Scuola universitaria professionale
della Svizzera italiana
SUPSI

Università
della
Svizzera
italiana

Istituto Dalle Molle di studi sull'intelligenza artificiale

Algorithms and Data Structures

Algorithms on graphs

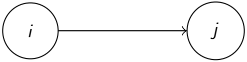
Matteo Salani
matteo.salani@idsia.ch

Graph basics

A **Graph** $G = (N, E)$ is a mathematical object that is suitable to represent a large class of decision problems.

- N: set of nodes (vertices)
- E: set of node pairs (edges)

A **Digraph** $D = (N, A)$ (oriented graph) instead of edges there are arcs (ordered pair of nodes).

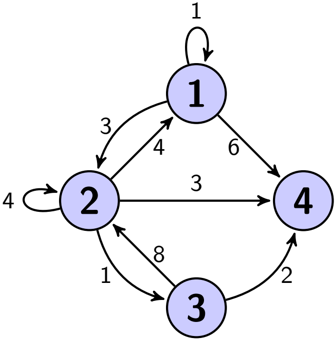


There are also multi-graphs (mode edges/arcs) and hyper-graphs (nodes as set of nodes) but it goes beyond the purpose of this introduction.

Graph basics

Incidence matrix: is one of the possible ways to define a graph.
It can be used to define the **weighting** of a graph. The weighting is a **function** that defines one or more attributes to an edge/arc.
In the figure below, it is illustrated a simple single integer attribute function.

$$\begin{bmatrix} 1 & 3 & \infty & 6 \\ 4 & 4 & 1 & 3 \\ \infty & 8 & \infty & 2 \\ \infty & \infty & \infty & \infty \end{bmatrix}$$

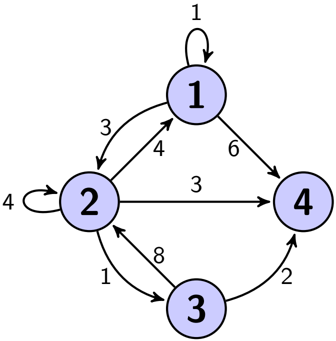


Useful for dense graphs or when speed is needed to check the existence of an arc

Graph basics

Adjacency list or list of incident edges: alternative to incidence matrix, effective for sparse graphs.

$$\begin{bmatrix} 1: & (1,1) & (2,3) & (4,6) & \\ 2: & (1,4) & (2,4) & (3,1) & (4,3) \\ 3: & (2,8) & (4,2) & & \\ 4: & & & & \end{bmatrix}$$



See Ex. 22.1-8 for suggestions of variations for faster edge lookup.

Graph basics

Notation:

- The arc (i, j) exits from i and enters in j
- The edge $[i, j]$ is incident in i and j
- Considering the arc (i, j), i is predecessor of j and j is successor of i
- Considering the edge $[i, j]$, i and j are adjacent
- The degree of a node is the number of incident edges
- The in(out) degree is the number of entering (exiting) arcs
- The neighborhood $N(v)$ of $v \in N$ is the set of nodes adjacent to v (similar for Digraphs)
- A star $\delta(v)$ is the set of edges incident in v. We also define entering star and existing star $\delta^-(v), \delta^+(v)$ for digraphs

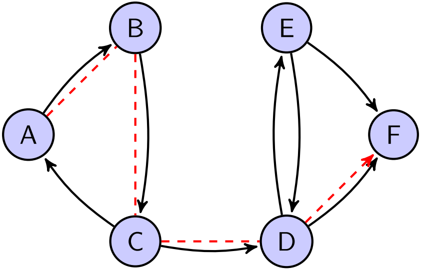
Graph basics

Notation:

- A path is an ordered sequence of consecutive arcs or edges.
- A path that starts and ends in the same node is a cycle.
- A graph is said **connected** if there exist a path between any pair of nodes.
- A Digraph is said **strongly connected** if there esist a path from any node to any other node.

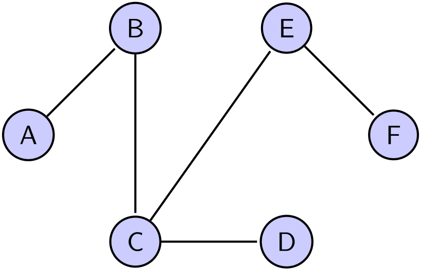
Graph basics

Path (A) – (B) – (C) – (D) – (F)



Graph basics

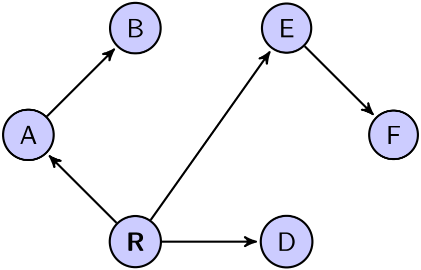
A tree is a connected set of edges without cycles



it is said **spanning** if it touches all nodes of the graph.

