FACIT TO
Ordinary Exam
DATA STRUCTURES AND ALGORITHMS DVG B03

130115 08:15 – 13:15

Course Coordinator: Donald F. Ross

Help Material: A Dictionary from the student's home language to English.

***** OBS *****

Students who have studied the course as from (>=) Autumn Term 2006

Grading Levels:

Course:	Max 60p, pass with special distinction 50p, pass with distinction 40p, pass 30p **(of which a minimum 15p from the exam, 15p from the labs)**
Exam:	Max 30p, grade 5: 26p-30p, grade 4: 21p-25p, grade 3: 15p-20p
Labs:	Max 30p, grade 5: 26p-30p, grade 4: 21p-25p, grade 3: 15p-20p

Students who have studied the course before (<) Autumn Term 2006

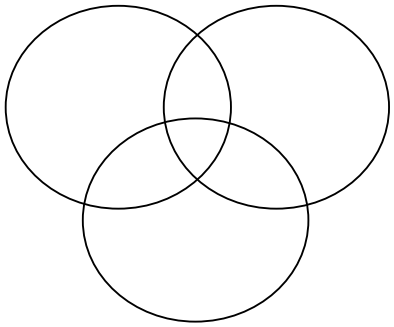
Grading Levels:

Course:	Max 60p, pass with special distinction 50p, pass with distinction 40p, pass 30p **(of which a minimum 20p from the exam, 10p from the labs)**
Exam:	Max 40p, grade 5: 34p-40p, grade 4: 27p-33p, grade 3: 20p-26p
Labs:	Max 20p, grade 5: 18p-20p, grade 4: 14p-17p, grade 3: 10p-13p

Write legibly – read all questions carefully

(1) Give a short answer to each of the following questions:-

(a) Show the relationship between a binary search tree, a complete tree and a full tree with the help of a Venn-diagram from set theory.



(b) What is the maximum load factor in hashing with quadratic probing as the collision management technique which guarantees that a space can always be found?

50% **see lecture notes on Hashing slide 9**

(c) What is ”big-O”?

An indication of an algorithm’s worst case performance usually with respect to the number of items processed, n.

(d) What does Warshall’s algorithm do?

Calculates the transitive closure of a graph i.e. determines if there is a path from any node A to any node B (A and B not necessarily distinct)

(e) Give a definition of an AVL-tree.

An AVL-tree is a BST where the height difference between the left sub-tree and the right sub-tree is no more than 1. |height(LC) – height(RC)| <= 1

A BST (Binary Search Tree) is a binary tree where all the values in the LC are less than the value at the root and all the values in the RC are greater than the value at the root.

(f) Which grammatical class in natural language corresponds to a relation in the Entity Relationship model?

Verb. **Entity = noun, attribute = adjective**

(g) Which algorithm requires a DAG as a starting point?

Arguable all tree algorithms since a tree can be viewed as a DAG.

Topological sort on a DAG (whether it e a tree or graph)

(h) Which data structures are ordered?

Sequence, ordered trees for example a binary tree – and by extension BST and AVL

B-Tree

(i) Which algorithm is $O(n^3)$?

Warshall's, Floyd's (sometimes known as Floyd-Warshall's).

(j) Give a definition of a recursive function.

A function which <u>conditionally</u> calls itself.

Total 5p

(2) Abstraction

Discuss in detail why abstraction is so important in data structures. Are there any possible disadvantages to abstraction? Which 3 definitions of the term abstraction have we used in this course?

5p

- **Modelling abstraction: real world ➔ model (entities + attributes + relationships)**
- **Collections (sets) of entities: abstraction of sequences, trees & graphs (these add relationships to the set)**
- **Implementation abstraction: ADTs (sets, sequences, trees & graphs)**
- **Programming abstraction: UI – Front End – Back End model**

Points to note in your answer – marks for interesting points:

- By abstracting to produce ADTs, basic operations can be defined and the ADT applied to a range of concrete examples – for e.g.
  - The abstract data structure SEQUENCE may be used on a list of elements, a sequence of characters (text editing, parsing, pattern matching) or as a basis for other ADTs such as trees and graphs
- Abstracting allow us to focus on general aspects which are not dependent on the underlying implementation or programming language – for e.g. general sequence operations may be implemented using arrays + indexes OR structures + references.
- Using ADTs, systems may be designed on paper before actually being implemented as a programmed system
- Levels of abstraction may also be used, the most general being a collection (set), with operations is_empty, cardinality, add_element, remove_element, find_element, display collection.
- At the next level down we may have (abstract) sequences, tree and graphs – more specific operations may be added here – for e.g. with a binary tree left_child, right_child – where the tree is still an abstraction (and the implementation details hidden)
- Abstraction provides a mental (conceptual) tool useful for programming
  Abstraction provides a mental (conceptual) tool useful for creating a "tool kit" for programming – it is easier to map the solution to a problem to a small number of known techniques than to reinvent the wheel!

