CSE 101: ALGORITHMS -- General notes

Syllabus
Graph algorithms: strongly connected components, shortest paths, minimum spanning trees
Divide-and-conquer strategies
Dynamic programming
Linear programming and network flows
NP-completeness and ways to cope with it

Course materials
1. Required text (available at bookstore): Dasgupta-Papadimitriou-Vazirani
2. Recommended text for background material: Neapolitan and Naimipour (on reserve at S&E library)

Discussion sections
You are welcome to attend more than one section per week
William Matthews
Wed 12-1 in HSS 2150
Wed 1-2 in HSS 2321
Yatharth Saraf
Wed 11-12 in HSS 1315
Wed 2-3 in HSS 2150

Examinations
Midterm 1: Jan 30
Midterm 2: Feb 27
Final: March 20, from 7p to 10p, in Center 109

Homework policy
Collaboration:
A good way to understand the course material is to discuss it with your peers.
You are welcome to collaborate in small groups on the homework assignments, but *anything you turn in must be in your own words*.
Lateness:
Homeworks should be turned in at the beginning (first five minutes) of class, on the due date.
No late homeworks will be accepted.
Your lowest homework score will be dropped.

Grading
Homeworks: 20%
Midterms: 20% each
Final: 40%

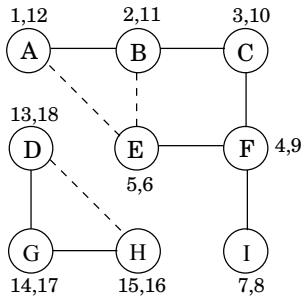
CSE 101: ALGORITHMS -- Reading

Date	*Topic*	*Reading*
Jan 9	Analyzing algorithms	Chapter 0
Jan 11	Graphs; undirected depth-first search	Chapter 3.1, 3.2
Jan 16	DFS in directed graphs; DAGs	Chapter 3.3
Jan 18	Strongly connected components	Chapter 3.4
Jan 23	Breadth-first search	Chapter 4.1, 4.2, 4.3
Jan 25	Dijkstra's algorithm	Chapter 4.4, 4.5
Feb 1	Bellman-Ford algorithm, shortest paths in dags	Chapter 4.6, 4.7
Feb 6	Properties of trees	Chapter 5.1
Feb 8	Minimum spanning trees	Chapter 5.1
Feb 13	Divide-and-conquer algorithms	Chapter 2 (omit 2.6)
Feb 15	Divide-and-conquer algorithms	Chapter 2 (omit 2.6)
Feb 20	Dynamic programming	Chapter 6
Feb 22	Dynamic programming	Chapter 6
Mar 1	Flows in networks	Chapter 7.2
Mar 6	Flows and matchings	Chapter 7.2, 7.3
Mar 8	Linear programming	Chapter 7.1
Mar 13	Intractability	Chapter 8
Mar 15	Intractability	Chapter 8

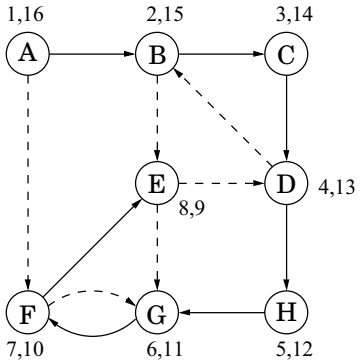
Solutions to Homework Two

CSE 101

3.1. In the figure below, `pre` and `post` numbers are shown for each vertex. Tree edges are solid, back edges are dashed.

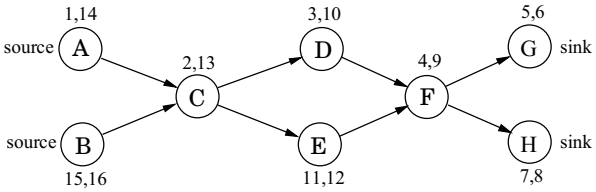


3.2. (a)

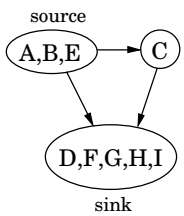


Solid edges are tree edges. Among the dotted edges, (A, F) and (B, E) are forward edges while the rest are back edges. There are no cross edges.

3.3. In the figure below, the algorithm finds the ordering: B, A, C, E, D, F, H, G. There are 8 possible orderings.



3.4. (ii)



The SCCs are found in the order: $\{D, F, G, H, I\}, \{C\}, \{A, B, E\}$. The additional of a single edge (such as $D \rightarrow A$) will make the graph strongly connected.

3.11. *Design a linear-time algorithm which, given an undirected graph G and a particular edge e in it, determines whether G has a cycle containing e.*

- Given: undirected graph G and edge $e = \{u, v\}$ in it
- Run `explore`$(G - e, u)$. If node v gets visited, return `yes`, otherwise return `no`.

Justification: G has a cycle containing edge $e = \{u, v\}$ if and only if $G - e$ (G without edge e) has a path from u to v. Removing edge e from G takes linear time: simply edit the adjacency list of nodes u, v. The rest of the algorithm runs in linear time because DFS does.

3.13. (a) *Prove that in any connected undirected graph $G = (V, E)$ there is a vertex $v \in V$ whose removal leaves G connected.*

Let G be any connected undirected graph. Run DFS on G, and consider any leaf node u in the resulting search tree T. Removing u from T does not disconnect the tree; all remaining nodes are still connected to the root of the tree and thus to each other. Since T is a subgraph of G, removing u does not disconnect G either.

(b) *Give an example of a strongly connected directed graph $G = (V, E)$ such that, for every $v \in V$, removing v from G leaves a directed graph that is not strongly connected.*

Let G be a directed cycle.

(c) *In an undirected graph with 2 connected components it is always possible to make the graph connected by adding only one edge. Give an example of a directed graph with two strongly connected components such that no addition of one edge can make the graph strongly connected.*

A graph consisting of two nodes and no edges.

