CS 170 Spring 2000
Solutions and grading standards for exam 1
Clancy

Exam information

179 students took the exam. Scores ranged from 5 to 30, with a median of 16 and an average of 16.3. There were 19 scores between 23 and 30,82 between 16 and 22, 73 between 8 and 15, and 5 between 1 and 7. (I estimate, based on past experience in other classes, that three-quarters of the points on exams plus good grades on homework and projects would earn you an A-; similarly, a test grade of 16 may be projected to a B-.)

There were two versions of the exam, A and B. (The version indicator appears at the bottom of the first page.)

If you think we made a mistake in grading your exam, describe the mistake in writing and hand the description with the exam to your discussion t.a. or to Mike Clancy. We will regrade the entire exam.

Solutions and grading standards

Problem 0 (1 point)

If you earned some credit on a problem and did not put your name on the page, you lost 1 point. If you did not indicate your discussion section and t.a., you lost 1 point. (The reason for this apparent harshness is that exams can get misplaced or come unstapled, and we would like to make sure that every page is identifiable. We also need to know where you will expect to get your exam returned.) If you did not identify where you were sitting, you lost 1 point.

Problem 1 (4 points)

The two versions of this problem involved the following sequences of operations:

Version A	Version B
Union (0, 1);	Union (4, 5);
Union (2, 3);	Union (2, 3);
Union (0, 2);	Union (2, 4);
Union (4, 5);	Union (0, 1);
Union (0, 4);	Union (2, 0);

Both sequences lead to similar trees; thus we present only the solution to version A.

operation	*resulting structure*	*comments*
`Union (0, 1)`	rank 1	A singleton set has rank 0. In the union of two singleton sets, the representative element of the second argument becomes the parent and the result has rank 1 higher than the arguments.
`Union (2, 3);`	both rank 1	Same as the previous operation.
`Union (0, 2);`	rank 2	The two representative elements are 1 and 3. Since both structures have rank 1, 3 becomes the parent. Path compression wasn't relevant here since 0 and 2 both already pointed to their respective parents.
`Union (4, 5);`	rank 2 and rank 1	Same as operation 1.
`Union (0, 4)`	rank 2	The representative element for 0 is 3, and path compression happens while determining that. The representative element of 4 is 5, and it has rank 1. Since that's lower than the rank of the tree rooted at 3, 5 becomes a child of 3.

This was a 4-point problem. Points were deducted as follows: You lost 1 point for a rank or link wrong in the last step, neglecting to compress one of the paths in the last step (a common error), or for all ranks off by 1. You lost 2 points for a more serious error, for example, getting the argument order or ranks backward, compressing after combining the trees rather than before, or linking to a nonrepresentative element. All of these errors were also pretty common. A second serious error lost you only 1 more point. Losing a node was a 3-point error.

Problem 2 (6 points)

Both versions of this problem dealt with the effect of using an unsorted linked list to implement a priority queue in one of the graph algorithms we covered. Version A asked about Prims minimum spanning tree algorithm; version B asked about Dijkstras shortest path algorithm. The two algorithms as stated in CLR (modified slightly to incorporate C syntax) appear

Prim's algorithm

```
MSTPrim (G, w, r
Q = V[G];
for (each u ∈ Q) {
    key[u] = ∞;

key[r]
π[r] =
while (Q not empty) {
    u = ExtractMin (Q);
    for (each v ∈ Adj[u]) {
        if (v ∈ Q && w(u,v) < key[v]
            π[v] = u;
            key[v] = w(u,v);
```

Dijkstra's algorithm

```
Dijkstra (G, w, s):
InitializeSingleSource (G,
S = empty;
Q = V[G];
while (Q not empty) {
    u = ExtractMin (Q);
    S = S ∪ {u};
    for (each v ∈ Adj[u]) {
        Relax (u, v, w);
```

below.

Part a asked you to give an estimate of the worst-case running time on a graph with n vertices and e edges. In Dijkstras algorithm, the priority queue is used in four places: its initialized in the assignment to Q, its checked in the while condition, the vertex u is extracted from it, and a value of one of its elements v may be updated as a result of relaxation. Prims algorithm has all those uses plus one more, the test for membership of v in the queue in the innermost loop.

