Operations Research (Master's Degree Course)

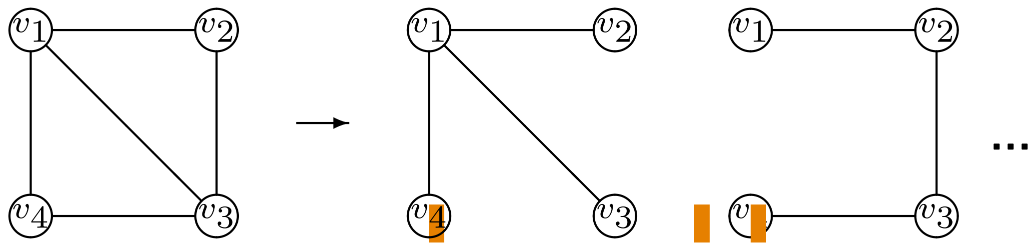
7.2 Problems on Graphs: Basic Problems

Silvano Martello

DEI "Guglielmo Marconi", Università di Bologna, Italy

Shortest Spanning Trees in undirected graphs

- **Tree** = graph with n vertices which
  - is connected, and does not contain circuits; *or, equivalently,*
  - is connected, and contains $n-1$ edges; *or, equivalently,*
  - $\forall v_i, v_j\ \exists$ exactly one path from v_i to v_j.
- Given $G = (V, E)$, a subgraph $G' = (V, E')$ $(E' \subseteq E)$ that forms a tree is called a **Spanning Tree (ST)** of G.



- **Shortest Spanning Tree (SST)** problem: given $G = (V, E)$ weighted and connected, find a ST $G' = (V, E')$ such that $\sum_{e \in E'} w(e)$ is a minimum.

  (We assume by simplicity that $w(e) \geq 0\ \forall\ e \in E$.)

  Applications:
  - electric circuits (Kirchhoff's laws);
  - connect towns through (water, gas, ...) pipelines at minimum cost;
  - subproblem to be solved for solving more complex problems.

Shortest Spanning Trees in undirected graphs (cont'd)

- **Theorem** (Prim, 1957) *Given* $G = (V, E)$, *and a* partial tree *(subgraph forming a tree)* (W, E') *with* $W \subset V, E' \subset E$, *let* $(\overline{u}, \overline{v})$ *be the shortest edge among the edges* $(u, v) : u \in W, v \in V \setminus W$. *Then among all spanning trees of* G *that contain* E' *there exists an optimum one that also contains* $(\overline{u}, \overline{v})$.

  Proof let SST^* be the shortest ST containing E', and suppose by absurd that it does not contain $(\overline{u}, \overline{v})$. SST^* must have a path between $\overline{u}$ ed $\overline{v}$. Such path must contain an edge (u, v) with $u \in W$ and $v \in V \setminus W$. By removing (u, v) we obtain two separate trees. By adding $(\overline{u}, \overline{v})$ we obtain an ST that is shorter than SST^* (contradiction). $\square$

- It follows that if (W, E') is optimum then $(W \cup \{\overline{v}\}, E' \cup \{(\overline{u}, \overline{v})\})$ is optimum.

- Immediate simple polynomial-time algorithm:

begin

  $W := \{v_1\}$, $E' := \emptyset$ (**comment:** empty partial tree, hence optimum);

  while $|W| < n$ **do**

    begin

      find $(\overline{u}, \overline{v})$ as defined by Prim's Theorem;

      $W := W \cup \{\overline{v}\}, \ E' := E' \cup \{(\overline{u}, \overline{v})\}$

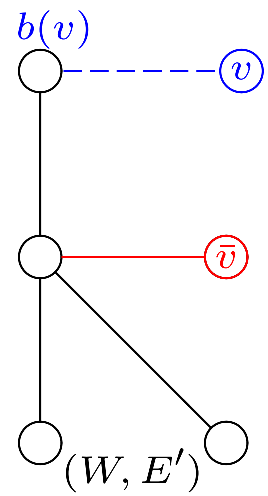
    end

end.