3.25. *Given a directed graph in which each node $u \in V$ has an associated price p_u, design an algorithm which fills in the entire* `cost` *array: for each $u \in V$,*

$$\textit{cost}[u] = \textit{price of the cheapest node reachable from } u \textit{ (including } u \textit{ itself).}$$

(a) *Give a linear-time algorithm which works for directed acyclic graphs.*

Suppose u has edges to nodes $w_1, \ldots, w_k$. Then the nodes reachable from u are precisely: u itself, and the nodes reachable from all the w_i. This gives us a simple recursive formula for `cost` values:

$$\text{cost}[u] = \min\{p_u, \min_{(u,w)\in E} \text{cost}[w]\}.$$

To make it iterative, it would help if we could make sure to compute all the cost$[w]$ values before we get to cost$[u]$. Well, for a dag this is easy: just handle vertices in reverse topological order! Here's the algorithm, which is linear time because topological sorting is linear time.

> Topologically sort the dag.
> For each node $u \in V$, in reverse topological order:
> cost$[u] = \min\{p_u, \min_{(u,w)\in E} \text{cost}[w]\}$.

(b) *Extend this to a linear-time algorithm which works for all directed graphs.*

If two nodes u, v are in the same SCC, then the nodes reachable from u are identical to those reachable from v, so cost$[u]$ = cost$[v]$. Therefore we do not need to distinguish between nodes in the same SCC, and we can pretty much work with the metagraph, which is a dag!

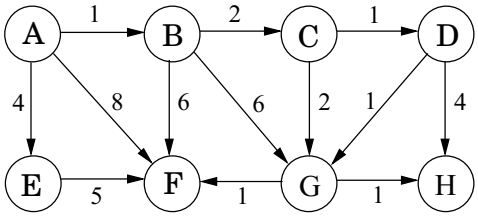
- Find the strongly connected components of G.
- For each SCC C: let the (meta-)price p^*_C for component C be the smallest price of its nodes, that is, $p^*_C = \min_{u\in C} p_u$.
- Run the dag version of the algorithm (from part (a)) on the metagraph, using these metaprices p^*_C. This returns component metacosts cost$^*[C]$.
- For each SCC C, and each node $u \in C$: cost$[u]$ = cost$^*[C]$.

This takes linear time because the algorithm for strongly connected components and the algorithm from part (a) are both linear time.

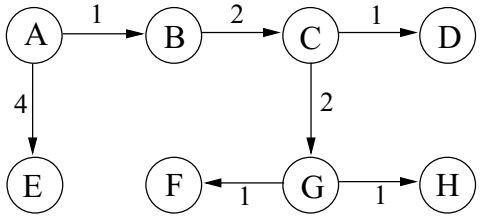
Solutions to Homework Three

CSE 101

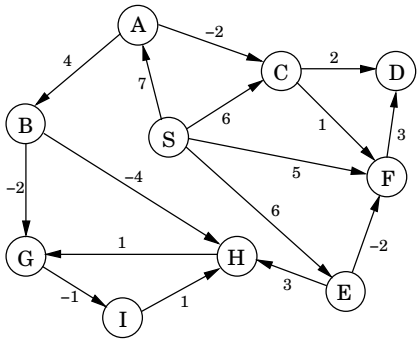
4.1. *Dijkstra example (starting from node A).*



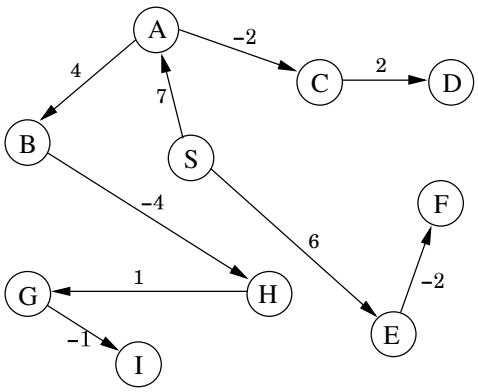
A	B	C	D	E	F	G	H
[0]	∞	∞	∞	∞	∞	∞	∞
0	[1]	∞	∞	4	8	∞	∞
0	1	[3]	∞	4	7	7	∞
0	1	3	[4]	4	7	5	∞
0	1	3	4	[4]	7	5	8
0	1	3	4	4	7	[5]	8
0	1	3	4	4	[6]	5	6
0	1	3	4	4	6	5	[6]
0	1	3	4	4	6	5	6



4.2. *Bellman-Ford example (starting from node S).*



#	S	A	B	C	D	E	F	G	H	I
0	0	∞	∞	∞	∞	∞	∞	∞	∞	∞
1	0	7	∞	6	∞	6	5	∞	∞	∞
2	0	7	11	5	8	6	4	∞	9	∞
3	0	7	11	5	7	6	4	9	7	∞
4	0	7	11	5	7	6	4	8	7	8
5	0	7	11	5	7	6	4	8	7	7



4.7. *You are given a directed graph $G = (V, E)$ with edge weights $l(\cdot)$, along with a specific node $s \in V$ and a tree $T = (V, E'), E' \subseteq E$. Give a linear-time algorithm to determine whether T is a shortest-path tree for G, starting at node s.*

Algorithm:

```
run BFS on T, returning arrays bfsdist[·] and prev[·]
dist[s] = 0
for all u ∈ V − {s}, in order of increasing bfsdist:
   dist[u] = dist[prev[u]] + l(prev[u], u)

run one round of Bellman-Ford on G with these dist[·] values
if any dist[·] values change:
   reply ``not a shortest path tree''
else:
   reply ``shortest path tree''
```

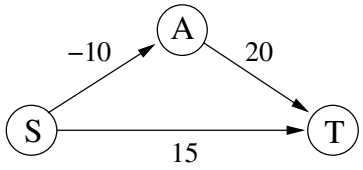
Justification: The first part of the algorithm computes distances in T, starting at node s. Because T is a tree, it has a unique path from s to any other node u. The result of BFS is two arrays: bfsdist$[u]$ gives the number of hops (ignoring edge lengths) from s to u and prev$[u]$ gives the second-last node on the path from s to u. These are then used to fill in distance values along the tree.

By our analysis from class, a single round of Bellman-Ford changes dist$[\cdot]$ values only in two cases: the distances are not optimal, or there is a negative cycle. These are exactly the two cases in which T is not a shortest-path tree.

Running time: The various components of the algorithm – BFS, the sorting of vertices by bfsdist$[\cdot]$, the loop in the middle, and a single round of Bellman-Ford – are all linear.

