DSA3 term paper

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

Abstract / Introduction

In this term paper I introduce breadth-first search (BFS) and depth-first search (DFS) algorithms, together with their properties and analyze which algorithm is most applicable for use as a web crawler. Both BFS and DFS are tree-search algorithms which return a spanning tree and can be used to determine whether a given graph is connected. This paper starts with theoretical background needed for tree search algorithms, introduces BFS and DFS, gives examples of both algorithms and finally analyses which one is most applicable for web crawling.

1 BFS and DFS

1.1 Theoretical background

Theorem

A graph is connected if and only if, you can not split it into two such that there are no edges between them. [1], [2], [3], [4], [5], [6]

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.

$$v \ \varepsilon V$$

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

1.2 BFS

1.2.1 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).

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 we have used (added them to the tree) we are looking at the 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*.

For a connected graph, the algorithm will return:

- A spanning tree (**BFS** spanning tree) given by its predecessor function,
- the level function,
- the time function (order in which vertices are added to the tree)

1.2.2 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

$Q := \emptyset, \quad Q \leftarrow r, \quad l(r) := 0 \quad t(r) := 1, \quad \text{mark } r, i := 1$

while $Q \neq \emptyset$

consider the head x of Q

if x has unmarked neighbor y **then**

$i++$

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

else

remove head of Q

end if

end while

return everything

1.2.3 BFS properties

BFS-trees have two basic properties, the first of which justifies our referring to l as a level function.

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 $v \in V$

1.3 DFS

1.3.1 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 (**DFS** spanning tree) given by its predecessor function,
- the level function,
- the time function (order in which vertices are added to the tree)

1.3.2 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, mark y, $p(y) := x$, $l(y) := l(x) + 1$, $t(y) := i$

else

remove x from S

end if

end while

return everything

1.3.3 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 very important property of **DFS** trees. Every edge goes vertically, from ancestor to predecessor.

The following proposition provides a link between the input graph G, its **DFS**-tree T, and the two time functions t and l returned by **DFS**.

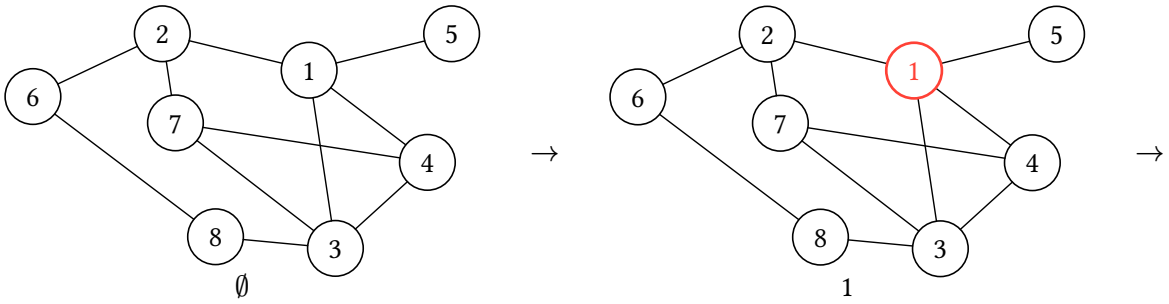
Proposition Let u and v be two vertices of G, with $f(u) < f(v)$.

a) If u and v are adjacent in G, then $l(v) < l(u)$.

b) u is an ancestor of v in T if and only if $l(v) < l(u)$.

1.4 Algorithm examples

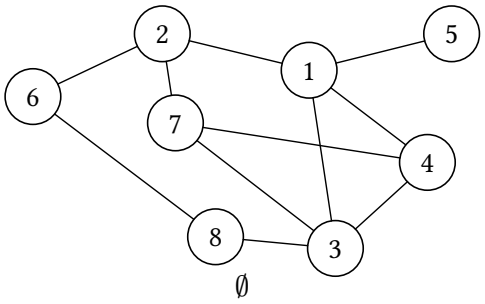
1.4.1 BFS example



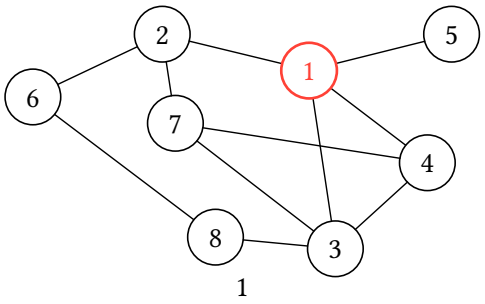
detailed *(queue is used)*

$\emptyset \rightarrow 1 \rightarrow 12 \rightarrow 123 \rightarrow$
$1234 \rightarrow 12345 \rightarrow 2345 \rightarrow$
$23456 \rightarrow 234567 \rightarrow 34567$
$\rightarrow 345678 \rightarrow 45678 \rightarrow$
$5678 \rightarrow 678 \rightarrow 78 \rightarrow 8 \rightarrow$
$\emptyset$

