Algorithms for Graphs and Networks – A first helping

Appie van de Liefvoort©, CSEE, UMKC

October 14, 2018

Please review and/or study the basics of Graphs in your text book. These notes are highly condensed and give only incidental insight in graphs and their algorithms. For a solid working knowledge you need additional explanations and examples so that you learn to take advantage of graphs as a tool to solve problems apply this design principle in a different problem settings.
I am also counting on these outside sources for standard graph notions (nodes, edges, paths, cycles, degrees, and so on) and for various implementation options. These notes will help you solidify the abstract notions and algorithmic understanding. There are a number of typo's and perhaps confusing language, for which I apologize. Furthermore, due to me changing my mind a number of times about the order and structure: some things are repeated two or three times (copy-and-paste). It is important to consult the textbook.

Contents

I apologize in advance for typo's that are ever present.
Please let me know the ones you find and you find misleading or irritating.

1 The difference between a tree and a graph

Trees and Graphs are both common discrete objects in our discipline, yet they differ significantly in a number of ways.

	Trees (Binary or general)	**Graph (simple and connected)**
a Their formal definition	$\text{GenTree} = \begin{cases} \varnothing \\ \textit{root} \,\|\, (\text{GenTree})^* \end{cases}$ In words: if the GenTree is not empty, then there is a root, followed by an ordered list of Gentree's.	$G = (V, E)$ In words: A set V of nodes and a set (we assume indeed a set, rather than a multi-set) E of edges, the edges together represent a relationship between nodes.
b Special Node	Root is a special node, providing a focus for the structure	No Special node, no focal point.
c Edges	An edge is not formally defined, although the parent-child relationship leads naturally to the introduction of a directed simple and connected graph. However, this can be done in (at least) two different ways: Parent-Child (one for each child), **or** FirstChild & NextSibling. These would lead to different visualizations and different graphs.	Edges are part of the definition.
d Implied Hierarchy	The Parent-child relationships imply hierarchy; imply depth, height, level and so on.	Edges are to be interpreted as Peer-to-peer relationships. In particular, there is no implied hierarchy. There is still the distance between nodes: path length
e Number of edges	Has $m = n - 1$ parent-child relationships	Has m peer-to-peer relation ships, where $0 \leq m \leq \binom{n}{2} = \frac{n(n-1)}{2}$ in an undirected graph $0 \leq m \leq 2\binom{n}{2} = n(n-1)$ in a directed graph
f Cycles	No cycles.	May have cycles
g Unique paths	Paths between nodes are unique	Paths may not be unique
h Ordered neighbors	Children are ordered	neighbors are not ordered
i Performance of algorithms	The performance of all tree based algorithms are a function of n: $T(n)$	The performance of all graph algorithms are a function of both n and m: $T(n, m)$. The underlying data structures that are used to implement the graph must be chosen carefully in any and all concrete implementation to reach the algorithms predicted performance.
j Visual representation	A visual tree can often be constructed to aide in understanding and designing tree-like algorithms, (including hierarchical control structures). These visualizations are often illuminating.	Again, a visual graph can often be constructed to aide in understanding and designing graph algorithms, except that this forces structural choices to be made: which node is the first to be named, in which order do we chose neighbors, which direction, and at what distance do we visualize neighbors. A picture can give both insight and be misleading at the same time.

Note that we also have to be careful with some notation: A tree with n nodes has $n - 1$ parent-child relationships and gives naturally rise to a connected simple graph with n nodes and $m = n - 1$ edges. The identity of the root has generally been lost in the process. Conversely, a connected simple graph with n nodes and $m = n - 1$ edges can be made into a tree by upon selecting one of the nodes as a root. We will use the terms *rooted tree* and *free tree* to keep them apart, should this be needed.

The most important difference is of course that by definition, a (general or binary) tree has a lot of structure: it has a root, and there is an order between its children. This makes it easy to answer the simple question: *Are the two trees A and B equal?*. The lack of formal structure in a graph (even if it is simple and connected) provides freedom, but also makes it hard to ask the same question: *Are the two graphs G and H equal?*. In particular: which node in G should be identified with which node in H, and in which order should neighboring nodes be selected for comparisons? A brute force method of exhaustive trial-and-error that compares all possibilities results in a very slow as the combinatorial possibilities is factorial.

