Data Structures and Algorithms

Lecture 10 – Advanced Applications of Graph

Pengju Ren
Institute of Artificial Intelligence and Robotics
Xi'an Jiaotong University

http://gr.xjtu.edu.cn/web/pengjuren

Problem Solving (6)—— System of difference constraints

Can you find a solutions for the following difference constraints?

$$x_1 - x_2 \leq -2$$

$$x_1 - x_3 \leq -1$$

$$x_2 - x_3 \leq 4$$

$$x_4 - x_2 \leq 5$$

$$x_3 - x_4 \leq 2$$

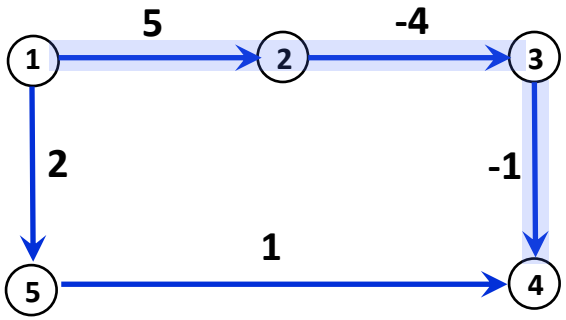
$$x_4 - x_3 \leq -2$$

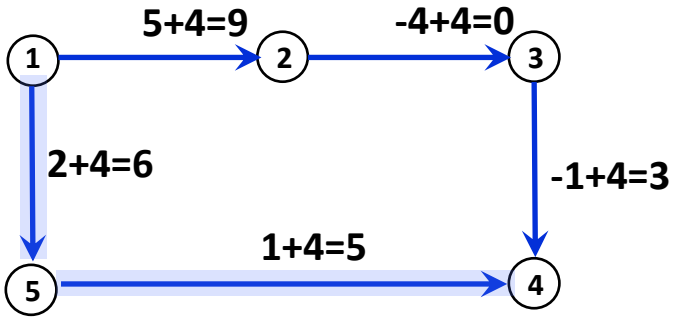
$$x_5 - x_3 \leq -3$$

$$x_5 - x_4 \leq 3$$

Recap: Dijkstra v.s Floyd

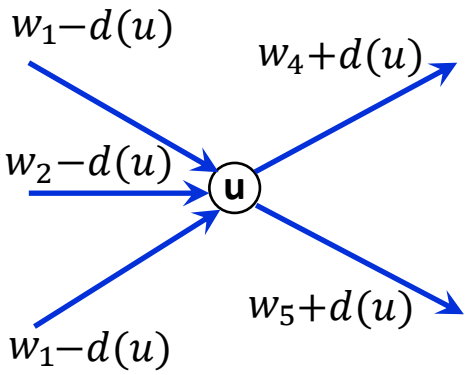
- If run Djikstra algorithm $|V|$ times for each vertex, it can also be used to solve the ASAP problem, the time complexity is $O(V \cdot E \log V)$ is much better than Floyed, which is $O(V^3)$
- Can we eliminate the edges with negative weights?

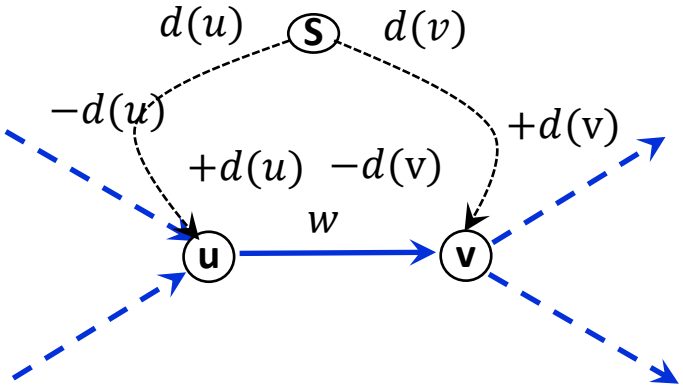




Solution：Johnson

Adjust the weights of the edges, *eliminate negative edges*, while *not change the choice of the shortest path*





$$0 \le w' = w + d(u) - d(v)$$

$$d(v) \le w + d(u)$$

The Johnson algorithm reweights the potential function *d(x)* for every vertex, converting all edge weights to *non-negative*, and then runs *Dijkstra* $|V|$ times.

Properties of reweighting

The weights of the retimed ***path***:

$p = \mathrm{V}_0 \xrightarrow{w_0} \mathrm{V}_1 \xrightarrow{w_1} \ldots \xrightarrow{w_{m-1}} \mathrm{V}_m$ is given by $w'(p) = w(p) + d(\mathrm{V}_m) - d(\mathrm{V}_0)$

Proof: $w'(p) = \sum_{i=0}^{m-1} w'_i$

$$= \sum_{i=0}^{m-1} (w_i + d(\mathrm{V}_{i+1}) - d(\mathrm{V}_i))$$

$$= \sum_{i=0}^{m-1} w_i + (\sum_{i=0}^{m-1} d(\mathrm{V}_{i+1}) - \sum_{i=0}^{m-1} d(\mathrm{V}_i))$$

$$= w(p) + d(\mathrm{V}_m) - d(\mathrm{V}_0)$$

- **DOES NOT** change the total weights in a cycle
- Between any two points, the weight added by any path is equal
- Adding the ***constant value j*** to the weight of each node does not change the results

$$w' = w + (d(V) + j) - d(U) + j)) = w + d(V) - d(U)$$

Solving Systems of Inequalities (1)

- Given a set of M($|\mathbf{E}|$) equalities in N($|\mathbf{V}|$) variables, use ***shortest path algorithm*** to solve the results

Step 1: draw a constraint graph

- Draw the node x_i for each of the N variables x_i , i=1,2,…,N
- Draw the dummy node S
- For each inequality $x_i - x_j \leq k$, draw the edge $x_j \rightarrow x_i$ from the node x_j to node x_i with length k **(note the direction)**
- For each node x_i , i=1,2,…n, draw the edge $S \rightarrow x_i$ from the node S to the node x_i with length 0

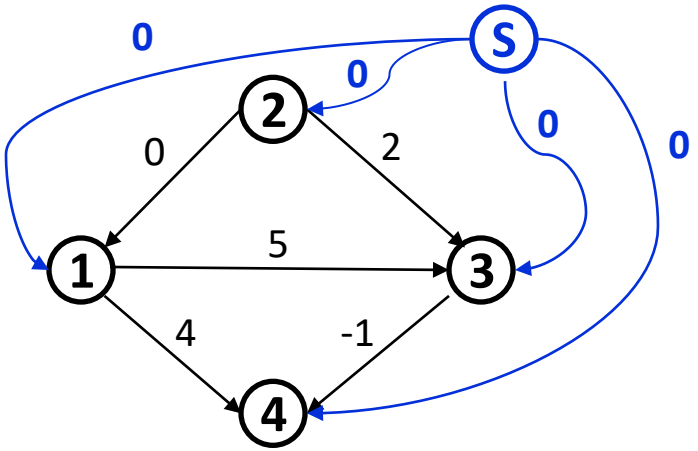
Solving Systems of Inequalities (2)

Step 2: Solve using a shortest path algorithm

- The system of inequalities has a solution **if and only if** the constraint graph contains *no negative cycles*
- If a solution exists, one solution is where x_i is the *minimum-length path* from the node S to the node x_i

Example

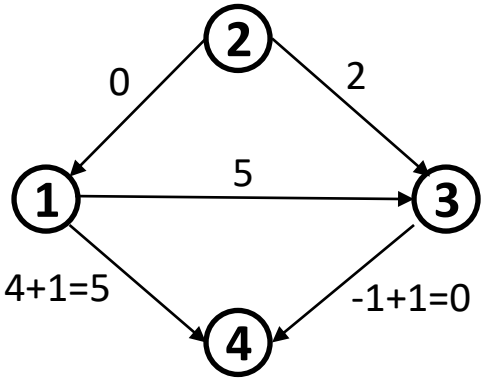
$$x_1 - x_2 \le 0$$
$$x_3 - x_1 \le 5$$
$$x_4 - x_1 \le 4$$
$$x_4 - x_3 \le -1$$
$$x_3 - x_2 \le 2$$



Bellman-Ford shortest path algorithm:

$$R^{(6)} = \begin{bmatrix} \infty & \infty & 5 & 4 & \infty \\ 0 & \infty & 2 & 1 & \infty \\ \infty & \infty & \infty & -1 & \infty \\ \infty & \infty & \infty & \infty & \infty \\ 0 & 0 & 0 & -1 & \infty \end{bmatrix}$$

$$x_1 = 0, x_2 = 0, x_3 = 0, x_4 = -1$$



Johnson Algorithm

- **As long as there is no negative cycle in the graph, Johnson's algorithm will always find a valid potential function, which means could *eliminate negative edges but not change the choice of the shortest path.***
- **Reweighting yields non-negative edges**:

$$\widehat{w}(u, v) = w(u, v) + d(u) - d(v) \geq 0$$

- **Hence all reweighted edges are non-negative, and Dijkstra can be applied correctly.**
- **Time complexity** : $O(VE \log V)$

NOTICE: If the graph contains a negative cycle, Bellman-Ford will detect it, and shortest paths can be arbitrarily negative, so Johnson cannot solve it.

