t=8
- Not st-planar for s=1 and t=6



st-planar graphs – Uppermost path

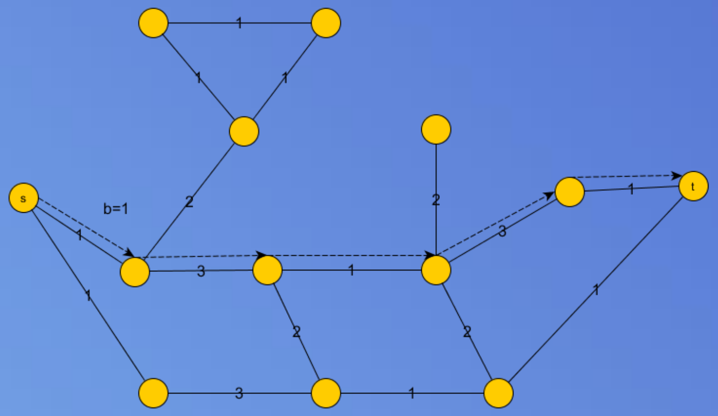
- Initialize
  - Start with zero flow
    $\forall e \in E \; set \quad f(e) = 0$
- Find the uppermost path
  if none exists then stop

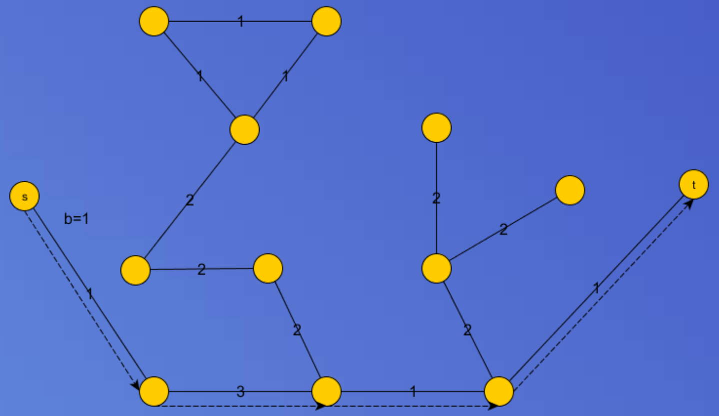


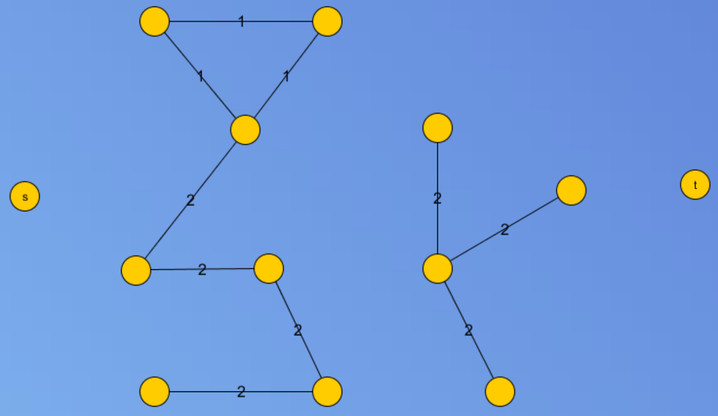


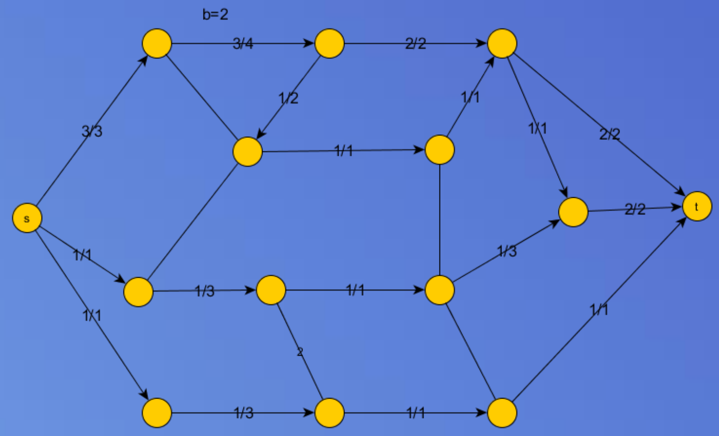
st-planar graphs – Uppermost path

- Initialize
  - Start with zero flow
    $\forall e \in E \; set \quad f(e) = 0$
- Find the uppermost path
  if none exists then stop
- Let $b = \min\{c(e) : e \in P\}$
- Increase the flow by b units along P
- Decrease capacities
- Delete edges of zero capacity