Shortest Spanning Trees in undirected graphs (cont'd)

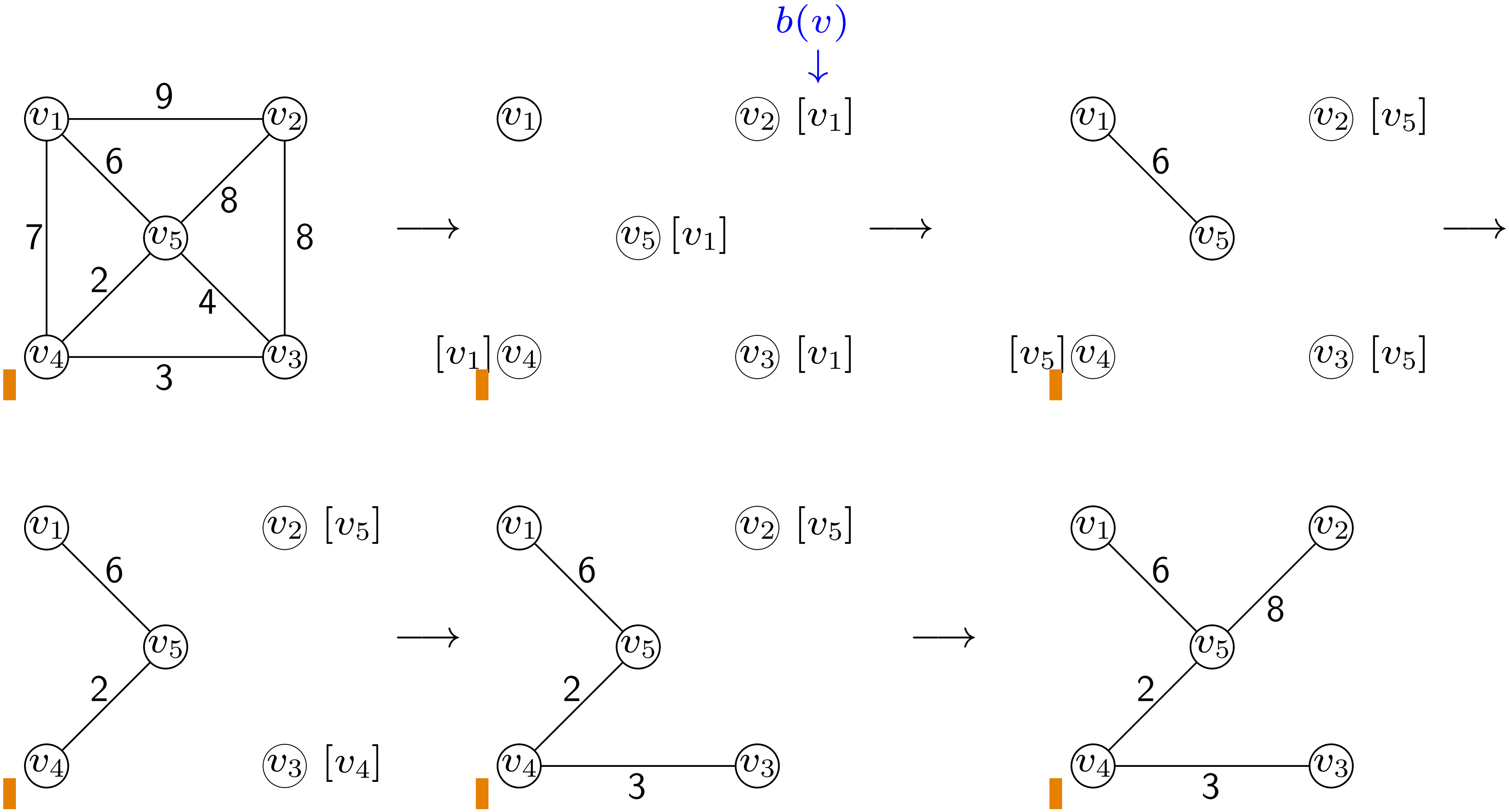
- **Computational effort** required by the simple algorithm, as a function of the graph size:
  - $n-1$ iterations of the **while** loop;
  - at each iteration, number of operations proportional to $|E|$, i.e., to n^2.
  - overall time proportional to n^3: **the algorithm takes $O(n^3)$ time.**
- A better implementation (**Prim's algorithm**):

procedure Shortest_Spanning_Tree:
begin
  $W := \{v_1\}$; $E' := \emptyset$;
  comment: $b(v) =$ vertex $\in W : w(v, b(v)) = \min_{r \in W}\{w(v,r)\}$;
  for each $v \in V \setminus \{v_1\}$ **do** $b(v) := v_1$;
  while $W \neq V$ **do**
    begin
      find $\overline{v} \in V \setminus W \; : w(\overline{v}, b(\overline{v})) = \min_{v \in V \setminus W}\{w(v, b(v))\}$;
      $W := W \cup \{\overline{v}\}$; $E' := E' \cup \{(\overline{v}, b(\overline{v}))\}$;
      for each $v \in V \setminus W$ **do if** $w(v, \overline{v}) < w(v, b(v))$ **then** $b(v) := \overline{v}$
    end
end.



