Quicksort

Quicksort is another divide-and-conquer algorithm:

- divide into sub-problems;
- recursively solve the sub-problems;
- this time, we don't need to combine the solutions!

Quicksort

Quicksort is another divide-and-conquer algorithm:

- divide into sub-problems;
- recursively solve the sub-problems;
- this time, we don't need to combine the solutions!

Quicksort sorts a list by:

- picking a value called the **pivot**, p;
- rearranging the array so all values $\leq p$ come first;
- recursively sorting the values $\leq p$ and the values $> p$.

Quicksort: choosing a pivot

- If half the array is less than the pivot and half the array is greater, we split the array in half.
- In this case, the recursion depth is $\log n$, like mergesort.

Quicksort: choosing a pivot

- If half the array is less than the pivot and half the array is greater, we split the array in half.
- In this case, the recursion depth is $\log n$, like mergesort.
- So we want to use the array's **median** as pivot.

Quicksort: choosing a pivot

- If half the array is less than the pivot and half the array is greater, we split the array in half.
- In this case, the recursion depth is $\log n$, like mergesort.
- So we want to use the array's **median** as pivot.
- This is expensive to compute so we estimate it.
- e.g., mean of first and last values; median of first, last, middle; a random element.

Quicksort: choosing a pivot

- If half the array is less than the pivot and half the array is greater, we split the array in half.
- In this case, the recursion depth is $\log n$, like mergesort.
- So we want to use the array's **median** as pivot.
- This is expensive to compute so we estimate it.
- e.g., mean of first and last values; median of first, last, middle; a random element.
- We still get $O(n \log n)$ performance if the pivot is fairly close to the median.

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	10	8	15	4	13	5	7	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	10	8	15	4	13	5	7	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	10	8	15	4	13	5	7	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	10	8	15	4	13	5	7	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	10	8	15	4	13	5	7	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	10	8	15	4	13	5	7	14	12

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	7	8	15	4	13	5	10	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\mathrm{first} + \mathrm{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	7	8	15	4	13	5	10	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	7	8	15	4	13	5	10	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	7	5	15	4	13	8	10	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	7	5	15	4	13	8	10	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	7	5	15	4	13	8	10	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	7	5	15	4	13	8	10	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

3	7	5	4	15	13	8	10	14	12

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

$$\underbrace{\boxed{3}\,\boxed{7}\,\boxed{5}\,\boxed{4}}_{\leq \text{ pivot}}\,\underbrace{\boxed{15}\,\boxed{13}\,\boxed{8}\,\boxed{10}\,\boxed{14}\,\boxed{12}}_{> \text{ pivot}}$$

Quicksort example

To partition, we look for an element that's $> p$ to the left of an element that's $< p$ and swap them.

Use $(\text{first} + \text{last})/2 =$ **7.5 as pivot**. (Median is 9.)

$$\underbrace{\boxed{3}\,\boxed{7}\,\boxed{5}\,\boxed{4}}_{\leq \text{ pivot}}\;\underbrace{\boxed{15}\,\boxed{13}\,\boxed{8}\,\boxed{10}\,\boxed{14}\,\boxed{12}}_{> \text{ pivot}}$$

After recursion on the two regions, the whole array is sorted.

Quicksort in Java

Source: adapted from Wikipedia.

```
static void qSort (int[] ints, int left, int right) {
    if (right - left < 1) return;

    int i = left-1;
    int j = right+1;
    int pivot = (ints[left] + ints[right])/2;

    while (i<j) {
        do { i++; } while (ints[i] < pivot);
        do { j--; } while (ints[j] > pivot);
        if (i < j) swap (ints, i, j);
    }
    qSort (ints, left, j);
    qSort (ints, j+1, right);
}
```

Quicksort running time (1)

- Partitioning the array takes time $\Theta(n)$.
- If pivot is fairly close to median, depth of recursion is around $\log n$.
- Typical running time is $O(n \log n)$.

Quicksort running time (1)

- Partitioning the array takes time $\Theta(n)$.
- If pivot is fairly close to median, depth of recursion is around $\log n$.
- Typical running time is $O(n \log n)$.
- In the worst case, pivot is always largest or smallest element and we partition into arrays of length 1 and $n - 1$.
- Then, recursion depth is n and running time $O(n^2)$.

Quicksort running time (1)

- Suppose the split is never worse than 25%–75%.
- Recursion depth is the number of times n can be multiplied by $3/4$ before the answer is 1.

Quicksort running time (1)

- Suppose the split is never worse than 25%–75%.
- Recursion depth is the number of times n can be multiplied by $3/4$ before the answer is 1.
- This is the number of times it can be divided by $4/3$.

Quicksort running time (1)

- Suppose the split is never worse than 25%–75%.
- Recursion depth is the number of times n can be multiplied by $3/4$ before the answer is 1.
- This is the number of times it can be divided by $4/3$.
- So the depth is $\log_{4/3} n = \Theta(\log n)$.

Quicksort – summary

- Quicksort partitions an array based on a pivot, then recurses.

Quicksort – summary

- Quicksort partitions an array based on a pivot, then recurses.
- Typical running time is $O(n \log n)$.
- Very fast in practice.

Quicksort – summary

- Quicksort partitions an array based on a pivot, then recurses.
- Typical running time is $O(n \log n)$.
- Very fast in practice.
- Widely used (e.g., Java's `Arrays.sort()` is a variant of quicksort).
- Sort is in-place.
- Not well-suited to lists.

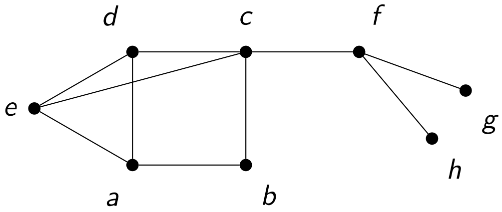
Quicksort – summary

- Quicksort partitions an array based on a pivot, then recurses.
- Typical running time is $O(n \log n)$.
- Very fast in practice.
- Widely used (e.g., Java's `Arrays.sort()` is a variant of quicksort).
- Sort is in-place.
- Not well-suited to lists.
- Health warning: do not try to implement quicksort if you can avoid it.

Graphs

Graphs

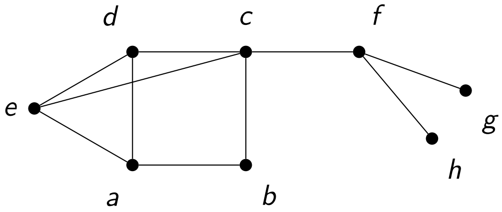
A **graph** is a collection of **vertices** (singular: vertex) and **edges**. Each edge links two vertices.



We write, e.g., (a, b) for the edge between vertices a and b

Graphs

A **graph** is a collection of **vertices** (singular: vertex) and **edges**. Each edge links two vertices.



