CSE 3101 Design and Analysis of Algorithms
Solutions for Practice Test for Unit 3
Graph Search and Network Flow

Jeff Edmonds

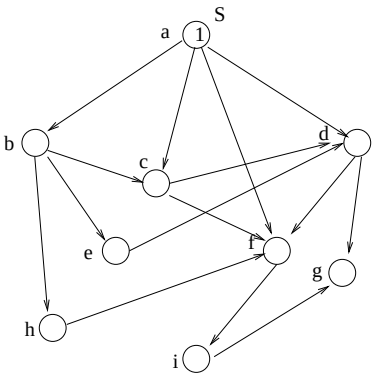
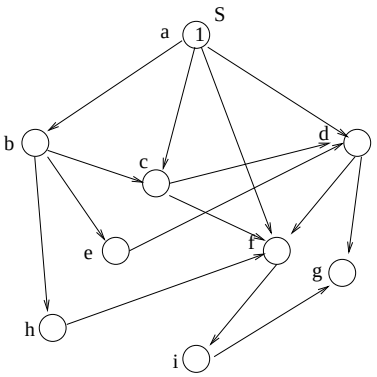
1. Trace BFS and DFS.

   (a) What are the generic loop invariants used for searching a graph starting from some node s?

   (b) What are the extra loop invariants used for Breadth First Search.

   (c) What are the extra loop invariants used for Depth First Search?

   (d) For each do the following:

Breadth First Search (BFS)	Depth First Search (DFS)
	
Data Structure?	Data Structure?

   i. Start at node s and when there is a choice follow edges from left to right. Number the nodes $1, 2, 3, \ldots$ in the order that they are found, starting with Node $s = 1$.

   ii. Darken the edges of the Tree specified by the predecessor array π.

   iii. What is the data structure used by BFS/DFS to store nodes that are found but not yet handled?

   iv. Circle the nodes that are in this data structure when Node 8 is first found.

- Answer:

   (a) The generic loop invariant is that nodes are considered to be either *not found, found and not handled, or found and handled* and each node has a *parent* node $\pi(v)$.

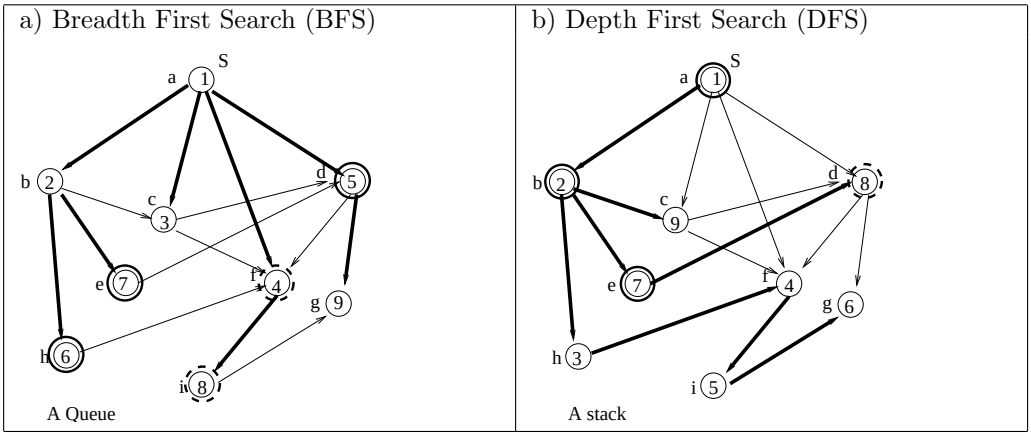
   Found: If a node v is considered *found*, then we have trace a path from s to the node. This path can be traced backwards using $v, \pi(v), \pi(\pi(v)), \pi(\pi(\pi(v))), \ldots, s$.

   Handled: If a node v is considered *handled*, then all of its neighbors have been found.

   (b) The extra loop invariants used for Breadth First Search is that so far the nodes have been found in the order of their distance from s.

   (c) The extra loop invariants used for Depth First Search is that the nodes in the stack form a path starting at s going to the current node u being considered.

   (d)



2. DFS

   (a) What are the pre and post conditions for the Recursive Depth First Search algorithm?

   (b) Trace out the iterative and the recursive DFS algorithms on the same graph and see how they compare. Do they have the same running time?

   - Answer:

     (a) The preconditions for the Recursive Depth First Search algorithm is that a graph is given (directed or undirected) with some of the nodes marked *found* using bread crumbs and some node s' is specified. The post condition is that every node reachable from s' without visiting a node marked *found* is found.