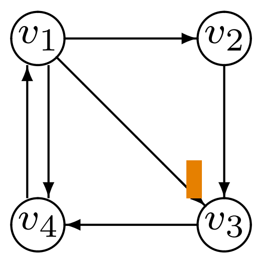
- $n-1$ iterations of the **while** loop; at each iteration, number of operations proportional to $|V \setminus W|$, i.e., to n. Overall time proportional to n^2: **the algorithm takes $O(n^2)$ time.**

Example:



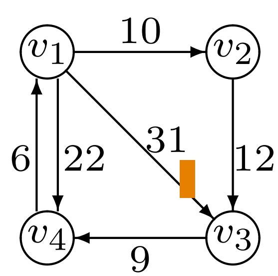
Data structures for representing graphs and networks

- Also useful for other kinds of data.
- Different data structures if the graph has “many” or “few” edges/arcs.
- **Dense graphs** ($m \approx n^2$):
  - unweighted: **Adjacency matrix** $[a_{ij}]$ $(n \times n)$: $a_{ij} = \begin{cases} 1 & \text{if } (v_i, v_j) \in A \;\; [\in E], \\ 0 & \text{otherwise;} \end{cases}$



$$[a_{ij}] = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

  - weighted: **Weight matrix** $[w_{ij}]$ $(n \times n)$: $w_{ij} = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in A \;\; [\in E], \\ \infty & \text{otherwise.} \end{cases}$



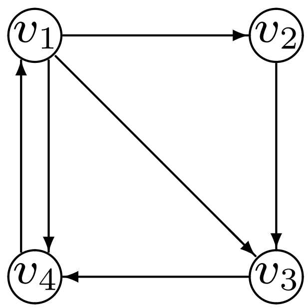
$$[w_{ij}] = \begin{bmatrix} \infty & 10 & 31 & 22 \\ \infty & \infty & 12 & \infty \\ \infty & \infty & \infty & 9 \\ 6 & \infty & \infty & \infty \end{bmatrix}$$

  - $w_{ij} = +\infty / -\infty$ for non-existing edges/arcs in minimization/maximization problems;
  - similarly for capacities;
  - the matrices are symmetric for undirected graphs

    $\Rightarrow$ possible use of additional data structures to save half of the space.

Data structures for representing graphs and networks (cont'd)

- **Sparse graphs** ($m \ll n^2$):
  - unweighted: **forward star**: 2 vectors to only store existing arcs:
    - · vector $p(n+1)$ of pointers ($p_1 = 1$, $p_{n+1} = m+1$),
    - · vector $u(m)$: $(u_{p_i}, \dots, u_{p_{i+1}-1})$ = indices of those vertices v : $\exists$ arc (v_i, v).



$$\begin{array}{lclllllll} p' & = & (& 1, & 4, & 5, & 6, & 7) & \\ u' & = & (& 2, & 3, & 4,| & 3,| & 4,| & 1|) \end{array}$$

  - To also easily access entering arcs, **backward star**:
    - · vector $q(n+1)$ of pointers ($q_1 = 1$, $q_{n+1} = m+1$),
    - · vector $e(m)$: $(e_{q_i}, \dots, e_{q_{i+1}-1})$ = indices of those vertices v : $\exists$ arc (v, v_i).

$$\begin{array}{lclllllll} q' & = & (& 1, & 2, & 3, & 5, & 7) & \\ e' & = & (& 4,| & 1,| & 1, & 2,| & 1, & 3|) \end{array}$$

  - weighted: forward/backward star "+" vector $w(m)$ (w_k = weight of the arc $\leftrightarrow u_k/e_k$).