System of difference constraints

$$x_1 - x_2 \leq -2$$

$$x_1 - x_3 \leq -1$$

$$x_2 - x_3 \leq 4$$

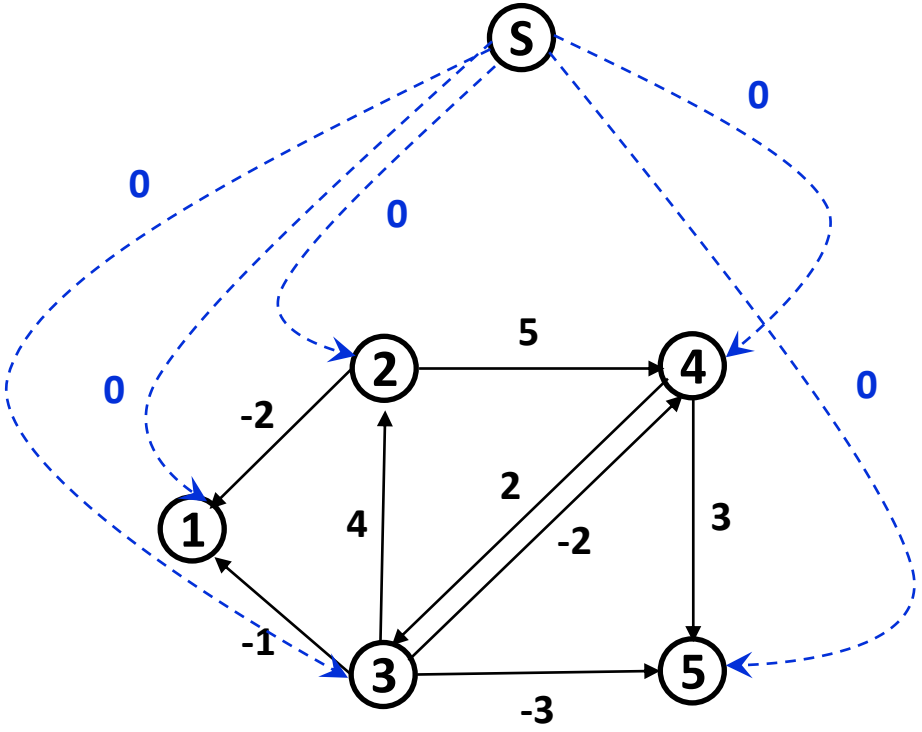
$$x_4 - x_2 \leq 5$$

$$x_3 - x_4 \leq 2$$

$$x_4 - x_3 \leq -2$$

$$x_5 - x_3 \leq -3$$

$$x_5 - x_4 \leq 3$$

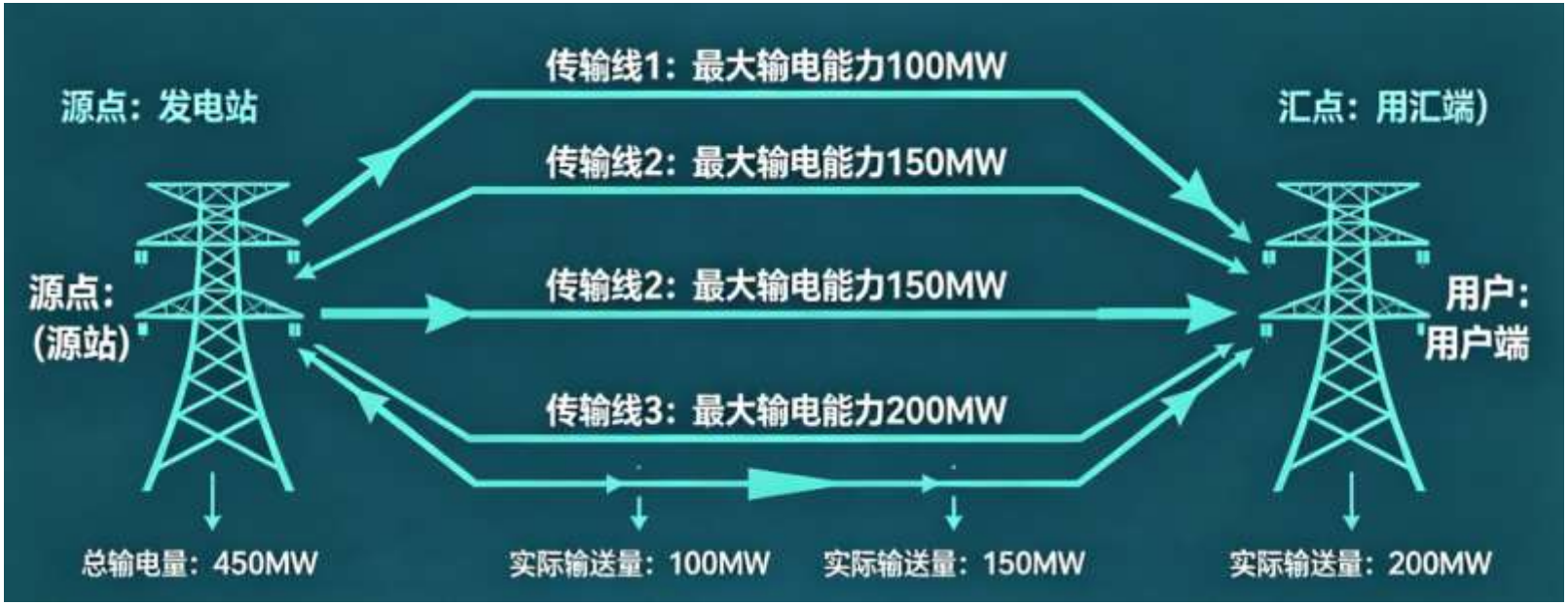


$$(-1, -3, 0, -2, -3)$$

$$(1, -1, 2, 0, -1)$$

Problem Solving (7)—— Maximum Flow problem

In a power system where there are multiple transmission lines of different capacities between power generation stations and the receiving end, how should the power distribution be planned to maximize the current supply between the *source* and the *sink*?



Maximum Flow problem

Given a flow network $G = (V, E)$ with a source vertex s and a sink vertex t, where each edge $(u, v) \in E$ has a non-negative capacity $c(u, v)$,the **maximum flow problem** is to find a flow assignment f that maximizes the total flow from s to t subject to:

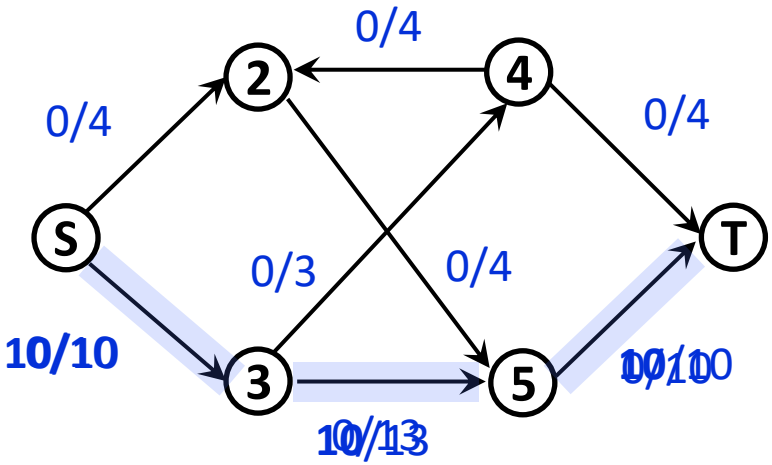
- **Capacity constraint**: $0 \leq f(u, v) \leq c(u, v)$ for all $(u, v) \in E$.
- **Flow conservation**: For every vertex $v \in V \setminus \{s, t\}$,the sum of incoming flows equals the sum of outgoing flows.

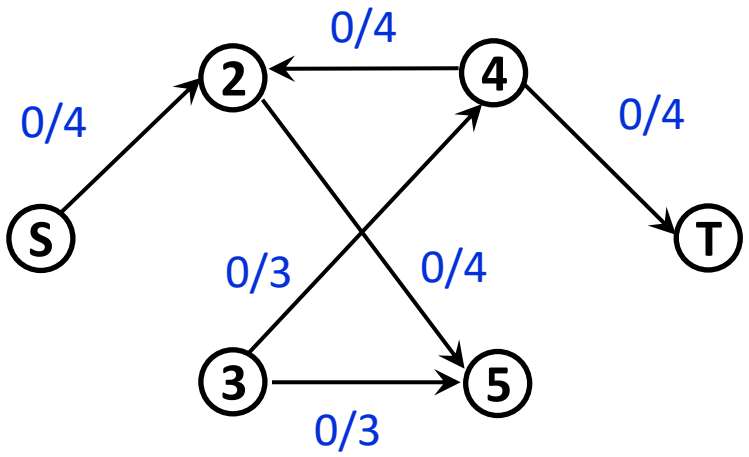
$$\forall v \neq s, t, \qquad \sum_{e:into\ v} f(e) = \sum_{e:out\ of\ v} f(e)$$

1. **Start from *S*, find a feasible path to *T*, and increase the flow by the maximum amount that this path can support;**
2. **Then, in the *residual network*, find another feasible path and increase the corresponding flow;**
3. **Repeat the above steps until the *maximum flow* is reached.**

Maximum Flow problem

1. Start from *S*, find a feasible path to *T*, and increase the flow by the maximum amount that this path can support;
2. Then, in the *residual network*, find another feasible path and increase the corresponding flow;
3. Repeat the above steps until the *maximum flow* is reached.

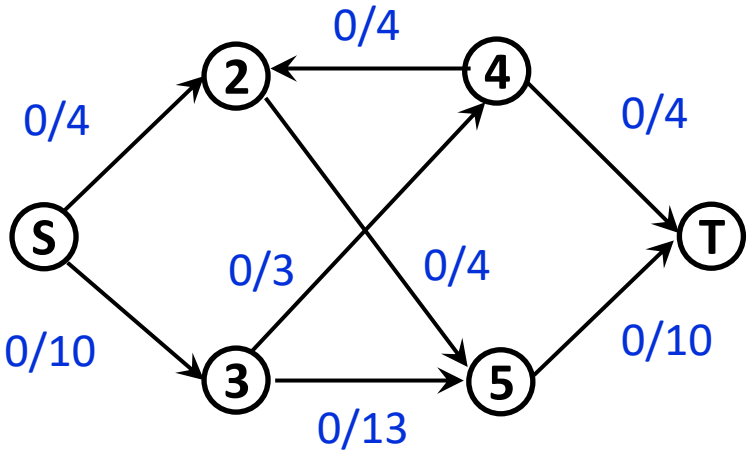


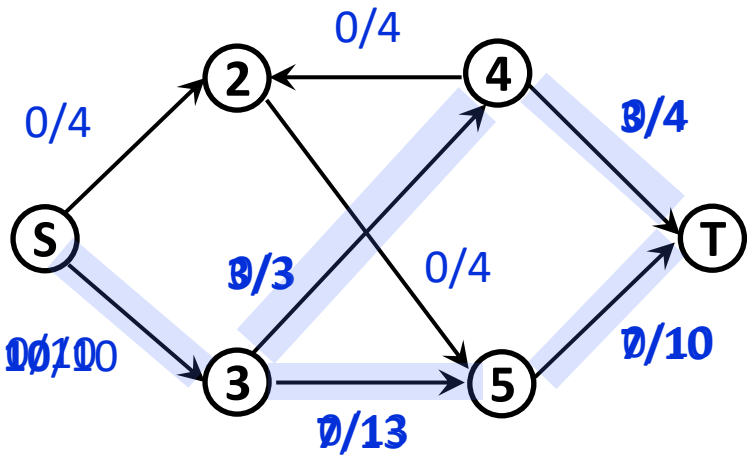


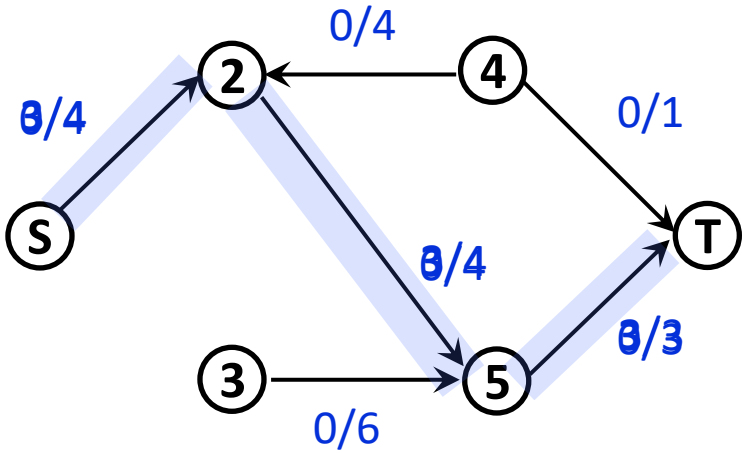
residual network

Maximum flow = 10 ?

Maximum Flow problem







residual network

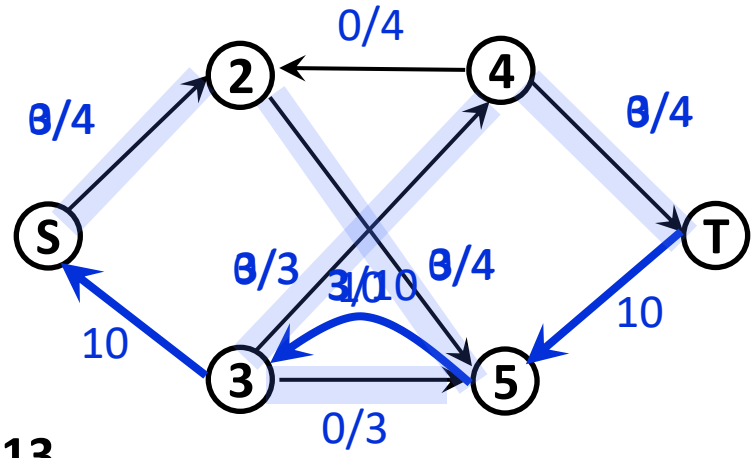
Maximum flow = 10+3=13

Residual Network

Given a flow network $G = (V, E)$ and a flow f on it, the **residual network** G_f is a directed graph consisting of the same vertex set V and an edge set E_f defined as follows:

for each original edge $e = (u, v) \in E$,

- if $f(e) < c(e)$,we add a forward edge (u, v) with residual capacity $c_f(u, v) = c(e) - f(e)$;
- if $f(e) > 0$,we add a **backward** edge (v, u) with residual capacity $c_f(v, u) = f(e)$.

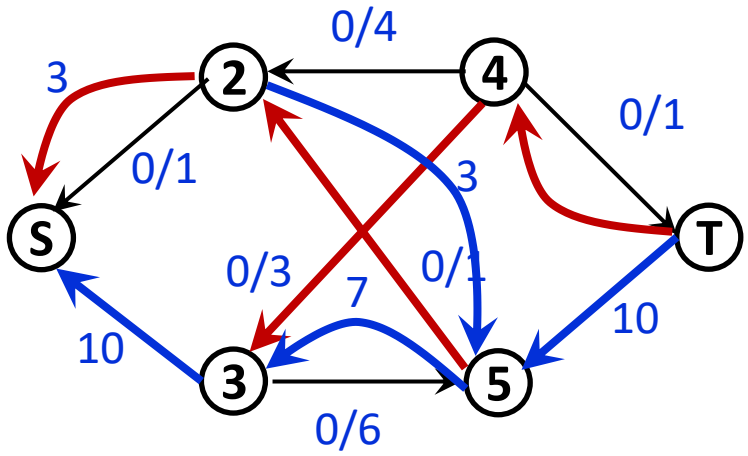


$S \to 2 \to 5 \to 3 \to 4 \to T$

Maximum flow = 10+3=13

residual network

Residual Network



residual network

At this point, there is no path from s to t in the residual network, and the current flow (13) is a *maximum flow*.

