Graph theory and graph algorithms basics

Václav Hlaváč

Czech Technical University in Prague

Czech Institute of Informatics, Robotics and Cybernetics

160 00 Prague 6, Jugoslávských partyzánů 1580/3, Czech Republic

http://people.ciirc.cvut.cz/hlavac, vaclav.hlavac@cvut.cz

Courtesy: Paul Van Dooren, Jeff Erickson, Jaehyun Park, Jianbo Shi

Outline of the talk:

- Graph-based image segmentation, ideas
- Graphs, concepts
- Graph traversal
- Graph coloring
- Special graphs
- Computer representation of graphs
- Flow network, graph cut
- Minimum spanning tree

Graphs, concepts

- It is assumed that a student has studied related graph theory elsewhere. Main concepts are reminded here only.
- http://web.stanford.edu/class/cs97si/, lecture 6, 7
- https://www.cs.indiana.edu/~achauhan/Teaching/B403/LectureNotes/10-graphalgo.html
- https://en.wikipedia.org/wiki/Glossary_of_graph_theory_terms

Concepts

- Graphs: directed, undirected
- Adjacency, similarity matrices
- Path in a graph
- Special graphs

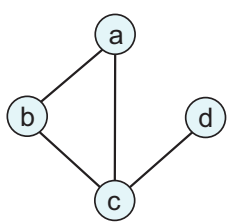
Related graph algorithms

- Shortest path
- Max graph flow = min graph cut
- Minimum spanning tree
-

Undirected/directed graph

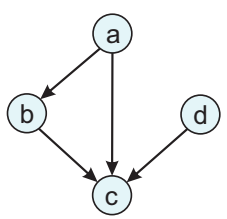
Undirected graph, also graph

- *E.g. a city map without one-way streets.*
- $G = (V, E)$ is composed of vertices V and undirected edges $E \subset V \times V$ representing an unordered relation between two vertices.
- The number of vertices $|V| = n$ and the number of edges $|E| = m = \mathcal{O}(|V|^2)$ is assumed finite.



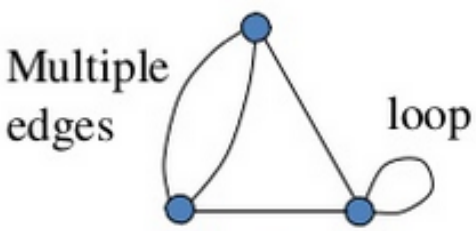
Directed graph

- *E.g. a city map with one-way streets.*
- $G = (V, E)$ is composed of vertices V and directed edges E representing an ordered relation between two vertices.
- Oriented edge $e = (u, v)$ has the tail u and the head v (shown as the arrow $\rightarrow$). The edge e is different from $e' = (v, u)$ in general.

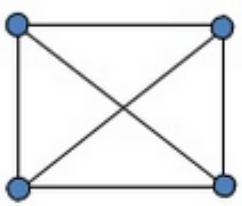


Subgraph, simple graph

- A graph H is a subgraph of graph G if and only if its vertex and edge sets are subset of corresponding sets in graph G.
- A simple graph is a graph, which has no loops and multiple edges.



It is not simple.



It is a simple graph.

Simple graph, multigraph

- Aside: Set and bag (also multiset): The elements of set are distinct. Sometimes we need duplicates. E.g. there are duplicated persons with the same given name and surname in Prague. Mathematically, a bag is a set with duplicates.
- A multigraph allows multiple edges between the same vertices, i.e. E is a bag of undirected edges.

  E.g., the call graph in a program (a function can get called from multiple points in another function).

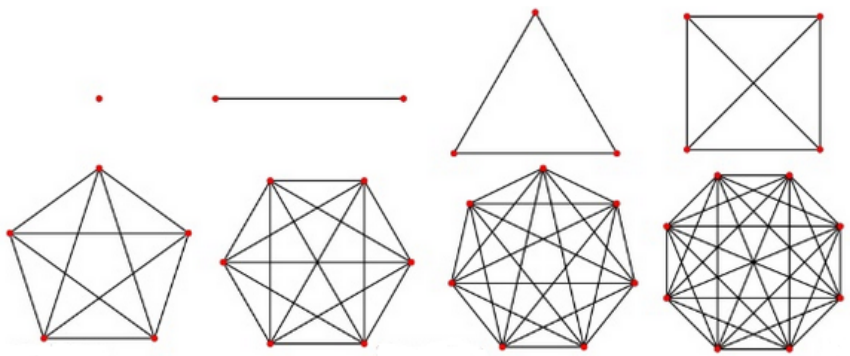
  Multigraphs do not depict relations. We can label edges in a multigraph to distinguish edges.

Vertex degree, the even degree property

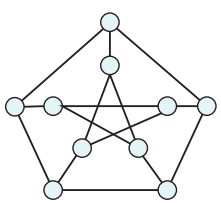
- The degree $\deg(v)$ of a vertex V is its number of incident edges.
  - A self-loop counts for 2 in the degree function.
  - An isolated vertex has degree 0.
- Proposition (trivial):
  The sum of the degrees of a graph or multigraph $G = (V, E)$ equals $2\,|E| = 2\,m$.
- Corollary (trivial):
  The number of vertices of odd degree is even.
- The handshaking theorem: Let $G(E, V)$ be an undirected graph. Then $2|E| = \sum_{v \in V} \deg(v)$.
  In any group of people the number of people who have shaken hands with odd number of other people from the group is even. Zero is an even number.

Complete graph, k-regular graph

- Complete graph K_n is a simple graph, in which every pair of vertices are adjacent.
- If number of graph vertices $= n$ then there are $\frac{n(n-1)}{2}$ edges in a corresponding complete graph.



- A k-regular graph is a simple graph with vertices of equal degree k.



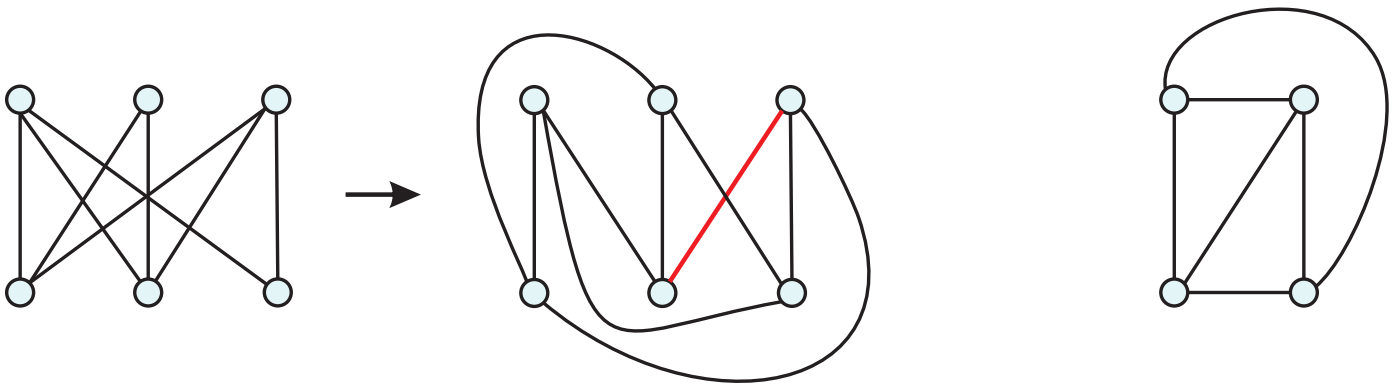
- The complete graph K_n is $(n-1)$-regular.

