Algorithms and Data Structures

C6. Shortest Paths: Algorithms

Gabriele Röger and Patrick Schnider

University of Basel

May 21, 2025

Edsger Dijkstra

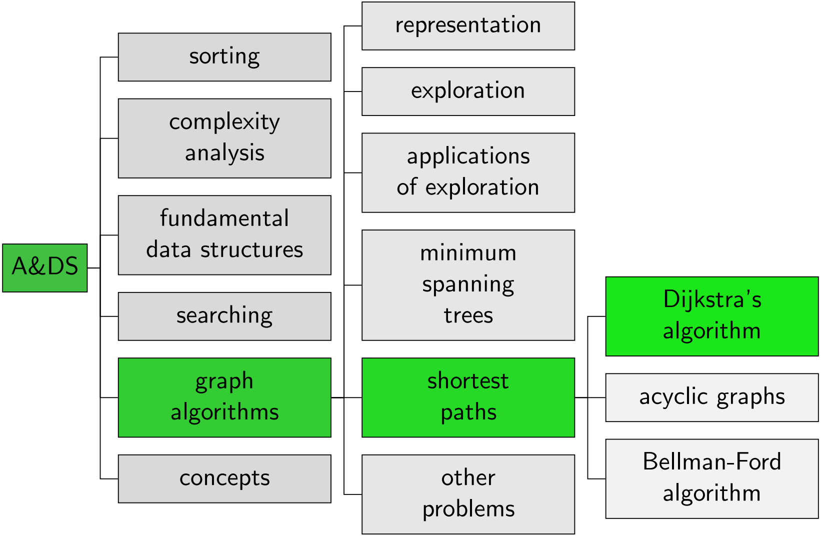
Edsger Dijkstra

- Dutch mathematician, 1930–2002
- Advocate and co-developer of structured programming
  - Contributed to the development of programming language Algol 60
  - 1968: Essay "Go To Statement Considered Harmful"
- 1959: Shortest-path algorithm
- Winner of Turing Award (1972)

"Do only what only you can do."

Dijkstra's Algorithm

Content of the Course



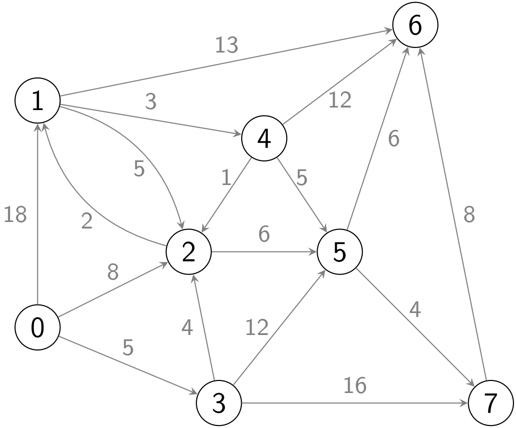
Dijkstra's Algorithm: High-Level Perspective

Dijkstra's algorithm (for non-negative edge weights)

Grow shortest-paths tree starting from vertex s:

- Consider vertices (that are not yet in the tree) in increasing order of their distance from s.
- Add the next vertex to the tree and relax its outgoing edges.