Initialization of the queue is 0(n) and checking for an empty queue is 0(1), regardless of implementation. There are two places that using an unsorted linked list affects the running time: ExtractMin, where all elements must be checked to determine the element with smallest value, and updating an element, which may require movement in a binary heap but requires no extra processing in an unsorted list.

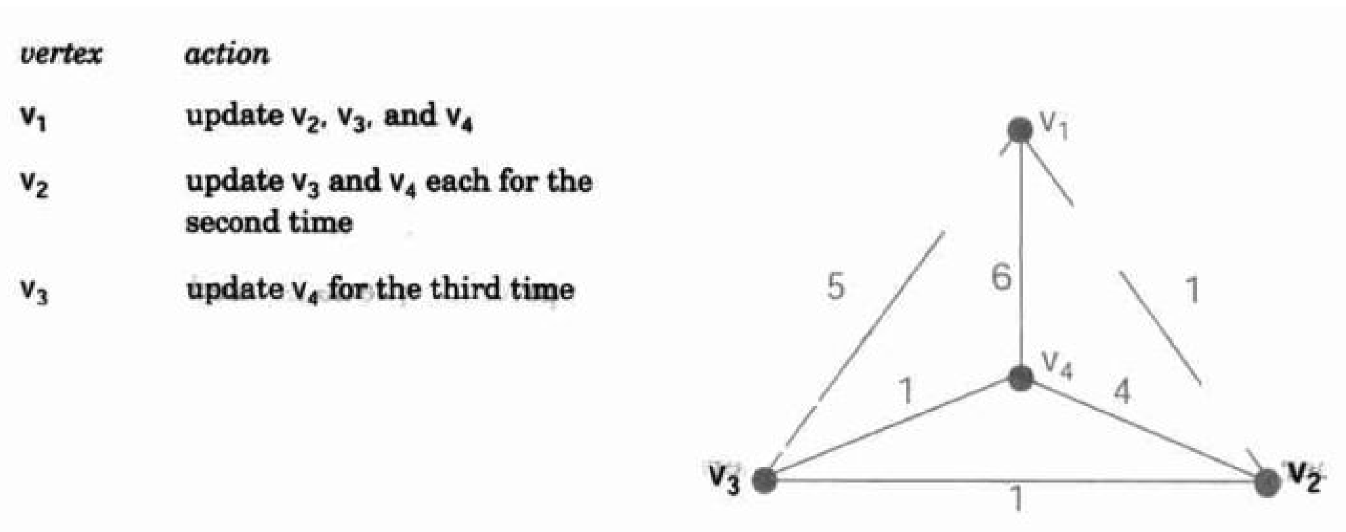
Prims algorithm contains the extra test for membership in the queue. This can be implemented merely by a test of a vertexs pointer into the queue; a null pointer means that the vertex is not queued. A less efficient implementation is to search the queue, which requires 0(n) operations. This would not be any less efficient than the corresponding operation using a binary heap, however.

At any rate, CLRs analysis noted that the outer loop in each algorithm executed once for each vertex, while the inner loop was executed once for each edge. Thus ExtractMin contributes n * 0(n) operations, and updating contributes e * 0(1) for a total of 0(n^2 + e) = 0(n^2) in each algorithm. The membership test in

Prims algorithm, if coded inappropriately, contributes e * 0(n) operations, which dominates the total running time.

Part a was worth 4 points. Grading awarded 1 point to repeating the information from previous analyses about the number of iterations for each loop, 2 points for noting that ExtractMin was 0(n) and updating an element was 0(1), and 1 point to providing some rationale for these values. The most common error here was to overlook the updating operation, which lost 1 point. A number of you provided no rationale for the estimates, also a 1-point deduction. Finally, a few of you seemed not to understand that no sorting was taking place.