Dense/sparse graph

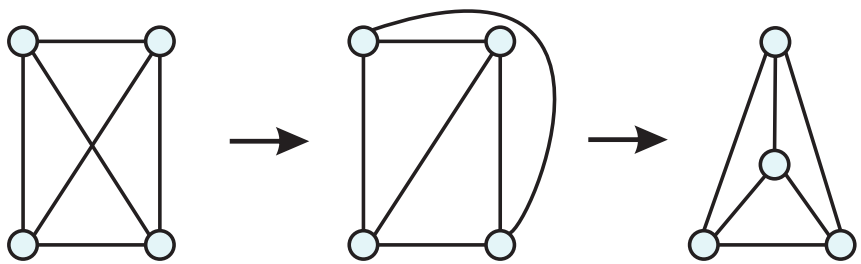
- Running times are typically expressed in terms of $|E|$ and $|V|$.
- The graph is dense if $|E| \approx |V|^2$.
- The graph is sparse if $|E| \approx |V|$.
- Many interesting graphs are sparse.
  E.g., planar graphs, in which no edges cross, have $|E| = \mathcal{O}(|V|)$ *by Euler's formula.*
- If you know you are dealing with dense or sparse graphs, different data structures may make sense.

Planar graph

- A planar graph can be drawn on a plane without crossing edges.

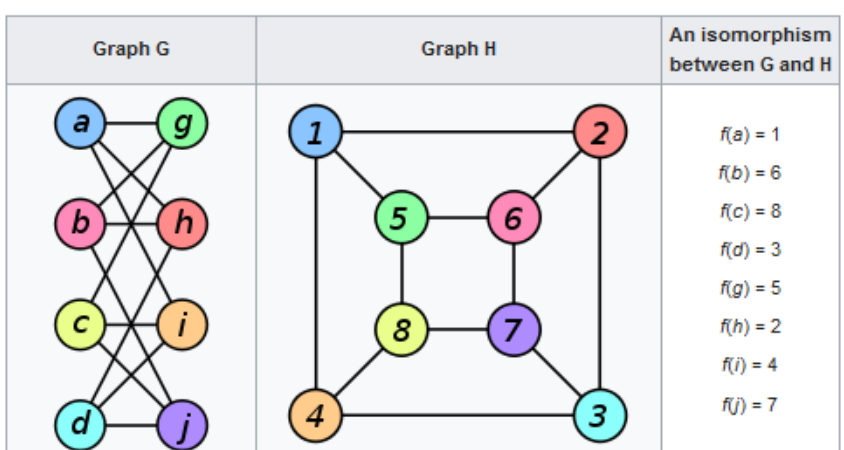


- Proposition (I. Fáry, 1948, independently K. Wagner 1936, S.K. Stein 1951): Any simple planar graph can be drawn without crossings so that its edges are straight line segments.



Graph isomorphism (1)

- The isomorphism of graphs G, H denoted $G \cong H$ is the bijection f between their set of vertices $V(G)$, $V(H)$ written as $f: V(G) \to V(H)$ such that any two vertices $v_i, v_j \in V(G)$ adjacent in G.
  and if and only if vertices $f(v_i)$ and $f(v_j)$ are adjacent in H.
- The function f is called the isomorphism.
- Said informally: Two graphs are isomorphic if they differ only in their drawing (i.e. how their vertices and edges are labeled).



Graph isomorphism (2)

- Needed conditions for the existence of the graph isomorphism:
  - The same number of graph vertices and graph edges.
  - The degrees of all vertices are the same.
- Finding the graph isomorphism is still believed to be NP hard.

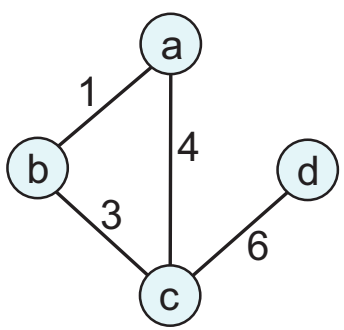
Undirected/directed weighted graph

- A weighted graph associates numerical weights (often non-negative integers) with either the edges or the vertices or both.

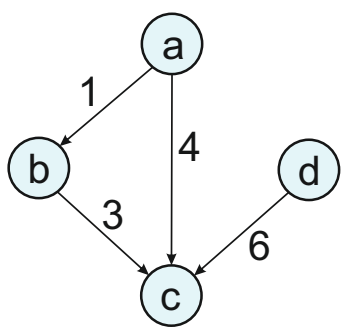
  E.g., a road map: graph edges are weighted with distances.

- A weighted graph is a special type of labeled graph.

Undirected weighted graph



Directed weighted graph



Examples of edge-weighted graphs

- Vertices correspond to towns. Edges are roads connected towns. Each edge is weighted by a distance between corresponding towns.
- Vertices match to translation stations for transmitting/receiving GSM signal. Edges are available radio connections between pairs of translation stations. Each edge is attributed by a communication channel capacity between corresponding translation stations.
- Vertices are students. Edges are bonds between students who know each other. Edges are attributed by an integer between 0 and 5 expressing how much the students like each other.

Courtesy: Marcel Jiřina, jr.

Examples of vertex-weighted graphs

- Vertices match to students. Edges correspond to relations to students who know each other. Vertices are attributed by the student sex; 0=male student or 1=female student.
- Vertices correspond to car dealer shops. Edges express that the shops sell cars of the same brand (e.g. Škoda). Each vertex tells how many cars of the brand the shop has in stock.
- Vertices are towns. Edges are roads connected towns. Each vertex is attributed by the number of ambulance cars available currently in the town.

Courtesy: Marcel Jiřina, jr.

Walking in a graph, cycles

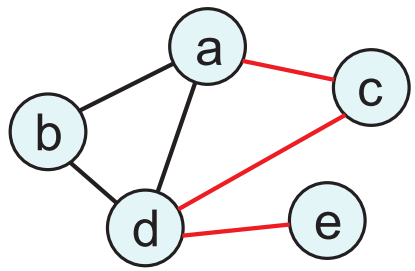
- A walk of length k from vertex v_0 to vertex v_k is a non-empty graph $P = (V, E)$ of the form $V = \{v_0, v_1, \ldots, v_k\}$, $E = \{(v_0, v_1), \ldots, (v_{k-1}, v_k)\}$, where edge j connects vertices $j-1$ and j (i.e. $|V| = |E| + 1$).
- A path from a vertex u to a vertex v is a sequence $(v_0, v_1, \ldots, v_k)$ of vertices, where $v_0 = u$, $v_k = v$, and $(v_i, v_{i+1}) \in E$ for $i = 0, 1, \ldots, k-1$.

  Note: We will also simplify notation for a path from vertex u to vertex v as path $u \rightsquigarrow v$.
- A simple path is a path, in which all vertices, except possibly the first and the last, are different (equivalently: do not appear more than once).
- A walk or path or simple path is closed when $v_0 = v_k$.
- A cycle is a walk with different vertices except for $v_0 = v_k$.

Length of a path, connected graph

- The length of a path is defined as the number of edges in the path.

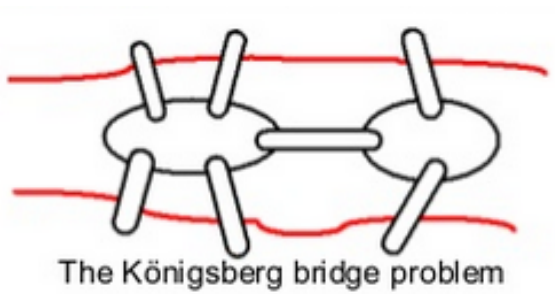
Example: The red edges show a simple path acde *from vertex* a *to vertex* e *of length 3. We could also consider a non-simple path* acdabde *of length 6.*

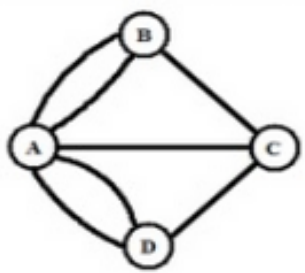


- If the graph is weighted then the length of the path is the sum of edges weights in the path.
- If the graph G is connected then there is a path between every pair of vertices. $|E| \geq |V| - 1$.
- Cliques are connected subgraphs in a bigger graph.

Graph cycle

- A cycle of a graph G is a subset of the edge set of G that forms a path such that the first vertex of the path corresponds to the last vertex.
- Hamilton cycle is a cycle that uses each graph vertex of a graph exactly once. A very hard problem. Still unsolved.
- Euler cycle (for undirected connected graphs) is a sequence of vertices that visits every edge exactly once and comes back to the starting vertex.
  - A single stroke/line drawing task.
  - Euler cycle exists if and only if G is connected and each vertex has an even degree. Leonhard Euler formulated this precondition on 7 bridges of Königsberg example in 1736.


