MINIMUM-COST SPANNING TREES

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil

Centro de Informática
UFPE

Universidade Federal de Pernambuco

Agenda

Introduction

Given n points, connect them in the cheapest possible way so that there will be a path between every pair of points

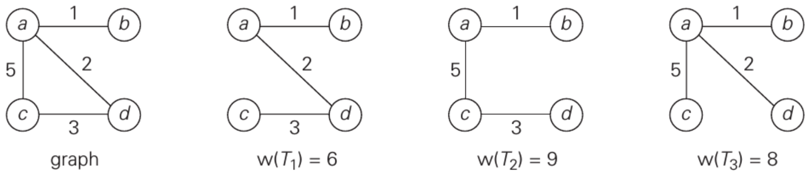
Applications

- Design of all kinds of networks
- Classification purposes
- Constructing approximate solutions

Introduction[1]

Minimum-cost spanning tree (MST)

- Let G be an undirected connected graph, an MST is a connected acyclic subgraph (i.e, a tree) that contains all vertices of G. If G has weights, an MST is the tree with the smallest weight (sum of all weights on all its edges).



[1] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Agenda

Prim's algorithm

Constructs an MST through a sequence of expanding subtrees

- Initial subtree consists of a single arbitrary vertex
- On each iteration, expands the current tree in a greedy manner by attaching to it the nearest vertex not in this tree
  - Nearest vertex: a vertex no in the tree connected to a vertex in the tree by an edge of the smallest weight
- Algorithm stops after including all vertices

Prim's algorithm: similar to the Dijkstra's algorithm

- Can be used on weighted graphs with negative weights (and also with negative cycles)

Time efficiency (same as of Dijkstra's)

- Matrix with no heap: $\Theta(\mid V \mid^2)$
- List with heap: $\Theta((\mid V \mid + \mid E \mid) \, log \mid V \mid)$

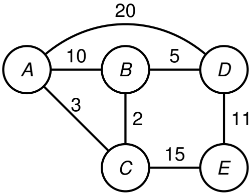
Centro de Informática UFPE

Universidade Federal de Pernambuco

Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

```
for i ← 0 to n(G) − 1 do
    D[i] ← ∞ ; V[i] ← −;
    setMark(G, i, UNVISITED);
H[1] ← (0, 0, 0) ; D[0] ← 0;
for i ← 0 to n(G) − 1 do
    repeat
        (p, v) ← removemin(H);
        if v = NULL then return;
    until getMark(G, v) = UNVISITED;
    setMark(G, v, VISITED) ; V[v] ← p;
    w ← first(G, v);
    while w < n(G) do
        if getMark(G, w) ≠ VISITED ∧
          D[w] > weight(G, v, w) then
            D[w] ← weight(G, v, w);
            insert(H, (v, w, D[w]));
        w ← next(G, v, w);
```



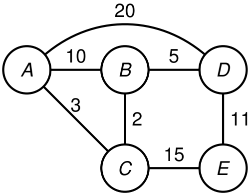
	A	B	C	D	E
Mark	×	×	×	×	×
D	0	∞	∞	∞	∞
V	−	−	−	−	−

Centro de Informática UFPE
Universidade Federal de Pernambuco

Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

```
for i ← 0 to n(G) − 1 do
    D[i] ← ∞ ; V[i] ← −;
    setMark(G, i, UNVISITED);
H[1] ← (0, 0, 0) ; D[0] ← 0;
for i ← 0 to n(G) − 1 do
    repeat
        (p, v) ← removemin(H);
        if v = NULL then return;
    until getMark(G, v) = UNVISITED;
    setMark(G, v, VISITED) ; V[v] ← p;
    w ← first(G, v);
    while w < n(G) do
        if getMark(G, w) ≠ VISITED ∧
          D[w] > weight(G, v, w) then
            D[w] ← weight(G, v, w);
            insert(H, (v, w, D[w]));
        w ← next(G, v, w);
```



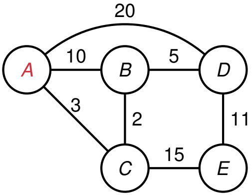
	A	B	C	D	E
Mark	×	×	×	×	×
D	0	∞	∞	∞	∞
V	–	–	–	–	–

(A,A,0)

Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

```
for i ← 0 to n(G) − 1 do
    D[i] ← ∞ ; V[i] ← −;
    setMark(G, i, UNVISITED);
H[1] ← (0, 0, 0) ; D[0] ← 0;
for i ← 0 to n(G) − 1 do
    repeat
        (p, v) ← removemin(H);
        if v = NULL then return;
    until getMark(G, v) = UNVISITED;
    setMark(G, v, VISITED) ; V[v] ← p;
    w ← first(G, v);
    while w < n(G) do
        if getMark(G, w) ≠ VISITED ∧
          D[w] > weight(G, v, w) then
            D[w] ← weight(G, v, w);
            insert(H, (v, w, D[w]));
        w ← next(G, v, w);
```



