Algorithms for Graphs and Networks – A first helping

Appie van de Liefvoort©, CSEE, UMKC

November 1, 2018

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.

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

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** GenTree $= \begin{cases} \varnothing \\ root \vert\vert \text{(GenTree)}^* \end{cases}$ 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**
$ColorLabel(start) \leftarrow$GRAY
`ProcessNode(`$start$`)`
for all nodes w, **adjacent** to $start$ **do** `DFS.v1(`$MyGraph$`,` w`)` **end-for-all**
$ColorLabel(start) \leftarrow$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
$NumberLabel(start)$ ← ++$NodeIterator$
for all nodes w **adjacent** to $start$ and selected alphanumerically **do**
  if $ColorLabel(w)$ ==WHITE **then**
    $EdgeLabel(start, w)$ ← ++$EdgeIterator$
    `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** $ColorLabel[n] \leftarrow \{$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

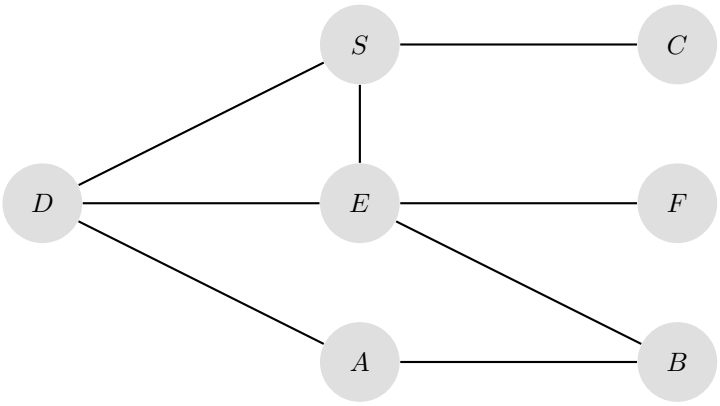
LINE 200: **Algorithm void** `DFSNums(`**graph** $MyGraph$**, node** $start$`)`
LINE 210: //Precondition: $ColorLabel[start] ==$ WHITE //
LINE 220: $ColorLabel[start] \leftarrow$ GRAY
LINE 230: $Pre\text{-}Label[start] \leftarrow$ ++CFL

LINE 300: **for all** w **adjacent to** $start$ and selected alphanumerically **do**
LINE 310: $Edge\text{-}Label[start, w] \leftarrow$ ++Ced
LINE 320: **if** $ColorLabel[w] ==$ 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: $ColorLabel[start] \leftarrow$ BLACK
LINE 420: **end-algorithm**

	$Pre\text{-}Label$	$In\text{-}Label$	$Post\text{-}Label$
S			
A			
B			
C			
D			
E			
F			
G			
H			

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



SE.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.

SE.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**

SE.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**

SE.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**

SE.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**

SE.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**

SE.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**

SE.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**

SE.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**

SE.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**

SE.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**

SE.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**

SE.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($MyGraph,\ start$)

SE.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.

SE.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.

SE.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.

SE.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.

SE.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)

SE.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.)

SE.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.)

SE.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?

SE.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 $\langle V[1], V[2] \dots V[n] \rangle$. 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.

SE.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** $\langle V[1], V[2] \dots V[n] \rangle$`)`
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** $start$`)`
$ColorLabel(start)$ ←GRAY
`ProcessNode(`$start$`)`
for all WHITE nodes w **adjacent to** $start$ **do** `AbstractDFS.a(`$MyGraph$`,` w`)` **end-for-all**
$ColorLabel(start)$ ←BLACK
end algorithm

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

3 BFS: Breadth First Search Traversal

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.

Contrary to the DFS, which is best explained and analyzed as a recursive algorithm, and which implicitly uses a stack as intermediate data structure, the Breath-First-Search (BFS) uses a queue, which is explicitly defined and maintained. The basic idea behind the algorithm is expressed as follows:

```
LINE 100: Algorithm basic-BFS(graph MyGraph, node start)
LINE 110: //Precondition: ColorLabel[start] == WHITE //
LINE 120:  queue MyQueue.new;                          /* A Queue to impose an ordering
LINE 130:  int ColorLabel[n] ← {WHITE};                /* A global Array, to indicate the color of a node

LINE 300: ColorLabel[start] ← GRAY
LINE 310: MyQueue.Insert(start)


LINE 400: while NOT MyQueue.IsEmpty do
LINE 410:     v ← MyQueue.Delete
LINE 420:     for all WHITE w adjacent to v and selected alphanumerically do
LINE 430:         ColorLabel[w] ← GRAY
LINE 440:         MyQueue.Insert(w)
LINE 450:     end-for-all
LINE 460:     ColorLabel[v] ← BLACK
LINE 470: end-while
LINE 480:  end-algorithm basic-BFS
```

When we presented the two DFS skeletons, the discussion centered around the question: do you process a newly discovered node w *before* or *after* you invoke a recursive call to yourself. This question is a matter of aesthetics for DFS, since the first-in-last-out nature of a stack. The controlling structure for the BFS is a queue, and its first-in-first-out control can make a substantial difference. Generally speaking, the WHITE nodes represent the nodes that have not yet been processed, the GRAY nodes are the nodes residing in the queue, while the BLACK nodes represent the nodes that have passed through the queue, have provided all information about their neighboring nodes, and are no longer needed. Processing of nodes can occur either when inserting into the queue (and the color changes from WHITE to GRAY), when deleting from the queue (and the color changes from GRAY to BLACK), or both. Similarly, counting of nodes (e.g. CFL), counting or labeling edges, and constructing spanning trees can be inserted at the appropriate locations. The particular application will dictate where this is performed so that the overall implementation is most efficient.

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

3.1 BFS Suggested Exercises

SE.24: This is the first one in a number of exercises that will help in understanding the BFS algorithm. In all cases, you are asked to write the resulting values for the labels (the **Pre-**, **In-**, and **Post-**) for all the nodes of the graph[3] given, as well as those for the edges. There may be an algorithm that determines the BFS-Spanning-Tree in the correct order, another determines the visit-order for all edges, several others have no traditional or obvious practical applications, other than educational. Thus, you are asked to find any practical interpretations or relationship(s), if any, between the final values of the labels, and n (number of nodes) and m (number of edges). Note: the various labels are initially set to 0, and are overwritten zero or more times, depending on the number of neighboring nodes. Not all Labels are used in all exercises.

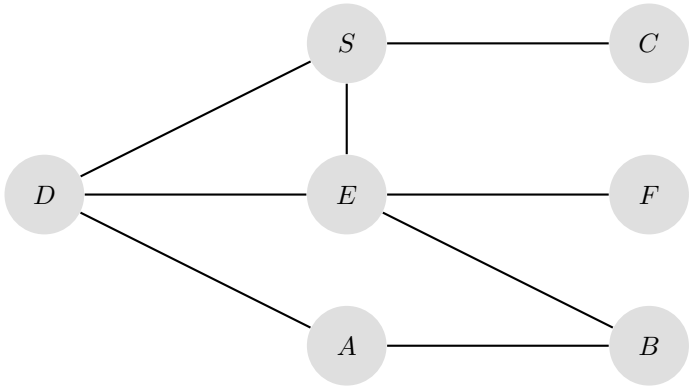
LINE 100: **Algorithm void** `BFSNums` (**graph** $MyGraph$, **node** $start$)
LINE 110: **int** $CFL \leftarrow 0$, /* global variable to Count First & Last
LINE 120: **int** $Ced \leftarrow 0$, /* global variable Count edges
LINE 130: **queue** $MyQueue.new$; /* A Queue to impose an ordering
LINE 140: **int** $Color[n] \leftarrow \{$WHITE$\}$; /* A global Array, to indicate the color of a node
LINE 150: **int** $Pre[n] \leftarrow \{0\}$; /* A global Array, reserving a 'Pre'- Label for each of n nodes
LINE 160: **int** $In[n] \leftarrow \{0\}$; /* A global Array, reserving an 'InA'- Label for each of n nodes
LINE 170: **int** $Dist[n] \leftarrow \{\infty\}$; /* A global Array to record the 'Distance'- Label for each of n nodes
LINE 180: **int** $From[n] \leftarrow \{$NILL$\}$; /* A global Array to record the 'Predecessor'- Label for each of n nodes
LINE 190: **int** $Post[n] \leftarrow \{0\}$; /* A global Array, reserving a 'Post'- Label for each of n nodes
LINE 200: **int** $Edge[u, v] \leftarrow \{0\}$; /* A global double Array, reserving an 'Edge-Label' for each of m edges

LINE 300: $Color[start] \leftarrow$ GRAY
LINE 310: $Pre[start] \leftarrow$ ++CFL
LINE 320: $Dist[start] \leftarrow 0$
LINE 330: $MyQueue.Insert(start)$

LINE 400: **while NOT** $MyQueue.IsEmpty$ **do**
LINE 410: $v \leftarrow MyQueue.Delete$
LINE 420: $In[v] \leftarrow$ ++CFL
LINE 430: **for all** w **adjacent to** v and selected alphanumerically **do**
LINE 440: $Edge[v, w] \leftarrow$ ++Ced
LINE 450: **if** $Color[w] ==$ WHITE **then**
LINE 460: $Color[w] \leftarrow$ GRAY
LINE 470: $Pre[w] \leftarrow$ ++CFL
LINE 480: $Dist[w] \leftarrow Dist\text{-}Label[v] + 1$
LINE 490: $From[w] \leftarrow v$
LINE 500: $MyQueue.Insert(w)$
LINE 510: **end-if** //
LINE 520: **end-for-all**
LINE 530: $Color[v] \leftarrow$ BLACK
LINE 540: $Post[v] \leftarrow$ ++CFL
LINE 550: **end-while** `not` $MyQueue.IsEmpty$

LINE 560: **end-algorithm**

	$Color$	Pre	In	$Dist$	$From$	$Post$
S						
A						
B						
C						
D						
E						
F						
G						
H						



[3] Or, if no graph is given, use the same one we used for the DFS-algorithm or create one or two yourself. Make sure it has a cycle as well as some 'dangling' legs.

3.2 Suggested Exercises

SE.25: Repeat the first algorithm `BFSNums` ($MyGraph$, $start$), but now replace the statement:
LINE 450: **if** $Color[w] ==$ WHITE **then** by the statement (but note that the algorithm may not always work correctly.)
LINE 1450: **if** $Color[start] ==$ WHITE **then**

SE.26: Repeat the first algorithm `BFSNums` ($MyGraph$, $start$), but now replace the statement:
LINE 450: **if** $Color[w] ==$ WHITE **then** by the statement (but note that the algorithm may not always work correctly.)
LINE 2450: **if** $Color[w] ==$ GRAY **then**

SE.27: Repeat the first algorithm `BFSNums` ($MyGraph$, $start$), but now replace the statement:
LINE 450: **if** $Color[w] ==$ WHITE **then** by the statement (but note that the algorithm may not always work correctly.)
LINE 3450: **if** $Color[w] ==$ BLACK **then**

SE.28: Repeat the first algorithm `BFSNums` ($MyGraph$, $start$), but now replace the statement:
LINE 500: $MyQueue.Insert(w)$ by the statement (but note that the algorithm may not always work correctly.)
LINE 1500: $MyQueue.Insert(start)$

SE.29: Use a BFS inspired algorithm to detect whether or not the graph has a cycle.

SE.30: Use a BFS inspired algorithm to find in-degrees of the nodes.

SE.31: Use a BFS inspired algorithm to find out-degrees of the nodes.

SE.32: Use a BFS inspired algorithm to construct a BFS-spanning tree. Add an additional Label to each node that tracks and records the number of children in this particular spanning tree.

SE.33: Consider a *knight* (as a chess piece like ♘) on a grid. A knight can move from it's current position to a new position, by making 2 horizontal steps followed by one vertical step, or by making 2 vertical steps followed by one horizontal step. For instance, from the current position, the potential next positions are marked by 1 (a distance of 1-knight hop). Use BFS to mark *all* table-entries with their knight-hop distance, (that is, with the lowest number of knight-hops that it would take to go from the marked (at distance 0) to that particular position). You can stop as soon as you have reached the King (♔). Thus: put a "2" inside all the squares at a "2"-hop distance, a "3" in the squares at a "3"-hop distance, and so on.

			1						
	♘ 0							♔	
			1						
1		1							

SE.34: Draw a grid with 4 rows and 9, each one is identified as $(r, c), r = 1 \ldots 4\ c = 1 \ldots 9$. Place a knight (chess piece) on location $(3, 3)$. What is the fewest number of (legal) moves for the knight from $(3, 3)$ to $(3, 8)$? What is the fewest number of (legal) moves for the knight from $(3, 3)$ to $(3, 8)$, if squares must be avoided such that the row and column number, when added together, is a multiple of 5 ?

SE.35: What can be said about the spanning tree constructed along with the BFS traversal, is it a free tree? a rooted tree? an ordered rooted tree? a binary tree? If you invoke the same algorithm with the same graph, but with different nodes as a starting node: Is the spanning tree always the same? always different? Sometimes the same, sometimes different? Is the BFSArray always the same? always different? Explain your answers and give examples or counter examples.

3.3 BFS Other Applications?

Many shortest path algorithms are essentially generalizations of the BFS algorithm, these are covered later. Future editions of this note will briefly go over some applications, mainly drawn from social networks where distance (i.e. in terms of "*hop count*") is important between friends, professionals, servers, and so on. Many other applications come from making smart choices, such as in games. In addition to the chess board example, use your imagination for a mutilated, non-square or expanded chessboard), or other games such as chutes-and-ladders.

4 DFS inspired Topological Sort

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.

Given a directed acyclic graph G (DAG). A Topological Ordering of the nodes is a linearization of the nodes, such that for all pairs of nodes v and w: if there is a path from node v to node w in the graph G, then v is listed before w in the Topological Sort.

Visually: If the nodes are listed topologically, then all directional arrows come from left to right, either to (right-) neighbors, or farther down the right.

Yet another viewpoint: This is a linearization of data for which only a partial order is defined, and not a total order, such that *if* $a < b$ for this (partial) order, then a precedes b in the linearization.

The algorithm for TOPSORT is a straight adaptation from the DFS, where the pre-processing is omitted, and where elements are inserted as first element of a list during the the post-processing. When a node is 'post-processed', then all the nodes that can be reached form the nodes have already been 'post-processed', (and thus already inserted in the list). An additional driver is needed to make sure all nodes are considered. The algorithm is presented in it's most barren form, with all bells-and-whistles' removed:

```
LINE 100:  Algorithm TopSort(graph MyGraph)
LINE 110:  int ColorLabel[n] ← {WHITE};          /* A global Array, indicating a node's color (initially WHITE)
LINE 120:  list MyList.Create;                   /* A global List; will contain the topologically sorted elements

LINE 200:  Algorithm TopSortHelper.v1(graph MyGraph, node start)
LINE 210:  //Precondition: ColorLabel[start] == WHITE //
LINE 220:  ColorLabel[start] ← GRAY
LINE 230:  for all WHITE w adjacent from start, selected alphabetically do TopSortHelper.v1(MyGraph, w)  end-for all
LINE 240:  MyList.InsertAtFirst(start)
LINE 250:  end-algorithm TopSortHelper.v1

LINE 500:  for all WHITE v ∈ V do TopSortHelper.v1(MyGraph, v)  end-for all
LINE 510:  //The list MyList now contains the nodes of the graph in topological order. //
LINE 520:  return(MyList)
LINE 530:  end-algorithm TopSort
```

Whereas the brute-force algorithm would seek the first node(s) in a topological sort, the source-nodes with their in-degree being zero. This DFS-inspired algorithm seeks the identify the last node(s) in a topological sort, the sink-nodes with their out-degree being zero. This last algorithm is sometimes attributed to a paper by R. Marimount, whose algorithm actually detects cycles in a directed graph by iteratively removing all sources and sinks in each iterative step. [4]

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

4.1 Suggested Exercises

SE.36: Construct a directed acyclic graph (DAG) with some 10 nodes and some 15 edges, randomly naming /numbering the nodes. Now find a topological sort of this DAG.

SE.37: A topological Sort is defined as a sequence of nodes, but neighboring nodes in the sequence may (or may not) be adjacent to one another in the graph, so it is not a path. If a topological sort is also a path, is it then unique? Prove or give a counter example.

SE.38: The topological sort of a directed acyclic graph may not be unique. In fact, it is only unique if the DAG itself is a single feedforward string of nodes, and it is no longer unique if there is at least one node with an out-degree of two or more. But you could perhaps have made different choices for start-node or for adjacent nodes. This raises an interesting problem: Is a particular given sequence of nodes a topological sort? Design and analyze an algorithm that checks (and prints YES/NO) when a proposed sequence of nodes is indeed a Topological Sort of a given a graph.

[4] Rosalind B. Marimont, A New Method of Checking the Consistency of Precedence Matrices, JACM, Vol 6 Issue 2, April 1959, Pages 164-171

4.2 Topological Sort of a Binary Search Tree

Suppose you have a binary search tree, and you want to make a copy of it that is structurally (concretely) equal. We have seen before that there are several algorithms that can be used for this purpose:

1. Use the PreOrder Traversal (together with the knowledge that the tree is a search tree).
2. Use the PostOrder Traversal (together with the knowledge that the tree is a search tree).
3. Use the LevelOrder Traversal and insert the elements back in that order.
4. More generally, use a Pre-/In-/ or Post-order traversal where "empty" sub-trees are physically represented by an "empty token".

The first few methods can be generalized to: Take any "Topological Order" of the binary search tree, and re-insert the elements back into a new binary search tree in that order. Indeed: the parent precedes the children in a topological ordering, so the parent is re-inserted into a new tree before it's children. So the structure of a BST can be recovered from any topological sort of the BST.

4.3 Counting the number of topological orders of a binary search tree

This can perhaps be used to find out how many permutations of n numbers would result into an identical BST: There are $n!$ different permutations, and only a Catalan number, $C(n) = \frac{1}{n+1}\binom{2n}{n}$ different binary trees. For instance, with $n = 3$, there are $3! = 6$ different permutations and $C(3) = 5$ different binary structures with 3 nodes. This is so, because "ABC" and "ACB" are two different topological orders of "one" binary tree.
So, given a particular binary tree structure, how many different topological sorts can be made form the given structure? If this can be found, then it is also equal to the number of permutations that give rise to an identical BST. Let $\mathcal{N}(root)$ be this number, and let the $root$ be the i^{th} element in rank-order, $\mathcal{R}(root) = i$, the $root$ must come first, followed by the remaining elements, and

$$\mathcal{N}(root) = \binom{n-1}{i-1} \cdot \mathcal{N}(leftchild) \cdot \mathcal{N}(rightchild)$$

Note however that this is far from a "normal" recurrence relation where the argument is an integer. In this case, the argument is a binary tree structure, which is given, and so is the structure of its subtrees. Thus, more correctly,

$$\mathcal{N}(T) = \binom{n_{\text{left}} + n_{\text{right}}}{n_{\text{left}}} \cdot \mathcal{N}(L) \cdot \mathcal{N}(R)$$

where T is the Binary tree under consideration, and L and R are its children, and where $n = n_{\text{left}} + 1 + n_{\text{right}}$ is the size of T.
We will no longer pursue this here.

4.3.1 Suggested Exercises

SE.39: (Graduate students) Can you write a tree-traversal algorithm that also counts the number of permutations that would also result in this BST?

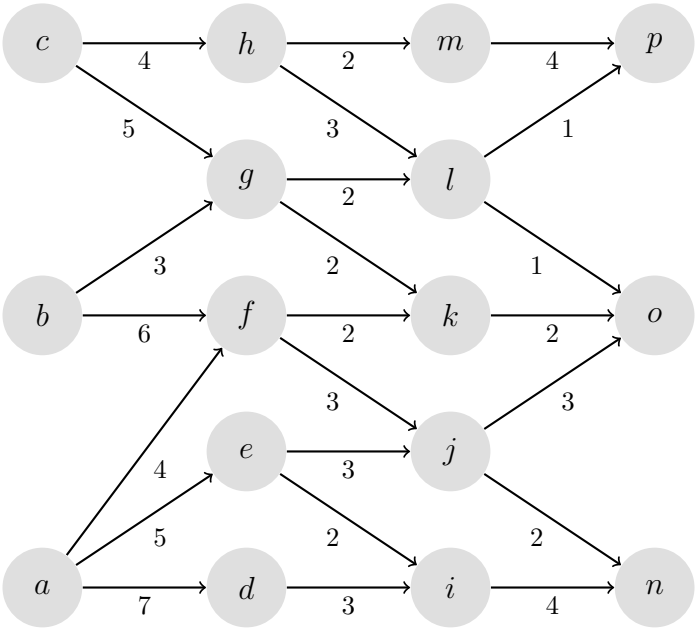
5 PERT Scheduling uses Topological Sort

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.

A prime example for a topological traversal of a directed graph is the determination of a PERT schedule. Suppose you are faced with a scheduling situation where packets need to be send from one node to another, and where traveling times between nodes is fixed. At each node, the packets from all the incoming connections are collected and redistributed towards their final density. Thus at each node, all incoming packets must have been received before a new transmission can be made towards the next node. So this is not a shortest path situation. Rather, it is a longest path situation which can be solved because this is an acyclic directed graph. For each node, introduce a label $EC[w]$, representing the earliest time that a new transmission can depart from node w. The value for this label can be determined as

$$EC[w] = \begin{cases} 0 & \text{if the in-degree of node } w \text{ is } 0 \\ \max\limits_{\text{all } x} \{\ EC[x] + cost[x,w]\} & \text{otherwise, where all edges } \langle x, w\rangle \in E \text{ are considered} \end{cases} \quad (1)$$

This is a dynamic programming situation and all the values $EC[x]$ should be determined first, for all nodes x that are a predecessor of w. But this is exactly the topological order that was determined in the earlier section.



	$EC[\ ]$	$LC[\ ]$
a	0	0
b	0	0
c	0	0
d	7	7
e	5	6
f	6	6
g	5	7
h	4	4
i	10	10
j	9	9
k	8	10
ℓ	7	9
m	6	6
n	14	14
o	12	12
p	10	10

	$EC[\ ]$	$LC[\ ]$
a		
b		
c		
d		
e		
f		
g		
h		
i		
j		
k		
ℓ		
m		
n		
o		
p		

For this example, if the nodes a, b and c start their transmissions at the same time, then the earliest time that the transmissions have arrived at the terminals n, o and p is 14, 12 and 10. What is the latest possible time that a node must transmit to the next node, without jeopardizing this earliest arrival time? Consider e.g. node ℓ: The earliest time it can transmit is $EC[\ell] = 7$, yet it can wait till 9 units without changing the end result (why?). What would be the latest time that a node *must* transmit without jeopardizing this earliest completion time for any one of the terminals? For each node, introduce a second label $LC[v]$, representing the latest time that a new transmission must depart from node v. The value for this label can be determined as

$$LC[v] = \begin{cases} EC[v] & \text{if the out-degree of node } v \text{ is } 0 \\ \min\limits_{\text{all } y} \{\ LC[y] - cost[v,y]\} & \text{otherwise, where all edges } \langle v, y\rangle \in E \text{ are considered} \end{cases}$$

Again, this is a dynamic programming situation and all the values $LC[y]$ should be determined first, for all nodes y that follow v in a topological order: update these values in a reverse topological order. Thus $LC[v]$ indicates the latest time a transmission must start without jeopardizing any individual terminal node. By construction, there is a nodes, say v, such that $EC[v] == LC[v]$. In fact, there are several nodes with that property, and these nodes are called the *critical nodes* and the edges that connect the critical nodes form the *critical path*. Any unexpected delay along the critical path will immediately impact the transmissions to at least one terminal. Similarly, any improvement along such an edge will immediately improve the overall transmission.

The time complexity for all of this is dominated by the time complexity of doing a topological sort (which in turn is identical to the DFS, $T_{\text{DFS}}(n,m) = (n+m)$, since the graph is directed). This Topological Sort is then stored and used twice: once in forward direction to compute the EC-labels, and another time in reverse direction to compute the LC-labels. Assuming that the data structures that are used also

support retrieval of all these edges without overhead, the complexity for determining EC and LC is $T_{\text{EC}}(n,m) = T_{\text{LC}}(n,m) = (n+m)$ and for a total of $T_{\text{PERT}}(n,m) = 3n + 4m$, if adjacency lists are used to implement the graph.

LINE 100: **Algorithm** `PERT` (**graph** $MyGraph$)
LINE 110: **int** $EC[n] \leftarrow 0$; /* A global Array, indicating a node's Earliest Completion Times (initially 0)
LINE 120: **int** $LS[n] \leftarrow 0$; /* A global Array, indicating a node's Latest Start Times (initially 0)

LINE 130: **for all** w in topological order **do**
LINE 140: $$EC[w] \leftarrow \begin{cases} 0 & \text{if the in-degree of node } w \text{ is } 0 \\ \max\limits_{\text{all } x} \{\ EC[x] + cost[x,w]\} & \text{otherwise, where all edges } \langle x,w\rangle \in E \text{ are considered} \end{cases}$$
LINE 150: **end-for all**

LINE 160: **for all** v in reverse topological order **do**
LINE 170: $$LS[v] \leftarrow \begin{cases} EC[v] & \text{if the out-degree of node } v \text{ is } 0 \\ \min\limits_{\text{all } y} \{\ LS[y] - cost[v,y]\} & \text{otherwise, where all edges } \langle v,y\rangle \in E \text{ are considered} \end{cases}$$
LINE 180: **end-for all**

LINE 190: **end-algorithm** `TopSort`

The nodes v with $EC[v] = LS[v]$ are called critical nodes, and the path connecting critical nodes is called the critical path. Along this path, you cannot tolerate any unexpected delay, with jeopardizing the total project.