The Königsberg bridge problem



Euler cycle, a constructive existence proof

- Pick any vertex in G and walk randomly without using the same edge more than once.
- Each vertex is of even degree, so when you enter a vertex, there will be an unused edge you exit through.

  (Except at the starting point, at which you can get stuck.)
- When you get stuck, what you have is a cycle
  - When you get stuck, what you have is a cycle.
  - Glue the cycles together to finish!

Graph traversal

- Graph traversal (called also graph search) visits (checks and/or updates) each vertex of a graph.
- The goal is to methodically explore every vertex and every edge of the graph.
- The most basic graph algorithm is the one that visits vertices of a graph in a certain order.
- Graph traversal is used as a subroutine in many other algorithms.
- Two basic algorithms:
  - Depth-First Search (DFS): uses recursion (stack).
  - Breadth-First Search (BFS): uses queue.
- Graph traversing/searching builds a tree on a graph depicting in which order the vertices/edges were traversed.

Depth-first search

- Consider a graph $G(V, E)$ and an arbitrary initial graph vertex $v \in V$.
- DFS(v) visits all the vertices reachable from v in a depth-first order, i.e. explores as far as possible along each branch before backtracking.
  - Mark vertex v as visited.
  - For each graph edge $v \to u$
    If u is not visited, call DFS(u).

Worst case time performance: $\mathcal{O}(|V| + |E|)$.

Use non-recursive version if recursion depth is too big (over a few thousands), i.e. replace recursive calls with a stack.

Breadth-first search

Consider a graph $G(V, E)$ and an arbitrary initial graph vertex $v \in V$.

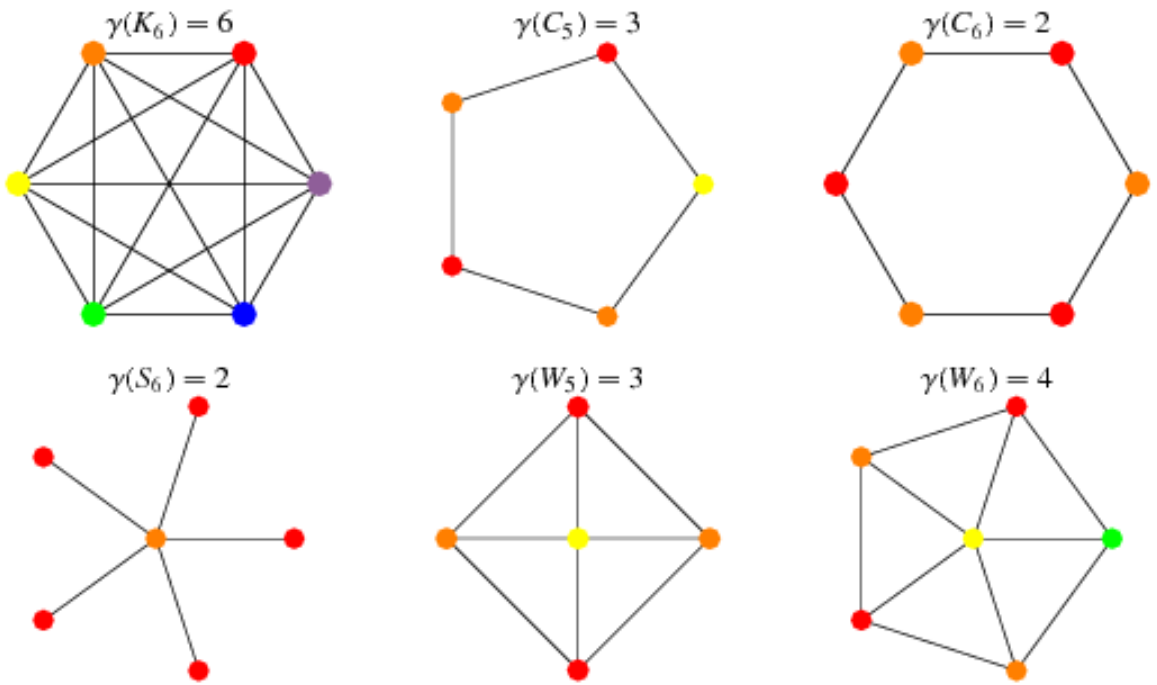
BFS(v) visits all the vertices reachable from v in a breath-first order, i.e. explores all of the neighbor vertices at the present depth prior to moving on to the vertices at the next depth level.

- Initialize a queue Q.
- Mark v as visited and push it to Q.
- While Q is not empty:
  - Take the front element of Q, i.e. a vertex w
  - For each graph edge $w \to u$:
    If u is not visited, mark it as visited and push it to Q.

Worst case time performance: $\mathcal{O}(|V| + |E|)$.

Chromatic number

The chromatic number of a graph G, written $\gamma(G)$, is the minimum number of colors needed to label the vertices so that adjacent vertices receive different colors.

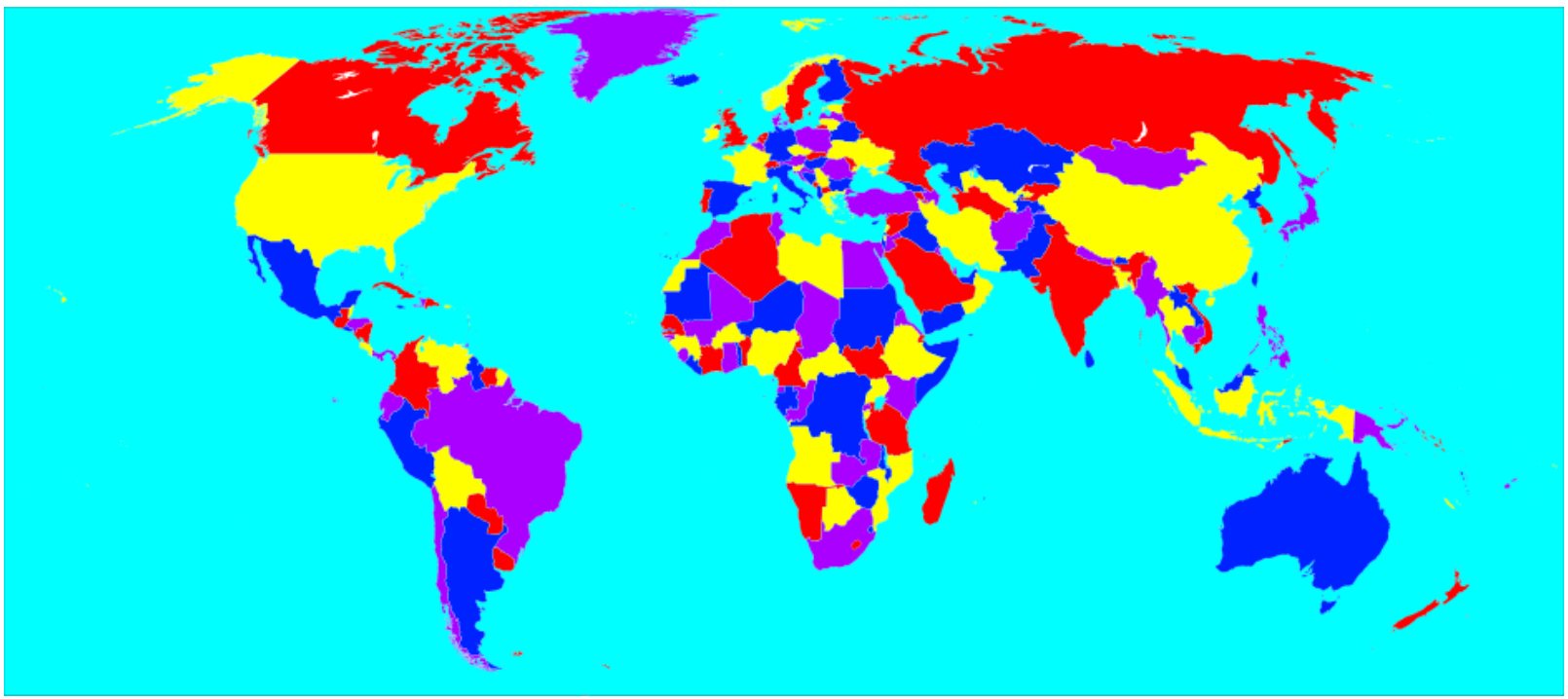


Maps and their coloring

- A map is a partition of the plane into connected regions, e.g. a political country map.
- Can we can color the regions of every map using four colors at most so that neighboring regions have different colors?
- Map coloring $\rightarrow$ graph coloring
  - A map region $\rightarrow$ a graph vertex.
  - Adjacency between regions $\rightarrow$ an edge in the graph.