Maximum Flow Solution : Ford-Fulkerson

```
FORD-FULKERSON(G, s, t):
    for each edge (u, v) in G.E:
        f[u][v] = 0
        f[v][u] = 0  // reverse flow initially zero
    while true:
        // find any path from s to t in residual network using DFS/BFS
        path = FIND_PATH(s, t, residual capacities)
        if no path: break
        flow = min(residual capacity along path)
        for each edge (u, v) in path:
            f[u][v] += flow
            f[v][u] -= flow
            // update residual capacities: c_f(u,v) = c(u,v) - f[u][v],
c_f(v,u) = f[u][v]
    return total flow from s
```

Time Complexity: $O(VE^2)$

Space Complexity: $O(E + V)$

Definition of Cut

Definition of a Cut (in a directed flow network)
Given a flow network $G = (V, E)$ with source s and sink t,
a **cut** (S, T) is a partition of the vertex set V such that:

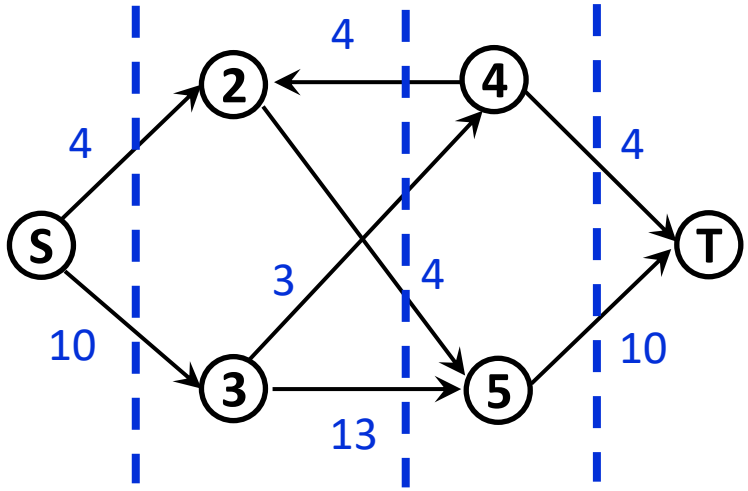
- $S \cup T = V$,
- $S \cap T = \emptyset$,
- $s \in S, t \in T$.

The **capacity** of the cut is defined as the sum of capacities of all edges from S to T:

$$\text{cap}(S, T) = \sum_{u \in S,\, v \in T} c(u, v)$$

Note: edges from T to S (backward edges) are not counted in the cut capacity.

Example of Cut



$C(\{s\}) = \{e_{s\to 2}, e_{s\to 3}\}$

$C(\{s, 2,3\}) = \{e_{2\to 5}, e_{3\to 4}, e_{3\to 5}\}$

$C(\{4,5\}) = \{e_{4\to 2}, e_{4\to T}, e_{5\to T}\}$

Definition of Minimum Cut

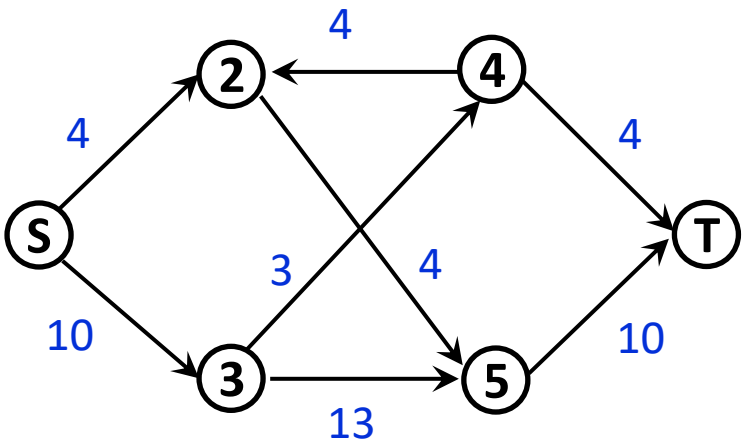
A **minimum cut** is an s-t cut with the smallest possible capacity among all s-t cuts (there may be multiple such cuts). That is:

$$\min_{(S,T)\ is\ an\ s-t\ cut} cap(S, T)$$

In any flow network, **the value of the maximum flow equals the capacity of the minimum cut**.

Therefore, when you find a *maximum flow*, you also obtain a minimum cut (for example, in the residual network, let S be the set of vertices reachable from s, and $T = V \setminus S$; then (S, T) is a minimum cut).

Example of Minimum Cut



$C(\{s\},\{2,3,4,5,t) = C(e_{s\to 2}) + C(e_{s\to 3}) = 14$

$C(\{s,2\},\{3,4,5,t\}) = C(e_{2\to 5}) + C(e_{s\to 3}) = 14$

$C(\{s,3\},\{2,4,5,t\}) = C(e_{s\to 2}) + C(e_{3\to 4}) + C(e_{3\to 5}) = 20$

$C(\{s,4\},\{2,3,5,t\}) = C(e_{s\to 2}) + C(e_{s\to 3}) + C(e_{4\to 2}) + C(e_{4\to T}) = 22$

$C(\{s,2,3\},\{4,5,t\}) = C(e_{2\to 5}) + C(e_{3\to 4}) + C(e_{3\to 5}) = 20$

$C(\{s,2,4\},\{3,5,t\}) = C(e_{s\to 3}) + C(e_{2\to 5}) + C(e_{4\to T}) = 18$

$C(\{s,2,3,4\},\{5,t\}) = C(e_{3\to 4}) + C(e_{3\to 5}) + C(e_{4\to T}) = 20$

$\boldsymbol{C(\{s,2,3,5\},\{4,t\}) = C(e_{3\to 4}) + C(e_{5\to T}) = 13}$

$C(\{s,2,3,4,5\},\{t\}) = C(e_{4\to T}) + C(e_{5\to T}) = 14$

Baseball Elimination (Champion Selection)

w[i] : wins so far
r[i] : total remaining games for team i
g[i][j] : number of games still to be played between i and j

i	team	w[i] wins	l[i] loss	r[i] left	g[i][j] Atl	Phi	NY	Mon
0	Atlanta	83	71	8	-	1	6	1
1	Philadelphia	80	79	3	1	-	0	2
2	New York	78	78	6	6	0	-	0
3	Montreal	77	82	3	1	2	0	-

A team is *mathematically eliminated* if, even if it wins all its remaining games, there is at least one other team that will inevitably finish with more wins.

$$w[3] + r[3] = 77 + 3 = 80 < 83 = w[0]$$

Baseball Elimination (Champion Selection)

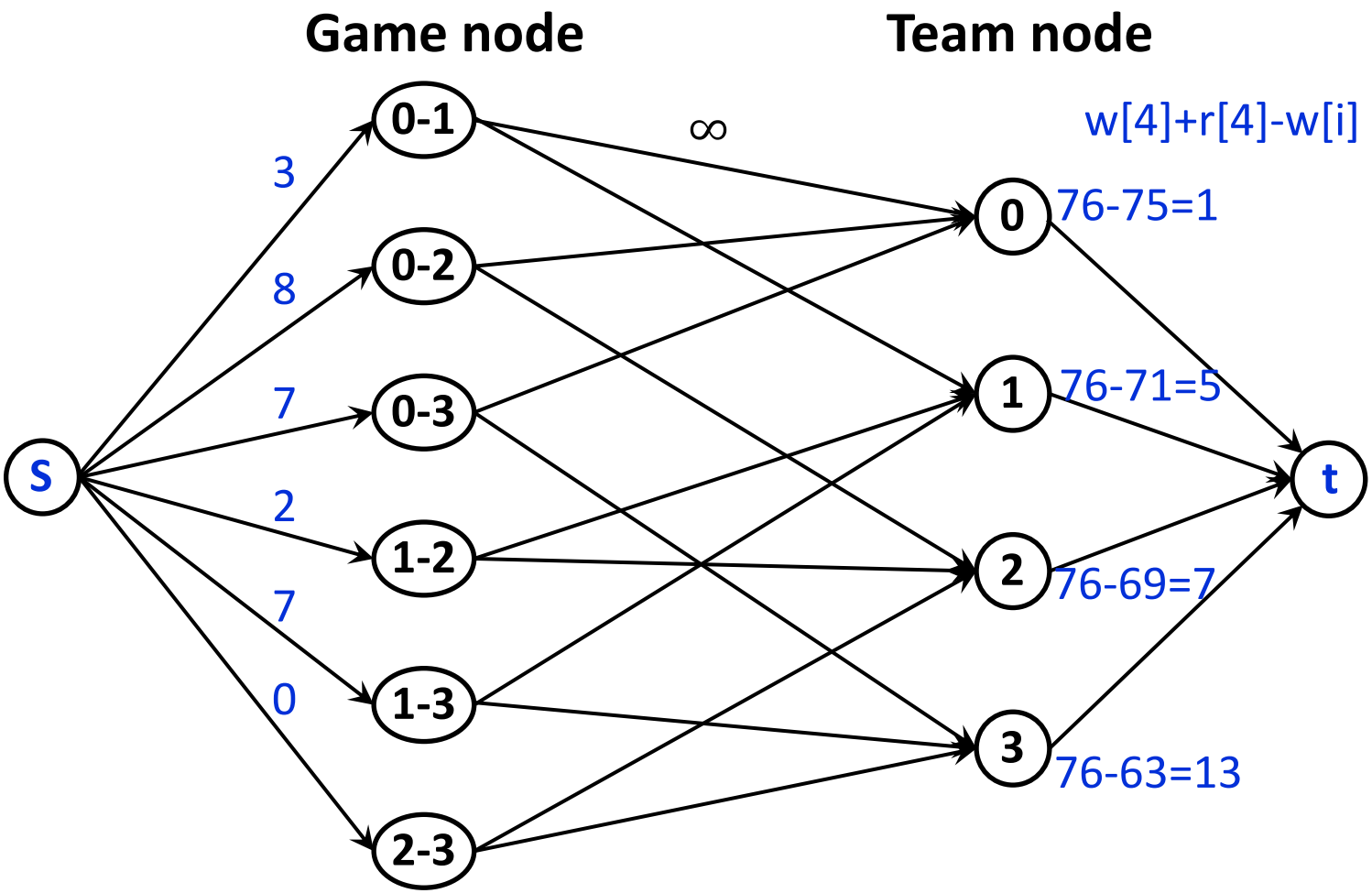
w[i] : wins so far
r[i] : total remaining games for team i
g[i][j] : number of games still to be played between i and j

i	team	w[i] wins	l[i] loss	r[i] left	g[i][j] NY	Bal	Bos	Tor	Det
0	New York	75	59	28	-	3	8	7	3
1	Baltimore	71	63	28	3	-	2	7	7
2	Boston	69	66	27	8	2	-	0	3
3	Toronto	63	72	27	7	7	0	-	3
4	Detroit	49	86	27	3	7	3	3	-

A team is *mathematically eliminated* if, even if it wins all its remaining games, there is at least one other team that will inevitably finish with more wins.

$$w[4] + r[4] = 49 + 27 = 76 > 75 = w[0]$$

Flow Network Construction (Champion Selection)



$\sum 3 + 8 + 7 + 2 + 7 + 0 = 27$

$\sum 1 + 5 + 7 + 13 = 26$

Baseball Elimination (Play-off Selection Top-2)