Graph basics

In Digraphs, an acyclic set of connected arcs is called arborescence (or out-tree) and a node is defined as a root. It is quite common in IT. It is less common (yet exist) to have in-trees.



Graph basics

Notation:

- A cycle is said **Hamiltonian** if it touches once every node
- A cycle is said **Eulerian** if it traverses once every edge
- A **cut** is a set of edges that, if removed, disconnects the graph. $\delta_G(S) = \{[i,j] \in E : i \in S, j \in N \setminus S, S \subset N, S \neq \emptyset\}$
- A **matching** is a set of edges that are pairwise non adjacent
- A graph is **bi-partite** if it is possible to partition the set of nodes in N_1 and N_2 such that all edges are adjacent to a node in N_1 and N_2
- A graph is **complete** if it has an edge for each pair of nodes (a **clique** is a complete subgraph).

Online graph tool: http://graphonline.ru/en/

Algorithms on graphs - Depth First Search

Depth First Search (DFS):

- A basic algorithm that functions as a prototype for many others
- Explores the nodes and edges of a graph
- Runs in $O(V + E)$
- Not really useful in its basic version but can be used to compute **connected components**, determine **connectivity**, find **articulation points**
- Recursive algorithm
- Visits the first "feasible" edge randomly

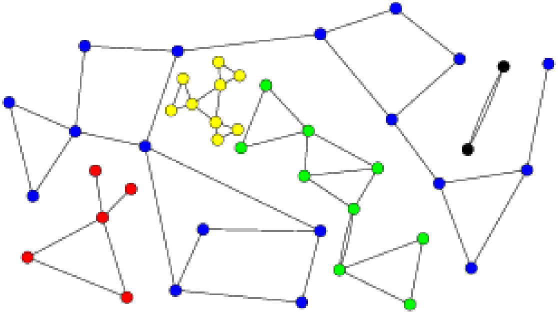
Algorithms on graphs - Depth First Search

```
#Global vars
graph = adjacency list representing the graph
visited = [false, ..., false]

def dfs(node):

    if visited[node]: return
    visited[node] = true

    for adj in graph[node]:
        dfs(adj)

start_node = 0
dfs(start_node)
```

Listing 1: DFS

Algorithms on graphs - Connected components

It is sometimes useful in application to identify and count the number of different connected components in a graph.



Algorithms on graphs - Connected components

```
n = number of nodes
graph = adjacency list representing the graph
visited = [false, ..., false]
count = 0 # number of connected components
components = [-1,...,-1] # stores id of node's component

def findComponents():
    for i in range(n):
        if not visited[i]:
            count = count + 1
            dfs(i, count)
    return (count, components)

def dfs(node, component):
    visited[node] = true
    component[node] = component
    for adj in graph[node]:
        if not visited[adj]:
            dfs(adj)
```

Listing 2: Connected components

Algorithms on graphs - BFS

Breadth First Search (BFS):

- A basic algorithm that functions as a prototype for many others
- Explores the nodes and edges of a graph
- Runs in $O(V + E)$
- Visits the nodes close to the start first

Algorithms on graphs - BFS

```
graph = adjacency list representing the graph
visited = [false, ..., false]

def bfs(node):
  queue = []
  queue.append(node)
  visited[node] = true

  while queue:
    s = queue.pop()
    for adj in graph[node]:
      if visited[i] == False:
        queue.append(adj)
        visited[adj] = True

start_node = 0
bfs(start_node)
```

Listing 3: BFS

Algorithms on graphs - Minimum spanning tree

Given $G = (N, E)$ and a cost function $w : E \mapsto \mathbb{Z}$ determine the spanning tree of minimum cost.

$$x_e = \begin{cases} 1 & \text{edge } e \text{ belongs to the tree} \\ 0 & \text{otherwise} \end{cases}$$

$$\min z = \sum_{e \in E} w(e) x_e$$
$$s.t.\ x_e \ \textit{is a spanning tree}$$

In order to guarantee that a set of edges is a spanning tree, $\sum_{e \in E} x_e = |N| - 1$ and there are no cycles.
MST is used to compute connectivity costs between elements of a set

Minimum spanning tree

Kruskal algorithm computes the optimal value for a MST.