5.1 Suggested Exercises

SE.40: In the above PERT-graph, change some values, e..g. $(b,f) = 5$, $(f,k) = 3$ and $(k,o) = 4$, and make the changes in the table.

SE.41: Let us change the final question in the PERT schedule a little: Suppose the "system" is only allowed to shut down after all (systemwide) transmissions have been received. This gives additional leeway between the individual terminals. In this case, introduce another node, a fictitious node q, and edges (n,q), (o,q) and (p,q), all with costs 0, and repeat the process.

6 DFS inspired Articulation Point Detection

Graduate Students Only

> *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.*

(This needs to be fine-tuned and Bi-connectivity needs to be added)

Given a connected simple graph, then an articulation point[5] is a node s such that, if it were to be removed, the graph breaks apart in two or more disconnected components. A good motivation to find articulation points is to detect vulnerabilities in any connection network: if the node is removed (or disabled, such as 'down for repairs') then there are at least two other remaining nodes in the network that are no longer connected. Thus, s is an articulation point if there are at least two other nodes, u_1 and u_2 such that s is on every path that connects these two nodes. Detecting articulation points is a challenge, certainly if you want to exhaustively check for all possible paths for all pairs of nodes. An articulation point is a structural property of the graph, and thus independent of any labeling or any traversals. Yet, the DFS can be used to detect whether or not a node s is (or is not) an articulation point by taking advantage of the DFS-numbering scheme. Although it is true that different choices of starting nodes, and different order of visiting neighboring nodes will lead to different spanning trees, they all have this in common: There is a root, there are nodes with both a parent and children, and there are leaves.

1. A root is an articulation point if-and-only-if it has two or more children. There are several ways how this can be incorporated.
2. A leaf is never an articulation point.
3. An internal point has both a parent and one or more children. Let s be this internal node, and $Num(s)$ it's rank according to the order that nodes are discovered during the DFS. We now have a perfect partition into three: The node s, the earlier nodes, and the later nodes. Node s was discovered at rank $Num(s)$. All earlier nodes have rank strictly less than $Num(s)$ and are already connected in the spanning tree under construction and as such form an equivalence set, while the later nodes have a rank strictly larger than $Num(s)$. Remember, s is an articulation point if there are at least two other nodes, u_1 and u_2 such that s is on every path that connects these two nodes. So take u_1 as any node with a number less than $Num(s)$ (e.g. the parent of s), and try to find a u_2 as a descendant of s. The node s is *not* an articulation point if every descendant of s has an alternate connection with u_1, that is, using a *back edge*. The DFS algorithm will thus be adjusted by tracking the connections using back edges. This is accomplished using a label that is traditionally called $Low(s)$: the smallest of the numbers $Num(vv)$ where uu is a descendant of s and (uu, vv) is a back edge. The information is updated in a "*post order*" fashion, so that $Low(s)$ is considered for updating when any child (in the DFS spanning tree under construction) is colored BLACK.

This can be accomplished by the introduction of additional labels:

$Num[s]$ These is the ranking of the node, assigned in the oder that nodes are discovered. The root has number 1, and they are assigned in pre-order fashion.

$Predecessor[w]$ This is the parent of a node w in the DFS tree. If w is the root, then this field is either empty or points to itself. The labels are assigned in pre-order fashion and are used to distinguish between a tree-edge and a back-edge.

$Children[s]$These keep track of the number of children for each node s, and is in the end only needed to determine whether or not the root is an articulation point. They are initialized as 0 and are updated in pre-order fashion.

$Low[s]$ This node with lowest rank that can be reached by zero or more tree-edges and one back-edge, as explained above.

This $Low[s]$ is initialized as $Low[s] \leftarrow Num[s]$ when the node s is discovered,

This $Low[s]$ is updated in a post-order fashion with information gathered from each w that is adjacent to s:

$$\begin{cases} Low[s] \leftarrow \min\{Low[s], Num[w]\} & \text{if } (s,w) \text{ is a back edge} \\ Low[s] \leftarrow \min\{Low[s], Low[w]\} & \text{if } (s,w) \text{ is a (spanning) tree edge} \end{cases}$$

The value for s is finalized when all adjacent nodes have been explored and the color switches to BLACK.

Note that the value of $Low(s)$ is not needed for determining wether or not node s itself is an articulation point, but the value is needed to determine whether the parent of s (or earlier ancestor) is an articulation point.

[5] Sometimes called cut point or separation point

ArticulationPointsFinder

LINE 100: **Algorithm** `ArticulationPointsFinder(`**graph** $MyGraph$`)`
LINE 110: **int** $CFL \leftarrow 0$, /* global variable to Count First & Last
LINE 120: **bool** $ArtNode[v] \leftarrow \{$FALSE$\}$; /* A global Array; a node is NOT an ArtPoint by default.
LINE 130: **int** $Num[v] \leftarrow \{0\}$; /* A global Array, indicating the rank of the node in discovery process
LINE 140: **int** $Low[v] \leftarrow \{0\}$; /* A global Array, see above
LINE 150: **int** $CameFrom[v] \leftarrow$ NILL; /* A global Array, pointing to the parent of a node in the DFS-ST
LINE 160: **int** $Children[v] \leftarrow \{0\}$; /* A global Array, counting the number of children per node.

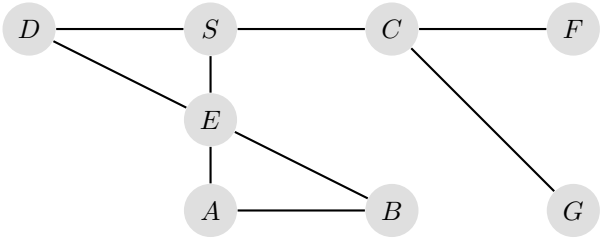
LINE 200: **Algorithm** `FindArtPointsHelper(`**graph** $MyGraph$`,` **node** $start$`)`
LINE 210: //Precondition: $Color[start] ==$ WHITE //
LINE 220: $Color[start] \leftarrow$ GRAY
LINE 230: $Num[start] \leftarrow$ ++CFL
LINE 240: $Low[start] \leftarrow Num[start]$

LINE 300: **for all** w **adjacent to** $start$ and selected alphanumerically **do**
LINE 310: **if** $Color[w] ==$ WHITE **then do** /* A new child w of $start$ has been discovered
LINE 320: $CameFrom[w] \leftarrow start$ /* and the edge $(start, w)$ is a new tree edge
LINE 330: $Children[start]$++
LINE 340: `FindArtPointsHelper(`$MyGraph$`,` w`)`
LINE 350: **if** $(Num[start] \leq Low[w])$ **then** $ArtNode[start] \leftarrow$ TRUE **end if**
LINE 360: $Low[start] \leftarrow \min\{Low[w],\ Low[start]\}$
LINE 370: **end do** /* Done processing a newly discovered child w of $start$
LINE 380: **else-if** $(CameFrom[start] \neq w)$ /* w was already discovered, and $(start, w)$ is a back-edge
LINE 390: **then** $Low[start] \leftarrow \min\{Low[w],\ Low[start]\}$ **end if**
LINE 400: **end-for all**
LINE 410: $Color[start] \leftarrow$ BLACK
LINE 420: **end-algorithm** `FindArtPointsHelper`

LINE 600: **begin main line** `ArticulationPointsFinder`
LINE 610: $start \leftarrow$ pick any node; (for this course: take S) /* To get the Articulation Point Finder started
LINE 620: `FindArtPointsHelper(`$MyGraph$`,` $start$`)`
LINE 630: //All articulation points have been identified, except perhaps for the root, which is next:
LINE 640: **if** $Children[start] > 1$) **then** $ArtNode[start] \leftarrow$ TRUE **end-if**
LINE 650: //All articulation points have now been identified.
LINE 660: **end-algorithm** `ArticulationPointsFinder`

6.1 Suggested Exercises

SE.42: You are asked to write the resulting values for the labels as indicated for the graph as given, or for the graph that you yourself should create. Make sure it has a cycles as well as some 'dangling' legs.



7 Matrix-based Algorithms for Shortest Paths and Spanning Trees

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.

With A the adjacency matrix, then the standard matrix multiplication is

$$A^2[i,\ j] = \sum_{k=1}^{n} A[i,k] \times A[k,j] \tag{2}$$

If the matrix A starts out as "0/1" entries, but still integers, then A^2 counts the number of 2-step connections between i and j. The matrix A^3 counts the number of 3-step connections, and so on. If the matrix A^n is identically zero, then there are no cycles in the graph.

If the matrix A starts out as "0/1" entries, but now "Boolean" and if "$\times$" is interpreted as "AND" and if "+" is interpreted as "OR", then A^2 is a matrix of Boolean entries indication "YES/NO" the presence of a 2-step connection between i and j.

Back to the matrix A starting out as "0/1" integers, then

$$A^2[i,\ j] = \min_{k=1}^{n} \left\{ A[i,k] + A[k,j] \right\} \tag{3}$$

is the matrix representing the shortest connection between i and j.

Raising an adjacency matrix A of a simple graph G to the k^{th} power (with perhaps different interpretations of $+\times$ gives the number paths of length k between two vertices i and j, or indicates the existence of such a path, or finds the k path of minimal length, or determines connectivity, graph closure, and so on.

This can be easily adjusted to find matrix-based algorithmic solutions to many path problems, spanning tree problems, flow problems, and so on. Now if you also replace the "multiplication" by the logical "AND" and "addition" by the logical "OR" (or the optimizing "MIN" or "MAX") then you have a solution framework for many graph based solutions. This is not often done in practice on a large scale, since a single matrix multiplication takes $\mathcal{O}(n^3)$ time. Most problems require multiple matrix multiplications, and the complexity would be increased to $\mathcal{O}(n^3\ \lg n)$ or $\mathcal{O}(n^4)$. The good thing is that this would give an upper bound.

8 Dijkstra's Shortest Path Algorithm

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.

There are many shortest path type problems, the most commonly known is Dijkstra's SPA between two given nodes (the so-called single source single destination). We will cover this algorithm (and in particular efficient implementations thereof) as well as the extension to single source - all destinations. Dijkstra's algorithms are only provably correct whenever all link-weights (read: distances) are strictly positive. Should there be links whose link-weights are possibly negative, then the algorithm known as the Bellman-Ford Algorithm can be used instead. It is of course more expensive, and is available only for the single source - all destinations. This algorithm finds the shortest path between a starting node and all other nodes, even in the presence of negative weights. Should a negative cycle be present, then the algorithm can detect this as well. Finally, the algorithm called the Floyd-Warshall Algorithm computes the shortest paths between all pairs of source-destination combinations simultaneously, and is probably the easiest to implement and test.

The BFSskeleton is designed for graphs where all edges have equal weights. If it is to be extended to include possibly weighted edges (but we will allow only positive weights for now), then this must be altered. The first difference is that the notion of distance must be changed. In a BFS, the distance was equal to the pathlength between the node nodes, the so-called 'hop-distance'. In a weighted graph, this distance is now defined as the sum of the weights of all the edges participating in the path, and there may not be a good relationship between the 'hop-distance' and the 'weight-distance'. Note that we will only allow positive weights for now, but we will allow these weights to be anything (i.e. even the triangle inequality may not be satisfied).

1. Every node (say node x) has a distance-label, $DistanceToS[\ x\ ]$, which tracks the distance from the $StartNode$ to node x. It is initialized as "∞" and is continually decreased until it is selected as $NextNode$ and the $DistanceToS$ is optimal. At this time the color of the node will change to BLACK.
2. The notion of a 'queue' was appropriate in the BFS because it guaranteed that the 'shortest' hop count path was selected, this is the order in which the nodes are 'discovered'. Such a simple queue is no longer appropriate as we need to select a node from all the nodes that are not yet BLACK with the smallest $DistanceToS$. This information must be available and stored, and is implemented with various data structures to improve the performance. We present the priority queue, implemented with a lazy MinHeap, but other data structures give slightly better performance.

First, we show the Dijkstra's Shortest Path Algorithm in it's most basic form, stripped from all bells and whistles, and without any data structures that make it perform well. We present various improvements with their time complexities later. In this bare-bones version, there are only two colors: BLACK and WHITE, where a BLACK node indicates that the $DistanceToS$-label for that node can no longer be improved and has reached it's optimum (i.e. minimum) value.

LINE 100: **Algorithm** `basic-DSPA(`**graph** $MyGraph$**, node** $StartNode$**, node** $StopNode$)
LINE 110: **int** $ColorLabel[n] \leftarrow \{$WHITE$\}$; /* Initially all colors WHITE in a global Array
LINE 120: **real** $DistanceToS[n] \leftarrow \{$*infinity*$\}$; /* Initially all distances are set to a large value

LINE 300: $ColorLabel[StartNode] \leftarrow$ BLACK /* The start node is initialized
LINE 310: $DistanceToS[StartNode] \leftarrow 0$ /* The start node is initialized

LINE 320: $NextNode \leftarrow StartNode$
LINE 400: **repeat**
LINE 410: **for all** w **adjacent from** $NextNode$ **do** `BasicRelaxEdge(`$NextNode, w$`)` **end-for-all**
LINE 420: **select** $NextNode$ as the WHITE node with minimal $DistanceToS$-label.
LINE 430: $ColorLabel[NextNode] \leftarrow$ BLACK
LINE 440: **until** $(NextNode == StopNode)$

LINE 500: **end-algorithm** `basic-DSPA`

where the helper function of `BasicRelaxEdge(`$NextNode, w$`)` at this time is the one-line statement

LINE 10: **Algorithm** `BasicRelaxEdge(`**node** $NextNode$**, node** w)
LINE 20: $DistanceToS[w] \leftarrow \min\{DistanceToS[w], DistanceToS[NextNode] + LinkLength[NextNode, w]\}$
LINE 30: **end-algorithm** `BasicRelaxEdge`

This checks whether or not the value of $DistanceToS[w]$ can be reduced with $DistanceToS[NextNode] + LinkLength[NextNode, w]$. This helper function will get more functionality shortly, and most certainly will include tracking information to actually construct a shortest path, should that be needed.

This basic version of Dijkstra's Shortest Path Algorithm is easy to prove correct, and easy to hand-simulate. It is also easy to see that the **repeat**. . .**until** is executed $n-1$ times if the $StopNode$ is the last node to be selected. This is the worst case, and this is what we will assume for the analysis of the time complexity.

8.1 Time Complexity

First let us conclude that the number of times that `BasicRelaxEdge` is executed is equal to m, the number of edges in the graph: As the **repeat**. . .**until** gets executed, the value of $NextNode$ will iterate through the node-set, and all edges adjacent to the current $NextNode$ will be explored and relaxed. However it should be noted that, íf the graph is implemented with an adjacency matrix, then there is a hidden cost of n^2 `EdgeDetections`.

Now consider the $NextNode$-**select**ion: each new value of $NextNode$ is the result of selecting a WHITE node with minimal $DistanceToS$-label. This is a double (and independent) selection criterion and íf the $ColorLabel$ or the $DistanceToS$-labels (or both) are maintained in a (linear) array, then this takes n "double" inspections for each iteration, for a total of $2n^2$ comparisons. So this very basic version of Dijkstra's SPA is $\sim 3n^2$ when implemented with an adjacency matrix, or $\sim 2n^2 + m = \Theta(n^2)$.

Although the basic version of Dijkstra's algorithm is easily understood in principle, there are several variations on how to more efficiently implement it, and each one has perhaps a different performance profile (average or worst case time complexity). At our university, students are exposed to Dijkstra's shortest path algorithm in earlier courses, where correctness and basic understanding are stressed. The basic version suffices for this and students do generally well with a hand-simulation of the basic version demonstrating they understand the *algorithm*. Of course, the number of nodes remains fairly small in these cases and and the priority queue is implemented as a a simple liner array, from which the "smallest distance label" is easily found by visual inspection.

The current course places more emphasis on the *efficient implementation* of the algorithm, and advocates using a priority queue which is implemented with a (min-)heap, see below. This drops the performance of the algorithm from $\Theta(n^2)$ down to $\Theta(m \lg n)$, which is great once you understand correctly the workings of a heap. Unfortunately, the benefit of using a more advanced data structure becomes only clear for larger values of the problem size. This is also the case for this implementation, with the added disadvantage that heap-manipulations are not as intuitive for human beings as straight arrays, whereas computers simply don't care.
Illustrating an efficient implementation of Dijkstra's algorithm with a MinHeap for a small graph is often not convincing (and perhaps even confusing) but using a large graph is time and space consuming. An illustration follows the improved algorithm.

8.2 Improved Implementations

If you want to reduce the complexity to below n^2 then you must 1. implement the graph with adjacency lists, and 2. you must reduce the number of comparisons. This latter is done through two independent approaches: Use the $ColorLabel$ GRAY to indicate the set of nodes that are candidates for the next minimal node. Furthermore, use a separate data structure just for the GRAY nodes to limit the number of comparisons between the candidates.

1. If the GRAY nodes are organized in a linear structure, then you still need a linear scan at each iteration. The complexity is brought down to $(n-1) + (n-2) + \ldots + 1 = {}^1\!/_2\, n^2$, thus giving a 50% savings over the basic version.
2. If the GRAY nodes are organized in a priority queue, and you implement this with a heap e.g., then you have only a logarithmic cost at each iteration. The priority queue can be implemented as either a minheap, a binomial queue, a Fibonacci Heap or other advanced priority queue. We use a "minheap" for this purpose, where the priority is the $DistanceToS$ label. An additional problem has to be solved: If the $DistanceToS[w]$ is improved for a node w that is already GRAY during a `BasicRelaxEdge` operation, then this improved value $DistanceToS[w]$ must be incorporated. But it's color is already GRAY, so the node is already in the "minheap".
   (a) Many authors solve this problem by extending the standard repertoire of MinHeap operations and include *priority management* operations, such as a "ImprovePriority" operation, which "locates" the node somehow inside the MinHeap and rearranges it (PerculateUp) to conform to the heap property, this keeps the size of the MinHeap the same. The are at most n entries in the MinHeap. The cost of a single "ImprovePriority" is the logarithm of it's size. Each time that an edge link $\langle NextNode, w\rangle$ is encountered, it could result in either an MinHeap-insertion or a MinHeap-ImprovePriority, so the total cost is $\sim m \lg n$. Add to this the cost of $n-1$ MinHeap-deletions, $\sim n \lg n$, to arrive at a cost of $(n+m)\lg n$ operations on the MinHeap. I have not included the cost of building and maintaining the additional structure to "Find/Locate" an internal node in a MinHeap.
   (b) Rather than extending the standard repertoire of MinHeap operations, we could also simply insert another copy of the node w to the MinHeap, but now together with an (improved) DistanceLabel. The MinHeap now contains tuples, $\langle node, distance\rangle$, so that there are perhaps multiple entries in the MinHeap for the same w-node, but with differing distances. The are at most m entries in the MinHeap. This means that the first time that w is served (i.e. removed) from the MinHeap, it's distance is in it's final value. The color of w is indeed still GRAY. It will be colored BLACK after processing it's adjacent neighbors. Any subsequent serves (i.e. removals) from the MinHeap that involve w, will have a distance that is larger, but the color of w is already BLACK, so we can just test the color and ignore this node w with it's obsolete distance value, and serve another

node. I will refer to this as a lazy heap and it is explained in an example below in the algorithm.

Each time that an edge link $\langle NextNode, w\rangle$ is encountered, it could result in a MinHeap-insertion, so the total cost is $\sim m\lg m - m$. Add to this the cost of up to m MinHeap-deletions, also $\sim m\lg m - m$, to arrive at a cost of $\sim 2m\lg m - 2m$ operations on the MinHeap. Add the cost of m relaxations, and we get $\sim 2m\lg m - m$. Notice that for a simple connected graph, m is between n and n^2, so that $\lg m$ is between $\lg n$ and $2\lg n$. The complexity is similarly between $\sim 2m\lg n$ and $\sim 4m\lg n$, both in the same order: $\Theta(m\lg n)$.

(c) The ideas from both options above can be combined in a MinHeap with a *lazy* "ImprovePriority", that is: the "ImprovePriority" is not incorporated at all, and only executed in a "just-in-time" fashion. The MinHeap has only nodes, their DistanceLabels are a simple array outside the heap-structure and consulted when needed. The algorithm is virtually identical to the one for option (b), except that the MinHeap-insertion needs to be adjusted. Rather than "inserting a new node into the sorted sequence from new-location to root" this now becomes: "inserting a new GRAY node into the sorted sequence of GRAY nodes new-location to root." In other words: simply ignore BLACK nodes when inserting. The analysis is pretty much similar to that of option (b), but proving correctness and doing a hand-simulation become a challenge. Yet, these are implementation options that are possible, and it is extremely important to document such implementation decisions very well for future generations of programmers/coders who need to maintain the legacy code you create.

(d) Yet additional options are possible: Rather then inserting into the "main"-priority queue directly, for each $SelectNode$ create a priority queue with all it's adjacent neighbors first. This "adjacent"-priority queue is then merged with the "main"-priority queue. In this case, you can use a binomial or Fibonacci queue. This will not be covered here, but please be aware of more advanced implementations of Dijkstra's SPA's.

Time Complexity Dijkstra SPA	$T^W(n,m) \sim$	$T^W(n,m) = \Theta(\ )$
Basic Algorithm Implemented with Adjacency Matrix, or simple array from which the $\min$ is taken	$\sim n^2$	$= \Theta(n^2)$
Improved Algorithm Implemented with Adjacency Lists, and Lazy MinHeap	$\sim 2m\lg m$	$= \Theta(m\lg n)$

8.3 Dijkstra implemented with a Lazy Heap

Dijkstra's Algorithm, where the GRAY nodes are implemented with a Lazy MinHeap, and where tracking information in maintained in the $CameFromLabel$s for every node in the graph.

LINE 100: **Algorithm** `LazyHeap-DSPA(`**graph** $MyGraph$**, node** $StartNode$**, node** $StopNode$`)`
LINE 110: **int** $ColorLabel[n] \leftarrow \{$WHITE$\}$; /* Initially all colors WHITE in a global Array
LINE 120: **real** $DistanceToS[n] \leftarrow \{$*infinity*$\}$; /* Initially all distances are set to a large value

LINE 300: $ColorLabel[StartNode] \leftarrow$ WHITE /* The start node is initialized
LINE 310: $DistanceToS[StartNode] \leftarrow 0$ /* The start node is initialized

LINE 320: $CameFrom[StartNode] \leftarrow NILL$
LINE 330: $ColorLabel[StartNode] \leftarrow$ GRAY (i.e. "in-the-priority-queue")

LINE 340: $MyPriorityQueue \leftarrow \mathbf{PriorityQueue.create}()$
LINE 350: $MyPriorityQueue.\mathbf{insert}(\langle StartNode, 0\rangle)$

LINE 400: **while** $MyPriorityQueue.\mathbf{IsNotEmpty}$ **do**

LINE 410: // Find the next node with minimal DistanceLabel $DistanceToS$ that is still GRAY,
LINE 420: // while discarding nodes with outdated Distance information, which is indicated by the color BLACK):
LINE 430: **Repeat** $\langle SelectNode, Dist\rangle \leftarrow MyPriorityQueue.\mathbf{remove}()$
LINE 440: **until** $ColorLabel[SelectNode] ==$ GRAY **end-repeat**

LINE 450: // The node $SelectNode$ has a distance $Dist$ that cannot be improved
LINE 460: $Color[SelectNode] \leftarrow$ BLACK
LINE 470: **if** $SelectNode == StopNode$ **then RETURN end if**

LINE 480: **for all** ($AdjNode$ **adjacent** to $SelectNode$ **with** $ColorLabel[AdjNode] \neq$ BLACK) **do**
LINE 490: `RelaxEdgeDijkstra(`$SelectNode$`,` $AdjNode$`)`
LINE 500: **end-for all**

LINE 510: **end while**

LINE 520: **end algorithm** `LazyHeap-DSPA`

where now the helper function of `RelaxEdgeDijkstra(`$NextNode, w$`)` has a little more functionality with a little more bookkeeping: It inserts elements into the MinHeap whenever they are first discovered, or whenever their distance can be improved. It also maintains a "came-from" data structure (a regular field, or a list if a minimal path is not unique).