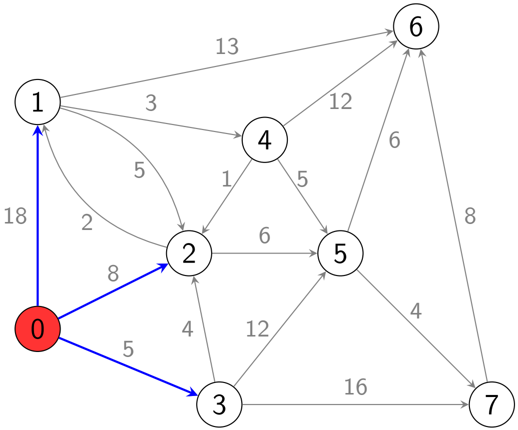
Dijkstra's Algorithm: Illustration



distance

0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞

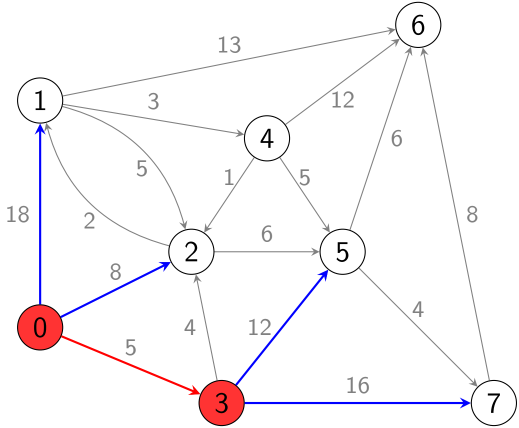
Dijkstra's Algorithm: Illustration



distance

0	0
1	18
2	8
3	5
4	∞
5	∞
6	∞
7	∞

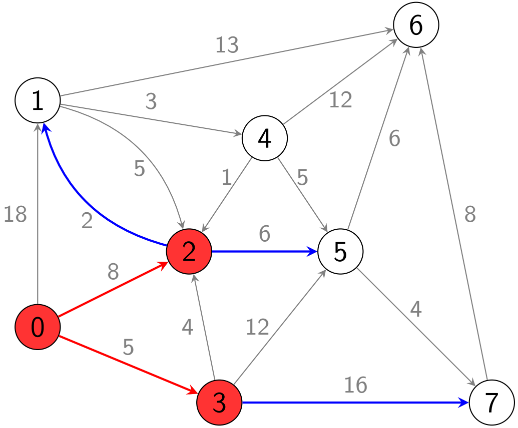
Dijkstra's Algorithm: Illustration



`distance`

0	0
1	18
2	8
3	5
4	∞
5	17
6	∞
7	21

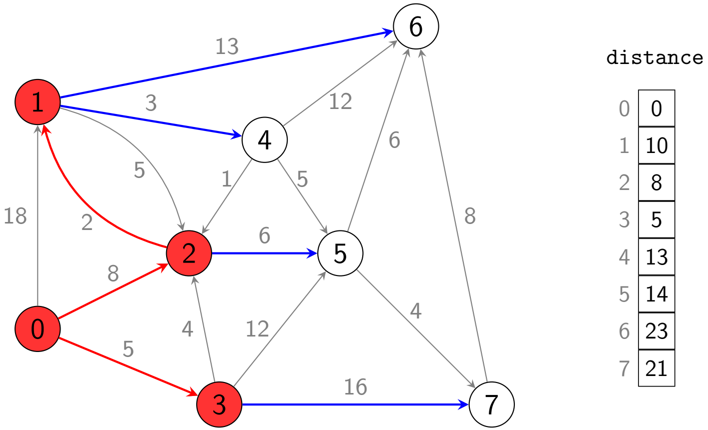
Dijkstra's Algorithm: Illustration



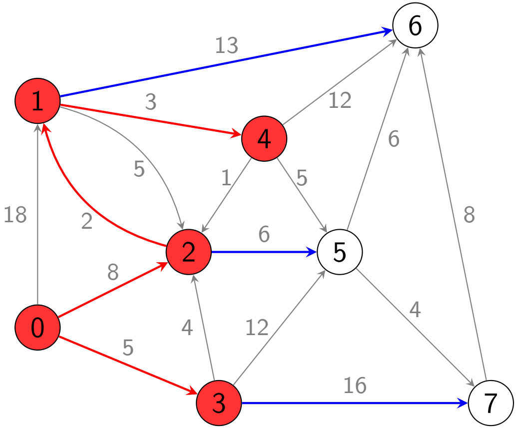
distance

0	0
1	10
2	8
3	5
4	∞
5	14
6	∞
7	21

Dijkstra's Algorithm: Illustration



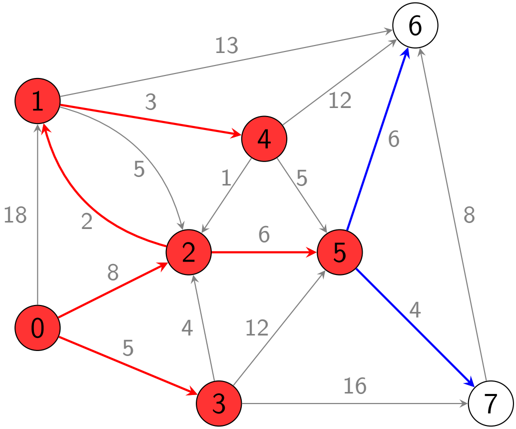
Dijkstra's Algorithm: Illustration



distance

0	0
1	10
2	8
3	5
4	13
5	14
6	23
7	21

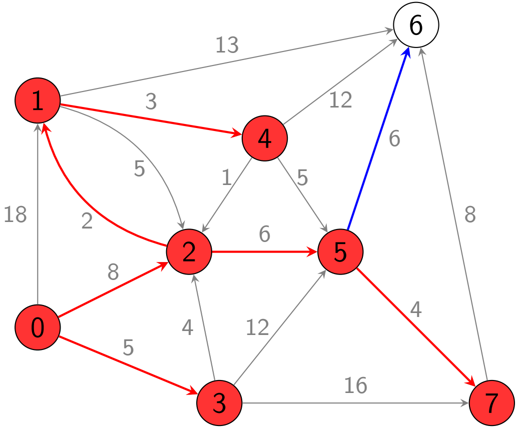
Dijkstra's Algorithm: Illustration



distance

0	0
1	10
2	8
3	5
4	13
5	14
6	20
7	18

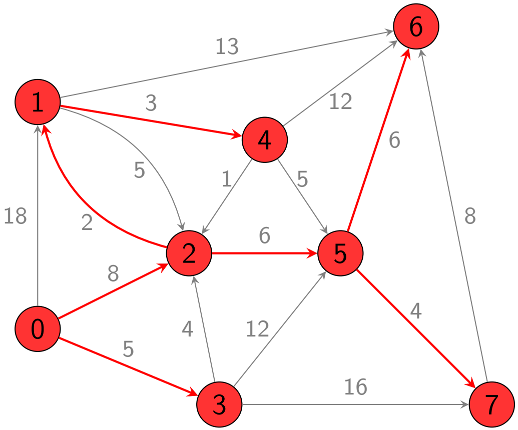
Dijkstra's Algorithm: Illustration



`distance`

0	0
1	10
2	8
3	5
4	13
5	14
6	20
7	18

Dijkstra's Algorithm: Illustration



distance

0	0
1	10
2	8
3	5
4	13
5	14
6	20
7	18

Data Structures

- edge_to: vertex-indexed array, containing at position v the last edge of a shortest known path.

Data Structures

- edge_to: vertex-indexed array, containing at position v the last edge of a shortest known path.
- distance: vertex-indexed array, containing at position v the cost of the shortest known paths from the start vertex to v.

Data Structures

- `edge_to`: vertex-indexed array, containing at position *v* the last edge of a shortest known path.
- `distance`: vertex-indexed array, containing at position *v* the cost of the shortest known paths from the start vertex to *v*.
