FACIT TILL OMTENTAMEN I DATASTRUKTURER OCH ALGORITMER DVG B03

140610 kl. 08:15 – 13:15

Ansvarig Lärare: Donald F. Ross

Hjälpmedel: Inga. Algoritmerna finns i de respektive uppgifterna.

***** OBS *****

Betygsgräns:

Kurs: Max 60p, Med beröm godkänd 50p, Icke utan beröm godkänd 40p, Godkänd 30p
(varav minimum 15p från tentamen, 15p från labbarna)
Tentamen: Max 30p, betyg 5: 26p-30p, betyg 4: 21p-25p, betyg 3: 15p-20p
Labbarna: Max 30p, betyg 5: 26p-30p, betyg 4: 21p-25p, betyg 3: 15p-20p

SKRIV TYDLIGT – LÄS UPPGIFTERNA NOGGRANT

Ange alla antaganden.

---

(1) Ge ett kortfattat svar till följande uppgifter ((a)-(j)).

(a) En definition av en mängd
Definition: An unordered collection of unique entities having a common property (attribute).
Example: Set of **colours**: {red, green, blue, yellow, ..., black}

(b) En definition av en sekvens
Definition: An ordered collection of entities (not necessarily unique) having a common property (attribute) with a successor relationship defined between entities.
Example: Days of the week: (Monday, Tuesday, ..., Sunday)

(c) En definition av ett träd
Definition: An unordered/ordered collection of entities (not necessarily unique) having a common property (attribute) with a descendant relationship defined between entities.
Example: File directories in computing systems

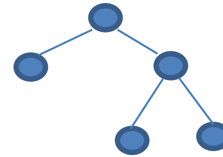
(d) En definition av en graf
Definition: An unordered collection of entities (not necessarily unique) having a common property (attribute) with a (general) relationship defined between entities.
Example: A Telephone / Computer network (in fact any network)

(e) En definition av en samling
Definition: An unordered/ordered number of entities (not necessarily unique) having a common property (attribute) possibly with a relationship defined between entities.
Example: set, sequence, tree, graph

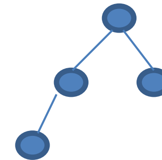
Förklara Dina exempel ((f)-(j)) med några meningar

FULL: **every node has exactly 2 or 0 children**
PERFECT: BT height **h** with exactly $\mathbf{2^h-1}$ elements
COMPLETE: **perfect** on the **next lowest level** AND the lowest level is filled **from the left**

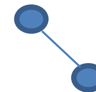
(f) Ett exempel av ett fullt träd som inte är komplett



(g) Ett exempel av ett komplett träd som inte är fullt



(h) Ett exempel av ett träd som är varken fullt eller komplett eller perfekt



(i) Ett exempel av implementationsabstraktion

Get and set functions on the attributes of an ADT

(j) Ett exempel av svansrekursion (tail recursion)

```
int: size (List)
{    if is_empty (List) return 0;
     return 1 + size(tail(List));
     }
```

Totalt 5p

(2) Ge ett kortfattat svar till följande uppgifter.

(a) En definition av rekursion

Definition 1: An entity *partially* defined in terms of itself is recursively defined
Definition 2: A function which calls itself *under certain conditions* is recursively defined.

(b) En förklaring av abstraktion

Definition 1: MODELLING ABSTRACTION
The process of selecting certain properties (attributes) of an entity to represent that entity in a given situation.

Example 1: In reality information about a student (a real world entity) may include properties (attributes) such as
STUDENT (name, date of birth, gender, address, pnumber, telephone number, mobile number, hair colour, eye colour, father's name, mother's name, etc. ....)

Some of this information can be "abstracted" as
STUDENT (name, date of birth, gender, address, pnumber)
in a database for example to "represent" (or model) a student

Definition 2: COLLECTION ABSTRACTION
The process of generalising common properties and operations of the set, sequence, tree and graph.
Each is a collection with the sequence, tree and graph also having relationships defined - sequence (successor/predecessor), tree (left child, right child), graph (a general relationship representing distance, cost, connection).