Definitions used in this course

1. modeling abstraction – abstracting entities and attributes from the real world to model as ADTs in a program
2. collection abstraction – abstracting the common properties of the ADTs (set, sequence, tree, graph) to an abstract collection
3. implementation abstraction – an ADT is considered independent of its implementation (usually using either pointers and structures OR arrays and indexes)

I) Modelling

- Extract “entities” which have “attributes” and possible “relationships” to other entities. E-R Model (Chen 1970)
- Allows non-essential details (from the real world) to be removed to allow a simplified model to be produced (+ve)
- If values can be attached to the attributes of an entity then computer models may be more easily defined and implemented (+ve)
- Potential loss of information (-ve)
- The model depends on the assumptions made and it must be clear what these are (-ve)

II) ADTs

- Allows programming in the abstract i.e. implementation and language independent (+ve)
- The essential properties of and operations on data structures may be discussed independently of ant programming language features (+ve)
- ADTs may be viewed as collections of entities (and thus connected to modelling) with common operations such as
    - Create and entity/collection
    - Add / Remove an entity to / from a collection
    - Find entities in the collection - search
    - Count the number of entities in the collection (cardinality)
    - Define predicates such as “is_empty” for the collection (i.e. cardinality = 0)
- Provides a “mental toolkit” for programmers (+ve)
- For (beginner) programmers this may make operations harder to follow since there is no reference to a specific programming language (-ve)

(3) Sequence

(a) Give a recursive definition of a sequence.

(1p)

S ::= H T | ¤ H- head, T tail, ¤ empty
H ::= element
T ::= S

(b) From your definition in (a) above, write recursive pseudo code to count up the number of elements in a sequence
State all assumptions.
Give an example of how your code works.

(2p)

```
   int card(listref L) {
   return is_empty(L) ? 0 : 1 + card(tail(L)); }
OR
   int card(listref L) {
         if is_empty(L) return 0;
         else return 1 + card(tail(L));
         }
```

(c) From your definition in (a) above, write recursive pseudo code to add an element to a sequence in increasing (sorted) order. Assume that a function called Cons which adds an element to the head of the list already exists.
Cons: element x list ➔ list.
State all assumptions.
Give an example of how your code works.

(2p)

```
   listref cons(listref e, listref L) {
         return set_tail(e, L);
         }

   listref b_add(int v, listref L)
   {
     return is_empty(L) ? create_e(v)
            : v < get_value(head(L)) ?
         cons(create_e(v), L)
            : cons(head(L), b_add(v, tail(L)));
      }
OR
   listref b_add(int v, listref L)
   {
         if (is_empty(L)) return create_e(v);
         else if (v < get_value(head(L)))
                  return cons(create_e(v), L);
         else return cons(head(L), b_add(v, tail(L)));
         }
```

Total 5p

(4) General questions – give a detailed answer with an example

(a) What is double hashing?

In hashing, there is a hash function which maps a key value to an index in the hash space H: key → index.

If/when collisions occur a collision resolution technique is required to find a new (empty) space in the hash space. This is written as H(key) + f(i) where "i" is the number of the collision – 1st, 2nd, 3rd, etc.

For linear probing f(i) = i
For quadratic probing f(i) = i*i
For double hashing f(i) = i * H'(key) which gives H(key) + i*H'(key) hence double hashing (H and H')

Example H'(key) = R – (key mod R) where R is prime and < size(Hash Table)

So for H(key) = key mod 10 and H´ could be 7 – (key mod 7)

So 4 36 44 5 7 64 24 would map to the hash table (0-9) as 6 ¤ ¤ ¤ 4 5 36 7 24 44

Show the calculations in your answer.