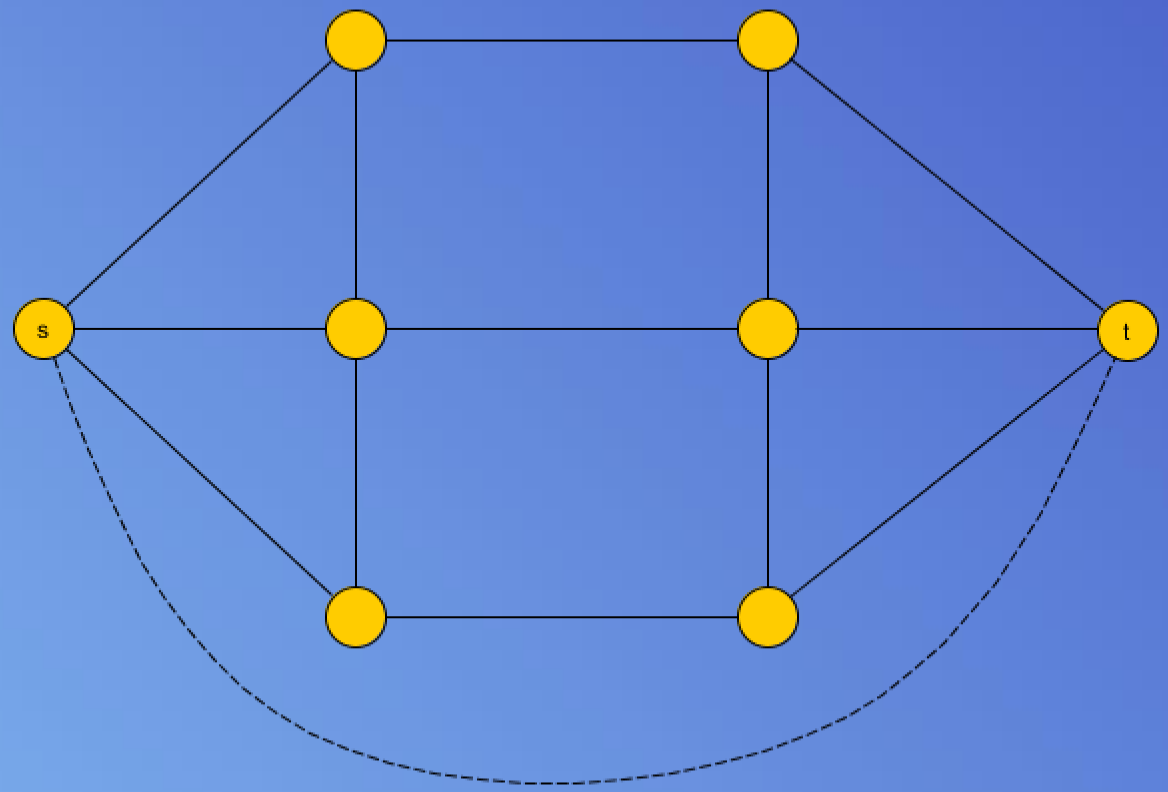






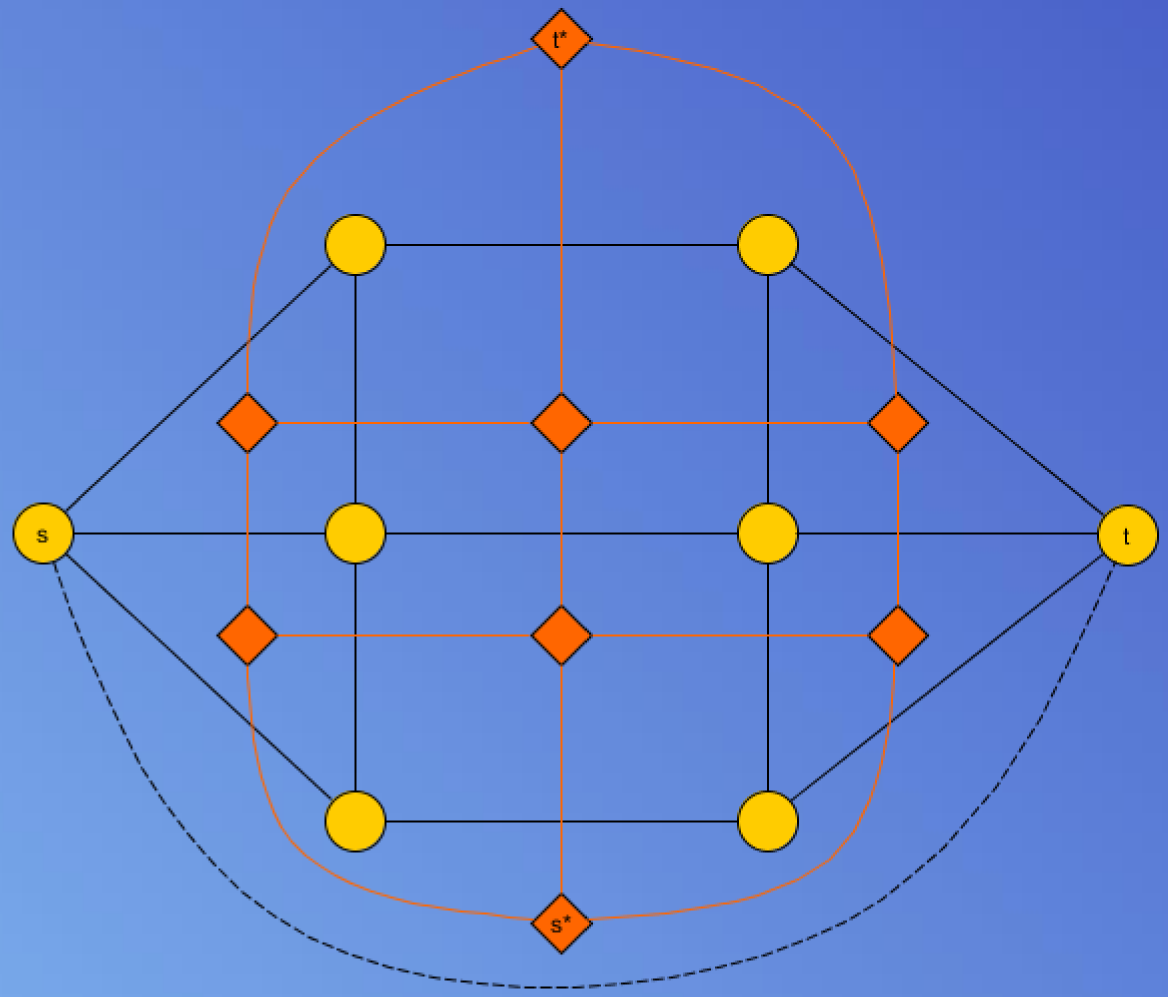
st-planar graphs - Algorithm

- Add edge (s, t) to E



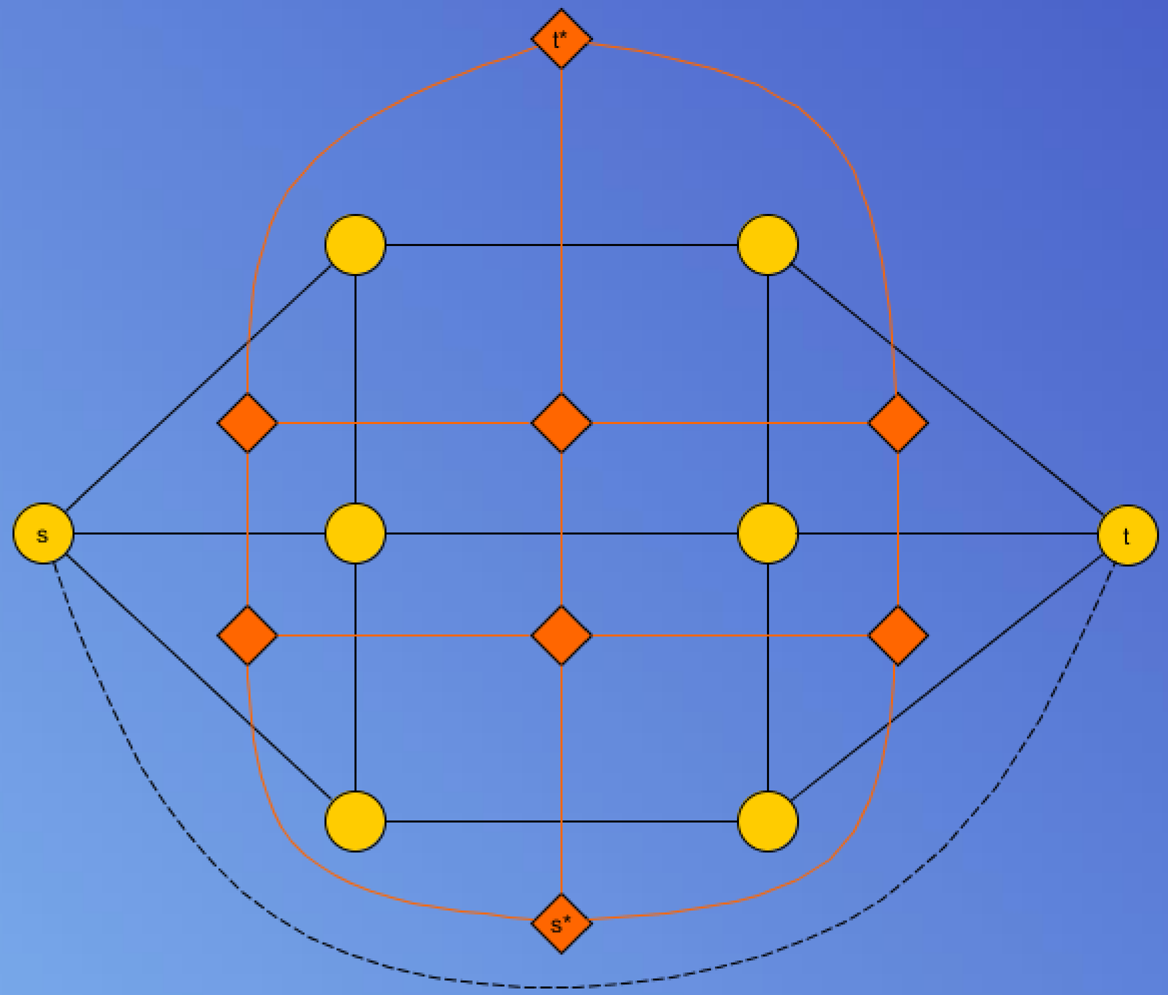
st-planar graphs - Algorithm

- Add edge (s, t) to E
- Construct Dual G^*
  - The new face is s^*
  - The unbounded face is t^*
  - No need for dual edge (s^*, t^*)