Algorithm 1 Kruskal

{Init}
$L \leftarrow$: list of edges ordered by non-decreasing cost
$i \leftarrow 0$, $T \leftarrow \emptyset$, Empty tree
repeat
    {Choose edge from L}
    repeat
        $i \leftarrow i + 1$
    until $Acyclic(T \cup \{L(i)\})$
    $T = T \cup \{L(i)\}$
until $|T| = |N| - 1$

Book's Kruskal pseudocode

```
KRUSKAL(G, w)
  A = ∅
  for each vertex v ∈ G.V
      MAKE-SET(v)
  sort the edges of G.E into nondecreasing order by weight w
  for each (u, v) taken from the sorted list
      if FIND-SET(u) ≠ FIND-SET(v)
          A = A ∪ {(u, v)}
          UNION(u, v)
  return A
```

Kruskal algorithm

- Polynomial, order of $O(E \log V)$ (depends on the implementation of the disjoint-set, see chapter 21)
- It is spanning ($N-1$ edges are chosen) and acyclic
- It is a greedy algorithm

Minimum spanning tree

Prim algorithm computes the optimal value for a MST.

Algorithm 2 Prim

$\{\text{Init}\}$
$X \leftarrow \{1\},\ Y \leftarrow \{2, 3, \cdots, |N|\},\ T \leftarrow \emptyset$
repeat
  $\{\text{Choose smallest edge connecting } X \text{ and } Y\}$
  $\bar{e} = [i, j] : w(\bar{e}) = \min\{w(e), e = [i, j], i \in X, j \in Y\}$
  $\{\text{Update}\}$
  $X \leftarrow X \cup \{\bar{j}\},\ Y \leftarrow Y \setminus \{\bar{j}\},\ T \leftarrow T \cup \{\bar{e}\}$
until $Y = \emptyset$

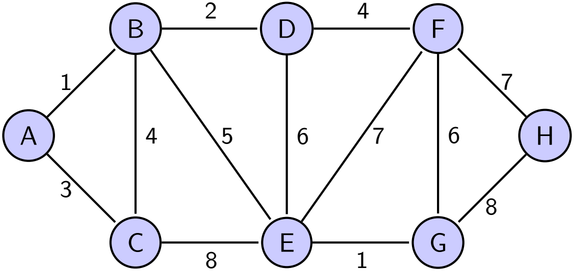
Book's Prim pseudocode

```
PRIM(G, w, r)
  Q = ∅
  for each u ∈ G.V
      u.key = ∞
      u.π = NIL
      INSERT(Q, u)
  DECREASE-KEY(Q, r, 0)        // r.key = 0
  while Q ≠ ∅
      u = EXTRACT-MIN(Q)
      for each v ∈ G.Adj[u]
          if v ∈ Q and w(u, v) < v.key
              v.π = u
              DECREASE-KEY(Q, v, w(u, v))
```

Prim algorithm

- Polynomial, order of $O(E \log V)$ (depends on implementation of priority queue)
- It is spanning (Y is empty at the end) and acyclic (the chosen edge never connects two elements in X)
- It is a greedy algorithm

Application of the algorithm



Algorithms on graphs - Shortest path

Given $G = (N, A)$ and a cost function $w : A \mapsto \mathbb{Z}^+$, a source node s and a destination node t determine the path from s to t of minimum cost.

$$x_{i,j} = \begin{cases} 1 & \text{node } j \text{ is visited just after node } i \\ 0 & \text{otherwise} \end{cases}$$

$$\min z = \sum_{i \in N} \sum_{j \in N} w(i,j) x_{ij}$$

$$s.t. \sum_{j \in N} x_{ji} - \sum_{k \in N} x_{ik} = \begin{cases} -1 & \text{for } i = s \\ 1 & \text{for } i = t \\ 0 & \forall i \neq s, i \neq t \end{cases}$$

$$x_{i,j} \in \{0, 1\}$$

Shortest path

Dijkstra algorithm

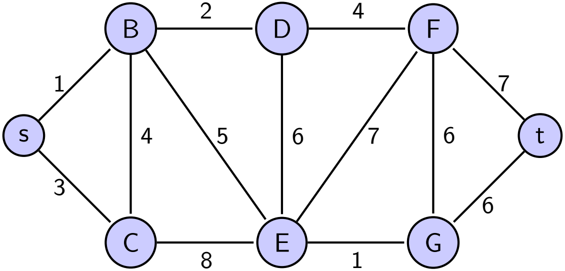
Algorithm 3 Dijkstra

{Init}
$dist[v] := \infty; pred[v] := -1; dist[s] := 0; Q \leftarrow N$
while $Q \neq \emptyset$ **do**
  $u := min_{i \in Q} dist[i];$
  $Q \leftarrow Q \setminus \{u\};$
  for $v \in \delta^+(u) : v \in Q$ **do**
    if $dist[u] + d_{uv} < dist[v]$ **then**
      $dist[v] := dist[u] + d_{uv};$
      $pred[v] := u;$
    end if
  end for
end while