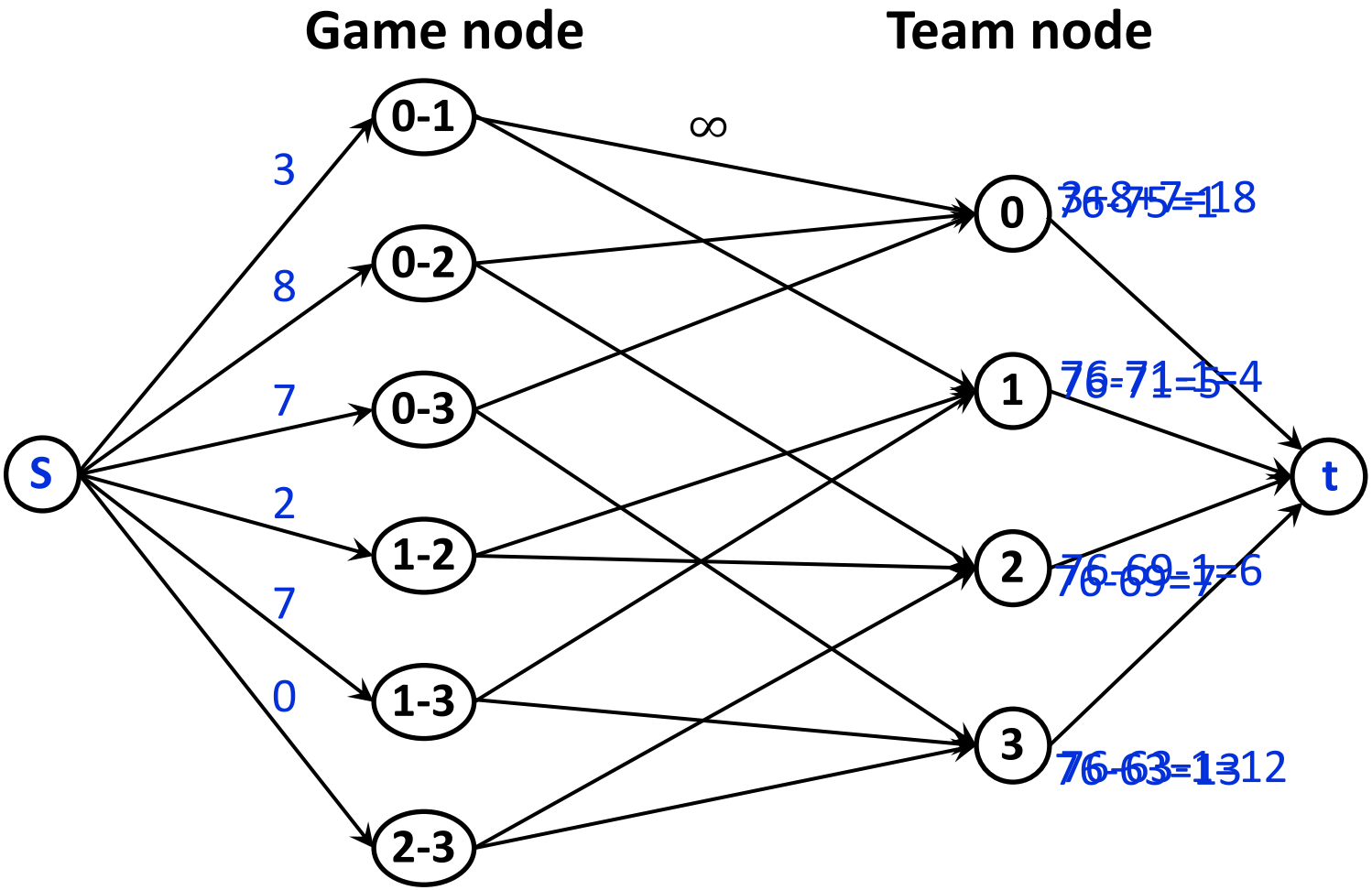
w[i] : wins so far
r[i] : total remaining games for team i
g[i][j] : number of games still to be played between i and j

i	team	w[i] wins	l[i] loss	r[i] left	g[i][j] NY	Bal	Bos	Tor	Det
0	New York	75	59	28	-	3	8	7	3
1	Baltimore	71	63	28	3	-	2	7	7
2	Boston	69	66	27	8	2	-	0	3
3	Toronto	63	72	27	7	7	0	-	3
4	Detroit	49	86	27	3	7	3	3	-

A team is mathematically eliminated if, even if it wins all its remaining games, there is at least one other team (cannot be tied) that will inevitably finish with more wins.

$$w[4] + r[4] = 49 + 27 = 76 > 75 = w[0]$$

Flow Network Construction (Play-off Selection Top-2)



Max-Flow Min-Cut Theorem

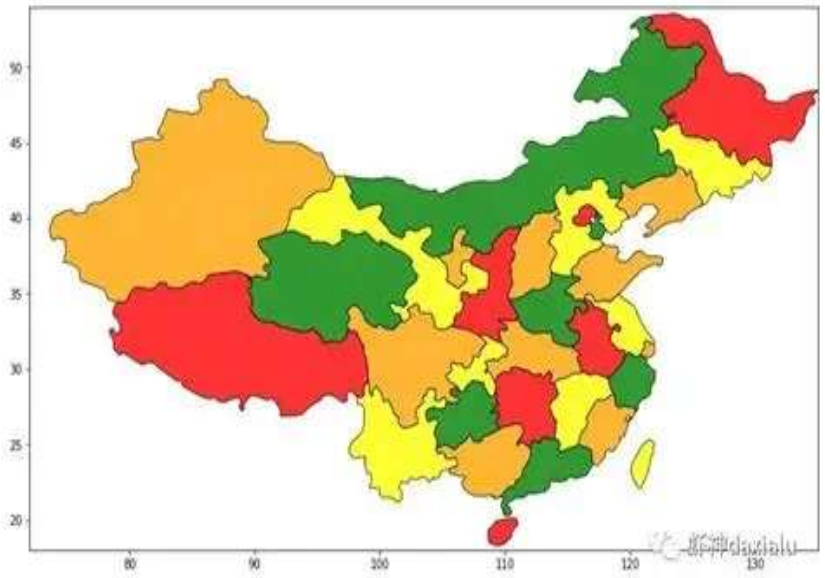
In a flow network (a directed graph with a source s, a sink t, and non-negative edge capacities), the following statements are equivalent:

1) f is the **max-flow** of G
2) Cannot find any path from s to t in the residual network G_f
3) $\text{cap}(S, T)$ is the **min-cut** of G, then $f = \text{cap}(S, T)$

The proof of equivalence of the three conditions is usually done in a cyclic manner: (1) ⇒ (2) ⇒ (3) ⇒ (1)

Problem Solving (8)—— Graph Coloring

How to color each province on a map of China with the fewest colors, with the requirement that adjacent provinces cannot use the same color?



Given an undirected graph $G = (V, E)$,a Graph coloring is an assignment of colors to each vertex such that no two adjacent vertices share the same color.

The **graph coloring problem** is to find the minimum number of colors needed for a proper coloring of G. This minimum number is called the **chromatic number** $\chi(G)$.

Constraint Satisfaction Problem (CSP)

A **Constraint Satisfaction Problem (CSP)** is defined by a triple (X, D, C) ,where:

- $X = \{x_1, x_2, \ldots, x_n\}$ is a set of **variables**,
- $D = \{D_1, D_2, \ldots, D_n\}$ is a set of **domains**, each D_i being the finite set of possible values for variable x_i,
- $C = \{c_1, c_2, \ldots, c_m\}$ is a set of **constraints**, each constraint c_j is a relation defined on a subset of variables (its **scope**), specifying which combinations of values are allowed.

The problem is to find an **assignment of values to all variables** (each $x_i \in D_i$ (such that all constraints in C are satisfied simultaneously.

Real world CSP —— Course timetabling

The *course timetabling problem* can be naturally transformed into a graph vertex coloring problem. How?

triple (X, D, C)

X：**variables ?**

D：**domains ?**

C：**constraints ?**

Real world CSP —— Course timetabling

Organization of the course timetabling problem into the standard CSP form (X, D, C) ,along with its equivalence to graph coloring.

Formal Definition:

- Let **X**= $\{x_1, x_2, \ldots, x_n\}$ be the set of class meetings, where each class meeting x_i has the following attributes:
  - $teacher(x_i)$:the teacher
  - $students(x_i)$:the set of enrolled students
  - $room(x_i)$:the required room
- Let **D**= $\{d_1, d_2, \ldots, d_n\}$ be the **time-slot-room pair**, indicating the specific time and location assigned to the class meeting.
  - $d_i = T \times R$ Let T be the set of time slots (e.g., Monday period 1, Monday period 2, …) and R the set of rooms.

Real world CSP —— Course timetabling

Organization of the course timetabling problem into the standard CSP form (X, D, C) ,along with its equivalence to graph coloring.

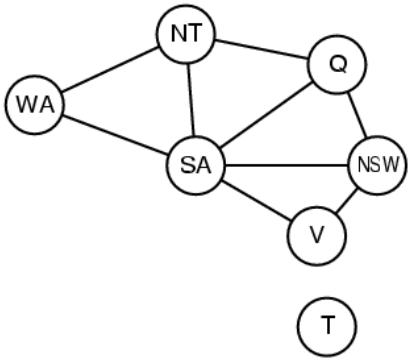
Formal Definition:

- Constraints ***C*** fall into three types:

$$teacher(x_i) = teacher(x_j) \Longrightarrow x_i \neq x_j$$
$$students(x_i) \cap students(x_j) \neq \emptyset \Longrightarrow x_i \neq x_j$$
$$room(x_i) = room(x_j) \Longrightarrow x_i \neq x_j$$

$$(\mathbf{tchr}(x_i) = \mathbf{tchr}(x_j))\Lambda(\mathbf{std}\ (x_i) \cap \mathbf{std}(x_j) \neq \emptyset)\Lambda(\mathbf{room}(x_i) = \mathbf{room}(x_j)) \Longrightarrow x_i \neq x_j$$

Graph Coloring Problem

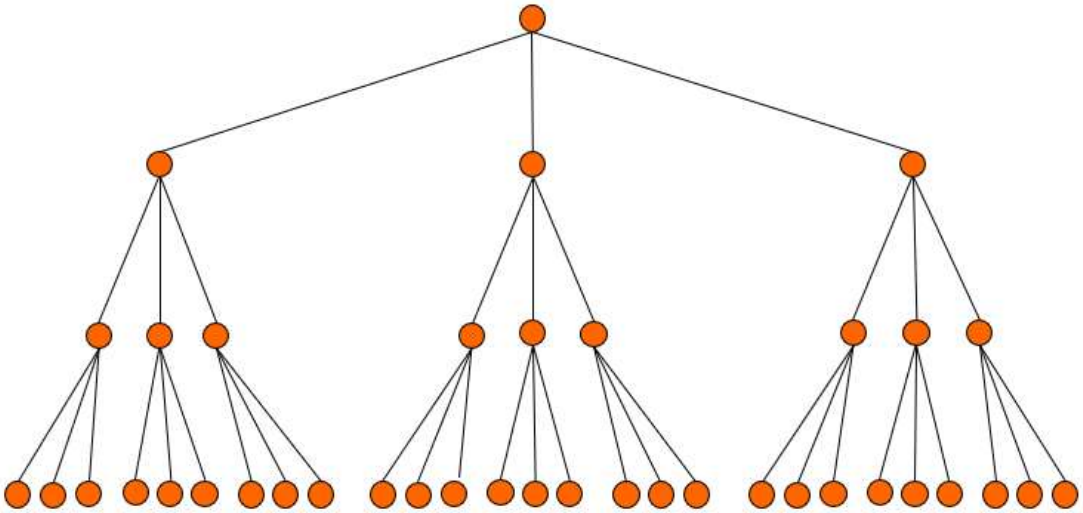


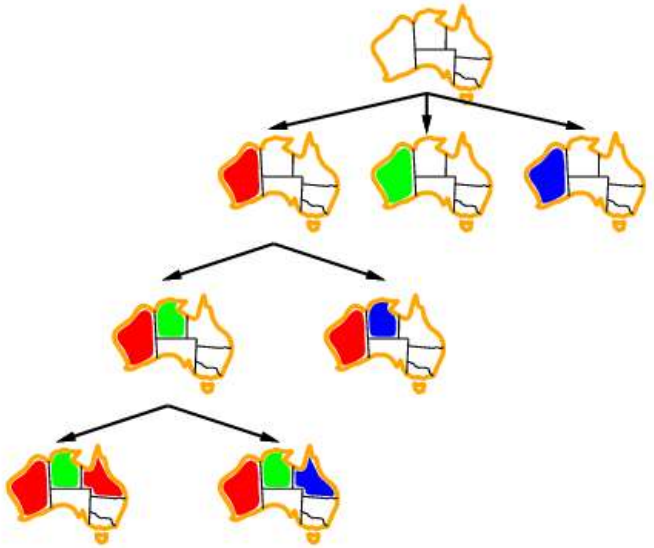
- **Variables:** ***WA, NT, Q, NSW, V, SA, T***
- **Domains:** D_i **= {red, green, blue}**
- **Constraints: adjacent regions must have different colors**
- **Solutions are complete and consistent assignments**

e.g., WA = red, NT = green, Q = red, NSW = green,

V = red, SA = blue, T = green

BFS + Pruning and DFS + Backtracking

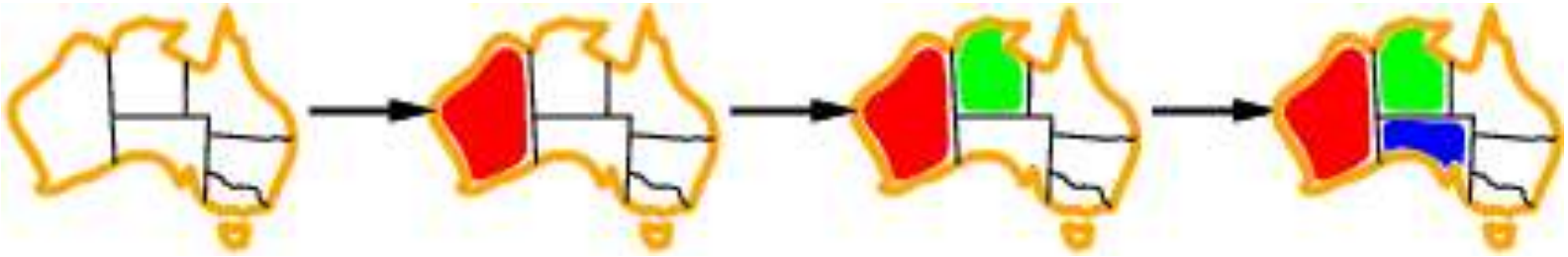




Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
  - Which variable(vertex) should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