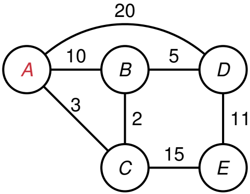
	A	B	C	D	E
Mark	✓	×	×	×	×
D	0	∞	∞	∞	∞
V	A	–	–	–	–

—

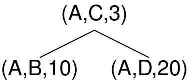
Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

```
for i ← 0 to n(G) − 1 do
    D[i] ← ∞ ; V[i] ← −;
    setMark(G, i, UNVISITED);
H[1] ← (0, 0, 0) ; D[0] ← 0;
for i ← 0 to n(G) − 1 do
    repeat
        (p, v) ← removemin(H);
        if v = NULL then return;
    until getMark(G, v) = UNVISITED;
    setMark(G, v, VISITED) ; V[v] ← p;
    w ← first(G, v);
    while w < n(G) do
        if getMark(G, w) ≠ VISITED ∧
          D[w] > weight(G, v, w) then
            D[w] ← weight(G, v, w);
            insert(H, (v, w, D[w]));
        w ← next(G, v, w);
```



	A	B	C	D	E
Mark	✓	×	×	×	×
D	0	10	3	20	∞
V	A	–	–	–	–

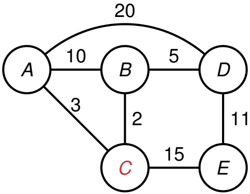


Centro de Informática UFPE
Universidade Federal de Pernambuco

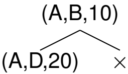
Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

```
for i ← 0 to n(G) − 1 do
    D[i] ← ∞ ; V[i] ← −;
    setMark(G, i, UNVISITED);
H[1] ← (0, 0, 0) ; D[0] ← 0;
for i ← 0 to n(G) − 1 do
    repeat
        (p, v) ← removemin(H);
        if v = NULL then return;
    until getMark(G, v) = UNVISITED;
    setMark(G, v, VISITED) ; V[v] ← p;
    w ← first(G, v);
    while w < n(G) do
        if getMark(G, w) ≠ VISITED ∧
          D[w] > weight(G, v, w) then
            D[w] ← weight(G, v, w);
            insert(H, (v, w, D[w]));
        w ← next(G, v, w);
```



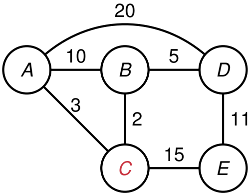
	A	B	C	D	E
Mark	✓	×	✓	×	×
D	0	10	3	20	∞
V	A	–	A	–	–



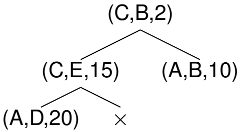
Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

```
for i ← 0 to n(G) − 1 do
    D[i] ← ∞ ; V[i] ← −;
    setMark(G, i, UNVISITED);
H[1] ← (0, 0, 0) ; D[0] ← 0;
for i ← 0 to n(G) − 1 do
    repeat
        (p, v) ← removemin(H);
        if v = NULL then return;
    until getMark(G, v) = UNVISITED;
    setMark(G, v, VISITED) ; V[v] ← p;
    w ← first(G, v);
    while w < n(G) do
        if getMark(G, w) ≠ VISITED ∧
          D[w] > weight(G, v, w) then
            D[w] ← weight(G, v, w);
            insert(H, (v, w, D[w]));
        w ← next(G, v, w);
```



	A	B	C	D	E
Mark	✓	×	✓	×	×
D	0	2	3	20	15
V	A	–	A	–	–

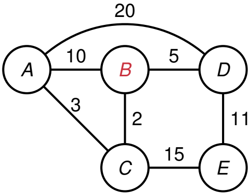


Centro de Informática UFPE
Universidade Federal de Pernambuco

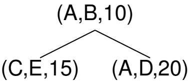
Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

for $i \leftarrow 0$ **to** $n(G) - 1$ **do**
  $D[i] \leftarrow \infty$; $V[i] \leftarrow -$;
  $setMark(G, i, UNVISITED)$;
$H[1] \leftarrow (0, 0, 0)$; $D[0] \leftarrow 0$;
for $i \leftarrow 0$ **to** $n(G) - 1$ **do**
  repeat
    $(p, v) \leftarrow removemin(H)$;
    if $v = NULL$ **then return**;
  until $getMark(G, v) = UNVISITED$;
  $setMark(G, v, VISITED)$; $V[v] \leftarrow p$;
  $w \leftarrow first(G, v)$;
  while $w < n(G)$ **do**
    if $getMark(G, w) \neq VISITED \wedge D[w] > weight(G, v, w)$ **then**
      $D[w] \leftarrow weight(G, v, w)$;
      $insert(H, (v, w, D[w]))$;
    $w \leftarrow next(G, v, w)$;



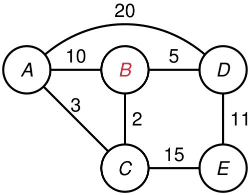
	A	B	C	D	E
Mark	✓	✓	✓	×	×
D	0	2	3	20	15
V	A	C	A	–	–



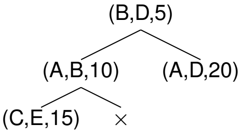
Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

```
for i ← 0 to n(G) − 1 do
    D[i] ← ∞ ; V[i] ← −;
    setMark(G, i, UNVISITED);
H[1] ← (0, 0, 0) ; D[0] ← 0;
for i ← 0 to n(G) − 1 do
    repeat
        (p, v) ← removemin(H);
        if v = NULL then return;
    until getMark(G, v) = UNVISITED;
    setMark(G, v, VISITED) ; V[v] ← p;
    w ← first(G, v);
    while w < n(G) do
        if getMark(G, w) ≠ VISITED ∧
          D[w] > weight(G, v, w) then
            D[w] ← weight(G, v, w);
            insert(H, (v, w, D[w]));
        w ← next(G, v, w);
```



	A	B	C	D	E
Mark	✓	✓	✓	×	×
D	0	2	3	5	15
V	A	C	A	–	–

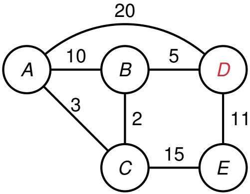


Centro de Informática UFPE
Universidade Federal de Pernambuco

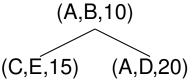
Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

for $i \leftarrow 0$ **to** $n(G) - 1$ **do**
  $D[i] \leftarrow \infty$; $V[i] \leftarrow -$;
  $setMark(G, i, UNVISITED)$;
$H[1] \leftarrow (0, 0, 0)$; $D[0] \leftarrow 0$;
for $i \leftarrow 0$ **to** $n(G) - 1$ **do**
  repeat
    $(p, v) \leftarrow removemin(H)$;
    if $v = NULL$ **then return**;
  until $getMark(G, v) = UNVISITED$;
  $setMark(G, v, VISITED)$; $V[v] \leftarrow p$;
  $w \leftarrow first(G, v)$;
  while $w < n(G)$ **do**
    if $getMark(G, w) \neq VISITED \wedge D[w] > weight(G, v, w)$ **then**
      $D[w] \leftarrow weight(G, v, w)$;
      $insert(H, (v, w, D[w]))$;
    $w \leftarrow next(G, v, w)$;



	A	B	C	D	E
Mark	✓	✓	✓	✓	×
D	0	2	3	5	15
V	A	C	A	B	–

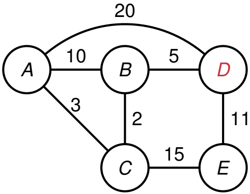


Centro de Informática UFPE
Universidade Federal de Pernambuco

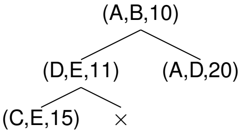
Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

```
for i ← 0 to n(G) − 1 do
    D[i] ← ∞ ; V[i] ← −;
    setMark(G, i, UNVISITED);
H[1] ← (0, 0, 0) ; D[0] ← 0;
for i ← 0 to n(G) − 1 do
    repeat
        (p, v) ← removemin(H);
        if v = NULL then return;
    until getMark(G, v) = UNVISITED;
    setMark(G, v, VISITED) ; V[v] ← p;
    w ← first(G, v);
    while w < n(G) do
        if getMark(G, w) ≠ VISITED ∧
          D[w] > weight(G, v, w) then
            D[w] ← weight(G, v, w);
            insert(H, (v, w, D[w]));
        w ← next(G, v, w);
```



	A	B	C	D	E
Mark	✓	✓	✓	✓	×
D	0	2	3	5	11
V	A	C	A	B	–



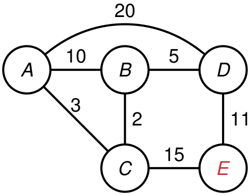
Centro de Informática UFPE

Universidade Federal de Pernambuco

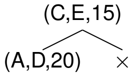
Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

```
for i ← 0 to n(G) − 1 do
    D[i] ← ∞ ; V[i] ← −;
    setMark(G, i, UNVISITED);
H[1] ← (0, 0, 0) ; D[0] ← 0;
for i ← 0 to n(G) − 1 do
    repeat
        (p, v) ← removemin(H);
        if v = NULL then return;
    until getMark(G, v) = UNVISITED;
    setMark(G, v, VISITED) ; V[v] ← p;
    w ← first(G, v);
    while w < n(G) do
        if getMark(G, w) ≠ VISITED ∧
          D[w] > weight(G, v, w) then
            D[w] ← weight(G, v, w);
            insert(H, (v, w, D[w]));
        w ← next(G, v, w);
```



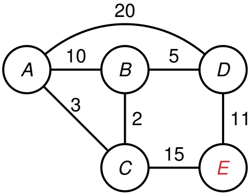
	A	B	C	D	E
Mark	✓	✓	✓	✓	✓
D	0	2	3	5	11
V	A	C	A	B	D



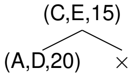
Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

```
for i ← 0 to n(G) − 1 do
    D[i] ← ∞ ; V[i] ← −;
    setMark(G, i, UNVISITED);
H[1] ← (0, 0, 0) ; D[0] ← 0;
for i ← 0 to n(G) − 1 do
    repeat
        (p, v) ← removemin(H);
        if v = NULL then return;
    until getMark(G, v) = UNVISITED;
    setMark(G, v, VISITED) ; V[v] ← p;
    w ← first(G, v);
    while w < n(G) do
        if getMark(G, w) ≠ VISITED ∧
          D[w] > weight(G, v, w) then
            D[w] ← weight(G, v, w);
            insert(H, (v, w, D[w]));
        w ← next(G, v, w);
```



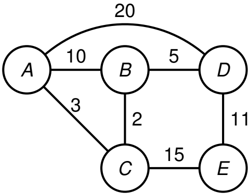
	A	B	C	D	E
Mark	✓	✓	✓	✓	✓
D	0	2	3	5	11
V	A	C	A	B	D



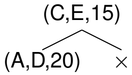
Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

```
for i ← 0 to n(G) − 1 do
    D[i] ← ∞ ; V[i] ← −;
    setMark(G, i, UNVISITED);
H[1] ← (0, 0, 0) ; D[0] ← 0;
for i ← 0 to n(G) − 1 do
    repeat
        (p, v) ← removemin(H);
        if v = NULL then return;
    until getMark(G, v) = UNVISITED;
    setMark(G, v, VISITED) ; V[v] ← p;
    w ← first(G, v);
    while w < n(G) do
        if getMark(G, w) ≠ VISITED ∧
          D[w] > weight(G, v, w) then
            D[w] ← weight(G, v, w);
            insert(H, (v, w, D[w]));
        w ← next(G, v, w);
```



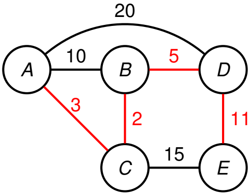
	A	B	C	D	E
Mark	✓	✓	✓	✓	✓
D	0	2	3	5	11
V	A	C	A	B	D



Prim's algorithm

Algorithm: void Prim(Graph G, int[] D, int[] V)

```
for i ← 0 to n(G) − 1 do
    D[i] ← ∞ ; V[i] ← −;
    setMark(G, i, UNVISITED);
H[1] ← (0, 0, 0) ; D[0] ← 0;
for i ← 0 to n(G) − 1 do
    repeat
        (p, v) ← removemin(H);
        if v = NULL then return;
    until getMark(G, v) = UNVISITED;
    setMark(G, v, VISITED) ; V[v] ← p;
    w ← first(G, v);
    while w < n(G) do
        if getMark(G, w) ≠ VISITED ∧
          D[w] > weight(G, v, w) then
            D[w] ← weight(G, v, w);
            insert(H, (v, w, D[w]));
        w ← next(G, v, w);
```



	A	B	C	D	E
Mark	✓	✓	✓	✓	✓
D	0	2	3	5	11
V	A	C	A	B	D

Centro de Informática UFPE

Universidade Federal de Pernambuco

Agenda

Disjoint subsets

Operations (DS = disjoint subset):

- int find(DS ds, int x);
- void union(DS ds, int x, int y);

```
ds ← create_disjointSubset(6); // {1},{2},{3},{4},{5},{6}
union(ds,1,4) ; union(ds,4,5) ; union(ds,1,2);
union(ds,3,6);                   // {1,4,5,2},{3,6}
if find(ds,1) = find(ds,5) then print(true);
else print(false);
```

Important: subset's representative element

Implementations: quick-find and quick-union

Centro de Informática UFPE

Universidade Federal de Pernambuco

Disjoint subsets: quick-find

Implementation based on quick-find

- Array: representative element for each subset elements
- Each subset implement as a linked list

Operations

- $create\_disjointSubset(n)$
  - Initialise n linked lists
  - Each element is its representative
- $find \in \Theta(1)$
- $union \in \Theta(n)$
  - Concatenate the corresponding lists
  - Update all representatives

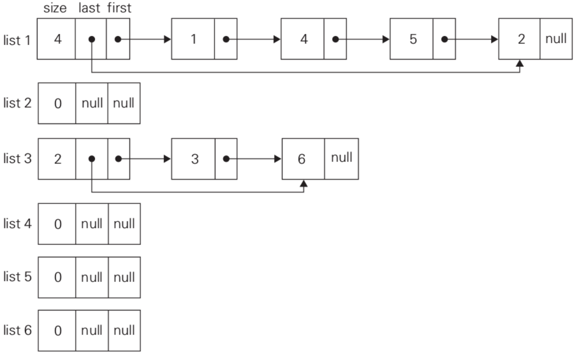
Optimisation (union by size): append shortest list to the largest one

Disjoint subsets: quick-find[2]

After: $union(ds, 1, 4)$; $union(ds, 4, 5)$; $union(1, 2)$; $union(ds, 3, 6)$

subset representatives

element index	representative
1	1
2	1
3	3
4	1
5	1
6	3



Optimisation: union by size

[2] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Disjoint subsets: quick-find

Algorithm: int find(DS ds, int curr)

return $ds.R[curr]$;

Algorithm: void union(DS ds, int a, int b)

$root1, root2 \leftarrow find(ds, a), find(ds, b)$;
if $root1 \neq root2$ **then**
  $l1, l2 \leftarrow ds.sets[root1], ds.sets[root2]$;
  if $l1.size < l2.size$ **then** $swap(l1, l2)$;
  $temp \leftarrow l2.first$;
  while $temp \neq NULL$ **do**
    $ds.R[temp.element] \leftarrow l1.first.element$;
    $temp \leftarrow temp.next$;
  $l1.last.next \leftarrow l2.first$; $l1.last \leftarrow l2.last$;
  $l1.size, l2.size \leftarrow (l1.size + l2.size), 0$;
  $l2.first \leftarrow l2.last \leftarrow NULL$;

Disjoint subsets: quick-union

Implementation based on quick-union

- Using a parent pointer tree: root is the representative element

Operations

- $create\_disjointSubset(n)$
  - Initialise n unitary trees
  - Each element is its representative
- $find \in \Theta(n)$
  - Traverse from x to the subtree's root
- $union \in \Theta(1)$
  - Root of x points to root of y

Optimisation (union by size/rank): link smallest tree to the largest one

- Size: number of nodes
- Rank: subtree height

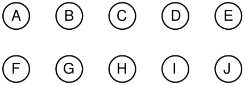
Centro de Informática UFPE

Universidade Federal de Pernambuco

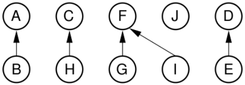
Disjoint subsets: quick-union[3]

- *union*(*ds*, *A*, *B*), *union*(*ds*, *C*, *H*), *union*(*ds*, *G*, *F*), *union*(*ds*, *D*, *E*), *union*(*ds*, *I*, *F*)
- *union*(*ds*, *H*, *A*), *union*(*ds*, *E*, *G*)

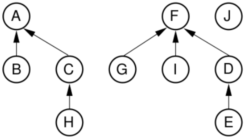
A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9



	0			3		5	2	5	
A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9



	0	0	5	3		5	2	5	
A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9

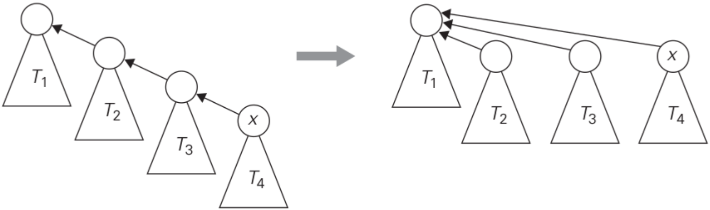


Optimisation:
union by rank/size

[3] Source: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

Disjoint subsets: quick-union[4]

Additional optimisation: path compression during find

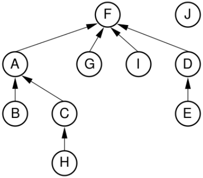


[4] Source: A. Levitin. Introduction to the Design and Analysis of Algorithms. 2011.

