DSA3 seminar assignment

Radoman Radoman 157m/21
University of Novi Sad, Faculty of Sciences
radoman88@gmail.com

1 Introduction

2 BFS and DFS

2.1 Purpose and theoretical background

Both BFS (breadth-first search) and DFS (depth-first search) are tree-search algorithms. They are both used to determine whether a graph is connected.

Theorem
A graph is connected if and only if, you can not split it into two such that there are no edges between them.

An approach to check wheter a graph is connected is by checking all the partitions (the problem is that there are 2^{n-1} of them so it is very inneficient). Another way to check whether a graph is connect is by checking all pairs of vertices - if there is a path connecting them (problem here again is that there are n^2 pairs of vertices and for each of them we have to check if there is a path which is also very inneficient on top of number of partitions that we would need to check).

Definition
A graph is connected if and only if it has a spanning tree.

Checking whether a graph is connected in such a way (by determining whether it has a spanning tree) is far and away the most efficient way to check graph connectedness.

Both DFS and BFS work in a similar way (they both produce spanning trees), but with differences that impact the priority of edges selected. Depending on the way we used to choose edges we get different trees.

A **tree-search** algorithm on G:
> Start with a single root vertex $r\ \varepsilon\ V(G)$. This is our initial tree (with just one vertex).
> Repeat (for as long as possible):
- Do we have a spanning tree?
- Is the tree edge cut empty?
- If not, add one edge from the cut to the tree.

If we want to check *connectedness* of G, that is the whole algorithm. As noted above, depending on the way we use to choose edges, different spanning tree will result (if the graph is connected of course).

Define: edge cut, rooted tree (also called r-tree), levels (distance from root to a specific vertex), ancestor, descendant, parent or predecessor and children.

Predecessor function: $p(v)$, for all $\frac{v\ \varepsilon V}{\{r\}}$

2.2 BFS

2.2.1 BFS instroduction

In **breadth-first search** the adjacency lists of vertices are considered on a first-come-first serve basis. **BFS** is implemented with a *queue* data structure.

A *queue* data structure operates in a FIFO (first-in, first-out) principle. Meaning first element that entered the queue, will be the first one to leave the *queue* (elements are removed in the same order in which they were inserted).

For a connected graph, the algorithm will return:
- A spanning tree given by its predecessor function,
- the level function,
- the time function (order in which vertices are added to the tree)

In BFS algorithm we will first consider all of the neighbors (one-by-one) before we look through the neighbours of any of them.

Therefore, first edges incident to r are selected, and only after that we are looking at neighbors of the neighbors of r.

This is called Breadth-first search algorithm. This approach expands (spreads) the tree as much as possible.

We start with just the root r in the queue and we repeatedly pop the head of the queue, and push all its new neighbors to the queue.

2.2.2 BFS algorithm

> INPUT: a connected graph G, a vertex $r \ \varepsilon V(G)$
> OUTPUT: an r-tree $T \subseteq G$, its predecessor function p, its level function l, the time function t

BFS algorithm
$Q := \emptyset, \quad Q \leftarrow r, \quad l(r) := 0 \quad t(r) := 1, \quad$ mark $r, i := 1$
while $Q \neq \emptyset$

  consider the head x of Q
  if x has unmarked neighbor y **then**

    $i++$
    $Q \leftarrow y, \quad$ mark $y, \quad p(y) := x, \quad l(y) := l(x) + 1, \quad t(y) := i$