- `pq`: indexed priority queue of vertices
  - vertex not yet in the tree
  - some path to the vertex is known
  - sorted by the cost of the shortest known path to the vertex.

Dijkstra's Algorithm

```
class DijkstraSSSP:
    def __init__(self, graph, start_node):
        self.edge_to = [None] * graph.no_nodes()
        self.distance = [float('inf')] * graph.no_nodes()
        pq = IndexMinPQ()
        self.distance[start_node] = 0
        pq.insert(start_node, 0)
        while not pq.empty():
            self.relax(graph, pq.del_min(), pq)

    def relax(self, graph, v, pq):
        for edge in graph.outgoing_edges(v):
            w = edge.to_node()
            if self.distance[v] + edge.weight() < self.distance[w]:
                self.edge_to[w] = edge
                self.distance[w] = self.distance[v] + edge.weight()
                if pq.contains(w):
                    pq.change(w, self.distance[w])
                else:
                    pq.insert(w, self.distance[w])
```

Correctness

Theorem

Dijkstra's algorithm solves the single-source shortest path problem in digraphs with non-negative edge weights.

Proof.

- If v is reachable from the start vertex, every outgoing edge $e = (v, w)$ will be relaxed exactly once (when v is relaxed).
- It then holds that $distance[w] \leq distance[v] + weight(e)$.
- Inequality stays satisfied:
  - $distance[v]$ won't be changed because the value was minimal and there are no negative edge weights.
  - $distance[w]$ can only become smaller.
- If all reachable edges have been relaxed, the optimality criterion is satisfied.