LINE 600: **AlgorithmHelper** `RelaxEdgeDijkstra(`**node** $FromNode$`,` **node** $ToNode$`)`

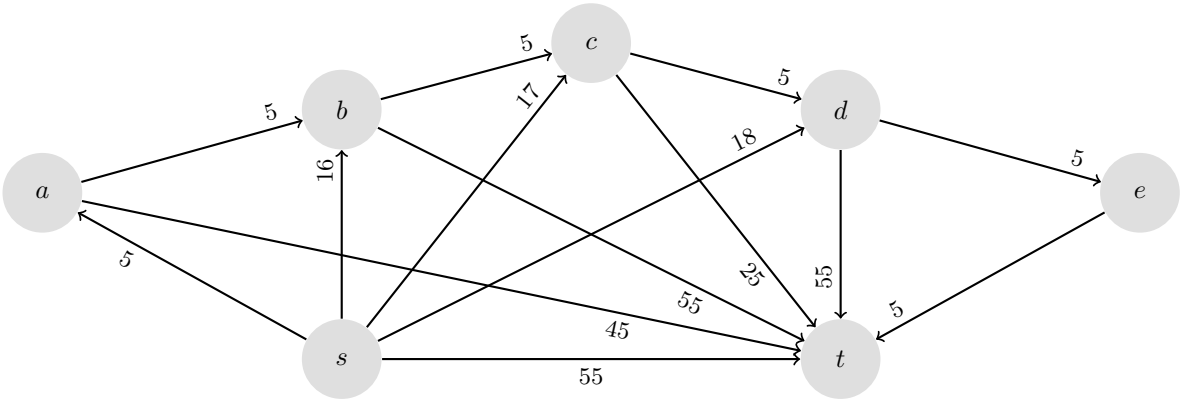
LINE 610: **if** $ColorLabel[ToNode] ==$ WHITE
LINE 620: **then** // the node is discovered. It needs to be Gray and included in the Heap
LINE 630: $ColorLabel[ToNode] \leftarrow$ GRAY // (i.e. "in-the-priority-queue")
LINE 640: $DistanceToS[ToNode] \leftarrow DistanceToS[FromNode] + Cost(FromNode, ToNode)$
LINE 650: $CameFromLabel[ToNode] \leftarrow FromNode$
LINE 660: $MyPriorityQueue.\mathbf{insert}(\langle ToNode, DistanceToS[ToNode]\rangle)$ // Pass both by value, not by reference.

LINE 670: **else if** $DistanceToS[ToNode] > DistanceToS[FromNode] + Cost(FromNode, ToNode)$
LINE 680: **then** // The Node is already in Heap, but an improved distance can be inserted.
LINE 690: $DistanceToS[ToNode] \leftarrow DistanceToS[FromNode] + Cost(FromNode, ToNode)$
LINE 700: $CameFromLabel[ToNode] \leftarrow FromNode$
LINE 710: $MyPriorityQueue.\mathbf{insert}(\langle ToNode, DistanceToS[ToNode]\rangle)$ // Pass both by value, not by reference.
LINE 720: **end if**

LINE 730: **end-algorithmHelper** `RelaxEdgeDijkstra`

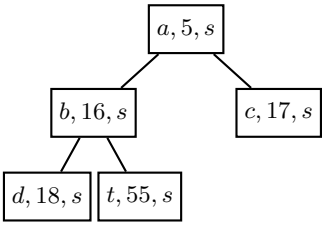
8.4 Dijkstra with a Lazy Heap – Example

At our university, students are exposed to Dijkstra's shortest path algorithm in earlier courses, where correctness and basic understanding are stressed. The number of nodes remains fairly small and the priority queue is implemented as a a simple array, from which the "smallest distance label" is easily found by visual inspection. The current course has more emphasis on efficient implementation of the algorithm, which uses a (min-)heap to implement the priority queue. This drops the performance of the algorithm from $\Theta(n^2)$ down to $\Theta(m \lg n)$, which is great once you understand correctly the workings of a heap *and* if the graph is large. Unfortunately, the benefit of using a heap only becomes clear for large graphs (my guess is: average heap size for the duration of the algorithm: 16 or larger), and using a heap is, in fact, more expensive for smaller graphs. So illustrating a heap-implementation of Dijkstra's Algorithm for a small graph is perhaps more confusing than enlightening. However, the various implementations of Dijkstra's SPA are at the core of internet routers and many of our alumni have found employment in this industry. Thus: here we go with an illustration of Dijkstra's SPA using a (specialized) Min-Heap. Keep in mind that following this example is extremely time consuming, simply because we are mere human beings. On the recent quizzes/exams of the last two semesters, I have only asked: "which elements are in the heap" and I no longer ask "which elements elements are in the heap, and where are they within the heap", which takes much less time.



Let us step through the algorithm:

1. First node s is inserted into the heap, and immediately removed again. It is now called $SelectNode$.
2. The color of this $SelectNode$, (s), is GRAY and all of s's five neighbors are WHITE so all are being `RelaxEdgeDijkstra`'d and all are placed in the heap. Technically, only two pieces of information, "Node" and "DistanceLabel" are needed for correct operation of the algorithm, but we will store extra information ("came from") for ease of understanding and store $\langle a, 5, s\rangle$ in the heap to indicate that node a was placed in the heap with DistanceLabel 5, and it was relaxed from s. We continue the tradition of inserting the nodes alphabetically, resulting in

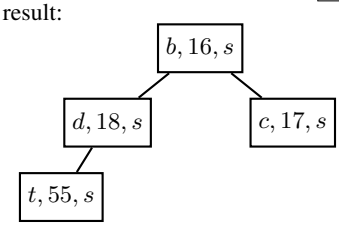


The Min-Heap as implemented as an array and it's visualization as a tree.

[$\langle a, 5, s\rangle$, $\langle b, 16, s\rangle$, $\langle c, 17, s\rangle$, $\langle d, 18, s\rangle$, $\langle t, 55, s\rangle$]

3. Node s is colored BLACK

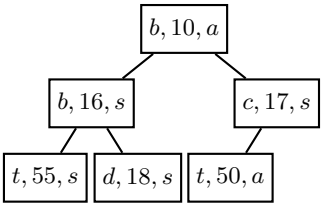
4. The item with lowest priority is $\boxed{a, 5, s}$ and it is served (i.e. removed) from the priority queue; the heap is slightly altered as a result:



The Min-Heap as implemented as an array and it's visualization as a tree.

[$\langle b, 16, s\rangle$, $\langle d, 18, s\rangle$, $\langle c, 17, s\rangle$, $\langle t, 55, s\rangle$]

5. The color of a is GRAY and the DistanceLabel for a is confirmed as being the smallest possible. All NON-BLACK nodes adjacent to a are being `RelaxEdgeDijkstra`'d. The node has only two such adjacent nodes (b and t) and both result in an improved $DistanceLabel$ (10 and 50) and both are added to the priority queue. After both have been inserted, the priority queue is



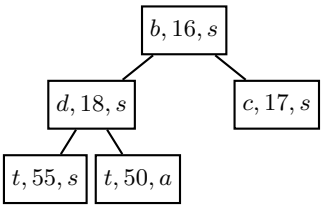
The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle b, 10, a\rangle, \langle b, 16, s\rangle, \langle c, 17, s\rangle, \langle t, 55, s\rangle, \langle d, 18, s\rangle, \langle t, 50, a\rangle\ ]$

Note that there are two nodes b inside the heap, each with a different DistanceLabel, and there are two nodes t inside the heap, also with a different DistanceLabel.

6. Node a is colored BLACK

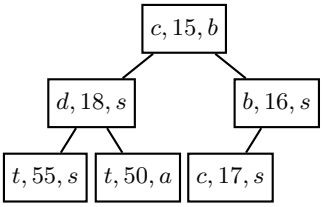
7. The item with lowest priority is $\boxed{b, 10, a}$ is served from the priority queue (i.e. removed); the heap is slightly altered as a result:



The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle b, 16, s\rangle, \langle d, 18, s\rangle, \langle c, 17, s\rangle, \langle t, 55, s\rangle, \langle t, 50, a\rangle\ ]$

8. The color of b is GRAY and the DistanceLabel for b is confirmed as being the smallest possible. All all the NON-BLACK nodes adjacent to b are being `RelaxEdgeDijkstra`'d. The node has only two such adjacent nodes (c and t) and only neighboring node c results in an improved $DistanceLabel$ (15). Thus $\langle c, 15, b\rangle$ is inserted in the priority queue/heap. The DistanceLabel for node t is not improved, and no further action is required. (In particular, no insertion in the heap). After the insertion of $\langle c, 15, b\rangle$, the priority queue is

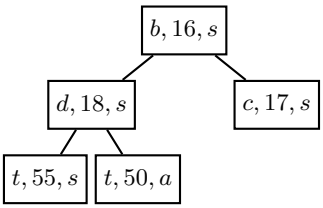


The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle c, 15, b\rangle, \langle d, 18, s\rangle, \langle b, 16, s\rangle, \langle t, 55, s\rangle, \langle t, 50, a\rangle, \langle c, 17, s\rangle\ ]$

9. Node b is colored BLACK

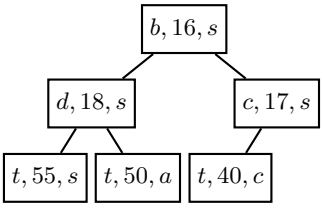
10. The next item to be served is $\boxed{c, 15, b}$ and the heap is slightly altered,



The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle b, 16, s\rangle, \langle d, 18, s\rangle, \langle c, 17, s\rangle, \langle t, 55, s\rangle, \langle t, 50, a\rangle\ ]$

11. The color of c is GRAY and the DistanceLabel for c is confirmed as being the smallest possible. All NON-BLACK nodes adjacent to c are being `RelaxEdgeDijkstra`'d. The node has only two such adjacent nodes (d and t). The $DistanceLabel$ for node d is not improved, but the $DistanceLabel$ for node t can be improved to 40, and $\langle t, 40, c\rangle$ is inserted into the heap.

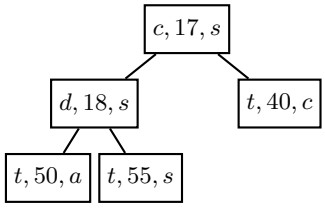


The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle b, 16, s\rangle, \langle d, 18, s\rangle, \langle c, 17, s\rangle, \langle t, 55, s\rangle, \langle t, 50, a\rangle, \langle t, 40, c\rangle\ ]$

12. Node c is colored BLACK

13. The next item to be served is $\boxed{b, 16, s}$ and the heap is slightly altered,

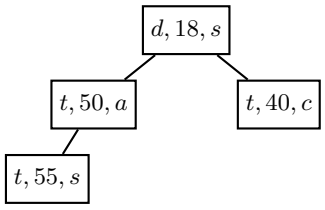


The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle c, 17, s\rangle, \langle d, 18, s\rangle, \langle t, 40, c\rangle, \langle t, 50, a\rangle, \langle t, 55, s\rangle\ ]$

14. The color of b is already BLACK so the item that was just served from the heap is no longer needed: it was superseded by an item with an improved DistanceLabel, and the improved Label was served earlier. This one is no longer needed, and no action is required.

15. The next item to be served is $\boxed{c, 17, s}$ and the heap is slightly altered,

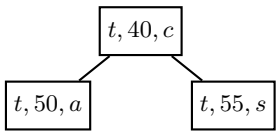


The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle d, 18, s\rangle, \langle t, 50, a\rangle, \langle t, 40, c\rangle, \langle t, 55, s\rangle\ ]$

16. The color of c is already BLACK and is no longer needed: it was superseded by an item with an improved DistanceLabel, and the improved Label was served earlier. This one is no longer needed, and no action is required.

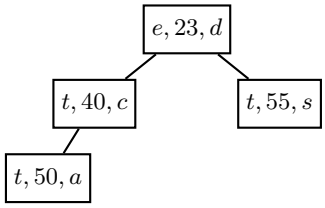
17. The next item to be served is $\boxed{d, 18, s}$ and the heap is slightly altered,



The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle t, 40, c\rangle, \langle t, 50, a\rangle, \langle t, 55, s\rangle\ ]$

18. The color of d is GRAY and the DistanceLabel for d is confirmed as being the smallest possible. All NON-BLACK nodes adjacent to d are being `RelaxEdgeDijkstra`'d. The node has only two such adjacent nodes (e and t) and only neighboring node e results in an improved $DistanceLabel$ (23). Thus $\langle e, 23, d\rangle$ is inserted in the priority queue/heap. The DistanceLabel for node t is not improved, and no further action is required. (In particular, no insertion in the heap). After the insertion of $\langle e, 23, d\rangle$, the priority queue is

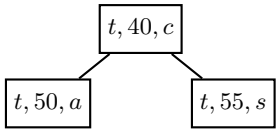


The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle e, 23, d\rangle, \langle t, 40, c\rangle, \langle t, 55, s\rangle, \langle t, 50, a\rangle\ ]$

19. Node d is colored BLACK

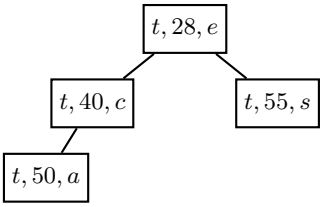
20. The next item to be served is $\boxed{e, 23, d}$ and the heap is slightly altered,



The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle t, 40, c\rangle, \langle t, 50, a\rangle, \langle t, 55, s\rangle\ ]$

21. The color of e is GRAY and the DistanceLabel for e is confirmed as being the smallest possible. All NON-BLACK nodes adjacent to e are being `RelaxEdgeDijkstra`'d. There is only one such adjacent node (t) and has an improved $DistanceLabel$ (28). Thus $\langle t, 28, e\rangle$ is inserted in the priority queue/heap:

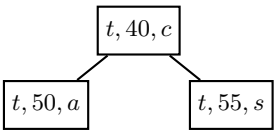


The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle t, 28, e\rangle, \langle t, 40, c\rangle, \langle t, 55, s\rangle, \langle t, 50, a\rangle\ ]$

22. Node e is colored BLACK

23. The next item to be served is $\boxed{t, 28, e}$ and the heap is slightly altered,

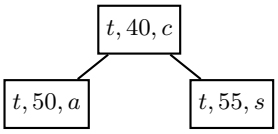


The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle t, 40, c\rangle, \langle t, 50, a\rangle, \langle t, 55, s\rangle\ ]$

24. The color of t is GRAY and the DistanceLabel for t is confirmed as being the smallest possible. There are no more nodes adjacent to t that are NON-BLACK.
25. Node t is colored BLACK
26. The algorithm now stops.

At the time that the terminal node t is colored BLACK, there are still some elements in the heap:



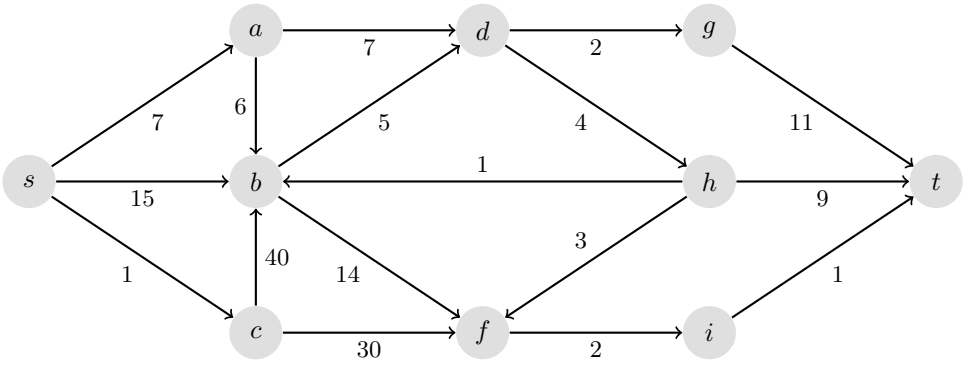
The Min-Heap as implemented as an array and it's visualization as a tree.

$[\ \langle t, 40, c\rangle, \langle t, 50, a\rangle, \langle t, 55, s\rangle\ ]$

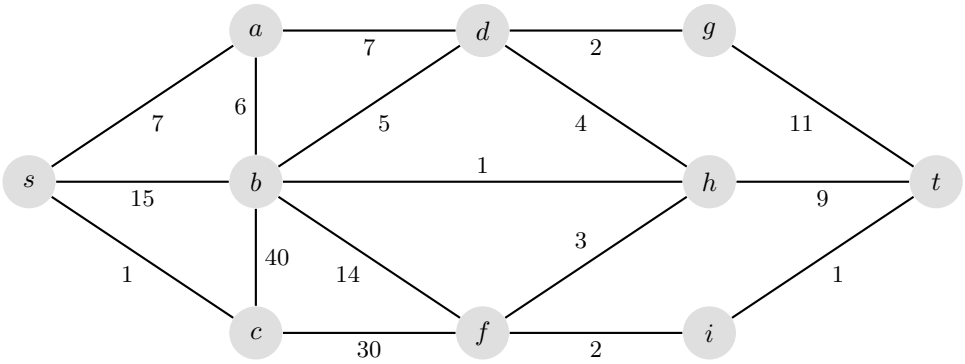
8.5 Suggested Exercises

SE.43: It turns out that only node t is still in the heap with outdated distance labels. The question is: Is this an indication that this is always true, or could it be that other nodes still reside in the priority queue/heap with outdated labels? Is it possible to change the value of the label between s and b so that t, x, a is still inside the heap for some specific value of x? Add an edge to the original graph so that there is another outdated node in the heap.

SE.44: Illustrate the execution of Dijkstra's Shortest Path Algorithm for the graph below, by labeling the nodes with the $DistanceToS$ Labels. Rather than implementing the priority queue with a $MinHeap$, feel free to implement the priority queue with a simple array. You are asked to show the content of the priority queue (either $MinHeap$ or array) at the time that the node t is colored BLACK.



SE.45: Repeat the exercise, but now the graph is undirected: Illustrate the execution of Dijkstra's Shortest Path Algorithm for the graph below, by labeling the nodes with the Distance Labels. Rather than implementing the priority queue with a $MinHeap$, feel free to implement the priority queue with a simple array. You are asked to show the content of the priority queue (either $MinHeap$ or array) at the time that the node t is colored BLACK.



SE.46: (Graduate students) Study the following algorithm. This is essentially a DFS, where edges are selected by minimum-weight-adjacent edge, instead of selecting them alphabetically.

Algorithm void `GreedyDFSSPA(`**graph** $MyGraph$**,** **node** $StartNode$`)`
//Precondition: $ColorLabel[StartNode] ==$ WHITE //
$ColorLabel[start] \leftarrow$ GRAY
for all v **in** $myGraph$.NodeSet **do** $ColorLabel[v] \leftarrow$ WHITE **end-for all**
$DistanceToS[StartNode] \leftarrow 0$
for all $ToNode$ **adjacent to** $StartNode$ and selected in order of increasing edge weight **do**
  if $ColorLabel[ToNode] ==$ WHITE **then**
    $ColorLabel[ToNode] \leftarrow$ BLACK
    $DistanceToS[ToNode] \leftarrow DistanceToS[FromNode] + Cost(FromNode, ToNode)$
    `GreedyDFSSPA(`$MyGraph,\ ToNode$`)`
  end-if
end-for all
end-algorithm `GreedyDFSSPA`

In particular:

1. Find a graph with 5 or 6 nodes such that this algorithm correctly identifies all shortest paths to all the nodes.
2. Find a graph with 5 or 6 nodes such that this algorithm does not correctly identify all shortest paths to all the nodes.
3. Determine the worst case time complexity if the graph is implemented with adjacency lists.
4. Determine the worst case time complexity if the graph is implemented with an adjacency matrix.

SE.47: (Graduate students) Study the following algorithm. This is essentially a BFS, where edges are selected by minimum-weight-adjacent edge, instead of selecting them alphabetically.
Assume all nodes are initially WHITE.

Algorithm `Greedy-BFS-SPA(`**graph** $MyGraph$**,** **node** $StartNode$`)`
//PRECONDITION: $ColorLabel[StartNode] ==$ WHITE //
queue $MyQueue.new$; /* A QUEUE TO IMPOSE AN ORDERING
Labels $ColorLabel[StartNode] \leftarrow \{$WHITE$\}$; /* A GLOBAL ARRAY, TO INDICATE THE COLOR OF A NODE

$ColorLabel[StartNode] \leftarrow$ GRAY
$DistanceToS[StartNode] \leftarrow 0$
$MyQueue.Insert(StartNode)$

while $MyQueue.IsNotEmpty$ **do**
  $FromNode \leftarrow MyQueue.Serve$
  for all WHITE $ToNode$ **adjacent to** $FromNode$ **and** selected in order of increasing edge weight **do**
    $DistanceToS[ToNode] \leftarrow DistanceToS[FromNode] + Cost(FromNode, ToNode)$
    $ColorLabel[ToNode] \leftarrow$ GRAY
    $MyQueue.Insert(ToNode)$
  end-for-all
  $ColorLabel[FromNode] \leftarrow$ BLACK
end-while
end-algorithm `Greedy-BFS-SPA`

In particular:

1. Find a graph with 5 or 6 nodes such that this algorithm correctly identifies all shortest paths to all the nodes.
2. Find a graph with 5 or 6 nodes such that this algorithm does not correctly identify all shortest paths to all the nodes.
3. Determine the worst case time complexity if the graph is implemented with adjacency lists.
4. Determine the worst case time complexity if the graph is implemented with an adjacency matrix.

8.6 Greedy or Dynamic Programming?

While most textbooks in CS claim that Dijkstra's Algorithm is a greedy one in nature, and this could certainly be argued for the version "*single-source-all-destinations*". For the version "*single-source-single-destinations*" it should be argued that in fact, it is a dynamic programming that is implemented iteratively. See cc3536 (paper and book by Sniedovich) for details.

9 Bellman - Ford shortest path algorithm

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.

Dijkstra's SP algorithm is designed for graphs where all edges have positive weights. If it is to be extended to include possibly negative weighted edges (that does not have negative cycles) then this must be altered. One clever way is to realize that a single `RelaxEdge(` $FromNode$`,` $ToNode$`)` has the potential to make a small improvement along that edge. Since we do not know, in advance, which edge to relax, we simply give every edge an opportunity to make improvements. The maximal length of a simple path is $n-1$, so we iterate $n-1$ times. This can even be used to detect a negative cycle: If there is still an improvement, with the n^{th} iteration, then there must be a cycle. Thus:

LINE 100: **Algorithm** `Basic-Bellmann-Ford-SP(`**graph** $myGraph$`,` **node** $StartNode$`,` **node** $StopNode$`)`

LINE 110: **for all** v **in** $myGraph$.NodeSet **do** $DistanceLabel[v] \leftarrow \infty$ **end-for all**
LINE 120: $DistanceToS[StartNode] \leftarrow 0$

LINE 130: **for** $Iteration$ **from** 1 **to** $n-1$ do
LINE 140: **for all** ($FromNode$, $ToNode$) **in** $myGraph$.EdgeSet **do** `BasicRelaxEdge(`$FromNode$`,` $ToNode$`)` **end**
LINE 150: **end** //$Iteration$

LINE 160: **if** (Another Round of Relaxations still decreases any Distance)
LINE 170: **then** REPORT A NEGATIVE CYCLE
LINE 180: **else** LENGTHS OF ALL SHORTEST PATHS ARE ESTABLISHED

LINE 190: **end algorithm** `Basic-Bellmann-Ford-SP`

where

LINE 100: **AlgorithmHelper** `BasicRelaxEdge(`**node** $FromNode$`,` **node** $ToNode$`)`
LINE 110: $DistanceToS[ToNode] \leftarrow \min\left\{DistanceToS[ToNode],\ DistanceToS[FromNode]+Cost(FromNode,ToNode)\right\}$
LINE 120: **end-algorithmHelper** `BasicRelaxEdge`

Om the next page, you will find a somewhat detailed implementation of the algorithm that has more bells and whistles. For instance, it prevents relaxation attempts that would not result in improved distances. A relaxation attempt between two WHITE nodes will not result in any improvement whatsoever. So at the cost of some administration, attempt only to relax with at least one GRAY node. The basic idea is: start with Dijkstra's SPA, but simply do not color any node to BLACK, which means that all GRAY nodes remain in the pool of candidates. In fact, we introduce many shades of GRAY to track edges that have the potential to make a successful relaxation, and track a "MyNodelist" for each iteration. However, the worst case would remain the same: there are $n-1$ attempts to relax along *all* m edges, so that $T^W(n,m) = \Theta(n \cdot m)$. Thus if the graph is sparse, and $m \approx 4\,n$, say, then $T^W(n,m) = \Theta(n^2)$, whereas if the graph is dense and $m \approx n^2$, say, then $T^W(n,m) = \Theta(n^3)$.