4.8. *Professor F. Lake wants to compute the shortest path from s to t by adding a constant to all edges to make them positive, and then running Dijkstra's algorithm.*

Counterexample: in the graph below, the shortest path from S to T is $S \to A \to T$. However, adding 10 (or more) to each edge will change the shortest path to $S \to T$.



4.20. *There is a network of roads $G = (V, E)$ connecting a set of cities V, with road lengths $l(\cdot)$. A new road is to be added, chosen from a set of potential roads E' (also with associated lengths), so as to minimize the driving distance from s to t. Give an efficient algorithm to pick the best $e' \in E'$.*

Let's assume G is undirected. Here's the algorithm.

```
dists[·] ← Dijkstra(G,l,s)
distt[·] ← Dijkstra(G,l,t)
pick (u', v') ∈ E' to minimize min{dists[u'] + l(u', v') + distt[v'], dists[v'] + l(u', v') + distt[u']}
```

Justification: The distance from s to t using a new stretch of road (u', v') in addition to G is simply the sum of the distances $s \to u' \to v' \to t$ (or alternatively $s \to v' \to u' \to t$). The two calls to Dijkstra's algorithm precompute all distances from s and all distances from t (equivalently, *to* t).

Running time: There are two calls to Dijkstra's algorithm, plus a linear-time scan over E', for a total running time of $O(|E'| + (|V| + |E|) \log |V|)$.

4.22. *The tramp steamer problem. Your steamship can ply between port cities V: a visit to city i earns you p_i dollars while the transportation cost from i to j is $c_{ij} > 0$. You want to find a cyclic route in which the ratio of profit to cost is maximized. Consider a directed graph $G = (V, E)$ whose nodes are the ports, and which has edges between each pair of ports. For any cycle C in this graph, the profit-to-cost ratio is*

$$r(C) \;=\; \frac{\sum_{(i,j)\in C} p_j}{\sum_{(i,j)\in C} c_{ij}}.$$

Let r^* be the maximum ratio achievable by a simple cycle. One way to determine r^* is by binary search: by first guessing some ratio r, and then testing whether it is too large or too small.

Consider any positive $r > 0$. Give each edge (i, j) a weight of $w_{ij} = rc_{ij} - p_j$.

(a) *Show that if there is a cycle of negative weight, then $r < r^*$.*
If there is a cycle C of negative weight, then $\sum_{(i,j)\in C} w_{ij} = \sum_{(i,j)\in C}(rc_{ij} - p_j) < 0$; rearranging,

$$r \sum_{(i,j)\in C} c_{ij} < \sum_{(i,j)\in C} p_j \quad \Rightarrow \quad r < \frac{\sum_{(i,j)\in C} p_j}{\sum_{(i,j)\in C} c_{ij}} = r(C).$$

Thus $r < r(C) \le r^*$.

(b) *Show that if all cycles in the graph have strictly positive weight, then $r > r^*$.*
By the same equations as above, if every cycle has positive weight, then $r > r(C)$ for all cycles C. Thus $r > r^*$.

(c) *Give an efficient algorithm that takes as input a desired accuracy $\epsilon > 0$ and returns a simple cycle C for which $r(C) \ge r^* - \epsilon$. Justify the correctness of your algorithm and analyze its running time in terms of $|V|$, ϵ, and $R = \max_{(i,j)\in E}(p_j/c_{ij})$.*
First we'll show $r^* \le R$. To see this, notice that there is some cycle C^* for which $r^* = r(C^*)$, so

$$r^* = \frac{\sum_{(i,j)\in C^*} p_j}{\sum_{(i,j)\in C^*} c_{ij}} \le \frac{\sum_{(i,j)\in C^*} Rc_{ij}}{\sum_{(i,j)\in C^*} c_{ij}} = R$$

(since $p_j \le Rc_{ij}$ for all $(i, j) \in E$ by definition of R).
Therefore $0 \le r^* \le R$. We'll do a binary search, always maintaining an interval $[r_{lower}, r_{upper}]$ in which r^* is contained. Here's the algorithm.

```
compute R = max_{(i,j)∈E}(p_j/c_ij)
r_lower ← 0, r_upper ← R
let C_lower be any cycle in the graph (eg. do dfs and look at a backedge)
repeat until r_upper − r_lower < ε:
    set r = (r_lower + r_upper)/2
    assign weights w_ij = rc_ij − p_j for all (i,j) ∈ E
    run Bellman-Ford on (G,w) to look for negative cycles
    if a negative cycle C is found:
        r_lower = r
        C_lower = C
    else:
        r_upper = r
return C_lower
```

Justification: From parts (a) and (b), it follows that r^* is always in $[r_{lower}, r_{upper}]$. And by construction, $r(C_{lower})$ is always at least r_{lower}. So when the process ends, we have

$$r(C_{lower}) \;\ge\; r_{lower} \;>\; r_{upper} - \epsilon \;\ge\; r^* - \epsilon.$$

Running time: The number of iterations is at most $\log R/\epsilon$, and each iteration is a Bellman-Ford computation taking time $O(|V| \cdot |E|)$. Thus the total running time is $O(|V| \cdot |E| \cdot \log(R/\epsilon))$.

Solutions to Homework Four

CSE 101

5.1. *MST example.*

The graph has two minimum spanning trees, of cost 19. There are several different orders in which Kruskal might potentially add the edges, for instance:

Edge	one side of cut
$A-E$	$\{A\}$
$E-F$	$\{A,E\}$
$B-F$	$\{A,E,F\}$
$F-G$	$\{A,B,E,F\}$
$G-H$	$\{H\}$
$C-G$	$\{C\}$
$D-G$	$\{D\}$

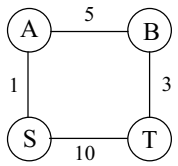
5.5. *Suppose that undirected graph $G=(V,E)$ has nonnegative edge weights and these are raised by* 1.

(a) *Can the minimum spanning tree change?*

No, it cannot. The weight of each spanning tree is increased by exactly $|V|-1$. The identity of the lightest spanning tree is therefore preserved.

(b) *Can shortest paths change?*

Yes. Here's an example: when the edge lengths are increased, the shortest path from S to T changes from $S \to A \to B \to T$ to $S \to T$.