□

Comparison to Prim's Algorithm

Dijkstra's algorithm is very similar to the eager variant of Prim's algorithm for minimum spanning trees.

- Both successively grow a tree.
- Prim's next vertex: minimal distance from the grown tree.
- Dijkstra's next vertex: minimal distance from the start vertex.

Comparison to Prim's Algorithm

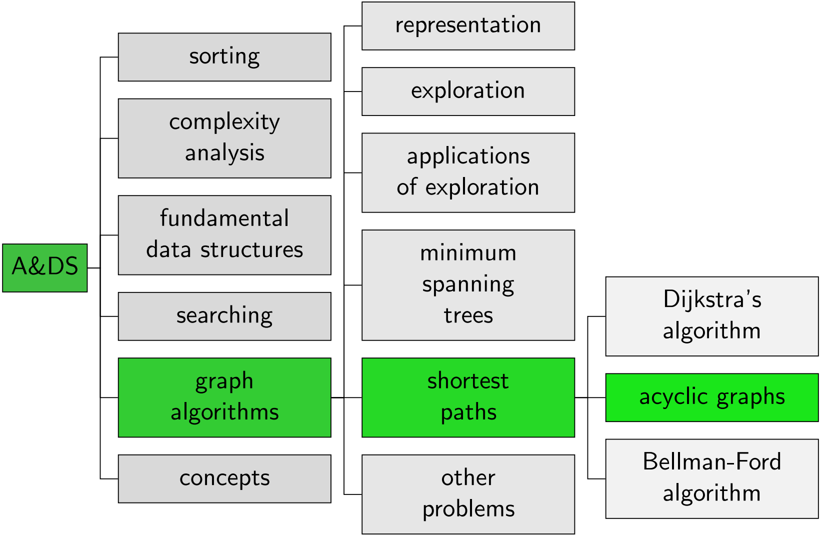
Dijkstra's algorithm is very similar to the eager variant of Prim's algorithm for minimum spanning trees.

- Both successively grow a tree.
- Prim's next vertex: minimal distance from the grown tree.
- Dijkstra's next vertex: minimal distance from the start vertex.

Running time $O(|E| \log |V|)$ and memory $O(|V|)$ directly transfer.

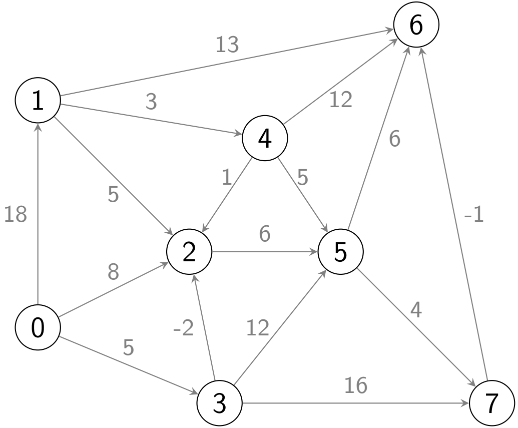
Acyclic Graphs

Content of the Course



Exploiting Acyclicity

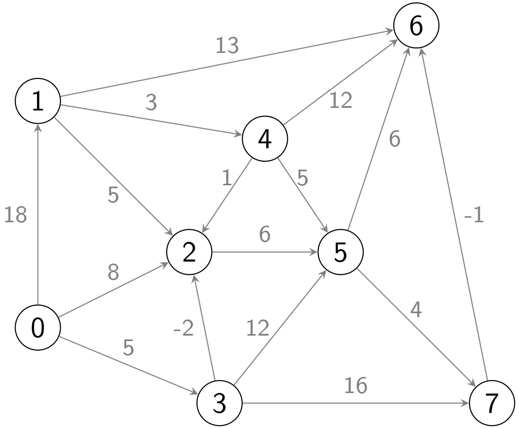
Given: acyclic weighted digraph



Can we exploit acylicity during the computation of shortest paths?