When discussing trees, we started out by discussing the several methods with which we could linearize the non-linear structure. We did this by several tree traversal algorithms, and found that many, if not most, tree algorithms use these tree-traversals as a template. The same can be done in graphs, except that these traversals are usually called 'search' algorithms, and indeed, many, if not most, graph algorithms use graph traversals (DFS and BFS) as a template.

k Traversals

Pre and Post order Traversal	Depth First Search (DFS), and variations thereof
In-order traversal	Not normally generalized, although another variation of DFS could be construed
Level order traversal	Breath First Search (BFS)

Thus a graph is much more general than a tree. What additional conditions do we need to impose on a graph, such that the graph becomes a tree? A connected graph without cycles is not yet a tree (sometimes called a free tree). A connected graph without cycles and with a node designated as root is not yet a tree (sometimes called rooted tree). A connected graph without cycles, with a designated node as root, and with an ordering defined between neighbors is indeed a tree.

2 DFS: Depth First Search

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

Let us first consider the DepthFirstSearch, which generalizes pre-order traversal of trees, which we will revisit here:

```
algorithm PreOrderTrav(tree $MyTree$)
if is empty ($MyTree$) then return() end-if
ProcessNode($MyTree.root$)
for $child$ from $oldest$ to $youngest$ do
    PreOrderTrav($MyTree.child$)
    end-for
end algorithm
```

Now in order to make this work on graphs, we have to realize that

1. A graph has no 'root': Either a special first, or start node must be provided, or one must be (randomly?) picked. For the purposes of this class, we supply the first node to be processed and call it either S or $start$.
2. A node in a graph has no children, but only neighbors.
3. There is no order defined at all between neighbors in a graph: we will introduce a new algorithmic control statement to accommodate this: '**for all**', as in **for all** w **adjacent** to v **do** *statement* **end-for all**. This does not specify a particular order at the algorithmic level, and leaves the order selection to the implementation level, where actual control structures are chosen to implement the order, reflecting the implementation option used to identify the adjacent nodes w. For the purposes of this class, we ask that the choices be made in alphabetical order, to help in comparing results of algorithms. And yes: we actually assume that nodes have name-labels. Graphs need not have labels, and are defined without labels, but we will introduce various labels of various kinds for various reasons, and you should de the same.
4. The recursive nature of the algorithm guaranteed that all nodes in the left sub-tree were visited before any of the nodes in the right sub-tree, so that essentially all the nodes that are left-children (or first children in case of a general tree), are processed first. The same recursive structure can be used to ensure that nodes will be visited in a 'depth-first' fashion, as long as cycles are prevented, see next item.
5. The hierarchy in the tree (i.e. the direction of the parent-child relationship) ensured that there are no cycles, and thus ensured that the pre-order traversal did not degenerate into an infinite loop. There may be cycles in the graph, but even if there are no cycles, the edges are not directed, so the nodes must be labeled to make sure they are not **processed** a second time, a third time, and so on. Labeling nodes is a common mechanism to indicate various processing stages in graph algorithms, and nodes are typically labeled with a color-label[1], where the color WHITE indicates unknown, or not-yet discovered, or similar connotation. Similarly, the color GRAY indicates that the node has already been discovered, and is being processed, but we still need to hold on to this node for some reason. Finally, the color BLACK indicates that we are done with this node, and that we do no longer need further access to this node. In this case, we assume that in the main-line, the color-label for all the nodes are set to WHITE. Then, after we processed a node, it's status is changed to BLACK.

