Lecture 13

Algorithms & Data Structures

Goldsmiths Computing

January 28, 2019

Outline

Outline

Lecture

1. Implicit data structures (...cont'd)
   - Heaps

VLE activities

Binary search quiz

Statistics so far:

- A attempts: average mark B
- C students: average mark D
  - E under 4.00, F over 6.99, G at 10.00

Quiz closes at 16:00 on Friday 1st February

- no extensions
- grade is
  - 0 (for no attempt)
  - $30 + 70 \times (\text{score}/10)^2$

VLE activities (cont'd)

Hashing quiz

VLE activities (cont'd)

Hashing quiz

VLE activities (cont'd)

Hashing quiz

VLE activities (cont'd)

Binary search submission

Outline

Motivation

Many, many problems can be expressed in terms of graphs.

Definition

A graph is a set of vertices (nodes) which are linked by zero or more edges from nodes to nodes, each of which can have a weight.

Operations

vertices	return the collection of vertices {v} in the graph
edges	return the collection of edges {(u,v)} in the graph
addVertex[v]	add vertex v to the graph
addEdge[e]	add edge e to the graph
neighbours[v]	return the collection of vertices directly reachable from v
weight[u,v]	return the weight of the edge between u and v
constructor(V, E)	make a new graph with with given vertex and edge collection

Edge operations

from	return the source vertex of this edge
to	return the destination vertex of this edge
weight	return the weight of this edge

Representations

adjacency matrix — a matrix of edge information linking vertices

adjacency list — an array of vertices, each vertex contaning edges from that vertex

edge list — a list of edges in the graph, along with a set of vertices

Other definitions

Directed

A directed graph has edges that are one-directional: an edge from u to v does not imply an edge from v to u.

Undirected

An undirected graph has two-directional edges: an edge from u to v with weight w implies an edge from v to u with weight w.

Tree

A tree (in the context of graphs) is an undirected graph where any two vertices are joined by exactly one path.

Directed acyclic

A directed acyclic graph or DAG is a directed graph where no sequence of edges returns to its starting point.

Work

1. Reading:
   - CLRS, chapter 22
   - Drozdek, section 8.1
   - DPV, section 3.1

Outline

Introduction

Graphs

Spanning trees

Path finding

Motivation

- Internet routing
- Electricity, cable, road networks
- Maze generation

Definition

A tree T is a spanning tree of a graph G if it is:

a subgraph of G includes only edges that are present in G; and

spans G includes all vertices of G

Properties

If G has |V| vertices, a spanning tree has:

- |V| vertices
- |V|−1 edges (proof?)

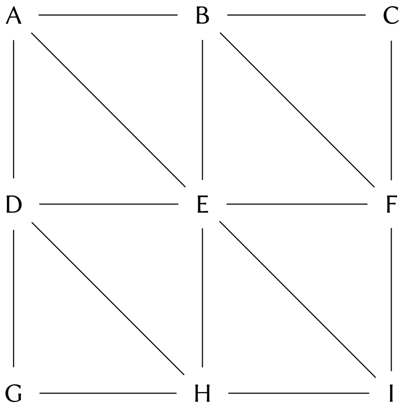
Random spanning tree

A B C

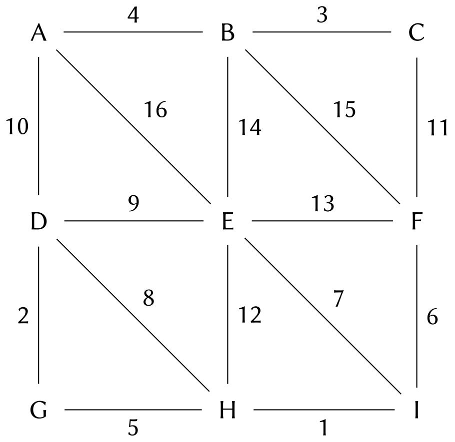
D E F

G H I

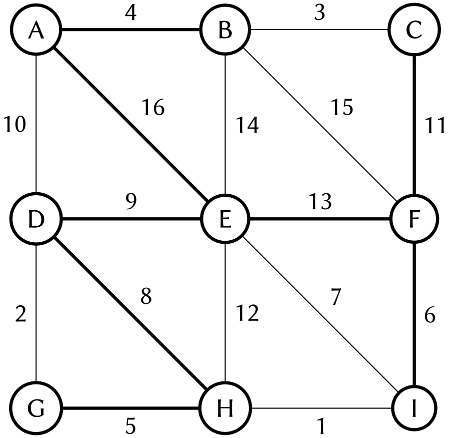
Random spanning tree



Random spanning tree



Random spanning tree



Minimum spanning tree

A minimum spanning tree for graph G is a spanning tree of graph G whose edge weights sum to the minimum possible total weight for that graph.