5.9. *True or false?*

(a) *If a graph has more than $|V|-1$ edges, and there is a unique heaviest edge, then this edge cannot be part of an MST.*

False. Consider a graph consisting of a cycle (all weights 1) and a single edge leading out of a cycle node to a new node (weight 100).

(d) *If the lightest edge in a graph is unique, then it must be part of every MST.*

True. Call this edge e, and consider any spanning tree T that does not contain e. We'll show that T is not a minimum spanning tree.

Add e to T; this creates a cycle. Removing any other edge e' from the cycle yields $T'=T+e-e'$, which is connected (because removing a cycle edge cannot disconnect a graph) and has $|V|-1$ edges. Therefore T' is a spanning tree. Moreover, the weight of T' is less than that of T (since e is lighter than e'); therefore T cannot be a minimum spanning tree.

5.21. *A feedback edge set is an undirected graph is a set of edges whose removal renders the graph acyclic. Give an efficient algorithm to find a feedback edge set of minimum weight.*

Here's the algorithm, assuming the graph is connected (otherwise run each component separately).

- Given as input graph $G=(V,E)$ and edge weights w_e, find a maximum spanning tree of G by running Kruskal's algorithm on $(G,\{-w_e\})$.
- Return all edges *not* in this spanning tree.

Justification: By definition, a feedback edge set is any set of edges whose removal from G leaves behind a forest. The lightest feedback edge set therefore corresponds to the heaviest possible remaining forest, namely the maximum spanning tree.

Running time: The running time is that of Kruskal's algorithm, $O(|E| \log |V|)$.

5.22. *A new algorithm for finding minimum spanning trees is based upon the following property: "Pick any cycle in the graph, and let e be the heaviest edge in that cycle. Then there is a minimum spanning tree that does not contain e."*

(a) *Prove this property carefully.*
Call the cycle C. We'll show that for any spanning tree T that contains e, there is another spanning tree T' which doesn't contain e and whose weight is at most that of T.
Remove e from T; this splits T into two subtrees, T_1 and T_2. Since e crosses the cut (T_1, T_2), the cycle C must contain at least one other edge e' across this cut. Let $T' = T - e + e'$. T' is connected and has $|V| - 1$ edges; therefore it is a spanning tree. And since $w(e) \geq w(e')$, the weight of T' is at most that of T.

(b) *Here is the new MST algorithm. Prove it is correct.*
Let G_t be what remains of the graph after t iterations of the loop. Part (a) tells us that for any t, we have $\text{MST}(G_{t+1}) = \text{MST}(G_t)$; by induction, we therefore have $\text{MST}(G_T) = \text{MST}(G)$, where G_T is the final graph. This G_T has no remaining cycles and is therefore a spanning tree, whereby it must be a minimum spanning tree of G.

(c) *Give a linear-time algorithm to check if G has a cycle containing a specific edge e.*
Algorithm: If $e = \{u, v\}$, run explore$(G - e, u)$. If v is visited return *yes* else return *no*.
Justification: G has a cycle containing e if and only if there is a path from u to v in $G - e$.

(d) *What is the overall time taken by this algorithm, in terms of $|E|$? Explain your answer.*
Sorting takes $O(|E| \log |E|)$ and there are $|E|$ iterations of the loop, each of which takes time $O(|E|)$. The total is therefore $O(|E|^2)$.

5.23(b) *You're given a graph $G = (V, E)$ with edge weights w_e, and a minimum spanning tree $T = (V, E')$. If the weight of some edge $e \notin E'$ is reduced to $\hat{w}(e)$, show how to update the MST in linear time.*

Algorithm: Add e to T and remove the heaviest edge from the cycle thus created.

Justification: This one is a bit tricky to argue. A little lemma will make things easier.

Lemma: T is an MST of G if and only if T has a lightest edge across every possible cut.

(Here's a quick proof of the lemma. If T doesn't contain a lightest edge across some cut $(S, V - S)$, then it certainly can't be an MST: just add in the lightest edge and remove a cut edge in the cycle that is created, and the result will be a lighter spanning tree. Conversely, if T does contain a lightest edge across every cut, then you can build up T one edge at a time and invoke the cut property to show that you are always on track towards getting an MST.)

Now, getting back to our problem, T initially has a lightest edge across every cut, with respect to weights w. Then we add in e, get a cycle C and remove heavy edge e' in this cycle. We'll show that $T' = T + e - e'$ also has a lightest edge across every cut, with respect to the new weights.

Consider any cut $(S, V - S)$. There are two cases:

- e doesn't cross the cut. T used to have a lightest edge across this cut, and therefore T' still has this lightest edge, unless the edge happened to be e'. But if the edge was e', then we're still okay because T' contains the rest of cycle C, and at least one other edge in the cycle crosses the cut and has weight $\leq$ that of e'.
- e crosses the cut. In this case, T' must have the lightest edge across the cut: it has all the cut edges T used to have, plus e (minus the heavier edge e').

Solutions to Homework Five

CSE 101

5.26. *Here's a problem that occurs in automatic program analysis. For a set of variables $x_1, \ldots, x_n$, you are given some equality constraints, of the form "$x_i = x_j$" and some disequality constraints, of the form "$x_i \neq x_j$." Is it possible to satisfy all of them? Give an efficient algorithm that takes as input m constraints over n variables and decides whether the constraints can be satisfied.*

Denote the set of variables by V. Let E be the equality constraints and D the disequality constraints.

```
for all u ∈ V:  makeset(u)
for all (u, v) ∈ E:  union(u, v)
for all (u, v) ∈ D:
   if find(u) = find(v):  halt and output ''not satisfiable''
output ''satisfiable''
```

Justification: Consider an undirected graph with nodes V and edges E. The connected components of this graph are precisely the sets of variables which must have the same value. The problem is not satisfiable if and only if there is a disequality constraint between two of these connected components.

Running time: Since union and find operations take time $O(\log n)$, the overall running time is $O(n + (|D| + |E|) \log n) = O(n + m \log n)$.