Disjoint subsets: quick-union[5]

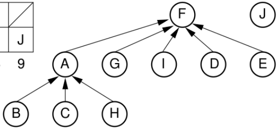
Example: $union(ds, H, E)$

5	0	0	5	3	/	5	2	5	/
A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9



Example (with path compression): $union(ds, H, E)$

5	0	0	5	5	/	5	0	5	/
A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9



[5] Source: C. Shaffer. Data Structures and Algorithm Analysis. 2013.

Disjoint subsets: quick-union

Algorithm: int find(DS ds, int curr)

if $ds.A[curr] = NULL$ **then return** $curr$;
$ds.A[curr] \leftarrow find(ds, ds.A[curr])$;
return $ds.A[curr]$;

Algorithm: void union(DS ds, int a, int b)

$root1 \leftarrow find(ds, a)$;
$root2 \leftarrow find(ds, b)$;
if $root1 \neq root2$ **then** $ds.A[root2] \leftarrow root1$;

Agenda

Kruskal's algorithm

Another greedy algorithm

- Initially, $n = |V|$ MSTs
- On each iteration, chooses an edge of G with the smallest weight that unions two MSTs (i.e., without introducing a cycle)

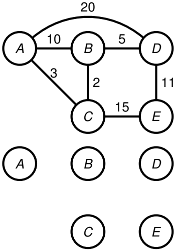
Time efficiency: similar to Prim's

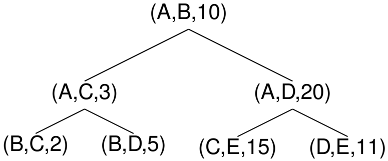
- Prim: best for dense graphs
- Kruskal: best for sparse graphs

Kruskal's algorithm

Algorithm: void Kruskal(Graph G, Graph G')

```
edgecnt ← 1;
for i ← 0 to n(G) − 1 do
    w ← first(G, i);
    while w < n(G) do
        H[edgecnt++] ←
          (i, w, weight(G, i, w));
        w ← next(G, i, w);
HeapBottomUp(H);
ds ← create_disjointSubset(n(G));
numMST ← n(G);
while numMST > 1 do
    (v, u, wt) ← removemin(H);
    if find(ds, v) ≠ find(ds, u) then
        union(ds, v, u);
        setEdge(G', v, u, wt);
        numMST--;
```



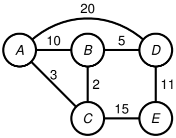


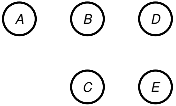
Note: the shown heap is a simplification

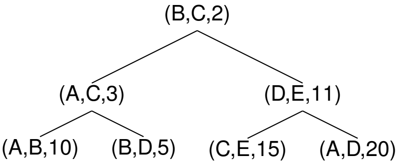
Kruskal's algorithm

Algorithm: void Kruskal(Graph G, Graph G')

```
edgecnt ← 1;
for i ← 0 to n(G) − 1 do
    w ← first(G, i);
    while w < n(G) do
        H[edgecnt++] ←
          (i, w, weight(G, i, w));
        w ← next(G, i, w);
HeapBottomUp(H);
ds ← create_disjointSubset(n(G));
numMST ← n(G);
while numMST > 1 do
    (v, u, wt) ← removemin(H);
    if find(ds, v) ≠ find(ds, u) then
        union(ds, v, u);
        setEdge(G', v, u, wt);
        numMST--;
```







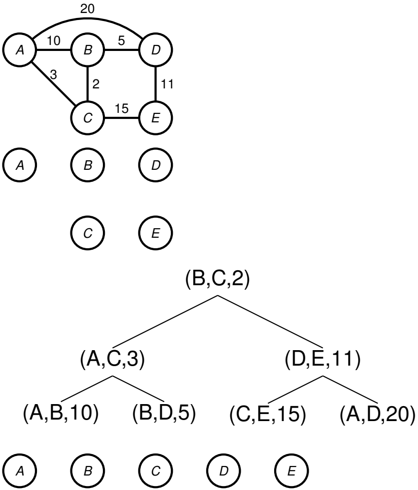
Note: the shown heap is a simplification