The improved implementation starts with Dijkstra'a Algorithm, to which we make a number of changes:

1. A relaxation attempt can only be successful if the $FromNode$ has already been discovered, and has a finite value for the Distance Label, so iterate only with edges whose $FromNode$ is WHITE .
2. After a node has been selected as a "$FromNode$", then it is still possible to improve at a later time, as new nodes (and potential negative edges) are being discovered. Thus the $FromNode$ is not colored BLACK and is not removed from consideration, but in stead, it remains GRAY, until no further improvements are made, and either the distances to all nodes have been determined (and all nodes are colored BLACK; all at the same time), ór there is a negative loop.
3. There is no need to maintain a queue, or to maintain a priority queue. A simple list of all GRAY nodes suffices, but we may need a need to have separate list for each iteration, or a circular list to make sure all GRAY nodes are considered in each iteration. The implementation as shown creates a new list for each iteration for reasons of clarity and transparency.
4. If no improvement is made in a particular iteration (indicated by a Boolean flag), then the algorithm can stop. Otherwise, test for the number of iterations.

LINE 100: **Algorithm** `Bellmann-Ford-SP(`**graph** $myGraph$, **node** $StartNode$, **node** $StopNode$, **counter** $NextIteration$`)`
LINE 110: **for all** v **in** $myGraph$.NodeSet **do** $DistanceLabel[v] \leftarrow \infty$ **end-for all**
LINE 120: $CompletedIters \leftarrow 0$
LINE 130: $DistanceToS[StartNode] \leftarrow 0$
LINE 140: $ToDoNodeList[1] \leftarrow \mathbf{List.create}()$
LINE 150: $ToDoNodeList[1].\mathbf{insert}(StartNode)$
LINE 160: $ContinueToRelax \leftarrow$ TRUE
LINE 170: **while** ($ContinueToRelax$ **and** $CompletedIters \leq n$) **do**
LINE 180: $ContinueToRelax \leftarrow$ FALSE
LINE 190: $CurrentIter \leftarrow CompletedIters + 1$
LINE 200: $NextIter \leftarrow CurrentIter + 1$
LINE 210: $ToDoNodeList[NextIter] \leftarrow \mathbf{List.create}()$
LINE 220: **while** $\big($**list.is.not.empty.**$ToDoNodeList[CurrentIter]\big)$ **do**
LINE 230: $FromNode \leftarrow ToDoNodeList[CurrentIter]$.**remove**
LINE 240: **for all** ($AdjNode$ **adjacent** to $FromNode$) **do**
LINE 250: `RelaxEdgeBF(`$FromNode$, $AdjNode$, $NextIter$`)`
LINE 260: **end-for all** //$AdjNode$
LINE 270: **end-while //list.is.not.empty**
LINE 280: $CompletedIters \leftarrow CurrentIter$
LINE 290: **end while**
LINE 300: **if** ($CompletedIters == n$ **and** $ContinueToRelax$)
LINE 310: **then** REPORT A NEGATIVE CYCLE
LINE 320: **else** LENGTHS OF ALL SHORTEST PATHS ARE ESTABLISHED
LINE 330: **end algorithm** `Bellmann-Ford-SP`

where

LINE 100: **AlgorithmHelper** `RelaxEdgeBF(`**node** $FromNode$, **node** $ToNode$`)`
LINE 110: **if** $DistanceToS[ToNode] > DistanceToS[FromNode] + Cost(FromNode, ToNode)$
LINE 120: **then**
LINE 130: $ContinueToRelax \leftarrow$ TRUE
LINE 140: $DistanceToS[ToNode] \leftarrow DistanceToS[FromNode] + Cost(FromNode, ToNode)$
LINE 150: $CameFromLabel[ToNode] \leftarrow FromNode$
LINE 160: $ToDoNodeList[NextIter].\mathbf{insert}(ToNode)$
LINE 170: **end if**
LINE 180: **end-algorithmHelper** `RelaxEdgeBF`

9.1 Small Example

Any example for this algorithm must be short and simple, just because the complexity is $T^W(n, m) = \Theta(n \cdot m)$, so even with $n = 5$ and $m = 7$, there are a lot of steps to present. If all list-selections are made alphanumerically, and if we present them after each iteration, it is not so bad, and I might add this next semester. Meanwhile, and fortunately, other instructors have placed their examples online. See for instance,
`https://www2.cs.arizona.edu/classes/cs545/fall09/ShortestPath2.prn.pdf` and
`https://algs4.cs.princeton.edu/lectures/44DemoBellmanFord.pdf`

9.2 Suggested Exercises

SE.48: Make an exercise here.

10 Floyd Warshall's shortest path algorithm

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.

Consider the algorithm named after Floyd and Warshall to find the shortest paths for all combinations of start- and end points. Assume that the nodes are numbered from 1 through n, and that there is a certain positive weight associated for each edge in the graph. Assume for now these weights are positive. Construct an adjacency matrix whose $(i,\ j)^{\text{th}}$ entry is either the weight of the edge between i and j, or whose entry is ∞ (or other special entry) if there is no edge between i and j.

The algorithm is designed around the question: The shortest path between any two nodes either *does* include node n, or *does not* include node n. This is a typical dynamic programming formulation. Advancing the argument, let $D^{(k)}$ be the table where the $(i,\ j)$-th element represents the minimal cost (shortest path) to travel from node i to node j, while only allowing nodes $1, 2, \cdots k$ on such a path. The initial values are given by $D^{(0)}$, which is a copy of the given weighted adjacency matrix. The driving equation is

$$D^{(k)}(i,\ j) = \min_{\text{2 options}} \left\{ D^{(k-1)}(i,\ j)\ ,\ D^{(k-1)}(i,\ k) + D^{(k-1)}(k,\ j) \right\} \tag{4}$$

As an aside, the sequence of tables/matrices $D^{(0)}$, $D^{(1)}$, $D^{(2)}$ $\ldots$ $D^{(n)}$ can all be stored *in place*, so that the space complexity is one single table with n^2 entries only. This is demonstrated in the execises.

Of course, you need to track the decisions you made. This can be done in two different ways, and is usually assigned as homework exercise in other books. However, one of the leading textbooks has (or had?) a typographical error, which persisted through several editions and which was copied in several lecture notes that are based on that book, and that are available on-line. Anyway, you can track the decision by one of two tables FS and LS, defined as:

$FS[i, j]$: If $FS[i, j] = p$, then the edge (i, p) is the first step on the minimal path from node i to node j, ór
$LS[i, j]$: If $LS[i, j] = q$, then the edge (q, j) is the last step on the minimal path from node i to node j.

The algorithm then is,

LINE 100: **Algorithm** `Floyd-Warchall-SP` (**graph** $myGraph$)
LINE 110: $D^0[i,\ j] \leftarrow C[i,\ j]$ for all $(i,\ j) \in E$ // Copy the Cost Matrix
LINE 120: $D^0[i,\ j] \leftarrow \infty$ for all $(i,\ j) \notin E$
LINE 130: $FS[i,\ j] \leftarrow j$ for all $i,\ j$ // Initialize the First Step Tracking Matrix
LINE 140: $LS[i,\ j] \leftarrow i$ for all $i,\ j$ // Initialize the Last Step Tracking Matrix
LINE 150: **for** k **from** 1 **to** n **do**
LINE 160: **for all** $(i,\ j)$ **from** $(1, 1)$ **to** (n, n) **do** RELAXFW$(D[i,\ j], k)$ **end-do**
LINE 170: **end-do**
LINE 180: **end algorithm** `Floyd-Warchall-SP`

where the helper algorithm RELAXFW$(D[i,\ j], k)$ is short for

LINE 10: **AlgorithmHelper** `RelaxFW`$(D[i,\ \ j], k)$
LINE 20: **if** $\left(D^{(k)}(i,\ j) > D^{(k-1)}(i,\ k) + D^{(k-1)}(k,\ j)\ \right)$
LINE 30: **then**
LINE 40: $D^{(k)}(i,\ j) \leftarrow D^{(k-1)}(i,\ k) + D^{(k-1)}(k,\ j)$ and éither
LINE 50: $FS(i,\ j) \leftarrow k$ ór
LINE 60: $LS(i,\ j) \leftarrow LS(k,\ j)$
LINE 70: **end-if**
LINE 80: **end-algorithmHelper** `RelaxFW`

10.1 Floyd-Marshall example

As an example, a particular instance of the problem resulted in the following matrices:

$$C = \begin{bmatrix} 0 & 5 & 8 & 9 & 1 \\ 1 & 0 & 9 & 9 & 5 \\ 6 & 5 & 0 & 1 & 8 \\ 9 & 1 & 6 & 0 & 8 \\ 5 & 9 & 1 & 5 & 0 \end{bmatrix} \quad D^{(5)} = \begin{bmatrix} 0 & 4 & 2 & 3 & 1 \\ 1 & 0 & 3 & 4 & 2 \\ 3 & 2 & 0 & 1 & 4 \\ 2 & 1 & 4 & 0 & 3 \\ 4 & 3 & 1 & 2 & 0 \end{bmatrix} \quad FS = \begin{bmatrix} 0 & 5 & 5 & 5 & 5 \\ 1 & 0 & 1 & 1 & 1 \\ 4 & 4 & 0 & 4 & 4 \\ 2 & 2 & 2 & 0 & 2 \\ 3 & 3 & 3 & 3 & 0 \end{bmatrix} \quad LS = \begin{bmatrix} 0 & 4 & 5 & 3 & 1 \\ 2 & 0 & 5 & 3 & 1 \\ 2 & 4 & 0 & 3 & 1 \\ 2 & 4 & 5 & 0 & 1 \\ 2 & 4 & 5 & 3 & 0 \end{bmatrix}$$

You may want to draw the original graph and reconstruct some of the paths.

10.2 Suggested Exercises

SE.49: Suppose you use this algorithm on an 9-node graph, and let $D^{(0)}$ and $D^{(4)}$ be given as follows.

$$D^{(0)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 6 & \infty & \infty & \infty & \infty & \infty & \infty & 7 \\ 9 & 0 & 5 & \infty & \infty & \infty & \infty & \infty & \infty \\ \infty & 6 & 0 & 9 & \infty & \infty & \infty & \infty & \infty \\ \infty & \infty & 11 & 0 & 7 & \infty & \infty & \infty & \infty \\ \infty & \infty & \infty & 3 & 0 & 8 & \infty & \infty & \infty \\ \infty & \infty & \infty & \infty & 6 & 0 & 7 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 7 & 0 & 7 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 7 & 0 & 10 \\ 8 & \infty & \infty & \infty & \infty & \infty & \infty & 7 & 0 \end{array}\right] \end{array}$$

and

$$D^{(4)} = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 6 & 11 & 20 & 27 & \infty & \infty & \infty & 7 \\ 9 & 0 & 5 & 14 & 21 & \infty & \infty & \infty & 16 \\ 15 & 6 & 0 & 9 & 16 & \infty & \infty & \infty & 22 \\ 26 & 17 & 11 & 0 & 7 & \infty & \infty & \infty & 33 \\ 29 & 20 & 14 & 3 & 0 & 8 & \infty & \infty & 36 \\ \infty & \infty & \infty & \infty & 6 & 0 & 7 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & 7 & 0 & 7 & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty & 7 & 0 & 10 \\ 8 & 14 & 19 & 28 & 35 & \infty & \infty & 7 & 0 \end{array}\right] \end{array}$$

Determine the values for $D^{(5)}(4,6)$, $D^{(5)}(6,4)$, $D^{(5)}(6,9)$, and $D^{(5)}(9,6)$.

10.3 Construct the shortest paths

The Floyd-Warshall algorithm finds the lengths of all the shortest paths between all-source-all-destinations. It also tracks the decisions with either a FS-table or a LS-table. (We showed both, when in fact, only one is needed.) Tracking information is maintained and the algorithm here prints a path.

A proposed algorithm is given below, but it may (or may not) give the end-point(s). You are expected to address this in the exercises.

LINE 100: **Algorithm** `PrintPath(`**TrackingMatrix** FS`, `n`, `s`, `t`)`
LINE 110: // Precondition: The Tracking Matrix has n by n elements, the elements are positive integers $1 \ldots n$
LINE 120: // We assume the standard matrix notation, where the row and column index go from 1 through n
LINE 130: // We also assume that s and t are row and column indices: $1 \leq s, t \leq n$.
LINE 140: **if** $FS[s,t] \neq t$ **then**
LINE 150: **print**$(FS[s,t])$
LINE 160: `PrintPath(` FS`, `n`, `$FS[s,t]$`, `t`)`
LINE 170: **end if**
LINE 180: **end algorithm** `PrintPath`

10.4 Suggested Exercises

SE.50: A FS-tracking table is shown below, and the `PrintPath` algorithm is invoked with `PrintPath`(FS, 12, 3, 7). What is actually printed?

$$FS = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \end{array} \begin{array}{c} \begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \\ \left[\begin{array}{cccccccccccc} 0 & 5 & 4 & 4 & 5 & 4 & 4 & 4 & 9 & 10 & 4 & 4 \\ 9 & 0 & 9 & 9 & 9 & 6 & 9 & 8 & 9 & 9 & 8 & 12 \\ 1 & 2 & 0 & 1 & 1 & 1 & 1 & 1 & 9 & 9 & 1 & 1 \\ 3 & 8 & 3 & 0 & 8 & 8 & 8 & 8 & 8 & 10 & 8 & 8 \\ 2 & 2 & 2 & 2 & 0 & 2 & 7 & 8 & 2 & 2 & 11 & 12 \\ 3 & 12 & 3 & 4 & 12 & 0 & 12 & 8 & 12 & 10 & 11 & 12 \\ 5 & 5 & 5 & 4 & 5 & 11 & 0 & 4 & 5 & 5 & 11 & 5 \\ 1 & 2 & 10 & 4 & 11 & 11 & 11 & 0 & 2 & 10 & 11 & 11 \\ 10 & 2 & 10 & 10 & 5 & 6 & 7 & 8 & 0 & 10 & 8 & 6 \\ 3 & 2 & 3 & 3 & 5 & 6 & 7 & 3 & 2 & 0 & 3 & 6 \\ 1 & 6 & 3 & 4 & 6 & 6 & 6 & 6 & 6 & 6 & 0 & 6 \\ 7 & 7 & 7 & 7 & 7 & 6 & 7 & 8 & 7 & 7 & 7 & 0 \end{array}\right] \end{array}$$

SE.51: Repeat for PrintPath (FS, 12, 12, 5, 2).

SE.52: How would you change, or add, the algorithm so that it also prints the starting node and the terminal node, without listing intermediate nodes twice?

SE.53: When the matrix $D^{(k)}$ is computed from $D^{(k-1)}$ for all $(i,\ j)$, how are the values $D^{(k)}[i,\ k]$ (for all i), and the values $D^{(k)}[k,\ j]$ (for all j) effected? In other words, Is there a value for i, and is there a value for j such that $D^{(k)}[i,\ k] \neq D^{(k-1)}[i,\ k]$ or $D^{(k)}[k,\ j] \neq D^{(k-1)}[k,\ j]$?

SE.54: Take a two grids of, say 8×8 and place them side-by-side. Pretend that the left grid is the matrix $D^{(k-1)}$ and the right matrix is the matrix $D^{(k)}$. Now for all values of $(i,\ j)$ for which $D^{(k)}[i,\ j]$ could potentially be different, color all the entries in the right matrix gray. Furthermore, in the left matrix, color all elements gray that have been consulted. Notice that the pattern in the left and right matrix complements each other, and conclude that $D^{(k-1)}$ and $D^{(k)}$ could occupy the same space.

SE.55: (Graduate Students) Now formally proof the observation that all $D^{(k)}$ -matrices could occupy the same space.

11 Spanning Tree Algorithms

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.

A spanning tree is a subset of the graph with n nodes and with $n-1$ edges, such that the subset forms a (free) tree (and thus all nodes are in a single connected component). The weight of a spanning tree is the sum of all the edge-weights. For a given graph with more than n edges, there are several spanning trees. A minimum spanning tree is a spanning tree, such that it's combined edge weight is minimal. There are several algorithms known to construct a MST, the most common ones are named after Kruskal and Prim.

11.1 Kruskal's Algorithm for MST

The basic idea behind Kruskal's algorithm is to continually find (and remove from consideration) the edge with smallest cost. If the end-points of the edge are in different connected components, then select this edge for inclusion into the spanning tree (i.e. it becomes a tree-edge). On the other hand, if the end-points of the edge are already in the same connected component, then this edge does not become a tree-edge. Either way, this process continues until $n-1$ tree-edges have been selected as a tree-edge.

To implement this idea efficiently, first create a priority queue with edges and using edge weights as priority criterion (a min-heap, for instance, at the cost of $\Theta(m)$). Once this is constructed, you repeatedly remove an edge to determine whether the end-points are in the same or in different connected components. This is best accomplished using the Union-Find data structure. Thus:

LINE 100: **Algorithm** `Basic-Kruskal-MST` (**graph** $myGraph$)
LINE 110: **Create** a UNION-FIND data structure with n singletons
LINE 120: **Create** a MINHEAP data structure with m edges and their values. Use **Heapify** to do this in $\Theta(m)$ time.
LINE 130: **repeat**
LINE 140: $(FromNode, ToNode) \leftarrow$ MIN-HEAP.`Remove`
LINE 150: **if** UNION-FIND.FIND($FromNode$) $\neq$ UNION-FIND.FIND($ToNode$)
LINE 160: **then** //Then the end-nodes are in different connected components
LINE 170: UNION-FIND.JOIN($FromNode, ToNode$) // i.e. connect them
LINE 180: **Label** and **Count** this edge as tree-edge
LINE 190: **end-if**
LINE 200: **until** ($n-1$ edges have been accepted)
LINE 210: **end algorithm** `Basic-Kruskal-MST`

The time complexity is completely dominated by the heap manipulation and the number of times the heap-removal is needed. In the worst case, the *heapify* cost is linear in m, after which any *heap.removal* takes $\lg j$ where j is the size at that time, for a total of $T^W_{\text{Kruskal}}(n,m) = m + \lg m + \lg(m-1) \ldots \lg 2 = \Theta(m \lg m) = \Theta(m \lg n)$, since m is between n and n^2 and thus $\lg m$ between $\lg n$ and $2 \lg n$.

11.2 Prim's Algorithm for MST

The basic idea behind Prim's algorithm is to take any random node (the start node) which forms the dedicated growing component. It then incrementally grows this dedicated component by one new edge at a time, by selecting the minimum weight edge from among all candidate edges that connect a node inside the dedicated component to a node outside the component. If the color of the nodes in the dedicated component are BLACK, then a candidate edge connects a BLACK with a NON-BLACK node. This process continues until n nodes are connected in the single component, and $n-1$ tree-edges have been selected as a tree-edge.

An efficient algorithm calls for a priority queue structure (e.g. MinHeap) for the candidate edges, where the lazy version should be used to avoid excessive manipulations. Although Prim's algorithm should be clear by now, there are two slightly different viewpoints to implement it, and these implementation choices are based on the fact that edges are not always represented explicitly, but rather by using labels attached to nodes, much like this is done in Dijkstra's SPA. Whereas in Dijkstra's algorithm the distance labels indicate the distance to S, in Prim's STA, the distance labels indicate the distance to the closest BLACK node. We present both variations of the algorithm. We will not make use of the color GRAY, which could be used to indicate that fact that the node has been discovered (and placed in the MinHeap). So, we have nodes that initial all WHITE, and will only be colored BLACK at the time that they are included in the dedicated component.

11.2.1 Prim's Algorithm, with a Heap of Edges.

The algorithm becomes

LINE 100: **Algorithm** `Basic-Prim-MST(`**graph** $myGraph$`,` **node** $StartNode$`)`

LINE 110: **for all** v **in** $myGraph$.NodeSet **do** $ColorLabel[v] \leftarrow$ WHITE **end-for all**

LINE 120: **Create** a MIN-HEAP data structure for edges, initially empty, using edge weights as heap-values.

LINE 130: $Color[StartNode] \leftarrow$ BLACK
LINE 140: **for all** ($AdjNode$ **adjacent** to $StartNode$) **do**
LINE 150: MIN-HEAP.`insert`($\mathbf{edge}\langle StartNode, AdjNode\rangle$) // The heap is a heap of edges, with edge weights as heap-criterion
LINE 160: **end-for all**

LINE 170: **while** (fewer than $n-1$ edges have been accepted)

LINE 180: // Find the next edge with minimal cost and with $Color[ToNode]$ still WHITE
LINE 190: **Repeat** $edge\langle FromNode, ToNode\rangle \leftarrow$ MIN-HEAP.`Remove` **until** $Color[ToNode] ==$ WHITE **end-repeat**

LINE 200: $Color[ToNode] \leftarrow$ BLACK
LINE 210: **Label** and **Count** the edge $\mathbf{edge}\langle FromNode,\ ToNode\rangle$ as tree-edge

LINE 220: **for all** ($NewAdjNode$ **adjacent** to $ToNode$ **with** $ColorLabel[NewAdjNode] ==$ WHITE) **do**
LINE 230: $MyPriorityQueue.\mathbf{insert}(\langle ToNode, NewAdjNode\rangle)$
LINE 240: **end-for all**

LINE 250: **end while**

LINE 260: **end algorithm** `Basic-Prim-MST`

11.2.2 Prim's Algorithm, with a Heap of Nodes.

Prim's Algorithm is very similar to Dijkstra's Shortest Path Algorithm: In Dijkstra's SPA, the shortest Path Network is started by including the start node, and then grown one node at a time, where the next node is selected according to the next shortest path node. In Prim's algorithm, a minimal Spanning Network is is started by including the start node, and then grown one node (with edge) at a time, where the next node is selected according the next minimal edge-weight, the smallest distance to the ever growing spanning tree. Here is indeed the major difference between Dijkstra's SPA and Prim's MST: In the SPA algorithm, the distance label tracks the distance to the start-node, whereas in the MST algorithm, the distance label tracks the distance to the ever growing spanning-tree-to-be. Since all the nodes in this spanning-tree-to-be are BLACK, we call the criterion $DistanceToBlack[ToNode]$ as opposed to $DistanceToS[ToNode]$. We still use a Lazy-Heap, and we do away with the color GRAY, which is really used for educational purposes only.

A minor difference is the stopping criterion: Dijkstra's SPA stops whenever the distance between S and the stopping node has been finalized (and the stopping node colored BLACK), whereas the MST algorithm stops whenever *all* nodes have been finalized.

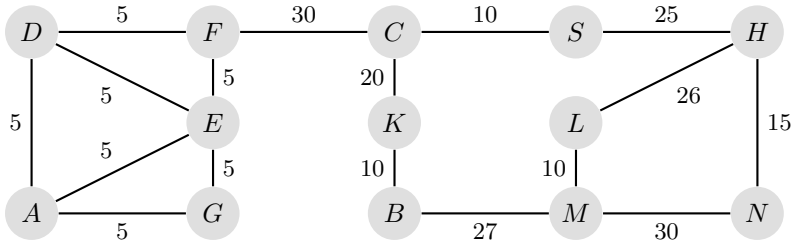
LINE 100: **Algorithm** `PrimST(`**graph** $myGraph$**,** **node** $StartNode$`)`
LINE 110: **for all** v **in** $myGraph$.NodeSet **do**
LINE 120: $ColorLabel[v] \leftarrow$ WHITE
LINE 130: $DistanceToBlack[v] \leftarrow \infty$
LINE 140: **end-for all**
LINE 150: $DistanceToBlack[StartNode] \leftarrow 0$
LINE 160: $CameFrom[StartNode] \leftarrow NILL$
LINE 170: $MyPriorityQueue \leftarrow \mathbf{PriorityQueue.create}()$
LINE 180: $MyPriorityQueue.\mathbf{insert}(StartNode)$
LINE 190: **while** (less than $n-1$ edges have been accepted)
LINE 200: // Find a node with minimal $DistanceToBlack$ that is WHITE,
LINE 210: // (if a BLACK node is served from the $MyPriorityQueue$, then the edge weight is obsolete
LINE 220: // and this node was reached earlier with a cheaper edge)
LINE 230: **Repeat** $SelectNode \leftarrow MyPriorityQueue.\mathbf{remove}()$ **until** $ColorLabel[SelectNode] ==$ WHITE **end-repeat**
LINE 240: $Color[SelectNode] \leftarrow$ BLACK
LINE 250: **Label** and **Count** the **edge** $\langle CameFrom[SelectNode],\ SelectNode\rangle$ as tree-edge
LINE 260: **for all** ($AdjNode$ **adjacent** to $SelectNode$ **with** $ColorLabel[AdjNode] ==$ WHITE) **do**
LINE 270: `RelaxEdgePrim(`$SelectNode$`,` $AdjNode$`)`
LINE 280: **end-for all**
LINE 290: **end while**
LINE 300: **end algorithm** `PrimST`

where
LINE 100: **AlgorithmHelper** `RelaxEdgePrim(`**node** $FromNode$**,** **node** $ToNode$`)`
LINE 110: **if** $Cost(FromNode, ToNode) < DistanceToBlack[ToNode]$
LINE 120: **then**
LINE 130: $DistanceToBlack[ToNode] \leftarrow Cost(FromNode, ToNode)$
LINE 140: $CameFromLabel[ToNode] \leftarrow FromNode$
LINE 150: $MyPriorityQueue.\mathbf{insert}(ToNode)$
LINE 160: **end if**
LINE 170: **end-algorithmHelper** `RelaxEdgePrim`

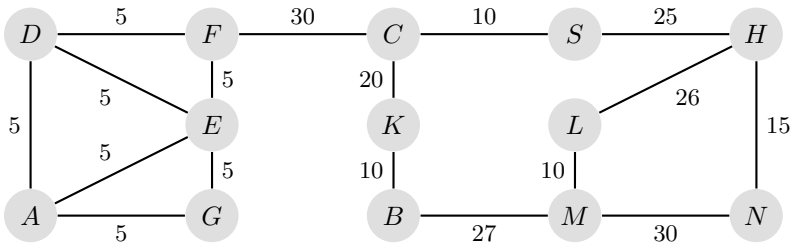
The time complexity is again dominated by the heap manipulation and the number of times the heap-removal is needed. The analysis is similar to that of Dijkstra's SPA, and results in $T^{W}_{\text{Prim}}(n,m) = \Theta(m \lg n)$ as well.