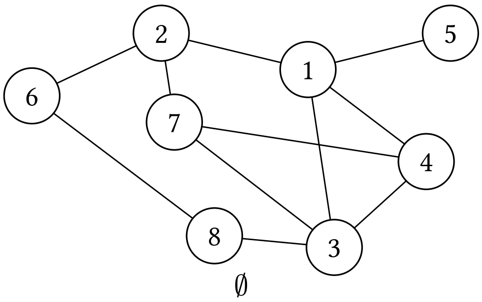
  else

    remove head of Q

  end if

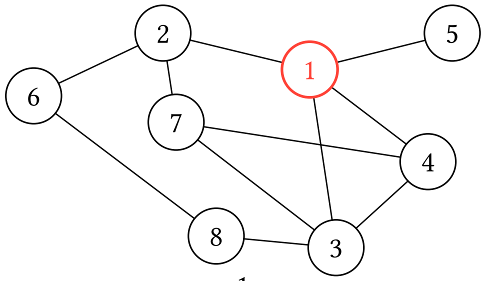
end while
return everything

2.2.3 BFS example



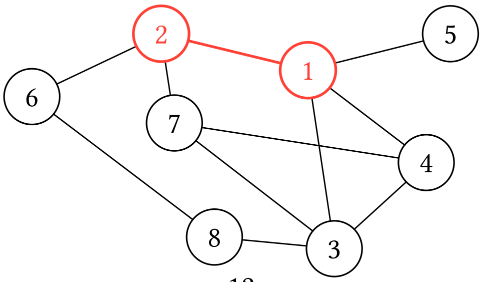
∅

→



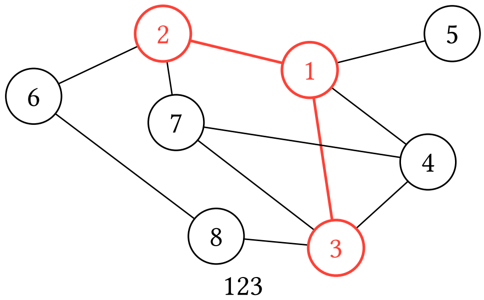
1

→



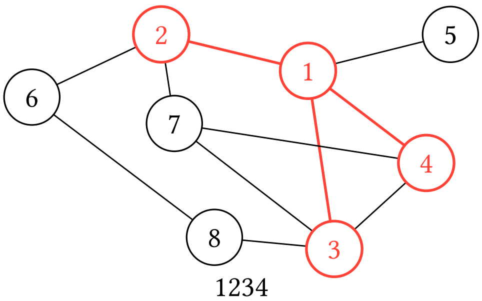
12

→



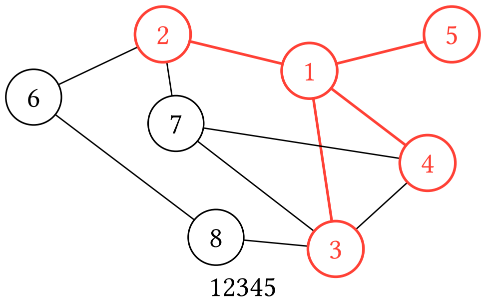
123

→



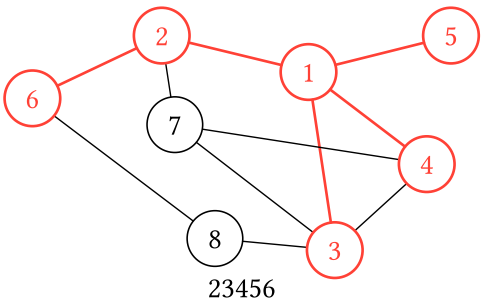
1234

→



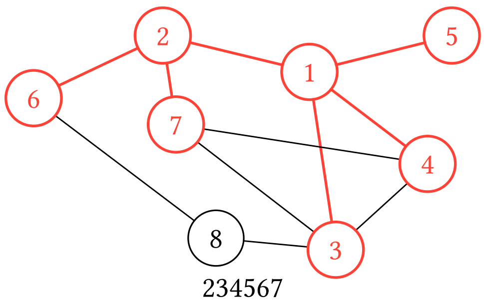
12345

→



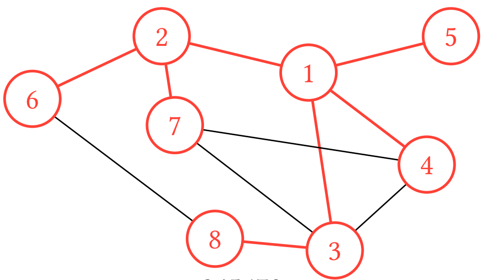
23456

→



234567

→



345678

detailed *(queue is used)*

∅ → 1 → 12 → 123 →
1234 → 12345 → 2345 →
23456 → 234567 →
34567 → 345678 →
45678 → 5678 → 678 →
78 → 8 → ∅

2.2.4 BFS properties

Theorem Let T be a BFS tree of G, with root r.

a.) $l(v) = d_T(r, v),\ $, for every $v \in V$,

b.) $|l(u) - l(v)| \leqslant 1$, for every $uv \in E(G)$.