Kruskal's algorithm

Algorithm: void Kruskal(Graph G, Graph G')

```
edgecnt ← 1;
for i ← 0 to n(G) − 1 do
    w ← first(G, i);
    while w < n(G) do
        H[edgecnt++] ←
          (i, w, weight(G, i, w));
        w ← next(G, i, w);
HeapBottomUp(H);
ds ← create_disjointSubset(n(G));
numMST ← n(G);
while numMST > 1 do
    (v, u, wt) ← removemin(H);
    if find(ds, v) ≠ find(ds, u) then
        union(ds, v, u);
        setEdge(G', v, u, wt);
        numMST--;
```

Note: the shown heap is a simplification



Centro de Informática UFPE

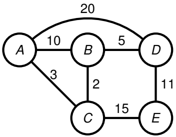
Universidade Federal de Pernambuco

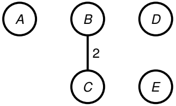
Kruskal's algorithm

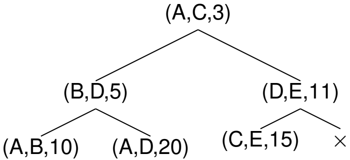
Algorithm: void Kruskal(Graph G, Graph G')

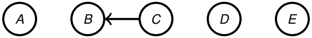
```
edgecnt ← 1;
for i ← 0 to n(G) − 1 do
    w ← first(G, i);
    while w < n(G) do
        H[edgecnt++] ←
          (i, w, weight(G, i, w));
        w ← next(G, i, w);
HeapBottomUp(H);
ds ← create_disjointSubset(n(G));
numMST ← n(G);
while numMST > 1 do
    (v, u, wt) ← removemin(H);
    if find(ds, v) ≠ find(ds, u) then
        union(ds, v, u);
        setEdge(G', v, u, wt);
        numMST--;
```

Note: the shown heap is a simplification









Centro de Informática UFPE

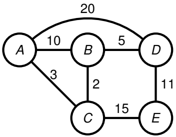
Universidade Federal de Pernambuco

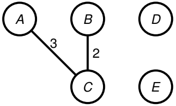
Kruskal's algorithm

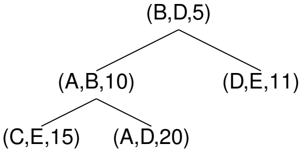
Algorithm: void Kruskal(Graph G, Graph G')

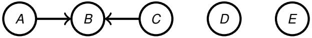
$edgecnt \leftarrow 1$;
for $i \leftarrow 0$ **to** $n(G) - 1$ **do**
    $w \leftarrow first(G, i)$;
    while $w < n(G)$ **do**
        $H[edgecnt{+}{+}] \leftarrow (i, w, weight(G, i, w))$;
        $w \leftarrow next(G, i, w)$;
$HeapBottomUp(H)$;
$ds \leftarrow create\_disjointSubset(n(G))$;
$numMST \leftarrow n(G)$;
while $numMST > 1$ **do**
    $(v, u, wt) \leftarrow removemin(H)$;
    if $find(ds, v) \neq find(ds, u)$ **then**
        $union(ds, v, u)$;
        $setEdge(G', v, u, wt)$;
        $numMST{-}{-}$;

Note: the shown heap is a simplification









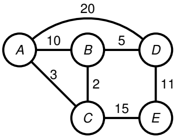
Centro de Informática UFPE

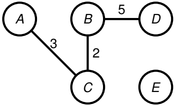
Universidade Federal de Pernambuco

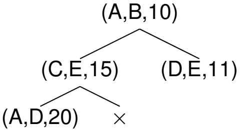
Kruskal's algorithm

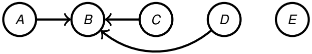
Algorithm: void Kruskal(Graph G, Graph G')

```
edgecnt ← 1;
for i ← 0 to n(G) − 1 do
    w ← first(G, i);
    while w < n(G) do
        H[edgecnt++] ←
          (i, w, weight(G, i, w));
        w ← next(G, i, w);
HeapBottomUp(H);
ds ← create_disjointSubset(n(G));
numMST ← n(G);
while numMST > 1 do
    (v, u, wt) ← removemin(H);
    if find(ds, v) ≠ find(ds, u) then
        union(ds, v, u);
        setEdge(G', v, u, wt);
        numMST--;
```







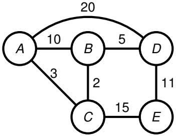


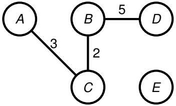
Note: the shown heap is a simplification