Part b, which contributed the other 2 points, was to produce a worst-case graph and explain how it produces the worst case. No points were awarded for an answer without an explanation. Interestingly, ExtractMin exhibits worst-case behavior with any assignment of weights to edges, since every element in the queue must be examined to find the minimum. We gave full credit to an answer that noted this. A more detailed analysis, however, reveals that the number of updates will depend on the assignment of edge weights; in the worst case, a neighbors key value (in Prims algorithm) or d value (in Dijkstras) will be updated every time the neighbor is examined. A pattern that produces the worst case and an example of a corresponding assignment of edge weights are shown below.



Many students produced a graph in which the last vertex in the queue was always the vertex with minimum value. As noted previously, the position of the minimum-valued vertex in the queue does not affect the running time, so this answer lost 1 point.

Problem 3 (3 points)

This problem involved getting a *theta* value for the solution to a recurrence. Version As recurrence was

T(n) = 9 * T(n / 3) + n^2 + n * log(n)

while version Bs was

T(n) = 4 * T(n / 2) + sqrt(n) + n * log(n)

Both recurrences could be solved with the Master Theorem. In version A, the variables a, b, and f have values a = 9, b = 3, and f(n) = n^2 + n log(n), which is *Theta*(n^2). The value of log_base_b(a) is 2 since 9 = 3^2. Case 2 of the Master Theorem applies to give an estimate of *Theta* (n^2 * log(n)). Version Bs recurrence

required slightly more work. There, a = 4, b = 2, and f(n) = n^(1/2)+ log(n), which is O(n^(1/2)). The value of log_base_b(a) is 2 since 4 = 2^2. Case 1 of the Master Theorem applies, with the desired epsilon being 1.5, giving an estimate of *Theta*(n^2).

An alternate form of the Master Theorem, given on page 9 of the Readings, uses a d variable, which is 2 in version As recurrence and 1/2 in the version B recurrence. Using this version of the Master Theorem was fine.

Points were awarded as follows: 1 for identifying the variable values necessary to apply the Master Theorem, 1 more for correctly specifying the order of f, and 1 more for successfully applying the Master Theorem. You lost 1 point for not simplifying log_base_3(9) or log_base_2(4) to 2. Another common error that lost 1 point on version B was to note that f(n) = O(n^2). This does not provide enough information to determine which case of the Master Theorem applies.

With difficulty, one could produce a solution using the iteration method instead of the Master Theorem. Only one person succeeded with this approach. You received 1 point if you made any forward progress in the analysis.

Problem 4 (8 points)

This problem was the same on both versions of the exam. You were to prove that a directed graph G = (V, E) with no isolated vertices is strongly connected if and only if there is a cycle in G that includes every edge at least once (and possibiy more than once). As with any if and only if proposition, there were two things to prove:

1. If G is strongly connected, then there is a cycle that contains every edge at least once.

2. If there is a cycle that contains every edge at least once, then G is strongly connected.

The second part is the easier of the two. Since the assumed cycle contains every edge and there are no isolated vertices, the cycle also contains every vertex. Consider any two vertices u and v. A path from u to v is found by starting at u on the cycle and following it around to v; a path from v to u is found in the same way. These two paths, found for all pairs of vertices in the graph, satisfy the conditions for strong connectivity.

There were two successful approaches to the first part:

- a proof by contradiction that involved assuming that no cycle could include every edge;
- a proof by construction, involving the repeated augmenting of a cycle to include more edges.

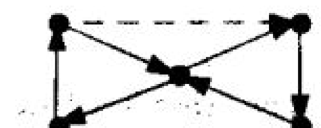
Here are sample solutions for each approach.

Suppose that G is strongly connected, yet no cycle includes every edge in the graph. Let C be a cycle that includes the maximum number of edges of G, and let (u, v) be an edge not included. Let w be some vertex on the cycle. Since G is strongly connected, there is a path P1 from w to u and a path P2 from v to w. These two paths, along with (u, v) and C, comprise a longer cycle. It starts somewhere on C, reaches w, detours on P1 through (u,v), returns to w on F2, and continues along C to the starting point. It also contains all the edges in C plus an edge not in C, namely (u,v), which contradicts the assumption that C included the maximum number of edges possible. Thus there is no cycle that includes as many edges as possible without including them all, which is what we wanted to prove.