- Most constrained variable(vertex):
  - choose the variable(vertex) with the fewest legal values

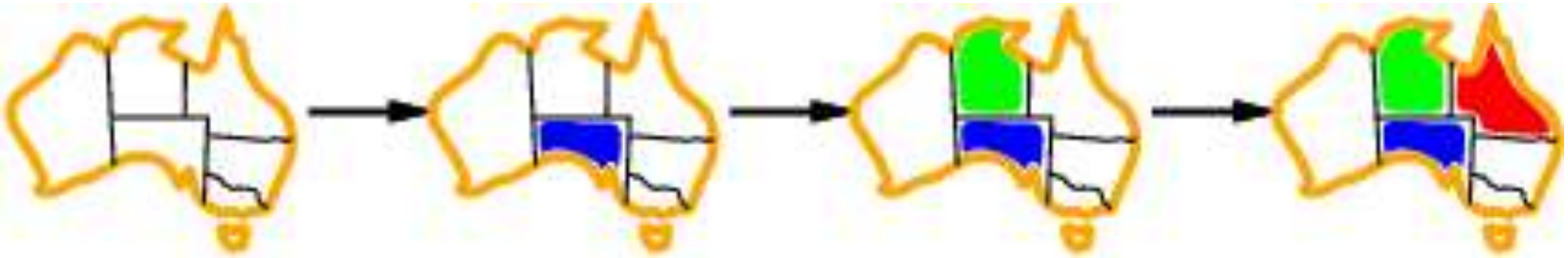


a.k.a. minimum remaining values (MRV) heuristic

Improving backtracking efficiency

- General-purpose methods can give huge gains in speed:
  - Which variable(vertex) should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?

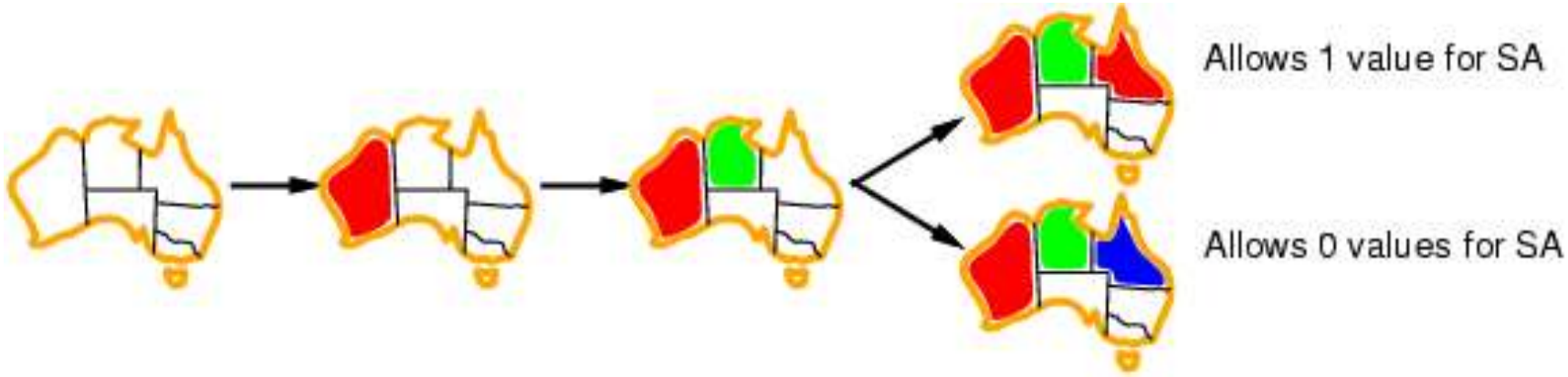
- Most constraining variable (vertex) :
  - choose the variable (vertex) with the most constraints on remaining variables



a.k.a. Degree heuristic

Improving backtracking efficiency

- **General-purpose methods can give huge gains in speed:**
  - **Which variable should be assigned next?**
  - **In what order should its values be tried?**
  - **Can we detect inevitable failure early?**
- **Given a variable, choose the least constraining value:**
  - **the one that rules out the fewest values in the remaining variables**

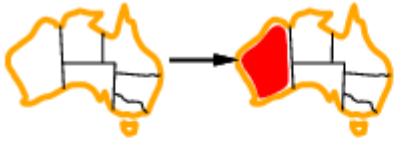


Forward checking

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Filtering： cross off bad options (violate constrains)
  - Terminate search when any variable has no legal values

Forward checking

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Filtering： cross off bad options (violate constrains)
  - Terminate search when any variable has no legal values



Forward checking

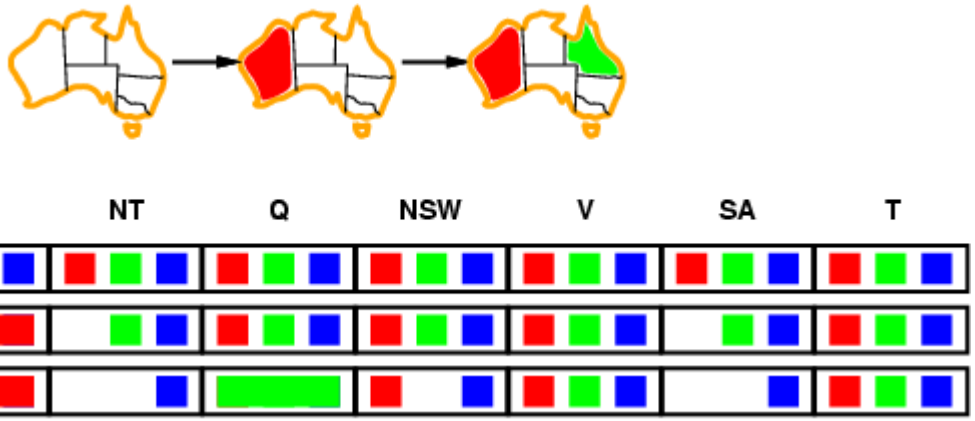
- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Filtering： cross off bad options (violate constrains)
  - Terminate search when any variable has no legal values

Forward checking

- General-purpose methods can give huge gains in speed:
  - Which variable should be assigned next?
  - In what order should its values be tried?
  - Can we detect inevitable failure early?
- Idea:
  - Keep track of remaining legal values for unassigned variables
  - Filtering： cross off bad options (violate constrains)
  - Terminate search when any variable has no legal values

Constraint propagation

- Forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures:

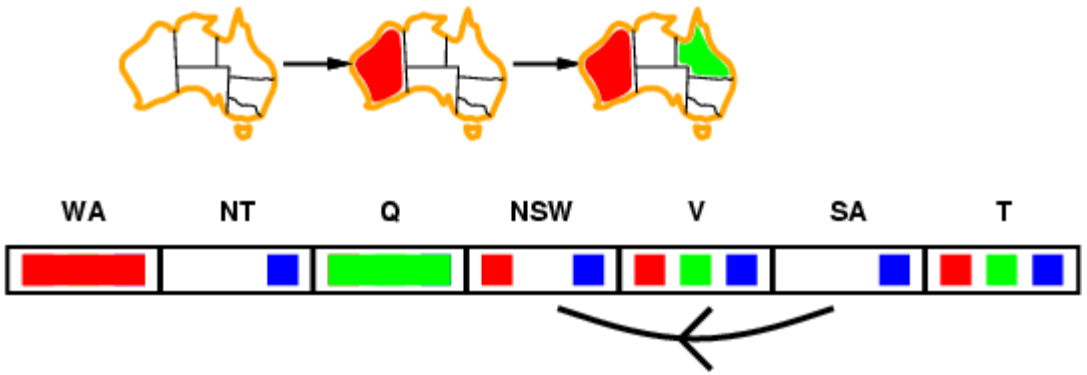


- NT and SA cannot both be blue!
- Constraint propagation repeatedly enforces constraints locally

Arc consistency

- Simplest form of propagation makes each arc consistent
- *X* → *Y* is consistent iff

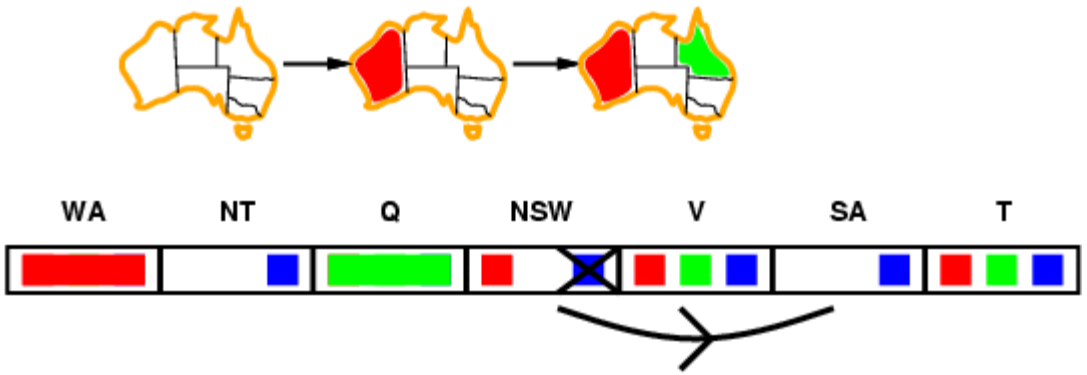
  for every value *x* of *X* there is some allowed *y*



Arc consistency

- Simplest form of propagation makes each arc consistent
- *X* → *Y* is consistent iff

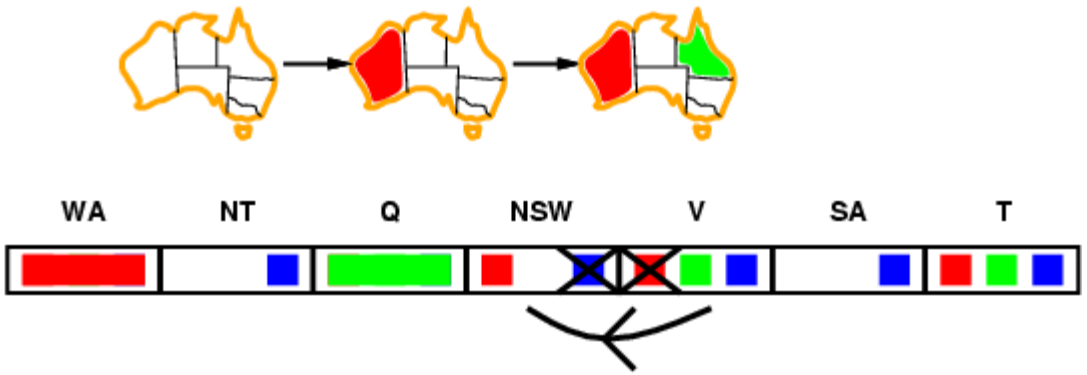
  for every value *x* of *X* there is some allowed *y*