2.4. *What are the running times of each of these algorithms, and which would you choose?*

- *Algorithm A solves problems by dividing them into five subproblems of half the size, recursively solving each subproblem, and then combining the solutions in linear time.*
- *Algorithm B solves problems of size n by recursively solving two subproblems of size $n-1$ and then combining the solutions in constant time.*
- *Algorithm C solves problems of size n by dividing them into nine subproblems of size $n/3$, recursively solving each subproblem, and then combining the solutions in $O(n^2)$ time.*

Algorithm A: Running time $T(n) = 5T(n/2) + O(n) = O(n^{\log_2 5})$.

Algorithm B: Running time $T(n) = 2T(n-1) + O(1) = O(2^n)$.

Algorithm C: Running time $T(n) = 9T(n/3) + O(n^2) = O(n^2 \log n)$.

Of these, Algorithm C is the fastest.

2.22. *You are given two sorted lists of size m and n. Give an $O(\log m + \log n)$ time algorithm for computing the kth smallest element in the union of the two lists.*

Here's the algorithm (assuming for convenience that m, n are powers of two).

```
function getelement(x[1...n], y[1...m], k)
if n = 0:  return y[k]
if m = 0:  return x[k]
if x[n/2] > y[m/2]:
   if k < (m + n)/2:
      return getelement(x[1...n/2], y[1...m], k)
   else:
      return getelement(x[1...n], y[(m/2) + 1...m], k − m/2)
else:
   if k < (m + n)/2:
      return getelement(x[1...n], y[1...m/2], k)
   else:
      return getelement(x[(n/2) + 1...n], y[1...m], k − n/2)
```

Brief justification: If $x[n/2] > y[m/2]$, then the top half of array x is greater than the bottom halves of both arrays. Therefore, the entire top half of array x must lie above the median of the combined arrays. Similarly, the entire bottom half of array y must lie below the median of the combined arrays. By comparing k to $(m+n)/2$, we can therefore eliminate one of these two half-arrays. The other cases are similar.

Running time: In each recursive call, a constant amount of time is taken and either m or n gets halved in value. This can happen at most $\log m + \log n$ times before one of them reaches zero. Therefore the total running time is $O(\log m + \log n)$.

2.23. *An array $A[1 \ldots n]$ is said to have a majority element if more than half of its entries are the same. Given an array, the task is to design an efficient algorithm to tell whether the array has a majority element, and, if so, to find that element. The elements of the array are not necessarily from some ordered domain like the integers, and so there can be no comparisons of the form "is $A[i] > A[j]$?". However you can answer questions of the form: "is $A[i] = A[j]$?" in constant time.*

(a) *Show how to solve this problem in $O(n \log n)$ time.*

Here's a divide-and-conquer algorithm:

```
function majority (A[1...n])
if n = 1: return A[1]
let A_L, A_R be the first and second halves of A
M_L = majority(A_L) and M_R = majority(A_R)
if neither half has a majority:
   return ''no majority''
else:
   check whether either M_L or M_R is a majority element of A
   if so, return that element; else return ''no majority''
```

Brief justification: If A has a majority element x, then x appears more than $n/2$ times in A and thus appears more than $n/4$ times in either A_L or A_R; it follows that x must also be a majority element of one (or both) of these two arrays.

Running time: $T(n) = 2T(n/2) + O(n) = O(n \log n)$.

(b) *Can you give a linear-time algorithm?*

```
function majority (A[1...n])
x = prune(A)
if x is a majority element of A:
   return x
else:
   return ''no majority''

function prune (S[1...n])
if n = 1: return S[1]
S' = [ ] (empty list)
for i = 1 to n/2:
   if S[2i − 1] = S[2i]: add S[2i] to S'
return prune(S')
```

Justification: We'll show that each iteration of the **prune** procedure maintains the following invariant: if x is a majority element of S then it is also a majority element of S'. The rest then follows.

Suppose x is a majority element of S. In an iteration of **prune**, we break S into pairs. Suppose there are k pairs of Type One and l pairs of Type Two:

- Type One: the two elements are different. In this case, we discard both.

- Type Two: the elements are the same. In this case, we keep one of them.

Since x constitutes at most half of the elements in the Type One pairs, x must be a majority element in the Type Two pairs. At the end of the iteration, what remains are l elements, one from each Type Two pair. Therefore x is the majority of these elements.

Running time. In each iteration of **prune**, the number of elements in S is reduced to $l \leq |S|/2$. Therefore, the total time taken is $T(n) \leq T(n/2) + O(n) = O(n)$.

2.25. *In class we described an algorithm that multiplies two n-bit binary integers x and y in time n^a, where $a = \log_2 3$. Call this procedure* `fastmultiply`(x, y).

(a) *An algorithm to convert the decimal integer 10^n into binary (assume n is a power of two).*

function pwr2bin(n)
  `if` $n = 1$: `return` 1010_2
  `else:`
    $z =$ `pwr2bin`$(n/2)$
    `return fastmultiply`(z, z)

The running time is $T(n) = T(n/2) + O(n^a) = O(n^a)$.

(b) *Next, an algorithm to convert any decimal integer x with n digits (where n is a power of 2) into binary.*

function dec2bin(x)
  `if` $n = 1$: `return binary`$[x]$
  `else:`
    `split` x `into two decimal numbers` x_L, x_R `with` $n/2$ `digits each`
    `return fastmultiply`$($`pwr2bin`$(n/2),$`dec2bin`$(x_L)) +$ `dec2bin`(x_R)

The running time is $T(n) = 2T(n/2) + O(n^a) = O(n^a)$.

Solutions to Homework Six

CSE 101

6.1. *A contiguous subsequence of a list S is a subsequence made up of consecutive elements of S. Give a linear-time algorithm for the following task:*

> *Input: A list of numbers, $a_1, a_2, \ldots, a_n$.*
> *Output: The contiguous subsequence of maximum sum; if you like, you can just return this sum, rather than the subsequence. Note: a subsequence of length zero has sum zero.*

Subproblem: Let $T(j)$ be the sum of the maximum-sum contiguous subsequence which ends exactly at a_j (but is possibly of length zero). We want $\max_j T(j)$.

Recursive formulation: The subsequence defining $T(j)$ either (i) has length zero, or (ii) consists of the best subsequence ending at a_{j-1}, followed by element a_j. Therefore,

$$T(j) = \max\{0, a_j + T(j-1)\}.$$

For consistency $T(0) = 0$.