Still, there are essentially two versions possible: The first assumes that the starting node $start$ has *not* been processed yet, the second does not make this assumption.

```
algorithm DFS.v1(graph $MyGraph$, node $start$)
if $ColorLabel(start)$ IS NOT WHITE then return() end-if
$ColorLabel(start)$ ←GRAY
ProcessNode($start$)
for all nodes $w$, adjacent to $start$ do DFS.v1($MyGraph$, $w$) end-for-all
$ColorLabel(start)$ ←BLACK
end algorithm
```

This algorithm closely resembles the PreOrder traversal and is pretty easy to understand. The algorithm introduces a new control statement: "**for all** nodes w, **adjacent** to $start$ **do** ...... **end-for-all**", which is intuitively clear. It is not clear how adjacent nodes are to be found. A subtle, yet crucial distinction needs to be kept in mind: a graph has nodes and edges; an edge connects two adjacent nodes, and adjacency between two nodes is a derived notion. So finding "nodes w that are **adjacent**" can only be accomplished by following the edge structures associated with the node $start$. We will visit this issue later, Also, it calls itself recursively, and the first statement is to check the color label. In fact, the number recursive calls of this version is $\approx 2m = \Theta(m)$. Such inefficiency should be avoided and we rewrite to algorithm and inspect the color label before calling itself, and by introducing the precondition that the ColorLabel is WHITE[2].

[1] In earlier versions of this write-up, I used the words *unknown*, *discovered*, *visited*, *processed*, *not-yet discovered*, or similar connotation. Some of these words still persist in the more modern coloring-scheme

[2] Such a precondition could be best in a secure programming environment, in which case two versions are simultaneously maintained: one which is available to end-users, and one that is only available i a secure environment.

The number recursive calls of this version is $\Theta(n)$.

```
algorithm DFS.v2(graph $MyGraph$, node $start$)
//Precondition: $ColorLabel(start)$ ==WHITE
$ColorLabel(start)$ ←GRAY
ProcessNode($start$)
for all WHITE nodes $w$ that are adjacent to $start$ and are aplhanumerically selected // (teachers choice)
    do DFS.v2($MyGraph$, $w$) end-for-all
$ColorLabel(start)$ ←BLACK
end algorithm
```

At the core of this algorithm is finding "WHITE nodes w that are **adjacent**", which is not so trivial as it is for trees (be it a binary or general tree). In order to understand how underlying data structures may impact the performance, we present the same algorithm somewhat more procedurally and have a separate test on the color label:

```
algorithm DFS.v3(graph $MyGraph$, node $start$)
//Precondition: $ColorLabel(start)$ ==WHITE
$ColorLabel(start)$ ←GRAY
ProcessNode($start$)
for all nodes $w$ adjacent to $start$ and selected alphanumerically do
    if $ColorLabel(w)$ ==WHITE then DFS.v3($MyGraph$, $w$) end-if
    end-for-all
$ColorLabel(start)$ ←BLACK
end algorithm
```

It is this version that we will use as the template for many graph algorithms. So it is important to understand its working and its performance. To do this, let us introduce Number Labels for both nodes and edges, and adapt the algorithm to number the nodes according to their DFS-order. The very first node will receive number 1 in the DFS-order in a graph, the other nodes are number sequentially according to the order in which they are processed. We assume that there are *global variables* $NodeIterator$ and $EdgeIterator$, which are initialized as 0 in the mainline that is driving the algorithm. At the same time, we will adapt the `DFS.v3` algorithm to construct a DFS- Spanning Tree. This spanning tree consists of all nodes, of course, and all those edges (v, w) that are encountered with that $ColorLabel(v)$ = GRAY and $ColorLabel(w)$ is currently still WHITE and selected for changing to GRAY at the next level of instantiation. This particular algorithm does not actually construct the Spanning Tree, which can be done when the edges are numbered.

```
algorithm DFS.SpTree(graph $MyGraph$, node $start$)
//Precondition: $ColorLabel(start)$ ==WHITE
$ColorLabel(start)$ ←GRAY
$NumberLabel(start) \leftarrow$ ++$NodeIterator$
for all nodes $w$ adjacent to $start$ and selected alphanumerically do
    if $ColorLabel(w)$ ==WHITE then
        $EdgeLabel(start, w) \leftarrow$ ++$EdgeIterator$
        DFS.SpTree($MyGraph$, $w$)
    end-if
end-for-all
$ColorLabel(start)$ ←BLACK
end algorithm
```