     (b) They are identical. In fact, the stack used by the iterative algorithm parallels the stack used for recursion.

3. What are the extra loop invariants used for Depth First Search? How is the extra loop invariant for Depth First Search used to look for cycles? If the graph has a cycle, is the cycle guaranteed to be found in this way?

   - Answer: The extra loop invariants used for Depth First Search is that the nodes in the stack form a path starting at s going to the current node u being considered. If there is an edge from u to some node v in the stack, then this forms a cycle from u, along this edge to v, and down the path in the stack back to u. If you have a cycle in the graph in mind, then this particular cycle might not be found, but some cycle (that is a superset of the one in mind) will be found. Let v be the first node in the cycle that you have in mind that is put on the stack. Let u' and u the neighbors of v in this cycle such that u' is put on the stack before u. Because u has is reachable form u' visiting nodes that have not yet been found, u will be put on the stack before v and u' are removed. Before u is removed from the stack the edge from u to v will be followed. Because v is still on the stack, this allows the algorithm to detect a cycle.

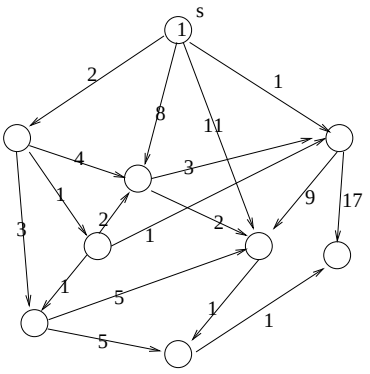
4. Give a Total Order (Topological Sort) of the DAG in the previous question using the letters a,b,....

   - Answer: DFS is run on the graph starting from a "random" node (in this case s). As nodes are handled and popped from the stack, they are outputted in reverse order giving a,b,c,e,d,h,f,i,g.

5. Dijkstra's Shortest-Weighted Paths Algorithm

   (a) What are the extra loop invariants used for this algorithm?

(b) On the following example, do the following.



i. Start at node s and when there is a choice follow edges from left to right. Number the nodes $1, 2, 3, \ldots$ in the order that they are handled, starting with Node $s = 1$.

ii. Darken the edges of the Tree specified by the predecessor array π.

iii. What is the data structure used to store nodes that are found but not yet handled?

iv. For each node, give the list of every d value that it has at various points during the computation.

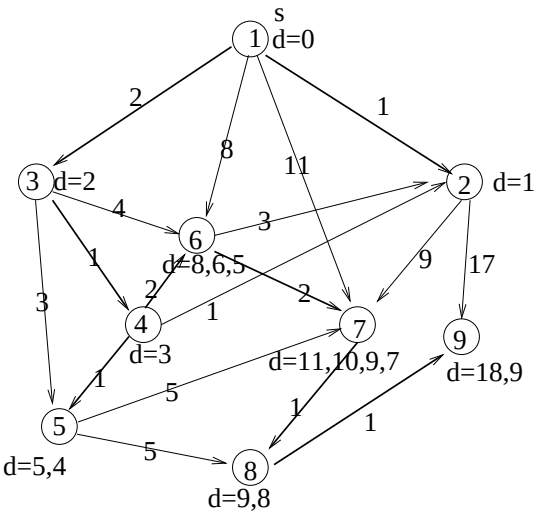
- Answer:

(a) The extra loop invariants are

Handled: For each handled node v, the values of $d(v)$ and $\pi(v)$ give the shortest length and a shortest path from s (and this path contain only handled nodes).

Found: For each of the unhandled nodes v, the values of $d(v)$ and $\pi(v)$ give the shortest length and path from among those paths that have been *handled*. We say that a path has been handled if it contains only handled edges. Such paths start at s, visit as any number of handled nodes, and then follow one last edge to a node that may or may not be handled.

(b)



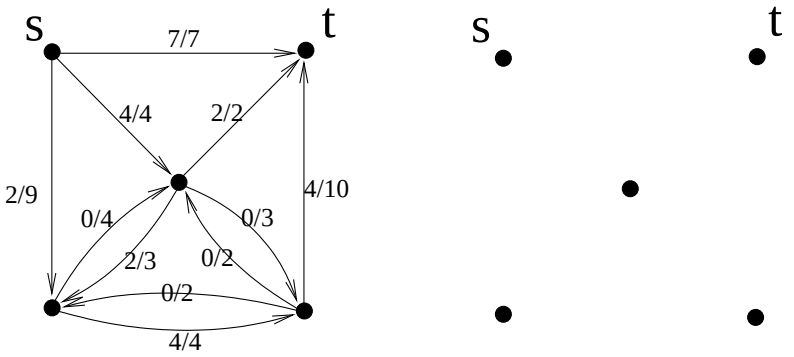
A Priority Queue

6. Given a graph where each edge weight is one, compare and contrast the computation of the BFS shortest paths algorithm from Section **??** and that of this Dijkstra shortest-weighted paths algorithm. How do their choices of the next node to handle and their loop invariants compare?