Arc consistency

- Simplest form of propagation makes each arc consistent
- *X* → *Y* is consistent iff

  for every value *x* of *X* there is some allowed *y*

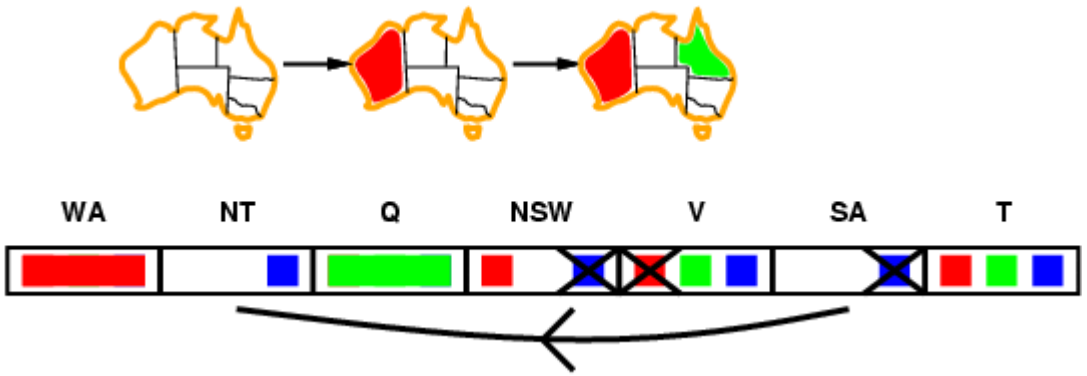


- If *X* loses a value, neighbors of *X* need to be rechecked

Arc consistency

- Simplest form of propagation makes each arc consistent
- *X* → *Y* is consistent iff

  for every value *x* of *X* there is some allowed *y*



- If *X* loses a value, neighbors of *X* need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

Arc consistency algorithm AC-3

function AC-3(*csp*) **returns** false if an inconsistency is found and true otherwise
  inputs: *csp*, a binary CSP with components (X, D, C)
  local variables: *queue*, a queue of arcs, initially all the arcs in *csp*

  while *queue* is not empty **do**
    $(X_i, X_j) \leftarrow$ REMOVE-FIRST(*queue*)
    if REVISE(*csp*, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** *false*
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to *queue*
  return *true*

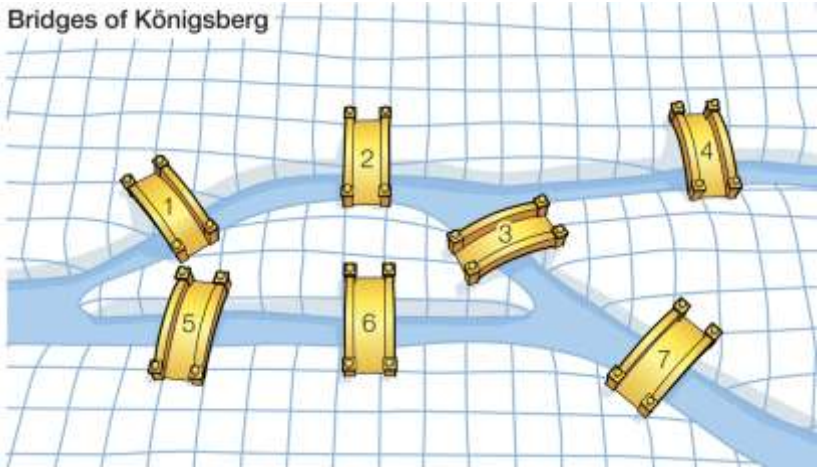
---

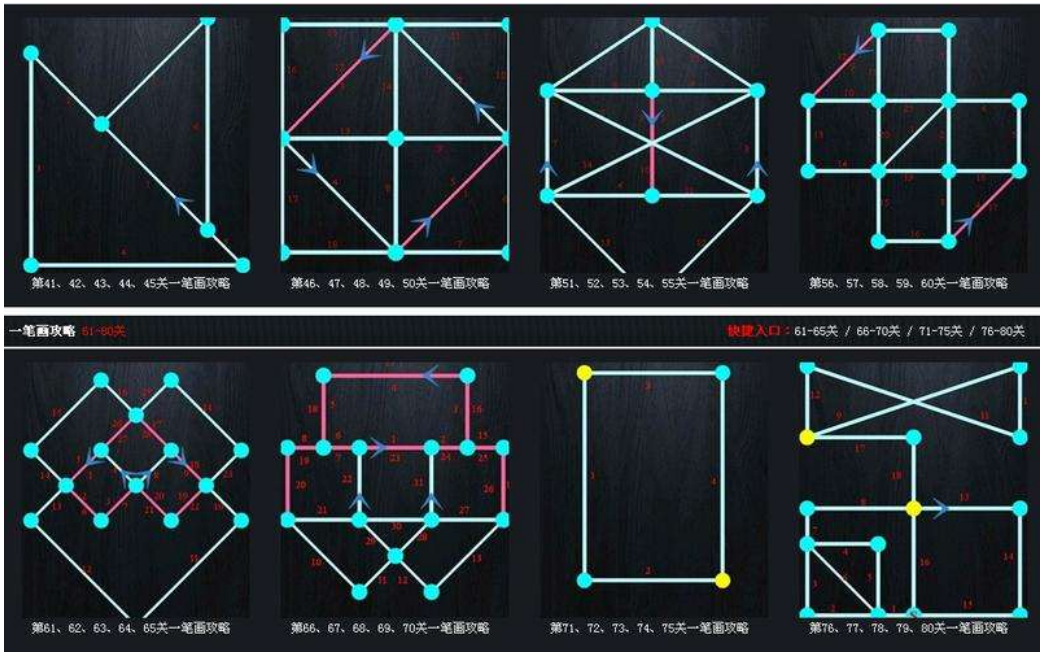
function REVISE(*csp*, X_i, X_j) **returns** true iff we revise the domain of X_i
  revised $\leftarrow$ *false*
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      revised $\leftarrow$ *true*
  return *revised*

- **Time complexity: $O(n^2d^3)$, n is number of variables;**
  d is number of domains;

Problem Solving (9)——Eulerian circuit/trail

Seven Bridges of Königsberg： if the seven bridges of the city of Königsberg over the river Preger, can all be traversed in a single trip without doubling back



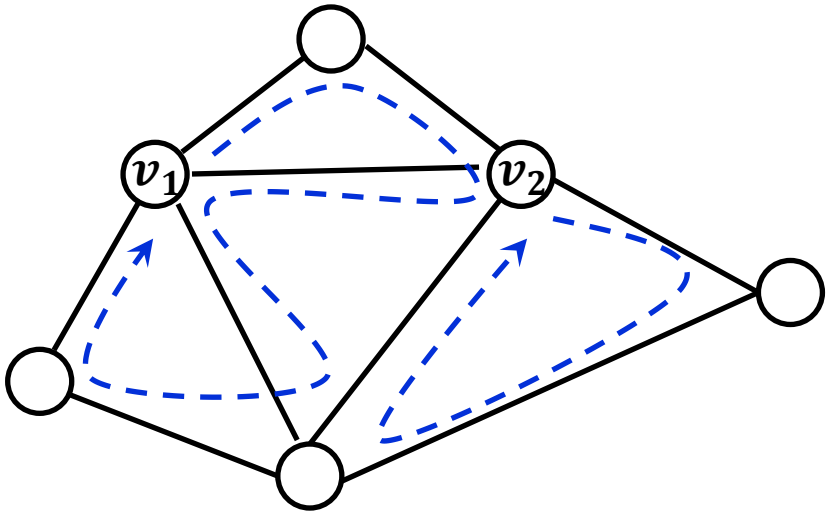


Eulerian circuit (or cycle) : a closed trail in a graph that **visits every edge exactly once** and *returns to its starting vertex*.

Eulerian trail (or path): a trail that **visits every edge exactly once**, but *does not necessarily return to the starting vertex*.

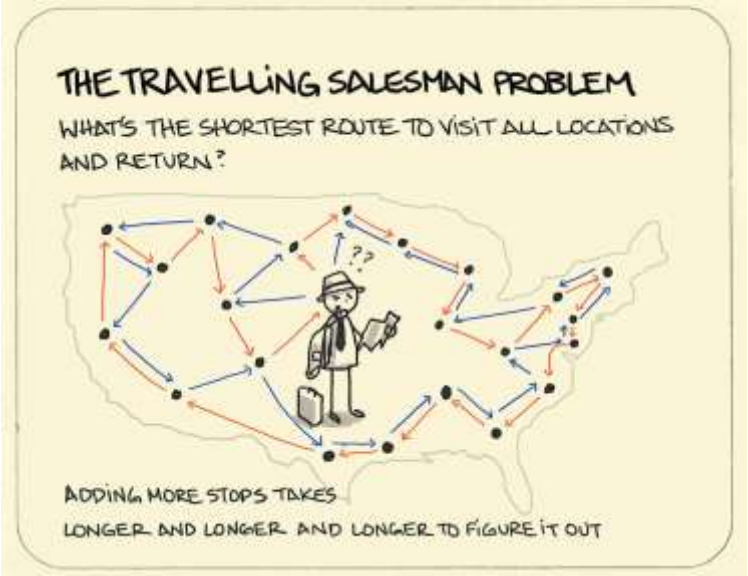
Is there always a solution?

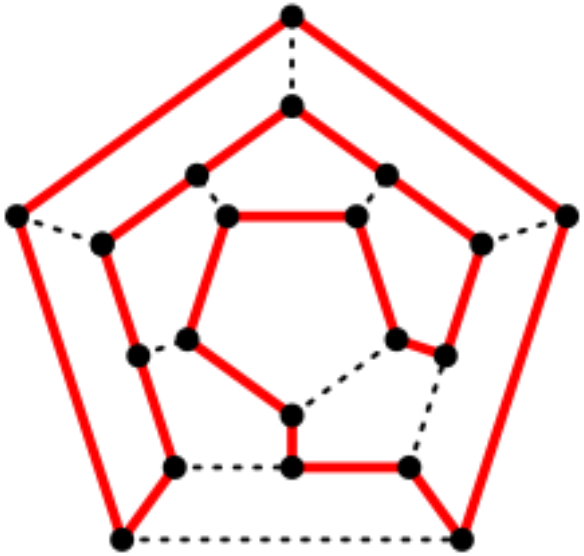
- An Eulerian circuit exists ***iff*** every vertex has even degree.
- An Eulerian path exists ***iff*** exactly (0 or 2) vertices have odd degree (those two as endpoints).
- For directed graphs, an Eulerian circuit exists ***iff*** each vertex has in-degree = out-degree and the graph is strongly connected.



Problem Solving (10)——Hamiltonian circuit/trail

Travelling Salesman Problem (TSP) seeks the shortest possible route that visits each city exactly once and returns to the starting point.

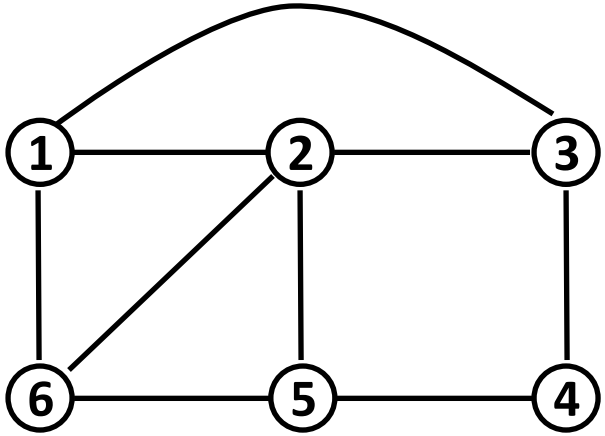




Hamiltonian circuit (or cycle) : a cycle in a graph G, that **visits each vertex of G exactly once** and *returns to the starting vertex*.

Hamiltonian trail (or path): a trail that visits each vertex of $\boldsymbol{G}$ exactly once but *does not necessarily return to the starting vertex*.