To study the complexity, note that all nodes are processed (i.e. numbered) once, and that all edges are inspected to see if it would lead to another WHITE node that can be incorporated in the tree. Upon deeper reflection, all edges are inspected at least once at the **for all** -loop, and sometimes even twice (once in each direction in an undirected graph), resulting in an ideal complexity for this 'DFS' variations of $\Theta(n + 2m)$, which is of course still in the same asymptotic class as $\Theta(n + m)$. To finish up this preliminary discussion, we still need to see how this abstract time complexity can be realized by using implementation data structures that match the time complexity. The core of the complexity is dominated by the statement '**for all** w **adjacent** to $start$ **do**', and this is indeed where implementation options differ. In an adjacency matrix approach, each potential edge is represented and inspected, resulting in n operations for each time a different '$start$' node is processed, for a total of n^2, resulting in an actual time complexity of $\Theta(n + n^2) = \Theta(n^2)$. If however adjacency lists are used, then the actual time complexity can match the ideal one, $\Theta(n+m)$. Some additional time and space complexity is hidden in the recursive calls: A recursive call is made for every edge where the other side has not been visited yet, resulting in $T_{RecCalls}(n, m) = n-1 = \Theta(n)$ and a corresponding space overhead in terms of stack-space, $T^{W}_{space}(n, m) = n - 1 = \Theta(n)$. (Question: When does this worst-case space-complexity occur and what about the best-case for space complexity?)

In short:

Time Complexity DFS	$T^W(n,m) \sim$	$T^W(n,m) = \Theta(\ \ )$
When implemented with Adjacency Lists	$n + 2m$	$\Theta(n+m)$
When implemented with Adjacency Matrix	$n + n^2 = n^2$	$\Theta(n^2)$

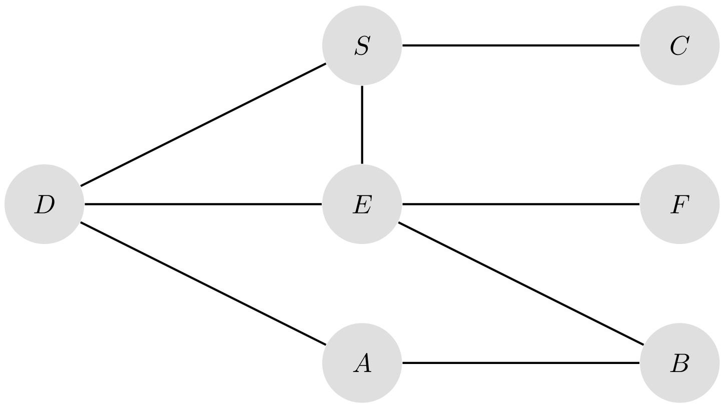
Before continuing with new algorithms, the reader should get a solid working knowledge and do most of the suggested exercises, including the design of algorithms to determine some of the graph properties.

2.1 DFS Suggested Exercises

```
LINE 100: main() Global Variables
LINE 110: int $CFL \leftarrow 0$,                          /* global variable to Count First & Last
LINE 120: int $Ced \leftarrow 0$,                          /* global variable Count edges
LINE 130: int $ColorLabel[n] \leftarrow \{\text{WHITE}\}$;   /* A global Array, to indicate the color of a node
LINE 140: int $Pre\text{-}Label[n] \leftarrow \{0\}$;      /* A global Array, reserving a 'Pre'- Label for each of $n$ nodes
LINE 150: int $In\text{-}Label[n] \leftarrow \{0\}$;       /* A global Array, reserving an 'InA'- Label for each of $n$ nodes
LINE 160: int $Post\text{-}Label[n] \leftarrow \{0\}$;     /* A global Array, reserving a 'Post'- Label for each of $n$ nodes
LINE 170: int $Edge\text{-}Label[u,v] \leftarrow \{0\}$;   /* A global double Array, reserving an 'Edge-Label' for each of $m$ edges

LINE 200: Algorithm void DFSNums(graph $MyGraph$, node $start$)
LINE 210: //Precondition: $ColorLabel[start] == \text{WHITE}$ //
LINE 220: $ColorLabel[start] \leftarrow \text{GRAY}$
LINE 230: $Pre\text{-}Label[start] \leftarrow$ ++$CFL$

LINE 300: for all $w$ adjacent to $start$ and selected alphanumerically do
LINE 310:     $Edge\text{-}Label[start,w] \leftarrow$ ++$Ced$
LINE 320:     if $ColorLabel[w] == \text{WHITE}$ then
LINE 330:         DFSNums($MyGraph$, $w$)
LINE 340:         $In\text{-}Label[w] \leftarrow Pre\text{-}Label[start]$
LINE 350:     end-if
LINE 360: end-for all

LINE 400: $Post\text{-}Label[start] \leftarrow$ ++$CFL$
LINE 410: $ColorLabel[start] \leftarrow \text{BLACK}$
LINE 420: end-algorithm
```