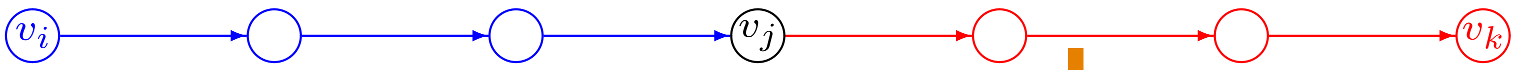
$$\begin{array}{lclllllll} p' & = & (& 1, & 4, & 5, & 6, & 7) & \\ u' & = & (& 2, & 3, & 4, & 3, & 4, & 1) \\ w' & = & (& 10, & 31, & 22, & 12, & 9, & 6) \end{array}$$

Shortest Path Problems (SPP)

- One of the most important families of problems on graphs.
- **Basic problem**: Given a weighted directed graph $G = (V, A)$, and a vertex $s \in V$,

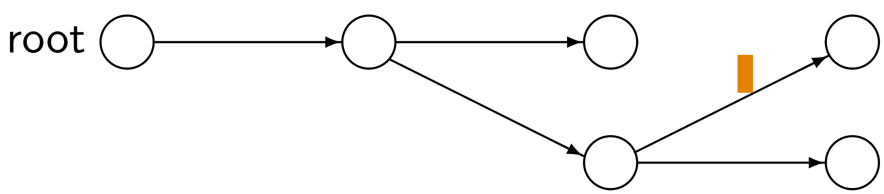
  find, $\forall\ v \in V$, the shortest path from s to v.

- **Property** *If the shortest path from v_i to v_k passes through v_j, then it is given by the shortest path from v_i to v_j concatenated with the shortest path from v_j to v_k.*



  Proof A shorter path from v_i to v_j (or from v_j to v_k) would be used for the shortest path from v_i to v_k. $\square$

- **Arborescence =** directed loopless graph in which:
  - · one vertex (called *root*) has no entering arc;
  - · all other vertices have exactly one entering arc:



- **Property** *The shortest paths emanating from s form an arborescence with root in s.*

  Proof Each vertex but s must have at least an entering arc. If a vertex v had two entering arcs one could eliminate the one belonging to the longest path from s to v. $\square$

Shortest Path Problems (cont'd)

- **A relevant consideration on the costs of arc/edges of graph**
- Usually the arc/edge costs are non-negative (e.g., road lengths, traveling costs, ...).
- In some applications we can have arcs/edges with negative cost. For example, the graph can represent economic transaction, in which a transaction can produce a gain or a loss.
- One of the basic algorithms of graph theory (Dijkstra, 1957) assumes that **the costs are non-negative**:
- **Basic structure:**
  - the algorithm starts with the source s (current partial arborescence A),
  - and builds a complete arborescence by adding at each iteration a vertex and an arc to A.
  - The vertices are added to A by increasing distance from s, i.e.,:
    - first s;
    - then the vertex closest to s (shortest arc emanating from s);
    - then the next closest vertex, and so on.
- The structure of the algorithm is similar to that of the Prim algorithm for the SST;
- we will first describe the algorithm, and then prove its correctness.

Dijkstra's algorithm (Non-negative distance matrix)

procedure **SHORTEST_PATHS**:

begin

$S := \{s\};\ \ell(s) := 0;\ p(s) := \emptyset;$

comment: S = set of the vertices already reached by a shortest path (current arborescence);

comment: $\ell(v)$ = length of the shortest s to v path that only uses vertices $\in S$;

comment: $p(v)$ = predecessor of v in the path of length $\ell(v)$;

for each $v \in V \setminus \{s\}$ **do**

$\ell(v) := w(s, v);$

$p(v) := s$

enddo;

while $S \neq V$ **do**

find $\overline{v} \in V \setminus S\ :\ \ell(\overline{v}) = \min_{v \in V \setminus S} \{\ell(v)\};$

$S := S \cup \{\overline{v}\};$

for each $v \in V \setminus S$ **do** (**comment**: update the labels)

if $\ell(\overline{v}) + w(\overline{v}, v) < \ell(v)$ **then**

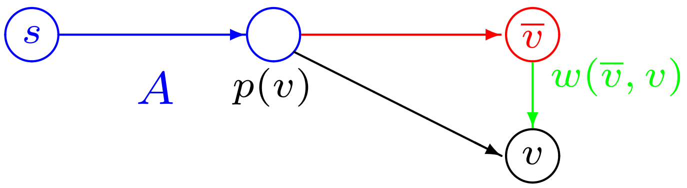
$\ell(v) := \ell(\overline{v}) + w(\overline{v}, v);$