Dijkstra algorithm

- It exploits the fact that if a node i belongs to the shortest path from s to t, then the path from s to i is also the shortest.
- Example of dynamic programming algorithm
- Complexity for simple implementations is $O(n^2)$. Advanced implementations with priority queues reach $O(n \log n)$.

Application of the algorithm



Network flows

We can represent a transportation network (of goods, information, vehicles) using a weighted graph

- Arcs are media in which goods/information/vehicles transit
- Weights on arcs represent capacity or costs
- We can also define node weights representing the enrty or exit of goods in the network

Network flow

More formal definition:

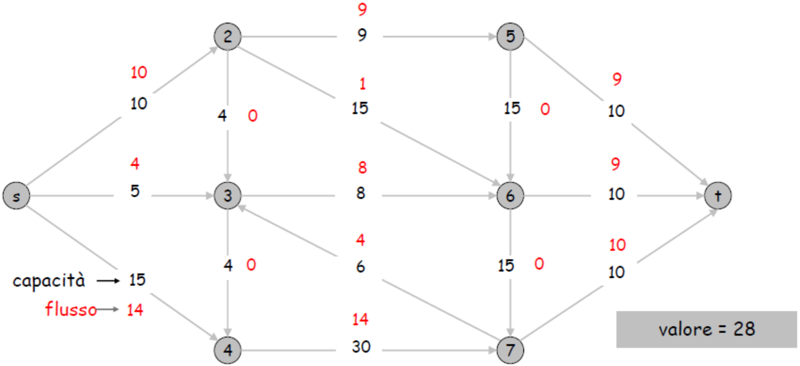
- each node $i \in N$ is associated with a real value b_i:
  - positive: it represents the quantity **exiting** the network and it represents a sort of **demand** of the node. Therefore the node is called **destination** or output node.
  - negative: it represents the quantity **entering** the network and it represents a sort of **offer** of the node. Therefore the node is called **origin** or input node.
  - null, the node is called **transit** or transfer node.
- each arc $a = (i, j)$ is associated with a cost c_a (or c_{ij}), indicating the unitary cost for traversing the arc and a lower capacity l_a (l_{ij}) and upper capacity u_a (u_{ij}), indicating the minimum and maximum amount of units that can transit along the arc.

All network flow problems can be defined on an equivalent network with exactely one origin and one destination (blackboard).

Maximum flow problem

In the maximum flow problem, $c_a = 0$ and $l_a = 0 \; \forall a \in A$. We want to determine the maximum amount of flow that can be shipped from the origin to the destination respecting the capacity on the arcs and the flow conservation in intermediate arcs.

Example


s
2
3
4
5
6
7
t
10
10
4
5
15
14
9
9
1
15
4
0
8
8
4
6
4
0
14
30
15
0
15
0
9
10
9
10
10
10
capacità
flusso
valore = 28

Mathematical model

$$
\begin{aligned}
& \max f \\
& s.t. \sum_{j \in \delta^-(i)} x_{ji} - \sum_{j \in \delta^+(i)} x_{ij} = \begin{cases} -f & i = s \\ 0 & i \neq s, t \\ f & i = t \end{cases} \\
& \quad 0 \leq x_{ij} \leq u_{ij}, \quad \forall (i, j) \in A
\end{aligned}
$$

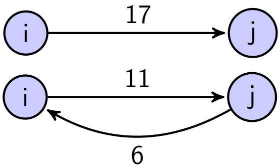
Ford-Fulkerson algorithm

It uses the concept of **residual** or **complementary** network $G'(N', A')$ in which $N' = N$.

Given a flow x, $A' = \{(i,j) : x_{ij} < u_{ij}\} \cup \{(j,i) : x_{ij} > 0\}$
Capacity: $u'_{ij} = u_{ij} - x_{ij}$ per $(i,j) \in A$, $u'_{ij} = x_{ji}$ per $(i,j) \in A$

Arc in the original network, with capacity 17

For a given flow of $x_{ij} = 6$ the residual network has two arcs



Every feasible network flow defines a **different** residual network!

Ford-Fulkerson algorithm

Augmenting path: a path from s to t on the residual network.

