

Elementary Introduction to Graph Theory

(01YEIG 2026/2027)

Lecture 5



Francesco Dolce
dolcefra@cvut.cz

October 23, 2026

updated: October 23, 2026

PDF available at the address:
dolcefra.pages.fit/ens/2627/YEIG-lecture-05.pdf

Solutions of Exercises in the previous Lecture

Solution of Exercise [Non-isomorphic graphs]. Suppose that $G = (V, E)$ and $H = (U, F)$ are isomorphic graphs, with isomorphism φ . Each vertex $v \in V$ is sent to a vertex $\varphi(v) \in U$ having the same degree. Indeed, φ is, in particular, a bijection between $N_G(v)$ and $N_H(\varphi(v))$. Thus,

$$d_G(v) = \#N_G(v) = \#N_H(\varphi(v)) = d_H(\varphi(v)).$$



Figure 1: Two non-isomorphic graphs with the same order and size.

Since the first graph in Figure 1 has degree sequence $(2, 2, 2, 2)$ while the second has degree sequence $(3, 2, 2, 1)$, the two graphs cannot be isomorphic.

1 Isomorphism of rooted trees

We would like to modify the encoding of ordered trees so that it can be used to decide the isomorphism of rooted trees.

Which properties used in the previous case remain valid in the case of rooted trees?

- 1) a vertex is a root, ✓
- 2) a vertex is a leaf, ✓
- 3) a vertex is a child of a vertex, ✓
- 4) children of the same vertex are ordered. ✗

So we have to compensate for the fact that children of a vertex do not have a fixed (under isomorphism) order. To do that, we:

- choose an ordering $\leq$ on strings over $\{0, 1\}$ (e.g., lexicographical¹ or radix²),
- if v has children $v_1, v_2, \dots, v_k$ such that $C(v_1) \leq C(v_2) \leq \dots \leq C(v_k)$, then we define $C(v) = 0C(v_1)C(v_2) \dots C(v_k)1$.

Example 1 Let $T_1(\mathbf{d})$ and $T_2(\mathbf{C})$ be the rooted trees in Figure 2.

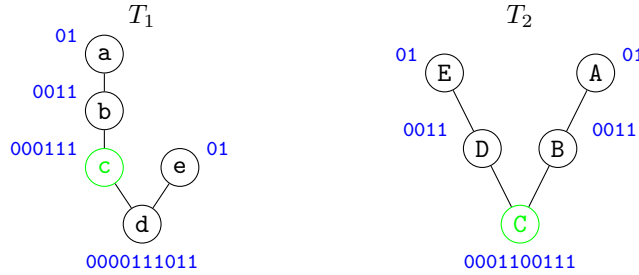


Figure 2: $T_1 \cong T_2$ but $T_1(\mathbf{d}) \not\cong_R T_2(\mathbf{C})$.

We have

$$C(\mathbf{a}) = C(\mathbf{e}) = 01, \quad C(\mathbf{b}) = 0011 \quad \text{and} \quad C(\mathbf{c}) = 000111.$$

If we use the lexicographic order $\leq_{lex}$ we have

$$C(\mathbf{c}) = 000111 <_{lex} 01 = C(\mathbf{e}),$$

¹For two words u, v : $u <_{lex} v$ if $u = pau', v = pbv'$ with p, u', v' words and $a < b$ letters.

²For two words u, v : $u <_{rad} v$ if $|u| < |v|$, or $|u| = |v|$ and $u <_{lex} v$.

so

$$C(T_1) = C(\mathbf{d}) = 0.C(\mathbf{c}).C(\mathbf{e}).1 = 0000111011$$

Similarly, we have

$$C(\mathbf{A}) = C(\mathbf{E}) = 01, \quad C(\mathbf{B}) = C(\mathbf{D}) = 0011, \quad \text{and} \quad C(\mathbf{D}) = 0011 = C(\mathbf{B}).$$

Hence, $C(T_2) = C(\mathbf{C}) = 0001100111 \neq C(T_1)$. Thus, $T_1(\mathbf{d}) \not\cong_R T_2(\mathbf{C})$.

On the other hand, one can check that the three trees in Figure 3 all have the same encoding (using the lexicographic order) 0001101011.