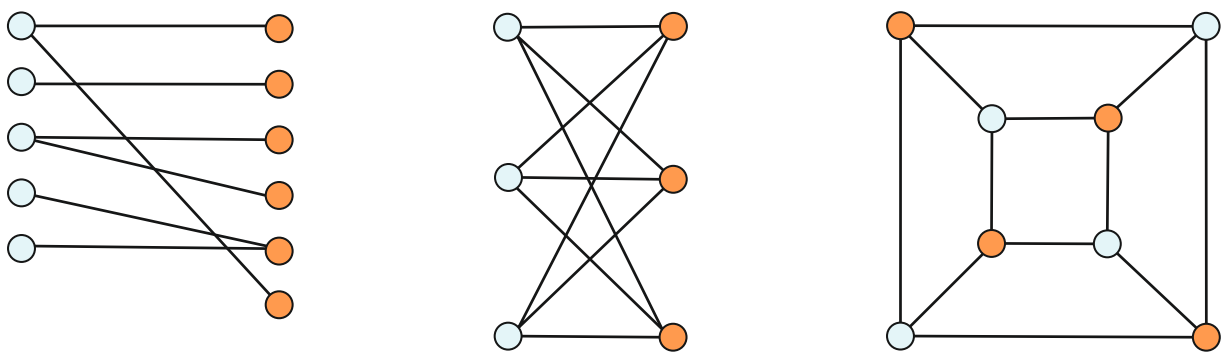
Four color map theorem

- Four color map theorem states: given any separation of a plane into contiguous regions, producing a figure called a map, no more than four colors are required to color the regions of the map so that no two adjacent regions have the same color.
- Theorem was proved in 1976 as the first major theorem proved by computer.



A special graph: bipartite graph

- A bipartite graph has graph vertices decomposed into two disjoint sets such that no two graph vertices within the same set are adjacent. Used, e.g. for matching problems.

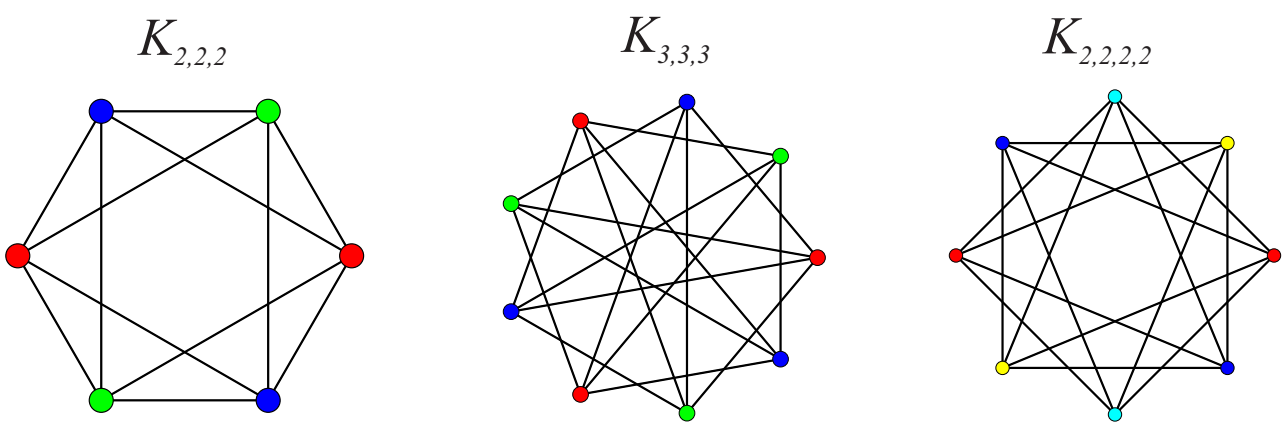


Courtesy: Wolfram MathWorld.

- A bipartite graph is a graph that does not contain any odd-length cycles.
- Bipartite graphs are equivalent to two-colorable graphs.
- A bipartite graph is a special case of a k-partite graph with $k = 2$.

k-partite graph

- Also named a multipartite graph.
- A k-partite graph is a graph whose vertices are or can be partitioned into k different independent sets.
- Equivalently, it is a graph that can be colored with k colors, so that no two endpoints of an edge have the same color.
- The notation is used with a capital letter K subscripted by a sequence of the sizes of each set in the partition. For instance, $K_{2,2,2}$ is the complete tripartite graph of a regular octahedron.



Examples of complete k-partite graphs

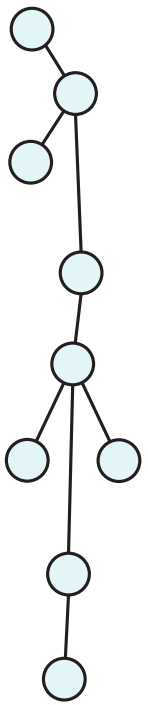
A special graph: acyclic graph

- A graph containing no cycles of any length is known as an acyclic graph. Other graphs are cyclic.
- An acyclic graph is bipartite.
- A cyclic graph is bipartite iff all its cycles are of even length.

  Skiena, S.: Cycles in Graphs. §5.3 in Implementing Discrete Mathematics: Combinatorics and Graph Theory with Mathematica. Reading, MA: Addison-Wesley, pp. 188-202, 1990.

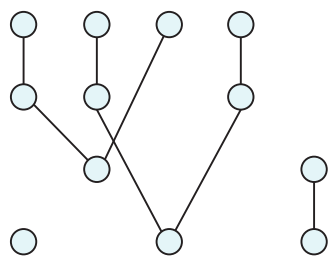
A special graph: tree

- A tree is an acyclic connected graph (in which any two vertices are connected by exactly one path).
- Every acyclic connected graph is a tree, and vice versa.
- The tree is the most important type of special graphs because many problems are easy to solve on trees.
- Alternative equivalent tree definitions:
  - A connected graph G with $|E| = |V| - 1$ edges.
  - An acyclic graph with with $|V| - 1$ edges.
  - There is exactly one path between every pair of vertices.
  - An acyclic graph but adding any edge results in a cycle.
  - A connected graph but removing any edge disconnects it.



A special graph: forest

- A forest is an acyclic undirected graph.
- Equivalently, a forest is a disjoint union of trees, i.e. its connected components are trees.



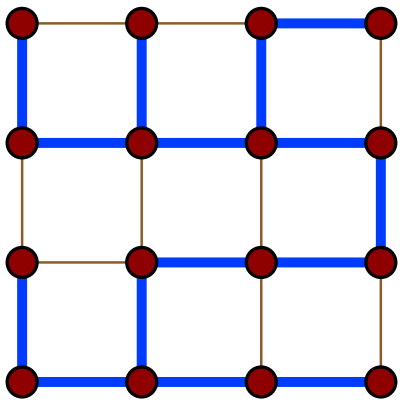
- Examples of the forest special cases: an empty graph, a single tree, and the discrete graph on a set of vertices (that is, the graph with these vertices that has no edges).
- Since for every tree $|V|$ - $|E| = 1$, we can easily count the number of trees that are within a forest by subtracting the difference between total vertices and total edges.

A special graph: directed acyclic graph (DAG)

- A directed acyclic graph (DAG) is a directed graph without cycles.
- Every DAG must have at least one vertex with in-degree zero (source) and at least one vertex with out degree zero (sink).
- DAG may have more than one source or sink.
- DAG is a directed graph that has a topological ordering, i.e. a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v, u comes before v in the ordering.
- DAGs arise in modeling many problems involving prerequisite constraints as construction projects, course prerequisites, document version control.