11.3 Suggested Exercises on Spanning Trees

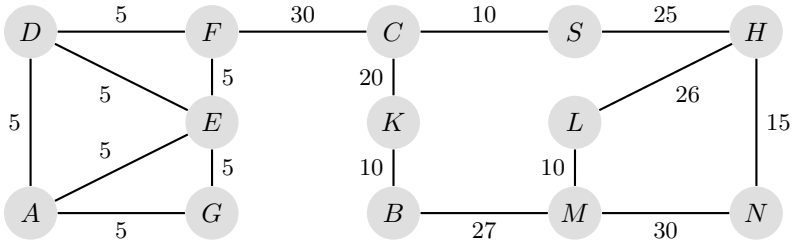
SE.56: Illustrate the execution of Kruskal's algorithm by labeling (i.e. numbering, ranking) the edges in the order they are removed from the priority queue (min-heap) and are included into the structure that becomes the Kruskal-Minimal Spanning Tree. Furthermore, show the content of the Min-Heap, as well as the content of the Union-Find array at the time that the while-loop terminates.



SE.57: Illustrate the execution of the first version of Prim's algorithm by labeling (i.e. numbering, ranking) the edges in the order they are removed from the priority queue (min-heap) and are included into the structure that becomes the Prim-Minimal Spanning Tree. Furthermore, show the content of the Min-Heap at the time that the while-loop terminates.



SE.58: Illustrate the execution of the second version of Prim's algorithm by labeling (i.e. numbering, ranking) the edges in the order they are removed from the priority queue (min-heap) and are included into the structure that becomes the Prim-Minimal Spanning Tree. Furthermore, show the content of the Min-Heap at the time that the while-loop terminates.



SE.59: (Graduate students) Compare and contrast the two versions of Prim's Algorithm? Give an example of a graph such that both algorithms have "about the same" time performance.

SE.60: (Graduate students) Compare and contrast the two versions of Prim's Algorithm? Give an example of a graph such that both algorithms have "definitely different " time performance.

SE.61: (Graduate students) When a graph G has an edge with a negative weight, then Dijkstra's Shortest Path Algorithm does not always return the correct values for the shortest paths. What about Prim's and Kruskal's MST algorithms? Do they still return he correct MST trees? Please explain carefully.

SE.62: The time complexities for Dijkstra, Kruskal, and Prim's algorithms were derived under the assumptions that

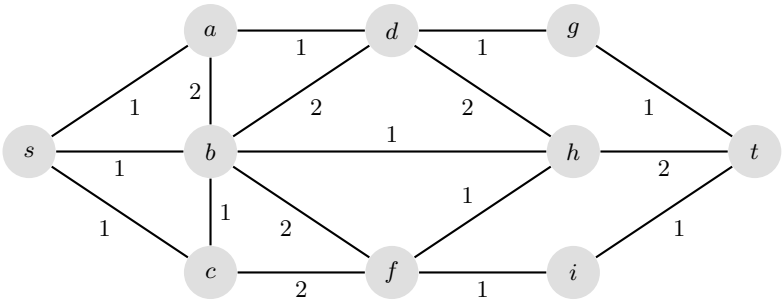
1. Adjacency lists were used to implement the graphs
2. MinHeaps were used to serve new edges/nodes.

For each of these algorithms,
1. undergraduate students are expected to be able to derive (or remember) the time complexity under the following assumptions, and
2. graduate students are expected to be able to derive the time complexities and should be prepared to derive time complexities under slightly altered conditions as well.

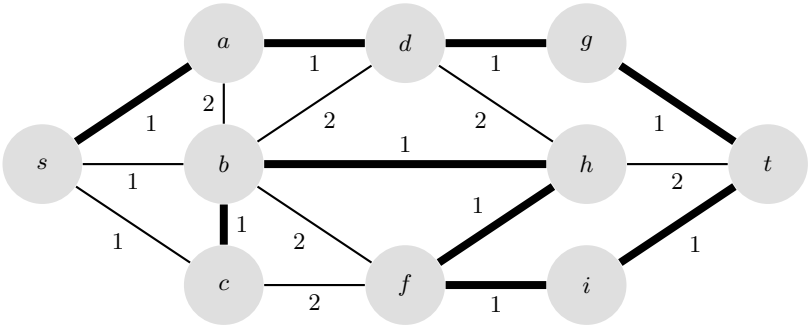
1. Adjacency lists were used to implement the graphs, but no heap: a straight scan is used each time a "best" node/edge needs to be located.
2. Adjacency Matrices were used to implement the graphs, and still a heap.
3. Adjacency Matrices were used to implement the graphs, but no heap: a straight scan is used each time a "best" node/edge needs to be located.

11.4 Shortest Paths and Minimal Spanning Trees: Rejoinder

Shortest Path algorithms and Minimal Spanning Tree algorithms are very similar, but have clearly different objectives. A path in a MST could be very long, because path length is not considered in its construction. Consider the following graph:



Notice that a MST has total length 9, and could be $s-a-d-g-t-i-f-h-b-c$ (others are possible). If you were to construct this MST however, then the path length between s and c would be 9. On the other hand, if you build the graph as given, then the largest distance between any two nodes is 4, and this maximal distance is attained for 6 node pairs. So the question now becomes: Construct a Spanning Tree for this particular graph, whose total cost is minimal, while the distance between any two nodes is maximally 4. Draw for instance the spanning trees for an MST and one induced by a SPA. Both of these spanning trees could be different. Are they always different?



SE.63: (Graduate students) Shortest Path algorithms and Minimal Spanning Tree algorithms are very similar, but have clearly different objectives. A path in a MST could be very long, because path length is not considered in its construction. Either create a tree with 6 or 7 nodes and assign weights such that both a MST ánd a SPT must be identical. Ór provide convincing arguments that this is not possible.

SE.64: (Graduate students) Shortest Path algorithms and Minimal Spanning Tree algorithms are very similar, but have clearly different objectives. A path in a MST could be very long, because path length is not considered in its construction. Either create a tree with 6 or 7 nodes and assign weights such that both a MST ánd a SPT must be different. Ór provide convincing arguments that this is not possible.

12 Graphs and their Algorithms: A second helping

We have now seen some graphs and some of the graph algorithms to create (shortest) paths and various spanning trees (minimal spanning trees and spanning trees induced by DFS and BFS, e.g.). It is time to pause for a reflection. We have not seen a data structure that can be called the inherent best data structure to represent a graph. The adjacency list structure does usually not represent edges as a separate entity, although it could easily point to a structural node with a pointer. The additional pointer is simply omitted for performance reasons and transparency. Most of the algorithms we have seen so far did not even create a new and separate graph structure for sub-graphs, such as spanning trees, but used labels attached to nodes of the original to represent the subgraphs. Shortest paths have received similar treatment: a path is not represented as a separate entity but also as a collection of labels attached to nodes. Thus, labels intended for subgraphs and path are not implemented separately but rather embedded in the original graph and are attached to nodes when implementing the algorithms. When you then look at a program that implements a graph algorithm, you find labels attached to nodes, which serve a multitude of functions:

1. provide information for the node itself (such as the actual name, geographical location, etc.) in the original graph,
2. provide information for the role of the node in a sub-graph (like the color-labels we used),
3. provide information on edges to one, several, or all adjacent nodes, like "spanning tree edges". In such a case, it would likely be a list (or stack, or queue) of labels.
4. provide information on edges to some adjacent nodes, as part of a path, like "if I am on a shortest path to *Start*, then my adjacent node *CameFrom* is also on a shortest path to *Start*, and is one hop closer to *Start*".

Thus, labeling is great method to provide additional information about graphs, about sub-graphs, and about paths. Yet, it could be bewildering if their algorithmic context is lost. If you only have a program to look at, or if an algorithm is presented more like a program (which unfortunately happens quite a bit), and the documentation is not the best (it usually is not the best), then it may be hard to reverse engineer what the program/algorithm is intended to do. But placing labels with nodes is the best way to provide labels on sub-graphs and/or paths, how else? For ease of discussion below, we use $\mathbf{E}\langle a, b\rangle$ to refer to an edge between a and b, and we use $\mathbf{P}\langle S, T\rangle$ to refer to a path between S and T if there no ambiguity, and we use $\mathbf{P}\langle S, v_1, v_2, \ldots v_k, T\rangle$ if the entire path needs to be specified.

Consider e.g. how to answer some questions. Note that the questions mostly sound easy, and the answers are as well, but only if you are used to labeling.

1. For a particular spanning tree $T = (V, E')$ in a graph $G = (V, E)$, is the edge $\langle\ a, b\ \rangle$ part of the T?
2. For a particular spanning tree $T = (V, E')$ in a graph $G = (V, E)$, and a particular node $a \in V$, how many tree-edges are adjacent to this node a?
3. For a particular spanning tree $T = (V, E')$ in a graph $G = (V, E)$, and a particular node $a \in V$, how many MST-edges are adjacent to this node a?
4. For a particular graph $G = (V, E)$ with MST $T = (V, E')$ and a particular tree-edge $\langle\ a, b\ \rangle \in E'$: Is this particular edge $\langle\ a, b\ \rangle$ always part of every MST for this graph?
5. For a particular graph $G = (V, E)$ with MST $T = (V, E')$ and a particular tree-edge $\langle\ a, b\ \rangle \in E'$: Is this particular edge $\langle\ a, b\ \rangle$ unique to the MST? (that is, if there another MST, then this edge is not necessarily part of this other MST)?

Additional questions will be explored below. Pay particularly attention to the shortest path problems where nodes add costs to the path: sometimes the costs are for absolute nodes, sometimes the costs are for relative nodes. In both case, labels are used: labels for nodes, and labels for paths.

Sometimes labeling appears to be correct approach, whereas it may be best to reduce the new problem and map the new problem to an older problem that already has a solution, which can be adapted to the new situation. Most text books (and we did as well) refer to this: "Applications of DFS", "Applications of BFS", and so on. Consider however the following (shortest) path type problems. Most people would try to find solutions by adjusting/expanding existing programs, rather then adjusting/expanding existing algorithms.

Let $\mathbf{P}_a = S_a \to a_1 \to a_2 \to \ldots a_k \to T_a$ and $\mathbf{P}_b = S_b \to b_1 \to b_2 \to \ldots b_\ell \to T_b$ be two shortest paths, and assume that $S_a \neq S_b$ and $T_a \neq T_b$.

6. Do the paths $\mathbf{P}_a$ and $\mathbf{P}_b$ have a node in common?
7. Do the paths $\mathbf{P}_a$ and $\mathbf{P}_b$ have an edge in common? If, so, what is the longest common path?
8. If the paths $\mathbf{P}_a$ and $\mathbf{P}_b$ do not have a node in common, then what is the shortest distance between the two paths?

12.1 Minimal cost to share a ride

For Graduate Students Only

SE.65: A person Z invites two friends A and B to come over for dinner at Z's house and Z is willing to pay the travel expenses as long as this is as cheap as possible. Thus A and B have to determine the cheapest total cost. Fortunately, all individual link costs are strictly positive. They used Floyd-Warshall to determine the shortest distances between all node pairs. Let $A = 1$, $B = 2$, and $Z = n = 9$.

$$D^{(9)} = \begin{array}{c} \\ A{=}1 \\ B{=}2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ Z{=}9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} A{=}1 & B{=}2 & 3 & 4 & 5 & 6 & 7 & 8 & Z{=}9 \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 9 & 9 & 10 & 9 & 11 & 17 & 18 & 19 \\ 11 & 0 & 8 & 15 & 9 & 19 & 17 & 11 & 18 \\ 9 & 6 & 0 & 10 & 12 & 19 & 12 & 17 & 20 \\ 18 & 11 & 12 & 0 & 8 & 11 & 11 & 11 & 18 \\ 10 & 10 & 7 & 6 & 0 & 10 & 11 & 9 & 10 \\ 11 & 11 & 9 & 10 & 12 & 0 & 6 & 11 & 8 \\ 18 & 18 & 12 & 12 & 9 & 7 & 0 & 6 & 8 \\ 18 & 18 & 12 & 12 & 8 & 13 & 6 & 0 & 7 \\ 19 & 19 & 17 & 18 & 17 & 8 & 12 & 9 & 0 \end{array}\right] \end{array}$$

Part a) For this particular situation: What is the shortest combined path length?

Part b) This particular example was for $n = 9$ nodes in the graph. If n has a value that is not specified, but large, what is the time complexity $T(n)$ to determine the minimal cost, now as an explicit function of n? (Not counting the cost for Floyd-Warshall, count only the additional cost).

Part c) This particular solution used the result of Floyd-Warshall, at a cost of $\Theta(n^3)$, after which a cheaper algorithm in $\Theta(n)$ was used to determine the answer. So perhaps using the full Floyd-Warshall is somewhat overkill since many shortest path pairs are not consulted. Can the whole problem be solved cheaper? Justify and explain your answer. What would a lower bound argument be?

12.2 Best place to meet up

SE.66: Two friends A and B plan to have dinner at a nice restaurant R. They want to find a restaurant that is closest by. The friends are locations 1 and 2, and the restaurants they would consider is at location 3, 4, . . . 9. They each want to drive separately and independently from one another. All individual link distances are strictly positive. They used Floyd-Warshall to determine the shortest distances between all node pairs.

$$D^{(9)} = \begin{array}{c} \\ A{=}1 \\ B{=}2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \end{array} \begin{array}{c} \begin{array}{ccccccccc} A{=}1 & B{=}2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \end{array} \\ \left[\begin{array}{ccccccccc} 0 & 9 & 9 & 10 & 9 & 11 & 17 & 18 & 19 \\ 11 & 0 & 8 & 15 & 9 & 19 & 17 & 11 & 18 \\ 9 & 6 & 0 & 10 & 12 & 19 & 12 & 17 & 20 \\ 18 & 11 & 12 & 0 & 8 & 11 & 11 & 11 & 18 \\ 10 & 10 & 7 & 6 & 0 & 10 & 11 & 9 & 10 \\ 11 & 11 & 9 & 10 & 12 & 0 & 6 & 11 & 8 \\ 18 & 18 & 12 & 12 & 9 & 7 & 0 & 6 & 8 \\ 18 & 18 & 12 & 12 & 8 & 13 & 6 & 0 & 7 \\ 19 & 19 & 17 & 18 & 17 & 8 & 12 & 9 & 0 \end{array}\right] \end{array}$$

Part a): The problem is that "closest by" has two different meanings:

1. A interprets distance in miles. So A recommends a restaurant at location R, such that their combined distance in miles to R is shortest:

$$\boxed{\min_{R,\text{ with } 3\le R\le n} \left\{ D^{(9)}[1,R] + D^{(9)}[2,R] \right\}} \tag{5}$$

   Which location(s) does A recommend?

2. B interprets distance in minutes (i.e. time), and knows that both travel simultaneously, such that the "last-one" to arrive is the distance in minutes. So B attempts to minimize the max-time:

$$\boxed{\min_{R,\text{ with } 3\le R\le n} \left\{ \max_{\text{2 options}} \left\{ D^{(9)}[1,R],\ D^{(9)}[2,R] \right\} \right\}} \tag{6}$$

   Which location(s) does B recommend?

Part b) This particular example was for $n = 9$ nodes in the graph. If n has a value that is not specified, but large, what is the time complexity $T(n)$ to determine the minimal cost according to B, but now as an explicit function of n? (Not counting the cost for Floyd-Warshall, count only the additional cost). Explain your reasoning.

Part c) This particular solution used the result of Floyd-Warshall, at a cost of $\Theta(n^3)$, after which a cheaper algorithm in $\Theta(n)$ was used to determine the answer. So perhaps using the full Floyd-Warshall is somewhat overkill. Can the whole problem be solved cheaper? Justify and explain your answer. What would a lower bound argument be?

12.3 Shortest Path with exact Hop-Count

The algorithm named after Floyd and Warshall finds the shortest paths for all combinations of start- and end points and with an unlimited number of hops (or links, or edges). FW was solved by numbering the nodes and asking a typical dynamic programming question: *Does* the shortest path between any two nodes include node n, or is node n *not included*?

Well, if you now want the shortest path with *exactly* k steps, then the situation is pretty similar, and comes in two flavors: Let $F^{(k)}$ be the table where the $(i,\ j)$-th element represents the minimal cost (shortest path) to travel from node i to node j, while only using *exactly* k hops on such a path. The initial values are given by $F^{(0)}$, which is a copy of the given weighted adjacency matrix, and remember that the diagonal are zero's since we assume a simple graph as input. The driving equation is either (this flavor connects a starting node with path of length $k-1$)

$$F^{(k)}(i,\ j) = \min_{\ell,\text{ with } 1\le\ell\le n} \Big\{\ F^{(0)}(i,\ \ell) + F^{(k-1)}(\ell,\ j)\Big\} \tag{7}$$

or (this flavor connects a stopping node with path of length $k-1$)

$$F^{(k)}(i,\ j) = \min_{\ell,\text{ with } 1\le\ell\le n} \Big\{\ F^{(k-1)}(i,\ \ell) + F^{(0)}(\ell,\ j)\Big\}. \tag{8}$$

This gives the shortest path with a hop-count exactly k, and might include loops. If you do not want/need loops, then limit the values of ℓ to $\ell \neq i$, and $\ell \neq j$ (or could you just put zero's in the the diagonal after each iteration?)

12.4 Shortest Path with limited Hop-Count

If you now want the shortest path with *at most* k steps, then let $G^{(k)}$ be the table where the $(i,\ j)$-th element represents the minimal cost (shortest path) to travel from node i to node j, while only using *at most* k hops on such a path. The initial values are again given by $G^{(0)}$, which is a copy of the given weighted adjacency matrix, and remember that the diagonal are zero's since we assume a simple graph as input. The driving equation is either (this flavor connects a starting node with path of length $k-1$)

$$G^{(k)}(i,\ j) = \min_{\text{2 options}} \Big\{G^{(k-1)}(\imath,\ j),\quad \min_{\ell,\text{ with } 1\le\ell\le n} \Big\{\ G^{(0)}(i,\ \ell) + G^{(k-1)}(\ell,\ j)\Big\}\Big\} \tag{9}$$

or (this flavor connects a stopping node with path of length $k-1$)

$$G^{(k)}(i,\ j) = \min_{\text{2 options}} \Big\{G^{(k-1)}(\imath,\ j),\quad \min_{\ell,\text{ with } 1\le\ell\le n} \Big\{\ G^{(k-1)}(i,\ \ell) + G^{(0)}(\ell,\ j)\Big\}\Big\} \tag{10}$$

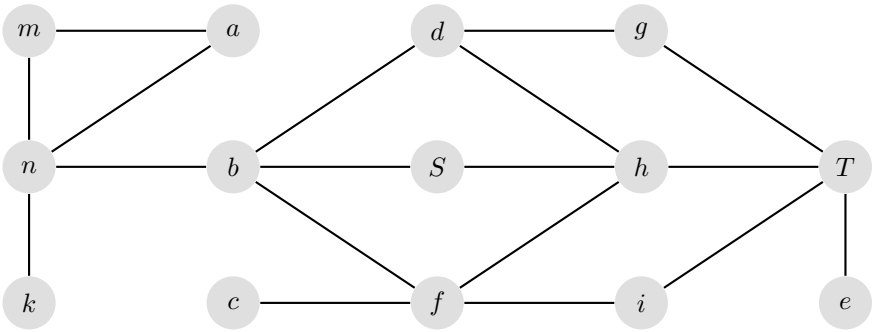
This gives the shortest path with a hop-count at most k, and does not include loops because the weights are positive.

12.5 Shortest Simple Path with a Detour

SE.67: *Finding a simple path that visits a third node* Let an undirected simple graph $G = (V, E)$ be given with n nodes and m edges, including three nodes S, T, and U. Design an efficient algorithms for the following problem: Is there a simple path (i.e. no cycles) from S to T, such that U is on that path. For the algorithm you present, you have to argue it's correctness, derive the asymptotic complexity in terms of n and m (number of nodes and edges), and argue whether or not you believe that more efficient algorithms exist.

Please note: For full credit, your algorithm must be correct, your derivation must be complete and correct and must be $O(n^4)$ if the graph is nearly complete, i.e. $m \approx n^2$, but it does not need not be the most efficient algorithm possible. Please be careful, if there is a simple path from S to U, and a simple path from U to T, then there still may not be a simple path as desired. Look at the example graph below: there is a simple path from S to a, and a simple path from a to T, but there is no simple path that visits a. Your algorithms should be for an undirected simple graph with n nodes and m edges. For the specific example graph below, the answers are

1. Yes, for nodes $U = b$, d, f, g, h, or i.
   No, for nodes $U = a$, c, e, k, m and n.



12.6 Nodes on a Simple Path

SE.68: Let an undirected simple graph be given with n nodes and m edges, including nodes S and T. Consider writing efficient algorithms for the following three problems:

1. Given an additional node x: Is there a simple path from S to T, such that x is on that path. (Please be careful, if there is a simple path from S to x, and a simple path from x to T, then there still may not be a simple path as desired. Look at the example graph below: there is a simple path from S to a, and a simple path from a to T, but there is no simple path that visits a.)
2. Find any one additional node y that is on any simple path from S to T.
3. Find all nodes z that are on a simple path from S to T.

For each of these problems, design an algorithm that solves it. Your algorithm must be correct, and must be $O(n^4)$, but need not be the most efficient possible. You have to argue it's correctness, determine the asymptotic complexity in terms of n and m (number of nodes and edges), and argue whether or not you believe that more efficient algorithms exist.

Your algorithms should be for an undirected simple graph with n nodes and m edges. For the specific example graph below, the answers are

1. Yes, for nodes $x = b$, d, f, g, h, or i.
   No, for nodes $x = a$, c, e, k, m and n.
2. Any of the nodes b, d, f, g, h, and i.
3. All of the nodes b, d, f, g, h, and i.

12.7 Shortest Path with Stop-over Cost per Location

SE.69: *Shortest path with deterministic (i.e. node-dependent) stop-over cost.* Let an undirected simple graph $G = (V, E)$ be given with n nodes and m edges. The edges have an associated positive weight: $w(x, y) > 0$ for the edge between nodes x and y. This questions concerns itself with a variation of shortest path. For the purpose of this question, if a particular path from x to y is $x \to a \to b \to c \to y$, then we refer to a, b, and c as lay-over nodes for that particular path. First some background: there are two kinds of well known shortest path problems:

1. The shortest hop-count distance between two nodes s and t is defined as finding a path from s to t with the fewest number of links (or equivalently, the fewest lay-over nodes) from among all paths from s to t. Weights that are possibly defined on edges are not used in this problem, and the problem can be solved with a (variation of) Breadth-First-Search.
2. The shortest weight path between two nodes s and t is defined as finding a path from s to t such that the sum of all the weights of the edges are minimal from among all the possible paths from s to t. The hop count (number of lay-over nodes) is not used in this problem. There are several algorithms known to solve this problem efficiently, such as the ones named after Dijkstra, Floyd and Warshall. Bellman and Ford are credited with extending the problem to graphs with potentially negative weights.

Now your question: Design an efficient algorithm that finds the shortest weighted distance between two given nodes s and t, with the additional requirement: There is a cost associated with lay-over nodes: If a is a lay-over node on the path, then you incur an additional cost of $c(a)$. Your algorithm should run in (at most) polynomial time. It need not be the optimal solution, but both the algorithm and it's analysis must be correct. More specific:

1. Make an assumption about the graph representation as either an adjacency matrix or with adjacency lists. Then design an algorithm that correctly solves the problem. Full credit will be given to an algorithm that is correct and whose correctness argument has appropriate rigor. It is more important to convey your approach and thought process at the algorithmic level, and less important to have a syntactically correct program. The latter will bog you down, and you are bound to make trivial mistakes.

2. Analyze the algorithm and provide it's asymptotic time complexity. Full credit will be given to a correct analysis that is described and communicated with appropriate rigor.

Several solution approaches are possible. The easiest is probably to extend the algorithm named after Dijkstra which finds the shortest paths from s to t. Next, whenever an edge (u, w) is relaxed, replace the standard relaxation comparison
if $DistanceToS[u] + LinkLength[u, w] < DistanceToS[w]$

by the updated criterion
if $DistanceToS[u] + LinkLength[u, w] + c(u) < DistanceToS[w]$

The rest is straightforward.
The time complexity is not changed: $\Theta(n^2)$ for adjacency matrices and $\Theta(m \lg n)$ for adjacency lists and min-heaps. Space complexity is also not changed in asymptotic class, but the coefficient corresponding to the n-term is increased by 1.