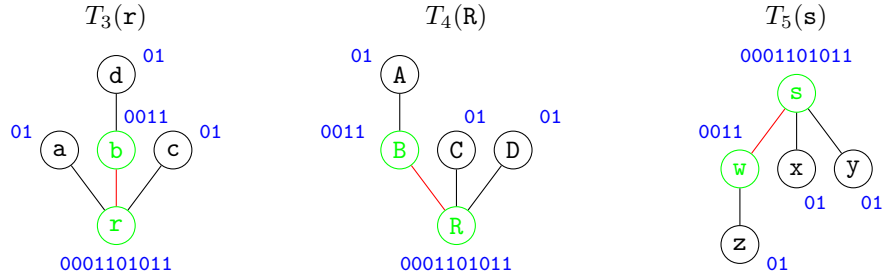


Figure 3: Three ordered trees.

Note that the particular choice of the order is irrelevant, provided that we use the same order for all subtrees.

Exercise [Isomorphism of rooted trees]. Prove, using the radix order, that:

- a) $T_1(\mathbf{d}) \not\cong_R T_2(\mathbf{C})$,
- b) $T_3(\mathbf{r}) \cong_R T_4(\mathbf{R}) \cong_R T_5(\mathbf{s})$.

2 Isomorphism of unrooted trees

Again, we would like to adapt the encoding of rooted trees to decide isomorphism of general trees. In switching from rooted trees to unrooted ones, the problems are much more serious. Indeed, there is no distinguished root; thus we cannot establish the child/parent relationship between nodes.

Therefore, we have to find a suitable (i.e., preserved under isomorphism) substitute for the root. To do that, we use the centre of a tree. Recall that, in a graph $G = (V, E)$, the eccentricity of a vertex v is $\text{ecc}_G(v) = \max\{d_G(v, u) \mid u \in V\}$, and the centre of G is

$$\mathcal{C}(G) = \left\{ v \in V \mid \text{ecc}_G(v) = \min_{u \in V} \text{ecc}_G(u) \right\}.$$

In a general graph $G = (V, E)$ the centre of G may contain an arbitrary number of vertices (see, e.g., $\mathcal{C}(C_3)$ where the centre is the entire set of vertices). In the case of trees, the situation is simpler.

Theorem 1 *Let $T = (V, E)$ be a tree. Then $\mathcal{C}(T) = \{x\}$ or $\mathcal{C}(T) = \{x, y\}$, with $x, y \in V$. Moreover, in the latter case we have $\{x, y\} \in E$.*

Example 2 Let T_1 and T_2 be the trees in Figure 2, and T_3, T_4, T_5 the ones in Figure 3. One can check that

$$\mathcal{C}(T_1) = \{\mathbf{c}\}, \mathcal{C}(T_2) = \{\mathbf{C}\}, \mathcal{C}(T_3) = \{\mathbf{b}, \mathbf{r}\}, \mathcal{C}(T_4) = \{\mathbf{B}, \mathbf{R}\}, \mathcal{C}(T_5) = \{\mathbf{w}, \mathbf{s}\}.$$

Since the only graph property used in the definition of centre of a tree is the distance, and since the distance of vertices is preserved under the isomorphism, we can use the centre of a tree instead of the root. In particular, when the centre is a singleton, we consider it as the root of the tree. When the centre is a set of two vertices, we delete the edge joining them. The two resulting components are seen as rooted trees, with roots exactly the two vertices above.

Algorithm 1: IsoTrees(T_1, T_2)

