Homework 6

Jeremy Wright
CSE598 - Analysis of Algorithms

November 13, 2013

Problem 6.13. Consider the chain of trades as a linked list.

Consider all the suffixes of this list

define a recurrence $r(i, j)$ where i is the starting index of the trade list, and j is the ending index of the trade list. $r(i, j)$ returns the trade ratio for that series of trades. We want to maximise these ratios, so lets "carefully" try all suffixes.

1. Subproblems. suffixes of trades[i:] (in python notation).
number of subproblems is n in the number of trades.

2. Guess where to start the trade cycle.
number of choices (n -1) since the first trade must be at least after the first stock.

3. recurrence

$$cycle(i) = max\{cycle([i:]) + r(i, j) \text{for i in range}(i + 1, n + 1)\} \quad (1)$$

At each stock the cycle is measured, and the maximum is returned. This measures the cycle at ever interval of stocks. Since we have over lapping subproblems if we memoize we can ignore the cose of the recursive calls, and this results in $\theta(n)$ sub problems since there are $\theta(n)$ stocks.

4. Total running time is time per subproblem * number of problems = $\theta(n^2)$

5. The original subproblem is simple the maxium cycle from the first stock $cycle(0)$

Problem 6.22. The number of shortest paths between a node s and t is at most the in-degree of t since any path to t, but include an edge incomming to t. Thus if we compute the single source shortest path, and guess on the incomming edge that it is part of the final solution.

1. Sub problems. prefixes of paths from t back to s. # number of subproblems is E in the number of edges in the graph. Consider the case where there are only 2 nodes in the graph s, and t, and all edges are the same length. In this case, all paths are shortest paths, thus the number of sub problems is E, all edges.

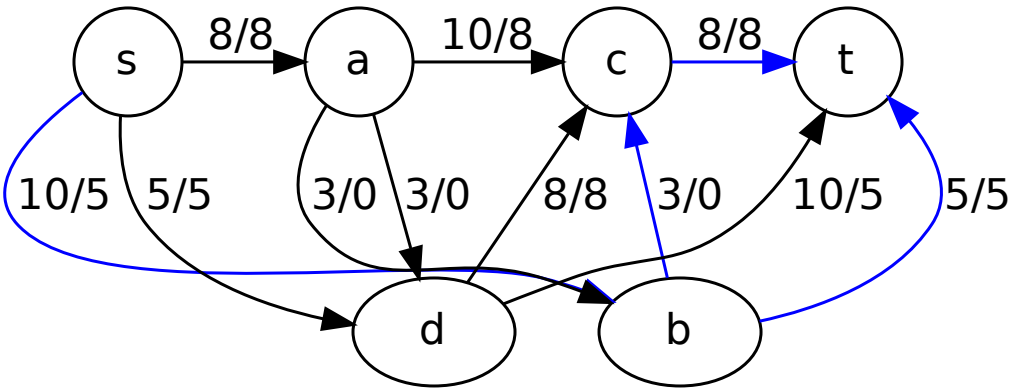
2. Guess which edge is part of the shortest path (could be all of them). # number of choices is the indegree of t, since the shortest path to t must include t.

3. recurrence

$$shortestPaths(p) = min\{shortestPaths([:p]) + weight(p) \text{for p in range}(t_{edges} + 1)\} \quad (2)$$