	Pre-Label	*In-Label*	*Post-Label*
S			
A			
B			
C			
D			
E			
F			
G			
H			

(Please show the edge-counts on the graph itself)



1. Fill in the table and construct a DFS-Spanning Tree if the algorithm is called using "DFSNums($MyGraph$, S)" and determine the complexity $T(n,m)$ in terms of both n (number of nodes/vertices) and m (number of edges). Distinguish between representations as Adjacency lists and adjacency matrices.

2. Repeat the first question/algorithm `DFSNums` ($MyGraph, S$), but replace the lines 300 – 399 with:

```
LINE 301: for all $w$ adjacent to $start$ and selected alphanumerically do
LINE 311:     if $ColorLabel[w] ==$ WHITE then
LINE 321:         $Edge\text{-}Label[start, w] \leftarrow$ ++$Ced$
LINE 331:         DFSNums($MyGraph$, $w$)
LINE 341:     end-if
LINE 351: end-for all
```

3. Repeat the first question/algorithm `DFSNums` ($MyGraph, S$), but replace the lines 300 – 399 with:

```
LINE 302: for all $w$ adjacent to $start$ and selected alphanumerically do
LINE 312:     if $ColorLabel[w] ==$ WHITE then
LINE 322:         DFSNums($MyGraph$, $w$)
LINE 332:         $Edge\text{-}Label[start, w] \leftarrow$ ++$Ced$
LINE 342:     end-if
LINE 352: end-for all
```

4. Repeat the first question/algorithm `DFSNums` ($MyGraph, S$), but replace the lines 300 – 399 with:

```
LINE 303: for all $w$ adjacent to $start$ and selected alphanumerically do
LINE 313:     if $ColorLabel[w] ==$ WHITE then
LINE 323:         DFSNums($MyGraph$, $w$)
LINE 333:     end-if
LINE 343:     $Edge\text{-}Label[start, w] \leftarrow$ ++$Ced$
LINE 353: end-for all
```

5. Repeat the first question/algorithm `DFSNums` ($MyGraph, S$), but replace the lines 300 – 399 with:

```
LINE 304: for all $w$ adjacent to $start$ and selected alphanumerically do
LINE 314:     $Edge\text{-}Label[start, w] \leftarrow$ ++$Ced$
LINE 324:     if $ColorLabel[w] ==$ WHITE then
LINE 334:         DFSNums($MyGraph$, $w$)
LINE 344:     end-if
LINE 354:     $Edge\text{-}Label[start, w] \leftarrow$ ++$Ced$
LINE 364: end-for all
```

6. Repeat the first question/algorithm `DFSNums` ($MyGraph, S$), but replace the lines 300 – 399 with:

```
LINE 306: for all $w$ adjacent to $start$ and selected alphanumerically do
LINE 316:     ++$In\text{-}Label[start]$
LINE 326:     ++$In\text{-}Label[w]$
LINE 336:     if $ColorLabel[w] ==$ WHITE then
LINE 346:         DFSNums($MyGraph$, $w$)
LINE 356:     end-if
LINE 366: end-for all
```

7. Repeat the first question/algorithm `DFSNums` ($MyGraph, S$), but replace the lines 300 – 399 with:

```
LINE 308: for all $w$ adjacent to $start$ and selected alphanumerically do
LINE 318:     if $ColorLabel[w] ==$ WHITE then
LINE 328:         ++$In\text{-}Label[start]$
LINE 338:         ++$In\text{-}Label[w]$
LINE 348:         DFSNums($MyGraph$, $w$)
LINE 358:     end-if
LINE 368: end-for all
```