Input: Two trees $T_1 = (V_1, E_1)$ and $T_2 = (V_2, E_2)$
Output: TRUE if $T_1 \cong T_2$; FALSE if not

```

1 Find centres  $\mathcal{C}(T_1)$  and  $\mathcal{C}(T_2)$ 
2 if  $\#\mathcal{C}(T_1) \neq \#\mathcal{C}(T_2)$  then
3   return FALSE //  $T_1 \not\cong T_2$ 
4 else if  $(\mathcal{C}(T_1) = \{x\} \text{ and } \mathcal{C}(T_2) = \{y\})$  then
5   return IsoRootTrees( $T_1(x), T_2(y)$ ) //  $T_1 \cong T_2 \Leftrightarrow T_1(x) \cong_R T_2(y)$ 
6 else
7   Let  $\mathcal{C}(T_1) = \{x_1, x_2\}$  and  $\mathcal{C}(T_2) = \{y_1, y_2\}$ .
8   We use 4 rooted trees: // remove central edges
9    $T_1^{(1)}(x_1)$ : the component of  $T_1 - \{x_1, x_2\}$  containing  $x_1$ ;
10   $T_1^{(2)}(x_2)$ : the component of  $T_1 - \{x_1, x_2\}$  containing  $x_2$ ;
11   $T_2^{(1)}(y_1)$ : the component of  $T_2 - \{y_1, y_2\}$  containing  $y_1$ .
12   $T_2^{(2)}(y_2)$ : the component of  $T_2 - \{y_1, y_2\}$  containing  $y_2$ .
13  return  $\left( \left( \text{IsoRootTrees}(T_1^{(1)}(x_1), T_2^{(1)}(y_1)) \text{ and } \right. \right.$ 
       $\left. \left. \text{IsoRootTrees}(T_1^{(2)}(x_2), T_2^{(2)}(y_2)) \right) \text{ or } \right.$ 
       $\left( \text{IsoRootTrees}(T_1^{(1)}(x_1), T_2^{(2)}(y_2)) \text{ and } \right.$ 
       $\left. \left. \text{IsoRootTrees}(T_1^{(2)}(x_2), T_2^{(1)}(y_1)) \right) \right)$ 
14 //  $T_1 \cong T_2 \Leftrightarrow \left( T_1^{(1)}(x_1) \cong_R T_2^{(1)}(y_1) \wedge T_1^{(2)}(x_2) \cong_R T_2^{(2)}(y_2) \right)$ 
       $\vee \left( T_1^{(1)}(x_1) \cong_R T_2^{(2)}(y_2) \wedge T_1^{(2)}(x_2) \cong_R T_2^{(1)}(y_1) \right)$ 

```

Example 3 Let T_1, T_2, T_3 and T_4 as in Figures 2 and 3.

- Clearly $T_1 \not\cong T_3$ since $\#\mathcal{C}(T_1) \neq \#\mathcal{C}(T_3)$.
- We have $T_1 \cong T_2$, since one can easily check that $T_1(\mathbf{c}) \cong_R T_2(\mathbf{C})$.
- We have $T_3 \cong T_4$, since one can build the rooted trees $T_3^{(1)}(\mathbf{b})$, $T_3^{(2)}(\mathbf{r})$, $T_4^{(1)}(\mathbf{B})$ and $T_4^{(2)}(\mathbf{R})$ (see Figure 4) and easily check that $T_3^{(1)}(\mathbf{b}) \cong_R T_4^{(1)}(\mathbf{B})$ and $T_3^{(2)}(\mathbf{r}) \cong_R T_4^{(2)}(\mathbf{R})$.

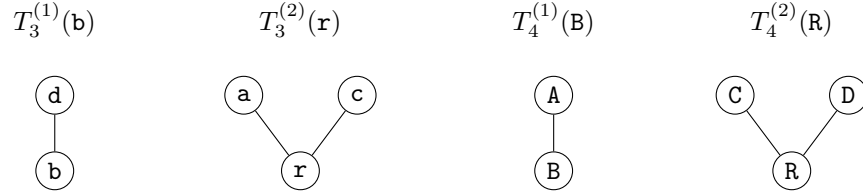


Figure 4: The trees obtained by deleting the central edges in T_3 and T_4 .

The algorithm above can be implemented in polynomial-time.

3 Directed graphs

A *directed graph*, or *digraph*, is a pair $D = (V, A)$, where V is a (finite) set, and $A \subseteq V \times V$.

Elements of V are called *vertices*, while elements of A are called *arcs* (or *directed edges*, or *oriented edges*).

Given an arc $a = (u, v) \in A$, we call u the *tail* of a and v the *head* of a . The vertices u, v are the *endpoints* of a . The vertex u is called an *in-neighbour* of v , and the vertex v an *out-neighbour* of u . Thus, it is natural to define

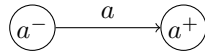
$$N_D^-(v) = \{u \mid (u, v) \in A\} \quad \text{and} \quad N_D^+(v) = \{u \mid (v, u) \in A\}.$$

The *in-degree* and the *out-degree* of a vertex v are defined, respectively, as

$$d_D^-(v) = \#N_D^-(v) \quad \text{and} \quad d_D^+(v) = \#N_D^+(v).$$

A loop (u, u) contributes once to both the in-degree and the out-degree.

Given an arc $a = (u, v)$, we also denote by $a^- = u$ its starting vertex and by $a^+ = v$ its ending vertex.