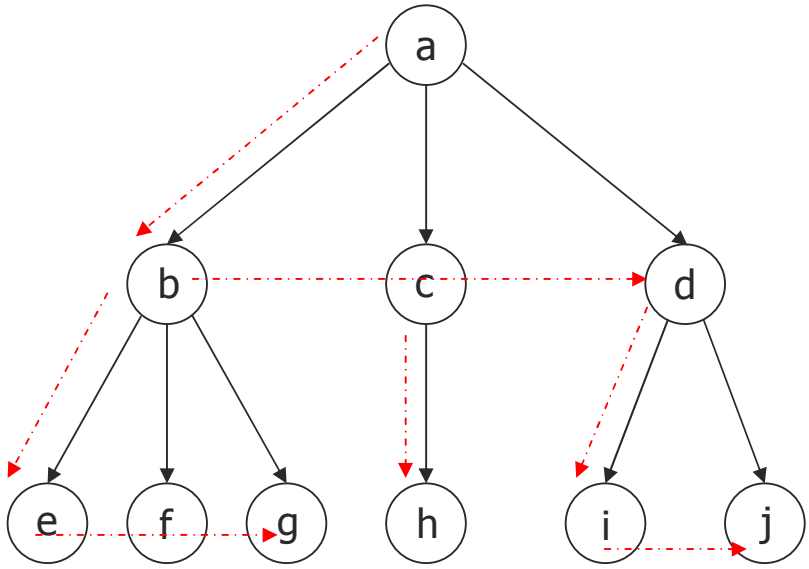
(b) What is a stack used for?

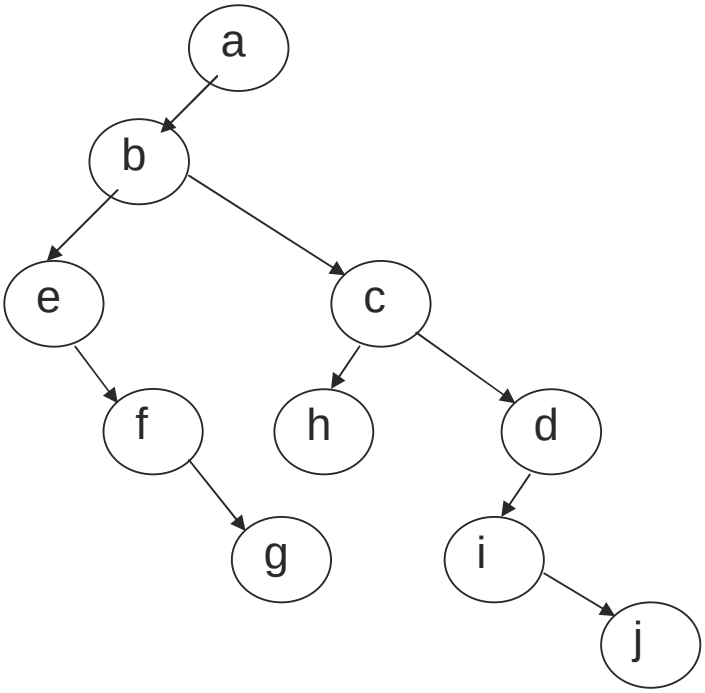
- As a memory – for e.g. to remember previous states, navigation through a maze
- To reorder (say) a sequence e.g. in infix to postfix transformation of arithmetic expressions 4*(2+2) ➔ 4 2 2 + *
- To execute postfix expressions such as 4 2 2 + * on a stack machine – push 4 (stack 4) push 2 (stack 4 2) push 2 (stack 4 2 2) apply + (stack 4 4) apply * (stack 16)
- To maintain the stack frames on a run-time stack for function/procedure calls in a programming language

(c) How is a general tree transformed to a binary tree?

1. The **first child** becomes the **left child** of the parent
2. The **subsequent children** become the **right child** of their **predecessor**

Draw an example and show how these rules are applied





(d) How does the add function work in a heap?

```
Add(H, v)
   let A = H.array
   A.size++
   i = A.size
   while i > 1 and A[Parent(i)] < v
            do       A[i] = A[Parent(i)]
                     i = Parent(i)
     end while
   A[i] = v
end Add
```

Draw an example stepwise to illustrate your point. See also the demonstration on the website.

(e) Describe a solution to the TSP-problem. TSP = Travelling Salesman Problem.

- **TSP Algorithm**
  - **variant of Kruskal's**
  - **edge acceptance conditions**
    - **degree(v) should not >= 3**
    - **no cycles unless # selected edges = |V|**
    - **greedy / near-optimal**

+ worked example to illustrate your point.

Total 5p

(5) Graph – Dijkstra + SPT

Extend Dijkstra's algorithm below in order to save and show the SPT.
(SPT: Shortest Path Tree).