A proof by construction is similar in many respects. (It is similar in addition to the Euler tour proof on an early homework assignment.) The construction starts with some cycle C1 that, say, contains a vertex w. For each edge (u,v) thats not on C1, we find the path from w to u and the path from v to w; this cycle, added to C1 in the same way as described in the previous proof, creates a cycle C2. Repeat this process until some Ck includes all the edges in G.

Another constructive proof gradually increases a path until it includes all the edges, then finds a way back to where it started. This approach picks a starting vertex, finds an edge (u, v) thats not yet traversed, and travels to u and then across the edge. Strong connectivity provides the path to u. >From v, the algorithm finds another untraversed edge, and extends the path through that edge in the same way. This process continues until all edges have been traversed, at which point it finds a path back to the start vertex provided by the strong connectivity assumption.

Few solutions were completely correct. One false claim was that one of the endpoints of an edge not on a cycle was unreachable from the cycle. Another was that since all vertices u and V are connected by paths from u to v and back, the collection of cycles that results cover all the edges of G. The graph below shows a counterexample in which the dotted edge is not in any of a particular set of cycles.



Some students attempted a proof by induction either on the number of vertices or on the number of edges. Such a proof is unfortunately difficult because removing a vertex or edge from a graph may result in a graph thats not strongly connected, and that therefore doesnt satisfy the induction hypothesis. No one provided a correct proof.

The if cycle, then strongly connected part was worth 2 points. You lost 1 point for not connecting the absence of isolated vertices to the proof. You lost 2 points for making no mention of how a path that contains all edges relates to a path that contains all vertices.

The if strongly connected, then cycle part was worth 6 points. Point allocation depended on your approach. For the proof by contradiction, you received 2 points for making clear what was to be proven, including the specification of a particular cycle and a missing edge that it doesnt contain. 1 more point came from situating the missing edge on some cycle. The other 3 points were awarded for joining that cycle to another cycle to complete the contradiction. For the proof by construction, you received 3 points for specifying a set of cycles that collectively included all the edges, 1 more point for attempting to combine the cycles, and 2 more points for giving a correct procedure for combining the cycles. Partial credit on the 3-point chunks could be earned with statements that were correct but unjustified or too vague.

A proof by induction on the number of vertices could earn as many as 4 points, divided as follows. 2 points were awarded for considering all edges into and out of the new vertex. 1 point was awarded for trying to splice these edges into a cycle assumed as the induction hypothesis, and 1 more point was awarded for doing this correctly.

Two other common errors involved misreading or misunderstanding the theorem to be proved. Some students viewed the given property that G has no isolated vertices as something to be proved. Others confused a cycle containing all the edges with a cycle containing all the vertices. (With the latter error, you could still earn substantial partial credit if you mimicked the correct approach to the proof.)

Unfortunately, some of you overlooked the if and only if nature of the proposition, losing either 2 or 6 points depending on which part you overlooked. You could earn 1 point on this problem merely by specifying the two things to prove.

Problem 5 (8 points)

This problem was the same on both versions. You were to find an efficient algorithm that, given a directed acyclic graph G = (V, E) and a vertex *a* in V, counts the number of paths from *a* to each other vertex (The original problem said all other vertices, but this was corrected at the exam. Another clarification made at the exam was that G contains no multiple edges; that is, for any u and v, the number of edges connecting u to v is at most 1, and the number of edges connecting V to u is at most 1.)

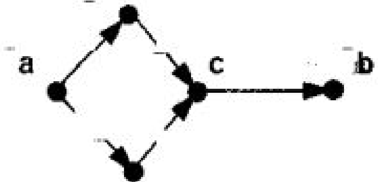
There were a number of solution approaches. The best one took advantage of the fact that G is a DAG and can therefore be topologically sorted. This leads to an algorithm that runs in time O(n + e):

> Set the path counts of all vertices to 0, and set as path count to 1.
> Topologically sort the vertices.
> For each vertex u in topological order, do the following:

For each neighbor v of u, add us path count to vs path count.