Example

Idea: Relax vertices in topological order
e.g. 0, 1, 3, 4, 2, 5, 7, 6

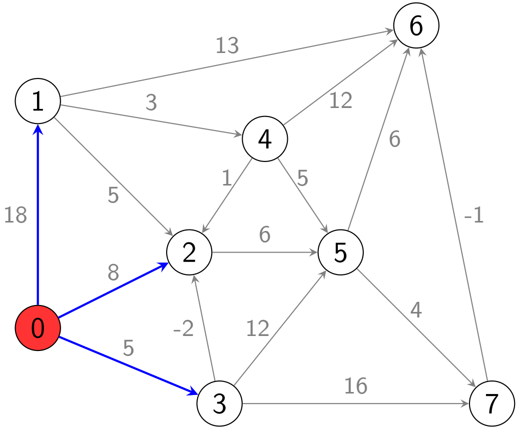


distance

0	0
1	∞
2	∞
3	∞
4	∞
5	∞
6	∞
7	∞

Example

Idea: Relax vertices in topological order
e.g. 0, 1, 3, 4, 2, 5, 7, 6

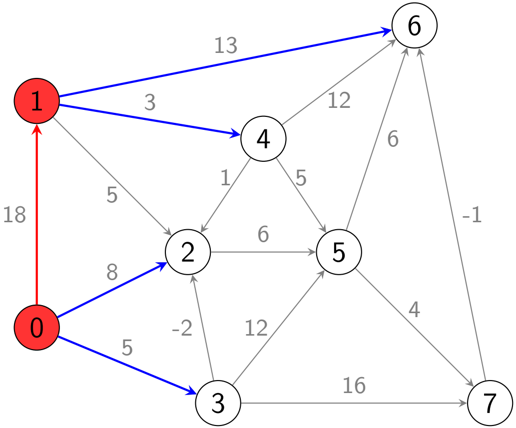


distance

0	0
1	18
2	8
3	5
4	∞
5	∞
6	∞
7	∞

Example

Idea: Relax vertices in topological order
e.g. 0, 1, 3, 4, 2, 5, 7, 6

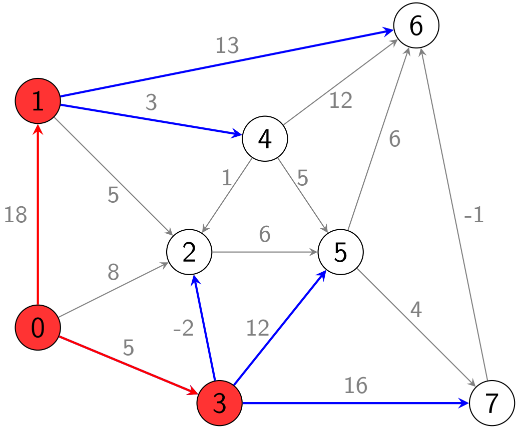


distance

0	0
1	18
2	8
3	5
4	21
5	∞
6	31
7	∞

Example

Idea: Relax vertices in topological order
e.g. 0, 1, 3, 4, 2, 5, 7, 6

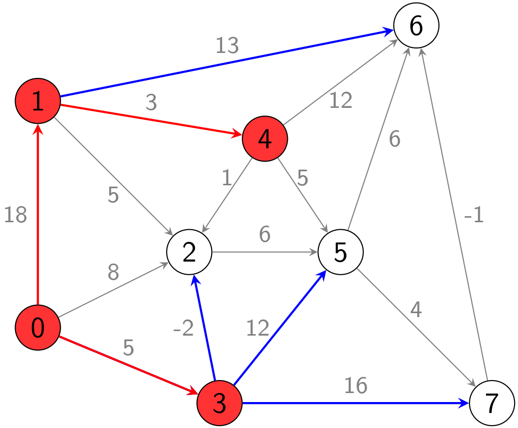


distance

0	0
1	18
2	3
3	5
4	21
5	17
6	31
7	21

Example

Idea: Relax vertices in topological order
e.g. 0, 1, 3, 4, 2, 5, 7, 6

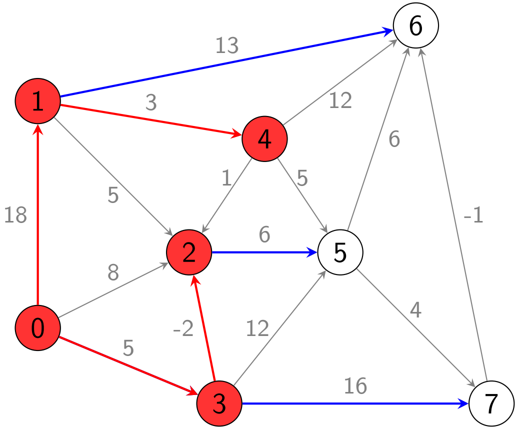


distance

0	0
1	18
2	3
3	5
4	21
5	17
6	31
7	21

Example

Idea: Relax vertices in topological order
e.g. 0, 1, 3, 4, 2, 5, 7, 6

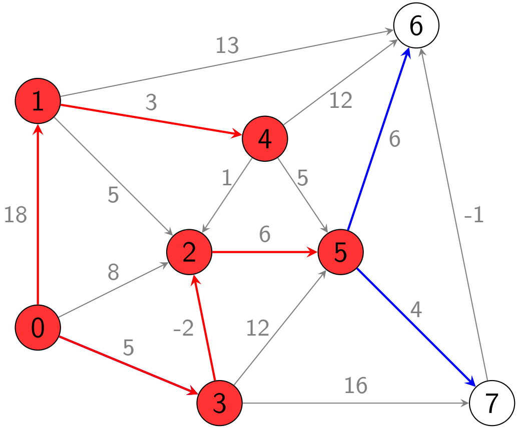


distance

0	0
1	18
2	3
3	5
4	21
5	9
6	31
7	21

Example

Idea: Relax vertices in topological order
e.g. 0, 1, 3, 4, 2, 5, 7, 6