Prim's algorithm

function PRIMMST(G)
  VS ← VERTICES(G)
  T ← **new** Graph(FIRST(vs),{})
  while |T| < |G| **do**
    E ← {e | e ∈ EDGES(G) ∧
      FROM(e) ∈ VERTICES(T) ∧ TO(e) ∉ VERTICES(T)}
    newE ← $\text{argmin}_{e \in E}$ WEIGHT(e)
    newV ← TO(newE)
    ADDVERTEX(T,newV); ADDEDGES(T,newE)
  end while
end function

Prim's algorithm

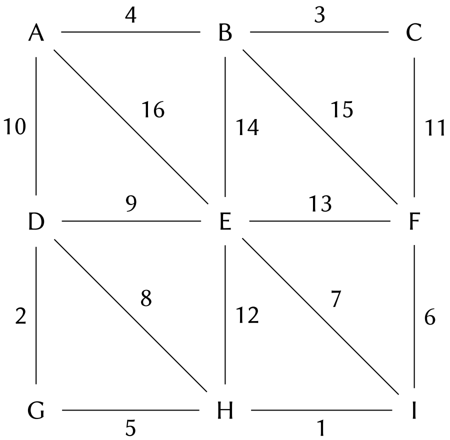
```
function PrimMST(G)
    vs ← vertices(G)
    T ← new Graph(first(vs),{})
    while |T| < |G| do
        E ← {e | e ∈ edges(G) ∧
            from(e) ∈ vertices(T) ∧ to(e) ∉ vertices(T)}
        newE ← argmin_{e∈E} weight(e)
        newV ← to(newE)
        addVertex(T,newV); addEdges(T,newE)
    end while
end function
```

1. Initialise the minimum spanning tree with a vertex from the graph.
2. Until all the vertices are included in the tree,
    - find the edge with smallest weight that links a vertex in the tree so far with a vertex not yet in the tree;
    - add that edge, and the new vertex, to the minimum spanning tree.

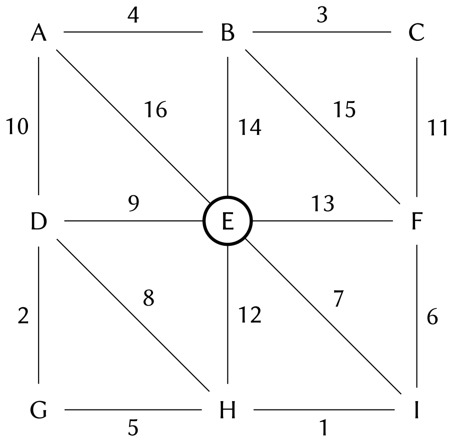
Prim's algorithm

```
function PrimMST(G)
    vs ← vertices(G)
    T ← new Graph(first(vs),{})
    while |T| < |G| do
        newE ← NIL; newV ← NIL; w ← ∞
        for e ∈ edges(G) ∧ from(e) ∈ vertices(T) ∧ to(e) ∉ vertices(T) do
            if weight(e) < w then
                w ← weight(e); newE ← e; newV ← to(e)
            end if
        end for
        addVertex(T,newV); addEdge(T,newE)
    end while
end function
```