Algorithm:

$T(0) = 0$
`for` $j = 1$ `to` n`:`
    $T(j) = \max\{0, a_j + T(j-1)\}$
`return` $\max_j T(j)$

Running time: Single loop: $O(n)$.

6.2. *You are going on a long road trip, and you will probably have to make overnight stops along the way. There are hotels at locations $a_1, a_2, \ldots, a_n$ miles from your starting point, where $a_1 < a_2 < \cdots < a_n$, and a_n represents your final destination. You are only allowed to stop at these hotels, and nowhere else. Moreover, you would ideally like to travel 200 miles a day. If you end up traveling x miles during some day, you incur a penalty of $(200 - x)^2$ for that day. What is the minimum total penalty (sum of daily penalties) you can incur while getting to your destination?*

Subproblem: Let $T(j)$ be the minimum penalty incurred upto location a_j, assuming you stop there. We want $T(n)$.

Recursive formulation: Suppose we stop at a_j. The previous stop is some $a_i, i < j$ (or maybe a_j is the very first stop). Let's try all possibilities for a_i:

$$T(j) = \min_{0 \le i < j} T(i) + (200 - (a_j - a_i))^2,$$

where for convenience we set $T(0) = 0$ and $a_0 = 0$.

Algorithm:

`for` $j = 1$ `to` n`:`
    $T(j) = (200 - a_j)^2$
    `for` $i = 1$ `to` $j - 1$`:`
        $T(j) = \min\{T(j), T(i) + (200 - (a_j - a_i))^2\}$
`return` $T(n)$

Running time: Two loops, $O(n^2)$.

6.7. *A subsequence is palindromic if it is the same whether read left-to-right or right-to-left. For instance, the sequence*

$$A, C, G, T, G, T, C, A, A, A, A, T, C, G$$

has many palindromic subsequences, including A, C, G, C, A and A, A, A, A (on the other hand, the subsequence A, C, T is not palindromic). Devise an algorithm which takes a sequence $x[1 \ldots n]$ and returns the (length of the) longest palindromic subsequence. Its running time should be $O(n^2)$.

Subproblem: Define $T(i, j)$ to be the length of the longest palindromic subsequence of $x[i \ldots j]$. We want $T(1, n)$.

Recursive formulation: In computing $T(i, j)$, the first question is whether $x[i] = x[j]$. If so, we can match them up and then recurse inwards, to $T(i+1, j-1)$. If not, then at least one of them is not in the palindrome.

$$T(i,j) = \begin{cases} 1 & \text{if } i = j \\ 2 + T(i+1, j-1) & \text{if } i < j \text{ and } x[i] = x[j] \\ \max\{T(i+1, j), T(i, j-1)\} & \text{otherwise} \end{cases}$$

For consistency set $T(i, i-1) = 0$ for all i.

Algorithm: Compute the $T(i, j)$ in order of increasing interval length $|j - i|$.

```
for i = 2 to n + 1:
    T(i, i − 1) = 0
for i = 1 to n:
    T(i, i) = 1
for d = 1 to n − 1:        (interval length)
    for i = 1 to n − d:
        j = i + d
        if x[i] = x[j]:
            T(i, j) = 2 + T(i + 1, j − 1)
        else:
            T(i, j) = max{T(i + 1, j), T(i, j − 1)}
return T(1, n)
```

Running time: There are $O(n^2)$ subproblems and each takes $O(1)$ time to compute, so the total running time is $O(n^2)$.

6.17. *Given an unlimited supply of coins of denominations $x_1, x_2, \ldots, x_n$, we wish to make change for a value v; that is, we wish to find a set of coins whose total value is v. This might not be possible: for instance, if the denominations are $5, 10$ then we can make change for 15 but not for 12. Give an $O(nv)$ dynamic-programming algorithm for the following problem.*

> *Input:* $x_1, \ldots, x_n; v$
> *Question: Is it possible to make change for v using coins of denominations $x_1, \ldots, x_n$?*

Subproblem: $T(u)$ is `true` if it is possible to make change for u using the given coins $x_1, x_2, \ldots, x_n$. The answer we want is $T(v)$.

Recursive formulation: Notice that

> $T(u)$ is `true` if and only if $T(u - x_i)$ is true for some i.

For consistency, set $T(0)$ to `true`.

Algorithm:

```
T[0] = true
for u = 1 to v:
    T[u] = false
    for i = 1 to n:
        if u ≥ x_i and T[u − x_i]:   T[u] = true
```

Running time: The table has size v and each entry takes $O(n)$ time to fill; therefore the total running time is $O(nv)$.

6.21. *A vertex cover of a graph $G = (V, E)$ is a subset of vertices $S \subseteq V$ which includes at least one endpoint of every edge in E. Give a linear-time algorithm for the following task.*

Input: An undirected tree $T = (V, E)$.

Output: The size of the smallest vertex cover of T.

Subproblem: Root the tree at any node r. For each $u \in V$, define

$$T(u) = \text{size of smallest vertex cover of the subtree rooted at } u.$$

We want $T(r)$.

Recursive formulation: In figuring out $T(u)$, the most immediate question is whether u is in the vertex cover. If not, then its children must be in the vertex cover. Let $C(u)$ be the set of u's children, and let $G(u)$ be its grandchildren. Then

$$T(u) = \min \begin{cases} 1 + \sum_{w \in C(u)} T(w) \\ |C(u)| + \sum_{z \in G(u)} T(z) \end{cases}$$

where $|C(u)|$ is the number of children of node u. The first case includes u in the vertex cover; the second case does not. Base case: $T(u) = 0$ for any leaf u.