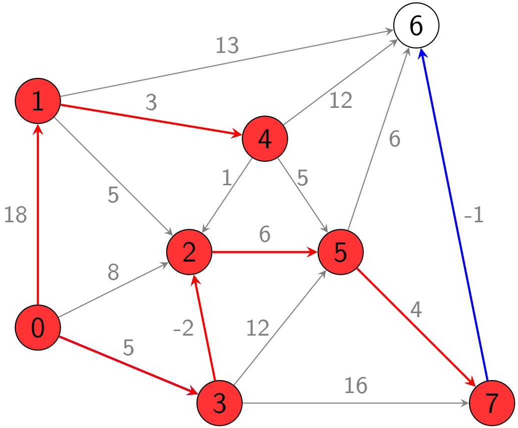


distance

0	0
1	18
2	3
3	5
4	21
5	9
6	15
7	13

Example

Idea: Relax vertices in topological order
e.g. 0, 1, 3, 4, 2, 5, 7, 6

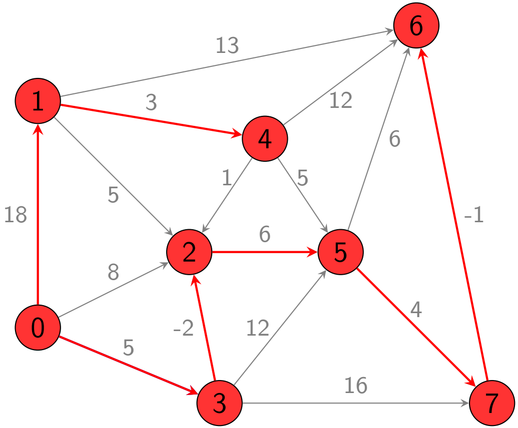


distance

0	0
1	18
2	3
3	5
4	21
5	9
6	12
7	13

Example

Idea: Relax vertices in topological order
e.g. 0, 1, 3, 4, 2, 5, 7, 6



	distance
0	0
1	18
2	3
3	5
4	21
5	9
6	12
7	13

Theorem

Theorem

Relaxing the vertices in topological order, we can solve the single-source shortest path problem for weighted acyclic digraphs in time $O(|E| + |V|)$.

Theorem

Theorem

Relaxing the vertices in topological order, we can solve the single-source shortest path problem for weighted acyclic digraphs in time $O(|E| + |V|)$.

Proof.

- Every edge $e = (v, w)$ gets relaxed exactly once.
  Directly afterwards it holds that
  $\texttt{distance}[w] \leq \texttt{distance}[v] + \mathit{weight}(e)$.
- Inequality satisfied until termination
  - $\texttt{distance}[w]$ never becomes larger.
  - $\texttt{distance}[v]$ does not get changed anymore because all incoming edges have already been relaxed.