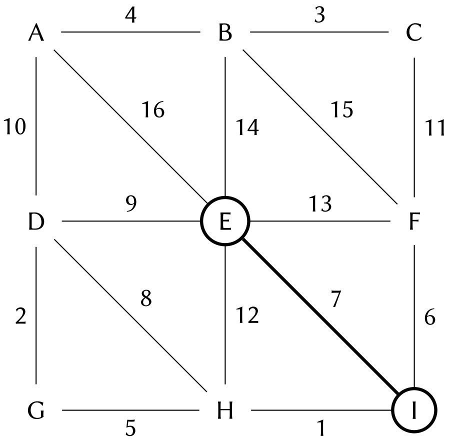
Prim's algorithm: example



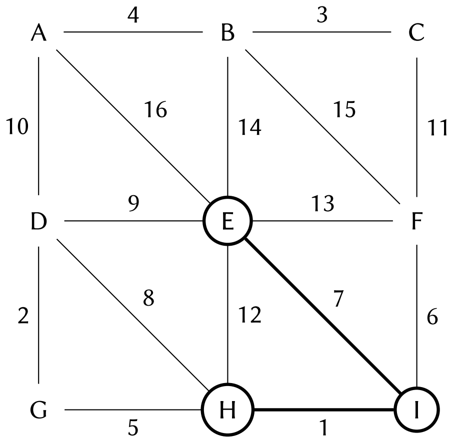
Prim's algorithm: example



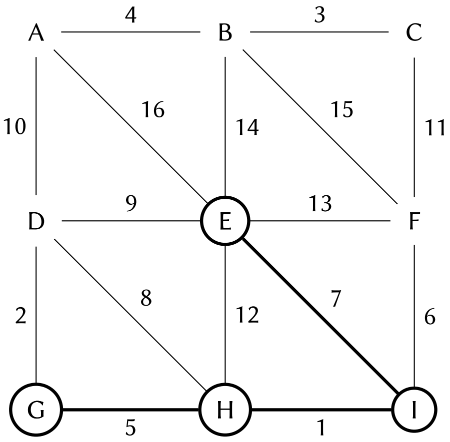
Prim's algorithm: example



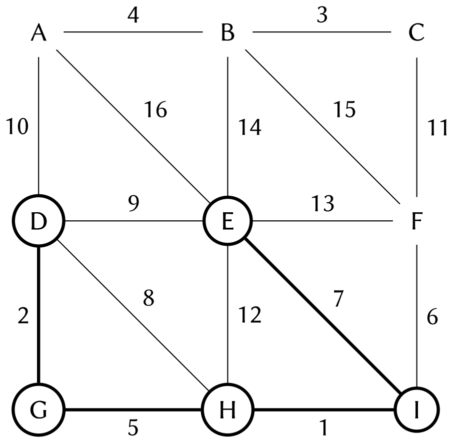
Prim's algorithm: example



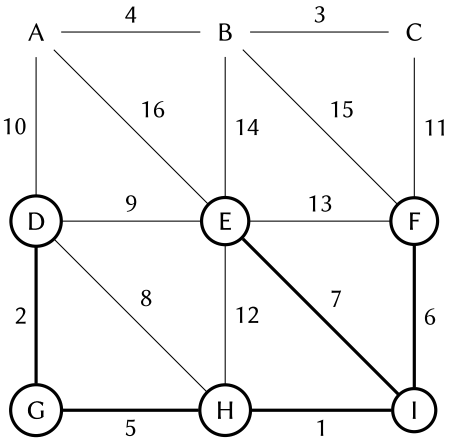
Prim's algorithm: example



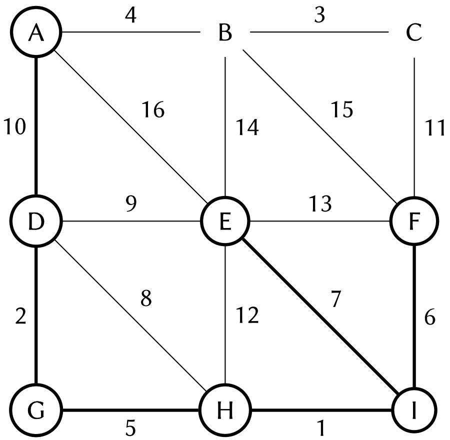
Prim's algorithm: example



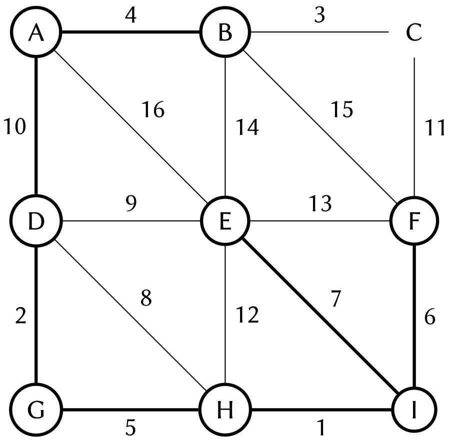
Prim's algorithm: example



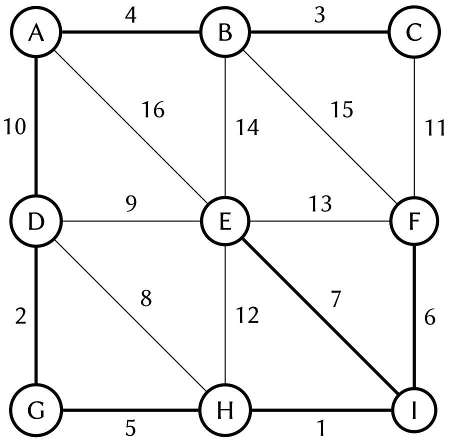
Prim's algorithm: example



Prim's algorithm: example



Prim's algorithm: example



Prim's algorithm: proof sketch

By contradiction. Let P be the spanning tree generated by Prim's algorithm on G, and T be the minimum spanning tree.

- if P = T, we are done.
- if P ≠ T,
    - there is an edge e in P not in T;
    - we added that edge to P at some point, joining a set of vertices V in Prim's tree with one of the set of vertices G-V;
    - find the edge f in T that joins V with G-V;
    - the tree T-f+e must have lower cost than T (why?) and is a spanning tree (why?).

