CS 624 - HW 5

Max Weiss

2 March 2020

1. Let me write $A(x)$ for the value returned by the algorithm as applied to node x. Let's define the *cost* $C(x)$ of node x by

$$C(x) = \min_{p \in \pi(x)} W(p) \tag{1}$$

where $W(p)$ is the sum of the weights of the edges in path p, and $\pi(x)$ is the set of paths from x to the end node e. Let's stipulate that $C(x)$ is infinite if $\pi(x)$ is empty. Now the claim to be proven is that $C(x) = A(x)$ for all nodes x.

Let's say that the *level* of x is the maximum length of a path from x. Also, say that the *e-level* of node x is the number of edges in the longest path from x to the end node e (if no such path exists, say that the e-level is infinite).

The proof splits into cases. Suppose that the e-level of x is infinite. Then there is no path from x to e, so that $\pi(x)$ is empty, and $C(x)$ must be infinite. I will argue by induction on the level of x that $A(x)$ must be infinite as well. If the level is 0, then x has no children, so $A(x)$ is infinite. So suppose it to hold for all nodes of level at most n, and suppose x has level $n + 1$. All children of x have level at most n, and of course there cannot be a path from a child to e. By induction hypothesis, $A(y)$ is infinite for all children y of x. Inspecting the code reveals that $A(x)$ is then the minimum of a set of infinite values, which is infinite.

Let's now suppose that the e-level of x is finite. I will handle this by induction too. If x has e-level 0, then $x = e$, so that $A(x) = 0$. The empty path is a path from e to e and it has cost 0, and therefore $C(x) \leq 0$. Hence, $C(x) \leq A(x)$. Conversely, all edges have nonnegative weights, and so $C(x) \geq 0$ and so $C(x) \geq A(x)$ as well.

For the inductive step, suppose that $A(y) = C(y)$ for all nodes y of e-level at most n, and let x have e-level $n + 1$. Let me write $w(x, y)$ for the weight of the edge from x to y if any exists; and let me write $E(x)$ for the set of all nodes y

such that there is an edge from x to y. I now contend that

$$\begin{aligned}
A(x) &= \min\{w(x,y) + A(y) : y \in E(x)\} && (2)\\
&= \min\{w(x,y) + C(y) : y \in E(x)\} && (3)\\
&= \min\{w(x,y) + \min_{p\in\pi(y)} W(p) \ : \ y \in E(x)\} && (4)\\
&= \min\{w(x,y) + W(p) : p \in \pi(y) \text{ for some } y \in E(x)\} && (5)\\
&= \min\{W(x \frown p) : p \in \pi(y) \text{ for some } y \in E(x)\} && (6)\\
&= \min\{W(p) : p \in \pi(x)\} && (7)\\
&= C(x) && (8)
\end{aligned}$$

Step (2) is clear by inspection of the code. Noting that if $y \in E(x)$, then y has e-level at most n, it follows that $A(y) = C(y)$ by induction hypothesis; and this justifies (3). Step (4) follows from the definition (1) of C. Step (5) is the crucial insight of the algorithm. In one direction, the set minimized in (5) is a subset of the set minimized in (4); the other direction follows from

$$w(x,y) + \min_{q\in\pi(y)} W(q) \le w(x,y) + W(p)$$

for all $p \in \pi(y)$. Step (6) is the definition of W; step (7) follows from the fact that the e-level $n+1$ of x is finite and positive; and step (8) is the definition of C.

2. The problem here is to analyze the number of maximal paths in the Kth "diamond" graph $G = G_K$, which has $2K+1$ rows.

Consider the larger diamond subgraph G_{2K}, and in particular, let G' be its maximal triangular subgraph, which results by dropping all rows of index $\ge K$. It will be easier to work with G' than G (though the path counts for common nodes are clearly the same).

Let's now write $p(n,k)$ for the number of paths to kth node of the nth row of G'. We find that

$$p(n,k) = \begin{cases} p(n-1,k-1) + p(n-1,k) \text{ for } 0 < k < n, \text{ and} \\ 1, \text{ otherwise} \end{cases} \qquad (9)$$

for all n with $0 < n \le 2K$.