If it exists, we can increase the flow on the network along the path by an amount equal to the lowest capacity of arcs belonging to the path.

$$\delta = \min_{(i,j)\in P} u'_{i,j}$$

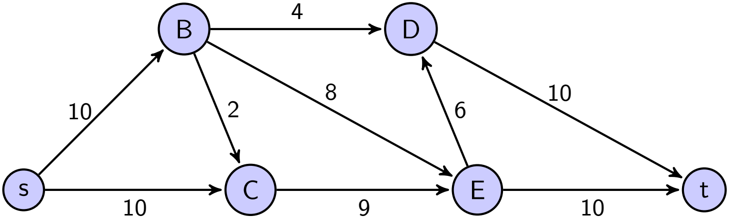
The augmenting path can be computed using a polynomial algorithm!

Ford-Fulkerson algorithm

Algorithm 4 Ford-Fulkerson

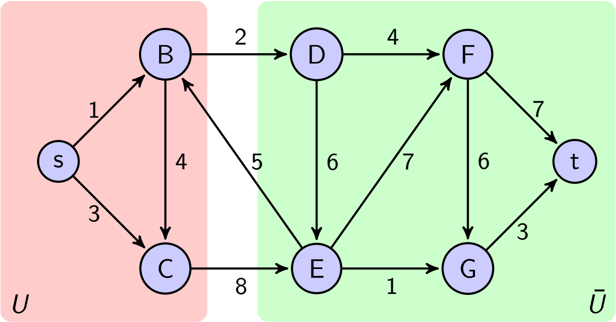
$\{$Init$\}$
x := 0; $\phi := 0$;
repeat
  $\{$Compute residual network $G'(N', A')$ associated with x$\}$
  $\{$Compute path P from s to t on $G'\}$
  if P does not exist **then**
    Optimal flow!
  else
    $\delta := \min\{u'_{ij} : (i,j) \in P\}$, $\phi := \phi + \delta$
    for $(i,j) \in P$ **do**
      $x_{ij} := x_{ij} + \delta$ per $(i,j) \in A$
      $x_{ij} := x_{ij} - \delta$ per $(j,i) \in A$
    end for
  end if
until *Ottimo*

Example



Cut of a network

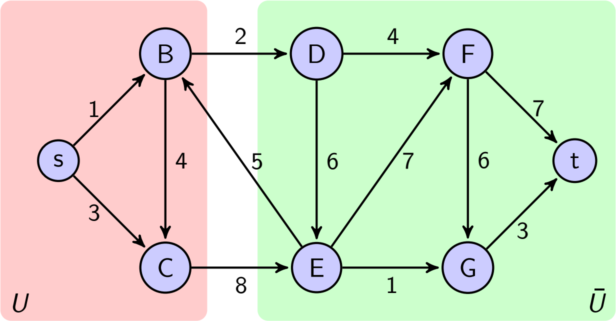
Given a digraph $G = (N, A)$, with $s, t \in N$ e $s \neq t$ we define an *s-t-cut* a partition of the nodes $C(U, \bar{U})_{s,t}$ such that $s \in U$ e $t \in \bar{U}$



Cut of a network

We define **capacity** of a cut $C(U, \bar{U})_{s,t}$

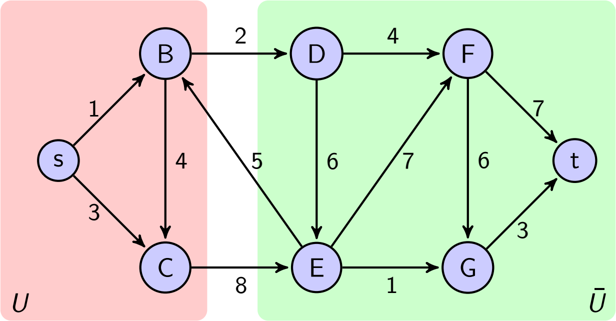
$$q\left(U, \bar{U}\right) = \sum_{i \in U, j \in \bar{U}} u_{ij}$$



In figure the cut has capacity 10 (arc E $\rightarrow$ B does not count).

Max flow - min cut theorem

In a network the maximum s-t-flow cannot be larger than the capacity of any s-t-cut.



$$\phi(U) = \sum_{(i,j)\in\delta^+(U)} x_{ij} - \sum_{(i,j)\in\delta^-(U)} x_{ij} \leq \sum_{(i,j)\in\delta^+(U)} u_{ij} = q(U, \bar{U})$$

In particular

$$\phi^* = q^*(U, \bar{U})$$