We write, e.g., (a, b) for the edge between vertices a and b

Alternative terminology: network=graph, node=vertex, arc=edge.

Examples of graphs

- transport networks (vertices are intersections, edges are roads),
- computer networks (computers and connections),

Examples of graphs

- transport networks (vertices are intersections, edges are roads),
- computer networks (computers and connections),
- molecular structures (atoms and bonds),

Examples of graphs

- transport networks (vertices are intersections, edges are roads),
- computer networks (computers and connections),
- molecular structures (atoms and bonds),
- general binary relations (e.g. for a dating site, vertices are people and an edge indicates compatibility)

Paths

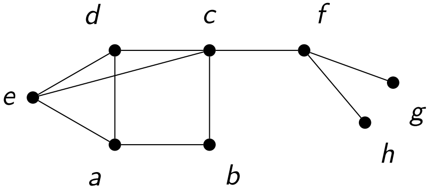
A **path** between vertices x and y is a sequence of edges

$$(x, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, y),$$

such that all the vertices $x, v_1, \ldots, v_k, y$ are distinct (i.e., no repetitions).

We often just list the vertices in order, e.g., $xv_1v_2 \ldots v_{k-1}v_ky$.

The **length** of a path is the number of edges (not vertices).



Paths

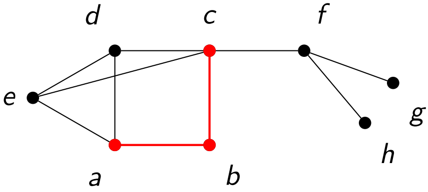
A **path** between vertices x and y is a sequence of edges

$$(x, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, y),$$

such that all the vertices $x, v_1, \ldots, v_k, y$ are distinct (i.e., no repetitions).

We often just list the vertices in order, e.g., $xv_1v_2 \ldots v_{k-1}v_ky$.

The **length** of a path is the number of edges (not vertices).



Examples of a–c paths: abc, adc, $aedc$.

Paths

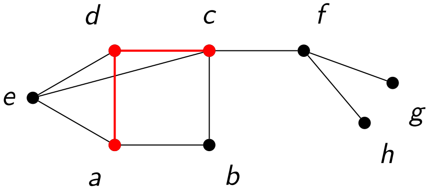
A **path** between vertices x and y is a sequence of edges

$$(x, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, y)\,,$$

such that all the vertices $x, v_1, \ldots, v_k, y$ are distinct (i.e., no repetitions).

We often just list the vertices in order, e.g., $xv_1v_2 \ldots v_{k-1}v_ky$.

The **length** of a path is the number of edges (not vertices).



Examples of a–c paths: abc, adc, $aedc$.

Paths

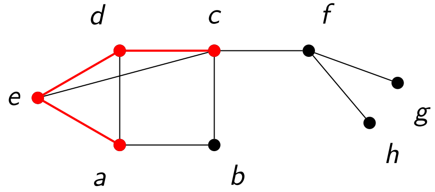
A **path** between vertices x and y is a sequence of edges

$$(x, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k), (v_k, y)\,,$$

such that all the vertices $x, v_1, \ldots, v_k, y$ are distinct (i.e., no repetitions).

We often just list the vertices in order, e.g., $xv_1v_2 \ldots v_{k-1}v_ky$.

The **length** of a path is the number of edges (not vertices).



Examples of a–c paths: abc, adc, $aedc$.

Cycles

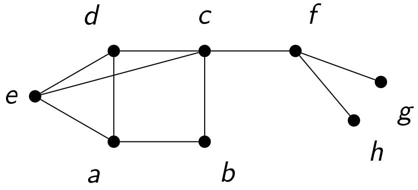
A **cycle** is a sequence of edges

$$(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1)\,,$$

such that all the vertices $v_1, \ldots, v_k$ are distinct (i.e., no repetitions).

We often just list the vertices in order, e.g., $v_1 v_2 \ldots v_{k-1} v_k v_1$.

The **length** of a cycle is the number of edges (or vertices).



Cycles

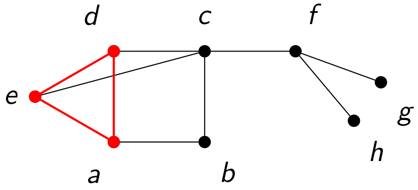
A **cycle** is a sequence of edges

$$(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1)\,,$$

such that all the vertices $v_1, \ldots, v_k$ are distinct (i.e., no repetitions).

We often just list the vertices in order, e.g., $v_1 v_2 \ldots v_{k-1} v_k v_1$.

The **length** of a cycle is the number of edges (or vertices).



Examples of cycles: *adea*, *aecba*, *aecda*.

Cycles

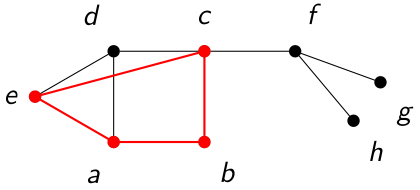
A **cycle** is a sequence of edges

$$(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1)\,,$$

such that all the vertices $v_1, \ldots, v_k$ are distinct (i.e., no repetitions).

We often just list the vertices in order, e.g., $v_1 v_2 \ldots v_{k-1} v_k v_1$.

The **length** of a cycle is the number of edges (or vertices).



Examples of cycles: *adea*, *aecba*, *aecda*.

Cycles

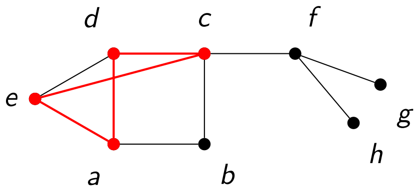
A **cycle** is a sequence of edges

$$(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k), (v_k, v_1)\,,$$

such that all the vertices $v_1, \ldots, v_k$ are distinct (i.e., no repetitions).

We often just list the vertices in order, e.g., $v_1 v_2 \ldots v_{k-1} v_k v_1$.

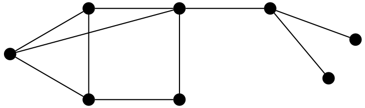
The **length** of a cycle is the number of edges (or vertices).



Examples of cycles: *adea*, *aecba*, *aecda*.

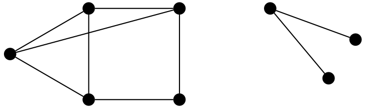
Connected graphs

A graph is **connected** if, for all distinct vertices x and y, there is a path from x to y.



Connected

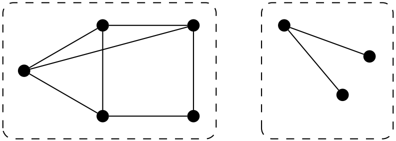
A graph is **connected** if, for all distinct vertices x and y, there is a path from x to y.



Not connected

Connected graphs

A graph is **connected** if, for all distinct vertices x and y, there is a path from x to y.