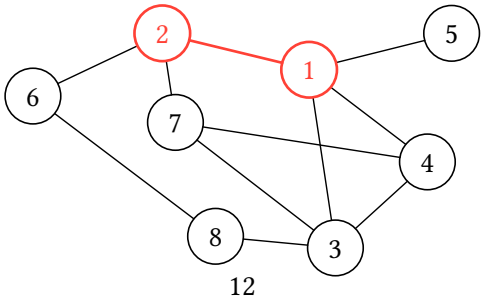
1.4.2 DFS example



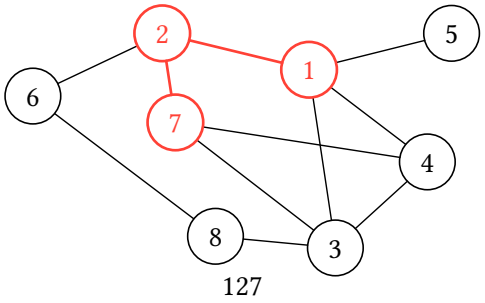
$\emptyset$ →



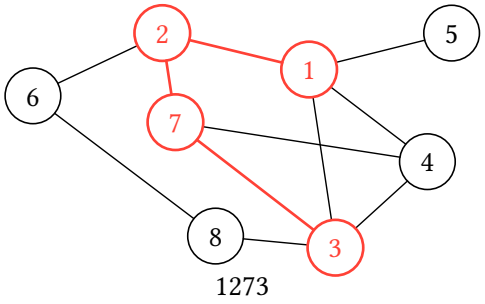
1 →



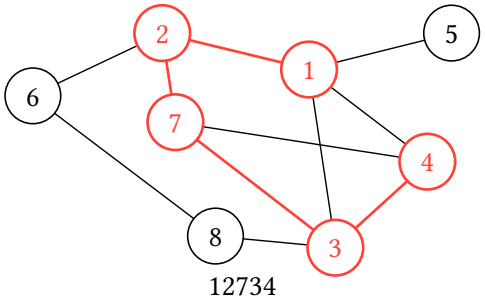
12 →



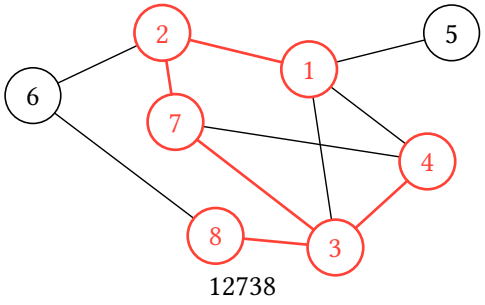
127 →



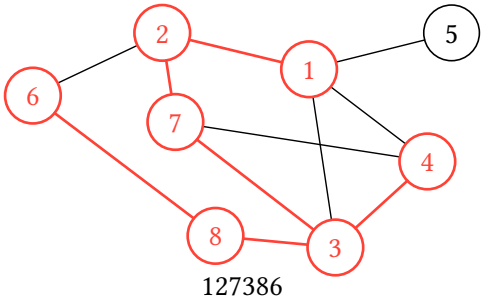
1273 →



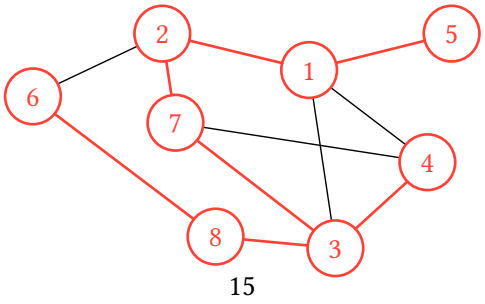
12734 →



12738 →



127386 →



15

detailed *(stack is used)*

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

2 Web crawler application

A web crawler is an internet bot, used by search engines, that systematically browses the world wide web and indexes pages.