Kruskal's algorithm

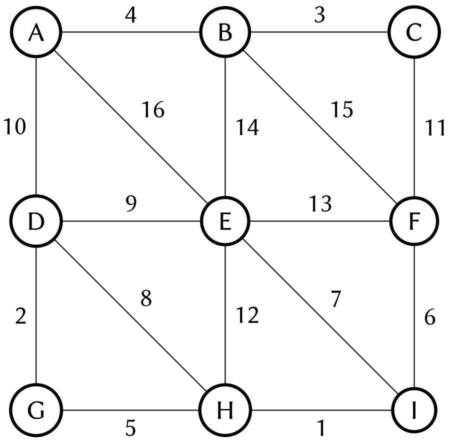
```
function KruskalMST(G)
    vs ← vertices(G)
    T ← new Graph(vs,{})
    Z ← new DisjointSet()
    for v ∈ vs do
        make-set(Z,v)
    end for
    for (u,v) ∈ edges(G) sorted by weight do
        if find(Z,u) ≠ find(Z,v) then
            addEdge(T,(u,v))
            union(Z,u,v)
        end if
    end for
end function
```

Kruskal's algorithm

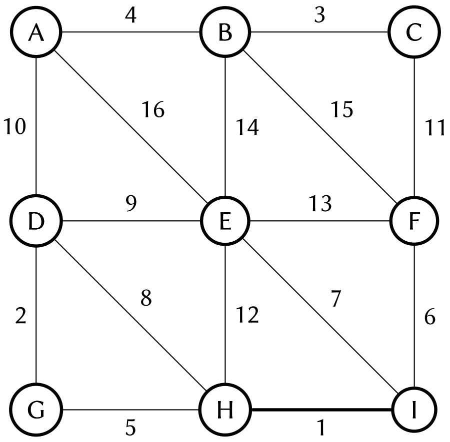
```
function KruskalMST(G)
    vs ← vertices(G)
    T ← new Graph(vs,{})
    Z ← new DisjointSet()
    for v ∈ vs do
        make-set(Z,v)
    end for
    for (u,v) ∈ edges(G) sorted by weight do
        if find(Z,u) ≠ find(Z,v) then
            addEdge(T,(u,v))
            union(Z,u,v)
        end if
    end for
end function
```

1. Initialise the minimum spanning tree with all vertices and no edges.
2. Put each vertex into its own equivalence class
3. Iterate over all the edges in the graph, ordered by weight:
    - add the edge to the tree if it joins two different equivalence classes;
    - make the edge's two vertices' equivalence classes be the same

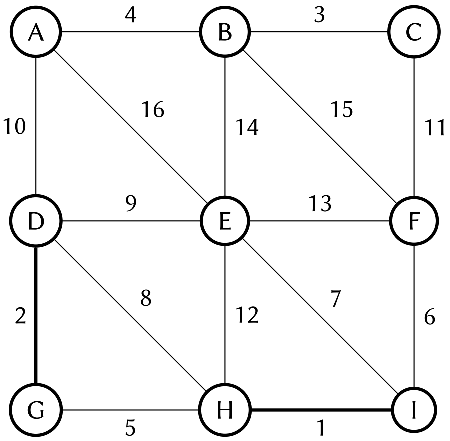
Kruskal's algorithm: example



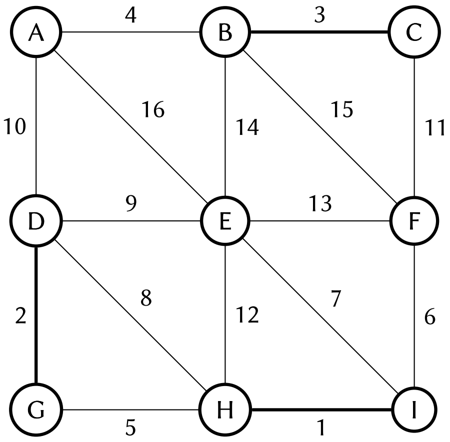
Kruskal's algorithm: example



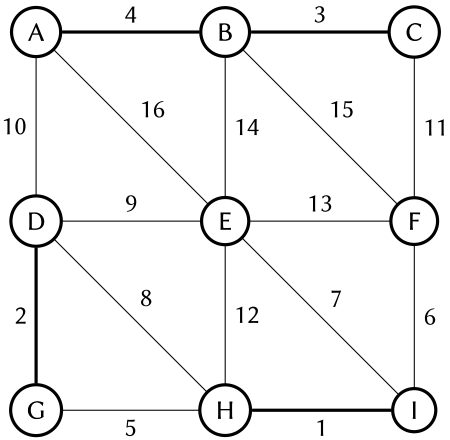
Kruskal's algorithm: example



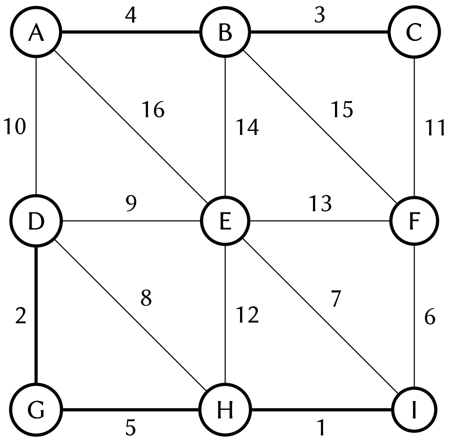
Kruskal's algorithm: example



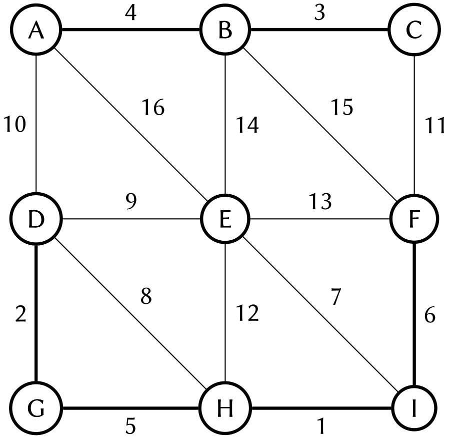
Kruskal's algorithm: example



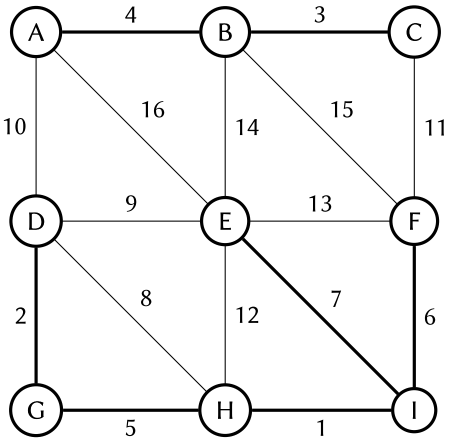
Kruskal's algorithm: example



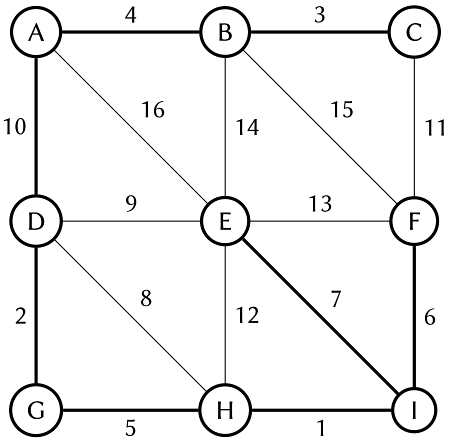
Kruskal's algorithm: example



Kruskal's algorithm: example



Kruskal's algorithm: example



Kruskal's algorithm: proof

Similar to Prim's algorithm: consider the first edge added in the spanning tree K that is not in T

Work

1. Reading:
   - CLRS, chapter 23
   - Drozdek, section 8.5
   - DPV, section 5.1
2. Questions:

   CLRS Exercises 23.1-7, 23.2-1, 23.2-2, 23.2-4, 23.2-5
   CLRS Problem 23-1
   DPV Exercises 5.1, 5.2, 5.5
3. Complete the proof of Kruskal's algorithm.
4. Choose a concrete representation for graphs and edge sets. For your choices, what is the time complexity of Prim's algorithm?

Outline

Introduction

Graphs

Spanning trees

Path finding

Motivation

- Exploration of known, partially known or unknown surroundings
- Component of various AI solutions
  - especially agents exploring some space:
    - ... game enemies
    - ... NPCs
    - ... self-driving cars

Definition

Single-source shortest path

- from a single source node:
  - find the shortest path to every node in the graph
  - stop early if we have a specific target node

Basic approach

"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."

Basic approach

> *"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."*

Initial state

Start at the start node

Basic approach

> *"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."*

Initial state

Start at the start node

Goal state

Stop when we have found a path (optimally: shortest) to the target node

- (if no target: stop when there are no nodes without the shortest path known)

Basic approach

"Begin at the beginning," the King said gravely, "and go on till you come to the end: then stop."

Initial state

Start at the start node

Goal state

Stop when we have found a path (optimally: shortest) to the target node

- (if no target: stop when there are no nodes without the shortest path known)

State expansion

Explore the graph using neighbours of already-visited nodes

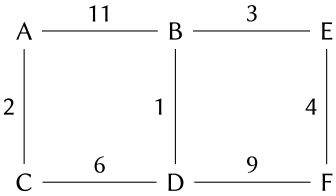
Greedy best-first search

Explore the graph using the neighbour of the current node which is closest to the target.

function GBFS(G,start,end)
  current $\leftarrow$ start
  result $\leftarrow$ **new** queue()
  while current $\neq$ end **do**
    ENQUEUE(result,current)
    ns $\leftarrow$ NEIGHBOURS(G,current)
    current $\leftarrow \arg\min_{n \in \mathrm{ns}} d(n, \mathrm{end})$
  end while
  return result
end function

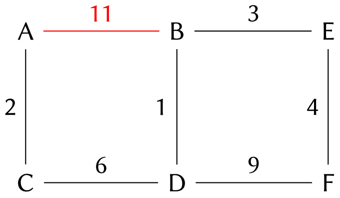
Example

Node	d(n,F)
A	14
B	6
C	12
D	7
E	4
F	0



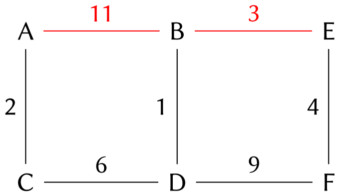
Example

Node	d(n,F)
A	14
B	6
C	12
D	7
E	4
F	0



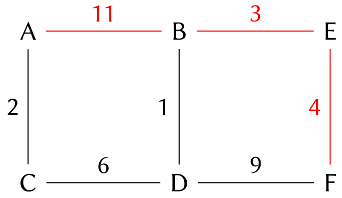
Example

Node	d(n,F)
A	14
B	6
C	12
D	7
E	4
F	0



Example

Node	d(n,F)
A	14
B	6
C	12
D	7
E	4
F	0



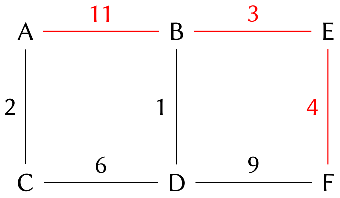
Result

path A ⟶ B ⟶ E ⟶ F

distance 18

Example

Node	d(n,F)
A	14
B	6
C	12
D	7
E	4
F	0



Result

path A $\longrightarrow$ B $\longrightarrow$ E $\longrightarrow$ F

distance 18

Problems

- does not necessarily find a solution!
- not guaranteed optimal

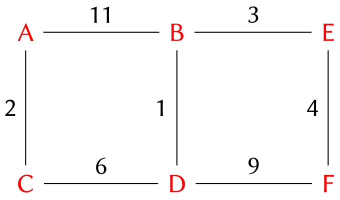
Dijkstra's algorithm

Explore the graph using the neighbour of the already-visited nodes with the smallest distance from the start node

```
function DIJKSTRA(G,start,end)
    dist ← new table(); prev ← new table()
    Q ← new min-heap(dist)
    for v ∈ G do
        dist[v] ← 0 if v = start else ∞; INSERT(Q,v)
    end for
    while ¬ EMPTY(Q) do
        u ← EXTRACT-MIN(Q)
        if u = end then
            s ← new stack()
            while u ≠ start do
                PUSH(s,u); u ← prev[u]
            end while
            return s
        end if
        for v ∈ NEIGHBOURS(G,u) do
            d ← dist[u] + WEIGHT(G,u,v)
            if d < dist[v] then
                dist[v] ← d; prev[v] ← u; DECREASE-KEY(Q,v,dist[v])
            end if
        end for
    end while
end function
```

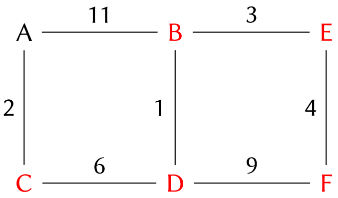
Example

Node	dist[n]	prev[n]
A	0	
B	∞	
C	∞	
D	∞	
E	∞	
F	∞	



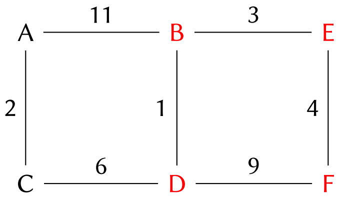
Example

Node	dist[n]	prev[n]
A	0	
B	11	A
C	2	A
D	∞	
E	∞	
F	∞	



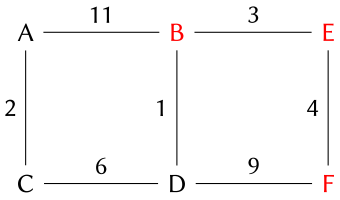
Example

Node	dist[n]	prev[n]
A	0	
B	11	A
C	2	A
D	8	C
E	∞	
F	∞	



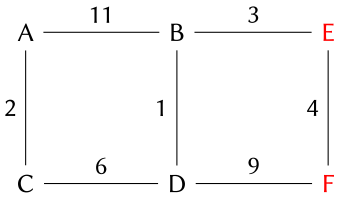
Example

Node	dist[n]	prev[n]
A	0	
B	9	D
C	2	A
D	8	C
E	∞	
F	17	D



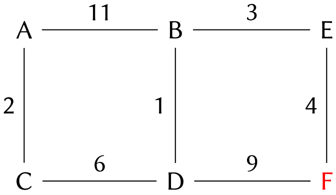
Example

Node	dist[n]	prev[n]
A	0	
B	9	D
C	2	A
D	8	C
E	12	B
F	17	D



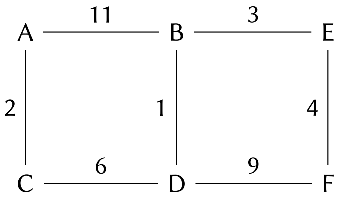
Example

Node	dist[n]	prev[n]
A	0	
B	9	D
C	2	A
D	8	C
E	12	B
F	16	E



Example

Node	dist[n]	prev[n]
A	0	
B	9	D
C	2	A
D	8	C
E	12	B
F	16	E



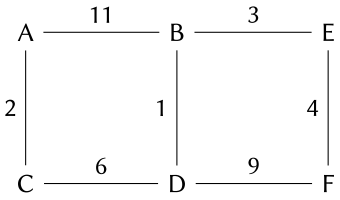
Result

path $A \longrightarrow C \longrightarrow D \longrightarrow B \longrightarrow E \longrightarrow F$

distance 16

Example

Node	dist[n]	prev[n]
A	0	
B	9	D
C	2	A
D	8	C
E	12	B
F	16	E



Result

path $A \longrightarrow C \longrightarrow D \longrightarrow B \longrightarrow E \longrightarrow F$

distance 16

Note

- requires all non-negative weights
- guaranteed to find shortest path
- need priority queue (min-heap) for efficient operation
- does not use distance estimate information