Example 2: A **collection** can be defined as a comprising entities and relationships, both with possible attributes and operations such as is_empty(), cardinality(), add(), remove(), find(), display().

Definition 3: IMPLEMENTATION ABSTRACTION
The process of selecting certain properties of a Data Type independent of the implementation of that Data Type - hence the expression Abstract Data Type

Example 3: A sequence may be implemented using

- an array of values and an index to that array
- arrays (value, next) and an index
- records/structures and pointers (a linked list)

In all cases, at a more abstract level, certain properties and operations apply in both implementations e.g. first, next, number of elements

In all cases we have a structure (array(s)/list) which represents a collection of elements and a reference (index/pointer) to an individual element in the collection.

(c) Principerna bakom hashning

Definition: The mapping of a value (often a key value) to a position in a linear space (hash space) by means of a hash function **h**.

h: key => position (a mapping from a key space to a hash space)

Issues:

- main advantage: **search time is O(1)** (i.e. constant)
- **collision handling** (i.e. if h(x) maps to the same position as h(y))
- the **distribution of values** in the key space
  if this is known, some form of optimisation may be applied
- the **load factor** in the hash space (see the animation below)
- the **number of probes** required (increases as the load factor increases)

(d) Principerna bakom Kruskals algoritm

Divide the graph into components (G') with one node in each component initially and a priority queue (PQ) for each edge (lowest cost first). Work through the PQ adding an edge to the graph (G') of nodes iff 2 distinct components are connected until the graph contains one component – the Minimal Spanning Tree (MST).

(e) Om man skulle lägga till sekvensen (1, 4, 22, 14, 34, 8, 44, 7) i ett hashutrymme (hash space) med 10 platser, vilken kollisionshanteringsmetod av linjär probning (linear probing) och kvadratisk probning (quadratic probing) är säkrare? **Förklara varför.**

Choose linear probing since quadratic probing cannot guarantee success if the load factor is above 50%. Here there are 8 elements and 10 places giving a load of 80%.

Totalt 5p

(3) Rekursion

Skriv (pseudo)kod till en **rekursiv funktion** (eller **två rekursiva funktioner**) för att räkna fram antalet kanter (edges) i en graf. Ange alla antagande.

5p

Assumptions:

Structure:

```
typedef struct nodeelem * noderef;

typedef struct nodeelem {
        char        nname;
        int         ninfo;
        noderef     edges;
        noderef     nodes;
        } nodeelem;
```

+ corresponding get/set functions per attribute and head/tail operations for both the node list (nodes) and the edge list (edges).

G is a reference to the graph, the is_empty(R) function is defined.

```
static int b_nedges(noderef E) {
  return is_empty(E) ? 0 : 1 + b_nedges(etail(E));
  }

static int b_esize(noderef G) {
  return is_empty(G) ? 0 : b_nedges(get_edges(nhead(G))) + b_esize(ntail(G));
  }
```

(4) Grafoperationer

En graf kan beskrivas som en virtuell (eller abstrakt) maskin. I graflaborationen, filen ”begraph.h” är faktiskt en beskrivning av denna virtuell maskin. Innehållet av ”begraph.h” ges nedan.

```
/*********************************************************************/
/* function prototypes - operations on the Graph (a virtual machine)     */
/*********************************************************************/
/* Graph = (V, E) where V is a set of vertices/nodes and E a set of edges */
/* There are a limited number of operations (9) which can be applied to G */
/*********************************************************************/
void be_display_adjlist();              /* display G as an adjacency list  */
void be_display_adjmatrix();            /* display G as an adjacency matrix*/

void be_addnode(char c);                /* add a vertex (node) to G        */
void be_remnode(char c);                /* remove a vertex (node) from G   */

void be_addedge(char cs, char cd, int v);/* add an edge (with weight) to G*/
void be_remedge(char cs, char cd);      /* remove an edge from G           */

int  be_is_nmember(char c);             /* is a  node a member of G?       */
int  be_is_emember(char cs, char cd);   /* is an edge a member of G?       */

int  be_size();                         /* the number of nodes in G        */
/*********************************************************************/
```

I **front-end:en** kan man specifiera förvillkor (preconditions) till varje operation ovan med hjälp av en eller flera av dessa 9 funktioner.