I'll now argue that $p(n,k) = \binom{n}{k}$. This is clear for $k = n$ or $k = 0$. Otherwise, I'll argue by induction on n that it holds for all k such that $0 < k < n$. It is clear for $n = 1$. So suppose that $p(n, k) = \binom{n}{k}$ holds whenever $0 < k < n$. Then

$$\begin{aligned}
p(n, k+1) &= p(n, k) + p(n, k-1) \\
&= \binom{n}{k} + \binom{n}{k-1} \\
&= \frac{n!}{(n-k)!k!} + \frac{n!}{(n-k+1)!(k-1)!} \\
&= \frac{(n+1)!}{(n+1-k)!k!} \\
&= \binom{n+1}{k}
\end{aligned}$$

provided $k < n$. In the remaining case of $k = n$,

$$\begin{aligned}
p(n+1, k) &= p(n, n) + p(n, n-1) \\
&= 1 + \binom{n}{n-1} \\
&= n + 1 \\
&= \binom{n+1}{k}
\end{aligned}$$

where to obtain the second summand in the second line we again use the induction hypothesis. Thus $p(n+1, k) = \binom{n+1}{k}$ whenever $0 < k < n+1$ which completes the induction step.

Applying this result to G_4: the bottom node of G_4 is the 4th node of the 8th row of G_8. So, the number of paths to the bottom node of G is $p(8, 4) = 70$.

Let's now return to general case of G_K. Its bottom node is the Kth node of the $2K$th row. Let's write $q(K) = p(2K, K)$. Then the number of paths to the bottom node will be given by

$$q(K) = \binom{2K}{K} = \frac{(2K)!}{K!(2K-K)!} = \frac{(2K)!}{K!K!} \tag{10}$$

To bound the growth rate of q, let's take the version of Stirling's formula

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n \left(1 + \Theta\left(\frac{1}{n}\right)\right)$$

plug it in for the last expression in (10), and simplify:

$$\begin{aligned}
q(K) &= \frac{\sqrt{2\pi 2K}(\frac{2K}{e})^{2K}\left(1 + \Theta\left(\frac{1}{2K}\right)\right)}{2\pi K(\frac{K}{e})^{2K}\left(1 + \Theta\left(\frac{1}{K}\right)\right)^2} \\
&= \frac{2^{2K}\left(1 + \Theta\left(\frac{1}{2K}\right)\right)}{\sqrt{\pi K}\left(1 + \Theta\left(\frac{1}{K}\right)\right)^2} \qquad (11)
\end{aligned}$$

The formula (11) contains a ratio of error terms

$$h(K) = \frac{1 + \Theta(\frac{1}{2K})}{(1 + \Theta(\frac{1}{K}))^2}.$$

By definition of Θ, it follows that there are constants c, d, e such that $h(K)$ is asymptotically bounded by

$$\frac{1 + \frac{c}{K}}{(1 + \frac{d}{K})(1 + \frac{e}{K})} = \frac{K(K + C)}{(K + d)(K + e)}$$

The latter is asymptotically bounded by 1, because it is the product of two ratios which are each so bounded.

It follows that

$$q(K) < \pi^{-1} \frac{2^{2K}}{\sqrt{K}}$$

for all sufficiently large K, so $q \in O\left(\frac{2^{2K}}{\sqrt{K}}\right)$, as desired.

Note for questions 3-5. These questions involve a pair of algorithms with a common pattern: each brings a "worker" function which expresses the meat of the algorithm, but then wraps this into a mutual recursion with a "guard". The guard is supposed to prevent the worker from being called more than once, by storing previously computed results in a memo.

```
def memoized(f, args):
    memo = HashTable()
    return guard(args)

    def guard(args):
        if n not in memo:
            memo[n] = f(guard, args)
        return memo[args]
```

Several conditions must be in place for this wrapper to implement memoization. Of course the ARGS must be hashable. More importantly, the supplied worker must not recurse on itself directly but only on its guarded form. Finally, the worker must call the guarded form only on arguments which are in some sense of lower "rank". More precisely, say that a relation $<$ is *wellfounded* if there is no infinite chain $x_1 > x_2 > x_3 \cdots$. Then, the applicability condition for the wrapper is that

- there exists a wellfounded relation $<$ such that ARGS1 $<$ ARGS for every invocation of the guarded worker $g($ARGS1$)$ within an execution of $f(g,$ ARGS$)$.