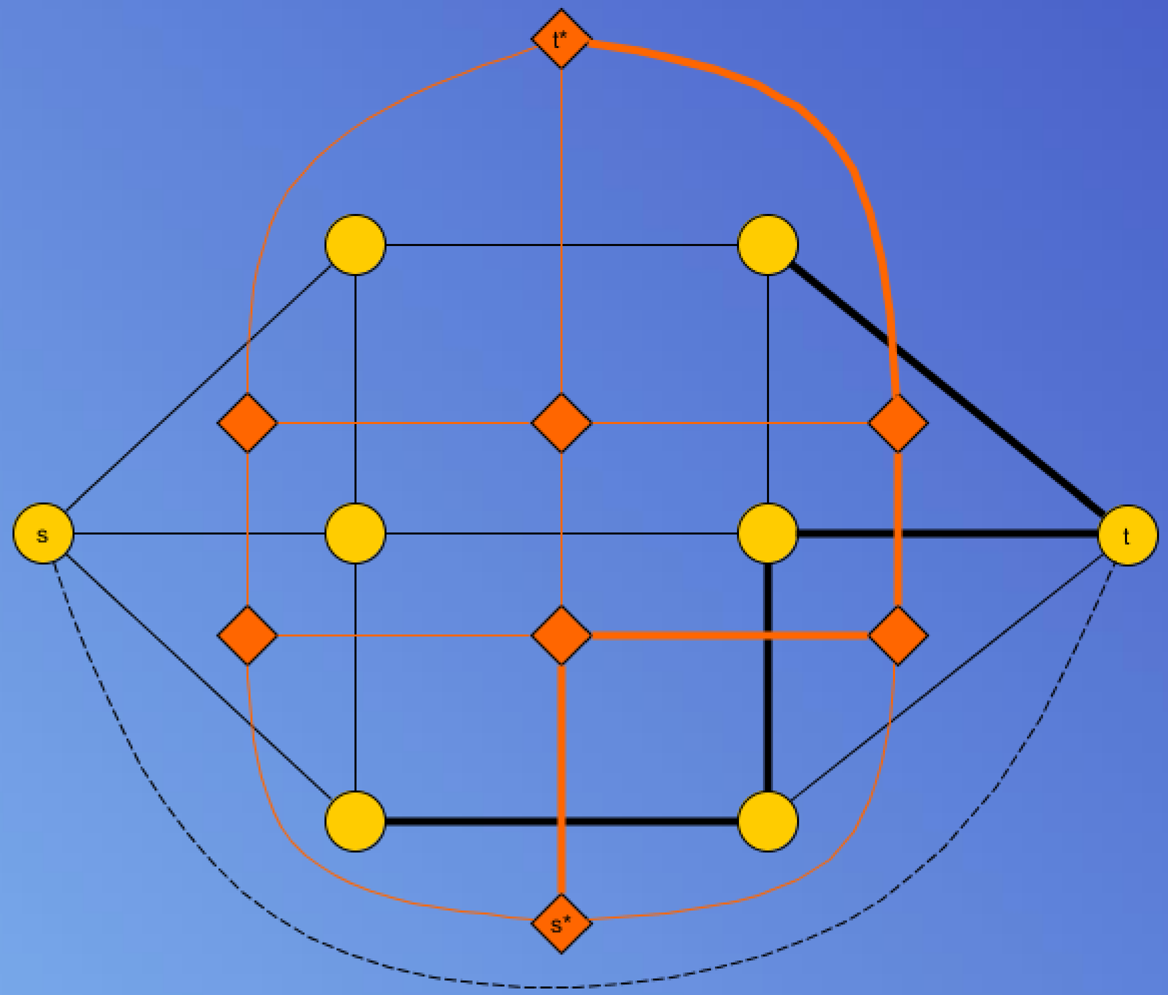
st-planar graphs - Algorithm

- Add edge (s, t) to E
- Construct Dual G^*
  - The new face is s^*
  - The unbounded face is t^*
  - No need for dual edge (s^*, t^*)
- Length $l(e^*) = c(e)$



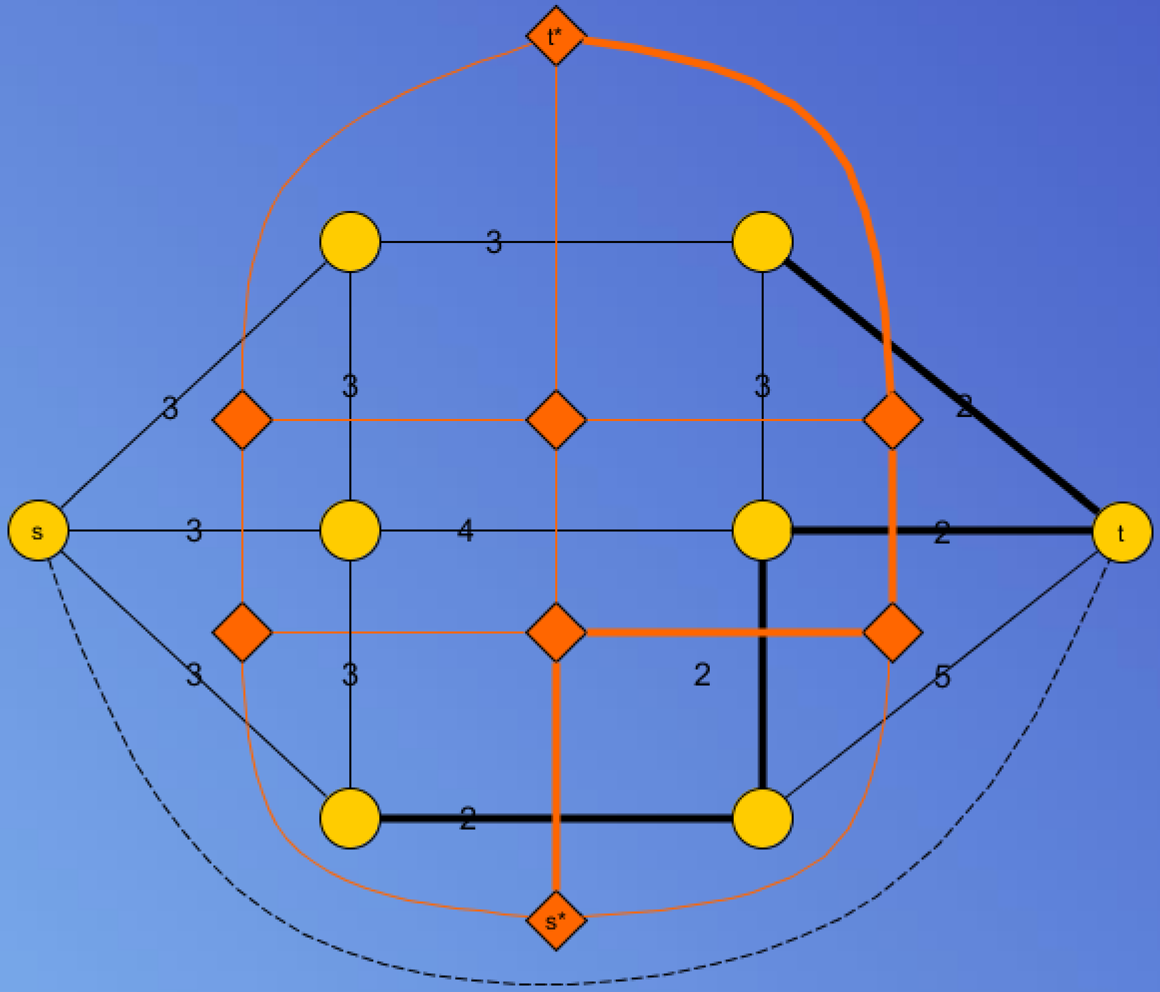
st-planar graphs - Algorithm

- Add edge (s, t) to E
- Construct Dual G^*
  - The new face is s^*
  - The unbounded face is t^*
  - No need for dual edge (s^*, t^*)
- Length $l(e^*) = c(e)$
- An st cut in G corresponds to an s^*t^* path in G^*