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

        for ( i in 2..n) D[i] = C[a, i]              -- initialise D

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

(3p)

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]
                        }
   }
}
```

Apply your **extended version of the algorithm** to the following **directed graph**:-

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

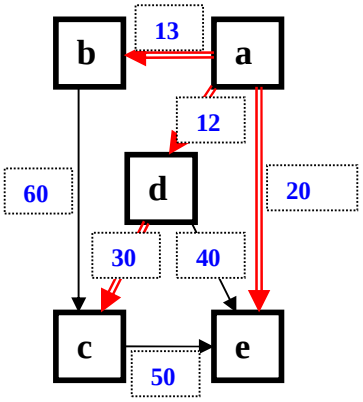
Start at node "a".
Show each step in your calculation.
State all assumptions and show all calculations and intermediate results.
Draw **each step** in the construction of the STP– i.e. show the nodes and which edges are added and subsequently removed.

(2p)

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

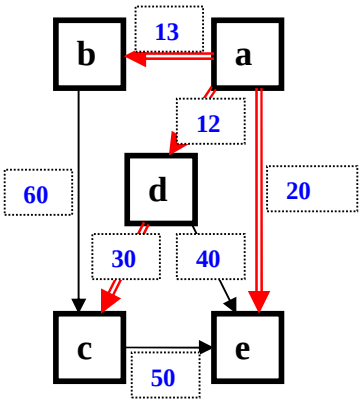


D: ¤ 13 42 12 20
E: ¤ a d a a
L: ¤ 13 30 12 20

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

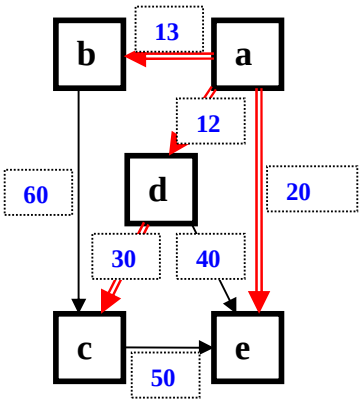


D: ¤ 13 42 12 20
E: ¤ a d a a
L: ¤ 13 30 12 20

w is e (min value in D) S = {a,d,b,e} V-S = {c}

v = c min (D[c], D[e]+C (e,c)) ➔ min(42, 20+§) → **no change**

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

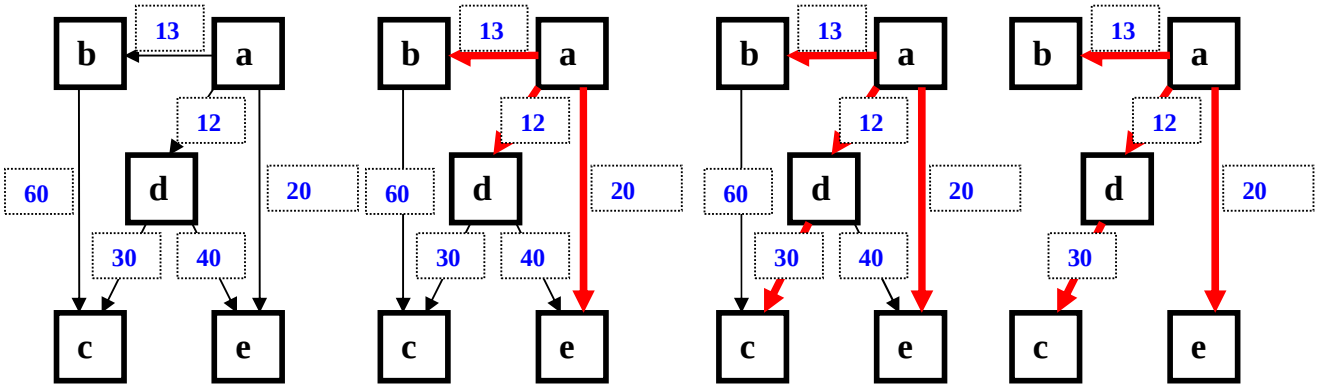


This is the final result.

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



Total 5p

(6) Graph – Prim's algorithm

Apply **the version of Prim's algorithm given below** to the undirected graph:-

(a-6-b, a-3-c, a-7-d, b-1-c, b-12-e, c-4-d, c-5-e, c-9-f, d-3-f, e-8-f).

Start with node "a".

State all assumptions and show all calculations and intermediate results.

(3p)

Explain **the principles** behind Prim's algorithm.
Use the example above in your explanation.

(2p)