- Answer: Despite differences in the algorithms, on a graph with edge weights one, the BFS and Dijkstra algorithms are identical. BFS handles the first node in its queue while Dijkstra handles the node with the next smallest $d(v)$. However, BFS's third loop invariant assures that the nodes are found and added to the queue in the order of distance $d(v)$. Hence, handling the next in the queue amounts to handling the next smallest $d(v)$. BFS's first loop invariant states that the correct minimal distance $d(v)$ to v is obtained when the node v is first found, while with Dijkstra

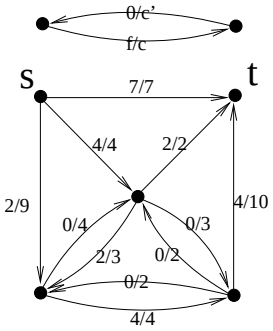
we are not sure to have it until the node is handled. However, with edge weights one, when v is first found in Dijkstra's algorithm, $d(v)$ is set to the length of the over shortest path and never changed again.

7. Starting with the flow given below, complete the network flow algorithm and prove the resulting flow is optimum.

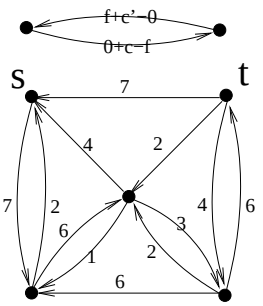


- Answer:

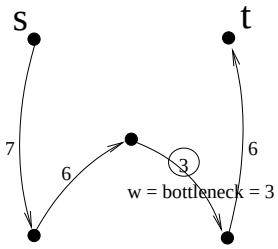
Given Flow



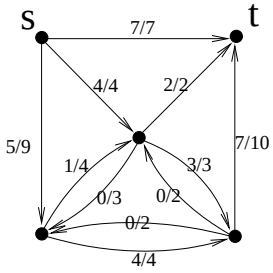
Augmenting Graph



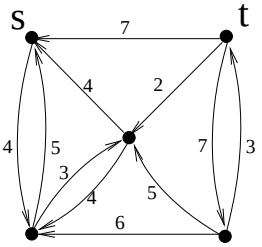
Augmenting Path



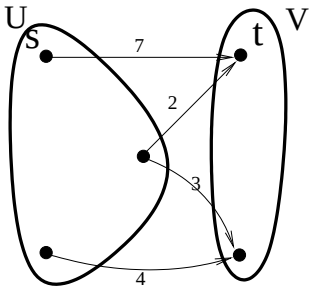
New Flow



Augmenting Graph



Cut



Add w when path passes forward through edge
Subtract w when passing backwards
Increasing total flow from s to t
while possibly decreasing the flow in some edges
and keeping that flow into each node equals flow out

Value of Flow = Flow out of s (minus that returning) = 7+4+5=16

The set U consists of the nodes reachable
from s in the last augmentation graph.
Note that because t is not reachable, it is not in U.
The edges included in the figure are those
from the original graph that cross from U to V.
Capacity of Cut = capacity of these edge crossing from U to V
= 7+2+3+4 = 16
Note that these edges are all at capacity in the flow.
Any edges going from V to U have no flow in them.
Hence, the flow across the cut,
which equals the flow out of s in the final Flow
equals the capacity of the cut.

The rate of this final flow is 16.

The cut (U, V) returned by the algorithm has capacity 16.

This flow witnesses that fact that a flow with rate 16 is obtainable.
This cut witnesses that fact that no flow can be bigger than 16.
Hence, the rate of the max-flow is 16, giving that this flow is optimum.

This cut witnesses that fact that a cut with capacity 16 is obtainable.
This flow witnesses that fact that no cut can be smaller than 16.
Hence, capacity of the min-cut is 16, giving that this cut is also optimum.

8. In hill climbing algorithms there are steps that make lots of progress and those that make very little progress. For example, the first iteration on the input given in Figure 1 might find add a flow of c along the path s, a, t and then during the second iteration add a flow of c along the path s, b, t. It might, however, find the path s, b, a, t through which only a flow of 1 can be added. How bad might the running time be when the computation is unlucky enough to always take the worst legal step allowed by the algorithm? Start by taking the step that increases the flow by 2 through the input given in Figure 1. Then continue to take the worst possible step. You could draw out each and every step, but it is better to use this opportunity to use loop invariants. What does the flow look like after $2i$ iterations?