→ Optimality criterion is satisfied at termination. □

Related Problems: Longest Path

Definition (Longest paths in acyclic graphs)

Given: weighted acyclic digraph, start vertex s

Question: Is there a path from s to vertex v?
If yes, return such a path with maximum weight.

Related Problems: Longest Path

Definition (Longest paths in acyclic graphs)

Given: weighted acyclic digraph, start vertex s

Question: Is there a path from s to vertex v?
If yes, return such a path with maximum weight.

Multiply all weights with -1 and use shortest-path algorithm.

Related Problems: Critical Path

Given:

- Set of jobs a, each requires time t_a
- Constraints $a \to a'$, requiring that a must have been finished before a' can be started (in solvable problems acyclic).

Related Problems: Critical Path

Given:

- Set of jobs a, each requires time t_a
- Constraints $a \rightarrow a'$, requiring that a must have been finished before a' can be started (in solvable problems acyclic).

Question:

- Assumption: We can do arbitrarily many jobs in parallel.
- How long do we need for getting all jobs done?

Related Problems: Critical Path

Create a weighted digraph:

- Vertices s, e + for every job a two vertices a_s and a_e
- for all a:
  - edge (s, a_s) with weight 0
  - edge (a_e, e) with weight 0
  - edge $(a_\mathrm{s}, a_\mathrm{e})$ with weight t_a
- for every constraint $a \rightarrow a'$ edge $(a_\mathrm{e}, a'_\mathrm{s})$ with weight 0

Related Problems: Critical Path

Create a weighted digraph:

- Vertices s, e + for every job a two vertices a_s and a_e
- for all a:
  - edge (s, a_s) with weight 0
  - edge (a_e, e) with weight 0
  - edge (a_s, a_e) with weight t_a
- for every constraint $a \to a'$ edge (a_e, a'_s) with weight 0

Critical path for job a is longest path from s to a_s.
Define start time for a as weight of a critical path.

Related Problems: Critical Path

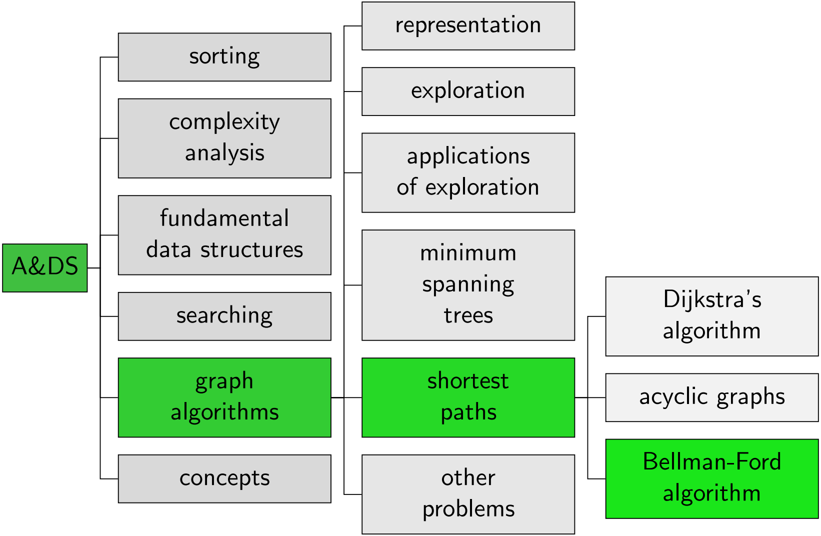
Create a weighted digraph:

- Vertices s, e + for every job a two vertices a_s and a_e
- for all a:
  - edge (s, a_s) with weight 0
  - edge (a_e, e) with weight 0
  - edge (a_s, a_e) with weight t_a
- for every constraint $a \rightarrow a'$ edge (a_e, a'_s) with weight 0

Critical path for job a is longest path from s to a_s.
Define start time for a as weight of a critical path.
$\rightarrow$ Results in optimal total execution time
($=$ weight of longest path from s to e)

Bellman-Ford Algorithm

Content of the Course



Problem

- With negative edge weights there can be negative cycles, i.e. cycles, where the sum of edge weights is negative.
- If a vertex of such a cycle is on a path from s to v, we can find paths whose weight is lower than any given value.
  $\rightarrow$ not a well-defined problem
- Alternative question: Find a shortest simple path?
  $\rightarrow$ NP-hard (= very hard) problem

Question

In many practical applications, negative cycles indicate a modeling error.