8. Repeat the first question/algorithm `DFSNums(`$MyGraph$`,` S`)`, but replace the lines 300 – 399 with:

```
LINE 1301: for all $w$ adjacent to $start$ and selected alphanumerically do
LINE 1311:     if $ColorLabel[w] ==$ WHITE then
LINE 1321:         DFSNums($MyGraph$, $w$)
LINE 1331:         ++$In\text{-}Label[start]$
LINE 1341:         ++$In\text{-}Label[w]$
LINE 1351:     end-if
LINE 1361: end-for all
```

9. Repeat the first question/algorithm `DFSNums(`$MyGraph$`,` S`)`, but replace the lines 300 – 399 with:

```
LINE 1303: for all $w$ adjacent to $start$ and selected alphanumerically do
LINE 1313:     if $ColorLabel[w] ==$ WHITE then
LINE 1323:         DFSNums($MyGraph$, $w$)
LINE 1333:     end-if
LINE 1343:     ++$In\text{-}Label[start]$
LINE 1353:     ++$In\text{-}Label[w]$
LINE 1363: end-for all
```

10. Repeat the first question/algorithm `DFSNums(`$MyGraph$`,` S`)`, but replace the statement:

LINE 320: **if** $ColorLabel[w] ==$ WHITE **then** by the statement (but note that the algorithm may not always work correctly.)
LINE 1320: **if** $ColorLabel[start] ==$ WHITE **then**

11. Repeat the first question/algorithm `DFSNums(`$MyGraph$`,` S`)`, but replace the statement:

LINE 320: **if** $ColorLabel[w] ==$ WHITE **then** by the statement (but note that the algorithm may not always work correctly.)
LINE 1320: **if** $ColorLabel[w] ==$ GRAY **then**

12. Repeat the first question/algorithm `DFSNums(`$MyGraph$`,` S`)`, but replace the statement:

LINE 320: **if** $ColorLabel[w] ==$ WHITE **then** by the statement (but note that the algorithm may not always work correctly.)
LINE 2320: **if** $ColorLabel[w] ==$ BLACK **then**

13. Repeat the first question/algorithm `DFSNums(`$MyGraph$`,` S`)`, but replace the statement:

LINE 330: `DFSNums(`$MyGraph$`,` w`)` by the statement (but note that the algorithm may not always work correctly.)
LINE 1330: `DFSNums(`$MyGraph$`,` $start$`)`

14. Design a DFS inspired algorithm to detect whether or not the graph has a cycle, and determine the complexity $T(n, m)$ in terms of both n (number of nodes/vertices) and m (number of edges). Distinguish between an adjacency matrix and adjacency lists representation of the graph.

15. Design a DFS inspired algorithm to find in-degrees of the nodes and determine the complexity $T(n, m)$ in terms of both n (number of nodes/vertices) and m (number of edges). Distinguish between an adjacency matrix and adjacency lists representation of the graph.

16. Design a DFS inspired algorithm to find out-degrees of the nodes and determine the complexity $T(n, m)$ in terms of both n (number of nodes/vertices) and m (number of edges). Distinguish between an adjacency matrix and adjacency lists representation of the graph.

17. Design a DFS inspired algorithm to construct a DFS-spanning tree. Add an additional Label to each node that tracks and records the number of children for internal nodes in this particular spanning tree.

18. A graph is said to be Eulerian if there is a cycle such that all edges are used exactly once and are on the cycle. There is a theorem that states that a graph G is Eulerian if-and-only-if G is connected and every vertex has an even degree. Design a DFS inspired algorithm to determine whether or not a given graph G is Eulerian or not. (No need to generate the cycle)

19. There is a theorem that states that a graph G is Eulerian if-and-only-if G is connected and every vertex has an even degree. Is it possible to design a DFS inspired algorithm to construct a Eulerian cycle. Distinguish between two cases: "you know in advance there is one" and "you do not know in advance that there is one." What are the time complexities? (Hint: doubly linked list.)