Web crawlers store entire pages or parts of page and methodically look for specific information. They store text from the web pages but not media or any scripts that give the page dynamic functionality.

Web crawlers are a part of systems that take (download) web pages and turn them into a structure that can be quickly searchable. As stated, web crawlers role in these systems is to download the relevant information from web pages, which later goes through various stages of transformation in order to index such information (parsing, tokenization and building of the inverted index). Finally, ranking algorithms are applied (such as the Page Rank algorithm).

Tree search algorithms play a vital part of the web crawling phase, when information from the web is fetched, which is later indexed and used in ranking algorithms. Their role is to decide the order of vising pages.

To achieve this a web crawler, during the crawling stage, does the following:

1. It starts from a set of seed URLs
2. Retreives each page
3. Extracts all links from that page
4. Decides which link (URL) to visit next
5. Continues until crawl limits are reached

This process structurally identical to searching a tree or a graph.

Due to this nature, the choice of the tree search algorithm used to go through this network determines how the web crawler explores the web.

In this structure each web page represents a node while each link represents an edge (a directed edge). In this sense the entire world wide web becomes a huge directed graph. Tree search algorithms enable navigation of this graph in a methodological way.

How the Tree search algorithm controls the web crawling process:

- It chooses which URL to crawl next. This is the **crawl frontier**, the core data structure holding URLs that are discovered but not yet crawled.
- It sets the order in which the pages are discovered. Different tree search algorithms crawl the web differently.

Tree search algorithms (explored in this seminar assignment) used in web crawling are both BFS and DFS, each with their own distinct approaches to crawling.

BFS explores level-by-level outward from seeds. On the other hand DFS, explores deep into one path.

BFS is the most commonly used Tree search algorithm for large web crawlers.

Its properties are:

- Its fair in a sense that it divides its attention among many websites rather than focusing on one. This approach prevents one page from monopolizing crawl time.

- It discovers many domains early, which is its major advantage in comparison to DFS. Because BFS explores all first-level links (pages linked directly from seeds) before diving deeper, it quickly discovers a huge number of new domains (in contrast to DFS which might take hours before moving on to the next first-level link). This is cruical if our goal is to build a broad index of the web.
- BFS avoids going too deep into any one site. This is important in order to avoid loops and traps (such as calendar pages) which may keep it forever in certain sections of a web page. BFS protects the web crawler form these traps, by discovering such pages and then adding them to the end of the queue. In such, BFS naturally limits maximum depth early in a crawl.
- BFS supports politeness (not hitting one web page repeatedly) because it automatically distributes requests across many domains.

BFS was used nearly purely in its form in the early search engines such as AltaVista and Yahoo mainly because it was simple, efficient and enabled high coverage of the world wide web.

Additionally majority of academic papers on crawling use BFS because it is easy to implement and its behaviour is predictable making it ideal choice for experimentation and theoretical analysis.

All of these characteristics make BFS the most common basis for large web crawlers.

On the other hand DFS is rarely used as an approach in web crawling due to its properties:

- It goes deep into one web page
- Its gets caught into traps easily (inherently prone to risks od being "stuck" in a page)
- Due to its approach it returns terrible coverage

Usually, if used, DFS is more suitable in settings when small crawlers are needed (for example local crowlers one one domain).

Modern crawling algorithm like Priority Search / Best-First Search is used by search engines such as Google and Bing where the frontier is a priority queue, not a plain FIFO queue.

3 Conclusion

Bibliography

[1] J. A. Bondy and U. S. R. Murty, *Graph Theory with Applications*. 1976.

[2] C. D. Manning, P. Raghavan, and H. Schütze, *An Introduction to Information Retreival*. 2009.

[3] R. Sedgewick and K. Wayne, *Algorithms Fourth Edition*. 2011.

[4] A. Nigam, "Web Crawling Algorithms," 2014.

[5] M. Kurant, A. Markopoulou, and P. Thiran, "On the bias of BFS," 2010.

[6] R. Baeza-Yates, M. Marin, C. Castillo, and A. Rodriguez, "Crawling a Country: Better Strategies than Breadth-First for Web Page Ordering," 2005.