12.8 Shortest Path with Stop-over Cost per Stop

Let an undirected simple graph $G = (V, E)$ be given with n nodes and m edges. The edges have an associated positive weight: $w(x, y) > 0$ for the edge between nodes x and y. This questions concerns itself with a variation of shortest path. For the purpose of this question, if a particular path from x to y is $x \to a \to b \to c \to y$, then we refer to a, b, and c as lay-over nodes for that particular path. First some background: there are two kinds of well known shortest path problems:

1. The shortest hop-count distance between two nodes s and t is defined as finding a path from s to t with the fewest number of links (or equivalently, the fewest lay-over nodes) from among all paths from s to t. Weights that are possibly defined on edges are not used in this problem, and the problem can be solved with a (variation of) Breadth-First-Search.
2. The shortest weight path between two nodes s and t is defined as finding a path from s to t such that the sum of all the weights of the edges are minimal from among all the possible paths from s to t. The hop count (number of lay-over nodes) is not used in this problem. There are several algorithms known to solve this problem efficiently, such as the ones named after Dijkstra, Floyd and Warshall. Bellman and Ford are credited with extending the problem to graphs with potentially negative weights.

Now your question: Design an efficient algorithm that finds the shortest weighted distance between two given nodes s and t, with the additional requirement: There is a cost associated with lay-over nodes. You can visit up to two lay-over nodes for free, but every additional lay-over will incur a fee. Let $c(k^{\text{th}})$ the cost associated with the k^{th} node on the path. Your algorithm should run in (at most) polynomial time. It need not be the optimal solution, but both the algorithm and it's analysis must be correct. More specific:

1. Make an assumption about the graph representation as either an adjacency matrix or with adjacency lists. Then design an algorithm that correctly solves the problem. Full credit will be given to an algorithm that is correct and whose correctness argument has appropriate rigor. It is more important to convey your approach and thought process at the algorithmic level, and less important to have a syntactically correct program. The latter will bog you down, and you are bound to make trivial mistakes.
2. Analyze the algorithm and provide it's asymptotic time complexity. Full credit will be given to a correct analysis that is described and communicated with appropriate rigor.

Several solution approaches are possible. The easiest is probably to extend the algorithm named after Dijkstra which finds the shortest paths from s to t regardless of the number of lay-over points in between. We now need to track the number of layovers as well. This is a path-property, not a node property. So introduce a $LayOver$- label for each path, which is programmatically accomplished by attaching it label to a node, much like the $CameFrom$-label. The $LayOver$- labels are initialized as 0's.

Next, whenever an edge (u, w) is relaxed, replace the standard relaxation comparison
if $DistanceToS[u] + LinkLength[u, w] < DistanceToS[w]$

by the updated criterion
if $DistanceToS[u] + LinkLength[u, w] + c(1 + LayOver[u]) < DistanceToS[w]$

The rest is straightforward: if the distance can be improved, update the LayOver-labels as well, (in addition to updating the Distance Labels and the Predecessor-labels.)

The time complexity is not changed: $\Theta(n^2)$ for adjacency matrices and $\Theta(m \lg n)$ for adjacency lists and min-heaps. Space complexity is also not changed in asymptotic class, but the coefficient corresponding to the n-term is increased by 1.

12.9 Shortest simple cycle of fixed length k.

SE.70: Find the shortest simple cycle that has exactly $k = 10$ lay-over nodes.

12.10 Shortest simple cycle of at least length k.

SE.71: Find the shortest simple cycle that has at least $k = 10$ lay-over nodes.

12.11 Shortest simple cycle with length restricted on both sides.

SE.72: *Shortest simple cycle of with length restriction - 2.* Find the shortest simple cycle that has between $k = 5$ and $k = 10$ lay-over nodes.

12.12 Shortest non-trivial cycle of any length.

SE.73: Let an undirected simple graph $G = (V, E)$ be given with n nodes and m edges. The edges have an associated positive weight: $w(x, y)$ for the edge between nodes x and y. This question concerns itself with shortest cycles, and for the purpose of this question, if a particular cycle from x to x is $x \to a \to b \to c \to x$, then we refer to a, b, and c as lay-over nodes for that particular cycle. First some background: there are two well known shortest cycle problems:

1. An Eulerian cycle is a cycle from a node x back to x such that the cycle traverses through *all* edges exactly once, while a node can be visited multiple times. A famous example is the Königburg's Seven Bridges. Various efficient algorithms are known and run in $\mathcal{O}(m)$ when a doubly linked list is used.
2. A Hamiltonian cycle is a cycle from a node x back to x such that the cycle traverses through *all* other nodes exactly once. (and if there is more than one Hamiltonian cycle, then the Traveling Salesman uses the one with minimal combined weight). Various algorithms are known, and none of the known algorithms run in polynomial time.

Now your question: Design an efficient algorithm that determines the shortest cycle from any node x back to x that uses at least two (2) different lay-over nodes. Edges can appear only once on the cycle: if there are ℓ lay-over nodes, then there $\ell + 1$ different edges on the cycle. Your algorithm should run in (at most) polynomial time. It need not be the optimal solution, but both the algorithm and it's analysis must be correct. More specific:

1. Make an assumption about the graph representation as either an adjacency matrix or with adjacency lists. Then design an algorithm that correctly solves the problem. Full credit will be given to an algorithm that is correct and whose correctness argument has appropriate rigor. It is more important to convey your approach and thought process at the algorithmic level, and less important to have a syntactically correct program. The latter will bog you down, and you are bound to make trivial mistakes.
2. Analyze the algorithm and provide it's asymptotic time complexity. Full credit will be given to a correct analysis that is described and communicated with appropriate rigor.

Several solution approaches are possible. The easiest is probably to first run one of the algorithms like Dijkstra or Floyd Warshall, which results in a table $D[i, j]$ so that the (i, j)-th element of this table represents the minimal cost (shortest path) to travel from node i to node j. So use the information $D[x, j]$ to find the minimal cost from x to some node j and complete the cycle back to x: so consider the minimal cost of $D[x, j] + A[j, x]$ over the allowable values of j. So at first, consider

$$AlmostShortestCycle = \min_{j=1\ldots n} \{ D[x, j] + A[j, x] \}$$

where A is the original adjacency matrix. There are two subtle problems. The first is that there must be at least one edge in the cycle, so make sure that the adjacency matrix (if used as a data structure) does not include $A[x, x] = 0$ which would falsely imply that there is a path with 0 length from x to x. Having $A[x, x] = \infty$ would prevent this. The second subtle problem is for an undirected graph where the shortest distance from x to j is possibly the 1-hop distance, i.e. $D[x, j] = A[x, j]$. This would give a 2-link cycle, and only 1 layover node. There are several ways to test for this. For instance, test to make sure that the predecessor node for node j is not equal to x. Let $\pi[j]$ the predecessor node for the shortest path to node j (as in Dijkstra or Floyd-Warshall), then

$$ShortestCycle = \min_{j, x \notin \pi\{j\}} \{ D[x, j] + A[j, x] \}$$

The time complexity is not changed: $m \lg n$ for Dijkstra, or $\Theta(n^3)$ for Floyd-Warshall. The added cost is linear which is well below the complexity of the first algorithm.

13 Former Projects

I have included some former projects in either the undergraduate or the graduate version of the algorithms course at our university. The descriptions do not present polished presentations and are intended to guide the student to explore and discover the answer. Projects are intended to be thought provoking and intended to be done in teams of 3 or 4 for full benefit of all. Often, multiple directions are described in the literature, these are all possible and all acceptable as long as the team justifies their choices. The process of deriving answers becomes the valued learning experience: out-of-the-box thinking and getting the satisfaction of the AHA-moment. If done well, the students gains a full understanding and appreciation of the problem, the algorithm, and it's application. Generally, students also appreciate the learning process itself and look back with pride at their accomplishments.

13.1 Runner Up to Shortest Paths

Please note that this is a former project. The description does not present a polished presentation and is intended to guide the student to explore and to discover the answer. The project is intended to be thought provoking and intended to be done in teams of 3 *or* 4 *for full benefit of all. Multiple directions are described in the literature, these are all possible and all acceptable as long as the team justifies their choices. The process of deriving answers becomes the valued learning experience: out-of-the-box thinking and getting the satisfaction of the AHA-moment. Students gains a full understanding and appreciation of the problem, the algorithm, and it's application. Generally, students also appreciate the learning process itself and look back with pride at their accomplishments.*

Was a project in CS404, Fall Semester 2014

Summary

A popular algorithm to find and generate paths with minimal cost in a graph is the Floyd-Warshall algorithm, which finds and generates the optimal paths between all possible source-destination pairs. It is popular because it is easy to understand and implement. But sometimes, secondary considerations come into play for route selection, and 'near'-optimal solutions might be preferred. This project explores whether or not Floyd-Warshall can be extended and such additional requests be accommodated.

Motivation

Commercial Airlines often model their routes are graphs. Traveling over roads is also modeled as a network. Computer and communication networks are very much the same. There are other examples in cloud computing, where information needs to stored in one or more locations, and this uses a network, of course. In a network, nodes are the cities, or intersections, or server nodes (sources of demand, such as traffic or communication sites), and the links connect hem. If you want to go from one node to another (by plane, by car, or routing internet traffic or information packet), the you need a plan: source, destination, and a planned path in between. Let us explain this in terms of an airline routing network as the motivating application, but keep in mind that this project might have additional uses. The main difference is that when planning an airline trip, you have some time (read: computing time) to find the route with the minimal cost. In the internet, you generally do not have time to compute this, and routing tables are pre-computed and stored at each node. (In actuality, the full routing table would take up too much space at the routers. All we need for this project is that we need to pre-compute routes.)

When you start to plan a trip, for instance from MCI to AMS, you quickly come to realize that there is no direct connection between the two nodes. So you look at several different routes, from several different airlines, and each with its own cost (in $$). As a student, you have more time than you have money, so you pick the route with minimal $$-cost. If all you need are the routes between two fixed end-points (like the air-line example), then you should use Dijkstra's algorithm for that particular source-destination pair. If you need to generate routes for all node-pairs, you should use Floyd-Warshall to generate your minimal-cost information for all node-pairs, together with enough info on how to generate the path.

However, there are secondary considerations. Like number of stops on the route, travel time, frequent flyer miles, traveling over combat zones, and so on. So you are looking for a system that provides you with options, so that you can decide for your self: take a flight for $1175 that takes 5 intermediate stops, or lay-over nodes (i.e. 6 hops) and a whopping 23 hours, or take the second best option (in terms of dollars), which is a route that costs $1200, has only 1 stop (2 hops), and takes only 10 hours. So how can you generate multiple paths? As a gut reaction, you might just say: look at all paths between all the node pairs. But this gives you a combinatorial explosion: as there are $n!$ separate cases to generate and evaluate. So we need to be more frugal. If all you need are the alternate routes between two fixed end-points (like the air-line example), then you should extend Dijkstra's algorithm for that particular source-destination pair. If you need to generate alternate routes for all node-pairs, you should extend Floyd-Warshall and generate your minimal-cost information for all node-pairs, together with enough info on how to generate the path. That is what you are asked to generate in this CS404 project.

Problem Statement

Extend and adjust the algorithm by Floyd-Warsall to generate three (3) shortest paths (in terms of $$ cost) for every source and destination pairs. Of course, these paths may have links in common. On the other hand, it could very well be that there is only one or two routes between some source-destination pairs. If the number of different paths is less than three, then your program should report this. The optimal path is simple (i.e. no loops) by construction, but the other paths are allowed to have loops. Furthermore, you do not need to prove that your 'second-best' route is indeed 'second best', since this is hard to do. But your policy should make sense. (Extra credit will be given if your algorithm never generates loops, if you follow it with an extra loop-removal algorithm, or if you give convincing arguments that your second route is indeed second best).
For this project, you must design an algorithm (with supporting abstract data structures in an algorithmic language), that solves the problem. You should test your algorithm, and hand simulate it to test your ideas and convince yourself it is correct. At this time, you should analyze your algorithm for time (and space) complexity; it is acceptable to only analyze for *best*- and *worst*-case situations.

After this is done, you must implement your algorithm in any language of your choice, but you must explain why you used that language. You are further somewhat limited by the fact that you must be able to demonstrate your program on one of the computers in the SCE-labs, and that it is able to read in the file that we generate. Recommended languages (in terms of speed) C, and $C++$, but also acceptable are perhaps JAVA, Python, R, ... Of course, we expect you to report on resource consumption by your program. In other words, your report should contain something like "Case ## took ### milliseconds to run", while indicating language, platform, computer, available memory, and other such things.

For this project, you must design an algorithm (with supporting abstract data structures in an algorithmic language), that solves the problem. Explore at least two options, keeping in mind that you need to efficiently find the optimal path plus 4 paths that you believe are near-optimal. For both options, you need to analyze the resource requirements, and decide on which option to implement. You must report and document your options, and their analyses. In particular, you need to give

1. Convincing arguments (proof? counter-example?) that your options is a good heuristic to find the correct alternate paths,
2. An analysis of your options for best-case and worst-case time complexity as well as space complexity before you implement them.
3. Conclude this part with a decision
4. Implement your option and tests for correctness.
5. For the cases that will be provided, measure the run times of your programs to experimentally verify your analytical findings.
6. Indicate whether or not you believe that your option indeed provides the best alternate paths.

This is a Draft

This is a draft, and there are still some things that Shuai and I need to discuss (such as the format of the input data, how to get you the input data, and how you can submit your project.) But at least you can get started on the project. Shuai will be at a conference next week, so it will be at least one week before the final version will be given.

Strong Recommendation

CS404 is a course on algorithm design, and I would expect that this would be the most time consuming task. Particularly, because you are only tasked with 'What' needs to be accomplished, and you yourself have to decide on the 'How'. This part is the most rewarding phase, but also the most frustrating phase of the project, since it is impossible to predict how long this takes for anyone, and it is impossible to realize how close you may be to a solution. Just do not panic, and go back to the basics: What do you currently have, where do you need to go, and how can you use what you learned in CS303 (and other courses) to your advantage. Structure your thinking, document what you could try and have tried already. Once you get the big idea, the programming is less challenging due to your strong programming and coding skills, but the interpretation of the results will take some time again. So set yourself realistic milestones, and build in extra time to recover from setbacks. The key is: Get started. Start thinking. And enjoy the aha-moment.

13.2 MultiCarrier Shortest Path Algorithm

Please note that this is a former project. The description does not present a polished presentation and is intended to guide the student to explore and to discover the answer. The project is intended to be thought provoking and intended to be done in teams of 3 or 4 for full benefit of all. Multiple directions are described in the literature, these are all possible and all acceptable as long as the team justifies their choices. The process of deriving answers becomes the valued learning experience: out-of-the-box thinking and getting the satisfaction of the AHA-moment. Students gains a full understanding and appreciation of the problem, the algorithm, and it's application. Generally, students also appreciate the learning process itself and look back with pride at their accomplishments.

Was a project in CS5592, Fall Semester 2002

Motivation
Traditional shortest path algorithms find the shortest path in a simple weighted graph. The classical solution is attributed to Dijkstra. If the graph is no longer simple however, then Dijkstra's algorithm fails to be correct, and that is the situation you have to solve.

Given a simple graph $G =< V, E >$, with $n = |V|$, and $m = |E|$. Assume now however, that for each edge in the graph there are really k parallel options you can choose. You can think of it as a geographical map with n cities. You can travel from one city to another using one of k different carriers: different airlines, trains, own car, rented car, motorcycle, bicycle, walking, hitchhiking, etcetera. Assume that each carrier has its own cost associated with the link, and that in each node you can switch carriers, of course at a switch-over cost. Thus you will be given k matrices, each of dimension $n \times n$ and representing the connection weights for a different carrier. Furthermore, you will be given n matrices, each of dimension $k \times k$, each representing the switch-over cost at every city.

You are to design, analyze, and implement an algorithm (with supporting data structures) to find the shortest path (both the shortest hop count as well as the minimal total weights, from the first node to all other nodes. You can make the following assumptions:

1. Entries are positive integers, except that a 0 indicates that no connection (or switch-over) is possible.
2. 'ShortestWeight' is defined as the minimum of the accumulated edge cost plus switchover costs from the starting node to the end node.
3. 'ShortestHop' is defined as the minimum of the edges from the starting node to the end node, i.e. switchovers are considered 'free'.
4. Your program should print out the 'ShortestHop' value, together with the path that has this 'ShortestHop' value, and it should print out the value of 'ShortestWeight', together with the path that has this 'ShortestWeight' value.
5. You only need to analyze for worst case time complexity, and you assume that the graph is completely connected, $m = \frac{n(n+1)}{2}$
6. In the program you write you must do an actual 'count' of the number of key-and-basic operations to validate your analysis.
7. Hint: You can solve the problem either by adjusting Dijkstra's algorithm to fit the graph, by adjusting the graph so that Dijkstra's algorithm can be used on the adjusted graph, or a combination of the two approaches.

So for this project, you must design several algorithms (with supporting abstract data structures in an algorithmic language), that solves the problem(s). You should test your algorithm(s), and hand simulate them to convince yourself they are correct. At this time, you should analyze your algorithms for time complexity, it is acceptable to only analyze for *worst*-case time complexity. After this is done, you must implement your algorithms, and you are free to use any language you desire, including C++, MAPLE, or whatever you decide is the appropriate language for this problem. The only restriction is that you must be able to demonstrate your program to me in person in Flarsheim Hall, so it must either run on a machine here, or on your laptop. At this time, and not any earlier, should you decide on how the abstract data structures should be implemented as implementation data structures (if at all!). Should you make a customized version? (Please do not forget to properly report the sources you use, only publicly available sources are allowed, not 'my friend'.) Do you need any other supporting data structures, like list, stack, queue, heap? Do you need a standard one or customized versions? After you made your program, you need to test the program to convince yourself it is a correct implementation of your algorithm. You can do this by constructing small test cases yourself, where you can compute (that is, predict) the output independently beforehand, and compare the program output with your precomputed answers. You should perhaps also generate some larger test cases and carefully inspect the program output to make sure that all output makes sense. Generally, testing means 'trying to break the program', attack it in every way to try to make it fail under the given assumptions. It is usually better to find the errors yourself. There will be at most $n = 100$ nodes per graph, and $k = 10$ different carriers for the input that I will provide.

Follow-up Project: Repeat project 1, but now you can no longer assume that all weights and switchover costs are positive, but could be negative. Your program should detect whether or not there is a negative cycles, and if there is no cycle, your program should find again the 'ShortestHop' and 'ShortestWeight' values, together with their paths.

13.3 First and Second Shortest Paths in a Grid

Please note that this is a former project. The description does not present a polished presentation and is intended to guide the student to explore and to discover the answer. The project is intended to be thought provoking and intended to be done in teams of 3 or 4 for full benefit of all. Multiple directions are described in the literature, these are all possible and all acceptable as long as the team justifies their choices. The process of deriving answers becomes the valued learning experience: out-of-the-box thinking and getting the satisfaction of the AHA-moment. Students gains a full understanding and appreciation of the problem, the algorithm, and it's application. Generally, students also appreciate the learning process itself and look back with pride at their accomplishments.

Was a project in CS404, Spring Semester 2016

Consider the problem of shortest route selection in a grid. There are several algorithms to find a shortest path, which you would have to adjust if you want to find the "second best" option as well. This is what we will explore in this project, where you will be asked to find the two best paths in a grid. The problem setting:

An m-by-n grid $\langle G \rangle$ has $m \times n$ grid-cells, $G[i, j]$, and each grid-location $[i, j]$ has an associated cost $C[i, j]$. You can assume that m and n are larger than 2 and that costs are natural numbers. A valid path from $G[1, 1]$ to $G[m, n]$ for this project, is a sequence P of grid-locations from $[1, 1]$ to $[m, n]$ so that

1. The path starts with location $G[1, 1]$
2. If location $G[i, j]$ is on the path, then either $G[i + 1, j]$ or $G[i, j + 1]$ is on the path
3. The path ends with location $G[n, m]$

The *cost of a path*, which we call *PathCost*, is defined as the total sum of the costs $C[i, j]$ on the path. There are many paths and each path has its PathCost. We are interested in finding a path minimal PathCost. The minimal PathCost is unique and well defined, although there may be several paths with the same minimal PathCost. For this project you are to find two paths: a *FirstPath* whose PathCost is minimal from among all paths and a *SecondPath* whose PathCost is either equal to that of the *FirstPath*, or whose PathCost is the second minimal cost. You are asked to identify a particular path as follows: Represent a step from $G[i, j]$ to $G[i + 1, j]$ by a "0" and a step from $G[i, j]$ to $G[i, j + 1]$ by a "1". Thus a path that starts with $G[1, 1] \rightarrow G[1, 2] \rightarrow G[1, 3] \rightarrow G[2, 3] \ldots$ is represented by $1\ 1\ 0 \ldots\ldots$. If there are multiple paths with the same cost, then you should present the path that corresponds to the smallest number when the $0/1$ pattern is interpreted as a binary number.

Possible approaches

This is a new problem, I presume, and there are several ways to resolve this. A straightforward enumeration or generation of all paths and tracking their costs is simply not feasible, even for fairly small grids. The number of valid paths from $G[1, 1]$ to $G[m, n]$ with our restrictions is a staggering $\binom{(m-1)+(n-1)}{m-1}$, see any book on algebra, discrete math, or probability for details. So for a 10-by-10 grid, there are $48,620$ paths, but for a 20-by-20 grid this is $\approx 3.5\ 10^{10}$, and for a 100-by-100 this has risen to $\approx 2.27\ 10^{58}$. So you need to look into alternate approaches

The next approach builds on the realization that a grid is just a graph with additional structure. Dijkstra's shortest path algorithm (DSPA) can be adjusted to generate a *FirstPath* and a *SecondPath*, and this is one of the approaches you are asked to take in this project. You can find more info on DSPA in the book, on wiki, and many other references.

Another approach takes advantage of the additional structure presented by a grid, and on the fact that a shortest path (with positive weights per node) satisfies the "principle of optimality". A dynamic programming approach can be thus be taken and the grid structure makes this fairly attractive and straightforward. Introduce $M[i\ ,\ j]$ as indicating the minimal cost possible for any path from $G[1, 1]$ to $G[i, j]$, then the driving equation is (with $M[1\ ,\ 1] = C[1\ ,\ 1]$ and $M[i\ j] = \infty$ whenever $i = 0$ or $j = 0$):

$$M[i\ ,\ j] = \min_{\text{2 options}} \{\ M[i-1\ ,\ j], M[i\ ,\ j-1]\} \quad + \quad C[i\ ,\ j]$$

Although this is recursively defined, it is best implemented iteratively, such as
LINE 100: $M[1, 1] \leftarrow C[1, 1]$
LINE 110: **for** j **from** 2 **to** m **do** $M[1, j] \leftarrow M[1, j - 1] + C[1, j]$ **end-do**
LINE 120: **for** i **from** 2 **to** n **do**
LINE 130: $\quad M[i, 1] \leftarrow M[i - 1, 1] + C[i, 1]$
LINE 140: $\quad$ **for** j **from** 2 **to** m **do** $M[i, j] \leftarrow \min\{M[i - 1, j],\ M[i, j - 1]\} + C[i, j]$ **end-do**
LINE 150: **end-do**

This approach can be also be adjusted to generate both a *FirstPath* and a *SecondPath*, and this is the other approach you are asked to take in this project.

13.4 The Sneaky Path

Please note that this is a former project. The description does not present a polished presentation and is intended to guide the student to explore and to discover the answer. The project is intended to be thought provoking and intended to be done in teams of 3 or 4 for full benefit of all. Multiple directions are described in the literature, these are all possible and all acceptable as long as the team justifies their choices. The process of deriving answers becomes the valued learning experience: out-of-the-box thinking and getting the satisfaction of the AHA-moment. Students gains a full understanding and appreciation of the problem, the algorithm, and it's application. Generally, students also appreciate the learning process itself and look back with pride at their accomplishments.

Was a project in CS404, Fall Semester 2016

Summary

Finding an optimal path in a network is not always the shortest path, shortest distance, or cheapest cost, but rather: most scenic, least traffic, get-away path, and so on. This project explores finding a *SneakyPath*.

Motivation

Commercial Airlines, roads, and computer and communication networks often model their routes as graphs. There are other examples in social networks and cloud computing, and so on. In such a network, nodes are the cities, or intersections, or server nodes (sources of demand, such as traffic or communication sites), and the links connect hem. If you want to go from one node to another (by plane, by car, or routing internet traffic or information packet), then you need a plan: source, destination, and a planned path in between. Let us explain this in terms of an road network between cities as the motivating application, but keep in mind that this project might have additional uses. When planning a car-trip, say from a to b, most people would want to find the shortest route, and there are several algorithms known that will help you find the shortest path. But this may not always be the criterion used. Some people may be interested in the least travelled road, to avoid as much congestion as possible. They want to avoid other cars as much as possible, they do not want to see other cars, they do not other cars to see them, or whatever. May be you are towing a 12-foot wide boat, and you do not want to inconvenience other drivers, so you look for the least travelled road. Even in computer networks this might be of interest: lowest probability of dropped packets. We will call such a path a *SneakyPath*.