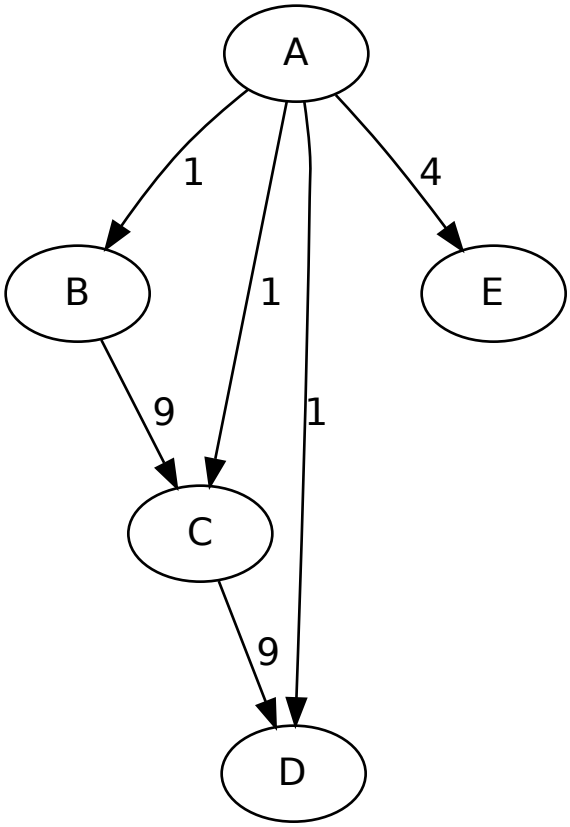
4. Total running time = $\theta(indegree(t) \cdot E$. 5. Our recurrence will find all shortest paths to t. We then use parent pointers to add each minimum included in the solution, to push on a list. Once the algorithm completes we simple query the length of the parent pointer list, which takes $\theta(1)$ time. Thus the running time is bounded by $\theta(indegree(t) \cdot E$

Problem 7.3. The flow of this graph is 13. This is the maximum flow since the source edges are all at capacity. The critical path is highlighted in blue.

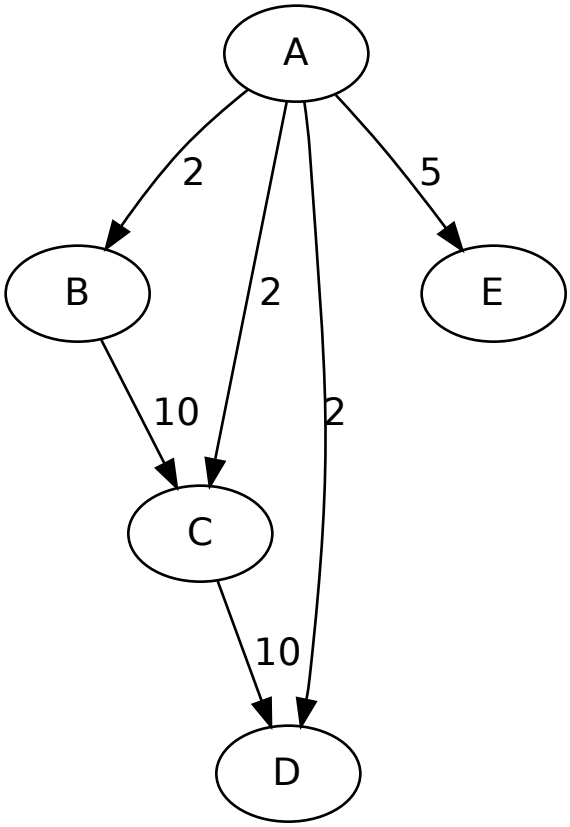


The minimum cut includes the edges $\{(1,t),(a,c),(a,b),(s,b),(s,d)\} = 12$

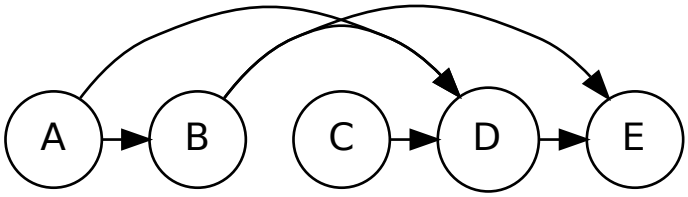
Problem 7.5. The minimum cut in this graph is $\{(A,B),(A,C),(A,D) = 3\}$



Adding 1 to every edge moves the minimum edge to $(A, E) = 5$

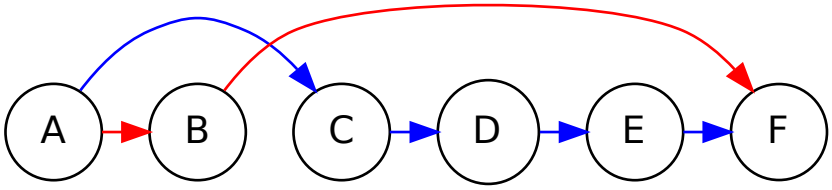


Problem 6.3. Given an ordered graph G, find the length of the longest path that begins at A and ends at E.