New Questions

Given: Weighted digraph, start vertex s

Question: Is there a negative cycle that is reachable from s?
If not, compute the shortest-path tree
to all reachable vertices.

Bellman-Ford Algorithm: High-Level Perspective

In graphs without negative cycles (but with negative weights);

Bellman-Ford Algorithm

- Initialize $distance[s] = 0$ for start vertex s, $distance[n] = \infty$ for all other vertices.
- Afterwards $|V|$ iterations, each relaxing all edges.

Bellman-Ford Algorithm: High-Level Perspective

In graphs without negative cycles (but with negative weights);

Bellman-Ford Algorithm

- Initialize $distance[s] = 0$ for start vertex s, $distance[n] = \infty$ for all other vertices.
- Afterwards $|V|$ iterations, each relaxing all edges.

Proposition

The approach solves the single-source shortest path problem for graphs without negative cycles in time $O(|E||V|)$ and with additional memory $O(|V|)$.

Proof idea: After i iterations, every found path to v has at most the weight as any path to v with at most i edges.

More Efficient Variant

- If *distance*[*v*] did not change in iteration i, relaxing an outgoing edge of v in iteration $i+1$ has no effect.
- Idea: Remember the vertices with a changed *distance* in a queue.
- Does not improve the worst-case behavior but in practice much faster.

What about Negative Cycles?

- If no negative cycles is reachable from s, then in the $|V|$-th iteration no vertex distance will get updated anymore.
- If there is a reachable negative cycle, this will lead to a cycle in the edges stored in `edge_to`.
- In practice, we test this after relaxing the outgoing edges of certain number of vertices (e.g. $|V|$ many).

Bellman-Ford Algorithm

```
class BellmanFordSSSP:
    def __init__(self, graph, start_node):
        self.edge_to = [None] * graph.no_nodes()
        self.distance = [float('inf')] * graph.no_nodes()
        self.in_queue = [False] * graph.no_nodes()
        self.queue = deque()
        self.calls_to_relax = 0
        self.cycle = None

        self.distance[start_node] = 0
        self.queue.append(start_node)
        self.in_queue[start_node] = True
        while (not self.has_negative_cycle() and
               self.queue):  # queue not empty
            node = self.queue.popleft()
            self.in_queue[node] = False
            self.relax(graph, node)

```

Bellman-Ford Algorithm (Continued)

```
    def relax(self, graph, v):
        for edge in graph.outgoing_edges(v):
            w = edge.to_node()
            if self.distance[v] + edge.weight() < self.distance[w]:
                self.edge_to[w] = edge
                self.distance[w] = self.distance[v] + edge.weight()
                if not self.in_queue[w]:
                    self.queue.append(w)
                    self.in_queue[w] = True
        self.calls_to_relax += 1
        if self.calls_to_relax % graph.no_nodes() == 0:
            self.find_negative_cycle()
```

Bellman-Ford Algorithm (Continued)

```
    def has_negative_cycle(self):
        return self.cycle is not None

    def find_negative_cycle(self):
        no_nodes = len(self.distance)
        graph = EdgeWeightedDigraph(no_nodes)
        for edge in self.edge_to:
            if edge is not None:
                graph.add_edge(edge)

        cycle_finder = WeightedDirectedCycle(graph)
        self.cycle = cycle_finder.get_cycle()
```

`WeightedDirectedCycle` detects directed cycles in weighted graphs.

→ Sequence of depth-first searches as in `DirectedCycle` (C2)

Summary

Summary

- Non-negative weights
  - Very common problem.
  - Dijkstra's Algorithm with running time $O(|E| \log |V|)$
- Acyclic Graphs
  - Should be exploited if it occurs in an application.
  - With topological order in linear time $O(|E| + |V|)$
- Negative weights or negative cycles
  - If there is no negative cycle, the Bellman-Ford algorithm finds shortest paths.
  - Otherwise it identifies a negative cycle.