20. There is a theorem that states that a graph G contains an Eulerian path if-and-only-if G is connected and every vertex has an even degree, ór, every vertex has an even degree except for two nodes that both have an odd degree. Is it possible to design a DFS inspired algorithm to detect and construct Eulerian path? What if you do not know in advance that there is one? What are the time complexities? (Hint: doubly linked list.)

21. (Graduate students) There is a theorem that states that a graph G has a Eulerian cycle if-and-only-if G is connected and every vertex has an even degree. Similarly, a graph G has a Eulerian path between nodes a and b if-and-only-if G is connected and every vertex has an even degree, except for the two nodes a and b that both must have an odd degree. Design an efficient algorithm that prints the edges in the order of a Eulerian path if there is such a path, or prints 'this graph has no Eulerian path'. The efficiency should be $\Theta(n + m)$, where n is the number of nodes, and m is the number of edges. What are the time complexities?

22. (Graduate students) The DFS of a graph is only unique if the start node is given, and if you make a solid rule on how to select any and all adjacent nodes, like we did in
for all w **adjacent to** $start$ and selected alphanumerically **do**".
But if you omit the " and selected alphanumerically " then the sequence may not be unique. So let another sequence of nodes be presented in an array $\langle V[1], V[2] \ldots V[n] \rangle$. Can you design an efficient algorithm to detect (i.e. the answer is Yes/No) whether of not the presented sequence of nodes is a valid DFS.

23. (Graduate students) The following algorithm has been presented to determine whether or not a given sequence (all nodes are listed no duplications) is a valid DFS-sequence. It is based on the observation that any such sequence corresponds to a DFS-tree: The first node in the DFS-sequence is the root of the DFS-tree, and all subsequent elements in the sequence are children to either the lowest parent or to one of it's direct ancestors. Thus for each node (except the root) associate a parent, which is done using a "parent"-label and which allows a track-back along the recursive path (without using recursion). A simple array ($Parent$) is used to represent the labels.

```
LINE 100: algorithm Detect.DFS(graph $MyGraph$, array $\langle V[1], V[2] \ldots V[n] \rangle$)
LINE 110: //Precondition: The $V$-Array is a linearization of the set $V$. Thus all nodes present and no duplicates

LINE 120: array $Parent$                     /** An array to track the parent of node $V[i]$
LINE 130: index $c$                          /** An index used to represent a child for which a parent need to be identified.
LINE 140: index $p$                          /** An index used to represent the node that is currently considered as a parent for $c$.
LINE 150: $Parent[1] \leftarrow 0$   //indicating nil
LINE 160: $p \leftarrow 1$
LINE 170: $c \leftarrow 2$

LINE 180: while $c \leq n$ do
LINE 190:     if $\langle V[p], V[c] \rangle \in E$
LINE 200:         then // this edge is a tree-edge, and advance both parent index and child candidate index
LINE 210:             $Parent[c] \leftarrow p$
LINE 220:             $p \leftarrow c$
LINE 230:             $c \leftarrow c + 1$
LINE 240:         else // $p$ is not the parent of $c$
LINE 250:             // before finding a parent for $c$,
LINE 260:             // make sure that $p$ does not have neighbors remaining that are eligible children:
LINE 270:             $x \leftarrow c + 1$
LINE 280:             while $(x \leq n)$ do
LINE 290:                 if $\langle V[p], V[x] \rangle \in E$ then NO, not a DFS, stop/break end-if
LINE 300:             end-while
LINE 310:             // Now find appropriate parent for $c$, if any.
LINE 320:             $p \leftarrow Parent[p]$
LINE 330:             while $(p > 0$ and $\langle A[p], A[c] \rangle \notin E)$ do $p \leftarrow Parent[p]$ end-while
LINE 340:             if $(p == 0)$ then NO, not a DFS, stop/break end-if
LINE 350:     end-if
LINE 360: end-while

LINE 370: YES, this is a DFS
LINE 380: end algorithm
```

For the parts below, assume that the graph is large (i.e. $n \approx 100$, and that the key-and-basic operations is asking either $\in E$ or $\notin E$.
a. Present convincing arguments why you think it is correct (or present a counter example).
b. Find the best case time complexity (and a proposed DFS sequence that attains this time complexity),
c. Find the worst case time complexity (and a proposed DFS sequence that attains this time complexity),

2.2 The interval Theorem

Please note, that this condensed description is not sufficient for a full understanding and appreciation of the problem, the algorithm, and it's application. Please read corresponding section in the text book.