Components

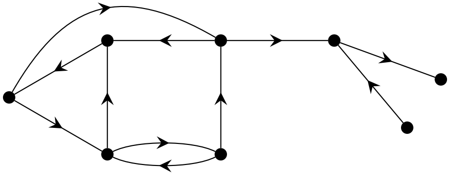
Directed graphs (1)

The graphs we've seen so far as **undirected**: the edge (x, y) is the same as (y, x).

Directed graphs (1)

The graphs we've seen so far as **undirected**: the edge (x, y) is the same as (y, x).

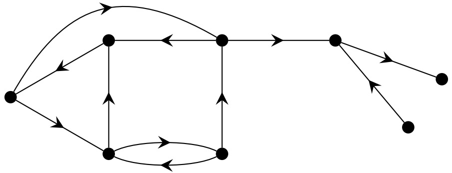
In **directed graphs** (aka **digraphs**), edges have a direction so (x, y) is different from (y, x).



Directed graphs (1)

The graphs we've seen so far as **undirected**: the edge (x, y) is the same as (y, x).

In **directed graphs** (aka **digraphs**), edges have a direction so (x, y) is different from (y, x).



Directed graphs (2)

Directed paths and **directed cycles** must follow the direction of the edges.

Directed graphs (2)

Directed paths and **directed cycles** must follow the direction of the edges.

A digraph is **strongly connected** if, for all distinct vertices x and y, there is a directed path from x to y.

Directed graphs (2)

Directed paths and **directed cycles** must follow the direction of the edges.

A digraph is **strongly connected** if, for all distinct vertices x and y, there is a directed path from x to y.

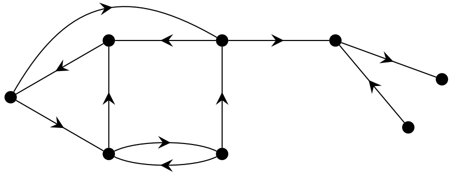
A digraph is **weakly connected** if it's connected when you ignore the edge directions.

Directed graphs (2)

Directed paths and **directed cycles** must follow the direction of the edges.

A digraph is **strongly connected** if, for all distinct vertices x and y, there is a directed path from x to y.

A digraph is **weakly connected** if it's connected when you ignore the edge directions.



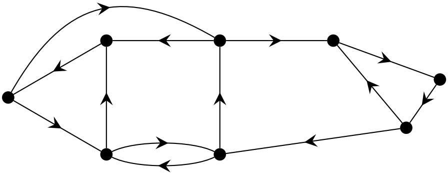
Weakly connected, not strongly connected

Directed graphs (2)

Directed paths and **directed cycles** must follow the direction of the edges.

A digraph is **strongly connected** if, for all distinct vertices x and y, there is a directed path from x to y.

A digraph is **weakly connected** if it's connected when you ignore the edge directions.



Weakly and strongly connected

Neighbours

Let x be a vertex of a graph.

- In an undirected graph, x's **neighbours** are the vertices y such that (x, y) is an edge.

Neighbours

Let x be a vertex of a graph.

- In an undirected graph, x's **neighbours** are the vertices y such that (x, y) is an edge.
- in a directed graph, x's **out-neighbours** are the vertices y such that (x, y) is an edge.

Neighbours

Let x be a vertex of a graph.

- In an undirected graph, x's **neighbours** are the vertices y such that (x, y) is an edge.
- in a directed graph, x's **out-neighbours** are the vertices y such that (x, y) is an edge.
- in a directed graph, x's **in-neighbours** are the vertices y such that (y, x) is an edge.

Weighted graphs

A **weighted graph** is a graph in which each edge has a numerical value assigned: its weight. Weights are usually positive.

We write (x, y, w) for an edge from x to y with weight w.

Weighted graphs

A **weighted graph** is a graph in which each edge has a numerical value assigned: its weight. Weights are usually positive.

We write (x, y, w) for an edge from x to y with weight w.

Weights can represent, e.g.,

- distance in transport networks,
- latency in communication networks,
- cost of traversing an edge,
- degree of compatibility, etc.

Weighted graphs

A **weighted graph** is a graph in which each edge has a numerical value assigned: its weight. Weights are usually positive.

We write (x, y, w) for an edge from x to y with weight w.

Weights can represent, e.g.,

- distance in transport networks,
- latency in communication networks,
- cost of traversing an edge,
- degree of compatibility, etc.

In a weighted directed graph, (x, y) and (y, x) can have different weights.

Implementing graphs (1)

There are two main representations of graphs. In both cases, we assume the vertices are labelled with the integers $0, \ldots, n-1$.

Implementing graphs (1)

There are two main representations of graphs. In both cases, we assume the vertices are labelled with the integers $0, \ldots, n-1$.

Adjacency matrix

- 2D array indicating which vertex pairs are edges.
- `booleans` for unweighted graphs, `int`/`double` for weighted.

Implementing graphs (1)

There are two main representations of graphs. In both cases, we assume the vertices are labelled with the integers $0, \ldots, n-1$.

Adjacency matrix

- 2D array indicating which vertex pairs are edges.
- `booleans` for unweighted graphs, `int`/`double` for weighted.
- Good for answering "Does edge (x, y) exist?"
- Memory-efficient for dense graphs (many edges).

Implementing graphs (2)

Adjacency list

- Each vertex has a list of its neighbours.
- Graph is represented as an array of these lists.

Implementing graphs (2)

Adjacency list

- Each vertex has a list of its neighbours.
- Graph is represented as an array of these lists.
- List out-neighbours for directed graphs.
- List includes weights in weighted graphs.

Adjacency list

- Each vertex has a list of its neighbours.
- Graph is represented as an array of these lists.
- List out-neighbours for directed graphs.
- List includes weights in weighted graphs.
- Good for answering "What are x's neighbours?"
- Memory-efficient for sparse graphs (few edges).

Implementing graphs (2)

Adjacency list

- Each vertex has a list of its neighbours.
- Graph is represented as an array of these lists.
- List out-neighbours for directed graphs.
- List includes weights in weighted graphs.
- Good for answering "What are x's neighbours?"
- Memory-efficient for sparse graphs (few edges).

Both implementations are useful: you must pick the one that best suits your situation.

Graph algorithms

Dijkstra's algorithm

Dijkstra's algorithm finds the shortest paths from a given vertex to every other vertex in the graph.

"Shortest" means fewest edges (unweighted graphs) or least total weight (weighted graphs).

Dijkstra's algorithm

Dijkstra's algorithm finds the shortest paths from a given vertex to every other vertex in the graph.

"Shortest" means fewest edges (unweighted graphs) or least total weight (weighted graphs).

We'll assume weighted graphs; represent unweighted graphs by giving every edge weight 1.

We'll assume all weights are positive.

Dijkstra's algorithm: the key idea (1)

Dijkstra finds paths in order of their length.

Suppose we start at vertex s.