You will be given a number of different input scenario's. Each scenario will have

1. The size n of the system (the total number of cities),
2. an adjacency matrix $\mathbf{E}$ with edge-weights. The $(i,j)^{\text{th}}$ entry $\mathbf{E}[i,j]$ represents the traveling time on the (direct) edge between the two specific nodes, i and j.

The matrix $\mathbf{E}$ is will allow you to compute the shortest path from a to b, and several algorithms could be used for this. The shortest path might take several edges (or links), and if you are the only one on the road, then this is the path you will take. However, you are not the only person/car on the road. You will also be given

3. a flow matrix $\mathbf{F}$, whose $(i,j)^{\text{th}}$ entry $\mathbf{F}[i,j]$ represents the number of (other) cars that travel the entire path from node i to node j every hour.

For the sake of argument, say that there are $\mathbf{F}[21,47] = 115$ cars traveling on the path from node v_{21} to node v_{47}, then there are 115 cars on each edge of that path. Suppose edge (v_{34}, v_{38}) is on the shortest path from node v_{21} to node v_{47}, then there are also 115 cars/hour on the edge (v_{34}, v_{38}) due to the flow $\mathbf{F}[21,47]$. But is very well possible that the edge (v_{34}, v_{38}) is also on the shortest path from node v_{17} to node v_{29}, and that $\mathbf{F}[17,29] = 75$. This means that there are at least 190 cars/hour on the edge (v_{34}, v_{38}). Thus for each edge (i,j), you can calculate the actual carried traffic load from all source-destination pairs on that edge, because everybody takes their shortest path. Put all this information in a new matrix $\mathbf{L}$, each entry representing the total load on the edge. Finally, you will be given

4. the starting node a and the terminal node b

And for this particular pair, you are asked to calculate (and print):

a) The *SneakyPath* from a to b, such that the total number of other cars on the road encountered is as small as possible,

b) the edge on this *SneakyPath* with the lowest number of other cars,

c) the edge on this *SneakyPath* with the highest number of other cars,

d) the average number of other cars on the *SneakyPath*, averaged over the number of links on the path.

This is however a course on algorithms, and you also have to report on the algorithms you used to actually find the information asked above. In particular,

e) The worst case time complexity for the algorithm(s) you used,

f) the measured CPU-usage of the programs when it solved each case,

g) the empirical validation that your programs have the correct asymptotic complexity.

Project expectations

You need to explore and expand the algorithms for finding the information as outlined above, and you should feel free to explore additional algorithms that you feel are appropriate. The goal is to find an efficient algorithm for the problem and to implement them efficiently. You can write separate algorithms/programs for the separate parts, or use the same one, whichever is most appropriate for the problem. You need to write a report in which you address the following issues:

1. Explain the reasoning for using the algorithms you implemented (as opposed to alternate choices), explain any supporting algorithms and data structures. Obvious potential choices are Dijkstra's SPA, and Floyd-Marshall SPA.
2. Explain the algorithm you used and implemented to actually generate the paths between nodes, so that the load per edge can be calculated. If there were two ideas you were working on, what are the pro's and con's? You may use secondary metrics, such as ease of programming, ease of software maintenance, portability, and so on.
3. Provide convincing arguments that your algorithms find a correct *SneakyPath* or demonstrate by counter example that it does not always find a *SneakyPath* and explain why this should be acceptable.
4. Present an analysis of your algorithms for best-case and worst-case time complexities as well as space complexities. Make sure to mention the key-and-basic operation you are using.
5. Implement the algorithms that is best suited (or acceptably suited) for networks where n is bounded above by $1,000$. Best-suited means better performance.
6. Provide convincing arguments that your have implemented your algorithms correctly and efficiently with the appropriate data structure.
7. Perform a timing study of your implementation and show that this conforms with your (pre-implementation) time complexity study. In other words, measure the run times of your programs to experimentally verify your analytical findings. The least you can do is a plot (size n versus observed times).
8. For the cases that will be provided, present the *SneakyPath*.
9. Feel free to add test cases yourself that illustrate certain algorithmic or program features
10. Indicate whether or not you believe that there is a better (i.e. faster run-time) solution with an explanation on why you believe this to be the case or not to be the case.
11. As a last item, reflect back on this project and on the design and data structure decisions you made. How did you handle any unforeseen situations and any 'newly discovered' problems? Now that you have done the project, what have you learned? If you have the opportunity to do this project again, perhaps with additional requirements, what would you do the same, and what would you do differently? What are the lasting lessons you have learned? What are the lessons for future students?

13.5 Stochastic Shortest Path

Please note that this is a former project. The description does not present a polished presentation and is intended to guide the student to explore and to discover the answer. The project is intended to be thought provoking and intended to be done in teams of 3 or 4 for full benefit of all. Multiple directions are described in the literature, these are all possible and all acceptable as long as the team justifies their choices. The process of deriving answers becomes the valued learning experience: out-of-the-box thinking and getting the satisfaction of the AHA-moment. Students gains a full understanding and appreciation of the problem, the algorithm, and it's application. Generally, students also appreciate the learning process itself and look back with pride at their accomplishments.

Was a project in CS5592, Spring Semester 2017

Roads, airspace, and computer and communication networks are often modeled as graphs. There are other examples in social networks and cloud computing, and so on. In such a network, nodes are the cities, or intersections, or server nodes (sources of demand, such as traffic or communication sites), and the links connect hem. If you want to go from one node to another (by plane, by car, or routing internet traffic or information packet), then you need a plan: source, destination, and a planned path in between. Let us explain this in terms of an road network between cities as the motivating application, but keep in mind that this project might have additional uses. When planning a car-trip, say from a to b, most people would want to find the shortest route, and there are several algorithms known that will help you find the shortest path. But please do not forget that you are not the only car on the road, and with more cars on the road will cause interference and extra delay because you share the road with more and more others. In other words: the delay along an edge is rarely a constant value, but rather fluctuates according to a random variable for each edge: the value is uncertain. We are still interested in finding some sort of shortest path, but shortest is now defined in a stochastic setting. We will call such a paths a *UncertainShortestPath*.

You will be given a number of different input scenario's. Each scenario will have

1. The size n of the system (the total number of cities), together with the index of two cities, a and b.
2. An adjacency matrix $\mathbf{E}$ with edge-weights. The $(i,j)^{\text{th}}$ entry $\mathbf{E}[i,j]$ represents the main characteristics of the random variable representing the traveling time on the edge between the two specific nodes, i and j, assuming that there are no other cars on the edge that might slow you down. The main characteristics we provide are T, the type of distribution, together with two more parameters, α and β, whose interpretation depends on the type, as follows:

Type	Name	Parameter 1 α	Parameter 2 β	Mean	Variance	c^2
1	Deterministic	at value	not used (ignore value)	α	0	0
2	Uniform$[\alpha,\beta]$	low value	high value	$\frac{\alpha+\beta}{2}$	$\frac{(\beta-\alpha)^2}{12}$	$\frac{1}{3}(\frac{\beta-\alpha}{\alpha+\beta})^2$
3	Exponential	mean rate	not used (ignore value)	$\frac{1}{\alpha}$	$\frac{1}{\alpha^2}$	1
4	Shifted Exponential	mean rate	shifted value	$\beta+\frac{1}{\alpha}$	$\frac{1}{\alpha^2}$	$(\frac{1}{1+\beta\alpha})^2$
5	Normal	location value	variance	α	β	$\frac{\beta}{\alpha^2}$
6	General	Mean value	squared coefficient of variation	α	$\alpha^2\,\beta$	β

   Thus, an entry with "13, 24, 2, 10, 20" indicates that the edge -cost between nodes 13 and 24 is distributed according to a (continuous) uniform distribution between the interval $[10, 20]$. Similarly, an entry with "14, 13, 3, 20, 33" indicates that the edge -cost between nodes 14 and 13 is distributed according to an exponential distribution with mean 20 (and the value 33 is ignored.

With the two nodes a and b, and the matrix $\mathbf{E}$ you can design variations of Dijkstra's Shortest Path algorithm, as long as we know what criteria to use for "shortest." You will be asked to generate shortest paths according to several criteria, and then compare these paths. Before reviewing these criteria, we review some probability and review Dijkstra's Algorithm.

Some Background on Random Variables

You will need some background from probability, and there is a lot of notation that comes with it. First, let the delay on the link between the nodes i and j be represented by the random variable (RV) $\mathsf{X}_{L(i,j)}$. Furthermore, let a path P from a to b be given by the sequence of nodes $P = \langle a, x_1, x_2 \cdots x_m, b \rangle$, then the path delay from a to b is given by the sum of the RV's on each edge on the path. I will use Y for RV's for a Path, and will use X for RV's on a link. Thus:

$$\mathsf{Y}_{P\langle a,x_1,x_2\cdots x_m,b\,\rangle} = \mathsf{X}_{L(a,x_1)} + \mathsf{X}_{L(x_1,x_2)} + \mathsf{X}_{L(x_2,x_3)} \ldots + \mathsf{X}_{L(x_m,b)} \tag{11}$$

Assuming that all Link-RV's are independent, then both the expected value and the variance of a sum of RV's is equal to the sum of the expected values and variances of the individual RV's:

$$\mathsf{E}\left[\mathsf{Y}_{P\langle a,x_1,x_2\cdots x_m,b\,\rangle}\right] = \mathsf{E}\left[\mathsf{X}_{L(a,x_1)}\right] + \mathsf{E}\left[\mathsf{X}_{L(x_1,x_2)}\right] + \ldots + \mathsf{E}\left[\mathsf{X}_{L(x_m,b)}\right] \tag{12}$$

$$\mathsf{V}\left[\mathsf{Y}_{P\langle a,x_1,x_2\cdots x_m,b\,\rangle}\right] = \mathsf{V}\left[\mathsf{X}_{L(a,x_1)}\right] + \mathsf{V}\left[\mathsf{X}_{L(x_1,x_2)}\right] + \ldots + \mathsf{V}\left[\mathsf{X}_{L(x_m,b)}\right] \tag{13}$$

Notice therefore that finding expected values and variances of Path-RV's are simple expressions. The coefficient of variation (of an RV) is defined as

$$c(\mathsf{X}) = \sqrt{\frac{\mathsf{V}[\mathsf{X}]}{\mathsf{E}[\mathsf{X}]^2}} \qquad \text{or} \quad c^2(\mathsf{X}) = \frac{\mathsf{V}[\mathsf{X}]}{\mathsf{E}[\mathsf{X}]^2} \tag{14}$$

and we often avoid using the radical (i.e. square root) by using the squared coefficient of variation. When you look at it, it just a normalized variance, and the pair "Mean+squared Coeff of Var" gives just as much information as just "Mean+Variance", it is just that in modeling delays in communication networks, it more commonly used as a "mean-independent" measure of variance. Generally speaking, an RV(or density) whose c^2 is less than 1 is more concentrated around it's mean, whereas an RV(or density) whose c^2 is larger than 1 is less concentrated around it's mean, and is used to represent time durations that are with small probability could be extremely long. Notice also, that the squared coefficients of variation for a sum of RV's is no longer linear in the individual c^2s. In fact, for any two independent RV's:

$$c^2(\mathsf{X}+\mathsf{X}') < \max\{c^2(\mathsf{X}), c^2(\mathsf{X}')\} \qquad \text{and} \tag{15}$$

$$c^2(\mathsf{X}+\mathsf{X}') = \frac{1}{2}c^2(\mathsf{X}) \qquad \text{if} \quad \mathsf{X} \text{ and } \mathsf{X}' \text{ are identically distributed} \tag{16}$$

so that, generally speaking, you can expect that the squared coefficient of variation becomes smaller as the path gets longer. For this project, we indeed assume that link-delays are all independent, but extra credit can be given if you incorporate positive correlations between two neighboring edges on a path. We note that finding it's exact expression is nearly impossible (remember the convolution integrals?). However, by taking Laplace Transforms, (and assuming independence), we get that the Laplace Transform of $\mathsf{Y}_{P\langle a,x_1,x_2\cdots x_m,b\,\rangle}$ is the product of the individual transforms:

$$\mathcal{F}_{P\langle a,x_1,x_2\cdots x_m,b\,\rangle} = \mathcal{F}_{L(a,x_1)} \cdot \mathcal{F}_{L(x_1,x_2)} \cdot \mathcal{F}_{L(x_2,x_3)} \cdots \mathcal{F}_{L(x_m,b)} \tag{17}$$

Review of Dijkstra's shortest path algorithm

It is suggested that you use Dijkstra's algorithm (implemented with a MIN-HEAP) to efficiently generate the uncertain paths needed for comparisons. The standard Dijkstra's algorithm uses both a Color-Label and a Distance-Label for each node. The colors are WHITE (for all nodes that have not yet been discovered), GRAY for all nodes that have been detected, but not been selected, and BLACK for nodes that have been selected. In a very condensed form, the algorithm is as follows:

algorithm `SimplifiedDijkstra( graph` $myGraph$`,` **node** a`,` **node** b`)`
////Color all nodes to WHITE and set all Distances to ∞

$MyMinheap \leftarrow \mathbf{minheap.create}()$
$Distance(a) \leftarrow 0$
$Color(a) \leftarrow$ GRAY (i.e. "in-the-heap")
$MyMinheap.\mathbf{insert}(a)$

while NOT $MyMinheap.\mathbf{IsEmpty}$ **do**
  do $Next \leftarrow MyMinheap.\mathbf{delete}()$ **until** $Color(Next) ==$ GRAY **end-do** /* Smallest distance GRAY node
  $Color(Next) \leftarrow$ BLACK
  if $Next == b$ **then RETURN end if**
  for every NON-BLACK $Node$ **adjacent to** $Next$ **do**
    // Try to RELAX the Distance
    if $Distance(Next) + Cost(Next, Node) < Distance(Node)$
      then do
      $Distance(Node) \leftarrow Distance(Next) + Cost(Next, Node)$
      $Color(node) \leftarrow$ GRAY (i.e. "in-the-heap")
      $MyMinheap.\mathbf{insert}(Node)$
      end do; endif
  end-for-every
end while
end algorithm `SimplifiedDijkstra`

It should be noted that the variables that are called "Distance for a node" are originally set to ∞, and they continue to decrease until they get their final value when the nodes are selected (and colored BLACK). For this project, the notion of Distance is no longer a simple positive real number, or a sequence of decreasing numbers, but rather a positive RV, or a sequence of RV's. The statement

if $Distance(Next) + Cost(Next, Node) < Distance(Node)$

needs to be replaced by

if $\mathsf{Y}_{P\langle a,x_1,x_2\cdots Next,b\,\rangle} + \mathsf{X}_{L(Next,Node)} < \mathsf{Y}_{P\langle a,x_1,x_2\cdots Next,Node\,\rangle}$

except that you cannot compare two RV's in this way. You can compare certain numerical characteristics of rv's however, and this project considers several of such characteristics.

Criteria to be used

The criteria to be used are:

Mean Value Pick the path whose total "expected value" is smallest from among all paths.

Optimist Pick the path such that it's "expected value - standard deviation" (as long as this is positive) is smallest from among all paths.

Pessimist Pick the path such that it's "expected value + standard deviation" is smallest from among all paths.

Double Pessimist Pick the path such that it's "expected value + 2 standard deviation" is smallest from among all paths.

Stable Pick the path such that it's "squared coefficient of variation" is smallest from among all paths.

We are sure you can come up with others (e.g. additional weights, or probabilities, or Laplace Transforms, or use Chebyshev's moment inequalities to find bounds, or .....). In fact, if your team has three members, you must add one additional criterion, and explain why it may make sense. Whichever additional strategy your team decides to adopt and adapt, you must add them to the 5 mentioned above and compare all 5 or more shortest paths from a to b.

Performance measures

Subsequently, you will then have to compare and analyze the paths as generated, For each of these paths, you are asked to determine the hop-count. Also, the path delay itself has an "expected value," "expected value - standard deviation," "expected value + standard deviation," "expected value + 2 standard deviation," and "squared coefficient of variation." You are asked to determine their values. You are also asked how these paths differ. Is there a link common to all paths? You must compare the paths according to at least the following characteristics. Present your result in a table, such as the one below. In the table, we use μ to indicate the path-mean, σ to indicate it's standard deviation. Also, "hops" indicates the hop-length.

	$\mu - \sigma$	μ	$\mu + \sigma$	$\mu + 2\sigma$	$c^2(\mathsf{Y})$	hops
Mean Value						
Optimist						
Pessimist						
Doubly Pess						
Stable						
add your own						

Finally, you are asked to check whether or not the strategies result in a different path. Alternately, for all links on any path, count how many paths use this link, and print five links that are shared with the largest number of paths. For instance,

	Mean Value	*Optimist*	*Pessimist*	*Doubly Pess*	*Stable*	*own*
edge (x,y)						
(..., ...)						
(..., ...)						
(..., ...)						

13.6 Least Congested Path

Please note that this is a former project. The description does not present a polished presentation and is intended to guide the student to explore and to discover the answer. The project is intended to be thought provoking and intended to be done in teams of 3 or 4 for full benefit of all. Multiple directions are described in the literature, these are all possible and all acceptable as long as the team justifies their choices. The process of deriving answers becomes the valued learning experience: out-of-the-box thinking and getting the satisfaction of the AHA-moment. Students gains a full understanding and appreciation of the problem, the algorithm, and it's application. Generally, students also appreciate the learning process itself and look back with pride at their accomplishments.

Was a project in CS5592, Fall Semester 2016

Roads, airspace, and computer and communication networks are often modeled as graphs. There are other examples in social networks and cloud computing, and so on. In such a network, nodes are the cities, or intersections, or server nodes (sources of demand, such as traffic or communication sites), and the links connect hem. If you want to go from one node to another (by plane, by car, or routing internet traffic or information packet), then you need a plan: source, destination, and a planned path in between. Let us explain this in terms of an road network between cities as the motivating application, but keep in mind that this project might have additional uses. When planning a car-trip, say from a to b, most people would want to find the shortest route, and there are several algorithms known that will help you find the shortest path. But please do not forget that you are not the only car on the road, and with more cars on the road will cause interference and extra delay because you share the road with more and more others. All cars are still interested in the shortest path, but shortest for a single car road is different from a shortest for a congested road. We will call such a paths a *CongestedPath*.

You will be given a number of different input scenario's. Each scenario will have

1. The size n of the system (the total number of cities),
2. an adjacency matrix $\mathbf{E}$ with edge-weights. The $(i,j)^{\text{th}}$ entry $\mathbf{E}[i,j]$ represents the traveling time on the edge between the two specific nodes, i and j, assuming that there are no other cars on the edge that might slow you down.

The matrix $\mathbf{E}$ is will allow you to compute the shortest path from a to b, and several algorithms could be used for this. The shortest path might take several edges (or links), and if you are the only one on the road, then this is the path you will take. However, you are not the only person/car on the network. You will also be given

3. a flow matrix $\mathbf{F}$, whose $(i,j)^{\text{th}}$ entry $\mathbf{F}[i,j]$ represents the number of (other) cars that travel on the entire path from node i to node j every hour.

For the sake of argument, say that there are $\mathbf{F}[21,47] = 115$ cars traveling form node v_{21} to node v_{47}, and that the edge (v_{34}, v_{38}) is on the shortest path from node v_{21} to node v_{47}, then there are also 115 cars/hour on the edge (v_{34}, v_{38}) due to the flow $\mathbf{F}[21,47]$. But is very well possible that the edge (v_{34}, v_{38}) is also on the shortest path from node v_{17} to node v_{29}, and that $\mathbf{F}[17,29] = 75$. This means that there are at least 190 cars/hour on the edge (v_{34}, v_{38}). Thus for each edge (i,j), you can calculate the actual carried traffic load from all source-destination pairs on that edge, because everybody takes their shortest path. Put all this information in a new matrix $\mathbf{L}$, each entry representing the total load on the edge. Next, you will be given

4. a capacity matrix $\mathbf{C}$, whose $(i,j)^{\text{th}}$ entry $\mathbf{C}[i,j]$ represents the maximum number of (other) cars that travel on the edge from node i to node j every hour.
5. the starting node a and the terminal node b

So now we have our first potential problem: we need to make sure that $\mathbf{L}[i,j] \leq \mathbf{C}[i,j]$ for all j and j. But even more important: The initial delay on edge (v_i, v_j) is $\mathbf{E}[i,j]$ was given under the assumption that there are no other cars on the edge. Now that we know there are $\mathbf{L}[i,j]$ other cars, we can adjust the delay over this edge. For the purpose of this project, we assume the congestion dependent travel times (weights) to be

$$\mathbf{G}[i,j] = \left(\frac{\mathbf{C}[i,j]+1}{\mathbf{C}[i,j]+1-\mathbf{L}[i,j]} \right) \mathbf{E}[i,j]$$

if $\mathbf{L}[i,j] \leq \mathbf{C}[i,j]$, and ∞ otherwise. So if the basis for computing the shortest paths was the "no-other-cars-edge-weights", then you derived the *absolute shortest path* and if you would travel the path from a to b, then you would experience, on each edge, the "actual congested-edge-weights" because of all the other cars. The combined path length in this case is the "*actual experienced path length*". Thus the actual experienced path length may not be close at all to the expected shortest path length after all, and a new shortest path needs be re-computed, or obtained in a different method. How to do this is not at all clear, and this is the objective of this project: bring the "*predicted*" total path length and the "*actual experienced*" path length closer together. This is a team project (each team has either 2 or 3 team members), and you need to fully develop, discuss, and evaluate all the options with your team members, and the decide which option to implement. There may not be a best answer, but there are several acceptable approaches. Just about all approaches are along the lines as: Come up with a way to adjust the weights based on some algorithm. Find the shortest paths from these adjusted weights (and call these shortest paths the "shortest predicted paths"), compute again the loading of each edge based on the adjusted weights, and determine the "actual path length" between a and b. Try to get the actual path length and the predicted path length within 10% of each

other.

Just to get your ideas flowing, you may want to consider the following possibilities:

a) Accept the "*absolute shortest path*" anyway, and suffer the consequences where most people in their cars are not happy at all by their experiences. They will all experience the congested path length.

b) Accept and Allocate "one car at one time" into the system, and iterate until all cars are accommodated.

c) Accept and Allocate "one car at one time for each s-t pair" into the system, and iterate until all cars are accommodated.

d) Accept and Allocate the full load for "one s-t pair at one time", and iterate with different pairs.

e) Accept and Allocate all s-t traffic, such that s and t are one hop away from each other, followed by all s-t pairs that are two hops away, and so on.

f) Iterate the system as a whole by allowing the elements $\mathbf{G}[i, j]$ to play the role of weights and re-calculate the actual delays.

This is only a short list of suggested approaches, I am sure you can come up with other. Whichever strategy your team decides to adopt and adapt, you must then implement the algorithms and calculate (and print):

a) Both shortest predicted path length and actual path length from a to b.

b) Both shortest predicted edge length and actual edge length from the first and last edge on the path from a to b.

c) The hop-count of the path between a and b.

This is however a course on algorithms, and you also have to report on the actual algorithms you used to actually find the information asked above. In particular,

e) If your team has x members, then your team must study at least x approaches. For each approach, develop an algorithmic description, present a worst case time complexity with a description of a situation where such a worst case might occur. Further, comment why this approach can be expected to produce quality result and on the ease with which the approach could be implemented. You may also use secondary metrics, such as ease of programming, ease of software maintenance, portability, and so on.

f) Next, compare and contrast the x possible approaches, list all the pro's and cons of each, and decide on which one to actually implement for networks where n is bounded above by $1,000$. Indicate whether or not you believe that there is a better (i.e. faster run-time) solution with an explanation on why you believe this to be the case or not to be the case.

g) There are several supporting algorithms your team needs to decide on, and you need to explain the reasoning for choosing the algorithms you implement for finding the shortest paths length, as well as actual path construction. Obvious potential choices are Dijkstra's SPA, Floyd-Marshall SPA and others.

h) Provide convincing arguments that your have implemented your algorithms correctly and efficiently with the appropriate data structure.

i) measure the CPU-usage of the programs when it solved each case,

j) the empirical validation that your programs have the correct asymptotic complexity.

k) For the cases that will be provided, present the final actual path length and the predicted path length for the given a and b.

l) Feel free to add cases yourself that illustrate certain algorithmic or program features

m) As a last item, reflect back on this project and on the design decisions you made. How did you handle any unforeseen situations and any 'newly discovered' problems? Now that you have done the project, what have you learned? If you have the opportunity to do this project again, perhaps with additional requirements, what would you do the same, and what would you do differently? What are the lasting lessons you have learned? What are the lessons for future students?