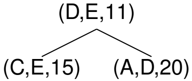
Kruskal's algorithm

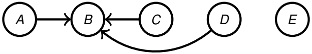
Algorithm: void Kruskal(Graph G, Graph G')

```
edgecnt ← 1;
for i ← 0 to n(G) − 1 do
    w ← first(G, i);
    while w < n(G) do
        H[edgecnt++] ←
          (i, w, weight(G, i, w));
        w ← next(G, i, w);
HeapBottomUp(H);
ds ← create_disjointSubset(n(G));
numMST ← n(G);
while numMST > 1 do
    (v, u, wt) ← removemin(H);
    if find(ds, v) ≠ find(ds, u) then
        union(ds, v, u);
        setEdge(G′, v, u, wt);
        numMST--;
```







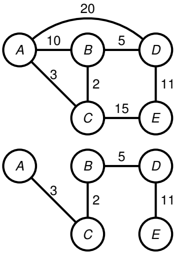


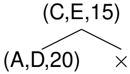
Note: the shown heap is a simplification

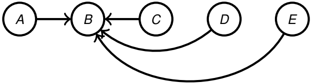
Kruskal's algorithm

Algorithm: void Kruskal(Graph G, Graph G')

```
edgecnt ← 1;
for i ← 0 to n(G) − 1 do
    w ← first(G, i);
    while w < n(G) do
        H[edgecnt++] ←
          (i, w, weight(G, i, w));
        w ← next(G, i, w);
HeapBottomUp(H);
ds ← create_disjointSubset(n(G));
numMST ← n(G);
while numMST > 1 do
    (v, u, wt) ← removemin(H);
    if find(ds, v) ≠ find(ds, u) then
        union(ds, v, u);
        setEdge(G', v, u, wt);
        numMST--;
```





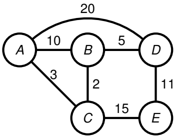


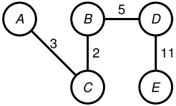
Note: the shown heap is a simplification

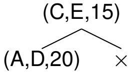
Kruskal's algorithm

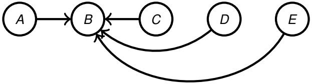
Algorithm: void Kruskal(Graph G, Graph G')

```
edgecnt ← 1;
for i ← 0 to n(G) − 1 do
    w ← first(G, i);
    while w < n(G) do
        H[edgecnt++] ←
          (i, w, weight(G, i, w));
        w ← next(G, i, w);
HeapBottomUp(H);
ds ← create_disjointSubset(n(G));
numMST ← n(G);
while numMST > 1 do
    (v, u, wt) ← removemin(H);
    if find(ds, v) ≠ find(ds, u) then
        union(ds, v, u);
        setEdge(G', v, u, wt);
        numMST--;
```









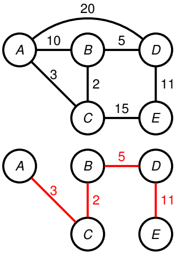
Note: the shown heap is a simplification

Kruskal's algorithm

Algorithm: void Kruskal(Graph G, Graph G')

$edgecnt \leftarrow 1$;
for $i \leftarrow 0$ **to** $n(G) - 1$ **do**
  $w \leftarrow first(G, i)$;
  while $w < n(G)$ **do**
    $H[edgecnt{+}{+}] \leftarrow (i, w, weight(G, i, w))$;
    $w \leftarrow next(G, i, w)$;
$HeapBottomUp(H)$;
$ds \leftarrow create\_disjointSubset(n(G))$;
$numMST \leftarrow n(G)$;
while $numMST > 1$ **do**
  $(v, u, wt) \leftarrow removemin(H)$;
  if $find(ds, v) \neq find(ds, u)$ **then**
    $union(ds, v, u)$;
    $setEdge(G', v, u, wt)$;
    $numMST{-}{-}$;

Note: the shown heap is a simplification

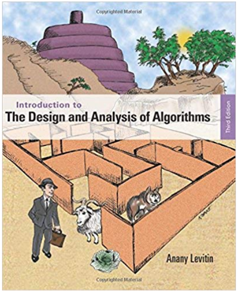


Agenda

Centro de Informática UFPE

Universidade Federal de Pernambuco

Bibliography



Chapter 9 (pp. 318–322, pp. 325–331)
Anany Levitin.
Introduction to the Design and Analysis of Algorithms.
3rd edition. Pearson. 2011.

Chapter 6 (pp. 199–206)
Chapter 11 (pp. 393 – 399)
Clifford Shaffer.
Data Structures and Algorithm Analysis. Dover, 2013.

Centro de Informática UFPE

Universidade Federal de Pernambuco

MINIMUM-COST SPANNING TREES

Gustavo Carvalho
(ghpc@cin.ufpe.br)

Universidade Federal de Pernambuco
Centro de Informática, 50740-560, Brazil