Advanced Programming Techniques

Conf. dr. ing. Guillaume Ducoffe

guillaume.ducoffe@fmi.unibuc.ro

About myself. . .

- Associate Professor (*Conferentiar*) at the Univ. of Bucharest, Faculty of Mathematics and Computer Science.

- Research Scientist at the National Institute of Research and Development in Informatics (ICI).

- PhD in Nice-Sophia Antipolis, France.

- Various visits/internships in the US, Chile, Germany,. . .

Research in **Graph Algorithms** *and Data Structures*

Come to see me if...

- **You want to learn more about (algorithmic aspects of) the class!**
- **You need help for your Bachelor/Master thesis**
  - Plenty of my problems need good students to find good solutions, and/or implement some proposed solutions.
- **You want to discuss about internship/PhD/job opportunities...**

Practical Information

- All materials (slides + documents) to be put on Moodle.

- My website – with contact information

  https://sites.google.com/view/guillaume-ducoffes-homepage/home

  Additional documents can be put on it – upon request

- **Attendance is NOT mandatory**. But...
  - Bonus for seminars works (max. 1p)

Your **grades**

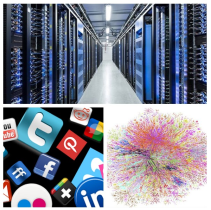
- One practical test at the end of the semester.
  - Programming languages: either C++ or Java.
  - 4 to 5 sets of subjects proposed (students have to pick only one).

Final grade: $\min\{10, \text{Practical Test} + \text{Bonus}\}$

There shall be NO “punct din oficiu”.

About the class

Growing size of networks



Social networks (Facebook $\geq$ 1.79 billion users)

Data Centers (Microsoft $\geq$ 1 million servers)

the Internet ($\geq$ 55811 Autonomous Systems)

"Efficient" algorithms on these graphs?

~~polynomial~~ $\rightarrow$ quasi-linear time

~~quadratic~~ $\rightarrow$ (sub)linear space

~~sequential~~ $\rightarrow$ **parallel**+distributed

Focus on key procedures in graph & relational databases, and beyond.

Tentative schedule for this semester

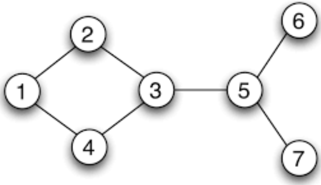
- Graph basics (**Today**).
- Parallel search
- Parallel connected components & spanning forests
- Centrality indices
- Pattern detection and couting
- Labelling schemes (Hub labels, etc.)
- Spanners & emulators
- Graph sketches
- Map reduce Graph algorithms
- Acyclic Database schemes

Graphs

- **(Undirected) Graph**: $G = (V, E)$

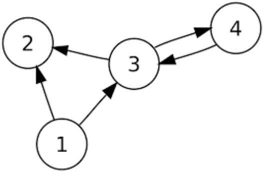
V = vertices = networks units

E = edges



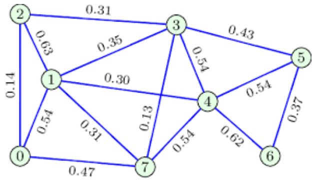
- **Directed Graph**: $\overrightarrow{G} = (V, A)$

A = arcs



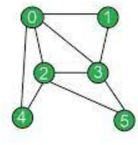
- **Weighted Graph**: $G = (V, E, w)$

$w : E \to \mathbb{R}$ is a weight function (length, cost)



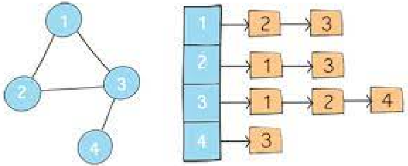
Graph representations

- Adjacency matrix



	0	1	2	3	4	5
0	0	1	1	1	1	0
1	1	0	0	1	0	0
2	1	0	0	1	1	1
3	1	1	1	0	0	1
4	1	0	1	0	0	0
5	0	0	1	1	0	0

- **Adjacency list**



undirected graph and its adjacency list

Basic operations on graphs

- Enumerate all edges
- Enumerate all neighbours of a vertex
- Output the degree of a vertex
- Decide whether two vertices are adjacent
- Addition/Removal of edges/vertices.
- **Contraction** of an edge $e = uv$: replace u, v by some new vertex x whose neighbourhood equals $N(u) \cup N(v) \setminus \{u, v\}$.