A graph $G = (V, E)$ is the *underlying graph* of a digraph $D = (V, A)$ if $\{u, v\} \in E$ whenever $(u, v) \in A$.

Example 4 The digraph $D = (V, A)$, with vertices $V = \{u, v, w, x, y\}$ and arcs $A = \{(u, v), (u, w), (v, x), (x, v), (y, v), (y, y)\}$ is shown in Figure 5 on the left. One can check that $d_D^-(v) = 3$ and $d_D^+(v) = 1$.

The underlying graph G of D is shown on the same figure on the right.



Figure 5: A digraph D (left) and its underlying graph G (right).

An *orientation* of a graph $G = (V, E)$ is a digraph obtained by choosing a direction for every edge in E . The *converse* of a digraph $D = (V, A)$ is the digraph $D' = (V', A')$ with set of vertices $V = V'$ and set of arcs given by $(u, v) \in A \Leftrightarrow (v, u) \in A'$.

A digraph and its converse have the same underlying graph.

Example 5 An orientation of the graph $G = (\{a, b, c\}, \{\{a, b\}, \{b, c\}\})$ is the digraph $D = (\{a, b, c\}, \{(a, b), (c, b)\})$. The converse of D is the digraph $D' = (\{a, b, c\}, \{(b, a), (b, c)\})$ (see Figure 6).

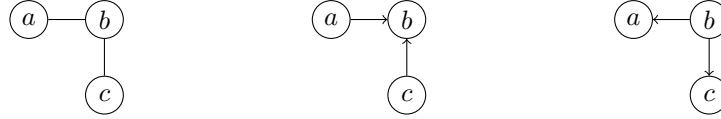


Figure 6: A graph (left) and two of its possible orientations (centre and right).

4 Out-branching

An *out-branching* is an orientation of a rooted tree in which the root has in-degree 0, and every other vertex has in-degree 1.

In this case (u, v) is an arc in an out-branching if and only if u is the parent of v .

Example 6 Let $T(\mathbf{r})$ be the rooted tree shown on the left of Figure 7. The out-branching D of $T(\mathbf{r})$ is shown on the right of Figure 7.



Figure 7: A rooted tree (left) and its out-branching (right).

5 Graph Traversal Algorithms (GTS)

A *graph traversal* is the process of visiting each vertex of a (connected) graph. It is a key ingredient of many graph algorithms.

Example 7 Let G be the graph in Figure 8. If the starting point is the vertex a , a possible traversal is the walk $(a, d, b, c, f, e, d, g, h, i)$.

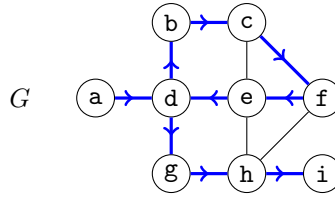


Figure 8: The walk $(a, d, b, c, f, e, d, g, h, i)$ is a traversal.

Note that in a traversal we do not necessarily use all edges, and we may visit the same vertex more than once.

Let us consider a graph traversal algorithm that produces a closed walk and that also traverses every edge exactly once in each direction.

Tarry's algorithm. The oldest known GTS algorithm, Tarry's algorithm, uses two rules while traversing the graph:

- (R1) every edge can be used at most once in every direction;
- (R2) the edge of the first arrival to a vertex can be used (in the opposite direction) only if there is no other possibility.

Theorem 2 *Tarry's traversal is finite.*

Moreover, when the traversal ends, we have returned to the starting vertex and every edge of G has been traversed exactly once in each direction.

The standard graph traversal (or searching) algorithms are:

- *Breadth-First Search* (using queues).

- *Depth-First Search* (using stacks).

The following algorithm can be used with D either a queue, for BFS, or a stack, for DFS.