Under this condition on f, I will now argue for the key memoization property: within any execution of MEMOIZED(f, ARGS), the function f is called on a given set of arguments at most once.

Note that each execution of f(GUARD, ARGS) must occur within an execution of GUARD(ARGS). Let t_1, t_2 be the invocation and return of the smallest execution of GUARD(ARGS) which includes the first execution of f(GUARD, ARGS). Within t_1 and t_2, there is an execution E of f(GUARD, ARGS) which includes all other calls to f within t_1, t_2. By the applicability condition, these must in turn be included in calls to GUARD(ARGS$'$) such that ARGS$' <$ ARGS, and must therefore act on such arguments of lower rank than ARGS. So no other call to f(GUARD, ARGS) gets issued between t_1 and t_2. But always after t_2, the node n is stored within MEMO, and so after t_2, no calls to GUARD(ARGS) can generate a call to f(GUARD, ARGS) either.

3. Let me rephrase the memoized version of the cost algorithm using the MEMOIZED wrapper above.

```
def cost(node):
    def worker(g, n):
        if n is end:
             return 0
        min_dist = infinity
        foreach child of n:
            new_dist = weight(n, child) + g(child)
            min_dist = min(min_dist, new_dist)
        return min_dist
    return memoized(worker, node)
```

I will now work to evaluate the running time of this algorithm. Let's begin by verifying the condition of applicability of the memoization. Write $x < y$ to mean that node x is a strict descendant of y (i.e., x is a descendant of y and $x \neq y$). From homework 4 (as I did it anyway) we know that $<$ is a partial order, so that it has a minimal element on every finite set, and it must (using strictness) therefore be wellfounded on finite trees. Since g is invoked only on children (which are strict descendants) of the input argument, this confirms the applicability condition. So we can conclude that within any execution of COST, the WORKER subroutine is called at most once per node.

I'll now count the number of executions of GUARD. As for the root node, every call to GUARD other than the initial one operates on a node of lower level, so GUARD is called on the root node only once. So suppose n is not the root. Then each call to GUARD(n) is triggered by a pass through the the main loop, in some WORKER(n') call, which corresponds to an edge from n' to n. Since WORKER(n') can be executed only once, it follows that there can be just as many invocations of GUARD(n) as there are edges into n.

Besides their subcalls to each other, each execution of WORKER or of GUARD makes only a constant number c_h or c_m of other operations (array or hashtable lookup, hashtable insertion, assignment, addition, etc). So where $c = \max(c_h, c_m)$, the runtime of the algorithm sketched above is bounded by $c(|E| + |V|)$. It follows that the runtime is $O(|E| + |V|)$.

4. Here is some pseudo-code for a memoized calculation of the longest common subsequence.

```
def lcs(X, Y):
    def worker(g, i, j):
        if i == 0 or j == 0:
            return 0
        if X[i] == Y[j]:
            return g(i-1, j-1) + 1
        else:
            return max(g(i-1, j), g(i, j-1))
    return memoized(worker, i, j)
```

5. As before, let's begin by extracting the key property of the memoization wrapper. Write $(i', j') < (i, j)$ iff $i' + j' < i + j$. For nonnegative integers, $<$ is wellfounded. Furthermore, within WORKER, it is only on positive integers that the subroutine g is invoked. Therefore, the function WORKER can be invoked on a given pair i, j only once.

It follows that the number of calls to WORKER can be at most mn, where m is the length of X and n is the length of Y.

Each (noninitial) call to GUARD(i, j) occurs within a call either of WORKER$(i+1, j)$ or of WORKER$(i, j+1)$ (or both). And in either execution, it can occur at most once. Furthermore, we saw above that each of those WORKER calls can occur at most once. So, GUARD(i, j) can be executed at most twice for each pair i, j.

The calls to GUARD and to WORKER are, then, each proportional in number to mn. Since besides their recursive mutual subcalls they employ only constant time operations, it follows that the algorithm is $O(mn)$.