With Adjacency List

- Enumerate all edges: scan the whole structure in $\mathcal{O}(n+m)$ time.

Better: Keep the lists of isolated vertices at the end, to avoid scanning them. $\rightarrow \mathcal{O}(m)$ time.

With Adjacency List

- Enumerate all edges: scan the whole structure in $\mathcal{O}(n+m)$ time.

  Better: Keep the lists of isolated vertices at the end, to avoid scanning them. $\rightarrow \mathcal{O}(m)$ time.

- Enumerate all neighbours of a vertex v: scan the corresponding list of the matrix in $\mathcal{O}(d(v))$ time.

With Adjacency List

- Enumerate all edges: scan the whole structure in $\mathcal{O}(n+m)$ time.

  Better: Keep the lists of isolated vertices at the end, to avoid scanning them. $\rightarrow \mathcal{O}(m)$ time.

- Enumerate all neighbours of a vertex v: scan the corresponding list of the matrix in $\mathcal{O}(d(v))$ time.

- Output the degree of a vertex: we output the list size in $\mathcal{O}(1)$ time.

With Adjacency List

- Enumerate all edges: scan the whole structure in $\mathcal{O}(n+m)$ time.

  Better: Keep the lists of isolated vertices at the end, to avoid scanning them. $\rightarrow \mathcal{O}(m)$ time.

- Enumerate all neighbours of a vertex v: scan the corresponding list of the matrix in $\mathcal{O}(d(v))$ time.

- Output the degree of a vertex: we output the list size in $\mathcal{O}(1)$ time.

- Addition of a new vertex: insertion of a new empty list at the end in amortized $\mathcal{O}(1)$ time (using a doubling array to store adjacency list).

With Adjacency List

- Enumerate all edges: scan the whole structure in $\mathcal{O}(n+m)$ time.

  Better: Keep the lists of isolated vertices at the end, to avoid scanning them. $\rightarrow \mathcal{O}(m)$ time.

- Enumerate all neighbours of a vertex v: scan the corresponding list of the matrix in $\mathcal{O}(d(v))$ time.

- Output the degree of a vertex: we output the list size in $\mathcal{O}(1)$ time.

- Addition of a new vertex: insertion of a new empty list at the end in amortized $\mathcal{O}(1)$ time (using a doubling array to store adjacency list).

- Removal of an isolated vertex v: Switch v with the last vertex, then remove the last vertex, in amortized $\mathcal{O}(1)$ time (using a doubling array to store adjacency list).

Reminder: Hash Table

Store a collection of pairs `(key,value)`.

- Three operations:
  - **int** `lookup(e);` Returns the value associated to some key e (if it is present in the table)
  - **void** `insert(e, v);` Adds a new pair `(e,v)` (if the key e is not already present in the table)
  - **void** `delete(e);` Deletes a pair `(key,value)` – given its key e.

Each operation runs in expected $\mathcal{O}(1)$ time.

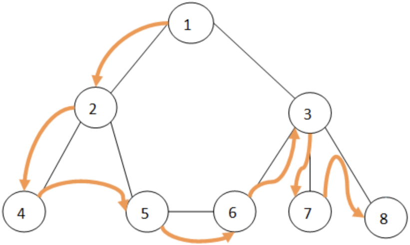
Adjacency Testing

We store every edge in a Hash table. To every edge uv, we associate pointers its positions in the respective adjacency lists of u and v.

- Adjacency testing: Lookup in the table in expected $\mathcal{O}(1)$ time.
- Addition of an edge uv: Insertion of u (v, resp.) at the bottom of the adjacency list of v (u, resp.). Insertion in expected $\mathcal{O}(1)$ time in the Hash table, along with pointers to the bottom positions in the respective adjacency lists of u and v.
- Removal of an edge uv: Lookup in the Hash-table in order to find the positions of u, v in the respective adjacency lists of v, u. Then, removal in both adjacency lists and in the table.
- Removal of a vertex: removal of every incident edge + removal of an isolated vertex.

Serial DFS

Pick the **most recently visited** vertex with at least one neighbour unvisited. Then, go to an arbitrary unvisited neighbour of this vertex.



Equivalently: either continue the search to any neighbour of the current vertex (if possible) or backtrack to the father node in the search tree.

Implementation

At any moment during the execution of the algorithm, we **keep in a stack** the path from the start vertex v_{n-1} to the current vertex v_i.

S := $\{\}$

S.push(v_{n-1})

Visit v_{n-1}

while !S.empty():

  u := S.top() *//current vertex*

  if there exists some $v \in N(u)$ unvisited:

    S.push(v)

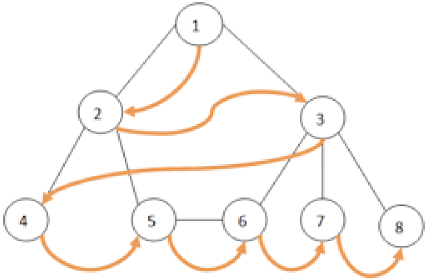
    Visit v

  else: S.pop()

Complexity: $\mathcal{O}(n+m)$

Serial BFS

Pick the **least recently visited** vertex with at least one neighbour unvisited. Then, go to an arbitrary unvisited neighbour of this vertex.



Implementation

We **keep in a queue** the next vertices to be visited, in order.

Q := {}

Q.enqueue(v_{n-1})

visited[v_{n-1}] := True

while !Q.empty():

  u := Q.dequeue() //*current vertex*

  Visit u

  for all $v \in N(u)$:

    if !visited[v]:

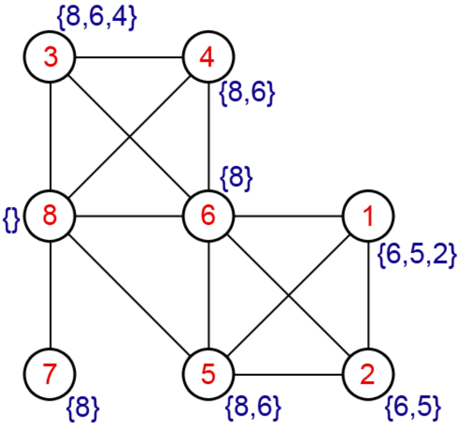
      Q.enqueue(v)

      visited[v] := True

Complexity: $\mathcal{O}(n+m)$

LexBFS

The visited neighbours of each vertex are ordered by decreasing label. At every step, the next vertex to be visited must have a label which is **lexicographically** maximum.



Partition Refinement

• Data Structure that maintains an ordered collection of pairwise disjoint sets, subject to the following basic operations:

- `init(V)`: initialize the structure with one set, equal to V.
- `refine(S)`: for each set X such that $X \cap S \neq \emptyset$ and $X \setminus S \neq \emptyset$, we replace X by the two consecutive new sets $X \cap S$ and $X \setminus S$.

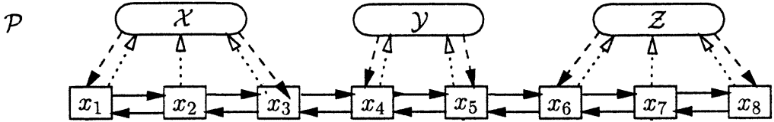
• Operation `init(V)` is in worst-case $\mathcal{O}(|V|)$. Each operation `refine(S)` is in worst-case $\mathcal{O}(|S|)$. Note that these are optimal runtimes!

Implementation

- Elements in V are maintained in a doubly-linked list $\mathcal{L}$, such that all elements in a same set X are consecutive.

- Each set X of the partition is represented by a structure with two fields: pointers to its first and last elements in $\mathcal{L}$.

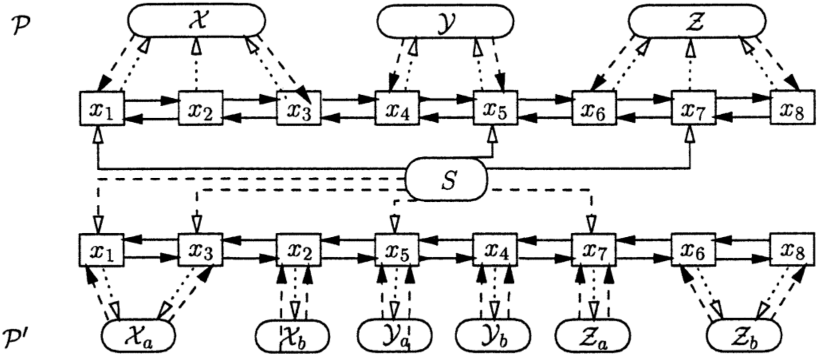
- Each node in the list $\mathcal{L}$ stores a pointer to the set X which contains it.



Refinement

- To each set X, "lazily" associate an empty list $L[X]$ (we effectively create the list only if it needs to be accessed at some point during the execution of the algorithm).
- For each $s \in S$, access to its set X and add a pointer to s in $L[X]$. Put a pointer to X in an auxiliary list $\mathcal{H}$ (the sets of $\mathcal{H}$ are those intersecting S).
- For each set X in $\mathcal{H}$, if $L[X] \neq X$, then:
  - Update the first and last element of X as its first and last elements not in S (forward/backward search in $\mathcal{L}$).
  - Remove all elements in $L[X]$ from $\mathcal{L}$;
  - Reinsert $L[X]$ immediately before the first element of X (or immediately after the last element of X);
  - Create a new set from $L[X]$.

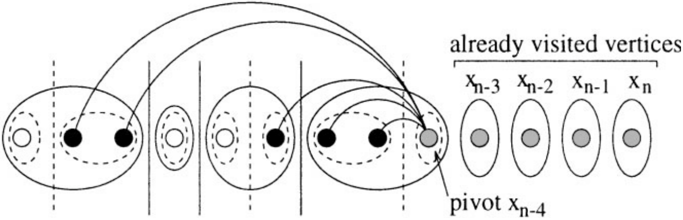
Refinement: illustration



Application to LexBFS

Traverse backward the list of all vertices – with the start vertex v_{n-1} at the end.

Repeatedly visit the next vertex v_i and refine the unvisited vertices according to $N(v_i)$.



Complexity: $\mathcal{O}(n+m)$

Weighted graphs

If all edge-weights are positive, we can use Dijkstra's algorithm as a replacement for BFS:

`H := empty heap`

`Insert every` $v \in V$ `in H with infinite value.`

`H[`v_{n-1}`] := 0` //*start vertex*

while `H is nonempty:`

    (v_i, d_i) `:= H.extract_min()`

    for all $u \in N(v_i)$ `such that` u `is in the heap:`

    if d_i `+` $w(v_i, u)$ `<` `H[`u`]:` `H.decrease_key(u,`d_i `+` $w(v_i, u)$`)`

We need to perform on the heap: n insertions, n deletions, and $\mathcal{O}(m)$ decrease key operations.

$\rightarrow \mathcal{O}(n \log n + m)$ time by using Fibonacci heaps.

Limitations of Dijkstra's algorithm

• Only applies to **positive** edge-weights (null-weights can be easily handled, but negative weights are a real issue).

• In some applications, we only need to compute a shortest path between a source vertex v_{n-1} and *any* vertex of some subset $P \subseteq V$. By contrast, Dijkstra's algorithm computes *all* distances from v_{n-1}.

• In real-life applications, the graph may be so large that we cannot even traverse it in full. It may even be infinite!

$\Longrightarrow$ An alternative search, proposed at the infant stages of AI: the A^* **heuristic** (Hart, Nilsson & Raphael, 1986).

A^* inputs

- A directed + arc-weighted graph $G = (V, A, w)$.

$\rightarrow$ Weights may be negative. However, we assume that there is no negative circuit (Assumption A1).

- A start vertex s + a destination subset $P \subseteq V$

$\rightarrow$ Since G may be infinite, we need to make the minimal assumption that there exists a path from s to P (Assumption A2)

- A **heuristic function** $h : V \rightarrow \mathbb{R}$, which must represent a lower bound estimate on the distance from a vertex to P, *i.e.*, $d(x, P) \geq h(x)$.