Spanning tree

- Definition: A spanning tree T of an undirected graph G is a subgraph that is a tree, which includes all of the vertices of G.
- A graph, which is not a tree, has more than one spanning tree.
- If a graph is not connected than it will not contain a spanning tree. (cf. spanning forest).
- If all of the edges of G are also edges of a spanning tree T of G, then G is a tree and is identical to T (that is, a tree has a unique spanning tree and it is itself).
- A complete graph with n vertices has $2^{(n-2)}$ different spanning trees.



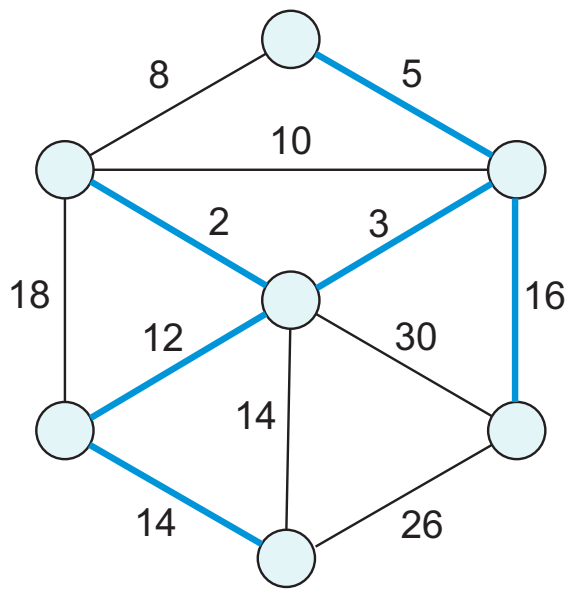
One of several possible spanning trees of a 4 × 4 grid with 4-neighborhood.

Courtesy: Wikipedia

Minimum spanning tree (MST)

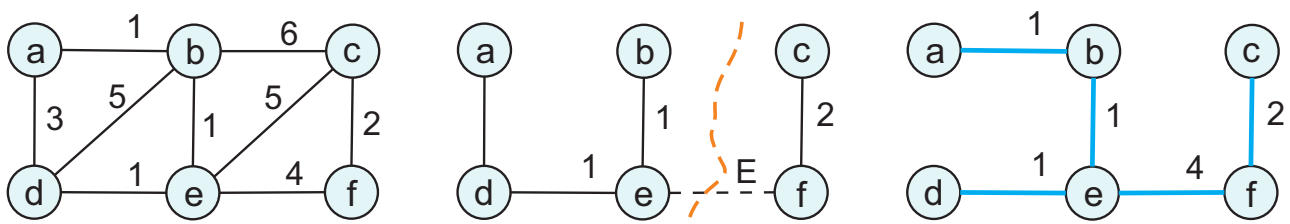
- Given: Given an undirected weighted graph $G = (V, E)$, see page 12.
- Task: Find a subset of E with the minimum total weight that connects all the vertice s into a tree.

A minimum spanning tree example. Blue edges constitute the minimum spanning tree.



Cycle and cut MST properties

- Cycle property: For a cycle C in the graph, if the weight of and edge E of C is larger than the individual weights of all other edges of C, then this edge cannot belong to an MST. (See Wikipedia for proof).
- Cut property: For any cut C of the graph, if the weight of an edge E in the cut-set of C is strictly smaller than the weights of all other edges of the cut-set of C, then this edge belongs to all MSTs of the graph. (See Wikipedia for proof).



*E*xample: T is the only MST of the graph. If $S = \{a, b, c, d\}$, $V \setminus S = \{c, f\}$. There are only 3 possible edges across the cut$(S, S \setminus V)$, i.e. bc, ec, ef of the original graph. Edge E is the only minimum weight edge for the cut and therefore $S \cup \{E\}$ is part of MST T.

Classic MST algorithms overview

1. Borůvka algorithm, Otakar Borůvka 1926, purpose: efficient electricity distribution network in Moravia, even before the graph theory was introduced.

   The algorithm proceeds in a sequence of stages. In each stage, called Borůvka step, it identifies a forest F consisting of the minimum-weight edge incident to each vertex in the graph G, then forms the graph $G_1 = G \setminus F$ as the input to the next step. Here $G \setminus F$ denotes the graph derived from G by contracting edges in F (by the Cut property, these edges belong to the MST). Each Borůvka step takes linear time.

2. Prim algorithm, discovered by Vojtěch Jarník in 1929 in a letter to Otakar Borůvka, paper in 1930, reinvented by Robert Prim in 1957.

3. Kruskal algorithm, by Joseph Kruskal 1956.

Shortest paths in a weighted directed graph

- Assume a weighted directed graph $G(V, E)$ with two selected special vertices s (source) and t (target).
- The goal is to find a shortest path from the source s to the target t. Image, e.g., that we seek a shortest path from Prague to Vienna in the road map.
- The weights on graph edges may be negative (maybe counter to the intuition). Negative edges bring complications, because the presence of a negative cycle might imply that there is no shortest path.
- In general, a shortest path $s \rightsquigarrow t$ exists if and only if there is at least one path from $s \rightsquigarrow t$.
- However, there is no path $s \rightsquigarrow t$ that touches a negative cycle. If any negative cycle is reachable from s and can reach t, we can always find a shorter path by going around the cycle one more time.

Single source shortest path, introduction

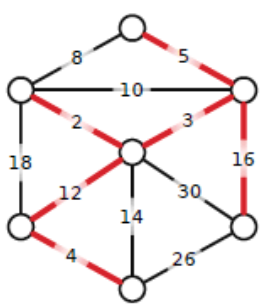
- Almost every algorithm known for solving shortest path actually solves (large portions of) the more general single source shortest path (also SSSP problem): Find the shortest path from the source vertex s to every other vertex in the graph.
- This problem is usually solved by finding a shortest path tree rooted at the source vertex s that contains all the desired shortest paths.
- If shortest paths are unique, then they form a tree (because any subpath of a shortest path is itself a shortest path).
- If there are multiple shortest paths to some vertices, we can always choose one shortest path to each vertex so that the union of the paths is a tree.

- To simplify the algorithms explanation in this lecture, we consider only directed graphs. All of the algorithms also work for undirected graphs with some minor modifications, but only if negative edges are prohibited.

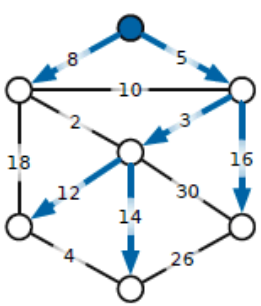
Minimal spanning trees × shortest path trees

Minimal spanning trees

- Minimum spanning trees are unrooted and undirected.
- Only undirected graphs have minimum spanning trees.
- If edge weights are distinct, there is only one minimum spanning tree.



minim. spanning tree



shortest path tree

Shortest path trees

- Shortest-path trees are rooted and directed.
- Shortest-path trees are most naturally defined for directed graphs.
- Every source vertex induces a different shortest-path tree.
- It is possible for every shortest path tree to use a different set of edges from the minimum spanning tree.

Single source shortest path

- There are several SSSP algorithms. All of them are special cases of a generic algorithm by L. Ford (1956) and G. Dantzig (1957).
- Each graph vertex v stores two values, which describe a tentative sortest path from the source vertex s to v (inductively).
  - $\mathrm{dist}(v)$ is the length of the tentative shortest path $s \rightsquigarrow v$, or ∞ if there is no path.
  - $\mathrm{pred}(v)$ in the predecessor of v in the tentative shortest path $s \rightsquigarrow v$, or Null if there is no such vertex.
- The predecessor pointers automatically define a tentative shortest path tree; they play the same role as the parent pointers in our generic graph traversal algorithm.
- Tense edge:

  We call a graph edge $u \to v$ tense if $\mathrm{dist}(u) + w(u \to v) < \mathrm{dist}(v)$, where $w(u \to v)$ is the weight of the edge $u \to v$.