$p(v) := \overline{v}$

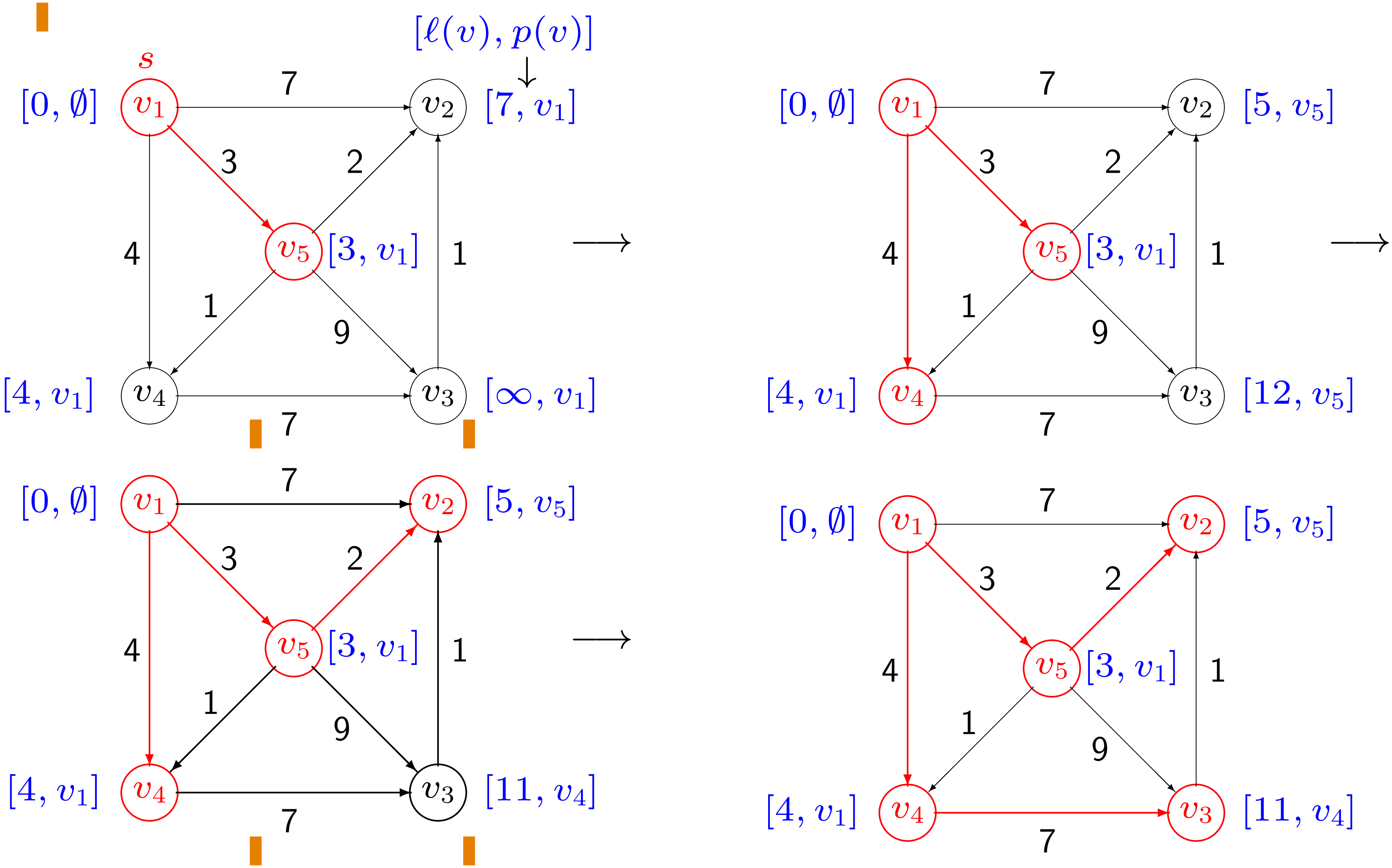
endif

endwhile

end.



Example



The shortest paths are obtained by through the predecessors (backward).

Dijkstra's algorithm (cont'd)

- **Theorem** *If $\ell(\overline{v}) = \min_{v \in V \setminus S} \{\ell(v)\}$, then the shortest path from s to $\overline{v}$ has length $\ell(\overline{v})$.*

  Proof We will show that any path P from s to $\overline{v}$ is at least long $\ell(\overline{v})$. Two possibilities exist:

  (a) if P only passes through vertices of S, the thesis is true by definition of ℓ;

  (b) otherwise, let h be the first vertex $\notin S$ on P: P is then given by the concatenation of

  – a path from s to h (of length at least $\ell(h) \geq \ell(\overline{v})$), and

  – a path from h to $\overline{v}$ (of length ≥ 0). $\square$

  (Note that the last consideration does not hold if the graph has negative cost arcs.)

- The algorithm performs $n - 1$ iterations of the **while** loop.
  At each iteration the number of operation is proportional to $|V \setminus S|$.
  The overall time is thus proportional to n^2: **the algorithm takes $O(n^2)$ time**.