The given algorithm is a greedy algorithm and always chooses the nearest node hoping to

get to the end as late as possible. However in the example below, the blue path is the longest path, with length 4, wile the red path is length 2. The greedy algorthm does not see beyond node 2, and incorrectly follows the red path.



Dynamic programming can help here by "carefully" evaulating all possible paths. First a few constraints. The graph must be acyclic,, and have a topolocial sort, which by definition it does.

Next we use a 5 step process.

1. Define sub problems.
2. Guess a metric to minimize/maximize.
3. Relate the sub problems (recurrence)
4. Solve the recurrence, and memoize the algorithm.
5. Recombine the subproblems (verify the original problem gets solved.)

1. Define sub problems. Consider the incomming edges to the final node. The longest path must be one of those edges, so use dynamic programming to carefully try them all.

2. Guess the last edge comming into the final node v. Next recurrsively find the longest path to u and add the longer path to v.

3. Relate sub problems. Define a function $\delta(s, v)$ which is the length of the edge from the path from s to v.

$$\delta(s, v) = max\{\delta(s, u) + (u \to v).\text{weight } \forall \text{ Edges} = \text{u}\} \tag{3}$$

Time per subproblem is $\delta(s, v) = indegree(v)$. Total time

$$\sum_{v \in G} indegree(v) = \theta(E + V) \tag{4}$$

Problem 6.5. Consider the string ilovemywife.

consider all the suffixes of this string.

define a recurrence $q(s, f)$ where s is the starting index, and f is the closed finish index. $q(s, f)$ returns the quality for that string. we want to maximze the quality of the sentence. so lets "carefully" try all suffixes.

1. subproblems. suffixes of words[i:] (in python notation).
number of subproblems is n in the number of letters

2. guess where to start the second word.
number of choices (n - 1) since the first letter must be at the first word.

3. recurrence

$$word(i) = max\{word([j :]) + q(i, j) \text{ for j in range(i+1, n + 1)}\} \tag{5}$$

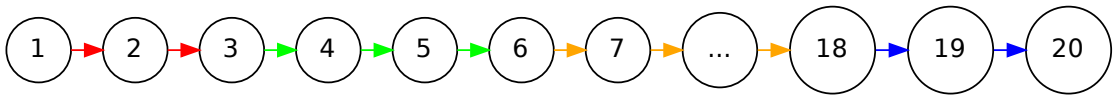
at each letter, the quality of $[i : j]$ is added to the recursive call. the suffix is then split at j, and recursively called. this measures the quality of all n suffixes of the word. since we have overlapping subproblems, if we memoize, we can ignore the cost of the recursive calls and this results in $\theta(n)$ since there are n letters in the set of suffixes.

4. total running time is time / subproblem * number of problems = $\theta(n^2)$.

5. original subproblem is the maximum quality starting from the first character i.e. $word(0)$.

Problem 6.20. From the problem we can see that the function f_i is a nondecreasing function, thus it has a topological sort ∴ we can apply dynamic programming since our problem is acyclic.

The goal is then to find a collection of contagious hours to spend on each class as show in the DAG below.



It is important to note that the hours are indivisible integers, since real numbers would make the dynamic program infinite. Minimally we have to divide the hours into some discrete steps to draw the DAG. Without a DAG, dynamic programming will not result in a correct algorithm.

This problem is now similar to problem 6.5 such that we are finding the divisions of contigious sequences.

1. Subproblems. Suffixes of the list of hours. Each list of hours will be fed to the function f_i and we want to maximize our grade at each point.

2. We'll start on the first hour, and try every set of hours after that. This results in the number of subproblems being $\theta(n)$

3. Recurrence

$$hours(i) = max\{hours([j:]) + f((i-j)) \text{ for j in range(i+1, n + 1)}\} \tag{6}$$

At each hour, we choose the hours until the end of the list, and calculate the expected grade for that set of hours. We then recurse on the hours left in the set. Since we have overlapping sub problems we don't have the count all the recursive calls, $\therefore$ we expect the time per subproblem to be $\theta(n)$.

4. Total running time is $\frac{time}{subproblem} \cdot$ number of subproblems $= \theta(n) \cdot \theta(n) = \theta(n^2)$

5. We don't spend any extra time combining the subproblems, thus we meet this with $\theta(n^2)$