Single source shortest path algorithm

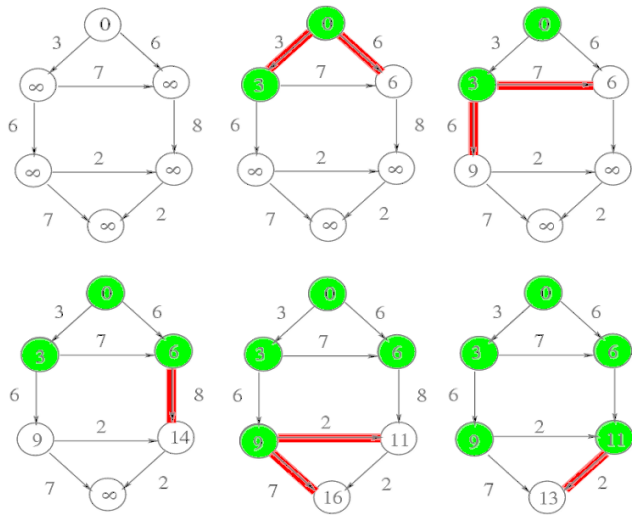
- We have seen that performing a depth-first-search or breath-first-search produces a spanning tree. Neither of those algorithms takes edge weights into account.
- There is a simple, greedy Dijksta SSSP algorithm that will solve this problem. We assume that there is a path from the source vertex s to every other vertex in the graph.
- Let V be the set of vertices whose minimum distance from the source vertex has been found. Initially, V contains only the source vertex s.
- The algorithm is iterative, adding one vertex to V on each pass.
- Greed: On each iteration, add to V the vertex not already in V for which the distance to the source is minimal.

Dijkstra algorithm for the shortest path

- s – source vertex
- $\mathrm{dist}(j)$ – the minimal distance from vertex s to vertex j
- $\mathrm{pred}(j)$ – predecessor of the vertex j

Dijkstra algorithm (1956) finds the shortest vertex from the source vertex s to all vertices.

% N – set of all graph vertices $V := s$ % visited vertices;
$U := N \setminus s$ % unvisited vertices;
$\mathrm{dist}(s) := 0$, $i := s$;
while $|V|<|N|$ **do**
    $\mathrm{choose}(i, j) : d(j) := \min_{k,m}\{\mathrm{dist}(k) + c_{km} \mid k \in V, m \in U\}$;
    $U = U \setminus \{j\}$;
    $V = V + \{j\}$;
    $\mathrm{pred}(j) := i$;
end



- Incrementally labels vertices with their distance-from-start.
- Produces optimal (shortest) paths.
- Performance $\mathcal{O}(|N|^2)$ with the heap reshuffling $\mathcal{O}(|N| \log |N|)$.

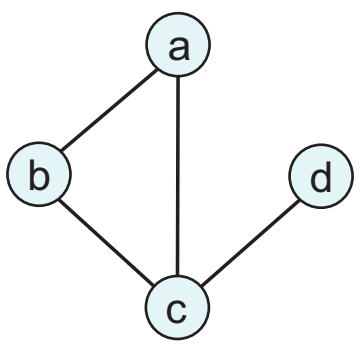
Graphs representations

- By a drawn figure (schema, drawing).
- By listing set of vertices, edges.

- By an adjacency list (suited for computer implementation).
- By an adjacency matric (suited for computer implementation).

Graph represented by the adjacency matrix

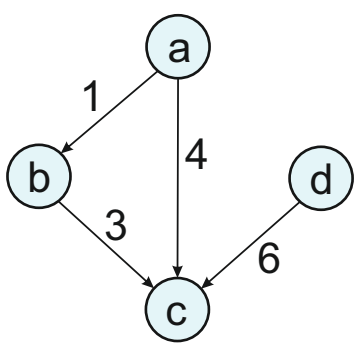
- Adjacency relationship is:
  - Symmetric if the graph G is undirected.
  - Not necessarily so if G is directed.
- Adjacency matrix A has elements a_{ij}, $i, j = 1, \ldots, |V|$.
  - Elements $a_{ij} = 1$ if graph vertices i, j share an edge; 0 otherwise.
  - Uses $\Omega(|V|)$ memory.



	a	b	c	d
a	0	1	1	0
b	1	0	1	0
c	1	1	0	1
d	0	0	1	0

Similarity matrix

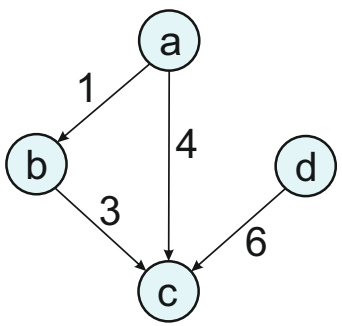
- In the case of the weighted graph, the adjacency matrix is generalized to the similarity matrix S of the graph.
- Elements of A have value of the weight of the edge between two respective vertices, $a_{ij} = w_{ij}$.



	a	b	c	d
a	0	1	4	0
b	0	0	3	0
c	0	0	0	1
d	0	0	6	0

Graph represented by the adjacency list

- Adjacency list: For each vertex $v \in V$, store a list of vertices adjacent to v (in other words: going out of v).
  - Easy to iterate over edges incident to a certain vertex.
  - The lists have variable lengths.
  - Asymptotic bound of the space usage from above: $\Omega(|E| + |V|)$.
- Our example:



- `Adj[a] = {b, c}`
- `Adj[b] = {c}`
- `Adj[c] = { }`
- `Adj[d] = {c}`

From	To
a	b, c
b	c
c	empty
d	c

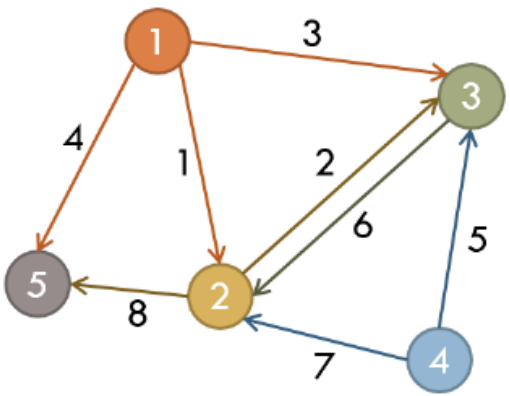
- Variation: For oriented graphs, it is possible to store the list of graph edges coming into the vertex.

Implementing adjacency list

Three solutions:

1. Using linked lists
   - Too much memory / time overhead.
   - Using dynamically allocated memory or pointers is bad.
2. Using arrays of vectors
   - Easier to code, no bad memory issues.
   - Very slow.
3. Using arrays
   - Assuming the total number of edges $|E|$ is known.
   - Very fast and memory efficient.

Implementation using arrays, example



ID	To	Next Edge ID
1	2	-
2	3	-
3	3	1
4	5	3
5	3	-
6	2	-
7	2	5
8	5	2

From	1	2	3	4	5
Last Edge ID	4	8	6	7	-

Courtesy: Jaehyun Park, Stanford University

Graphs. Implementation using arrays (1)

- Have two arrays; E of size $|E|$ and LE of size $|V|$.
  - E contains the edges.
  - LE contains the starting pointers of the edge lists.
- Initialize LE[i] = -1 for all i
  - LE[i] = 0 is also fine if the arrays are 1-indexed.
- Inserting a new edge from vertex u to vertex v with ID k

  E[k].to = v

  E[k].nextID = LE[u]

  LE[u] = k
- Iterating over all edges starting at u

```
for (ID = LE[u]; ID != -1; ID = E[ID].nextID
    // E[ID] is and edge starting from u
```

Graphs. Implementation using arrays (2)

- Once built, it is hard to modify the edges.
- The graph better be static!
- However, adding more edges is easy.