- If we only need the **shortest path from s to a specified vertex t**

  we can halt the execution as soon as t is added to S

  but the algorithm still takes $O(n^2)$ time. (We are interested in the **worst case**).

- If we need the **shortest path between all pairs of vertices**

  we execute the algorithm n times (with $s = v_1$, $s = v_2$, ..., $s = v_n$): **$O(n^3)$ time**.

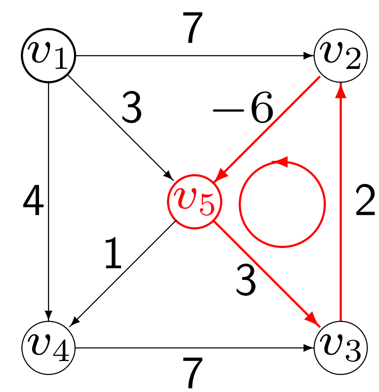
Shortest Path Problems (cont'd)

- **Case where the distance matrix can have negative entries:**
- **If** the graph contains **negative length circuits**

  then the problem has no meaning (shortest pats of length $-\infty$, by allowing to pass more than once through the same vertices);

  else (no negative circuit)

  - modified Dijkstra's algorithm: takes $O(n^3)$ time;
  - $O(n^3)$ algorithm by Floyd–Warshall (1962): shortest paths among all pairs of vertices;
  - both algorithms detect negative length circuits.

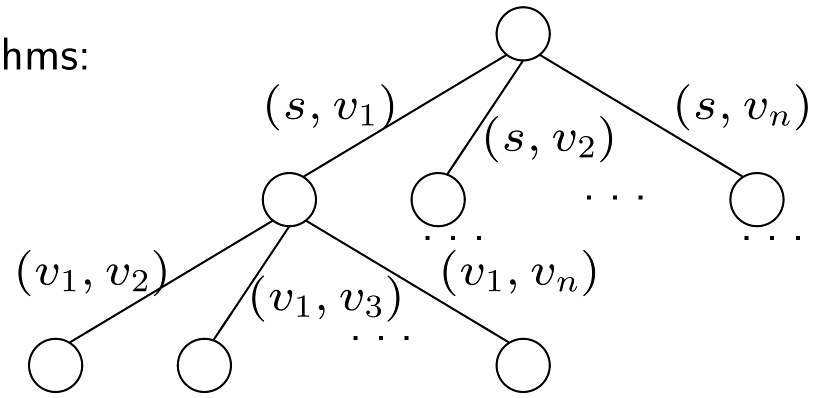


- In the **course web page**: **applets** for executing the Prim algorithm and the Dijkstra algorithm.

What if we want the longest path?

- SPP algorithms cannot be adapted.
- Solution obtained through **branch-and-bound** algorithms:

  $O((n-1)!)$ time in the worst case.
- The problem is $\mathcal{NP}$**-hard**.



Critical Path Method (CPM)

- Two main modeling and solution methods to plan and schedule complex projects (same theoretical foundations): PERT and CPM.
- These are among the most widely used graph theory based methodologies.
- Objective: handle the tasks involved in a given project, so as to determine the minimum time needed to complete the project.
- **Project** = set of **activities** of various **duration**, with **precedence** relationships:
  - **CPM** = **C**ritical **P**ath **M**ethod (deterministic activity times).
  - **PERT** = **P**rogram **E**valuation and **R**eview **T**echnique (probabilistic activity times);
  - independently developed by two different teams:
    - **CPM:** developed in 1957 by Catalytic Construction Company for scheduling the maintenance of the Du Pont de Nemours plants;
    - **PERT:** developed in 1958 by Booz, Allen & Hamilton, Inc. for the U.S. Navy Special Projects Office to optimize the U.S. Navy's Polaris nuclear submarine project;

      thousands of suppliers and subcontracts;