Algorithm 2: Traversal(G)

Input: a graph G , a vertex s

Output: all vertices in the component containing s are processed

Init: $D = \{s\}$

```

1 repeat
2   | pop a vertex from  $D$ 
3   | process it
4   | put all its undiscovered neighbours in  $D$ 
5 until  $D$  is empty

```

Example 8 A possible traversal of the graph G in Figure 8 starting from **a** is given by the queue (using BFS):

$$\emptyset \rightarrow (\underline{\mathbf{a}}) \rightarrow (\underline{\mathbf{d}}) \rightarrow (\underline{\mathbf{b}}, \underline{\mathbf{e}}, \underline{\mathbf{g}}) \rightarrow (\underline{\mathbf{e}}, \underline{\mathbf{g}}, \underline{\mathbf{c}}) \rightarrow (\underline{\mathbf{g}}, \underline{\mathbf{c}}, \underline{\mathbf{f}}, \underline{\mathbf{h}}) \rightarrow$$

$$\rightarrow (\underline{\mathbf{c}}, \underline{\mathbf{f}}, \underline{\mathbf{h}}) \rightarrow (\underline{\mathbf{f}}, \underline{\mathbf{h}}) \rightarrow (\underline{\mathbf{h}}) \rightarrow (\underline{\mathbf{i}}) \rightarrow \emptyset.$$

Example 9 A possible traversal of the graph G in Figure 8 starting from **a** is given by the stack (using DFS):

$$\emptyset \rightarrow \begin{bmatrix} \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{e}} \\ \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{f}} \\ \underline{\mathbf{e}} \\ \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{c}} \\ \underline{\mathbf{f}} \\ \underline{\mathbf{e}} \\ \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{b}} \\ \underline{\mathbf{c}} \\ \underline{\mathbf{f}} \\ \underline{\mathbf{e}} \\ \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{c}} \\ \underline{\mathbf{f}} \\ \underline{\mathbf{e}} \\ \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{f}} \\ \underline{\mathbf{e}} \\ \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{h}} \\ \underline{\mathbf{f}} \\ \underline{\mathbf{e}} \\ \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow$$

$$\rightarrow \begin{bmatrix} \underline{\mathbf{h}} \\ \underline{\mathbf{f}} \\ \underline{\mathbf{e}} \\ \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{i}} \\ \underline{\mathbf{h}} \\ \underline{\mathbf{f}} \\ \underline{\mathbf{e}} \\ \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{h}} \\ \underline{\mathbf{f}} \\ \underline{\mathbf{e}} \\ \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{f}} \\ \underline{\mathbf{e}} \\ \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{e}} \\ \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{d}} \\ \underline{\mathbf{a}} \end{bmatrix} \rightarrow \begin{bmatrix} \underline{\mathbf{a}} \end{bmatrix} \rightarrow \emptyset$$

We can use a GTS algorithm, e.g., to find the components of a (not necessarily connected) graph G :

1. run DFS starting from any unprocessed vertex $v \in V$;
2. when the search terminates, then the vertices processed are exactly the ones in the connected component of G containing v ;
3. if some vertices of G remain unprocessed, choose one of them and repeat;
4. continue until every vertex of G has been processed.

6 Tree-Search Algorithms: BFS vs DFS

A *spanning tree* of a graph $G = (V, E)$ is a tree $T = (V, E')$ with $E' \subseteq E$.

A graph that is not connected cannot contain any spanning tree.

On the other hand, any connected graph contains at least one spanning tree.

Example 10 Let G be the graph in Figure 9 on the left. Two possible spanning trees are represented on the right of the same figure.

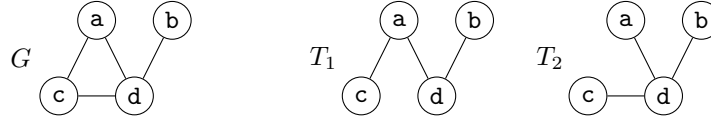


Figure 9: A graph and two of its possible spanning trees.

One can modify Algorithm 2 to construct a spanning tree of a graph. For instance, the following algorithm uses BFS, and a queue, to construct a spanning tree of a given graph.

Algorithm 3: Breadth-FirstSearch(G)

Input: a connected graph $G = (V, E)$

Output: a spanning tree $T = (V', E')$ of G

Init: choose a vertex $v \in V$

```

1   $Q = (v)$ 
2   $V' = \{v\}, E' = \emptyset$ 
3  repeat
4      let  $x$  be the head of  $Q$ 
5      foreach  $y \in N_G(x)$  do
6          if  $y \notin V'$  then
7              append  $y$  to  $Q$ 
8               $V' \leftarrow V' \cup \{y\}$ 
9               $E' \leftarrow E' \cup \{x, y\}$            //  $y$  is a child of  $x$ 
10     remove  $x$  from  $Q$ 
11 until  $Q$  is empty

```

Example 11 Let G be the graph in Figure 8. A spanning tree is given by the following queue (see also Figure 10):