Ange förvillkoret/förvillkoren <u>till varje</u> av de <u>första åtta operationerna</u> ovan. Dvs i (i) fe_display_adjlist(), (ii) fe_display_adjmatrix(), (iii) fe_addnode(), (iv) fe_remnode(), (v) fe_addedge(), (vi) fe_remedge(), (vii) fe_is_nmember(), (viii) fe_is_emember(). Kom ihåg att ifrån front-end:en kan man utföra ett “samtal” med användaren för att begära olika värden samt skriva felmeddelande.

(5p)

(i) fe_display_adjlist()

```
void fe_display_adjlist()  {
  if (be_size()==0) ui_putGraphEmpty();
  else { ui_putTitleList(); be_display_adjlist(); }
  }
```

(ii) fe_display_adjmatrix()

```
void fe_display_adjmatrix()  {
  if (be_size()==0) ui_putGraphEmpty();
  else { ui_putTitleMatrix(); be_display_adjmatrix(); }
  }
```

(iii) fe_addnode()

```
void fe_addnode()  {

  char c;

  c = ui_getNode();
  if (be_is_nmember(c)) ui_putNodeError(c); else be_addnode(c);
  }
```

(iv) fe_remnode()

```
void fe_remnode()  {

  char c;

  if (be_size()==0) ui_putGraphEmpty();
  else {
    c = ui_getNode();
    if(!be_is_nmember(c)) ui_putNoNodeError(c);
    else be_remnode(c);
    }
  }
```

(v) fe_addedge()

```
void fe_addedge()  {

  char cs, cd; int w;

  if (be_size()==0) ui_putGraphEmpty();
  else {
    cs = ui_getNode();
    if (!be_is_nmember(cs)) ui_putNodeError(cs);
    else {
      cd = ui_getNode();
      if (!be_is_nmember(cd)) ui_putNodeError(cd);
      else if (be_is_emember(cs, cd)) ui_putEdgeError(cs, cd);
      else {
        w  = ui_getWeight();
        be_addedge(cs, cd, w);
        if (ui_isModeU() && (cd != cs)) be_addedge(cd, cs, w);
        }
      }
    }
  }
```

(vi) fe_remedge()

```
void fe_remedge()  {

  char cs, cd;

  if (be_size()==0) ui_putGraphEmpty();
  else {
    cs = ui_getNode();
    if (!be_is_nmember(cs)) ui_putNoNodeError(cs);
    else {
      cd = ui_getNode();
      if (!be_is_nmember(cd)) ui_putNoNodeError(cd);
      else if (!be_is_emember(cs, cd)) ui_putNoEdgeError(cs, cd);
      else {
        be_remedge(cs, cd);
        if (ui_isModeU()) {
          if (!be_is_emember(cd, cs)) ui_putNoEdgeError(cs, cd);
          else be_remedge(cd, cs);
          }
        }
      }
    }
  }
```

(vii) fe_is_nmember()

```
void fe_is_nmember() {
  char c;

  if (be_size()==0) ui_putGraphEmpty();
  else {
    c = ui_getNode();
    if (be_is_nmember(c)) ui_putNodeFound(c);
    else  ui_putNodeNotFound(c);
    }
  }
```