Hamiltonian Cycles

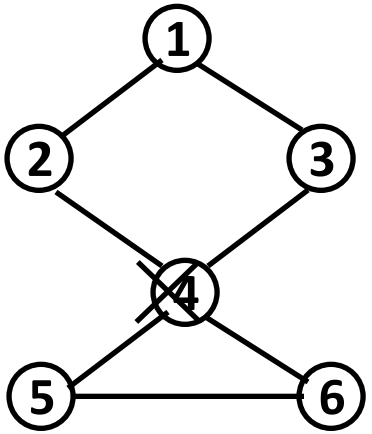


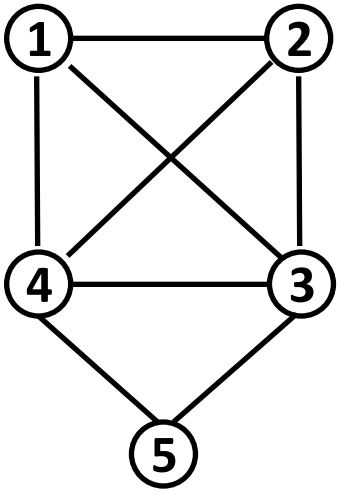
1, 2, 3, 4, 5, 6, 1

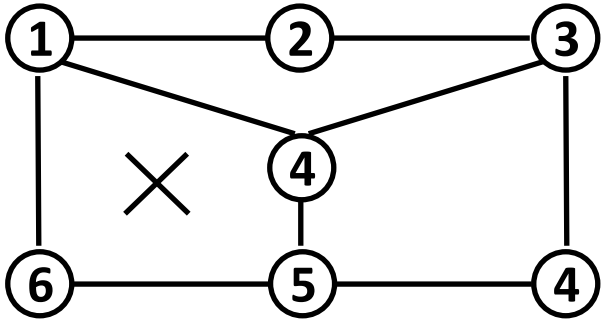
1, 2, 6, 5, 4, 3, 1

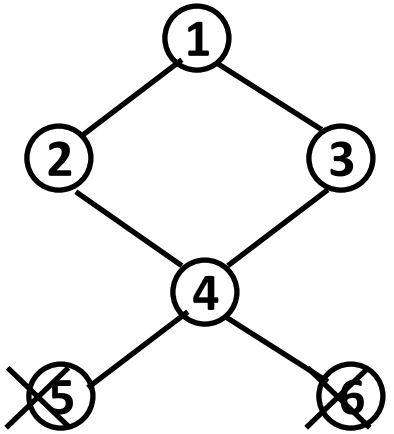
1, 6, 2, 5, 4, 3, 1

~~2, 3, 4, 5, 6, 1, 2~~







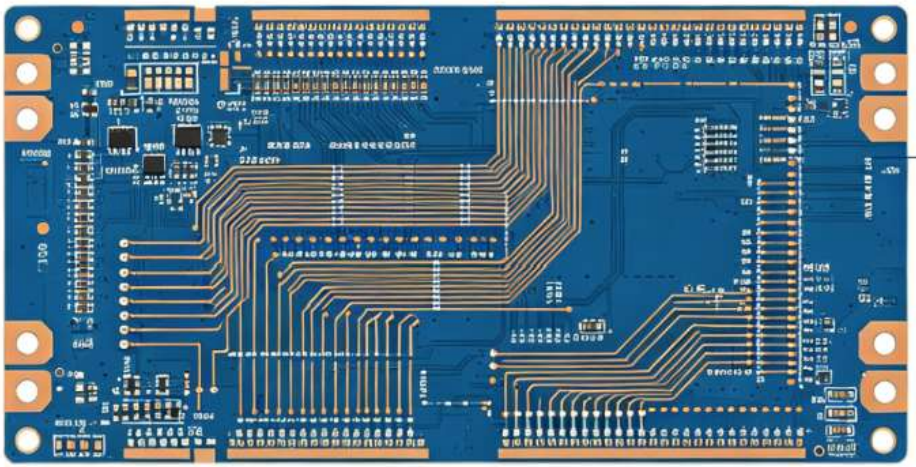


1, 2, 3, 5, 4, 1

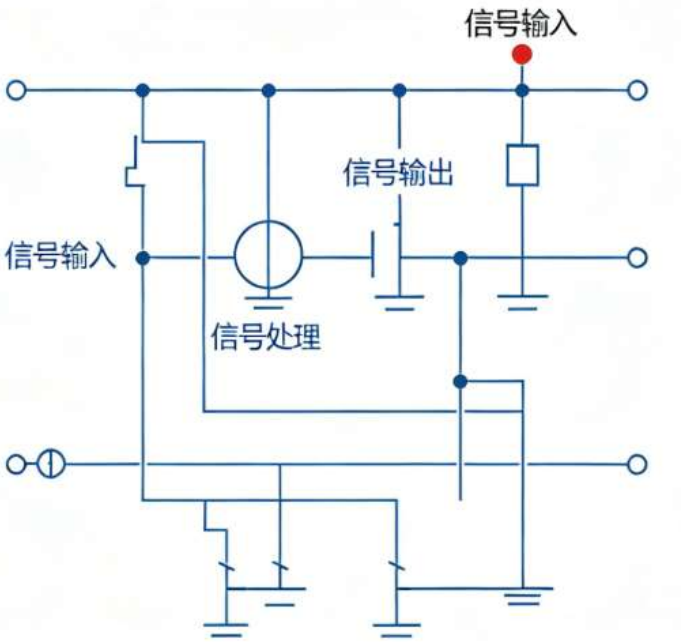
1, 2, 4, 5, 3, 1

Problem Solving (11)——Planar Graph

Can a single-layer circuit board complete the physical wire routing and layout for a given circuit schematic?

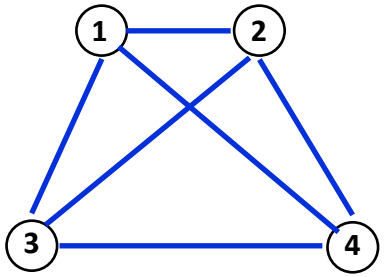


单层板走线层

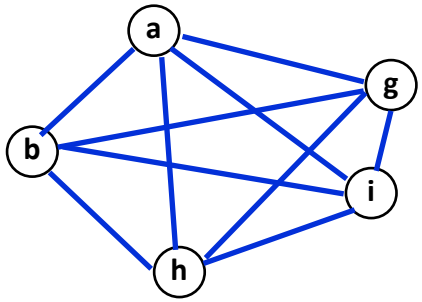


Problem Solving (11)——Planar Graph

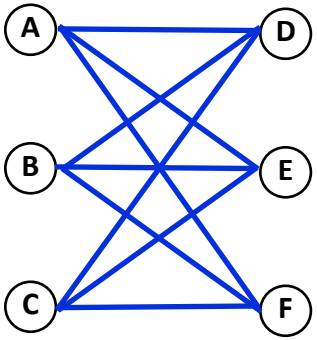
A graph is called planar if it can be drawn on the plane in such a way that its edges intersect only at their endpoints (i.e., *no two edges cross each other at a non-vertex point*).



K_4



K_5



$K_{3,3}$

Which of the above are planar graph?

Planar Graph (Necessary Conditions)

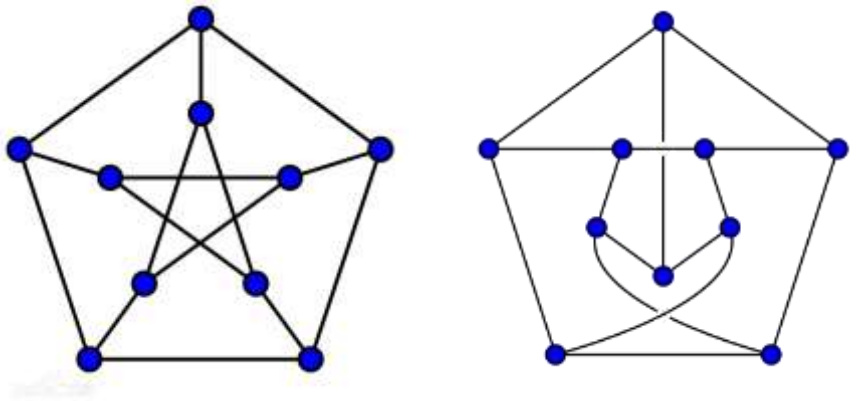
Let G be a connected simple planar graph with $V \geq 3$ vertices, E edges, and F faces. Then:

1.Euler's formula: $V - E + F = 2$.

2.Edge bound: $E \leq 3V - 6$.

If the graph has no triangles (i.e., shortest cycle length ≥ 4), then a stronger bound holds: $E \leq 2V - 4$.

3.Average degree: $\frac{2E}{V} \leq 6$ **(i.e., there exists at least one vertex of degree ≤ 5).**



Petersen graph $(V = 10, E = 15)$

Planar Graph (Necessary and Sufficient Condition)

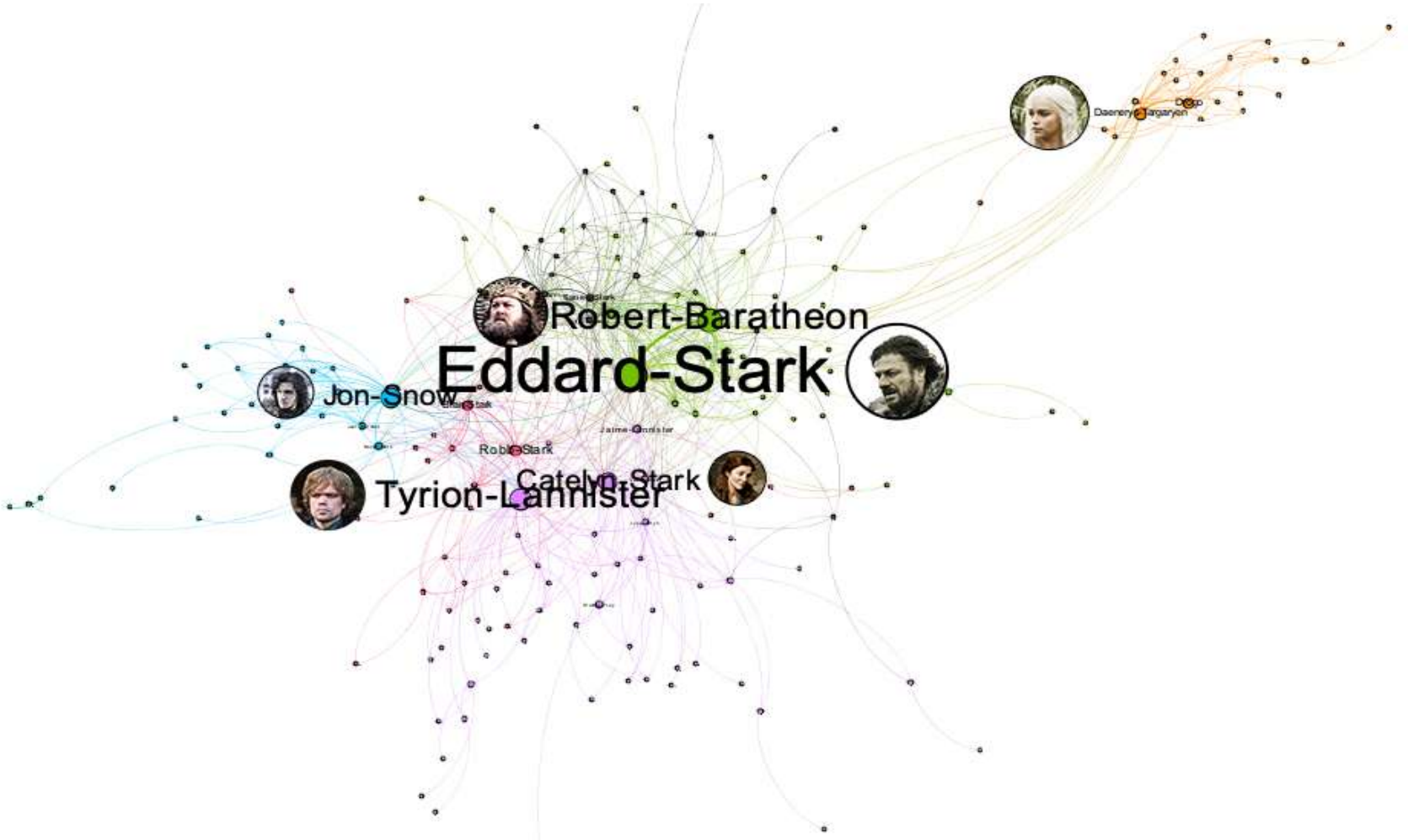
Kuratowski's Theorem :

A graph is planar **iff** it does not contain a ***subdivision*** of K_5 (the complete graph on 5 vertices) or $K_{3,3}$ (the complete bipartite graph with 3 vertices on each side).