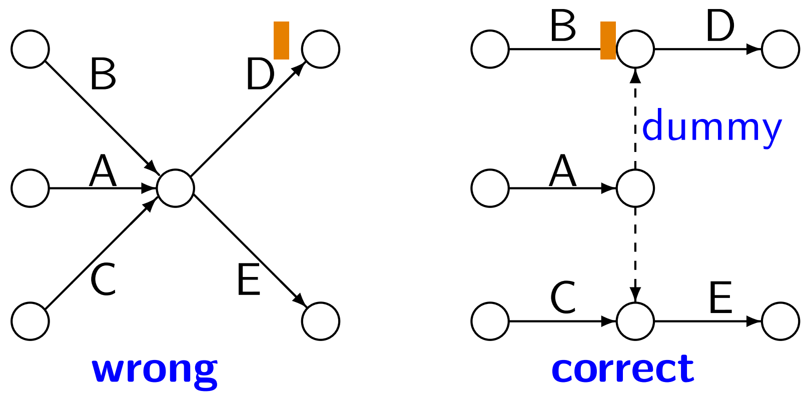
      results: expected time reduced by two years.
- **We will describe the CPM model and algorithm.**

Critical Path Method (cont'd)

- The activities involved in the project are represented through a weighted directed graph.
- Two equivalent approaches:
  - **AON** = **A**ctivities **O**n **N**odes;
  - **AOA** = **A**ctivities **O**n **A**rcs, **the one we will adopt**;
- arcs $a_h = (v_i, v_j)$ represent activities;
- vertices represent the start and end of activities;
- weight $d(v_i, v_j)$ is the duration of activity (v_i, v_j);
- the graph itself represents precedence relationships: to impose $a_i \prec a_j$, either
  - the ending vertex of a_i coincides with the starting vertex of a_j, or
  - $\exists$ a path containing a_i before a_j;
- **dummy activities** (of zero duration) can be used to impose precedences;
- the resulting graph must be **acyclic**.
- **Problem:**
  find the starting time of each activity so that the total duration (**makespan**) is a minimum.
- In the **course web page**: **applet** for applying the Critical Path Method.

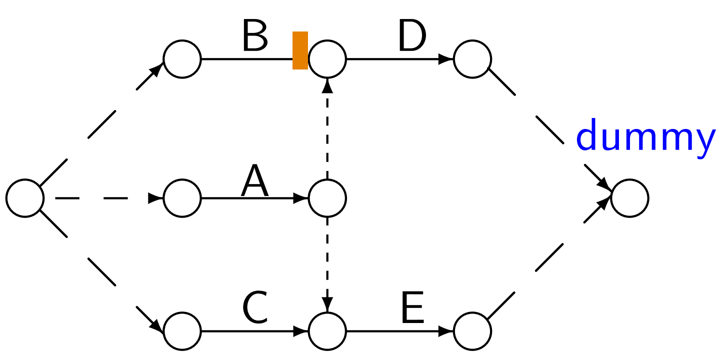
Critical Path Method (cont'd)

Example: A $\prec$ D, A $\prec$ E, B $\prec$ D, C $\prec$ E.



Step 1. the graph must have

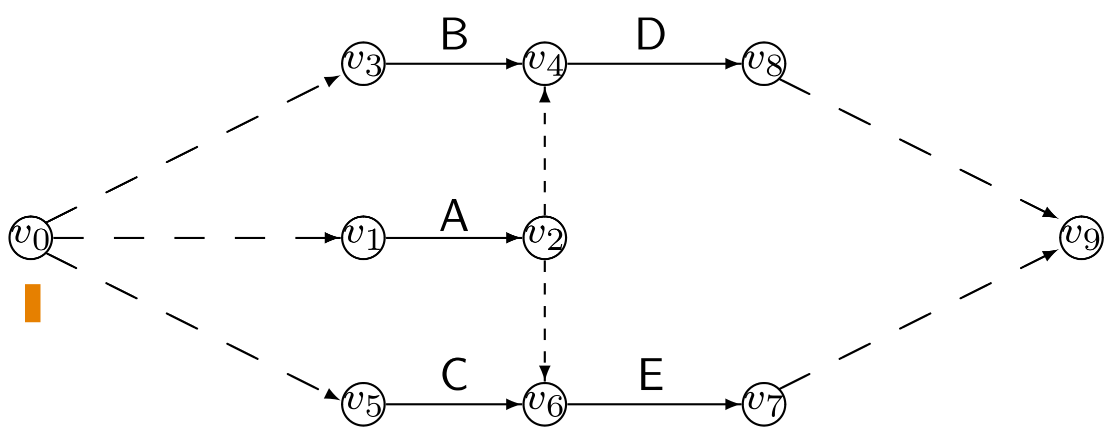
- a single **starting vertex** (in-degree $= 0$);
- a single **ending vertex** (out-degree $= 0$):



Critical Path Method (cont'd)

Step 2. The vertices are numbered so that $i < j \ \forall \ (v_i, v_j) \in A$ (possible: the graph is acyclic).

procedure Number:
begin
    if necessary, add to G dummy vertices v_0, v_{n+1}, and the corresponding arcs;
    $B := A$ (**comment:** working copy);
    $k := 0$;
    while $k \leq n + 1$ **do**
        begin (**comment:** Γ^- and Γ^+ refer to graph (V, B))
            select a non-numbered vertex $v : \Gamma^-(v) = \emptyset$;
            assign number k to v;
            $B := B \setminus \{(v, v_i) : v_i \in \Gamma^+(v)\}$;
            $k := k + 1$
        end
end.