(viii) fe_is_emember()

```
void fe_is_emember() {
  char cs, cd;

  if (be_size()==0) ui_putGraphEmpty();
  else {
    cs = ui_getNode();
    if (!be_is_nmember(cs)) ui_putNoNodeError(cs);
    else {
      cd = ui_getNode();
      if (!be_is_nmember(cd)) ui_putNoNodeError(cd);
      else if (be_is_emember(cs, cd)) ui_putEdgeFound(cs, cd);
      else ui_putEdgeNotFound(cs, cd);
      }
    }
  }
```

(5) Prims algoritm

a) Tillämpa **Prims algoritm** (nedan) på den **oriktade** grafen:

(a-3-b, a-3-c, a-3-d, b-3-c, b-3-e, c-3-d, c-3-e, c-3-f, d-3-f, e-3-f).

Börja med nod a. Ange alla antaganden och visa alla beräkningar och mellanresultat. Vad representerar resultatet? **Ange varje steg i din beräkning.** Anta att noderna lagras i ”närlistan” (adjacency list) i alfabetisk ordning samt att man söker efter det minimala värdet i alfabetisk ordning.

Rita delresultatet efter varje iteration.

(3p)

b) Förklara principerna bakom Prims algoritm.

(2p)

Totalt 5p

Ange *alla* antaganden och visa *alla* beräkningar och mellanresultat

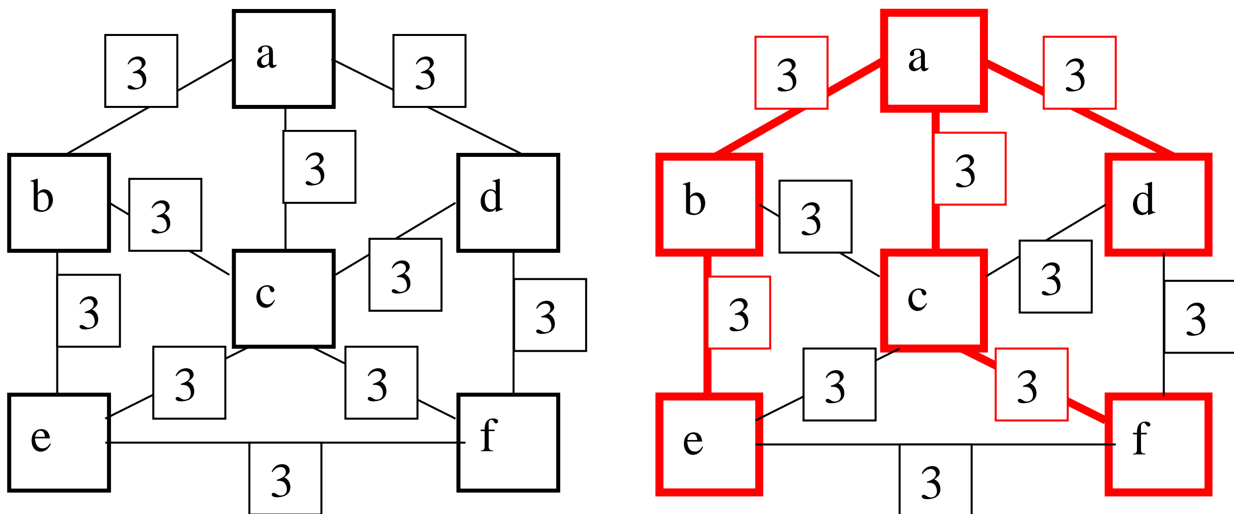
```
Prim ( node v) -- v is the start node

{   U = {v}; for i in (V-U) { low-cost[i] = C[v,i]; closest[i] = v; }

    while (!is_empty (V-U) ) {
                i = first(V-U); min = low-cost[i]; k = i;
                 for j in (V-U-k) if (low-cost[j] < min) { min = low-cost[j]; k = j; }
                display(k, closest[k]);
                U = U + k;
                for j in (V-U) if ( C[k,j] < low-cost[j] ) ) { low-cost[j] = C[k,j]; closest[j] = k; }
    }
}
```

The principle is that the MST ” grows” from the one component (here ”a”) by connecting this component to any other component (a node) by the cheapest edge SO FAR found – this last proviso reveals that Prim’s is a GREEDY algorithm i.e. used a local best solution.