We have used Labels to rank-order or time-stamp the nodes and/or edges. Could these numbers reveal any structural interpretation? This would not appear to be the case, since there is no ordering relationship defined between the neighbors of a node (there is no left-to-right; there is no oldest-to-youngest; there is nothing). We note however, that every node is labelled twice: once as part of the pre-processing and once as part of the post-processing. All nodes that can be reached from this node are recursively called and, in turn will receive a pre-processing and a post-processing label. The order of these other nodes is perhaps implementation dependent, but all will be finished before the post-order label will be given to the node at hand. This observation gives rise to both the interval-theorem (or parenthesis theorem) and to the Topological Sort algorithm.

The interval-theorem (also known as the parenthesis theorem): Given an undirected graph G, and any DFS-induced Spanning Tree, then

1. Node v is a descendent of node w in the DFS-SpTree if-and-only-if $Pre(w) < Pre(v) < Post(v) < Post(w)$
2. Node v is an ancestor of node w in the DFS-SpTree if-and-only-if $Pre(v) < Pre(w) < Post(w) < Post(v)$
3. Nodes v and w are not in an ancestor/descendent relationship in the DFS-SpTree if-and-only-if either $Post(w) < Pre(v)$ or $Post(v) < Pre(w)$.

Notice that the actual Label-numbers depend on the particular order that adjacent nodes have been selected, and that the Interval Theorem is independent of this order. In particular, the DFS spanning tree is not unique, and the numbering cannot expected to be unique either. But the interval theorem is valid for every spanning tree derived with a DFS adaptation.

2.3 DFS Rejoinder

We have now seen various variations around DFS, and the ColorLabels have been used to allow DFS to structure the nodes in some order. Notice, that DFS provides some structure to a graph, and that different $start$-nodes, and a differing ordering of neighbors would result in different structures, though these are all DFS-structures. The ColorLabels themselves are not intended to provide structure.
Further observation: The DFS-skeleton has been adopted for many different ideas and purposes. The exercises above have given an appreciation for the many opportunities to navigate a graph and to do simple counting or simple enumerations.There are additional opportunities for further operations and actions at all these locations. For instance, we could present the choices at a higher level of abstraction.

```
algorithm AbstractDFS.a(graph $MyGraph$, node $start$)
$ColorLabel(start) \leftarrow$GRAY
ProcessNode($start$)
for all WHITE nodes $w$ adjacent to $start$ do AbstractDFS.a($MyGraph$, $w$) end-for-all
$ColorLabel(start) \leftarrow$BLACK
end algorithm
```

```
algorithm AbstractDFS.b(graph $MyGraph$, node $start$)
$ColorLabel(start) \leftarrow$GRAY
ProcessNode($start$)
for all WHITE nodes $w$ adjacent to any NON-WHITE node $s$ do AbstractDFS.b($MyGraph$, $w$) end-for-all
$ColorLabel(start) \leftarrow$BLACK
end algorithm
```

Additionally, we could impose additional selection criteria, and we will do so for Dijkstra's Shortest Path Algorithm and Prim's Minimal Spanning Tree Algorithms,

```
algorithm AbstractDFS.c(graph $MyGraph$, node $start$)
$ColorLabel(start) \leftarrow$GRAY
ProcessNode($start$)
for all WHITE nodes $w$ adjacent to any NON-WHITE node $s$ and with minimal DISTANCE (resp. EDGE-WEIGHT)
    do AbstractDFS.c($MyGraph$, $w$) end-for-all
$ColorLabel(start) \leftarrow$BLACK
end algorithm
```

The selection criteria could include an ordering, but this may not be desired. For instance, should the selected node w be **adjacent** to any previously colored NON-WHITE node? Or should an ordering be imposed? This essentially destroys the pure DFS-paradigm. Also, the repeated need to find a node with some minimality constraint calls for the use of a priority queue (such as a MIN-HEAP), and both algorithms look much more like a generalization of the Breadth-First-Search.