```
1.  Prim ( node v)  {     -- v is the start node
2.  U = {v}; for i in (V-U) { low-cost[i] = C[v,i]; closest[i] = v; }
3.  while (!is_empty (V-U) ) {
4.      i = first(V-U); min = low-cost[i]; k = i;
5.      for j in (V-U-k) if (low-cost[j] < min) {min = low-cost[j]; k = j;  }
6.     display(k, closest[k]);
7.     U = U + k
8.     for j in (V-U) if ( C[k,j] < low-cost[j] ) )   {low-cost[j] = C[k,j]; closest[j] = k; }
9.     }
10. }
```

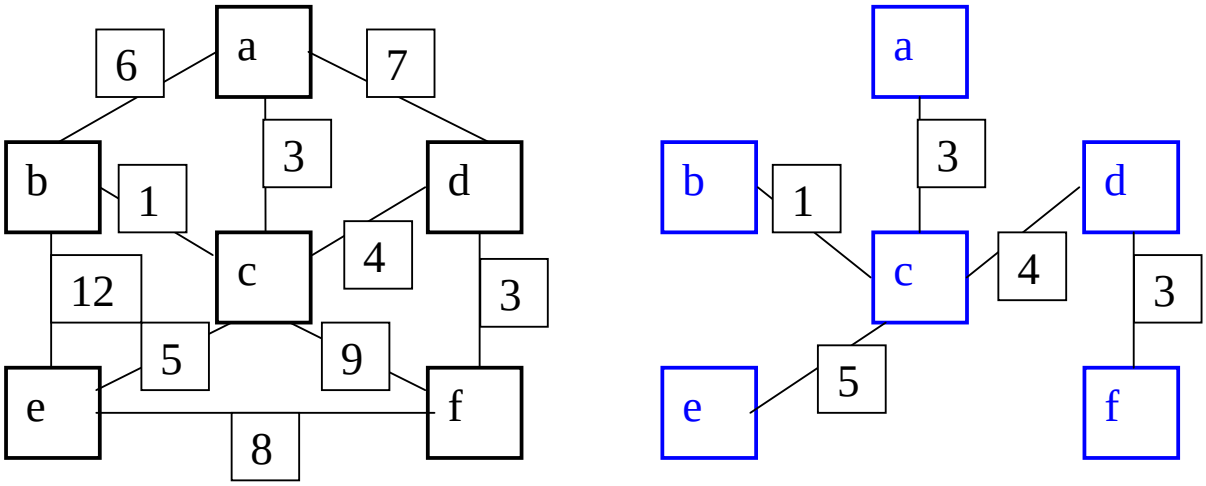
Total 5p

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 shortest direct of indirect path SO FAR CALCULATED – 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 16



Draw the cost matrix C and array D

	a	b	c	d	e	f
a		**6**	**3**	**7**		
b	**6**		**1**		**12**	
c	**3**	**1**		**4**	**5**	**9**
d	**7**		**4**			**3**
e		**12**	**5**			**8**
f			**9**	**3**	**8**	

a	**b**	**c**	**d**	**e**	**f**
lowcost	**6**	**3**	**7**	**§**	**§**
closest	**a**	**a**	**a**	**a**	**a**

Min edge**: lowcost: 6 3 7 § § --- closest: a a a a a ---** U = {a,c} V-U = {b,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 = b; if C[c,b] < lowcost[b] then { lowcost[b] = C[c,b]; closest[b] = c } → 1<6 ➔ **c-1-b**
j = d; if C[c,d] < lowcost[d] then { lowcost[d] = C[c,d]; closest[d] = c } → 4<7 ➔ **c-4-d**
j = e; if C[c,e] < lowcost[e] then { lowcost[e] = C[c,e]; closest[e] = c } → 5<§ ➔ **c-5-e**
j = f; if C[c,f] < lowcost[f] then { lowcost[f] = C[c,f]; closest[f] = c } → 9<§ ➔ **c-9-f**

Min edge**: lowcost: 1 3 4 5 9 --- closest: c a c c c ---** U = {a,c,b} V-U = {d,e,f} min = 1; k = b
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = d; if C[b,d] < lowcost[d] then { lowcost[d] = C[b,d]; closest[b] = b } → §<4 → no change