$\emptyset \rightarrow (a) \xrightarrow{\{a,d\}} (a, d) \rightarrow (d) \xrightarrow{\{d,b\}} (d, b) \xrightarrow{\{d,e\}} (d, b, e) \xrightarrow{\{d,g\}} (d, b, e, g)$
 $\rightarrow (b, e, g) \xrightarrow{\{b,c\}} (b, e, g, c) \rightarrow (e, g, c) \xrightarrow{\{e,f\}} (e, g, c, f)$
 $\xrightarrow{\{e,h\}} (e, g, c, f, h) \rightarrow (g, c, f, h) \rightarrow (c, f, h) \rightarrow (f, h) \rightarrow (h)$
 $\xrightarrow{\{h,i\}} (h, i) \rightarrow (i) \rightarrow \emptyset.$

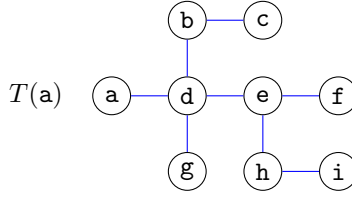


Figure 10: A spanning tree obtained by BFS.

Remember that, for a rooted tree $T(r)$ the level of a vertex v in T is defined as $\ell_T(v) = d_T(r, v)$.

Let $T(r)$ be a spanning tree of a connected graph G obtained by BFS. Then it is easy to show that for every vertex v of G we have $\ell_T(v) = d_G(r, v)$, i.e., the distance between r and v in the original graph is the same as the distance in the spanning tree.

Example 12 Let G be the graph in Example 7 and $T(a)$ be the rooted spanning tree obtained from it in Example 11. One can check that we have, e.g., $d_G(a, h) = d_T(a, h) = 3$.

Similarly we can construct a spanning tree of a graph using DFS, with a stack as data structure.

Algorithm 4: Depth-FirstSearch(G)

Input: graph $G = (V, E)$

Output: a spanning tree $T = (V', E')$ of (a connected component of) G

Init: choose a vertex v and push it to the top of S

```

1  $V' = \{v\}, E' = \emptyset$ 
2 repeat
3   let  $x$  be the top of  $S$ 
4   if  $N_G(x) \setminus V'$  is not empty then
5     choose  $y \in N_G(x) \setminus V'$ 
6     push  $y$  to the top of  $S$ 
7      $V' \leftarrow V' \cup \{y\}$ 
8      $E' \leftarrow E' \cup \{x, y\}$            //  $y$  is a child of  $x$ 
9   else
10    remove  $x$  from  $S$ 
11 until  $S$  is empty

```

Example 13 Let G be the graph in Figure 8. A spanning tree is given by the

following stack (see also Figure 11):

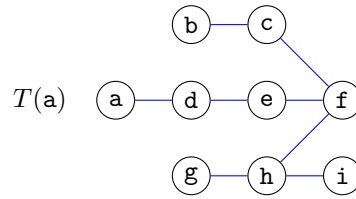
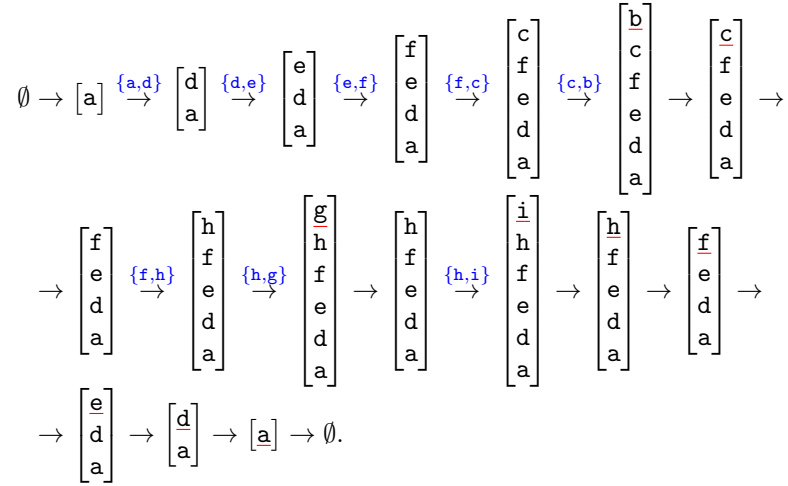


Figure 11: A spanning tree obtained by DFS.