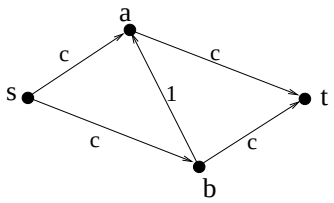


Figure 1: A network with a capacity of c on four of the edges and one on the cross edge.

(a) What is the worst case number of iteration of this network flow algorithm as a function of the capacity c?

(b) What is the worst case number of iteration of this network flow algorithm as a function of the number of edges m in the input network?

(c) What is the official "size" of a network?

(d) What is the worst case number of iteration of this network flow algorithm as a function of the "size" of the input network.

- Answer:

  (a) The loop invariant is that after $2i$ iterations there will be a flow of i in each of the four edges forming the diamond and a flow of zero in the cross edge $\langle b, a\rangle$. In the next iteration, you put a flow of one along the path s, b, a, t. Note this puts a flow of one in edge $\langle b, a\rangle$ which is its capacity. In the iteration after that you put a flow of one along the path s, a, b, t. Note this removes the flow of one in edge $\langle b, a\rangle$. This meets the loop invariant, that after $2i + 2$ iterations there will be a flow of $i + 1$ in each of the four edges forming the diamond and a flow of zero in the cross edge $\langle b, a\rangle$. With a capacity of c on each of the four edges forming the diamond, it will take $2c$ such iterations to finish the algorithm.

  (b) Note if we change the capacity c but keep the total number of edges fixed, then the worst case number of iteration of this network flow algorithm grows. Hence, number of iterations is *unbounded* as a function of the number of edges m in the input network.

  (c) The official "size" of a network would be the number of bits needed to write down the connections of these constant number of edges and their capacities c. This takes $size = 4 \log c + \Theta(1)$ bits.

  (d) The number of iterations is $2c = 2^{\Theta(size)}$. This is exponential time.

9. Give an algorithm for solving the *Min Cut Problem*. Given a network $\langle G, s, t\rangle$ with capacities on the edges, find a minimum cut $C = \langle U, V\rangle$ where $s \in U$ and $t \in V$. The cost of cut its capacity $cap(C) = \sum_{u\in U} \sum_{v\in V} c_{\langle u,v\rangle}$. Prove that it gives the correct answer. Hint, you have already been told how to do this.

   - Answer: Given a network $\langle G, s, t\rangle$, run the max flow algorithm on it. In addition to returning a maxiflow, it also returns a cut whose capacity C is equal to the value of the flow.

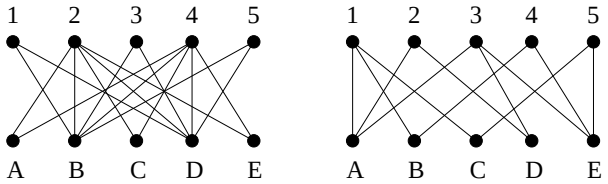
     This cut witnesses that fact that a cut with capacity C is obtainable.
     This flow witnesses that fact that no cut can be smaller than C.
     Hence, capacity of the min-cut is C, giving that this cut is optimum.

10. Express the network flow instance in Figure 1 as a linear program.

    - Answer: The variables in the linear programming instance are the flows $F_{\langle u,v\rangle}$ through each edge. The objective function to maximize is the $rate(F) = \sum_v \left[F_{\langle s,v\rangle} - F_{\langle v,s\rangle}\right]$. For each edge $\langle u, v\rangle$ there is the edge capacity linear constraint $F_{\langle u,v\rangle} \leq c_{\langle u,v\rangle}$. For each node $u \notin \{s, t\}$, there is the no-leaks linear constraint $\sum_v F_{\langle v,u\rangle} = \sum_v F_{\langle u,v\rangle}$.

11. (Too hard for a test.) Let $G = (L \cup R, E)$ be a bipartite graph with nodes L on the left and R on the right. A matching is a subset of the edges so that each node appears at most once. For any $A \subseteq L$, let $N(A)$ be the neighborhood set of A, namely $N(A) = \{v \in R \mid \exists u \in A \text{ such that } (u, v) \in E\}$. Prove Hall's Theorem which states that there exists a matching in which every node in L is matched if and only iff $\forall A \subseteq L,\ |A| \leq |N(A)|$.

    (a) For each of the following two bipartite graphs, either give a short witness to the fact that it has a perfect matching or to the fact that it does not. Use Hall's Theorem in your explanation as to why a graph does not have a matching. No need to mention flows or cuts.