j = e; if C[b,e] < lowcost[e] then { lowcost[d] = b} → 12<5 → no change
j = f; if C[b,f] < lowcost[f] then { lowcost[f] = C[b,f]; closest[e] = b } → §<9 → no change

Min edge**: lowcost: 1 3 4 5 9 --- closest: c a c c c ---** U = {a,c,b,d} V-U = {e,f} min = 4; k = d
j = e; if C[d,e] < lowcost[b] then { lowcost[e] = C[d,e]; closest[e] = d } → §<5 → no change
j = f; if C[d,f] < lowcost[e] then { lowcost[f] = C[d,f]; closest[f] = d } → 3<9 ➔ **d-3-f**

Min edge**: lowcost: 1 3 4 5 3 --- closest: c a c c d ---** U = {a,c,b,d,f} V-U = {e} min = 3; k = f
j = e; if C[f,e] < lowcost[e] then { lowcost[e] = C[f,e]; closest[e] = f } → 8<5 → no change

Finally add the remaining node – node e (there are no further calculations)
Min edge**: lowcost: 1 3 4 5 3 --- closest: c a c c d ---** U = {a,c,b,d,f e} V-U = {¤}

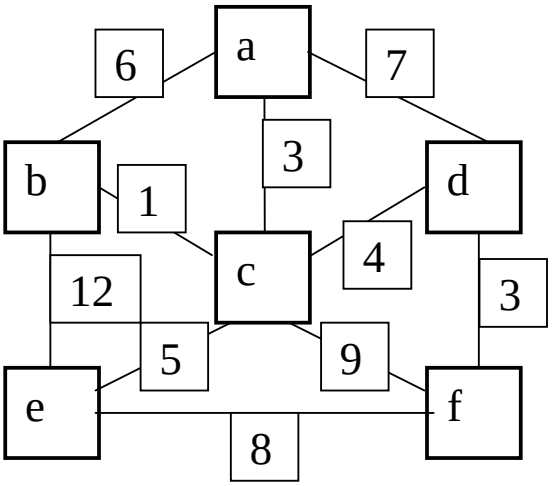
QED ☺ MST edges **c-1-b, a-3-c, c-4-d, c-5-e, d-3-f** **Total cost = 16**

(Confirm using Kruskal's)

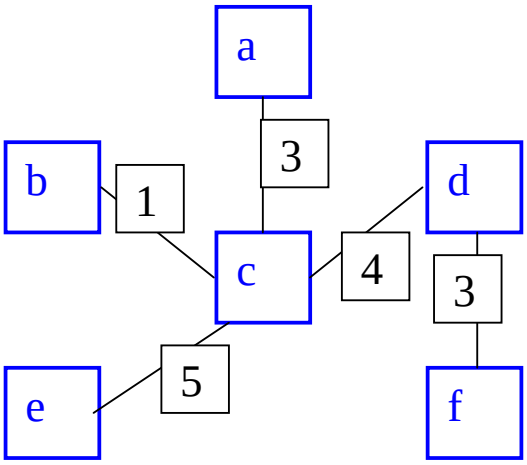
Principle: to build the MST from a single component by choosing the cheapest **edge** to non-component nodes from the last node added. Above start with a, add **edge** distances (infinite if no edge), choose the cheapest (a-3-c) and add this to the component. Now recheck if there are cheaper **edges** 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).

See below for the example in the question.

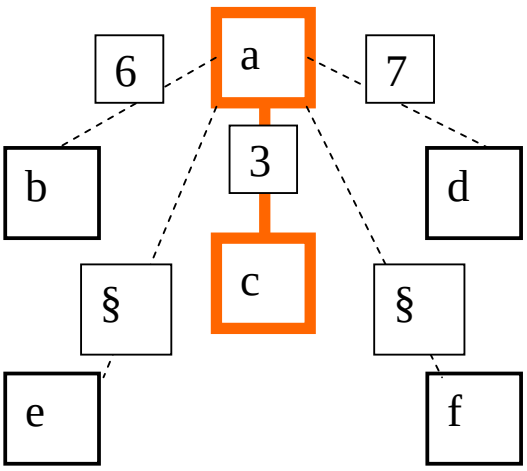
Start Graph

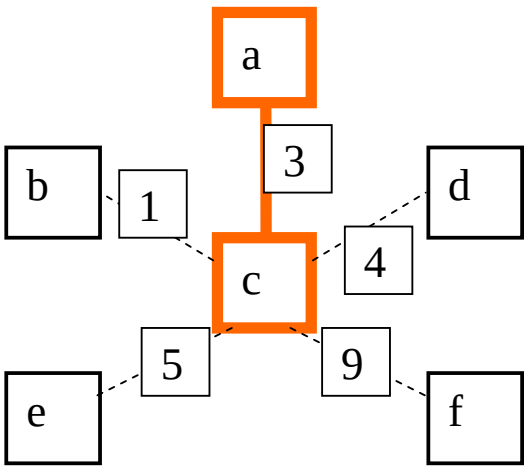


Solution



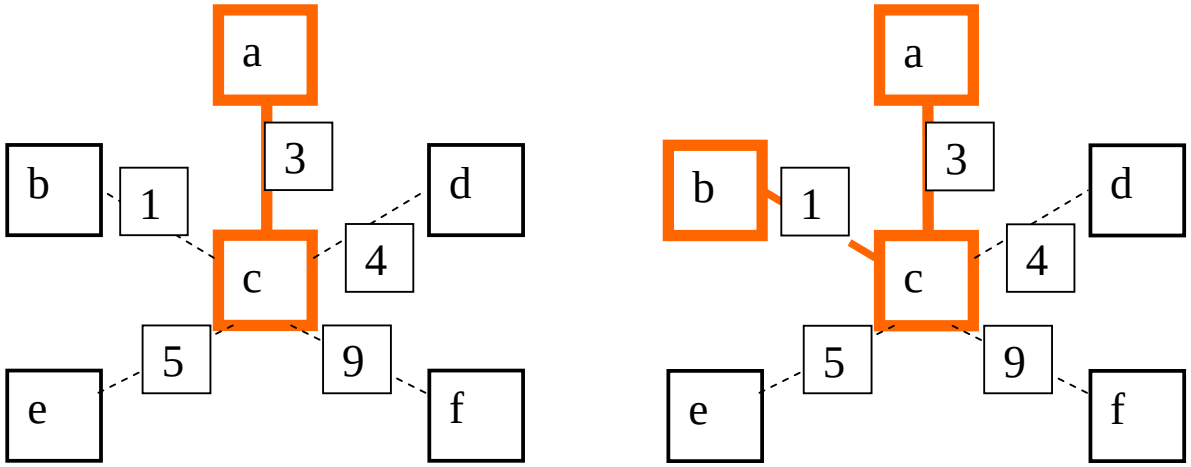
Start with node a – a is the component
Calculate the edges distances from a to the remaining nodes: no edge=infinity (§)
Calculate the shortest edge (lines 3-4 of the algorithm) – a-3-c