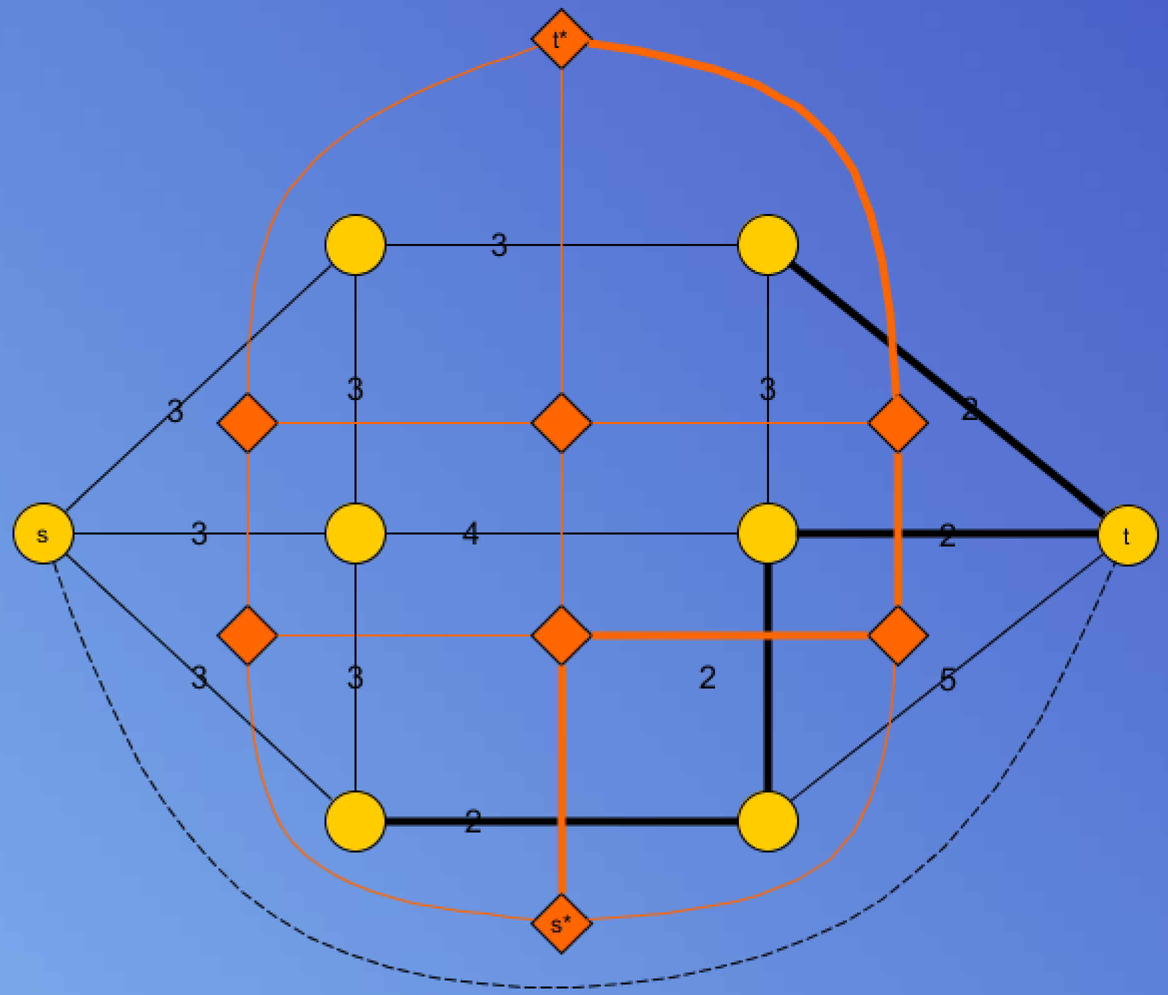
st-planar graphs - Algorithm

- Thus, min cut can be computed by computing a shortest path in G^*
- Motivation for adding extra node s^* is to convert a cycle problem into a path problem
- The cut does not by itself give the max flow



st-planar graphs - Algorithm

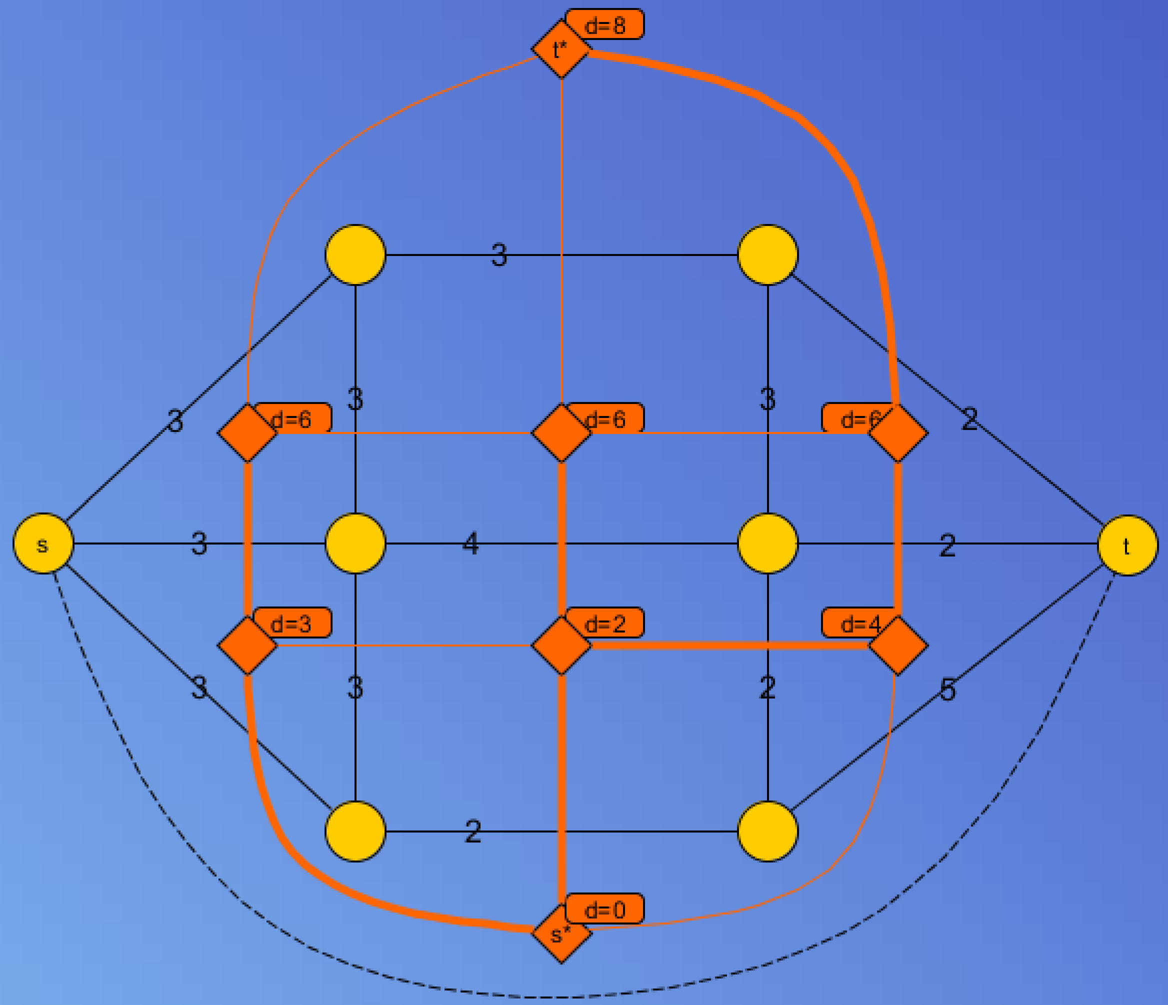
- Thus, min cut can be computed by computing a shortest path in G^*
- Motivation for adding extra node s^* is to convert a cycle problem into a path problem
- The cut does not by itself give the max flow