$\rightarrow$ *e.g.*, distances "as the crow flies" for road/street networks. If all weights are nonnegative, then we may simply set $h(x) = 0$.

A^* output

Compute $d(s, P)$, and a shortest path from s to a closest vertex in P.

- The A^* heuristic will stop as soon as it reaches a vertex in P.

- For this halting condition to make sense, we need to make the minimal assumption that $d(p, p') \geq 0$ for every two vertices $p, p' \in P$ (Assumption $A3$).

Initialization

The presentation mostly follows that proposed by Habib and Simonet in their research report (1991).

- We maintain two subsets: `Closed` (already visited vertices), and `Open` (vertices to be visited)).

$\rightarrow$ Initially, $\texttt{Closed} = \emptyset$ and $\texttt{Open} = \{s\}$.

As we shall see, we may reopen closed vertices if a better path from s has been found. This is different from Dijkstra's algorithm, where a visited vertex will never need to be visited again.

- We maintain an in-arborescence ("shortest-path tree") with root s. The predecessor of each vertex $x \neq s$ is stored in a variable $\texttt{pred}(x)$. The best-known distance from s to x is stored in variable $g(x)$.

$\rightarrow$ We need $g(x)$ to be defined only if x is in `Open`. Initially, $g(s) = 0$.

Main loop

The following procedure runs while $\texttt{Open}$ is nonempty:

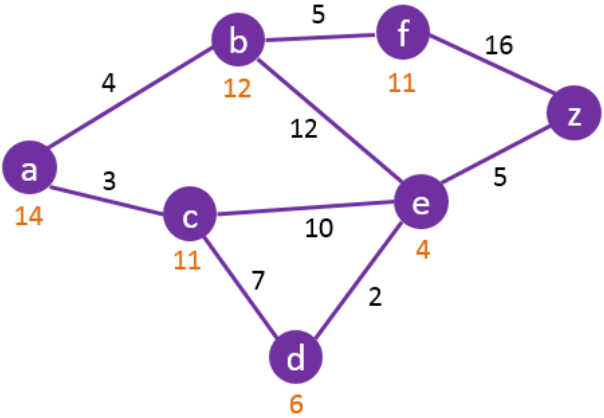
1) We compute a vertex $x \in \texttt{Open}$ such that $f(x) := g(x) + h(x)$ is minimized (estimated distance from s to P going by x).

2) We remove x from $\texttt{Open}$, then we insert x in $\texttt{Closed}$.

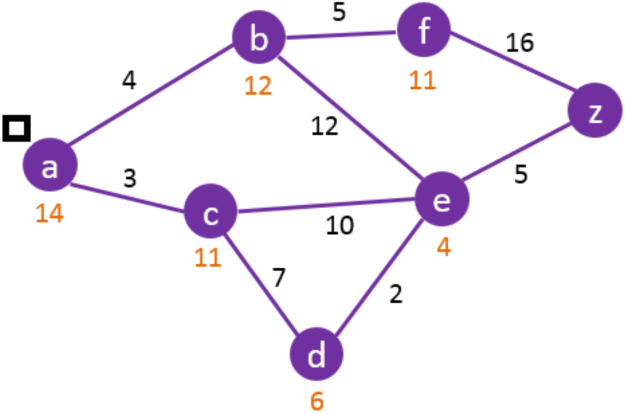
3) If $x \in P$, then we stop. Else, we do as follows for every $y \in N^+(x)$ (out-neighbours):

- If $y \in \texttt{Open}$ and $g(y) > g(x) + w(x, y)$, then: $g(y) := g(x) + w(x, y)$, $pred(y) := x$.
- If $y \in \texttt{Closed}$ and $g(y) > g(x) + w(x, y)$, then: $g(y) := g(x) + w(x, y)$, $pred(y) := x$. Furthermore, we remove y from $\texttt{Closed}$, then we insert y in $\texttt{Open}$.
- If $y \notin \texttt{Open} \cup \texttt{Closed}$, then: $g(y) := g(x) + w(x, y)$, $pred(y) := x$. Furthermore, we insert y in $\texttt{Open}$.