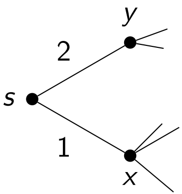
If (s, x, w) is the least-weight edge from s, then sx must be the shortest path to x.

Dijkstra's algorithm: the key idea (1)

Dijkstra finds paths in order of their length.

Suppose we start at vertex s.

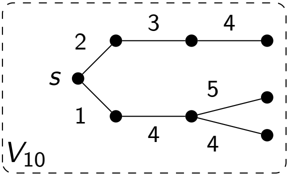
If (s, x, w) is the least-weight edge from s, then sx must be the shortest path to x.



Any path from s to x (or anywhere else) via y must have length at least 2.

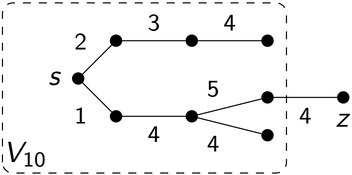
Dijkstra's algorithm: the key idea (2)

Let V_{10} be the set of vertices with a path of length less than 10 from s and suppose we've found the shortest path to every vertex in V_{10}.



Dijkstra's algorithm: the key idea (2)

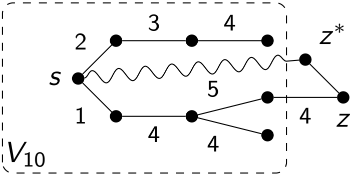
Let V_{10} be the set of vertices with a path of length less than 10 from s and suppose we've found the shortest path to every vertex in V_{10}.



Let $z \notin V_{10}$ be the vertex that can be reached by the shortest path of the form "Take the shortest path to some vertex in V_{10}, then go along one more edge."

Dijkstra's algorithm: the key idea (2)

Let V_{10} be the set of vertices with a path of length less than 10 from s and suppose we've found the shortest path to every vertex in V_{10}.



Let $z \notin V_{10}$ be the vertex that can be reached by the shortest path of the form "Take the shortest path to some vertex in V_{10}, then go along one more edge."

This must be the shortest path to z. If there was a shorter path, it must contain some other vertex $z^* \notin V_{10}$, which is closer to s than z is. Contradiction – we would have chosen z^* instead of z!

Dijkstra's algorithm: data structures

Dijkstra uses the following data structures, where each array has length n.

- An array `boolean[] solved`: stores whether we've found the shortest path to each vertex.
- An array `double[] distance`: the length of shortest path found to each vertex so far.
- A priority queue `Q`: unsolved vertices x with priority equal to `distance[x]`.

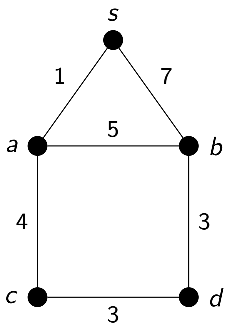
Dijkstra's algorithm: pseudocode (1)

```
// Initialization
solved = {false, ..., false};
distance = {INFINITY, ..., INFINITY};
Q = new empty queue;

solved[source] = true;
distance[source] = 0;

Q.insert (0, x);

...
```

Dijkstra's algorithm: pseudocode (2)

```
// Main loop
while (!Q.isEmpty() {
    x = Q.next();
    solved[x] = true;

    foreach neighbour y of x
        if (!solved[y]) {
            double newDistance = distance[x] + weight(x,y);
            if (newDistance < distance[y]) {
                distance[y] = newDistance;
                if (Q.contains (y))
                    Q.setPriority (newDistance, y);
                else
                    Q.insert (newDistance, y);
            }
        }
}
```

Dijkstra's algorithm: example

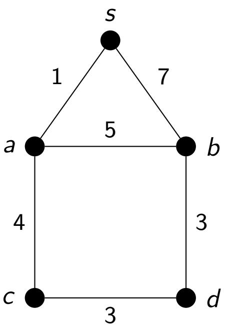


Queue:
dist:
Solved:

x =
y =
newDist =

```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Dijkstra's algorithm: example

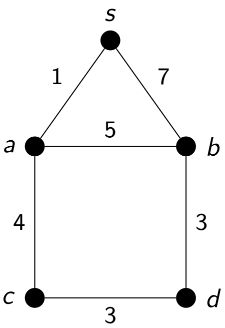



```
▷  initialize
   while (queue not empty)
       x = next();
       solved[x] = true;
       for each unsolved neighbour y of x
           newDist = dist[x] + weight(x,y);
           if (newDist < dist[y])
               dist[y] = newDist;
               update queue;
```

Queue: $s : 0$
dist: $s : 0$, $a : \infty$, $b : \infty$, $c : \infty$, $d : \infty$
Solved: none

x =
y =
newDist =

Dijkstra's algorithm: example




```
    initialize
▷   while (queue not empty)
        x = next();
        solved[x] = true;
        for each unsolved neighbour y of x
            newDist = dist[x] + weight(x,y);
            if (newDist < dist[y])
                dist[y] = newDist;
                update queue;
```

Queue: $s : 0$

dist: $s : 0$, $a : \infty$, $b : \infty$, $c : \infty$, $d : \infty$

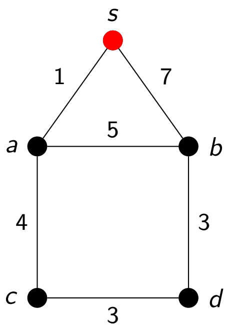
Solved: none

x =

y =

newDist =

Dijkstra's algorithm: example

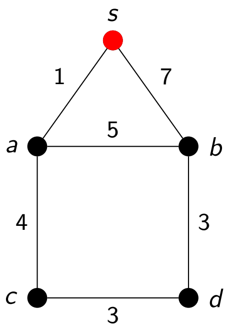



```
    initialize
    while (queue not empty)
▷       x = next();
        solved[x] = true;
        for each unsolved neighbour y of x
            newDist = dist[x] + weight(x,y);
            if (newDist < dist[y])
                dist[y] = newDist;
                update queue;
```

Queue: empty
dist: $s:0$, $a:\infty$, $b:\infty$, $c:\infty$, $d:\infty$
Solved: none

x = s
y =
newDist =

Dijkstra's algorithm: example

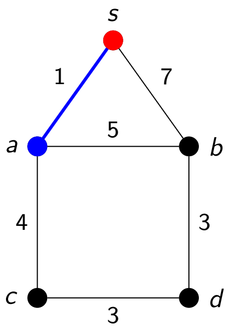