Critical Path Method (cont'd)

Step 3. For each event (vertex) v_k, find:

- $TMIN_k$ earliest time at which the activity can start without violating its precedences;
- (**Makespan** (length of the longest path from v_0 to v_{n+1}) $= TMIN_{n+1}$.)
- $TMAX_k$ latest time at which the activity must to terminate without delaying the project.

A special algorithm, only valid for **acyclic** graphs, takes **polynomial time** $O(n^2)$:

procedure Critical Path:

begin

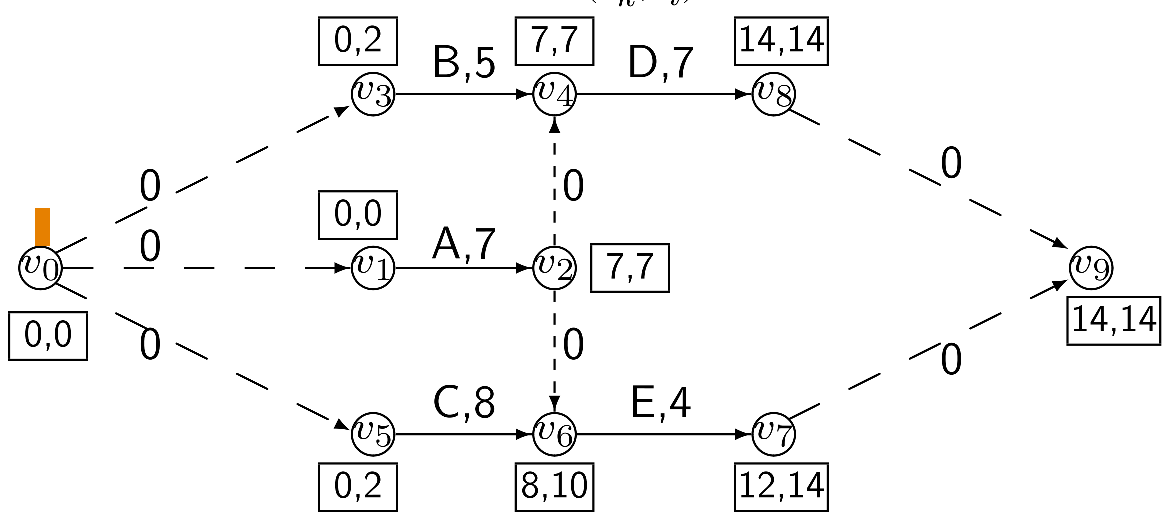
$TMIN_0 := 0$;

for $k := 1$ **to** $n+1$ **do** $TMIN_k := \max\limits_{i:(v_i,v_k)\in A} \{TMIN_i + d(v_i, v_k)\}$;

$TMAX_{n+1} := TMIN_{n+1}$;

for $k := n$ **downto** 0 **do** $TMAX_k := \min\limits_{i:(v_k,v_i)\in A} \{TMAX_i - d(v_k, v_i)\}$

end.



Critical Path Method (cont'd)

Step 4. For each activity (arc) $a_h = (v_i, v_j)$, compute

- **Early Start Time**: $EST(a_h) = TMIN_i$;
- **Late Start Time**: $LST(a_h) = TMAX_j - d(v_i, v_j)$;
- **Float**: $S(a_h) = LST(a_h) - EST(a_h)$ (if $LST(a_h) = EST(a_h)$ then a_h is **critical**);
- **Critical path** $=$ path from v_0 to v_{n+1} containing only critical activities.

Activity	i	j	$EST(v_i, v_j)$	$LST(v_i, v_j)$	$S(v_i, v_j)$
A	1	2	0	0	0
B	3	4	0	2	2
C	5	6	0	2	2
D	4	8	7	7	0
E	6	7	8	10	2

- A and D are critical. Critical path: $\{v_0, v_1, v_2, v_4, v_8, v_9\}$.
- Solution illustrated through a **Gantt chart**: