Algorithmics and Complexity
Lecture 2/7 : Shortest paths algorithm

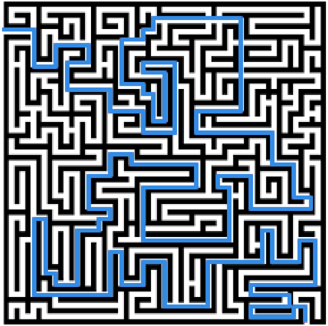
CentraleSupélec – Gif

ST2 – Gif

Plan

Plan

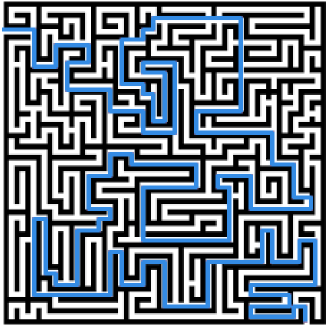
Reminder: maze problem



Searching for a path in a graph

- Depth-first search and breadth-first search
- Both produce a path

Reminder: maze problem

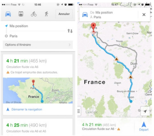


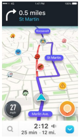
Searching for a path in a graph

- Depth-first search and breadth-first search
- Both produce a path

➜ *What happens if there are many?*

Application: Waze or Google Itinerary

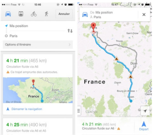


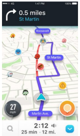


Find the **best** path

➜ Optimization problem

Application: Waze or Google Itinerary

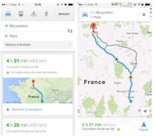


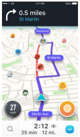


Find the best path

- ➜ Optimization problem
- ✓ Breadth-first search in an undirected graph
  - ➜ *Produces, indeed, a path made up of the smallest number of edges*

Application: Waze or Google Itinerary





Find the **best** path

- ➜ Optimization problem
- ✓ Breadth-first search in an undirected graph
  - ➜ *Produces, indeed, a path made up of the smallest number of edges*
- ✗ Not all roads are two-ways
  - ➜ Directed graphs
- ✗ Each road segment requires a different passing time
  - ➜ Weighted edges

Optimization: naive approach

Principle

Produce all possible paths and pick up the shortest one

Optimization: naive approach

Principle

Produce all possible paths and pick up the shortest one

Simplified algorithm

```
def all_paths(G,s,t):
    C = all_paths_between_s_and_t_in_G(G,s,t)
    return min(C, key=length)
```

Optimization: naive approach

Principle

Produce all possible paths and pick up the shortest one

Simplified algorithm

```
def all_paths(G,s,t):
    C = all_paths_between_s_and_t_in_G(G,s,t)
    return min(C, key=length)
```

Difficulty

There are up to $\mathcal{O}(|E|!)$ elements in C!

➜ Can we propose an efficient algorithm?

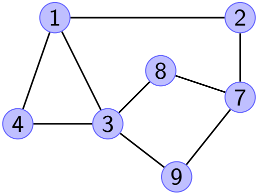
Plan

Graphs: definitions

Reminder: undirected graph

We consider $G = (V, E)$, where:

- V a set of vertices (or nodes);
- E a set of edges;
- An edge $e \in E$ is a pair of vertices from V;

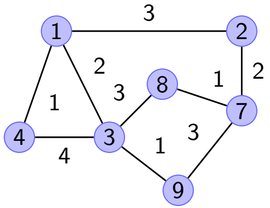


Graphs: definitions

Undirected weighted graph

We consider $G = (V, E)$, where:

- V a set of vertices (or nodes);
- E a set of edges;
- An edge $e \in E$ is a pair of vertices from V;
- $\omega : E \longrightarrow \mathbb{R}$ is a weight function (of edges);

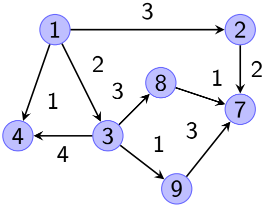


Graphs: definitions

~~Un~~Directed weighted graph

We consider $G = (V, E)$, where:

- V a set of vertices (or nodes);
- E a set of arcs;
- An arc $e \in E$ is a couple of vertices from V;
- $\omega : E \longrightarrow \mathbb{R}$ is a weight function (of arcs);



Graphs: definitions (continuation)

Path

In a directed graph:

- A path from x to y is a sequence of consecutive arcs connecting x to y.

Graphs: definitions (continuation)

Path

In a directed graph:

- A path from x to y is a sequence of consecutive arcs connecting x to y.

Distance

In a weighted directed graph:

- The cost of a path c is the sum of weights of the arcs on c:

$$cost(c) = \sum_{e \in c} \omega(e)$$

Graphs: definitions (continuation)

Path

In a directed graph:

- A path from x to y is a sequence of consecutive arcs connecting x to y.

Distance

In a weighted directed graph:

- The cost of a path c is the sum of weights of the arcs on c:

$$cost(c) = \sum_{e \in c} \omega(e)$$

- One can also say distance from x to y (for c connecting x to y).

Data structure

Adjacency list

- Memory space in $\mathcal{O}(|E| + |V|)$
- Browsing the set of neighbours of a vertex u in $\mathcal{O}(deg(u))$
- Storage of weights: `{a:{b:2,c:3},...}`
- Access to the weight of an arc in $\mathcal{O}(1)$

  `like add an arc, delete an arc,...`

Data structure

Adjacency list

- Memory space in $\mathcal{O}(|E| + |V|)$
- Browsing the set of neighbours of a vertex u in $\mathcal{O}(deg(u))$
- Storage of weights: `{a:{b:2,c:3},...}`
- Access to the weight of an arc in $\mathcal{O}(1)$
  `like add an arc, delete an arc,...`

Adjacency matrix

- Memory space in $\mathcal{O}(|V|^2)$
- Browsing the set of neighbours of a vertex u in $\mathcal{O}(|V|)$
- Storage of weights: $tab[i, j] = \omega(i, j)$
- Access to the weight of an arc in $\mathcal{O}(1)$
  `like add an arc, delete an arc,...`

The shortest path problem

Optimization problem

Input:

- Directed graph $G = (V, E)$
- Weight function $\omega : E \longrightarrow \mathbb{R}$
- Source $s \in V$ and terminal $t \in V$

The shortest path problem

Optimization problem

Input:

- Directed graph $G = (V, E)$
- Weight function $\omega : E \longrightarrow \mathbb{R}$
- Source $s \in V$ and terminal $t \in V$

Question:

➜ What is the shortest path from s to t?
Let C be a set of possible solutions (connecting s to t). We are looking for $c \in C$ such that $\forall c' \in C, cost(c') \geq cost(c)$

The shortest path problem

Optimization problem

Input:

- Directed graph $G = (V, E)$
- Weight function $\omega : E \longrightarrow \mathbb{R}$
- Source $s \in V$ and terminal $t \in V$

Question:

➜ What is the shortest path from s to t?
Let C be a set of possible solutions (connecting s to t). We are looking for $c \in C$ such that $\forall c' \in C, cost(c') \geq cost(c)$

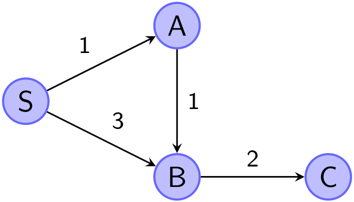
Observation

The problem definition also holds for undirected weighted graphs.

A shortest path

Idea of algorithm (inspired by Dijkstra, 1959)

Breadth-first search taking weights into account.

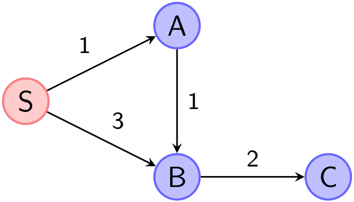


skip

A shortest path

Idea of algorithm (inspired by Dijkstra, 1959)

Breadth-first search taking weights into account.

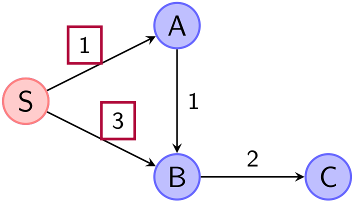


skip

A shortest path

Idea of algorithm (inspired by Dijkstra, 1959)

Breadth-first search taking weights into account.

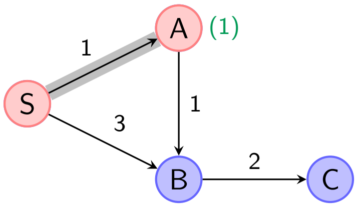


skip

A shortest path

Idea of algorithm (inspired by Dijkstra, 1959)

Breadth-first search taking weights into account.

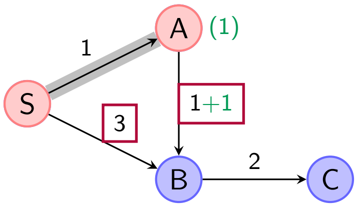


skip

A shortest path

Idea of algorithm (inspired by Dijkstra, 1959)

Breadth-first search taking weights into account.

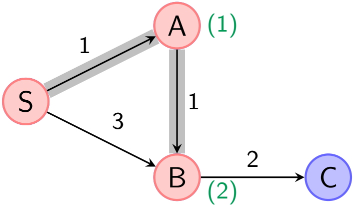


skip

A shortest path

Idea of algorithm (inspired by Dijkstra, 1959)

Breadth-first search taking weights into account.

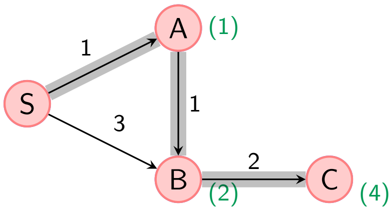


skip

A shortest path

Idea of algorithm (inspired by Dijkstra, 1959)

Breadth-first search taking weights into account.



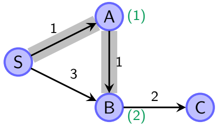
etc.

Shortest path algorithm

Data structures

The algorithm requires:

- The list of vertices to be visited (as in BFS)
  - ➜ In this context the term frontier is commonly used.
- The cost s of the best path at each vertex already visited
  - ➜ Store a distance at each vertex.
- Selected arcs
  - ➜ Store a predecessor for each vertex.



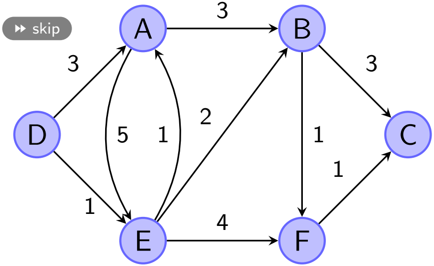
Shortest path algorithm in python

```
def shortest_path(graph,s):
    frontier = [s]
    parent = {}
    parent[s] = None
    dist = {}
    dist[s] = 0

    while len(frontier)>0:
        x = extract_min_dist(frontier,dist)
        for y in neighbors(graph, x):
            if y not in parent:
                frontier.append(y)
            # update
            new_dist = dist[x] + distance(graph,x,y)
            if y not in dist or dist[y] > new_dist:
                dist[y] = new_dist
                parent[y] = x

    return parent
```

Complete example

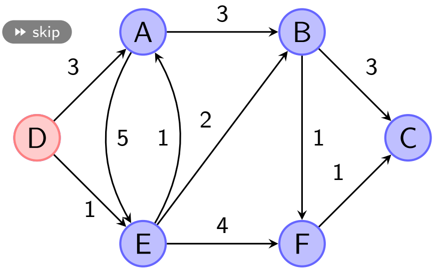


Node	Distance	Parent
A	∞	•
B	∞	•
C	∞	•
D	0	•
E	∞	•
F	∞	•

Frontier = $\{D\}$

x =

Complete example

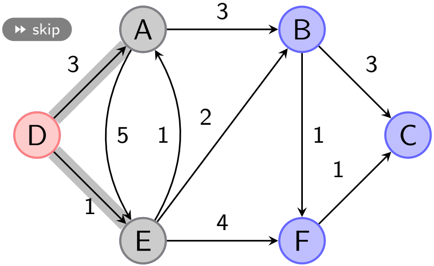


Node	Distance	Parent
A	∞	•
B	∞	•
C	∞	•
D	0	•
E	∞	•
F	∞	•

Frontier = $\{\}$

x = D

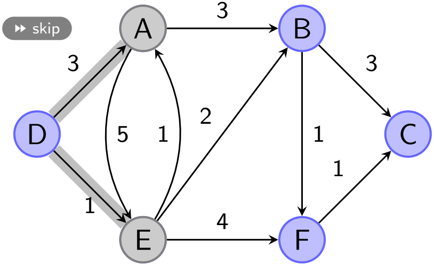
Complete example



Node	Distance	Parent
A	∞	•
B	∞	•
C	∞	•
D	0	•
E	∞	•
F	∞	•

Frontier = $\{A, E\}$
x = D

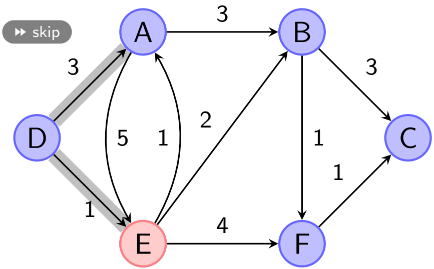
Complete example



Node	Distance	Parent
A	3	D
B	∞	•
C	∞	•
D	0	•
E	1	D
F	∞	•

Frontier $= \{A, E\}$
x $= D$

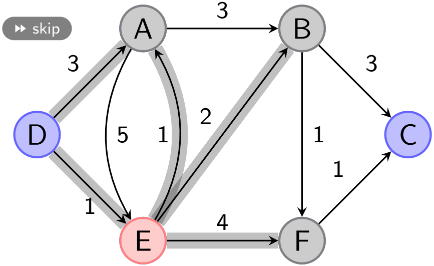
Complete example



Node	Distance	Parent
A	3	D
B	∞	•
C	∞	•
D	0	•
E	1	D
F	∞	•

Frontier $= \{A\}$
x $= E$

Complete example

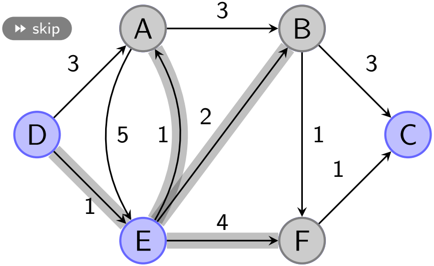


Node	Distance	Parent
A	3	D
B	∞	•
C	∞	•
D	0	•
E	1	D
F	∞	•

Frontier = $\{A, B, F\}$

x = E

Complete example

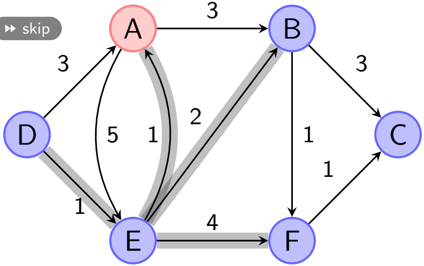


Node	Distance	Parent
A	1+1	E
B	1+2	E
C	∞	•
D	0	•
E	1	D
F	1+4	E

Frontier $= \{A, B, F\}$
x $= E$

Complete example

skip

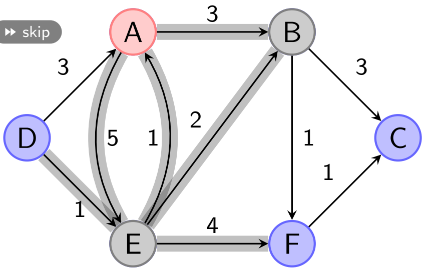


Node	Distance	Parent
A	2	E
B	3	E
C	∞	•
D	0	•
E	1	D
F	5	E

Frontier = $\{B, F\}$
x = A

Complete example

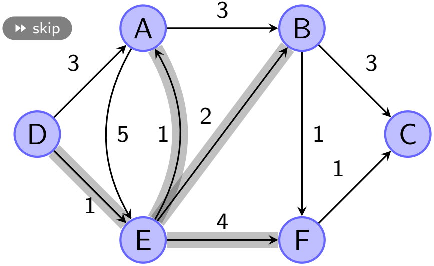
skip



Node	Distance	Parent
A	2	E
B	3	E
C	∞	•
D	0	•
E	1	D
F	5	E

Frontier $= \{B, F\}$
x $= A$

Complete example

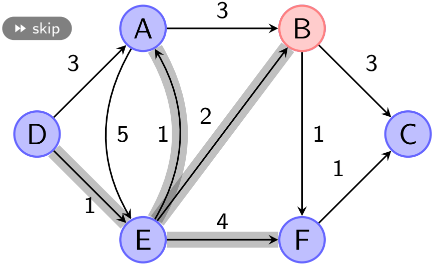


Node	Distance	Parent
A	2	E
B	3	E
C	∞	•
D	0	•
E	1	D
F	5	E

Frontier = $\{B, F\}$

x = A

Complete example

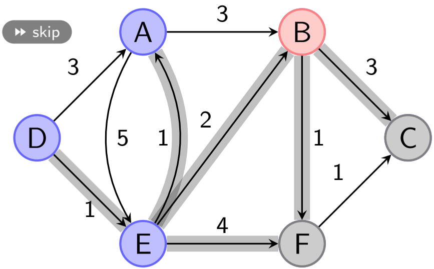


Node	Distance	Parent
A	2	E
B	3	E
C	∞	•
D	0	•
E	1	D
F	5	E

Frontier $= \{F\}$

x $= B$

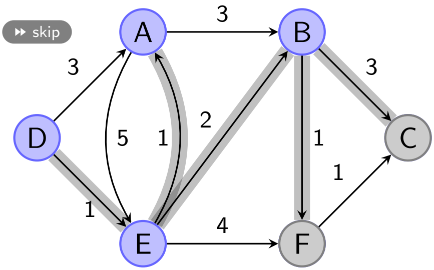
Complete example



Node	Distance	Parent
A	2	E
B	3	E
C	∞	•
D	0	•
E	1	D
F	5	E

Frontier $= \{C, F\}$
x $= B$

Complete example

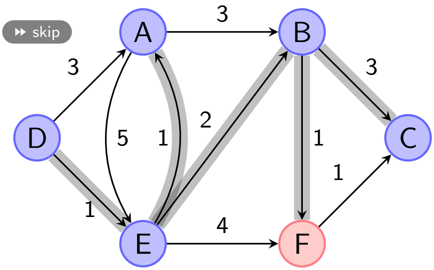


Node	Distance	Parent
A	2	E
B	3	E
C	3+3	B
D	0	•
E	1	D
F	3+1	B

Frontier = $\{C, F\}$

x = B

Complete example



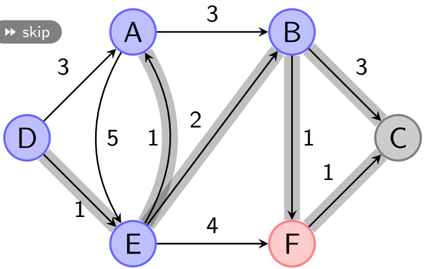
Node	Distance	Parent
A	2	E
B	3	E
C	6	B
D	0	•
E	1	D
F	4	B

Frontier $= \{C\}$

x $= F$

Complete example

skip

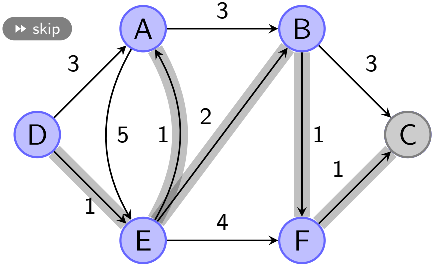


Node	Distance	Parent
A	2	E
B	3	E
C	6	B
D	0	•
E	1	D
F	4	B

Frontier = $\{C\}$

x = F

Complete example

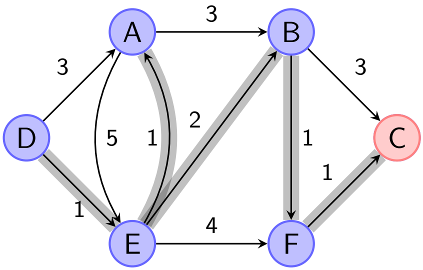


Node	Distance	Parent
A	2	E
B	3	E
C	4+1	F
D	0	•
E	1	D
F	4	B

Frontier = $\{C\}$

x = F

Complete example



Node	Distance	Parent
A	2	E
B	3	E
C	5	F
D	0	•
E	1	D
F	4	B

Frontier $= \{\}$

x $= C$

Reconstruction of the path

What is a path from s to t?

- The array "distance" stores the minimum cost from s to t;
- The array "parent" stores the predecessor of each visited node;

➜ How to construct the path from s to t using "parent"?

Reconstruction of the path

What is a path from s to t?

- The array "distance" stores the minimum cost from s to t;
- The array "parent" stores the predecessor of each visited node;

→ How to construct the path from s to t using "parent"?

Pseudo-code

```
def construct_path(parent,t):
    path = [t]
    current = t
    while not parent[current] is None:
        current = parent[current]
        path.insert(0,current)
    return path
```

Plan

A concrete problem

Graph problems...

- Find the shortest path in a graph (extract_min_dist)
- Build the minimum spanning tree (Lecture 3)

➜ Requires a priority queue!

Priority Queues

Storage of data along some priority order

A concrete problem

Graph problems. . .

- Find the shortest path in a graph (extract_min_dist)
- Build the minimum spanning tree (Lecture 3)

➜ Requires a priority queue!

Priority Queues

Storage of data along some priority order

Definition

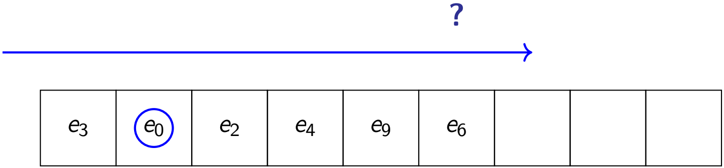
Abstract data structure specification with efficient operations on an ordered set for:

1. Finding the minimum (or maximum) element in the set;
2. Insert an element of given priority in the set;
3. Extract the element of smallest (or greatest) priority.

Implementation: array

Unsorted array of elements

- Find the smallest element ?



Implementation: array

Unsorted array of elements

- ✗ Find the smallest elements: $\mathcal{O}(n)$
- Insert an element ?

e_3	e_0	e_2	e_4	e_9	e_6			

Implementation: array

Unsorted array of elements

- ✗ Find the smallest elements: $\mathcal{O}(n)$
- ✓ Insert an element (at the end when there is a place): $\mathcal{O}(1)$

e_3	e_0	e_2	e_4	e_9	e_6	e_5		

Implementation: array

Unsorted array of elements

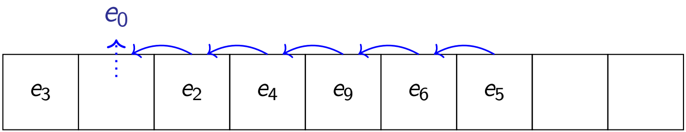
- ✗ Find the smallest elements: $\mathcal{O}(n)$
- ✓ Insert an element (at the end when there is a place): $\mathcal{O}(1)$
- Extract the smallest element **?**

e_3	e_0	e_2	e_4	e_9	e_6	e_5		

Implementation: array

Unsorted array of elements

- ✗ Find the smallest elements: $\mathcal{O}(n)$
- ✓ Insert an element (at the end when there is a place): $\mathcal{O}(1)$
- ✗ Extract the smallest element: $\mathcal{O}(n)$ (shift all elements)



Implementation: sorted array

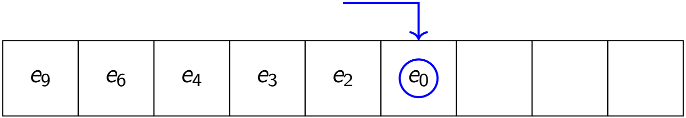
Sorted array of elements

e_9	e_6	e_4	e_3	e_2	e_0			

Implementation: sorted array

Sorted array of elements

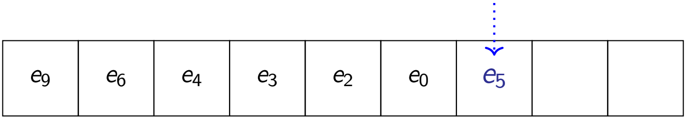
✓ Find the smallest element: $\mathcal{O}(1)$



Implementation: sorted array

Sorted array of elements

- ✓ Find the smallest element: $\mathcal{O}(1)$
- ✗ Insert an element: $\mathcal{O}(n)$ (shift all elements)



Insertion

Implementation: sorted array

Sorted array of elements

- ✓ Find the smallest element: $\mathcal{O}(1)$
- ✗ Insert an element: $\mathcal{O}(n)$ (shift all elements)

e_9	e_6	e_4	e_3	e_2	e_5	e_0		

Décalage

Implementation: sorted array

Sorted array of elements

- ✓ Find the smallest element: $\mathcal{O}(1)$
- ✗ Insert an element: $\mathcal{O}(n)$ (shift all elements)

e_9	e_6	e_4	e_3	e_5	e_2	e_0		

Décalage

Implementation: sorted array

Sorted array of elements

- ✓ Find the smallest element: $\mathcal{O}(1)$
- ✗ Insert an element: $\mathcal{O}(n)$ (shift all elements)

e_9	e_6	e_4	e_5	e_3	e_2	e_0		

Implementation: sorted array

Sorted array of elements

- ✓ Find the smallest element: $\mathcal{O}(1)$
- ✗ Insert an element: $\mathcal{O}(n)$ (shift all elements)

e_9	e_6	e_5	e_4	e_3	e_2	e_0		

Implementation: sorted array

Sorted array of elements

- ✓ Find the smallest element: $\mathcal{O}(1)$
- ✗ Insert an element: $\mathcal{O}(n)$ (shift all elements)

e_9	e_6	e_5	e_4	e_3	e_2	e_0		

Implementation: sorted array

Sorted array of elements

- ✓ Find the smallest element: $\mathcal{O}(1)$
- ✗ Insert an element: $\mathcal{O}(n)$ (shift all elements)
- ✓ Extract the smallest element: $\mathcal{O}(1)$ (if descending order)

e_0

e_9	e_6	e_5	e_4	e_3	e_2			

Implementation: heap

Definition

A heap is an abstract data structure used to manage priority lists in an efficient manner.

We need to:

- Access the maximum priority element as quickly as possible
- Find a performance tradeoff (between $\mathcal{O}(1)$ and $\mathcal{O}(n)$) for insertion and extraction

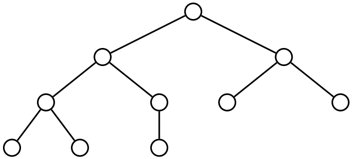
Implementation: heap

Definition

A heap is an abstract data structure used to manage priority lists in an efficient manner.

Tree

An undirected graph that is connected and acyclic is called a tree.



Implementation: heap

Definition

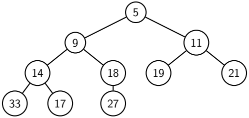
A heap is an abstract data structure used to manage priority lists in an efficient manner.

Principle

A tree whose vertices are the priority values, such that:

- min-heap : each node has a lower value than any of its children
- max-heap : each node has a greater value than any of its children

➜ In a min-heap (resp. max-heap), the root is the minimum (resp. maximum) value

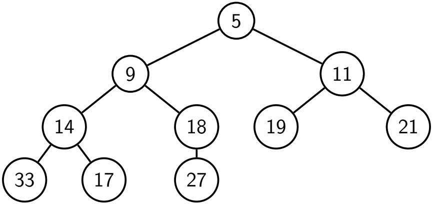


Binary heap

Binary heap

Quasi-complete binary tree:

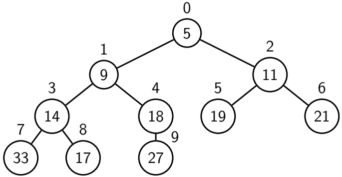
The binary tree is complete at all levels, except possibly the last one. If the last one is not complete, all available nodes are grouped onto the left most parents.



Implementation of a binary heap

Concretely

Un min-heap is an array with the following property:



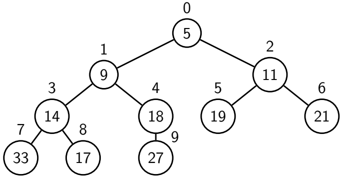
	0	1	2	3	4	5	6	7	8	9
tab:	5	9	11	14	18	19	21	33	17	27

root	:	node 0
left child of node i	:	node at $2i+1$
right child of node i	:	node at $2i+2$
parent of node i	:	node at ?
node at i is a leaf	:	?

Implementation of a binary heap

Concretely

Un min-heap is an array with the following property:



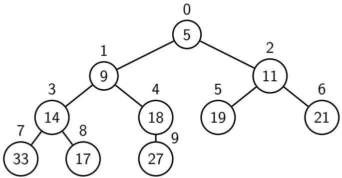
	0	1	2	3	4	5	6	7	8	9
tab:	5	9	11	14	18	19	21	33	17	27

root	:	node 0
left child of node i	:	node at $2i+1$
right child of node i	:	node at $2i+2$
parent of node i	:	node at $\lfloor (i-1)/2 \rfloor$
node at i is a leaf	:	?

Implementation of a binary heap

Concretely

Un min-heap is an array with the following property:



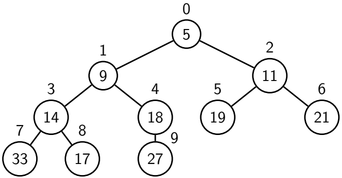
	0	1	2	3	4	5	6	7	8	9
tab:	5	9	11	14	18	19	21	33	17	27

root	:	node 0
left child of node i	:	node at $2i+1$
right child of node i	:	node at $2i+2$
parent of node i	:	node at $\lfloor (i-1)/2 \rfloor$
node at i is a leaf	:	$2i+1 \geq n$

Implementation of a binary heap

Concretely

Un min-heap is an array with the following property:



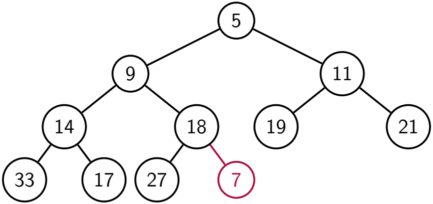
	0	1	2	3	4	5	6	7	8	9
tab:	5	9	11	14	18	19	21	33	17	27

$$\texttt{tab}[\lfloor (i-1)/2 \rfloor] < \texttt{tab[i]} \text{ pour tout } i \geq 1$$

How to insert in a min-heap

1. The new element v is inserted at the end of the last level of the tree
   - i.e. at the end of the array

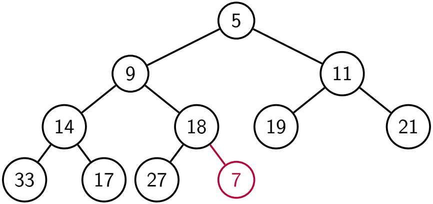
Example



How to insert in a min-heap

1. The new element v is inserted at the end of the last level of the tree
   - i.e. at the end of the array
2. While the key of v is smaller than the key of v parent:
   - Swap v and its parent

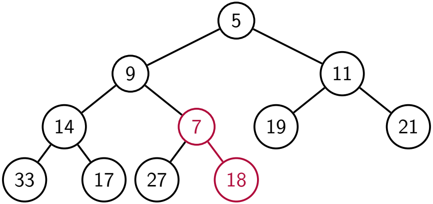
Example



How to insert in a min-heap

1. The new element v is inserted at the end of the last level of the tree
   - i.e. at the end of the array
2. While the key of v is smaller than the key of v parent:
   - Swap v and its parent

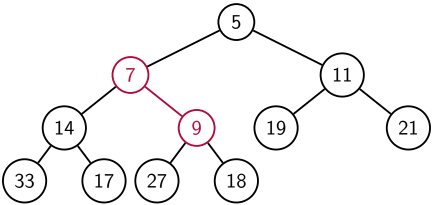
Example



How to insert in a min-heap

1. The new element v is inserted at the end of the last level of the tree
   - i.e. at the end of the array
2. While the key of v is smaller than the key of v parent:
   - Swap v and its parent

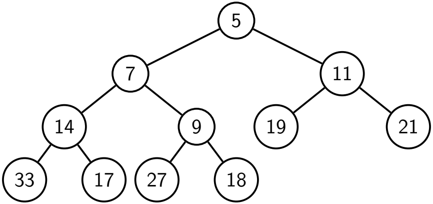
Example



How to insert in a min-heap

1. The new element v is inserted at the end of the last level of the tree
   - i.e. at the end of the array
2. While the key of v is smaller than the key of v parent:
   - Swap v and its parent

Example

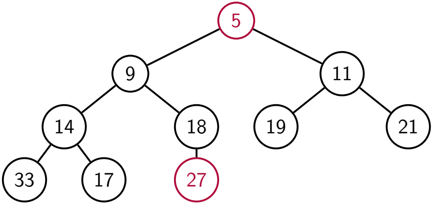


Extract the minimum of a min-heap

To remove the root element:

1. Replace the root element with the last element v in the tree
   - i.e. the last element in the array

Example

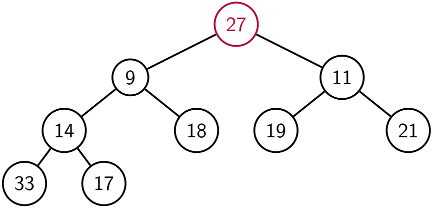


Extract the minimum of a min-heap

To remove the root element:

1. Replace the root element with the last element v in the tree
   - i.e. the last element in the array

Example

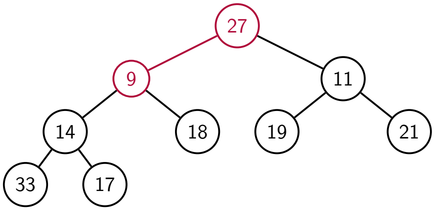


Extract the minimum of a min-heap

To remove the root element:

1. Replace the root element with the last element v in the tree
   - i.e. the last element in the array
2. While v is greater than one of its children:
   - We swap v with the smallest of its children

Example

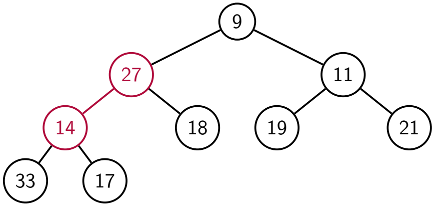


Extract the minimum of a min-heap

To remove the root element:

1. Replace the root element with the last element v in the tree
   - i.e. the last element in the array
2. While v is greater than one of its children:
   - We swap v with the smallest of its children

Example

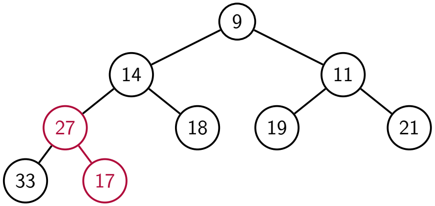


Extract the minimum of a min-heap

To remove the root element:

1. Replace the root element with the last element v in the tree
   - i.e. the last element in the array
2. While v is greater than one of its children:
   - We swap v with the smallest of its children

Example

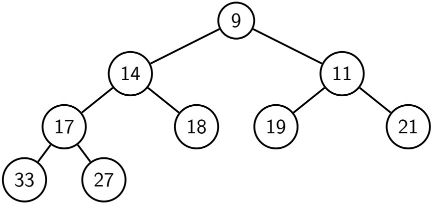


Extract the minimum of a min-heap

To remove the root element:

1. Replace the root element with the last element v in the tree
   - i.e. the last element in the array
2. While v is greater than one of its children:
   - We swap v with the smallest of its children

Example



Time complexity of heap operations

The height of a binary heap of n items is $\log_2 n$

Time complexity

Time complexity of heap operations

The height of a binary heap of n items is $\log_2 n$

Time complexity

- ✓ Find the smallest element: $\mathcal{O}(1)$

Time complexity of heap operations

The height of a binary heap of n items is $\log_2 n$

Time complexity

- ✓ Find the smallest element: $\mathcal{O}(1)$
- ✓ Insert an element: $\mathcal{O}(\log n)$

Time complexity of heap operations

The height of a binary heap of n items is $\log_2 n$

Time complexity

- ✓ Find the smallest element: $\mathcal{O}(1)$
- ✓ Insert an element: $\mathcal{O}(\log n)$
- ✓ Extract the smallest element: $\mathcal{O}(\log n)$

Time complexity of heap operations

The height of a binary heap of n items is $\log_2 n$

Time complexity

- ✓ Find the smallest element: $\mathcal{O}(1)$
- ✓ Insert an element: $\mathcal{O}(\log n)$
- ✓ Extract the smallest element: $\mathcal{O}(\log n)$
- ✓ Decrease an element key: $\mathcal{O}(\log n)$
  - we can move it up in the tree (much like insertion)
  - used by `update` in Dijkstra algorithm

Priority queue: complexity summary

Operation	*Array*	*Sorted Array*	*Binary Heap*
get min	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
insert (update)	$\mathcal{O}(1)$	$\mathcal{O}(n)$	$\mathcal{O}(\log n)$
extract min	$\mathcal{O}(n)$	$\mathcal{O}(1)$	$\mathcal{O}(\log n)$

Going from n to $log(n)$ is a real win...

For $n = 1000$, $log_2(n) \simeq 10$; for $n = 10^9$, $log_2(n) \simeq 30$!

Building a min-heap

Question

How to build a heap from an existing array?

Building a min-heap

Question

How to build a heap from an existing array?

Warning!

To build a min-heap from a non-ordered array, it is not optimal to simply apply the insertion method n times.
The total complexity can be better than $\mathcal{O}(n.log(n))$!

Building a min-heap

Question

How to build a heap from an existing array?

Warning!

To build a min-heap from a non-ordered array, it is not optimal to simply apply the insertion method n times.
The total complexity can be better than $\mathcal{O}(n.log(n))$!

Exercise

- **Indication**: start from the end of the array and go up...

➜ In total, the complexity is $\sum_{h=0}^{\lfloor \log n \rfloor} \frac{n}{2^{h+1}} \mathcal{O}(h) = \mathcal{O}(n)$!
(faster than sorting)

Plan

Reminder of the shortest path algorithm

```
def shortest_path(graph,s):
    frontier = [s]
    parent = {}
    parent[s] = None
    dist = {}
    dist[s] = 0

    while len(frontier)>0:
        x = extract_min_dist(frontier,dist)
        for y in neighbors(graph, x):
            if y not in parent:
                frontier.append(y)
            new_dist = dist[x] + distance(graph,x,y)
            if y not in dist or dist[y] > new_dist:
                dist[y] = new_dist
                parent[y] = x

    return parent
```

Have a look at the problem differently...

```
def shortest_path(graph,s):
    # Initialisation

    while len(frontier)>0:
        x = extract_min_dist(frontier,dist)
        for y in neighbors(graph, x):

            # updating frontier, parent and dist

    # Computing of the resulting path
```

Have a look at the problem differently. . .

```
def shortest_path(graph,s):
    # Initialisation
```
$\leftarrow \mathcal{O}(|1|)$

```
    while len(frontier)>0:
        x = extract_min_dist(frontier,dist)
        for y in neighbors(graph, x):

            # updating frontier, parent and dist

    # Computing of the resulting path
```

Have a look at the problem differently...

```
def shortest_path(graph,s):

    # Initialisation
```

$\leftarrow \mathcal{O}(|1|)$

```
    while len(frontier)>0:
```

this will be done $|V|$ times (one vertex withdrawn at each step)

```
        x = extract_min_dist(frontier,dist)

        for y in neighbors(graph, x):


        # updating frontier, parent and dist


    # Computing of the resulting path
```

Have a look at the problem differently...

```
def shortest_path(graph,s):

    # Initialisation                              ← O(|1|)


    while len(frontier)>0:
    this will be done |V| times (one vertex withdrawn at each step)
        x = extract_min_dist(frontier,dist)
                                                  complexity C_extract_min
        for y in neighbors(graph, x):


        # updating frontier, parent and dist


    # Computing of the resulting path
```

Have a look at the problem differently...

```
def shortest_path(graph,s):

    # Initialisation                                  ← O(|1|)


    while len(frontier)>0:
    this will be done |V| times (one vertex withdrawn at each step)
        x = extract_min_dist(frontier,dist)
                                                      complexity C_extract_min
        for y in neighbors(graph, x):
        It is more delicate ...we are only passing |E| times
        (updating takes place only when a new arc is visited)
        # updating frontier, parent and dist


    # Computing of the resulting path
```

Have a look at the problem differently...

```
def shortest_path(graph,s):

    # Initialisation                                ← O(|1|)


    while len(frontier)>0:
    this will be done |V| times (one vertex withdrawn at each step)
        x = extract_min_dist(frontier,dist)
                                                    complexity C_extract_min
        for y in neighbors(graph, x):
        It is more delicate ...we are only passing |E| times
        (updating takes place only when a new arc is visited)
        # updating frontier, parent and dist
                                                    complexity C_update

    # Computing of the resulting path
```

Have a look at the problem differently...

```
def shortest_path(graph,s):

    # Initialisation
```
$\leftarrow \mathcal{O}(|1|)$

```
    while len(frontier)>0:
```
this will be done $|V|$ times (one vertex withdrawn at each step)

```
        x = extract_min_dist(frontier,dist)
```
complexity $C_{extract\_min}$

```
        for y in neighbors(graph, x):
```
It is more delicate ...we are only passing $|E|$ times
(updating takes place only when a new arc is visited)

```
        # updating frontier, parent and dist
```
complexity C_{update}

```
    # Computing of the resulting path
```
$\leftarrow \mathcal{O}(|V|)$

In a nutshell...

Shortest path algorithm complexity

$$\mathcal{O}(\,1 + |V| \times C_{extract\_min} + |E| \times C_{update} + |V|\,)$$

In a nutshell. . .

Shortest path algorithm complexity

$$\mathcal{O}(\quad |V| \times C_{extract\_min} + |E| \times C_{update} \quad)$$

In a nutshell...

Shortest path algorithm complexity

$$\mathcal{O}(\quad |V| \times C_{extract\_min} + |E| \times C_{update} \quad)$$

Implementation

The values of $C_{extract\_min}$ and C_{update} depend on the implementation of the frontier.

In a nutshell...

Shortest path algorithm complexity

$$\mathcal{O}(\quad |V| \times C_{extract\_min} + |E| \times C_{update} \quad)$$

Implementation

The values of $C_{extract\_min}$ and C_{update} depend on the implementation of the frontier.

Implementation of the frontier: priority queue

- naive implementation with a simple list
- implementation with binary heap

Naive implementation

Complexity: $\mathcal{O}(|V| \times C_{extract\_min} + |E| \times C_{update})$

where:

- $C_{extract\_min} = \mathcal{O}(|frontier|)$
- $C_{update} = \mathcal{O}(1)$

The frontier contains at most $|V|$ elements.

Naive implementation

Complexity: $\mathcal{O}(|V| \times C_{extract\_min} + |E| \times C_{update})$

where:

- $C_{extract\_min} = \mathcal{O}(|frontier|)$
- $C_{update} = \mathcal{O}(1)$

The frontier contains at most $|V|$ elements.

Complexity: $\mathcal{O}(|V| \times |V| + |E|)$

where $|E|$ is between 0 and $|V|^2$ (dense graphs)

Naive implementation

Complexity: $\mathcal{O}(|V| \times C_{extract\_min} + |E| \times C_{update})$

where:

- $C_{extract\_min} = \mathcal{O}(|frontier|)$
- $C_{update} = \mathcal{O}(1)$

The frontier contains at most $|V|$ elements.

Complexity: $\mathcal{O}(|V| \times |V| + |E|)$

where $|E|$ is between 0 and $|V|^2$ (dense graphs)

In practice...

➜ Shortest path algorithm complexity is in $\mathcal{O}(|V|^2)$

Naive implementation

Complexity: $\mathcal{O}(|V| \times C_{extract\_min} + |E| \times C_{update})$

where:

- $C_{extract\_min} = \mathcal{O}(|frontier|)$
- $C_{update} = \mathcal{O}(1)$

The frontier contains at most $|V|$ elements.

Complexity: $\mathcal{O}(|V| \times |V| + |E|)$

where $|E|$ is between 0 and $|V|^2$ (dense graphs)

In practice...

➜ Shortest path algorithm complexity is in $\mathcal{O}(|V|^2)$

But for sparse graphs we may improve the complexity!

Implementation with binary heap

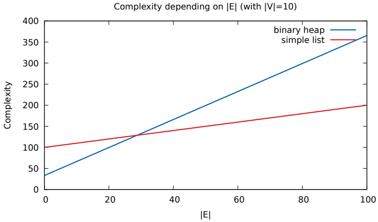
using a tree-based data structure (binary heap) we have:

- $C_{extract\_min} = \mathcal{O}(log(|V|))$
- $C_{update} = \mathcal{O}(log(|V|))$

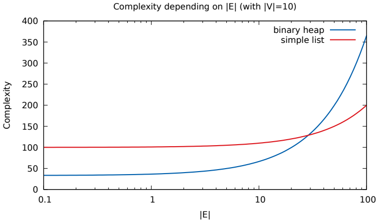
The complexity of the shortest path algorithm becomes:

$$\mathcal{O}((|V| + |E|) \times log(|V|))$$

Simple list vs Binary heap



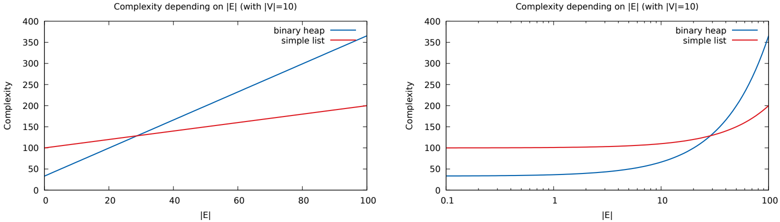
(a) Linear scale



(b) Logarithmic scale

Figure: Comparison of complexities depending on the graph density (function of $|E|$) for $|V| = 10$ fixed.

Simple list vs Binary heap



(a) Linear scale (b) Logarithmic scale

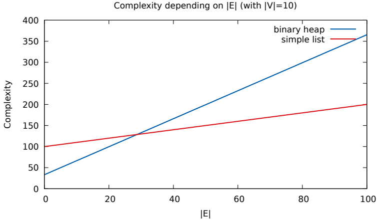
Figure: Comparison of complexities depending on the graph density (function of $|E|$) for $|V| = 10$ fixed.