```
initialize
while (queue not empty)
      x = next();
      solved[x] = true;
      for each unsolved neighbour y of x
           newDist = dist[x] + weight(x,y);
           if (newDist < dist[y])
                dist[y] = newDist;
                update queue;
```

Queue: empty
dist: $s : 0,\ a : \infty,\ b : \infty,\ c : \infty,\ d : \infty$
Solved: s

x = s
y =
newDist =

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: empty

dist: $s : 0$, $a : \infty$, $b : \infty$, $c : \infty$, $d : \infty$

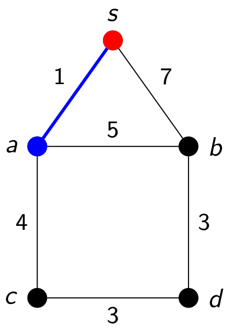
Solved: s

x = s

y = a

newDist =

Dijkstra's algorithm: example

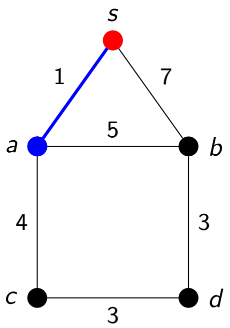



```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: empty
dist: $s : 0,\ a : \infty,\ b : \infty,\ c : \infty,\ d : \infty$
Solved: s

x = s
y = a
newDist = $0 + 1 = 1$

Dijkstra's algorithm: example

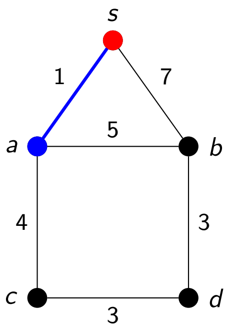



```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
▷        if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: empty
dist: $s:0,\ a:\infty,\ b:\infty,\ c:\infty,\ d:\infty$
Solved: s

x = s
y = a
newDist = $0+1=1$

Dijkstra's algorithm: example

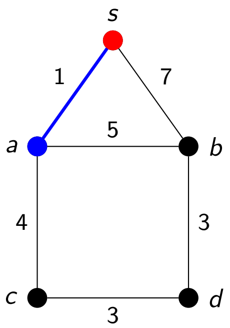



```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
▷           dist[y] = newDist;
            update queue;
```

Queue: empty
dist: $s:0,\ a:1,\ b:\infty,\ c:\infty,\ d:\infty$
Solved: s

x $=$ s
y $=$ a
newDist $=$ $0+1=1$

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $a : 1$

dist: $s : 0,\ a : 1,\ b : \infty,\ c : \infty,\ d : \infty$

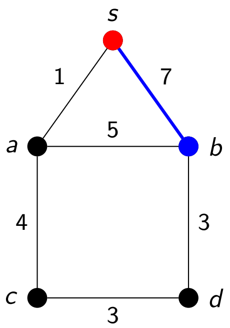
Solved: s

x = s

y = a

newDist = $0 + 1 = 1$

Dijkstra's algorithm: example

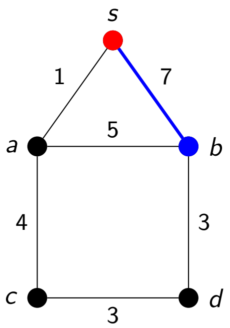



```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
▷    for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: $a : 1$
dist: $s : 0$, $a : 1$, $b : \infty$, $c : \infty$, $d : \infty$
Solved: s

x $=$ s
y $=$ b
newDist $=$

Dijkstra's algorithm: example

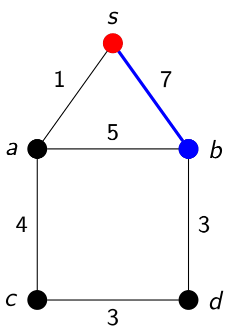



```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: $a : 1$
dist: $s : 0,\ a : 1,\ b : \infty,\ c : \infty,\ d : \infty$
Solved: s

x $=$ s
y $=$ b
newDist $=$ $0 + 7 = 7$

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
▷       if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $a : 1$

dist: $s : 0,\ a : 1,\ b : \infty,\ c : \infty,\ d : \infty$

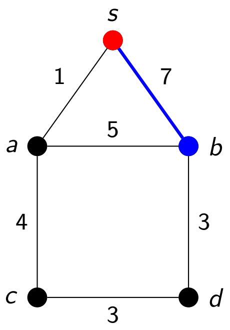
Solved: s

x = s

y = b

newDist = $0 + 7 = 7$

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: $a : 1$

dist: $s : 0,\ a : 1,\ b : 7,\ c : \infty,\ d : \infty$

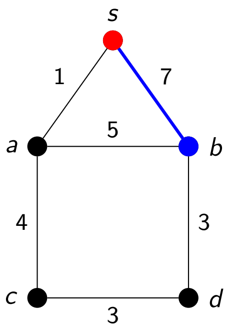
Solved: s

$\mathtt{x} =\ s$

$\mathtt{y} =\ b$

$\mathtt{newDist} =\ 0 + 7 = 7$

Dijkstra's algorithm: example

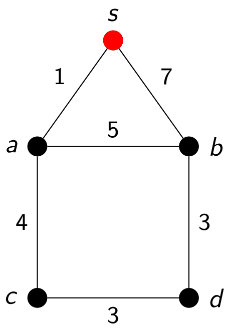



```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: $a : 1,\ b : 7$
dist: $s : 0,\ a : 1,\ b : 7,\ c : \infty,\ d : \infty$
Solved: s

x = s
y = b
newDist = $0 + 7 = 7$

Dijkstra's algorithm: example




```
    initialize
▷   while (queue not empty)
        x = next();
        solved[x] = true;
        for each unsolved neighbour y of x
            newDist = dist[x] + weight(x,y);
            if (newDist < dist[y])
                dist[y] = newDist;
                update queue;
```

Queue: $a : 1,\ b : 7$

dist: $s : 0,\ a : 1,\ b : 7,\ c : \infty,\ d : \infty$

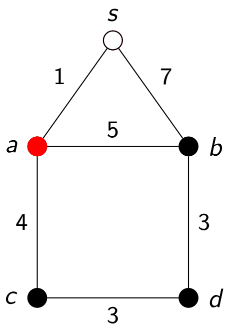
Solved: s

$\texttt{x} = s$

$\texttt{y} = b$

$\texttt{newDist} = 0 + 7 = 7$

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
▷     x = next();
      solved[x] = true;
      for each unsolved neighbour y of x
          newDist = dist[x] + weight(x,y);
          if (newDist < dist[y])
              dist[y] = newDist;
              update queue;
```

Queue: $b : 7$

dist: $s : 0,\ a : 1,\ b : 7,\ c : \infty,\ d : \infty$

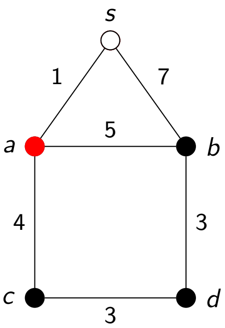
Solved: s

x = a

y =

newDist =

Dijkstra's algorithm: example




```
  initialize
  while (queue not empty)
      x = next();
▷     solved[x] = true;
      for each unsolved neighbour y of x
          newDist = dist[x] + weight(x,y);
          if (newDist < dist[y])
              dist[y] = newDist;
              update queue;
```

Queue: $b : 7$

dist: $s : 0,\ a : 1,\ b : 7,\ c : \infty,\ d : \infty$

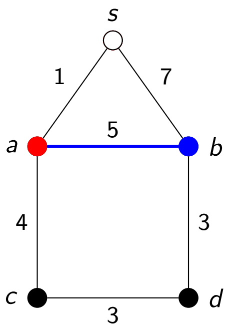
Solved: $s,\ a$

x = a

y =

newDist =

Dijkstra's algorithm: example

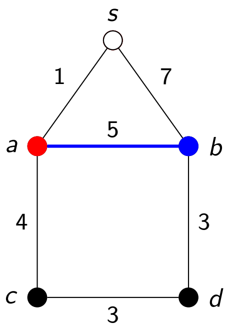



```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: $b : 7$
dist: $s : 0,\ a : 1,\ b : 7,\ c : \infty,\ d : \infty$
Solved: $s,\ a$

x = a
y = b
newDist =

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: $b : 7$

dist: $s : 0$, $a : 1$, $b : 7$, $c : \infty$, $d : \infty$

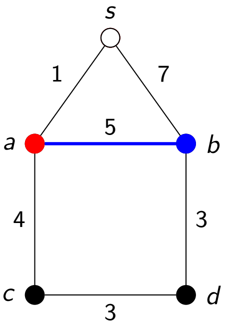
Solved: s, a

x = a

y = b

newDist = $1 + 5 = 6$

Dijkstra's algorithm: example

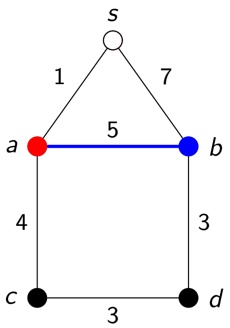



```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
▷       if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $b : 7$
dist: $s : 0$, $a : 1$, $b : 7$, $c : \infty$, $d : \infty$
Solved: s, a

x = a
y = b
newDist = $1 + 5 = 6$

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $b : 7$

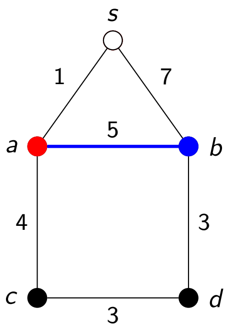
dist: $s : 0,\ a : 1,\ b : 6,\ c : \infty,\ d : \infty$

Solved: $s,\ a$

x $=$ a

y $=$ b

newDist $=$ $1 + 5 = 6$




```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $b : 6$

dist: $s : 0,\ a : 1,\ b : 6,\ c : \infty,\ d : \infty$

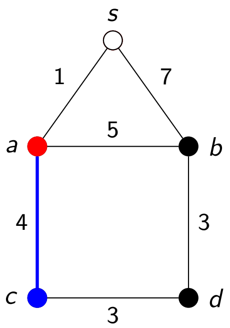
Solved: $s,\ a$

x = a

y = b

newDist = $1 + 5 = 6$

Dijkstra's algorithm: example

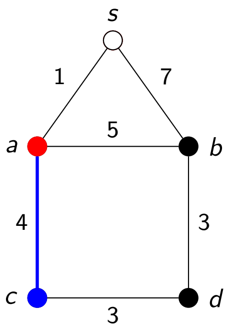



```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
▷    for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: $b : 6$
dist: $s : 0,\ a : 1,\ b : 6,\ c : \infty,\ d : \infty$
Solved: $s,\ a$

x $=$ a
y $=$ c
newDist $=$

Dijkstra's algorithm: example

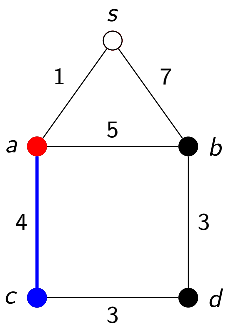



```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: $b : 6$
dist: $s : 0,\ a : 1,\ b : 6,\ c : \infty,\ d : \infty$
Solved: $s,\ a$

$\mathtt{x} =\ a$
$\mathtt{y} =\ c$
$\mathtt{newDist} =\ 1 + 4 = 5$

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $b : 6$

dist: $s : 0,\ a : 1,\ b : 6,\ c : \infty,\ d : \infty$

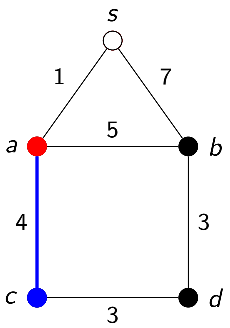
Solved: $s,\ a$

$\mathtt{x} =\ a$

$\mathtt{y} =\ c$

$\mathtt{newDist} =\ 1 + 4 = 5$

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: $b : 6$

dist: $s : 0$, $a : 1$, $b : 6$, $c : 5$, $d : \infty$

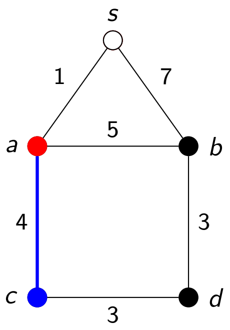
Solved: s, a

x = a

y = c

newDist = $1 + 4 = 5$

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: $c:5,\ b:6$

dist: $s:0,\ a:1,\ b:6,\ c:5,\ d:\infty$

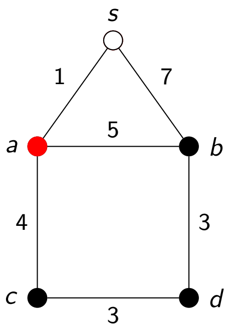
Solved: $s,\ a$

x = a

y = c

newDist = $1+4=5$

Dijkstra's algorithm: example




```
    initialize
▷   while (queue not empty)
        x = next();
        solved[x] = true;
        for each unsolved neighbour y of x
            newDist = dist[x] + weight(x,y);
            if (newDist < dist[y])
                dist[y] = newDist;
                update queue;
```

Queue: $c : 5,\ b : 6$

dist: $s : 0,\ a : 1,\ b : 6,\ c : 5,\ d : \infty$

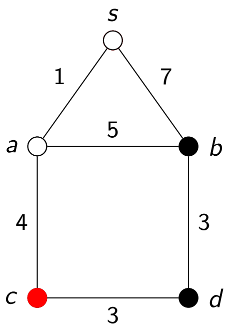
Solved: $s,\ a$

x = a

y =

newDist =

Dijkstra's algorithm: example




```
  initialize
  while (queue not empty)
▷     x = next();
      solved[x] = true;
      for each unsolved neighbour y of x
          newDist = dist[x] + weight(x,y);
          if (newDist < dist[y])
              dist[y] = newDist;
              update queue;
```

Queue: $b : 6$

dist: $s : 0,\ a : 1,\ b : 6,\ c : 5,\ d : \infty$

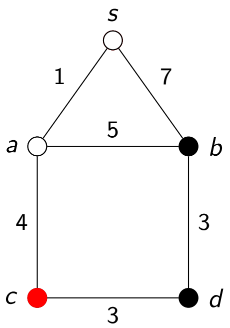
Solved: $s,\ a$

x = c

y =

newDist =

Dijkstra's algorithm: example

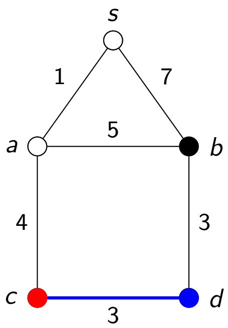



```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $b : 6$
dist: $s : 0,\ a : 1,\ b : 6,\ c : 5,\ d : \infty$
Solved: $s,\ a,\ c$

x = c
y =
newDist =

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: $b : 6$

dist: $s : 0,\ a : 1,\ b : 6,\ c : 5,\ d : \infty$

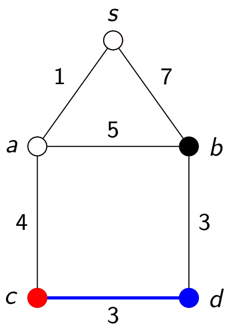
Solved: $s,\ a,\ c$

x = c

y = d

newDist =

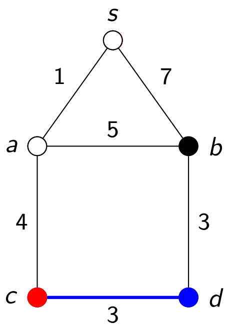
Dijkstra's algorithm: example




```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
▷        newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: $b : 6$
dist: $s : 0,\ a : 1,\ b : 6,\ c : 5,\ d : \infty$
Solved: $s,\ a,\ c$

$\mathtt{x} =\ c$
$\mathtt{y} =\ d$
$\mathtt{newDist} =\ 5 + 3 = 8$




```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $b : 6$

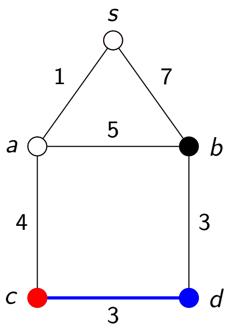
dist: $s : 0,\ a : 1,\ b : 6,\ c : 5,\ d : \infty$

Solved: $s,\ a,\ c$

x = c

y = d

newDist = $5 + 3 = 8$




```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
▷           dist[y] = newDist;
            update queue;
```

Queue: $b : 6$

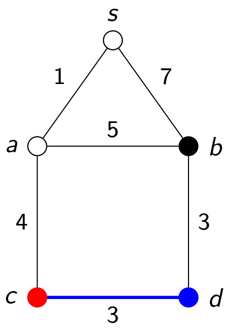
dist: $s : 0$, $a : 1$, $b : 6$, $c : 5$, $d : 8$

Solved: s, a, c

x $=$ c

y $=$ d

newDist $=$ $5 + 3 = 8$




```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $b:6,\ d:8$

dist: $s:0,\ a:1,\ b:6,\ c:5,\ d:8$

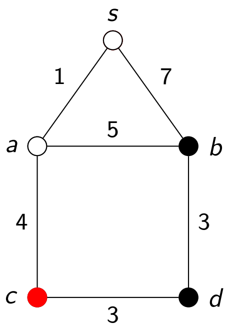
Solved: $s,\ a,\ c$

$\mathtt{x} =\ c$

$\mathtt{y} =\ d$

$\mathtt{newDist} =\ 5+3=8$

Dijkstra's algorithm: example




```
    initialize
▷   while (queue not empty)
        x = next();
        solved[x] = true;
        for each unsolved neighbour y of x
            newDist = dist[x] + weight(x,y);
            if (newDist < dist[y])
                dist[y] = newDist;
                update queue;
```

Queue: $b : 6,\ d : 8$

dist: $s : 0,\ a : 1,\ b : 6,\ c : 5,\ d : 8$

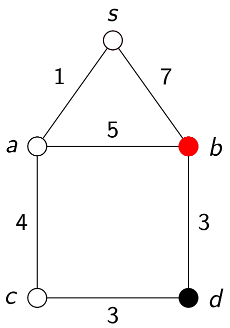
Solved: $s,\ a,\ c$

x = c

y =

newDist =

Dijkstra's algorithm: example

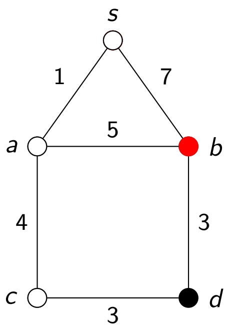



```
  initialize
  while (queue not empty)
▷     x = next();
      solved[x] = true;
      for each unsolved neighbour y of x
          newDist = dist[x] + weight(x,y);
          if (newDist < dist[y])
              dist[y] = newDist;
              update queue;
```

Queue: $d : 8$
dist: $s : 0$, $a : 1$, $b : 6$, $c : 5$, $d : 8$
Solved: s, a, c

x = b
y =
newDist =

Dijkstra's algorithm: example

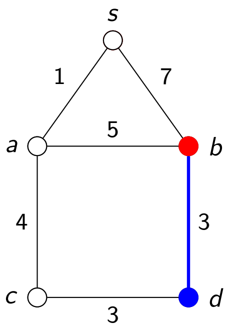



```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $d : 8$
dist: $s : 0$, $a : 1$, $b : 6$, $c : 5$, $d : 8$
Solved: s, a, b, c

x = b
y =
newDist =

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $d:8$

dist: $s:0$, $a:1$, $b:6$, $c:5$, $d:8$

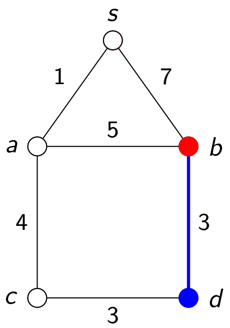
Solved: s, a, b, c

x = b

y = d

newDist =

Dijkstra's algorithm: example




```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $d : 8$

dist: $s : 0,\ a : 1,\ b : 6,\ c : 5,\ d : 8$

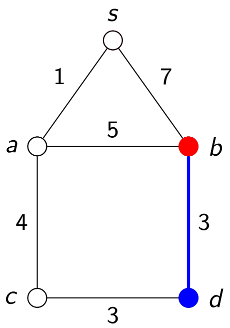
Solved: $s,\ a,\ b,\ c$

x = b

y = d

newDist = $6 + 3 = 9$

Dijkstra's algorithm: example

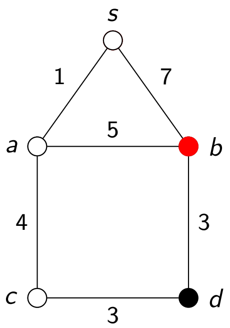



```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
▷       if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

Queue: $d : 8$
dist: $s : 0$, $a : 1$, $b : 6$, $c : 5$, $d : 8$
Solved: s, a, b, c

x $=$ b
y $=$ d
newDist $=$ $6 + 3 = 9$

Dijkstra's algorithm: example