This is a group project: choose your team mates wisely making sure that the individual strengths complement one another. Each team has either 3 or 4 members, but all members are equally responsible for the success of the team, and all team members get the same grade for the project. The project grade is composed of both the project (and report) itself (80%), as well as a specially designed "Project quiz", which is to be taken individually, and the grade of all individual members are averaged and entered as 20% of the project grade for the team. Also, all members must be present when the project is presented.

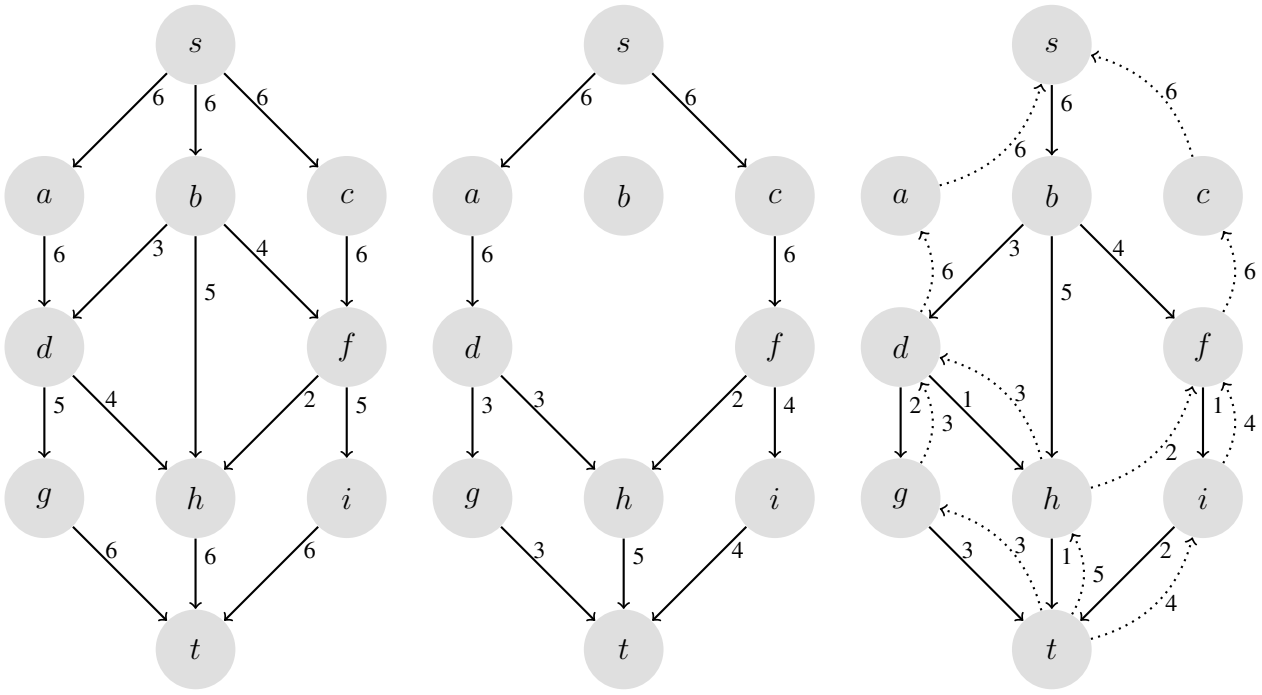
14 Ford-Fulkerson's Algorithm for Optimal Flow

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.

This is a great example of a graph algorithm that is essentially "self correcting": It allows decisions that themselves are counter productive, perhaps. But in subsequent steps these unwise decisions are undone. Such an attitude might not allows be best, but we should not always focus on preventing unwise decisions.

SE.74: The following three graphs show a partially completed working of Ford-Fulkerson's algorithm to determine the maximal flow from node s to node t. In particular, we show the original capacities, the current flow, and the residual capacity plus the reverse flow (also known as auxiliar).

CAPACITY GRAPH FLOW GRAPH RESIDUAL (SOLID) + +BACKWARDS (DOTTED)



Following the algorithm, you look for another shortest path in the right-most graph, labeled RESIDUAL (SOLID) + BACKWARDS (DOTTED). The shortest path includes both solid and dotted connections. For instance, a shortest path has been identified from $s \to b \to h \rightsquigarrow f \to i \to t$. The flow-graph can then be updated (improved) by adding the flows on the solid edges and subtracting the flows on the dotted edges.

14.1 Baseball Elimination

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.

For more explanation or deeper info, see you textbook or `cs.princeton.edu/~wayne/papers/baseball.pdf`

Problem Statement Given the "current standings" for a baseball fantasy league with 5 teams. Is it possible for team "E" to be undisputed winner of this league? That is, after all the games have been played, is it possible for team "E" to have more wins than any other team in the "final standings"?

Team	Wins	Games Left	Schedule A	B	C	D	E
A	78	15	-	3	4	3	5
B	80	14	3	-	4	4	3
C	79	16	4	4	-	5	3
D	82	14	3	4	5	-	2
E	77	13	5	3	3	2	-

Suppose team "E" wins all the games that it still needs to play. This would change the "current rankings" to "optimistic rankings" by adjusting the wins for team "E" and by adjusting the schedule between to other teams.

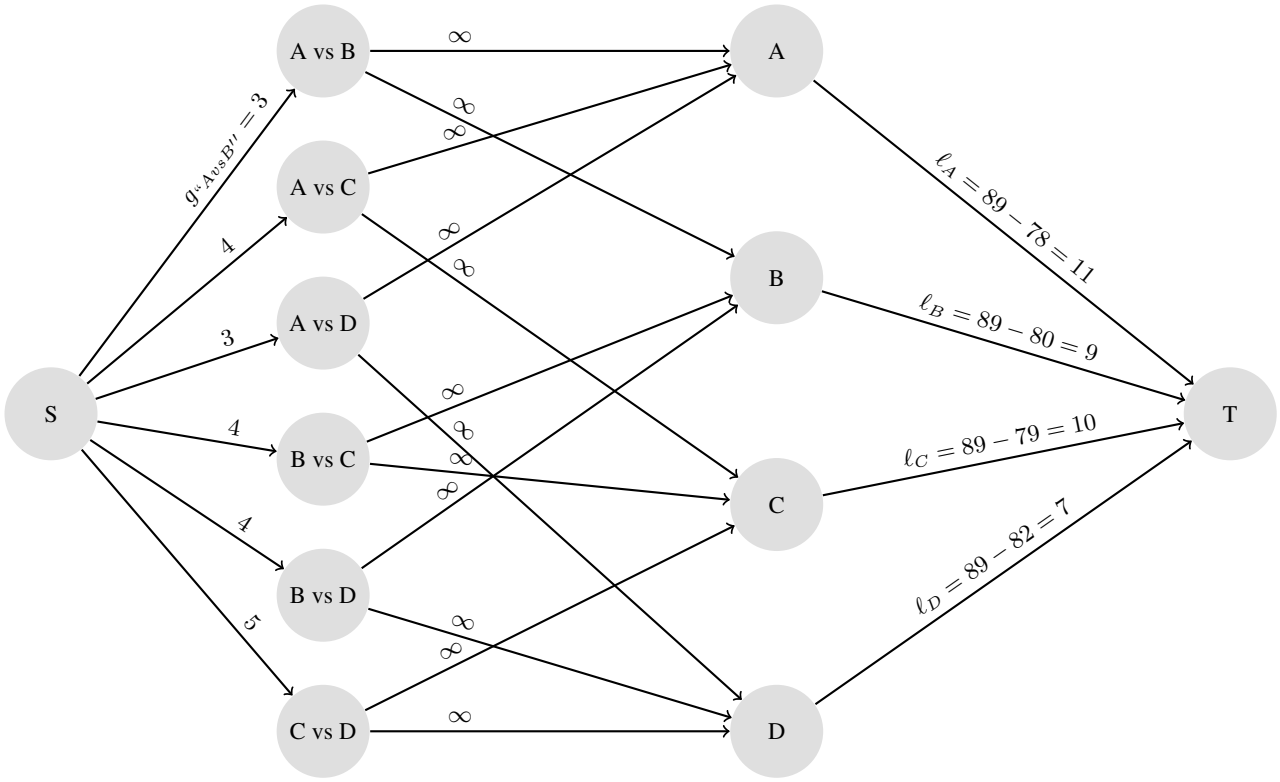
Team	Wins	Games Left	Schedule A	B	C	D
A	78	15-5	-	3	4	3
B	80	14-3	3	-	4	4
C	79	16-3	4	4	-	5
D	82	14-2	3	4	5	-
E	77+13					

So in these "optimistic rankings", team "E" has now accumulated $w_{\text{"E"}} = 90$ potential wins, while the other teams have 78, 80, 79 and 82 wins. Team "E" has no more games left to play in this "optimistic" situation, whereas the other teams still would need to play 23 games amongst each other. Thus, **if** Team "E" wins all it's remaining games, then they will be ranked higher than team "A", which can win at most $78 + 10 = 88$ games in total. But if team "D" wins only 8 of it's remaining 12 games, then team "D" will be ranked higher. So is it possible to find a configuration of potential outcomes of the 23 games between the 4 remaining teams, so that team "E" is the undisputed champion? The answer is "YES", if the wins by the other teams are distributed so that no other team has 90 or more wins. The number of wins for the other teams must be 89 or less for an undisputed championship for "E". Thus Team "A" must be limited to no more than $89 - 78 = 11$ additional wins (but they only play 10 more games anyway, so they are not a threat to team "E"). Team "B" has currently accumulated 80 wins, and has 11 more games to play against "A through D", so they can have up to 9 additional wins, without jeopardizing "E". So the original problem can now be refined:

Refined Problem Statement Given the "optimistic rankings" and the "remaining schedule" for a baseball fantasy league with 4 teams. Is it possible to distribute the wins/losses so that each team is limited to at most ℓ_{team} additional wins?

Team	Wins	Games Left	Schedule A	B	C	D	Win limit
A	78	10	-	3	4	3	$\ell_A = 89 - 78 = 11$
B	80	11	3	-	4	4	$\ell_B = 89 - 80 = 9$
C	79	13	4	4	-	5	$\ell_C = 89 - 79 = 10$
D	82	12	3	4	5	-	$\ell_D = 89 - 82 = 7$
E	90						

At this time, we can reduce the refined problem statement to a maximal flow problem statement. Construct a graph and introduce one node for each "series between any two teams", introduce one node for each team, and introduce a source S and a terminal node T. Now introduce weighted edges between S and the nodes for each series, where the weights are equal to the number of games left to be played between these teams. Next, introduce edges between the "series node" and the two "team" nodes, with unlimited capacity. Finally, introduce edges between the "team" nodes and terminal T, whose weight is equal to the maximal number of additional wins for the team. Consider the graph for our example.



So we have constructed an instance of a flow problem: **If** a maximal flow can be found in the graph we just constructed, and the maximal flow is equal to the number of all games that is left to be played between teams A, B, C, and D, **then** team E can still be the sole winner (i.e. is not eliminated from the top spot). Similarly, **if** the maximum flow is smaller then the number of all games that is left to be played between teams A, B, C, and D, **then** team E can no longer be the sole winner and is eliminated from the top spot.

The maximal flow can be determined e.g. by using the Ford-Fulkerson algorithm as discussed in class and shown in the book. Also, if you would like to allow "ties" at the top location, and thus a need for an extra tie-breaker game, then change the "89" in the above example to "90". In case you are wondering: Team "E" can still win outright.

Suggested Homework: Note that this graph and instance of a max-flow problem was formulated specifically for team E. If you want to know whether or mot team A has been eliminated from the top spot, then another instance (still for the max-flow problem) must be created. Using exactly the same numbers as given above. Create a graph, and introducing edge weights so that Ford-Fulkerson's maximal flow algorithm can be used (no need to do this part here) to determine whether of not team "A" has been eliminated. Construct a graph, identify the nodes and edges with correct edge weights, and explain what they represent.

15 Reliable Networks

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.

Suppose you have been given a directed graph $G = \langle V, E\rangle$ that represents a connection network where the connection between two nodes a and b may not be reliable. For instance, because of road conditions, link noise, or whatever the physical application. Thus each link, say link $\langle a, b\rangle$ has an associated probability that a message send at node a arrives correctly at node b. Each link has thus a Bernoulli random variable with a certain success probability that we call $\mathbf{P}[a, b]$. Each link has such a probability, and we place them in a table or matrix. Note, this is a matrix with individual link probabilities, but we should be very careful with it's interpretations. We are assuming for now that all links are failing independently of one another, so each matrix element is independent from all the others. It is just a weighted adjacency matrix, where he weights happen to be probabilities. It is NOT a probability transition matrix that you may have seen in relation to Markov Chains (or it's applications). In particular, the row-sums are not necessarily one, or even less than one. Each separate element simply represents the "all or nothing" probability that the link is available for the duration, and that the message arrives correctly at the other node. The matrix notation is however very useful in studying the reliability of a path. In particular, say you have a path from a to b that uses m intermediate nodes: $Path\langle a, x_1, x_2 \cdots x_m, b\,\rangle$. Then the probability that a message that is send at node a arrives correctly at b, and where it uses that particular path, is (remember that links are independent):

$$\Pr\,[\,Path\langle a, x_1, x_2 \cdots x_m, b\,\rangle\,] \quad = \quad \mathbf{P}[a, x_1] \cdot \mathbf{P}[x_1, x_2] \cdot \mathbf{P}[x_2, x_3] \cdots \mathbf{P}[x_m, b] \tag{18}$$

So the probability of a correctly using a particular *path* is composed as the products of individual matrix elements of the *link* probability matrix. This interpretation is common in graph algorithms, a little less common in applied probability. But indeed, standard matrix formulation can be adjusted for this particular application. For instance, the most reliable 2-link path from a to b is

$$\mathbf{P}^2[a, b] = \max_k\{\mathbf{P}[a, k] \cdot \mathbf{P}[k, b]\} \tag{19}$$

Continuing this approach, the most reliable 3-link path between a and b is

$$\mathbf{P}^3[a, b] = \max_k\{\mathbf{P}[a, k] \cdot \mathbf{P}^2[k, b]\} \tag{20}$$

and so on. Continuing along these lines, the most reliable paths that is of length $n - 1$ (the maximal length in a network of n nodes), is:

$$\mathbf{P}^{n-1}[a, b] = \max_k\{\mathbf{P}[a, k] \cdot \mathbf{P}^{n-2}[k, b]\} \tag{21}$$

Finally, the most reliable path from a to b that uses any number of intermediate nodes, is

$$\max_{\ell=0,1,\ldots n-2}\{\mathbf{P}^\ell[a, b]\} \tag{22}$$

(The matrix "thinking" is indeed appropriate, as long as "addition" operator is is overloaded/replaced by a $\max$ operation.) So: Given a weighted graph where the weights are interpreted as probabilities as indicated above, and is we are tasked with finding the most reliable connection, then this sounds like a dynamic programming exercise. And indeed it is. Dijkstra's SPA and the Floyd-Marshall APSP algorithm can both be adapted directly, more on that is below. The proofs are somewhat harder, and for that reason we use a technique called "problem reduction", from the most reliable path problem to the shortest path problem. When using this, we assume that an instance of the most reliable path problem is given. That is, we assume that a graph $G = (V, E)$ is given together with edge-labels, the weights, $\mathbf{P}[i, j]$ for all edges $\langle i, j\rangle \in E$. The problem is to find a most reliable path between two given nodes S and T. We could use shortcuts, but we will follow the formal approach to demonstrate the "problem reduction" technique: We create an instance of a Single Source Single Destination Shortest Path Problem, whose solution will allow the construction of the most reliable path. We do so, by creating a new weighted graph G' and with edge lengths that are inspired by the edge reliabilities, such that when a shortest path is found in graph G', then a most reliable path can be constructed in the original graph G. We start with motivating the direction.

Shortest path problems ask for the *minimal* "sum-of-edge-weights", whereas most reliable paths ask for a *maximal* "product-of-edge-probabilities". These seem quite different. Now consider two different paths from a to b: $Path\langle a, x_1, x_2 \cdots x_m, b\,\rangle$ and $Path\langle a, y_1, y_2 \cdots y_r, b\,\rangle$. The path over the x-nodes is preferred if

$$\mathbf{P}[a, x_1] \cdot \mathbf{P}[x_1, x_2] \cdot \mathbf{P}[x_2, x_3] \cdots \mathbf{P}[x_m, b] > \mathbf{P}[a, y_1] \cdot \mathbf{P}[y_1, y_2] \cdot \mathbf{P}[y_2, y_3] \cdots \mathbf{P}[y_r, b] \tag{23}$$

Taking the logarithm on both sides (and remembering that $\log(x_1\ x_2) = \log x_1 + \log x_2$, the "log of a product" is mathematically equal to the "sum of log's "), we have that the path over the x-nodes is preferred if

$$\log\,(\mathbf{P}[a, x_1]) + \log\,(\mathbf{P}[x_1, x_2]\,) + \ldots + \log\,(\mathbf{P}[x_m, b]\,) \quad > \quad \log\,(\mathbf{P}[a, y_1]) + \log\,(\mathbf{P}[y_1, y_2]\,) + \ldots + \log\,(\mathbf{P}[y_r, b]\,) \tag{24}$$

Now notice that probabilities are between 0 and 1, and the log's of these probabilities are negative. Thus the path over the x-nodes is preferred if the sum of the associated negative numbers is larger than that for the y-nodes. Now multiply both equations by negative one (-1), then each term becomes positive, and the $<$ inequality has flipped, Thus the path over the x-nodes is preferred if the sum of the associated numbers (formerly negative, now positive) is smaller (formerly larger) than that for the y-nodes.

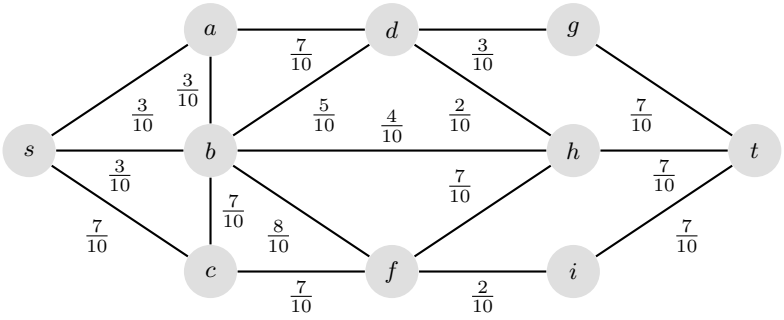
We now ready to construct a new graph, $G' = (V', E')$, and edge weights that are inspired by the reliability instance: Let $V' \leftarrow V$ and $E' \leftarrow E$, and define edge-weights $w(i,j) = -\log \mathbf{P}[i,j]$ for all edges $\langle i,j \rangle \in E'$. (Take care of edges for which $\mathbf{P}[i,j] = 0$ or $\mathbf{P}[i,j] = 1$) If the path $\langle a, x_1, x_2 \cdots x_m, b \rangle$ is shorter than the path $\langle a, y_1, y_2 \cdots y_r, b \rangle$ in the induced graph G', then also the path $\langle a, x_1, x_2 \cdots x_m, b \rangle$ is more reliable than the path $\langle a, y_1, y_2 \cdots y_r, b \rangle$ in the original graph G. Furthermore, if the path $\langle a, x_1, x_2 \cdots x_m, b \rangle$ is the shortest from among all paths from a to b in the induced graph G', than also that same path $\langle a, x_1, x_2 \cdots x_m, b \rangle$ is more reliable than any other paths from a to b in the original graph G.

So we have formally "reduced" the most reliable path problem to the shortest path problem in an induced graph. There are several algorithms to solve shortest paths problems, such as Dijkstra's and Floyd-Warshal, and many variants thereof. (We left a few minor issues to be solved by you: Does it work for both undirected and directed graphs? What is the appropriate bases for these logarithms?).

RELIABLE CONNECTION NETWORK – EXAMPLE

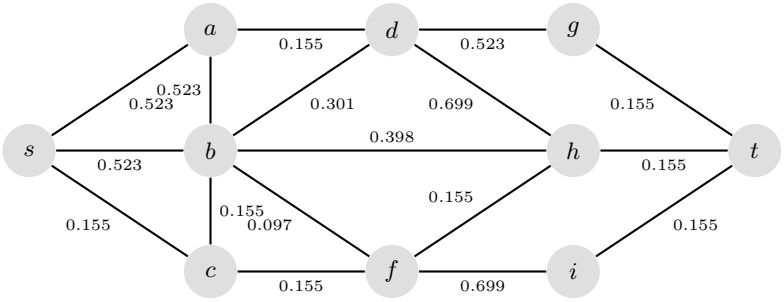
Consider for example the following reliability graph. Fortunately, each probability is a fraction with a common denominator (10), so use this value as base of the logarithm. Thus the probability of $\frac{7}{10}$ induces the weight $w = -\log_{10}(\frac{7}{10}) = 1 - \log_{10} 7 = 0.155$. Similarly, $-\log_{10}(\frac{2}{10}) = 0.699$, $-\log_{10}(\frac{3}{10}) = 0.523$, $-\log_{10}(\frac{8}{10}) = 0.097$, and so on.

RELIABLE CONNECTIONS – 1



The induced graph is as follows:

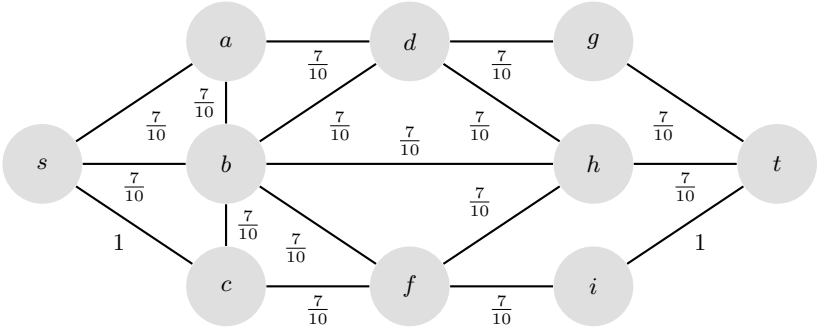
INDUCED WEIGHTS FOR RELIABLE CONNECTIONS – 1



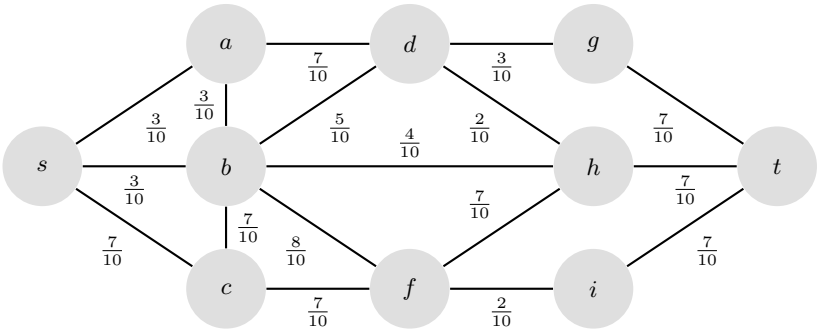
15.1 Reliable Networks: Suggested Exercises

(Graduate students only)

SE.75: Complete the computations for the graph below: what is the maximally reliable connection between s and t?



SE.76: Complete the computations for the graph below: what is the maximally reliable connection between s and t?



SE.77: Can Floyd-Warshall's algorithm be used, perhaps, to determine the "all-source-all-destination" most reliable connections? Number the nodes $1, \ldots n$ and Let $R^{(k)}$ be the table so that the (a, b)-th element of this table represents the most reliable path to communicate from node a to node b, while only allowing nodes $1, 2, \cdots k$ to be intermediary nodes on such a path. The initial values are given by $R^{(0)}$, the given weighted adjacency matrix $\mathbf{P}$. The driving equation is

$$R^{(k)}(a,\ b) = \max_{\text{2 options}} \left\{ R^{(k-1)}(a,\ b)\ ,\ R^{(k-1)}(a,\ k) \cdot R^{(k-1)}(k,\ b) \right\}$$

An actual algorithm takes just a few lines, and runs in n^3 time complexity (and uses only a single table R of size n^2, which is the space complexity.)

SE.78: Reliable Core How would you define the notion of a most reliable spanning tree? Be careful, it is not as straightforward as you might think. Now, how would you go about finding such a most reliable spanning tree? Can Prim's and/or Kruskal's algorithms be adapted to determine a most reliable spanning tree?

15.2 Reliability versus Distance Weights

The text above shows an almost trivial relation between the shortest path and the reliablist path (I just invented a new word). So when faced with a reliablist path problem, you have three practical choices:

1. Alter the input and use either Dijkstra or Floyd-Warshal from the library. (i.e. create the weights as indicted above, which might introduce numerical errors and solve the problem that $\lg 0$ is not a number.
2. Rewrite Dijkstra or Floyd-Warschal altogether (in fact, replace $\min$ by $\max$ and "sum" by "product"). Indeed, you need to change only two or three lines of code in the Basic Dijkstra algorithm.
3. Use the standard Dijkstra or Floyd-Warshal, but simply overload the operators (if you are in an object-oriented environment)

SE.79: List all pro's and con's of the three approaches. Make sure you address performance, easy of coding, maintenance, transparency, readability, testability, and other modern ways to compare two approaches to solve the same problem.