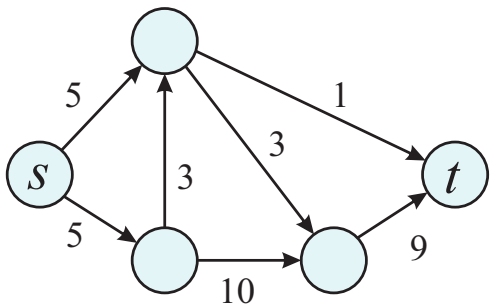
Networks and flows

- A network is a directed graph $N = (V, E)$ with a source vertex s and a terminal vertex t (also sink). Each graph edge has a positive capacity $c(e) > 0$.
- A flow is a real-valued (often integer) function $f\colon V^2 \to \mathbb{R}^+$ associated with each edge $e = (u, v)$ with the properties ($\approx$ G. Kirchhoff's law, 1845):
  - Capacity c constraint: $\forall\ u, v \in V,\ f(u, v) \leq c(u, v)$.
  - Skew symmetry: $\forall\ u, v \in V,\ f(u, v) = -f(v, u)$.
  - Flow conservation: $\forall\ u \in (V \setminus \{s, t\}),\ \sum_{v \in V} f(u, v) = 0$.
- The total flow F of the networks is then what leaves s or reaches t

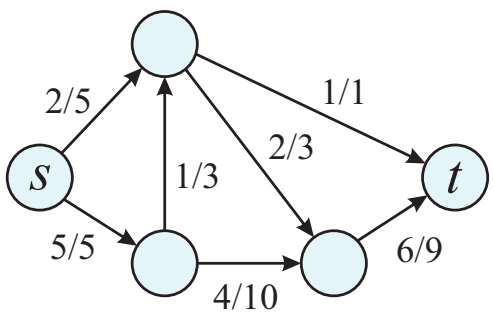
$$F(N) = \sum_{u \in V} f(s, u) - \sum_{u \in V} f(u, s) = \sum_{u \in V} f(u, t) - \sum_{u \in V} f(t, u)$$

Network and flows example

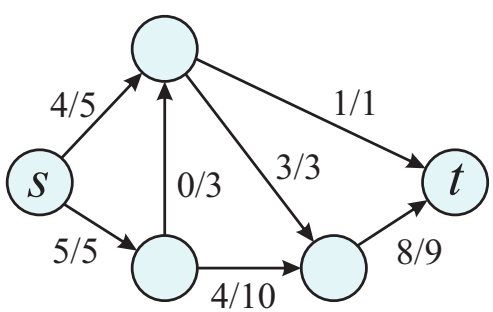
- Network example



- Flow example, $f(s,t) = 7$.



- Maximal flow example, $f(s,t) = 9$.

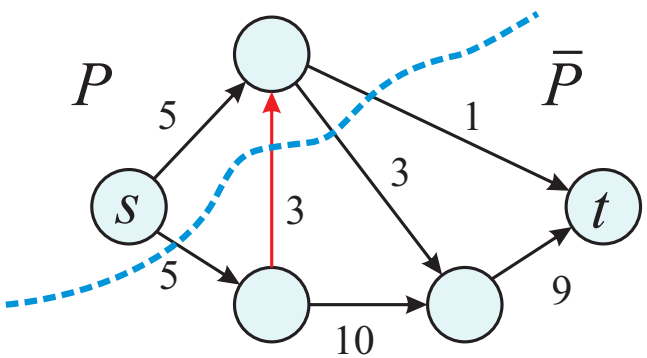


Cut of a network

- A cut of a network is a partition of the vertex set $V = P \cup \bar{P}$ in to two disjoint sets P (containing s, source) and $\bar{P}$ (containing t, terminal, sink).
- The capacity of a cut κ is a sum of the capacities between edges (u, v) between P and $\bar{P}$

$$\kappa(P, \bar{P}) = \sum_{u \in P; v \in \bar{P}} c(u, v)$$

- Example continued:



The capacity of the cut for the example above is $\kappa = 5 - 3 + 3 + 1 = 6$.

Graph cuts properties

- Let $(P, \bar{P})$ be be any cut of a network $N = (V, E)$ then the associated flow is given by

$$F(N) = \sum_{u \in P; v \in \bar{P}} f(u, v) - \sum_{u \in P; v \in \bar{P}} f(v, u)$$

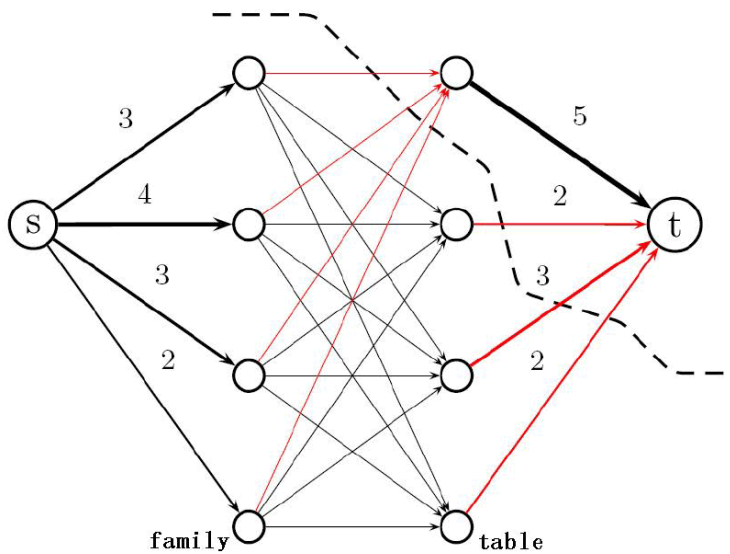
Idea of the proof:

- Show first that $F(N) = \sum_{u \in P} \left(\sum_{v \in \bar{P}} f(u, v) - \sum_{v \in \bar{P}} f(v, u) \right)$ by summing all contributions in P and using conservation.
- For all $v \in P$ the term between brackets is zero (conservation).
- Hence we need only to keep the edges across the partition.

- A minimal cut (with minimal capacity) also bounds $f(N)$.
We will construct one and will see that it equals to $f(N)$.

Graph cut application, dining problem

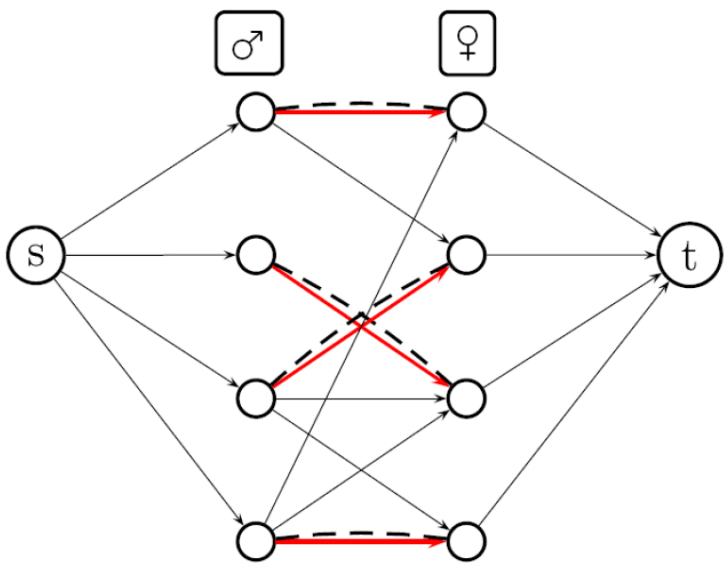
Can we seat 4 families with number of members (3,4,3,2) at 4 tables with number of seats (5,2,3,2) so that no two members of a same family sit at the same table?



The central edges are the table assignments (a capacity of 1). The cut shown has a capacity 11 which upper bounds $f(N)$. We can therefore not seat all 12 members of the four families.

Graph cut application, marriage problem

One wants to find a maximimum number of couplings between men and women where each couple has expressed whether or not this coupling was acceptable (central edges that exist or not).



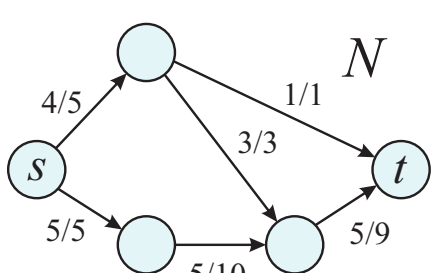
One wants to find a maximum number of disjoint paths in this directed graph. All the capacities of the existing edges are 1.