Level of v is exactly the distance from root r to v.

Every edge of the graph connects only vertices of the same level of the tree or difference by most 1.

Theorem Let T be a BFS tree of G, with root r. Then

$l(v) = d_G(r, v)$, for every $v \in V$

As seen from our example above.

2.3 DFS

2.3.1 DFS introduction

In contrast to BFS, where we first scan the whole adjacency list of the vertex on top of the *queue*, in **depth-first search** we scan the adjacency list of the most recent vertex x added to the *stack* and we look for its neighbour not in T.

If there is such a neighbor, we add it to T. If not, we backtrack to the vertex which was added to T just before x and examine its neighbours, and so on.

For DFS to be implemented, we use a ***stack*** data structure. A *stack* is a linear data structure (like a simple list) which has a top element and basic operations such as placing a new item on top of the *stack*, or removing the top element from the *stack*. In contrast to *queue*, a *stack* operates in LIFO (last-in, first-out) principle, meaning elements are inserted and removed exclusively at the designated end of the structure, referred to as the top.

In **depth-first search**, the *stack* S is initially empty. We pick a root vertex r and scan its neighbours in its adjacency list. We pick one element from that list and place it on top of the *stack* S. We then look at this new vertex, now acting as the new top element of *stack* S and we inspect its adjacency list. If in that list exists a vertex y which is not already in our tree T we select it and add it to the top of *stack* S again. And so on, we continue until there are no suitable vertices in the top element of *stack* S. If there indeed aren't any, we remove the top element of *stack* S and check the vertex that is the new top of the *stack* S.

Again for a connected graph, the algorithm will return:
- A spanning tree given by its predecessor function,
- the level function,
- the time function (order in which vertices are added to the tree)

2.3.2 DFS algorithm

Completely the same as BFS, except that we use a ***stack*** instead of a queue.

> INPUT: a connected graph G, a vertex $r\ \varepsilon V(G)$
> OUTPUT: an r-tree $T \subseteq G$, its predecessor function p, its level function l, the time function t

DFS algorithm
$S := \emptyset, \quad S \leftarrow r, \quad l(r) := 0 \quad t(r) := 1, \quad \text{mark } r, i := 1$
while $S \neq \emptyset$

  consider the top vertex x of S
  if x has unmarked neighbor y **then**

    $i++$
    move y to the top of S, $\quad$ mark y, $\quad p(y) := x, \quad l(y) := l(x) + 1, \quad t(y) := i$