See below for the calculations.

Draw the graph (and possibly sketch the answer – use Kruskalls for a quick check!):

Cost 15

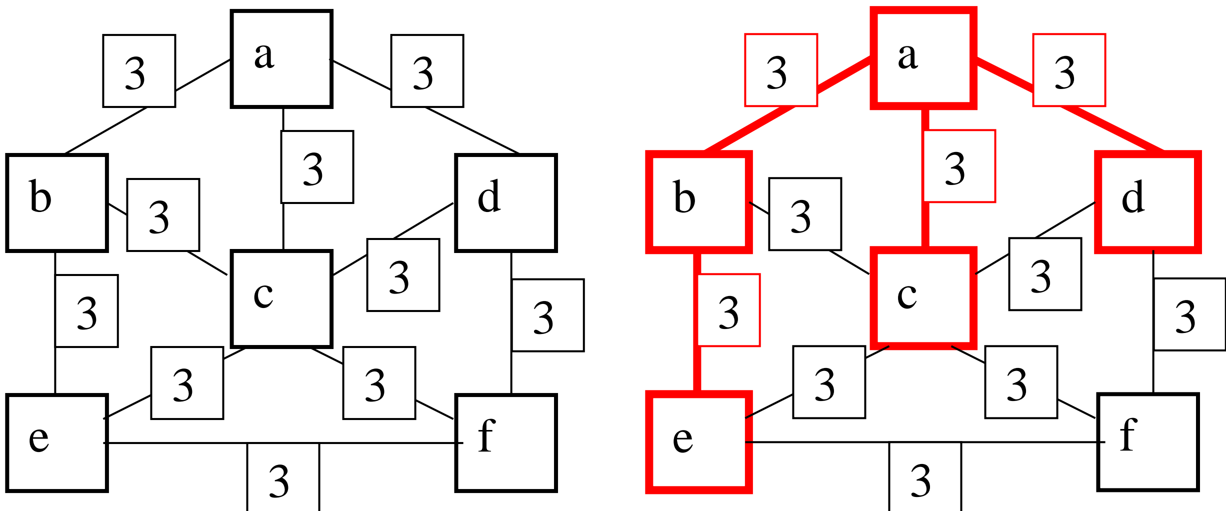


Draw the cost matrix C and array D

	a	**b**	**c**	**d**	**e**	**f**
a		**3**	**3**	**3**		
b	**3**		**3**		**3**	
c	**3**	**3**		**3**	**3**	**3**
d	**3**		**3**			**3**
e		**3**	**3**			**3**
f			**3**	**3**	**3**	