- SP distances in G^* can be used to obtain the max flow

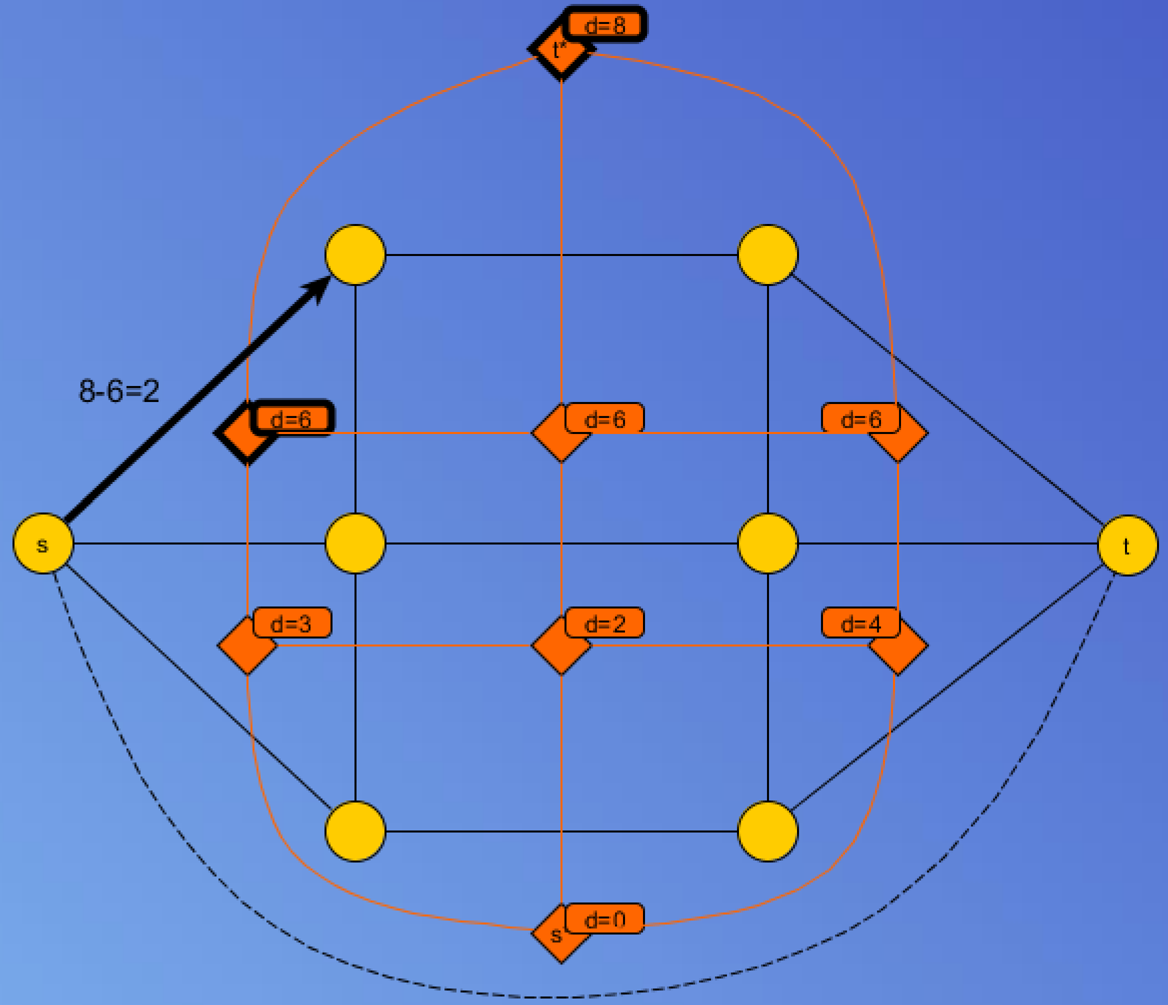
st-planar graphs - Algorithm

- Compute SP Tree rooted at s*



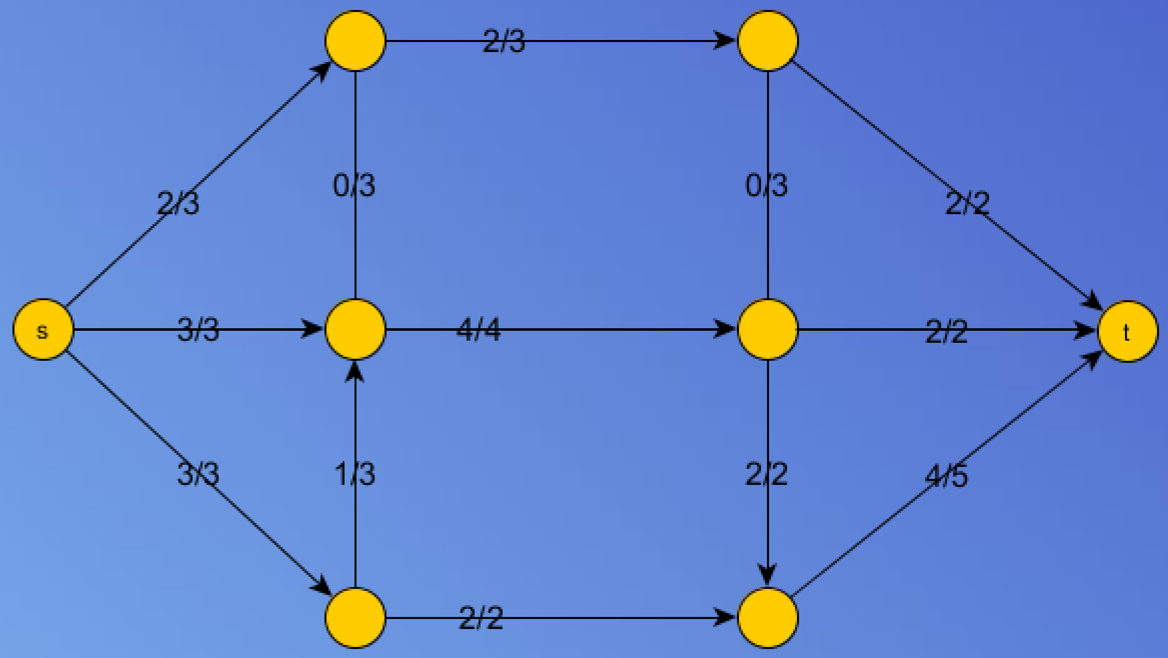
st-planar graphs - Algorithm

- Compute SP Tree rooted at s^*
- Flow f on edge (i, j) is
  $f(i, j) = d(j^*) - d(i^*)$



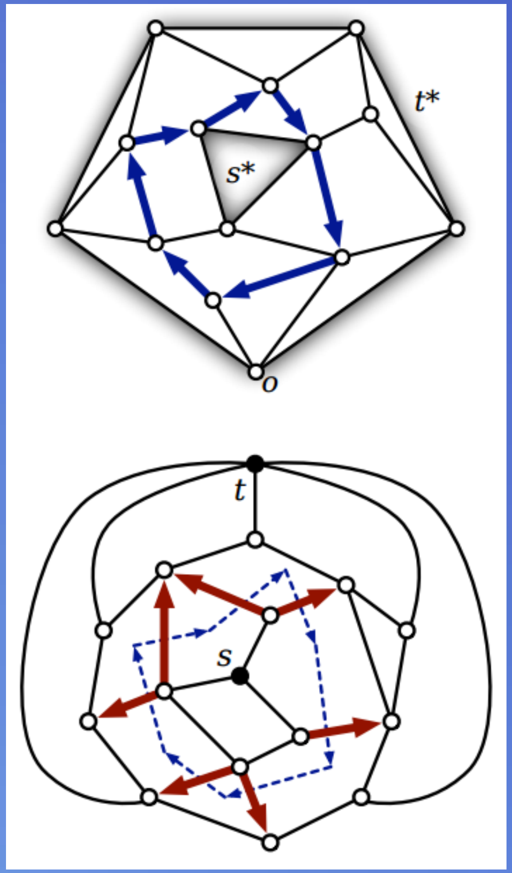
st-planar graphs - Algorithm

- Compute SP Tree rooted at s^*
- Flow f on edge (i, j) is
  $f(i, j) = d(j^*) - d(i^*)$
- SP distances are feasible flow function
  - Satisfy capacity constraints
  - Satisfy flow conservation



Feasible flows

- Cycle in G^* <=> cut in G
- Negative cycle in G^* <=> cut in G with negative residual capacities
- Flow is feasible <=> SP distances in G^* are well defined
  <=> SP distances respect capacities
  <=> No negative reduced lengths
  <=> G^* has no negative cycles



- $\exists$ feasible flow of value λ <=> G_λ^* contains no negative cycles
- Break condition: negative cycle in the SP Tree

Idea for Max flow Algorithm