Explanation:

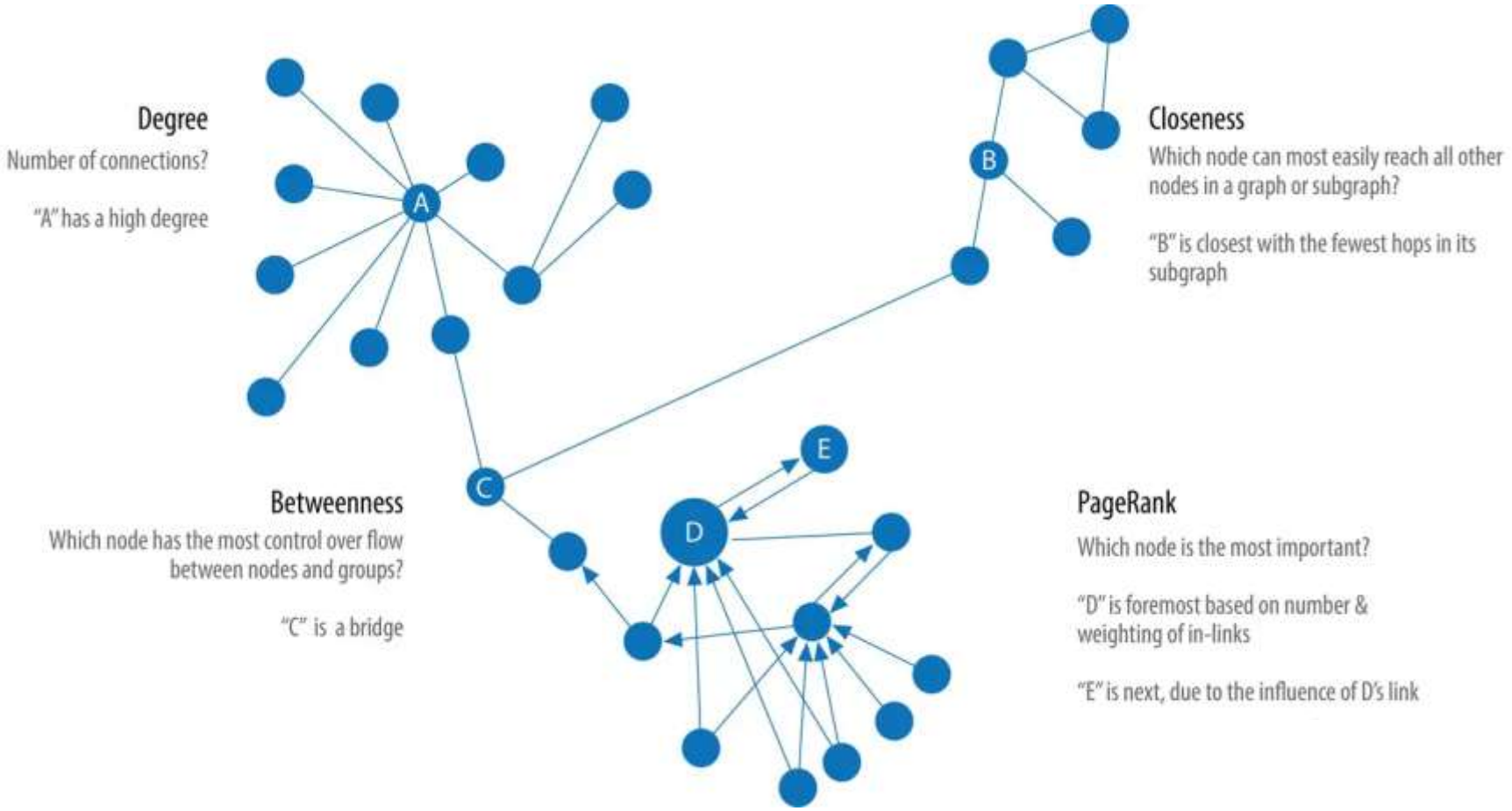
- A ***subdivision*** of a graph is obtained by inserting vertices of degree 2 on its edges (i.e., replacing edges with paths).
- In other words, if you repeatedly remove vertices of degree 2 (smoothing) and the result becomes K_5 or $K_{3,3}$,then the original graph is non-planar; otherwise it is planar.

Problem Solving (12) – Who is the MVP?



Who is the most important person in the “Game of thrones”

Problem Solving (12) – Who is the MVP?



Centrality measures quantify the *"importance" of nodes in a graph*. Different measures define "importance" from different perspectives and suit different scenarios.

Four Centrality Measures

Degree Centrality: A node with many neighbors is more central (*e.g. "social butterfly", super-spreaders*)

$$C_D(v) = \deg(v) \text{ or } \frac{deg(v)}{n-1} \text{ (normalized)}$$

Closeness Centrality: A node has short average distance to all others is more central (*e.g. locating hospitals or fire stations*)

$$C_C(v) = \frac{1}{\sum_{u \neq v} d(v,u)}, \text{ or } \sum_{u \neq v} \frac{1}{d(v,u)} \qquad d(v,u) \text{ is the shortest path}$$

Betweenness Centrality: A node that lies on many shortest paths between others acts as a broker or bottleneck (*e.g. "gatekeepers"*)

$$C_B(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}}, \quad \sigma_{st} \text{ is \#shorest path}, \sigma_{st}(v) \text{ is \# shorest path throught } v$$

Eigenvector Centrality: A node's importance depends not only on the number of its neighbors but also on the importance of those neighbors. (*e.g. highly influential paper, influential users, PageRank*)

$$X_{k+1} = AX_k, \quad A \text{ is the adjacency matrix}$$

Eigenvector Centrality

Initialize vector $\mathbf{x}^{(0)} = (1, 1, \ldots, 1)^T$ or a random positive vector).

Repeat until convergence:

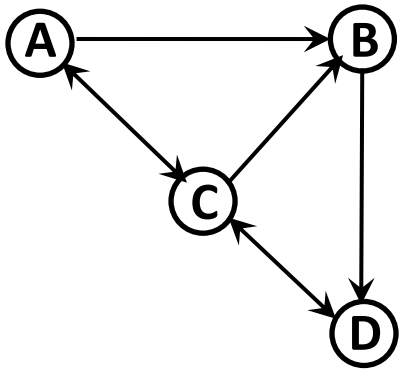
$$\boldsymbol{x} = \frac{\boldsymbol{Ax}}{\|\boldsymbol{Ax}\|} \Rightarrow \boldsymbol{Ax} = \lambda_{max}\boldsymbol{x}$$

λ_{max} is the largest eigenvalue of $\mathbf{A}$.

The components $\mathbf{x_v}$ are the centrality values (usually normalized, e.g., $\sum x_v^2 = 1$ or $\sum x_v = 1$

PageRank Algorithm

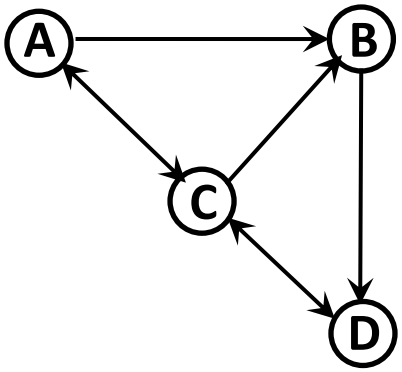
Define "rank" r_j for node j: $r_j = \sum_{i \to j} \frac{r_i}{d_i}$, d_i *is out − degree of node* i



	Init	Iteration 1	Iteration 2
A	¼	1/12	3/24
B	¼	5/24	4/24
C	¼	9/24	9/24
D	¼	1/3	8/24

PageRank Algorithm

Define "rank" r_j for node j: $r_j = \sum_{i \to j} \frac{r_i}{d_i}$, d_i *is out − degree of node i*



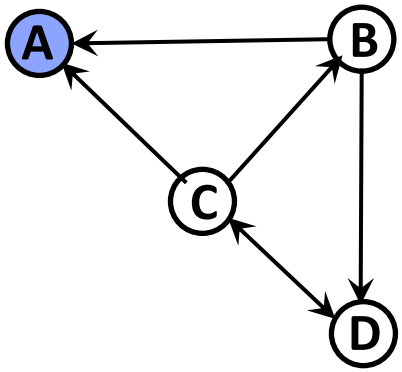
$$\begin{bmatrix} 0 & 0 & 1/3 & 0 \\ 1/2 & 0 & 1/3 & 0 \\ 1/2 & 0 & 0 & 1 \\ 0 & 1 & 1/3 & 0 \end{bmatrix}^{K} \begin{bmatrix} 1/4 \\ 1/4 \\ 1/4 \\ 1/4 \end{bmatrix}$$

$$\boldsymbol{r^K = Mr^{K-1} = M\left(M\left(\ldots(Mr)\right)\right) = M^K r}$$

Repeat until convergence ($\sum_i \left|r_i^{K+1} - r_i^K\right| < \epsilon$)

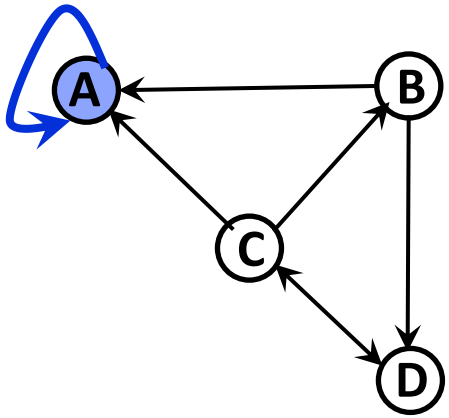
Two Special Cases

Dangling nodes



$out - degree\ of\ node\ A\ is\ 0$

Cyclic absorption



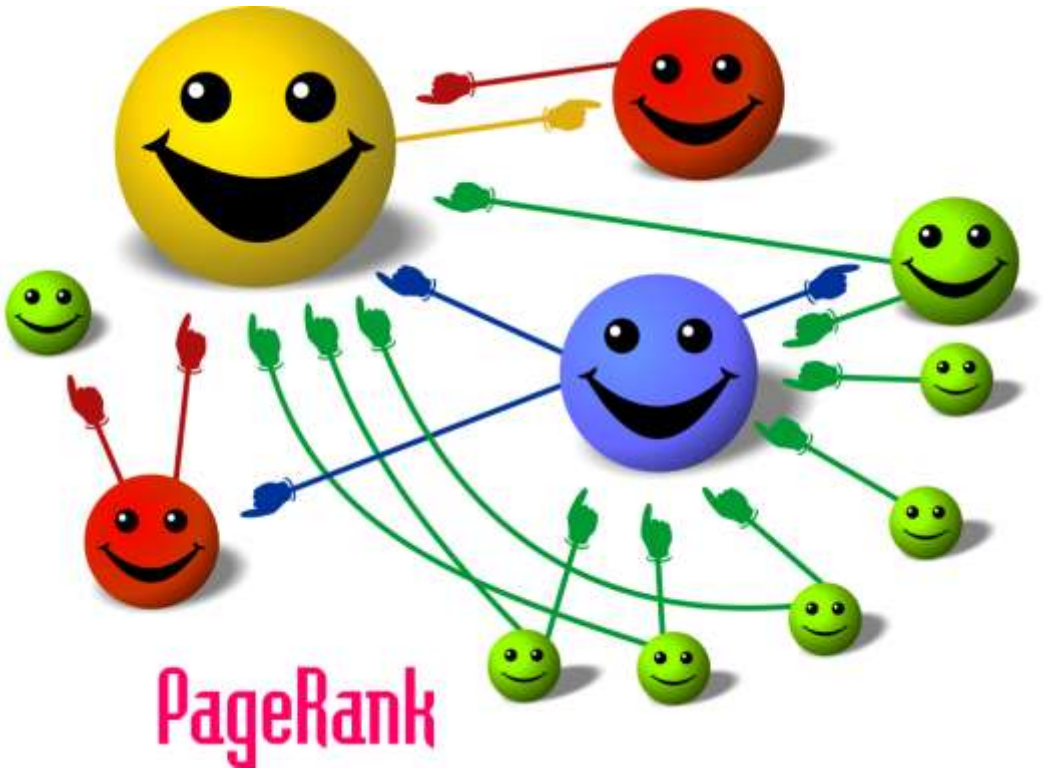
$node\ A\ has\ a\ selfloop$

$$r_c = \frac{1}{n} r_A + \frac{1}{d_D} r_D, \quad d_D\ is\ out - degree\ of\ node\ D$$

$$r_c = \alpha \frac{1}{d_D} r_D + \frac{1-\alpha}{V}, \quad \alpha\ is\ damping\ factor\ (usually\ 0.85), V\ is\ \#\ vertices$$

$$r_j = \alpha \sum_{i \to j} \frac{r_i}{d_i} + \frac{1-\alpha}{V}$$

Page Rank is the modified Eigenvector Centrality



It is the golden key for Google to get rich in the Internet era

Summary

- **Maximum Flow / Minimum Cut:** the fundamental theorem of network flow, widely used in transportation, matching, and network connectivity problems.
- **Graph Coloring:** Color vertices (or edges) with the fewest colors so that adjacent elements have different colors; a classic NP-complete problem.
- **Eulerian Circuit:** traverses every edge exactly once and returns to the start; it exists iff the graph is connected and every vertex has even degree.
- **Hamiltonian Cycle:** visits every vertex exactly once; deciding its existence is NP-complete, and the Traveling Salesman Problem (TSP) is its weighted version.
- **Planar Graph:** it can be drawn on a plane without edge crossings; Kuratowski's theorem states it must not contain a subdivision of K_5 or $K_{3,3}$.
- **Graph Centrality:** quantify node importance via degree, closeness, betweenness, or eigenvector measures – used to identify key nodes in social networks, transportation, epidemiology, etc.