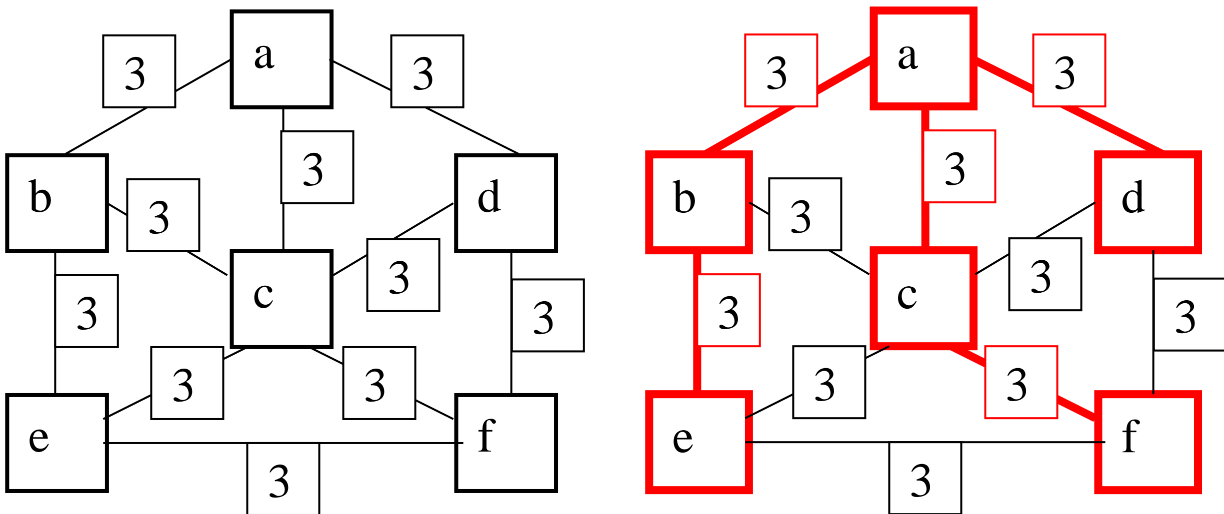
a	**b**	**c**	**d**	**e**	**f**
lowcost	**3**	**3**	**3**	**§**	**§**
closest	**a**	**a**	**a**	**a**	**a**

Min edge**: lowcost: 3 3 3 § § --- closest: a a a a a ---** U = {a,b} V-U = {c,d,e,f} min = 3; k = b
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = c; if C[b,c] < lowcost[c] then { lowcost[c] = C[b,c]; closest[c] = b } → 3<3 ➔ no change
j = d; if C[b,d] < lowcost[d] then { lowcost[d] = C[b,d]; closest[d] = b } → §<3 ➔ no change
j = e; if C[b,e] < lowcost[e] then { lowcost[e] = C[b,e]; **closest[e] = b** } → **3<§ ➔ b-3-e**
j = f; if C[b,f] < lowcost[f] then { lowcost[f] = C[b,f]; closest[f] = b } → §<§ ➔ no change



a	**b**	**c**	**d**	**e**	**f**
lowcost	**3**	**3**	**3**	**3**	**§**
closest	**a**	**a**	**a**	**b**	**a**

Min edge: **lowcost: 1 3 3 3 § --- closest: a a a b a ---** U = {a,b,c} V-U = {d,e,f} min = 3; k = c
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = d; if C[c,d] < lowcost[d] then { lowcost[d] = C[c,d]; closest[b] = c } → 3<3 → no change
j = e; if C[c,e] < lowcost[e] then { lowcost[e] = C[c,e]; closest[d] = c} → 3<3 → no change
j = f; if C[c,f] < lowcost[f] then { lowcost[f] = C[c,f]; **closest[e] = c** } → **3<§ → c-3-f**



a	b	c	d	e	f
lowcost	**3**	**3**	**3**	**3**	**3**
closest	**a**	**a**	**a**	**b**	**c**

Min edge: **lowcost: 3 3 3 3 3 --- closest: a a a b c ---** U = {a,c,b,d} V-U = {e,f} min = 3; k = d
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = e; if C[d,e] < lowcost[e] then { lowcost[e] = C[d,e]; closest[e] = d } → §<3 → no change
j = f; if C[d,f] < lowcost[f] then { lowcost[f] = C[d,f]; closest[f] = d } → 3<3 → no change

Min edge: **lowcost: 3 3 3 3 3 --- closest: a a a b c ---** U = {a,c,b,d,e} V-U = {f} min = 3; k = e
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = f; if C[e,f] < lowcost[f] then { lowcost[f] = C[e,f]; closest[f] = 3 } → 3<3 → no change

