HC to HP Reduction

```
bool HamCycle(graph G) {
    for each edge {u,v} in G {
        copy G to a new graph G'
        delete edge {u,v} from G'
        add new vertices x and y to G'
        add new edges {x,u} and {y,v} to G'
        if (HamPath(G')) return true
    }
    return false                          // failed for every edge
}
```

This is a rather inefficient reduction, but it does work. In particular it makes $O(e)$ calls to the `HamPath()` procedure. Can you see how to do it with fewer calls? (Hint: Consider applying this to the edges coming out of just one vertex.) Can you see how to do it with only one call? (Hint: This is trickier.)

As before, notice that we didn't really attempt to solve either problem. We just tried to figure out how to make a procedure for one problem (Hamiltonian path) work to solve another problem (Hamiltonian cycle). Since HC is NP-complete, this means that there is not likely to be an efficient solution to HP either.

Lecture 29: Final Review

(Tuesday, May 12, 1998)

Final exam: As mentioned before, the exam will be comprehensive, but it will stress material since the second midterm exam. I would estimate that about 50–70% of the exam will cover material since the last midterm, and the remainder will be comprehensive. The exam will be closed book/closed notes with three sheets of notes (front and back).

Overview: This semester we have discussed general approaches to algorithm design. The goal of this course is to improve your skills in designing good programs, especially on complex problems, where it is not obvious how to design a good solution. Finding good computational solutions to problems involves many skills. Here we have focused on the higher level aspects of the problem: what approaches to use in designing good algorithms, how generate a rough sketch the efficiency of your algorithm (through asymptotic analysis), how to focus on the essential mathematical aspects of the problem, and strip away the complicating elements (such as data representations, I/O, etc.)

Of course, to be a complete programmer, you need to be able to orchestrate all of these elements. The main thrust of this course has only been on the initial stages of this design process. However, these are important stages, because a poor initial design is much harder to fix later. Still, don't stop with your first solution to any problem. As we saw with sorting, there may be many ways of solving a problem. Even algorithms that are asymptotically equivalent (such as MergeSort, HeapSort, and QuickSort) have advantages over one another.

The intent of the course has been to investigate basic techniques for algorithm analysis, various algorithm design paradigms: divide-and-conquer graph traversals, dynamic programming, etc. Finally we have discussed a class of very hard problems to solve, called NP-complete problems, and how to show that problems are in this class. Here is an overview of the topics that we covered this semester.

Tools of Algorithm Analysis:

Asymptotics: O, Ω, Θ. General facts about growth rates of functions.

Summations: Analysis of looping programs, common summations, solving complex summations, integral approximation, constructive induction.

Recurrences: Analysis of recursive programs, strong induction, expansion, Master Theorem.

Sorting:

Mergesort: Stable, $\Theta(n \log n)$ sorting algorithm.

Heapsort: Nonstable, $\Theta(n \log n)$, in-place algorithm. A heap is an important data structure for implementation of priority queues (a queue in which the highest priority item is dequeued first).

Quicksort: Nonstable, $\Theta(n \log n)$ expected case, (almost) in-place sorting algorithm. This is regarded as the fastest of these sorting algorithms, primarily because of its pattern of locality of reference.

Sorting lower bounds: Any sorting algorithm that is based on comparisons requires $\Omega(n \log n)$ steps in the worst-case. The argument is based on a decision tree. Considering the number of possible outcomes, and observe that they form the leaves of the decision tree. The height of the decision tree is $\Omega(\lg N)$, where N is the number of leaves. In this case, $N = n!$, the number of different permutations of n keys.

Linear time sorting: If you are sorting small integers in the range from 1 to k, then you can applying counting sort in $\Theta(n + k)$ time. If k is too large, then you can try breaking the numbers up into smaller digits, and apply radix sort instead. Radix sort just applies counting sort to each digit individually. If there are d digits, then its running time is $\Theta(d(n + k))$, where k is the number of different values in each digit.

Graphs: We presented basic definitions of graphs and digraphs. A graph (digraph) consists of a set of vertices and a set of undirected (directed) edges. Recall that the number of edges in a graph can generally be as large as $O(n^2)$, but is often smaller (closer to $O(n)$). A graph is *sparse* if the number of edges is $o(n^2)$, and dense otherwise.

We discussed two representations:

Adjacency matrix: $A[u, v] = 1$ if $(u, v) \in E$. These are simple, but require $\Theta(n^2)$ storage. Good for dense graphs.

Adjacency list: $Adj[u]$ is a pointer to a linked list containing the neighbors of u. These are better for sparse graphs, since they only require $\Theta(n + e)$ storage.

Breadth-first search: We discussed one graph algorithm: *breadth first search*. This is a way of traversing the vertices of a graph in increasing order of distance from a source vertex. Recall that it colors vertices (white, gray, black) to indicate their status in the search, and it also uses a FIFO queue to determine which order it will visit the vertices. When we process the next vertex, we simply visit (that is, enqueue) all of its unvisited neighbors. This runs in $\Theta(n + e)$ time. (If the queue is replaced by a stack, then we get a different type of search algorithm, called depth-first search.) We showed that breadth-first search could be used to compute shortest paths from a single source vertex in an (unweighted) graph or digraph.

Dynamic Programming: Dynamic programming is an important design technique used in many optimization problems. Its basic elements are those of subdividing large complex problems into smaller subproblems, solving subproblems in a bottom-up manner (going from smaller to larger). An important idea in dynamic programming is that of the principal of optimality: For the global problem to be solved optimally, the subproblems should be solved optimally. This is not always the case (e.g., when there is dependence between the subproblems, it might be better to do worse and one to get a big savings on the other).

Floyd-Warshall Algorithm: (Section 26.2) Shortest paths in a weighted digraph between all pairs of vertices. This algorithm allows negative cost edges, provided that there are no negative cost cycles. We gave two algorithms. The first was based on a DP formulation of

building up paths based on the number of edges allowed (taking $\Theta(n^4)$ time). The second (the Floyd-Warshall algorithm) uses a DP formulation based on considering which vertices you are allowed to pass through. It takes $O(n^3)$ time.

Longest Common Subsequence: (Section 16.3) Find the longest subsequence of common characters between two character strings. We showed that the LCS of two sequences of lengths n and m could be computed in $\Theta(nm)$.

Chain-Matrix Multiplication: (Section 16.1) Given a chain of matrices, determine the optimum order in which to multiply them. This is an important problem, because many DP formulations are based on deriving an optimum binary tree given a set of leaves.

NP-completeness: (Chapt 36.)

Basic concepts: Decision problems, polynomial time, the class P, certificates and the class NP, polynomial time reductions, NP-completeness.

NP-completeness reductions: We showed that to prove that a problem is NP-complete you need to show (1) that it is in NP (by showing that it is possible to verify correctness if the answer is "yes") and (2) show that some known NP-complete problem can be reduced to your problem. Thus, if there was a polynomial time algorithm for your problem, then you could use it to solve a known NP-complete problem in polynomial time.

We showed how to reduce 3-coloring to clique cover, and how to reduce Hamiltonian cycle to Hamiltonian path.