- Compute feasible st flow with fixed value λ by reduction to a SSSP problem in appropriately weighted dual graph G^*

- Zero flow is always feasible
- Start with $\lambda = 0$ and increase continuously

- Construct SP Tree for each value of λ

Max flow Algorithm

- Search for max λ between 0 and C
  - binary search $O(\log C)$
  - C is bound on the integer capacities
- Construct SP Tree for each value of λ: $O(n * \log n)$
- Check for negative cycle and update λ accordingly
  - Negative cycle => λ too high
  - No negative cycle => λ too low

- Total time: $O(n * \log n * \log C)$

Max flow to parametric SP

Construct parametric SP Tree

- Maintain SP Tree G_{λ}^{*} as λ increases
  - distances induced by the costs
    $c(\lambda, e^*) = c(e) - \lambda * \pi(e^*)$
- In each iteration one edge is replaced: $O(n)$ iterations
  - Choose edge with lowest slack
  - $O(n)$ iterations, each takes $O(\log n)$
- Total time: $O(n * \log n)$

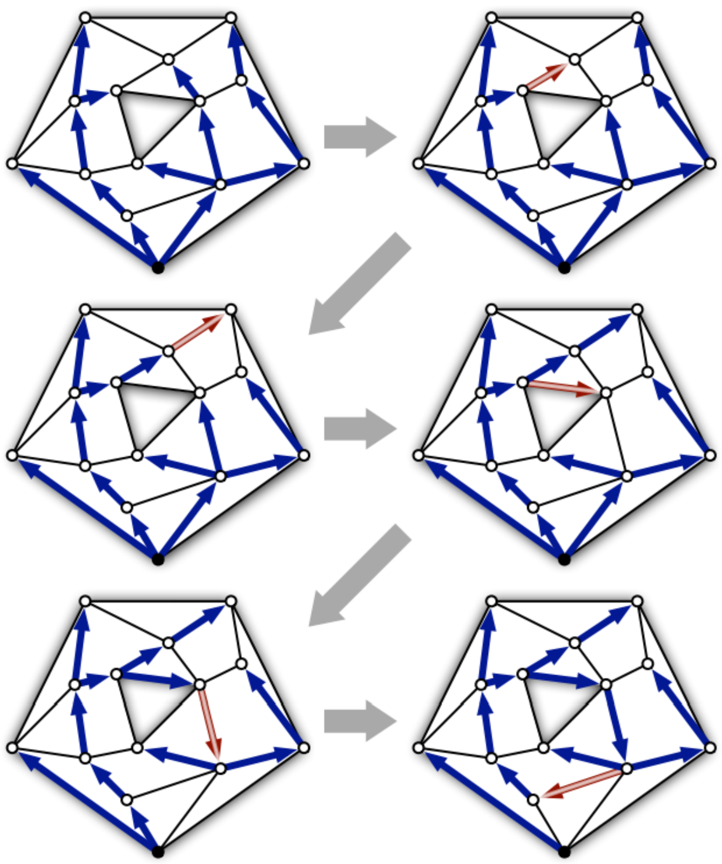


Figure 3. A possible sequence of shortest-path-tree pivots.