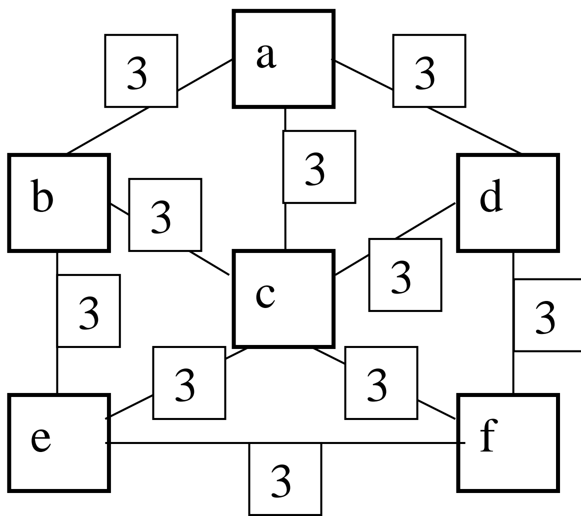
Finally add the remaining node – node e (there are no further calculations)
Min edge: **lowcost: 3 3 3 3 3 --- closest: a a a b c ---** U = {a,c,b,d,f e} V-U = {¤}

QED ☺ MST edges **a-3-b, a-3-c, a-3-d, b-3-e, c-3-f** **Total cost = 15**

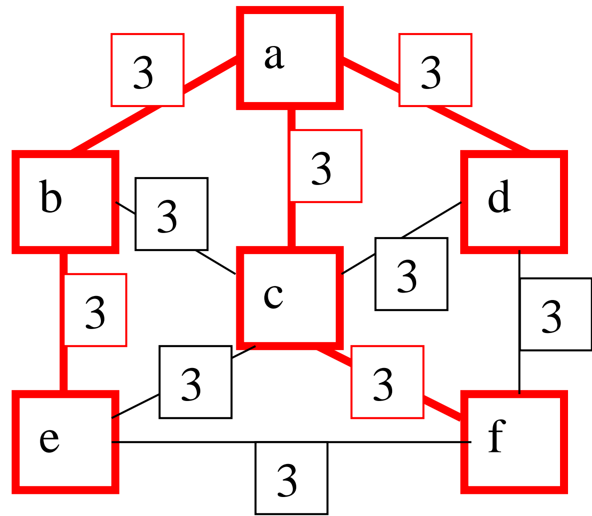
(Confirm using Kruskal's)

Principle: to build the MST from a single component by choosing the cheapest **edge** to non-component nodes from the last node added. Above start with a, add **edge** distances (infinite if no edge), choose the cheapest (a-3-c) and add this to the component. Now recheck if there are cheaper **edges** from c to the non-component nodes. Repeat until all the nodes are connected. So the component develops as (a), (a-3-c), (a-3-c, c-b-1, c-4-d, c-5-e, d-3-f) (see above).

Start Graph



Solution



a	b	c	d	e	f
lowcost	**3**	**3**	**3**	**3**	**3**
closest	**a**	**a**	**a**	**b**	**c**

(6) Dijkstra + SPT (Shortest Path Tree)

Tillämpa **den givna Dijkstra_SPT algoritmen (nedan)** på **den riktade grafen**,

(a, b, 13), (a, d, 12), (a, e, 20), (b, c, 60), (c, e, 50), (d, c, 30), (d, e, 40)

SPT = Shortest Path Tree - dvs kortaste väg trädet (KVT) från en nod till alla de andra.

Börja med nod "a".
Visa varje steg i dina beräkningar.
Ange *alla* antaganden och visa *alla* beräkningar och mellanresultat
Rita **varje steg** i konstruktionen av SPT:et – dvs visa till och med de noder och kanter som läggs till men sedan tas bort.

(3p)

Förklara **principerna** bakom **Dijkstras_SPT** algoritm.

(2p)

Totalt 5p

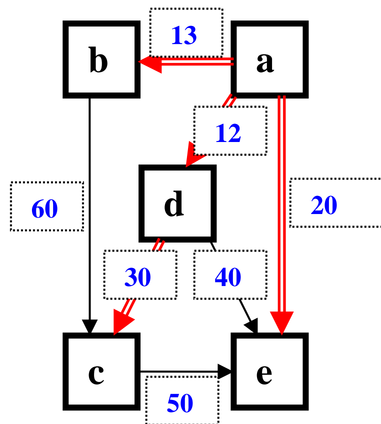
Dijkstras algoritm med en utökning för SPT

```
Dijkstra_SPT ( a )
{
     S = {a}

         for (i in V-S) {
           D[i]    = C[a, i]                    --- initialise D - (edge cost)
           E[i]    = a                          --- initialise E - SPT (edge)
           L[i]    = C[a, i]                    --- initialise L - SPT (cost)
         }

         for (i in 1..(|V|-1)) {
                  choose w in V-S such that D[w] is a minimum
                  S = S + {w}
                  foreach ( v in V-S )  if (D[w] + C[w,v] < D[v]) {
                            D[v] = D[w] + C[w,v]
                            E[v] = w
                            L[v] = C[w,v]
                   }
         }
}
```

Initialise D, E, L

D: ¤ 13 § 12 20
E: ¤ a a a a
L: ¤ 13 § 12 20
w is d (min value in D) S = {a,d} V-S = {b,c,e}
v = b min (D[b], D[d]+C (d,b)) ➔ min(13, 12+§) → **no change**
v = c min (D[c], D[d]+C (d,c)) ➔ min(§, 12+30) → **a-d-c 42**
v = e min (D[e], D[d]+C (d,e)) ➔ min(20, 12+40) → **no change**

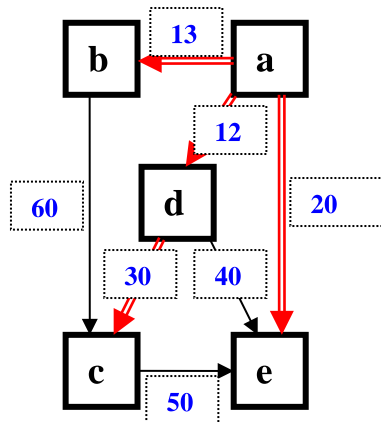
D: ¤ 13 42 12 20
E: ¤ a d a a
L: ¤ 13 30 12 20



D: ¤ 13 42 12 20
E: ¤ a d a a
L: ¤ 13 30 12 20

w is b (min value in D) S = {a,d,b} V-S = {c,e}
v = c min (D[c], D[b]+C (b,c)) ➔ min(42, 13+60) → **no change**
v = e min (D[e], D[b]+C (b,e)) ➔ min(20, 13+§) → **no change**

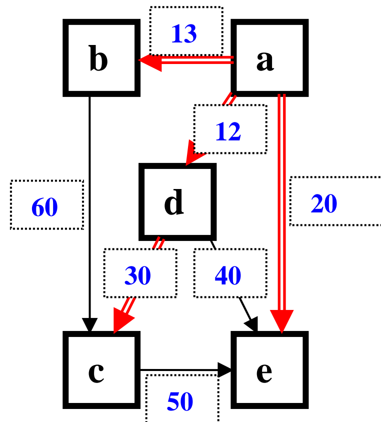
D: ¤ 13 42 12 20
E: ¤ a d a a
L: ¤ 13 30 12 20



D: ¤ 13 42 12 20
E: ¤ a d a a
L: ¤ 13 30 12 20

v = c min (D[c], D[e]+C (e,c)) w is e (min value in D) S = {a,d,b,e} V-S = {c}
➔ min(42, 20+§) → **no change**

D: ¤ 13 42 12 20
E: ¤ a d a a
L: ¤ 13 30 12 20



This is the final result.

Costs: a→b (13), a→d→c (42), a→d (12), a→e (20)

SPT edges: a→b (13), a→d (12), d→c (30), a→e (20)

Principle – similar to Prims i.e build a component step by step ➔ SPT