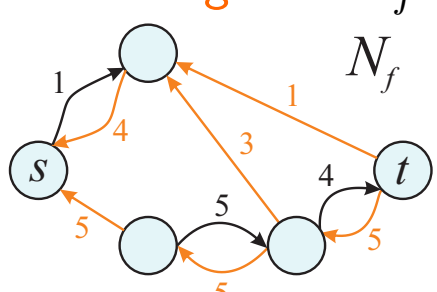
Residual network N_f

- The residual capacity of an edge with respect to a flow f is the difference between the edge capacity and its flow.
- Given a network $N(V, E)$ with a flow f on it, its residual network $N_f(V, E_f)$ is a network with the same set of vertices V and edges E_f but with new residual capacities c_f

$$c_f(u,v) = \begin{cases} c_f(u,v) - f(u,v) > 0 & \text{if } (u,v) \in E \text{ (forward edges)} \\ f(v,u) & \text{if } (v,u) \in E \text{ (backward edges)} \\ 0 & \text{otherwise} \end{cases}$$

- That means that there are units of flow $f(u, v)$, which we can undo by pushing the flow backward.
- Notice, if $c(u, v) = f(u, v)$ then there are only backward edges in N_f.





Augmenting path

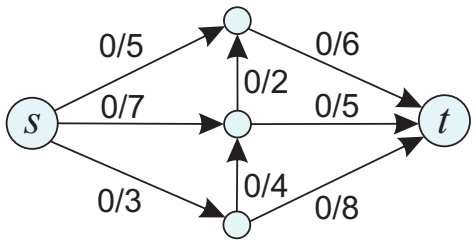
- Augmenting means making the flow larger here. The residual capacity of an edge tells what is the maximum amount the flow can be increased.
- An augmenting path is a path constructed by repeatedly finding a path of positive capacity from a source to a sink and then adding it to the flow.
- The augmenting path is a directed path $v_0, \ldots, v_k$ from $S = v_0$ to $t = v_k$ for which

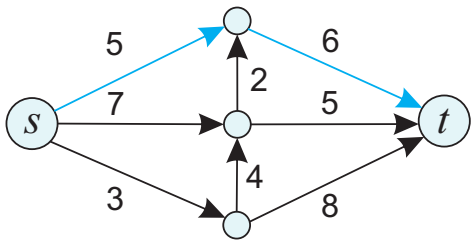
$$\begin{aligned} \Delta_i &= c(v_i, v_{i+1}) - f(v_i, v_{i+1}) > \forall (v_i, v_{i+1}) \in E \text{ or} \\ \Delta_i &= c(v_{i+1}, v_i) - f(v_{i+1}, v_i) > \forall (v_{i+1}, v_i) \in E \end{aligned}$$

Example, augmenting path 1

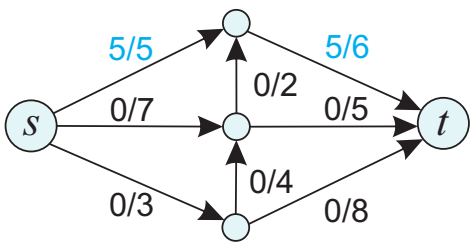
At each step, the graph (left) and the residual graph is displayed. Augmentation paths are shown in blue.

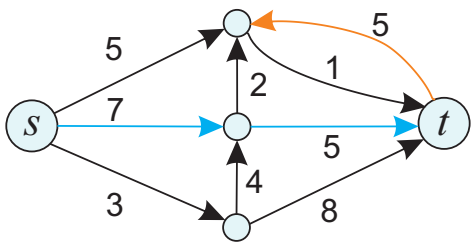
Step 1



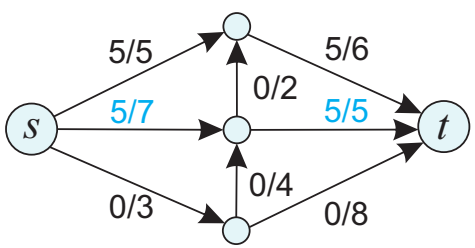


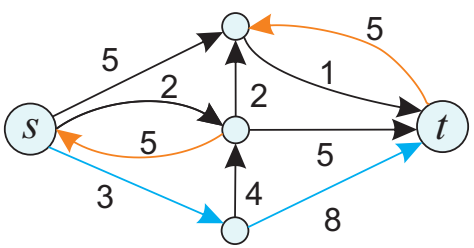
Step 2





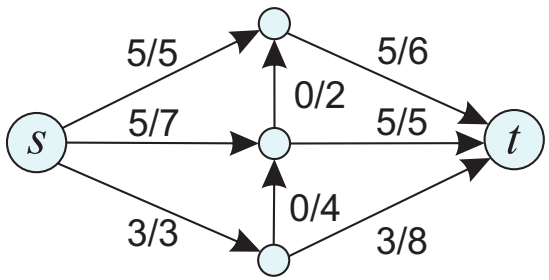
Step 3

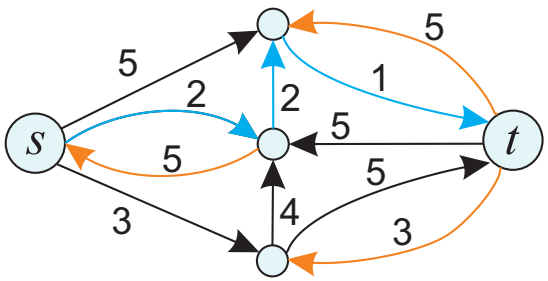




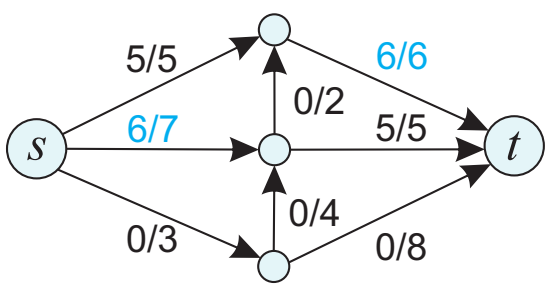
Example, augmenting path 2

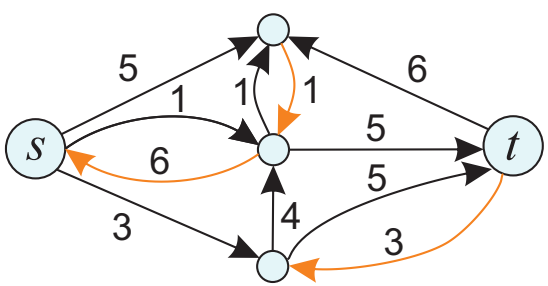
Step 4





Step 5





We find $F(N) = 14$ in five steps.

Max flow, min-cut

- A network $N(V, E)$ is at maximum flow if and only if there is no augmenting path in the residual network N_f.
- In a network N the following are equivalent
  1. A flow is optimal.
  2. The residual graph does not contain an augmenting path.
  3. $f(N) =$ capacity $\kappa(P, \bar{P})$ for some cut$(P, \bar{P})$
     The value of the optimal flow thus equals $f(N) = \min \kappa(P, \bar{P})$.
- Finding maximal flow / minimal cut becomes an linear programming task if flows $f(i, j)$ on edges (i, j)

$$\max \left(\sum_{i:(s,i)} f(s,i) = \sum_{i:(i,s)} f(i,s) \right) \text{ subject to } \sum_i f(i,j) = \sum_i f(j,i)$$

$$\text{and } 0 \leq f(i,j) \leq c(i,j)$$

Ford-Fulkerson algorithm

- The Ford-Fulkerson algorithm (1956) calculates this optimal flow using augmentation paths.

Algorithm MaxFlowFF(N, s, t)
$f(u, v) := 0 \quad \forall (u, v) \in E$;
while N_f *contains a path from* s *to* t **do**
    choose an augmentation path A_p from s to t
    $\Delta := \min_{(u,v) \in A_p} \Delta_i$
    Augment the flow by Δ along A_p
    Update N_f
end

- Finding a path in the residual graph can be implemented with a BFS or DFS exploration.
- At each step we show the graph (left) and the residual graph (right).
- Augmentation paths are in red. In 5 steps, we find $f(N) = 14$.