Erickson's Algorithm

<u>PlanarMaxFlow(G, c, s, t):</u>

Initialize the spanning tree L, predecessors, and slacks

while s and t are in the same component of L
- $LP \leftarrow$ the path in L from s to t
- $p{\to}q \leftarrow$ the edge in P^* with minimum slack
- $\Delta \leftarrow slack(p{\to}q)$
- for every edge e in LP
  - $slack(e^*) \leftarrow slack(e^*) - \Delta$
  - $slack(rev(e^*)) \leftarrow slack(rev(e^*)) + \Delta$
- delete $(p{\to}q)^*$ from L
- if $q \neq o$ ⟨⟨*that is, if* $pred(q) \neq \varnothing$⟩⟩
  - insert $(pred(q){\to}q)^*$ into L
- $pred(q) \leftarrow p$

for each edge e
- $\phi(e) \leftarrow c(e) - slack(e^*)$

return ϕ

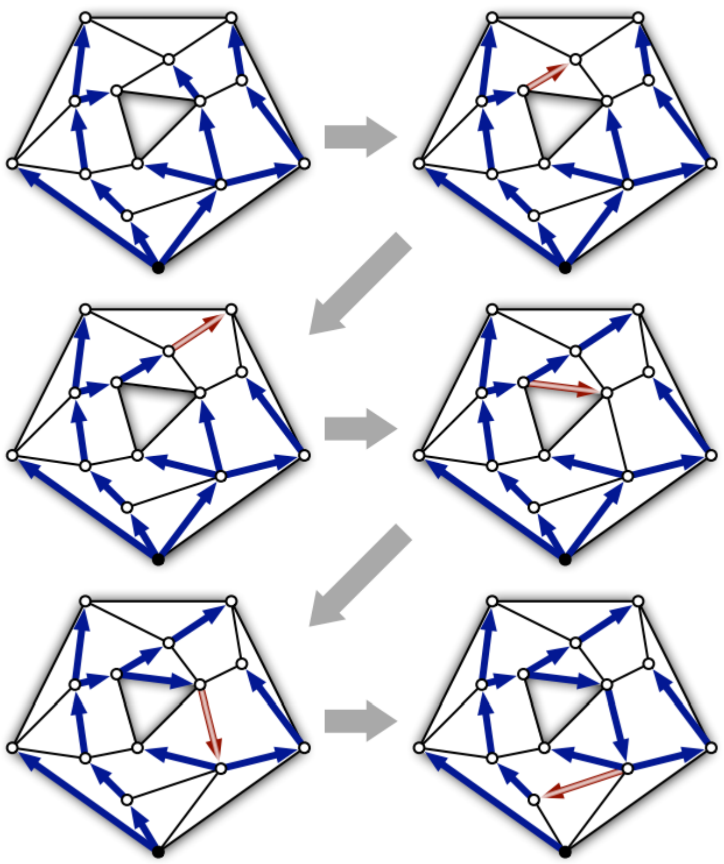
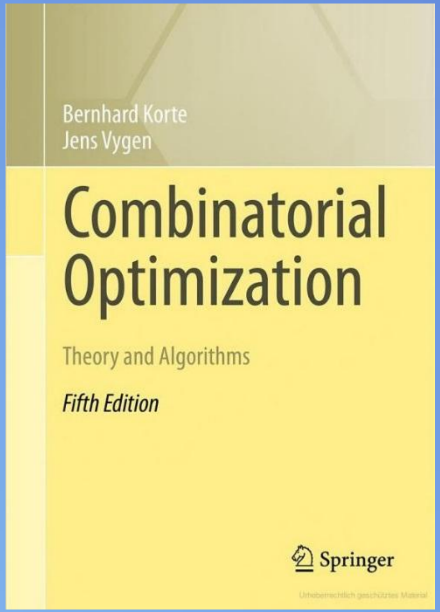


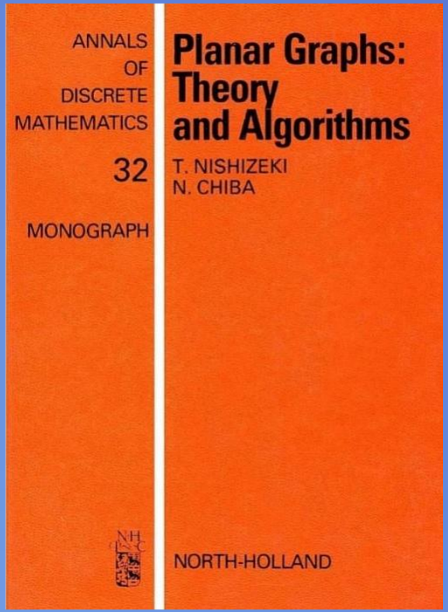
Figure 3. A possible sequence of shortest-path-tree pivots.

- Flows -> Flows in general undirected graphs -> Algorithm -> SP

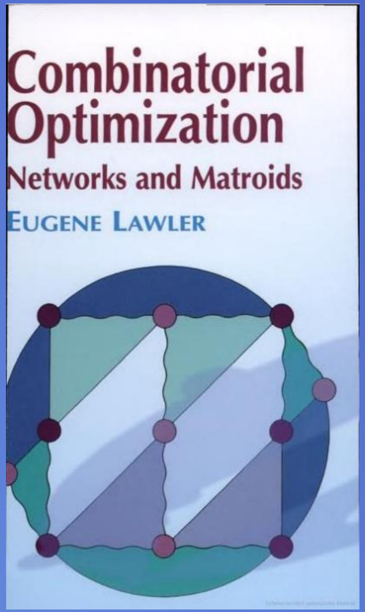
References



Combinatorial Optimization
Theory and Algorithms



Planar Graphs
Theory and Algorithms



Combinatorial Optimization
Networks and Matroids

Thanks for listening!