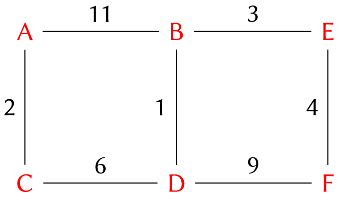
A^*

Explore the graph using the neighbour of the already-visited nodes with the smallest estimated distance from the start node to the target node

```
function A*(G,start,end)
    dist ← new table(); prev ← new table()
    Q ← new min-heap(dist)
    for v ∈ G do
        dist[v] ← 0 if v = start else ∞; INSERT(Q,v)
    end for
    while ¬ EMPTY(Q) do
        u ← EXTRACT-MIN(Q)
        if u = end then
            s ← new stack()
            while u ≠ start do
                PUSH(s,u); u ← prev[u]
            end while
            return s
        end if
        for v ∈ NEIGHBOURS(G,u) do
            d ← dist[u] + WEIGHT(G,u,v) + H(v)
            if d < dist[v] then
                dist[v] ← d; prev[v] ← u; DECREASE-KEY(Q,v,dist[v])
            end if
        end for
    end while
end function
```

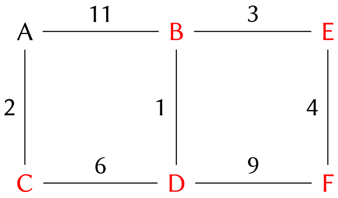
Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	∞	
C	12	∞	
D	7	∞	
E	4	∞	
F	0	∞	



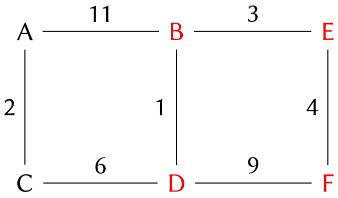
Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	11	A
C	12	2	A
D	7	∞	
E	4	∞	
F	0	∞	



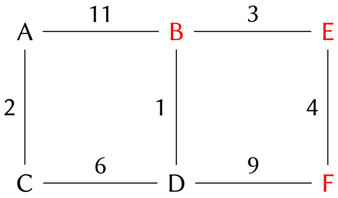
Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	11	A
C	12	2	A
D	7	8	C
E	4	∞	
F	0	∞	



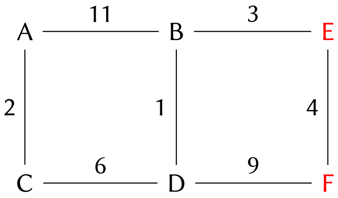
Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	9	D
C	12	2	A
D	7	8	C
E	4	∞	
F	0	17	D



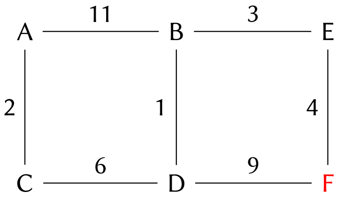
Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	9	D
C	12	2	A
D	7	8	C
E	4	12	B
F	0	17	D



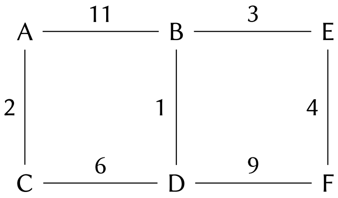
Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	9	D
C	12	2	A
D	7	8	C
E	4	12	B
F	0	16	E



Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	9	D
C	12	2	A
D	7	8	C
E	4	12	B
F	0	16	E



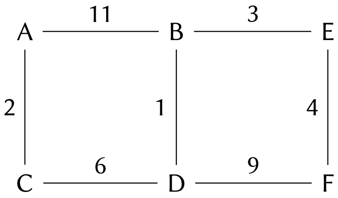
Result

path A → C → D → B → E → F

distance 16

Example

Node	d(n,F)	dist[n]	prev[n]
A	14	0	
B	6	9	D
C	12	2	A
D	7	8	C
E	4	12	B
F	0	16	E



Result

path A ⟶ C ⟶ D ⟶ B ⟶ E ⟶ F

distance 16

Note

- generalisation of Dijkstra's algorithm
- distance estimation h must be admissible
  - lower bound
  - non-negative
  - (Dijkstra's algorithm is A* with h(n) = 0)

Work

1. Reading
   - CLRS, chapter 24
   - Drozdek, sections 8.2, 8.3
2. Questions from CLRS

   Exercises 24.3-1