Algorithm:

```
Pick any root node r, and run DFS starting from r.  Along the way, record post
numbers and identify leaf nodes (of the rooted tree).

for all nodes u, in order of increasing post number:
   if u is a leaf:
      T(u) = 0
   else:
      S_1 = 1 (option 1:  include u in the vertex cover)
      for all (u,w) ∈ E such that post(w) < post(u):  (ie.  w = child of u):
         S_1 = S_1 + T(w)
      S_2 = 0 (option 2:  don't include u in the vertex cover)
      for all (u,w) ∈ E such that post(w) < post(u):  (ie.  w = child of u):
         S_2 = S_2 + 1
         for all (w,z) ∈ E such that post(z) < post(w):  (ie.  z = grandchild of u):
            S_2 = S_2 + T(z)
      T(u) = min{S_1, S_2}
return T(r)
```

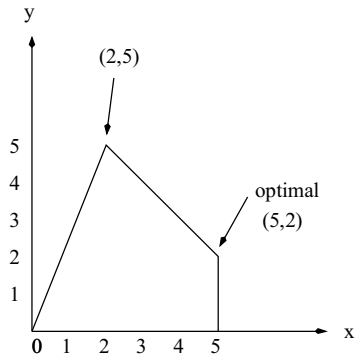
Running time: The amount of work done at each node is proportional to its number of grandchildren, $|G(u)|$. Since $\sum_u |G(u)| \leq |V|$ (each node has at most one grandparent), the overall work done is linear.

Solutions to Homework Seven

CSE 101

7.1. *Plot the feasible region and identify the optimal solution of the following linear program.*

$$\begin{aligned} \textit{maximize} \quad & 5x + 3y \\ 5x - 2y &\geq 0 \\ x + y &\leq 7 \\ x &\leq 5 \\ x &\geq 0 \\ y &\geq 0 \end{aligned}$$



7.3. *A cargo plane can carry a maximum weight of 100 tons and a maximum volume of 60 cubic meters. There are three materials to be transported, and the cargo company may choose to carry any amount of each, upto the maximum available limits given below.*

- *Material 1 has density 2 tons/cubic meter, maximum available amount 40 cubic meters, and revenue $1,000 per cubic meter.*
- *Material 2 has density 1 ton/cubic meter, maximum available amount 30 cubic meters, and revenue $1,200 per cubic meter.*
- *Material 3 has density 3 tons/cubic meter, maximum available amount 20 cubic meters, and revenue $12,000 per cubic meter.*

Write a linear program that optimizes revenue within the constraints.

Let x_1 denote the volume of Material 1, in cubic meters; similarly define x_2, x_3 for Materials 2 and 3.

$$\begin{aligned} \text{maximize} \quad & 1000x_1 + 1200x_2 + 12000x_3 \\ 2x_1 + x_2 + 3x_3 &\leq 100 \\ x_1 + x_2 + x_3 &\leq 60 \\ x_1 &\leq 40 \\ x_2 &\leq 30 \\ x_3 &\leq 20 \\ x_1, x_2, x_3 &\geq 0 \end{aligned}$$

7.5. *The Canine Products company offers two dog foods, Frisky Pup and Husky Hound, that are made from a blend of cereal and meat. A package of Frisky Pup requires* 1 *pound of cereal and* 1.5 *pounds of meat,*

and sells for \$7. *A package of Husky Hound uses* 2 *pounds of cereal and* 1 *pound of meat, and sells for* \$6. *Raw cereal costs* \$1 *per pound and raw meat costs* \$2 *per pound. It also costs* \$1.40 *to package the Frisky Pup and* \$0.60 *to package the Husky Hound. A total of* 240,000 *pounds of cereal and* 180,000 *pounds of meat are available each month. The only production bottleneck is that the factory can only package* 110,000 *bags of Frisky Pup per month. Needless to say, management would like to maximize profit.*

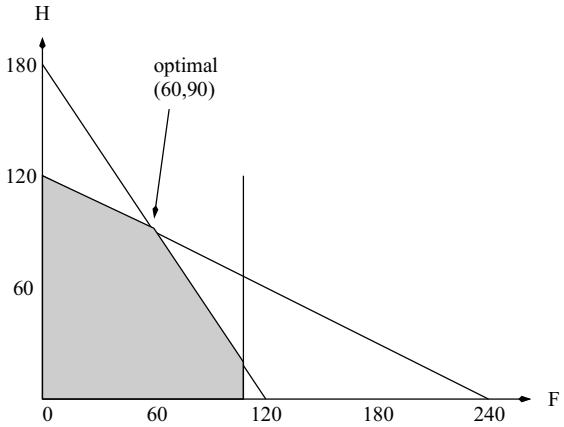
(a) *Formulate the problem as a linear program in two variables.*

Let F denote the number of bags of Frisky Pup produced per month, and H the number of bags of Husky Hound. The profit per bag of Frisky Pup is $\$7 - (\$1 + \$3 + \$1.40) = \$1.60$. The profit per bag of Husky Hound is $\$6 - (\$2 + \$2 + \$0.60) = \$1.40$.

$$\begin{aligned} \text{maximize} \quad & 1.6F + 1.4H \\ & F + 2H \leq 240000 \\ & 1.5F + H \leq 180000 \\ & F \leq 110000 \\ & F, H \geq 0 \end{aligned}$$

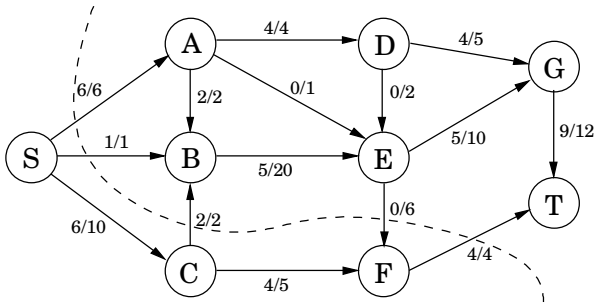
(b) *Graph the feasible region, give the coordinates of every vertex, and circle the vertex maximizing profit. What is the maximum profit possible?*

The graph below shows multiples of 1000. The vertices (again in multiples of 1000) are $(F, H) = (0, 0), (0, 120), (60, 90), (110, 15), (110, 0)$. The maximum profit is \$222,000.