    (b) $\Rightarrow$: Suppose there exists a matching in which every node in L is matched. For $u \in L$, let $M(u) \in R$ specify one such matching. Prove that $\forall A \subseteq L,\ |A| \leq |N(A)|$.

    (c) Look at both the slides and section 19.5 of the notes. It describes a network with nodes $\{s\} \cup L \cup R \cup \{t\}$ with a directed edge from s to each node in L, the edges E from L to R in the bipartite graph directed from L to R, and a directed edge from each node in R to t. The notes gives each edge capacity 1. However, The edges $\langle u, v\rangle$ across the bipartite graph could just as well be given capacity ∞.

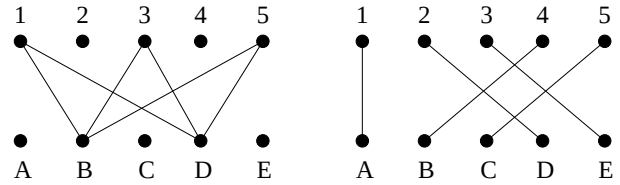
    Consider some cut (U, V) in this network. Note U contains s, some nodes of L, and some nodes of R, while V contains the remaining nodes of L, the remaining nodes of R, and t. Assume that $\forall A \subseteq L,\ |A| \leq |N(A)|$. Prove that the capacity of this cut, i.e. $cap(U, V) = \sum_{u\in U} \sum_{v\in V} c_{\langle u,v\rangle}$, is at least $|L|$.

    (d) $\Leftarrow$: Assume that $\forall A \subseteq L,\ |A| \leq |N(A)|$ is true. Prove that there exists a matching in which every node in L is matched. Hint: Use everything you know about Network Flows.

    (e) Suppose that there is some integer $k \geq 1$ such that every node in L has degree at least k and every node in R has degree at most k. Prove that there exists a matching in which every node in L is matched.

    - Answer:

(a) The first is does not have a matching. A witness is the fact that the nodes 1, 3, and 5 are only connected to B and D. Hence, these three can't be matched to these two. In the language of Hall's theorem, let $A = \{1, 3, 5\}$, then $N(A) = \{B, D\}$. Because $|A| > |N(A)|$, Hall's theorem gives that there is not a matching. The second does have a matching. A witness is the following matching.



(b) Consider an arbitrary $A \subseteq L$. Note that the set $B = \{M(u) \mid u \in A\}$ contains $|A|$ distinct nodes and that $B \subseteq N(A)$. Hence, $|A| \leq |N(A)|$.

(c) Let $A = U \cap L$ be set nodes that are both on the left side of the bipartite graph and on the left side of the cut. Consider any node $v \in N(A)$. Because $v \in N(A)$, there is a node $u \in A \subset U$ such that $\langle u, v \rangle$ is an edge. If $v \in V$ then this edge (u, v) crosses the cut. But this edge has capacity ∞. In this case, the capacity of the cut is well over $|L|$. On the other hand if $v \in U$ then the edge from v to t is across the cut. Now consider any node $u \in L - A \subset V$. The edge from s to u crosses the cut. This proves that the number of edges across the cut is at least $N(A) + (|L| - |A|)$, which by our assumption is at least $|A| + (|L| - |A|) = |L|$.

(d) We have seen that there is a matching with $|L|$ edges iff the max flow in this graph has value $|L|$ iff the min cut in this graph has capacity $|L|$. The cut with s on one side by itself has $|L|$ edges going across the cut, namely those edges from s to L. By the last question, if $\forall A \subseteq L,\ |A| \leq |N(A)|$ then every cut has at least $|L|$ edges across it. Hence, the min cut must be $|L|$. Hence, the max flow has value $|L|$. Hence, there is a matching with $|L|$ edges. All the nodes in L must be matched.

(e) By the last question, it is sufficient to prove that $\forall A \subseteq L,\ |A| \leq |N(A)|$ is true. Consider some set $A \subseteq L$. Because each every node in $A \subseteq L$ has degree at least k, we know that at least $k \cdot |A|$ edges leave A. All of the edges that leave A must enter its neighborhood set $N(A)$. Hence, the number that leave A is at most the number that enter $N(A)$. Because every node in $N(A) \subseteq R$ has degree at most k, we know that at most $k \cdot |N(A)|$ enter $N(A)$. Combining these gives It follows that $k \cdot |A| \leq \#$ leave $A \leq \#$ enter $N(A) \leq k \cdot |N(A)|$. Hence, $|A| \leq |N(A)|$ as is needed.