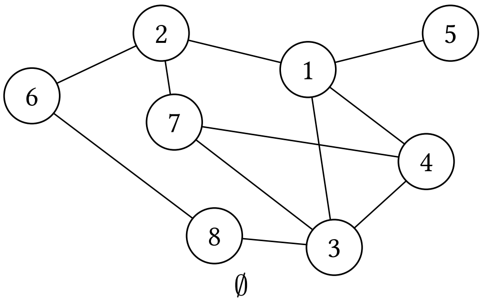
  else

    remove x from S

  end if

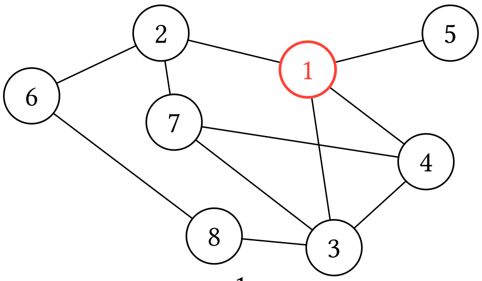
end while
return everything

2.3.3 DFS example



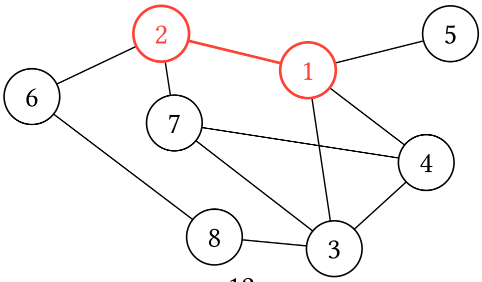
∅

→



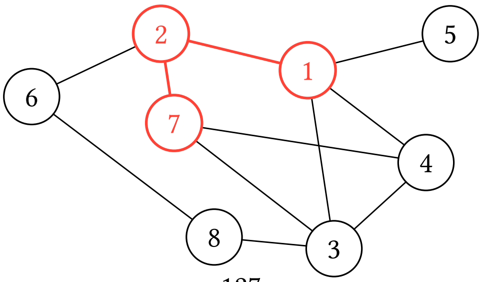
1

→



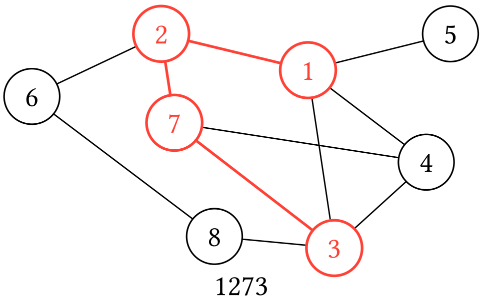
12

→



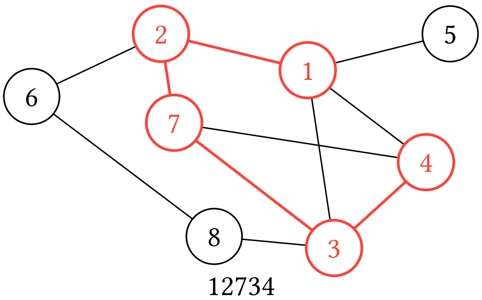
127

→



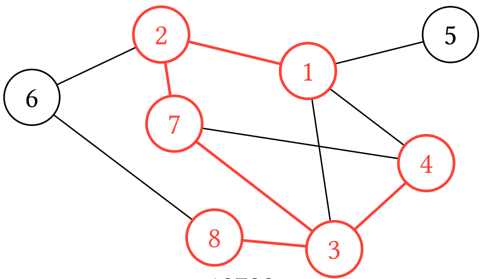
1273

→



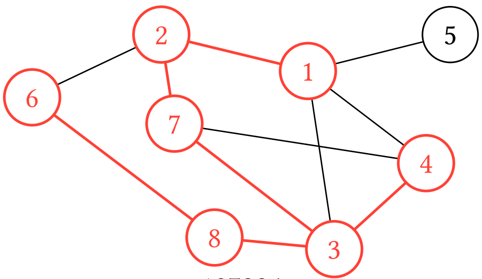
12734

→



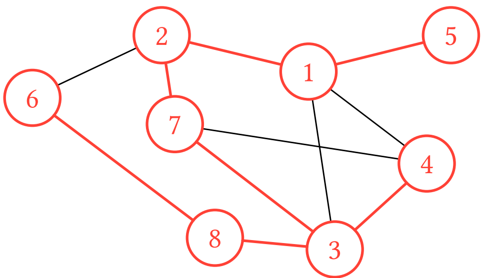
12738

→



127386

→



15

detailed *(stack is used)*

$\emptyset \to 1 \to 12 \to 127 \to$
$1273 \to 12734 \to 1273 \to$
$12738 \to 127386 \to$
$12738 \to 1273 \to 127 \to$
$12 \to 1 \to 15 \to 1 \to \emptyset$

2.3.4 DFS properties

Theorem Let T be a DFS tree of G. Every edge of G joins vertices related in T.

Like we had for BFS a theorem that says every edge of the graph skips at most one level, which is the most important property of BFS. Here we have this property. Every edge goes vertically, from ancestor to predecessor.

3 Applications

4 Conculsion