Add node c to the component (line 7 of the algorithm)
Recalculate the edge distances from c to (b,d,e,f) to see if there is a shorter edge than so far calculated (line 8 of the algorithm)





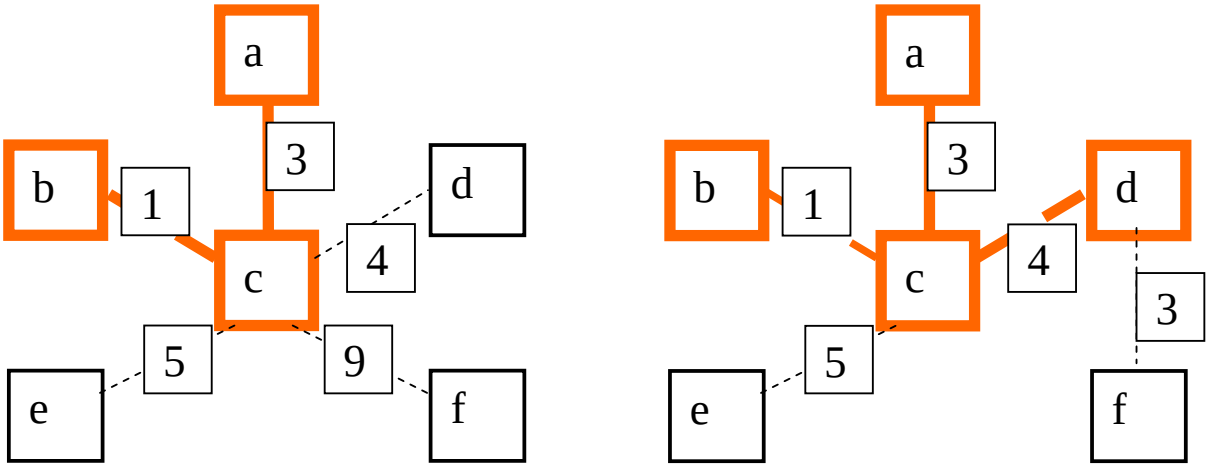
Min edge**: lowcost: 6 3 7 § § --- closest: a a a a a ---** U = {a,c} V-U = {b,d,e,f} min = 3; k = c
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = b; if C[c,b] < lowcost[b] then { lowcost[b] = C[c,b]; closest[b] = c } → 1<6 ➔ **c-1-b**
j = d; if C[c,d] < lowcost[d] then { lowcost[d] = C[c,d]; closest[d] = c } → 4<7 ➔ **c-4-d**
j = e; if C[c,e] < lowcost[e] then { lowcost[e] = C[c,e]; closest[e] = c } → 5<§ ➔ **c-5-e**
j = f; if C[c,f] < lowcost[f] then { lowcost[f] = C[c,f]; closest[f] = c } → 9<§ ➔ **c-9-f**
Result after this iteration: lowcost: 1 3 4 5 9 --- closest: c a c c c

Continue with node c – c is the new node in the component
Calculate the shortest edge (lines 3-4 of the algorithm) – c-1-b
Add node b to the component (line 7 of the algorithm)
Recalculate the edge distances from b to (d,e,f) to see if there is a shorter edge than so far calculated (line 8 of the algorithm)



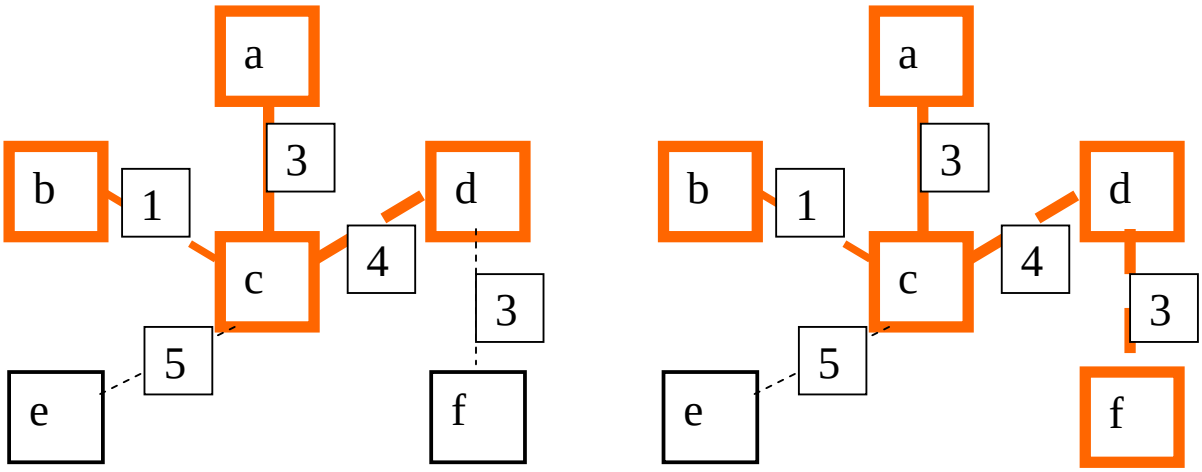
Min edge**: lowcost: 1 3 4 5 9 --- closest: c a c c c ---** U = {a,c,b} V-U = {d,e,f} min = 1; k = b
Readjust costs: if C[k,j] < lowcost[j] then { lowcost[j] = C[k,j]; closest[j] = k }
j = d; if C[b,d] < lowcost[d] then { lowcost[d] = C[b,d]; closest[b] = b } → §<4 → no change
j = e; if C[b,e] < lowcost[e] then { lowcost[e] = C[b,e]; closest[d] = b} → 12<5 → no change
j = f; if C[b,f] < lowcost[f] then { lowcost[f] = C[b,f]; closest[e] = b } → §<9 → no change