```
    initialize
▷   while (queue not empty)
        x = next();
        solved[x] = true;
        for each unsolved neighbour y of x
            newDist = dist[x] + weight(x,y);
            if (newDist < dist[y])
                dist[y] = newDist;
                update queue;
```

Queue: $d : 8$

dist: $s : 0$, $a : 1$, $b : 6$, $c : 5$, $d : 8$

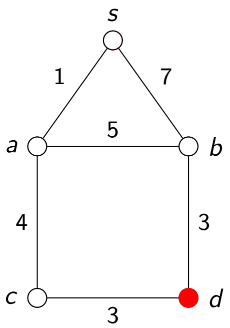
Solved: s, a, b, c

x = b

y =

newDist =

Dijkstra's algorithm: example

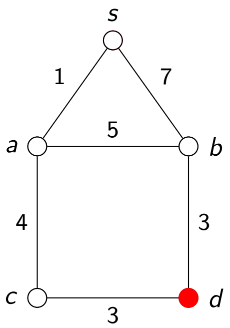



```
  initialize
  while (queue not empty)
▷     x = next();
      solved[x] = true;
      for each unsolved neighbour y of x
          newDist = dist[x] + weight(x,y);
          if (newDist < dist[y])
              dist[y] = newDist;
              update queue;
```

Queue: empty
dist: $s:0$, $a:1$, $b:6$, $c:5$, $d:8$
Solved: s, a, b, c

x = d
y =
newDist =

Dijkstra's algorithm: example

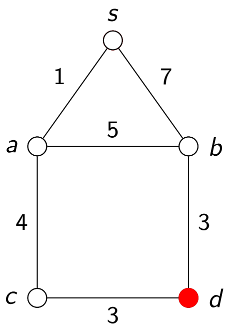



```
  initialize
  while (queue not empty)
      x = next();
▷     solved[x] = true;
      for each unsolved neighbour y of x
          newDist = dist[x] + weight(x,y);
          if (newDist < dist[y])
              dist[y] = newDist;
              update queue;
```

Queue: empty
dist: $s:0$, $a:1$, $b:6$, $c:5$, $d:8$
Solved: s, a, b, c, d

x = d
y =
newDist =

Dijkstra's algorithm: example

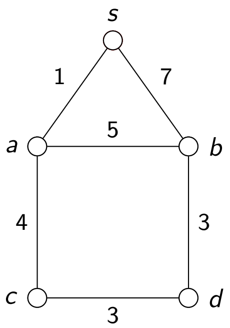



```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
         newDist = dist[x] + weight(x,y);
         if (newDist < dist[y])
             dist[y] = newDist;
             update queue;
```

Queue: empty
dist: $s : 0$, $a : 1$, $b : 6$, $c : 5$, $d : 8$
Solved: s, a, b, c, d

x = d
y =
newDist =

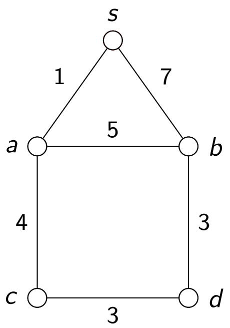
Dijkstra's algorithm: example




```
  initialize
▷ while (queue not empty)
      x = next();
      solved[x] = true;
      for each unsolved neighbour y of x
          newDist = dist[x] + weight(x,y);
          if (newDist < dist[y])
              dist[y] = newDist;
              update queue;
```

Queue: empty
dist: $s : 0$, $a : 1$, $b : 6$, $c : 5$, $d : 8$
Solved: s, a, b, c, d

x = d
y =
newDist =




```
initialize
while (queue not empty)
     x = next();
     solved[x] = true;
     for each unsolved neighbour y of x
          newDist = dist[x] + weight(x,y);
          if (newDist < dist[y])
               dist[y] = newDist;
               update queue;
```

Queue: empty
dist: $s:0,\ a:1,\ b:6,\ c:5,\ d:8$
Solved: $s,\ a,\ b,\ c,\ d$

x = d
y =
newDist =

Dijkstra's algorithm: finding the paths

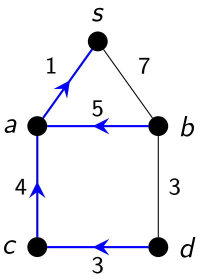
- The algorithm given just finds the lengths of the shortest paths.

Dijkstra's algorithm: finding the paths

- The algorithm given just finds the lengths of the shortest paths.
- To find the actual path, add an array `int[] pred`.
- When we find a new shortest path to y, set `pred[y]=x`, the vertex whose neighbours we were scanning when we found y.

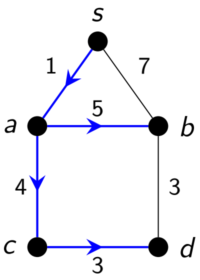
Dijkstra's algorithm: finding the paths

- The algorithm given just finds the lengths of the shortest paths.
- To find the actual path, add an array `int[] pred`.
- When we find a new shortest path to y, set `pred[y]=x`, the vertex whose neighbours we were scanning when we found y.



Dijkstra's algorithm: finding the paths

- The algorithm given just finds the lengths of the shortest paths.
- To find the actual path, add an array `int[] pred`.
- When we find a new shortest path to y, set `pred[y]=x`, the vertex whose neighbours we were scanning when we found y.



Dijkstra's algorithm: running time

```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

- For n vertices and e edges, initialization takes time $\Theta(n)$.

Dijkstra's algorithm: running time

```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

- For n vertices and e edges, initialization takes time $\Theta(n)$.
- Every vertex is removed from the queue once: takes time $O(n \log n)$.

Dijkstra's algorithm: running time

```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

- For n vertices and e edges, initialization takes time $\Theta(n)$.
- Every vertex is removed from the queue once: takes time $O(n \log n)$.
- In an undirected graph, each edge is processed twice in total by the "for each". Using adjacency lists means we don't need to search for edges.
- Each time, we may update the queue, in time $O(\log n)$.

Dijkstra's algorithm: running time

```
initialize
while (queue not empty)
    x = next();
    solved[x] = true;
    for each unsolved neighbour y of x
        newDist = dist[x] + weight(x,y);
        if (newDist < dist[y])
            dist[y] = newDist;
            update queue;
```

- For n vertices and e edges, initialization takes time $\Theta(n)$.
- Every vertex is removed from the queue once: takes time $O(n \log n)$.
- In an undirected graph, each edge is processed twice in total by the "for each". Using adjacency lists means we don't need to search for edges.
- Each time, we may update the queue, in time $O(\log n)$.
- Total running time is $O((n + e) \log n)$.