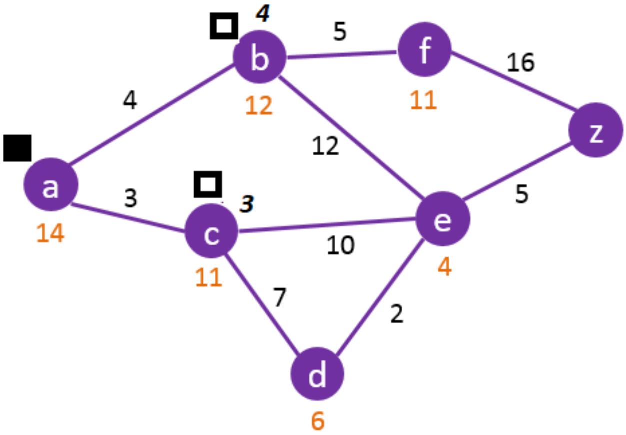
Example



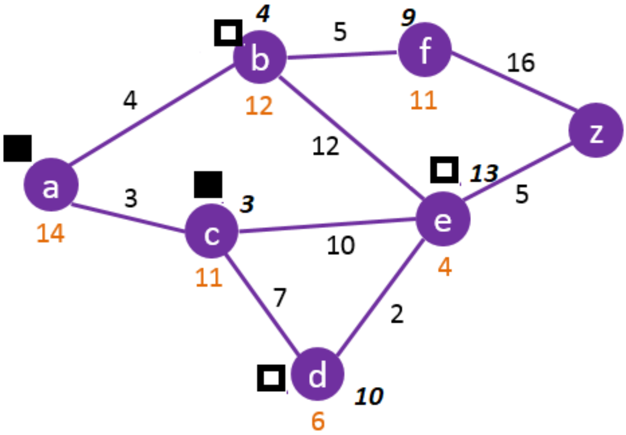
Example



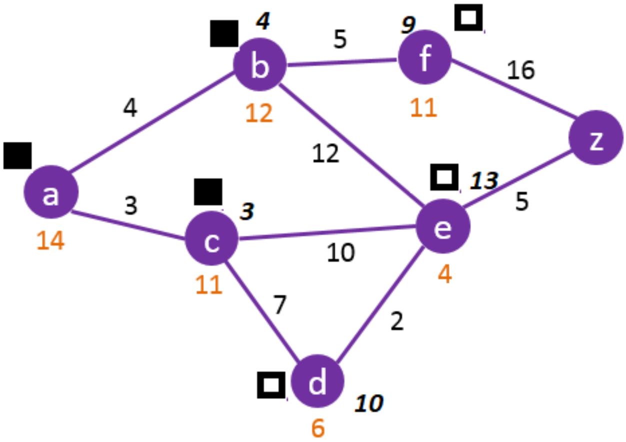
Example



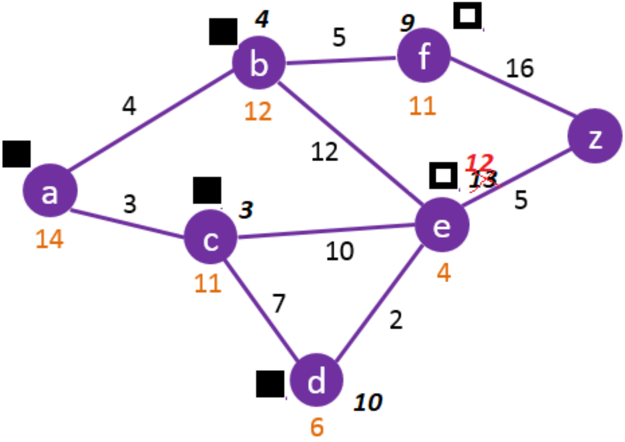
Example



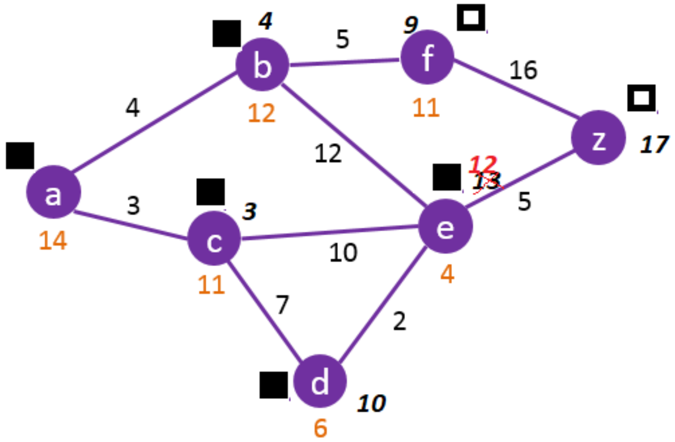
Example



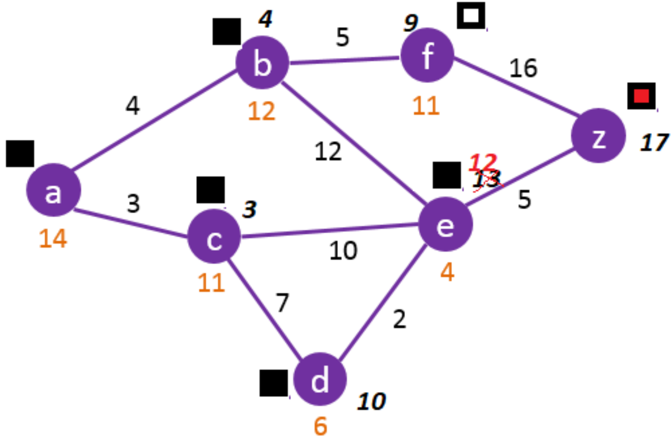
Example



Example



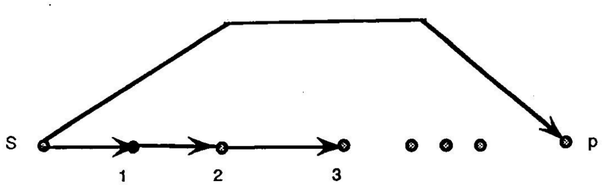
Example



No termination

Set $w(s,p) = 2$, $w(s,1) = 1$, $w(i,i+1) = 1/i(i+1)$ for every positive integer i.

Set $h(i) = 1/i$ for every positive integer i.



After i steps: $g(j) = 1 - 1/j$ for every $1 \leq j \leq i$.

$\rightarrow s, 1, 2, \ldots, i-1$ are closed, i is open and $f(i) = 1$.

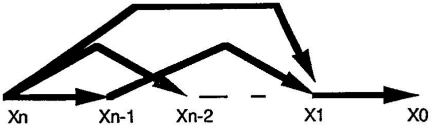
Consequence: we stay blocked on an infinite path!

The case of finite graphs

Proposition: If G is finite, then A^* always halts.

Proof: every time a vertex x is reopen, we found a better path from s to x. There are only a finite (but exponential in n) number of sx-paths.

- Unfortunately, the runtime may be exponential (Martelli):



- In the special case where all weights are nonnegative, and there is no estimate (*i.e.*, $h(x) = 0$), then we retrieve Dijkstra's algorithm. However, the larger h, the less vertices (without counting repetitions) are visited.

The B heuristic

- First proposed by Martelli as a variation of A^* for graphs with nonnegative arc weights.

<u>Intuition</u>: Recall that $f(x) = g(x) + h(x)$ is a lower bound estimate on the shortest path from s to P that goes by x. Let F be the maximum value $f(x)$ amongst all the *closed* vertices.

- If some open vertices x' satisfy $f(x') < F$, then amongst those we pick one minimizing $g(x')$ (i.e., we ignore h, or equivalently we perform Dijkstra's algorithm).
- Otherwise, we pick any open vertex x' that minimizes $f(x')$ (the same as for A^*).

Proposition: the number of iterations is at most $\mathcal{O}(n^2)$. Furthermore, it is never worse than for A^*.

Questions