Result after this iteration: lowcost: 1 3 4 5 9 --- closest: c a c c c – NO CHANGE

Now look for the closest non-component node to the component – node d
Calculate the shortest edge (lines 3-4 of the algorithm) – c-4-d
Add node d to the component (line 7 of the algorithm)
Recalculate the edge distances from d to (e,f) to see if there is a shorter edge than so far calculated (line 8 of the algorithm)



Min edge**: lowcost: 1 3 4 5 9 --- closest: c a c c c ---** U = {a,c,b,d} V-U = {e,f} min = 4; k = d
j = e; if C[d,e] < lowcost[b] then { lowcost[e] = C[d,e]; closest[e] = d } → §<5 → no change
j = f; if C[d,f] < lowcost[e] then { lowcost[f] = C[d,f]; closest[f] = d } → 3<9 → **d-3-f**
Result after this iteration: lowcost: 1 3 4 5 3 --- closest: c a c c d

Now look for the closest non-component node to the component – node f
Calculate the shortest edge (lines 3-4 of the algorithm) – d-3-f
Add node f to the component (line 7 of the algorithm)
Recalculate the edge distances from f to (e) to see if there is a shorter edge than so far calculated (line 8 of the algorithm)



Min edge**: lowcost: 1 3 4 5 3 --- closest: c a c c d** --- U = {a,c,b,d,f} V-U = {e} min = 3; k = f
j = e; if C[f,e] < lowcost[e] then { lowcost[e] = C[f,e]; closest[e] = f } → 8<5 → no change
Result after this iteration: lowcost: 1 3 4 5 3 --- closest: c a c c d

Now look for the closest non-component node to the component – node e
Calculate the shortest edge (lines 3-4 of the algorithm) – c-5-e
Add node e to the component (line 7 of the algorithm)
Recalculate the edge distances from e to (¤) to see if there is a shorter edge than so far calculated (line 8 of the algorithm) – no edges

Result after this iteration: lowcost: 1 3 4 5 3 --- closest: c a c c d