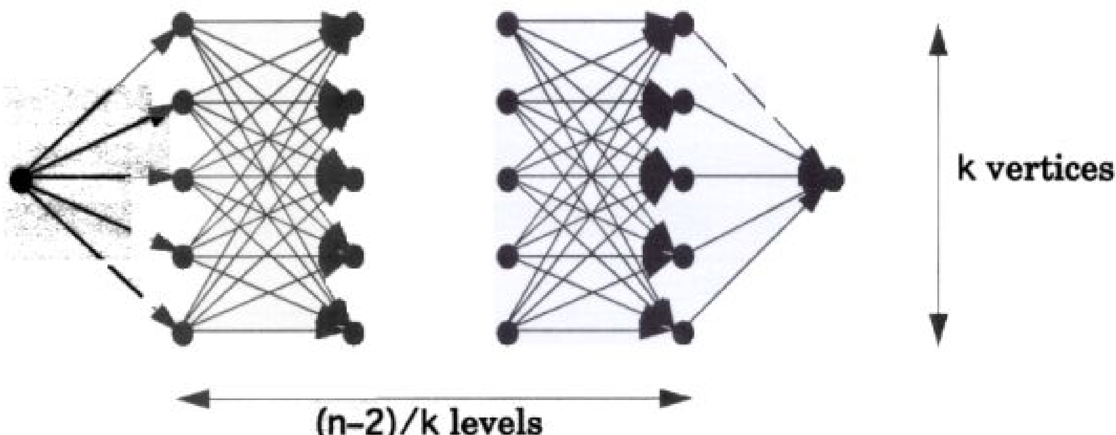
To prove this (which you didnt need to do), one would note that a vertex u isnt reached until all the path counts of vertices between it and *a* have been updated; this is because of the topological ordering. Thus, when U is processed in the outer loop, the number of paths to each of its predecessors is known and tallied in us own count.

Not taking account of multiple ways to get to a vertex from *a* was the flaw in another common solution, namely applying CLR's DFS or BFS algorithm directly and merely increment a vertexs path count each time it is visited. The problem with this is demonstrated in the graph below. The first time vertex c is encountered, its marked:



when c is next encountered on the second path out of *a*, there will be no way to go past it to add the second path into bs count.

A third approach attempted to solve the problem of missing paths by continuing the search through marked vertices. This searchagain, either depth first or breadth firsttraverses every path from *a* to every other vertex. Unfortunately, the tradeoff is speed; this algorithm run in time exponential in the number of vertices. An example on which it performs badly is shown below. If k is approximately equal to the



square root of n, the number of paths from one end of the graph to the other is approximately n^(sqrt(n)/2).

One last approach arose from noting that the (u,v)-th entry in the kth power of the adjacency matrix of G is the number of paths of length k from u to v. If the first n-1 powers of Gs adjacency matrix are added, the resulting matrix contains the desired path counts. Matrix multiplication (at least the standard algorithm) is O(n^3), and there are n-1 multiplications, yielding a run time estimate of O(n^4).

This problem was worth 8 points. You could lose points both for inefficiency and for incorrectness. For example, a solution that computed powers of the adjacency matrix correctly earned 6 out of 8, and a correct DFS-baaed solution earned 4 out of 8. Error deductions varied; an off-by-one initialization or update lost 1, while the DFS-based solution that marked vertices already encountered lost 4. Some 2-point errors were a more serious error in computing the path count, the use only of the n-1st power of the adjacency matrix rather than the sum of the powers, or a generally correct solution that topologically sorted the vertices but then accidentally gave a positive path count to a vertex not reachable from *a*. You also lost 1 point if you didnt maintain separate path counts for the individual vertices, instead doing a separate search for all paths to each destination vertex. Solutions that started by topologically sorting the vertices but then made no subsequent use of the topological ordering got no credit for the good start.

A few students mentioned memoization as a way to reduce the inefficiency. You had to be explicit about how it would be applied to get any credit for it, though. If your algorithm was incorrect, providing a DAG on which it would work incorrectly would earn you an extra point.

On the low end, if you invoked the term DFS or BFS or topological sort and then followed up with some kind of use of the technique, you earned at least 2 points.

Posted by HKN (Electrical Engineering and Computer Science Honor Society)
University of California at Berkeley
If you have any questions about these online exams
please contact examfile@hkn.eecs.berkeley.edu