In theory

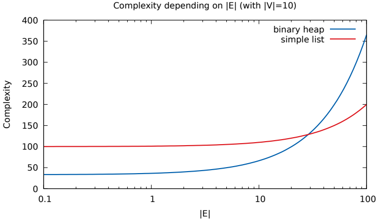
- When $|V|^2 + |E|$ is less advantageous than $(|V| + |E|) \times log(|V|)$?

It depends on the graph density : ($|E|$ in relation to $|V|$)

Simple list vs Binary heap



(a) Linear scale



(b) Logarithmic scale

Figure: Comparison of complexities depending on the graph density (function of $|E|$) for $|V| = 10$ fixed.

In practice

We will see in lab session that $\mathcal{O}(|E|)$ for the update is very overestimated. . . and that the binary heap is doing better than expected!

Plan

What you should remember

- Directed graph: $G = (V, E)$ with weight function $\omega : E \rightarrow \mathbb{R}$
- Shortest paths algorithm
  - Slightly modified BFS;
  - The shortest paths between s and all other vertices;
- Complexity depends on data structures chosen for implementation
  - Naive complexity (with a simple list) in $\mathcal{O}(|V|^2 + |E|)$
  - Complexity (with binary heap) in $\mathcal{O}((|V| + |E|) \times log(|V|))$

  ➜ The gain varies depending on instances.
- The correctness of the algorithm and the optimality of the result it produces are proven

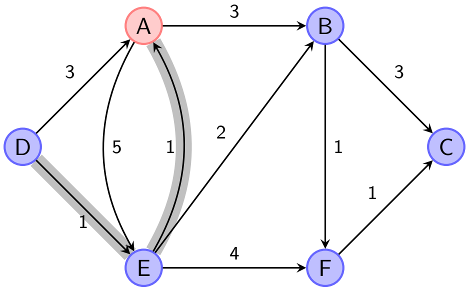
Plan

Optimality

Property

A vertex leaving the frontier has already its distance/path fixed.

Example: distance of E won't be updated when visiting A



Optimality

Property

A vertex leaving the frontier has already its distance/path fixed.
Example: distance of E won't be updated when visiting A

More formally...

Loop invariant is determined: Let S_n denote a set of vertices which have been already visited at step n. (by construction $|S_n| = n$)

1. $\forall x \in S_n$, $distance(x)$ is the length of the shortest path in G
2. $\forall x \notin S_n$, $distance(x)$ is the length of the shortest path in subgraph $S_n \cup \{x\}$

The property is a collorary of this invariant

Proof idea

Proof by recursion

The loop invariant is proven recursively over n

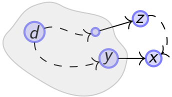
Proof idea

Proof by recursion

The loop invariant is proven recursively over n

What does mean the invariant?

- All vertices in S_n have their final value for $distance(x)$
- All the neighbors x of S_n (*i.e.* the frontier) have their minimum distance in $S_n \cup \{x\}$



$\rightarrow$ The path passing through z is longer (this part of the proof is more tricky; it works for non-negative weights only)

- The others are $+\infty$

Proof idea

Proof by recursion

The loop invariant is proven recursively over n

Invariant conclusion

Vertices in S_n (after having left the frontier) know their shortest path.

skip details

Optimality proof I

Proof by recurrence

The loop invariant is proven recursively over n

$n = 1$

For $S_1 = \{d\}$ and its neighbors have the arc weight for $distance(x)$.

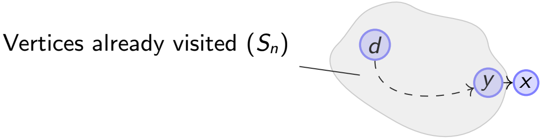
1. $distance(d) = 0$ is minimal (positive distances)
2. $\forall v$ a neighbor of d, $distance(v) = \omega((d, v))$ is minimal in the sub-graph $\{d, v\}$

Optimality proof II

General case

We suppose that the hypothesis holds at step n. Let x be the vertex chosen by the algorithm at step $n+1$:

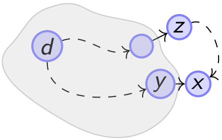
- It is a successor of S_n (otherwise $distance(x) = \infty$: it would not have been chosen)
- It has the smallest $distance(.)$
- Its predecessor in S_n is denoted by y



Optimality proof II

Proof *ad absurdum*

We now consider another path towards x and denote the first non visited vertex on this path as z:



This path is at least as long as the previous one.

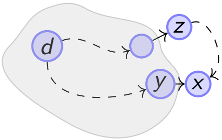
Optimality proof II

$distance(z)$ is minimal

$cost([d, \ldots, z]) \geq distance(z)$ because according to the recurrence hypothese 2, $distance(z)$ is minimal in $S_n \cup \{z\}$

$distance(x)$ is minimal

By definition, $distance(z) \geq distance(x)$ as x has been chosen, not z

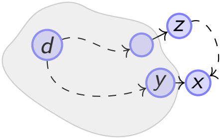


$$cost([d, \ldots, z]) \geq distance(x)$$

Optimality proof III

Total cost

We have $cost([d,\ldots,z,\ldots,x]) = cost([d,\ldots,z]) + cost([z,\ldots,x]$ by definition. By this way $cost([d,\ldots,z,\ldots,x]) \geq cost([d,\ldots,z]) \geq distance(x)$ in the case where $cost([z,\ldots,x] \geq 0$ (if this cost is negative, the proof will not work here!).



Any path outgoing S_n is at least as long as the one found.

Optimality proof IV

Proof by recurrence

If the hypothesis holds at step n, then the vertex x attached satisfies the property $distance(x) = dist(d, x)$.

Invariant:

- ✓ $\forall x \in S_{n+1}$, $distance(x)$ is minimal in G
- ➜ $\forall y \notin S_{n+1}$, $distance(y)$ is minimal in $S_{n+1} \cup \{y\}$

Second part

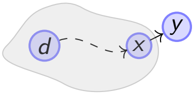
All successors y of x which are outside S_n should be considered.

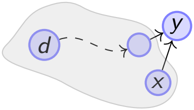
Optimality proof V

Second part (continuation)

We consider the shortest path to y in S_{n+1}:

- If it traverses x, vertex x is at the end (as x has already its shortest path in S_{n+1}).
  Therefore $distance(y) = distance(x) + \omega((x, y))$ is minimal
- Otherwise, $distance(y)$ has been already correctly fixed at the previous step. Thus $distance(y) \leq distance(x) + \omega((x, y))$ (otherwise it would be that the shortest path passes through x) and $distance(y)$ would not be modified; consequently the property remains satisfied in S_{n+1}





Optimality proof VI

Conclusion

If the hypothesis holds at step n, then it will also hold at step $n+1$
Invariant:

- ✓ $\forall x \in S_{n+1}$, $distance(x)$ is minimal in G
- ✓ $\forall y \notin S_{n+1}$, $distance(y)$ is minimal in $S_{n+1} \cup \{y\}$