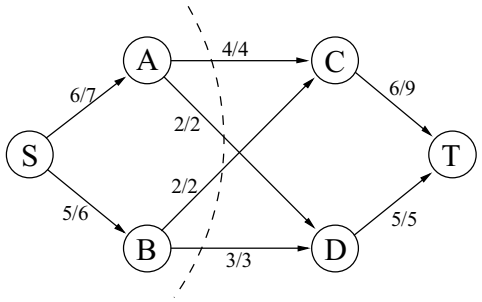


7.10. *For the following network, find the maximum flow and a matching cut.*

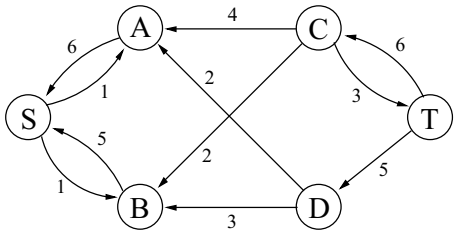
The maximum flow is 13, and is shown below. The dotted line shows a cut of capacity 13.



7.17. The network below includes the max-flow and a matching min-cut.



(b) *Draw the residual graph G_f (along with its edge capacities). In this residual network, mark the vertices reachable from S and the vertices from which T is reachable.*

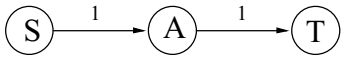


The vertices reachable from S are S, A, B. Vertex T is reachable from C, T.

(c) *An edge of a network is called a bottleneck edge if increasing its capacity results in an increase in the maximum flow. List all bottleneck edges in the above network.*

They are (A, C) and (B, C).

(d) *Give an example of a network which has no bottleneck edges.*



(e) *Give an efficient algorithm to identify all bottleneck edges in a network.*

Algorithm:

i. Run the Ford-Fulkerson algorithm, making sure on each iteration to pick the $s-t$ path with the fewest edges (this can be done by BFS). Let f be the maximum flow returned.

ii. Construct the residual graph G_f.

iii. Run `explore`(G_f, s) to find the set of vertices A reachable from s.

iv. Construct the reverse graph G_f^R (recall a previous homework) and run `explore`(G_f^R, t) to find the set of vertices B from which t is reachable.

v. For each edge $(u, w) \in E$: if $u \in A$ and $w \in B$ then output (u, w).

Justification: The algorithm is based on the following principle:

> Edge (u, w) is a bottleneck edge if and only if increasing its capacity would create a path from s to t in G_f.

There are two cases to check: if increasing $c(u, w)$ creates a path from s to t in G_f, then flow f can be increased along that path, so the edge is a bottleneck. Conversely, if increasing $c(u, w)$ does not create a path from s to t, then f remains a max flow (the Ford Fulkerson algorithm makes no further progress), and thus the edge is not a bottleneck.

In short: an edge is a bottleneck if and only if it leads from set A to set B.

Running time: This particular way of running the Ford-Fulkerson algorithm requires $O(|V| \cdot |E|)$ iterations, each taking $O(|E|)$ time, a total of $O(|V| \cdot |E|^2)$. The subsequent steps are all linear.

7.18. *Solve the following two flow problems by reducing them to the original max-flow problem.*

(a) *There are many sources and many sinks, and we wish to maximize the total flow from all sources to all sinks.*

Create a new graph G' which is just like the original one except that it has two additional nodes, s and t. Add edges from s to all the sources of G, and add edges from all the sinks of G to t. Give each of these new edges a capacity of C, where C is the sum of the capacities of all the edges in the original network (and therefore C is an upper bound on the maximum flow).

Now find the maximum flow in G'.

(b) *Each vertex also has a capacity on the maximum flow that can enter it.*

If a vertex u has capacity $c(u)$, replace it by two vertices, u' and u''. Edges which previously led into u should now lead into u'; edges previously leading out of u should now lead out of u''. Finally, add an edge (u', u'') with edge-capacity $c(u)$.

This creates a modified graph; find the maximum flow in it.

7.22. *In a particular network $G = (V, E)$ whose edges have integer capacities c_e, we have already found the maximum flow f from node s to node t. However, we now find out that one of the capacity values we used was wrong: for edge (u, v) we used c_{uv} whereas it should have been $c_{uv} - 1$. This is unfortunate because the flow f uses that particular edge at full capacity: $f_{uv} = c_{uv}$.*

We could redo the flow computation from scratch, but there's a faster way. Show how a new optimal flow can be computed in $O(|V| + |E|)$ time.

Algorithm:

(i) Let E' be the edges $e \in E$ for which $f(e) > 0$, and let $G' = (V, E')$. Run `explore`(G', s) and `explore`(G', v) to find a path P_1 from s to u and a path P_2 from v to t.

(ii) [Special case: If P_1 and P_2 have some edge e in common, then $P_1 \cup \{(u, v)\} \cup P_2$ has a directed cycle containing (u, v). In this case, flow along this cycle can be reduced by one unit without changing the size of the overall flow. Return the resulting flow.]

(iii) Reduce flow by one unit along $P_1 \cup \{(u, v)\} \cup P_2$.

(iv) Run Ford-Fulkerson with this starting flow.

Justification and running time:

Say the original flow has size F.

Let's ignore the special case (ii). After step (iii) of the algorithm, we have a legal flow which satisfies the new capacity constraint and moreover has size $F - 1$.

Step (iv), Ford-Fulkerson, then gives us the optimal flow under the new capacity constraint. However, we know this flow is at most F, and thus Ford-Fulkerson runs for just one iteration.

Since each of the steps is linear, the total running time is linear, that is, $O(|V| + |E|)$.

Homework One, due Tue 1/16

CSE 101

Notes.

1. At the top of your homework, write your name and ID number. Also indicate the discussion section at which you will pick up the graded homework.
2. This homework is due within the first five minutes of class on the due date.
3. Whenever asked to come up with an algorithm, you must always prove its correctness and running time.
4. Please make your answers as concise as possible.

Homework problems.

1. Exercise 0.1 (a), (c), (e), (g), (i), (k), (m), (o), (q). There is no need to justify your answers.

2. Exercise 0.3.

3. While discussing algorithms for computing the Fibonacci numbers F_n, we came across the recurrence relation

$$T(n) = \begin{cases} T(n-1) + T(n-2) + 1 & \text{if } n > 1 \\ 2 & \text{if } n = 1 \\ 1 & \text{if } n = 0 \end{cases}$$

   Show by induction that $T(n) = F_{n+3} - 1$. (Therefore $T(n) = \Theta(F_n)$.)

4. Exercise 3.5. *Make sure your pseudocode is completely unambiguous.*

5. Exercise 3.6.