CS 331: Algorithms and Complexity (Spring 2024)

Unique numbers: 50930/50935/50940/50945

Handout for Test 2

Recurrences and Master's Theorem

Solve the recurrences below by Master's Theorem where possible. If Master's theorem does not apply, mention why.

1. $T(n) = \sqrt{2}T(n/2) + \log n \rightarrow T(n) = \Theta(\sqrt{n})(Case\ 1)$
2. $T(n) = 3T(n/3) + n/2 \rightarrow T(n) = \Theta(n \log n)$
3. $T(n) = 2T(n/2) + n/\log n \rightarrow$ Master's theorem does not apply because of non-polynomial difference.
4. $T(n) = 64T(n/8) - n^2 \log n \rightarrow$ Does not apply because $f(n)$ is not positive.
5. $T(n) = 2^n T(n/2) + n^n \rightarrow$ Does not apply because a is not constant.
6. $T(n) = 2T(n/4) + n^{0.51} \rightarrow T(n) = \Theta(n^{0.51})(Case\ 3)$

Solve the recurrence below by manually unrolling it. Remember to provide a proof of your solution.

1. $T(n) = 2T(n-1) + 1$, $T(1) = 1$

Solution:

The below table shows how the given recurrence can be manually unrolled.

Level	**Size of problem**	**Time taken**	**Number of problems**
0	n	1	1
1	$n-1$	1	2
2	$n-2$	1	2^2
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$k-1$	$n-(k-1)$	1	2^{k-1}
k	$n-k$	1	2^k

When the size of the problem is 1, from the pattern shown in the table, it can be deduced that there will be 2^{n-1} problems. Let T_i represent the time accounted for by the i-th level of the recurrence tree. Then,

$$\begin{aligned} T_i &= 1 \cdot 2^i \\ &= 2^i \end{aligned}$$

Clearly,

$$T(n) = \sum_{i=0}^{n-1} T_i$$

Thus,

$$\begin{aligned} T(n) &= \sum_{i=0}^{n-1} 2^i \\ &= \frac{1 \cdot (2^n - 1)}{2 - 1} \\ &= 2^n - 1 \end{aligned}$$

The proof of the above formula by induction can be seen below:
Base Case: Substituting $n = 1$ into the above formula gives: $T(1) = 2^1 - 1 = 2 - 1 = 1$. Thus, the base case has been proved correct.
Induction Hypothesis: Assume the above formula works for $n = k$.
Induction Step: In this step, the formula will be proved for $n = k + 1$.

$$\begin{aligned} T(k+1) &= 2T(k) + 1 && \text{Given recurrence relation} \\ &= 2 \cdot (2^k - 1) + 1 && \text{Induction hypothesis} \\ &= 2^{k+1} - 2 + 1 \\ &= 2^{k+1} - 1 \end{aligned}$$

Hence, the formula has been proved right by induction.

Matrix Multiplication

Recall how matrix multiplication works: if we have a matrix A with n rows and k columns ($n \times k$), and a matrix B with k rows and m columns ($k \times m$), their product is defined by multiplying each row of A by each column of B, and their product is of size $n \times m$. This costs roughly $2mnk$ operations.
In the following equations, $\times$ is a placeholder to indicate some element of a matrix (we don't care about the contents, just the shape).

$$\begin{pmatrix} \times & \times & \times \\ \times & \times & \times \end{pmatrix} \begin{pmatrix} \times & \times \\ \times & \times \\ \times & \times \\ \times & \times \end{pmatrix} = \begin{pmatrix} \times & \times \\ \times & \times \end{pmatrix}$$

Matrix multiplication is associative: if we have matrices A,B, and C, $(AB)C = A(BC)$. However, even if the answers are the same, the *cost* is not. Consider the following example: A is 3×8, B is 8×2, and C is 2×5. ABC is 3×5.
If we do $(AB)C$, we need to calculate $A \cdot B$ and then multiply that result with C. The first multiplication takes $3 \cdot 8 \cdot 2 = 48$ operations and creates a 3×2 matrix. Multiplying this with C takes $3 \cdot 2 \cdot 5 = 30$ operations. This is a total of $48 + 30 = 78$ operations.
However, if we do $A(BC)$, we first multiply BC, at a cost of $8 \cdot 2 \cdot 5 = 80$ operations, creating an 8×5 matrix. We then multiply A with this, at a cost of $3 \cdot 8 \cdot 5 = 120$ operations. The total cost of this is 200 operations—more than twice as much work to get to the same answer!
This effect gets even worse when given a long sequence of matrices to multiply together. Sometimes, it is worth it to take some time to figure out what the optimal matrix multiplication order is, before carrying out the actual multiplicaton.
You are given a list of tuples. Each element of the list represents one matrix, and the goal is to multiply all of them together at the lowest possible cost. The above example would be given as $[(3,8), (8,2), (2,5)]$.
Create a dynamic programming algorithm that appropriately orders the matrix multiplication so that the cost of multiplying them all together is as low as possible.

Solution:

First, we transform the input. Since the internal coordinate is redundant (if we have $[(3,8),(\_,2)]$, we know that the blank must be 2), we can make a single list containing de-duplicated sizes. In this scheme, $[\,(3,8)\,,\ (8,2)\,,\ (2,5)\,]$ becomes $[3,8,2,5]$.

We then look at the recurrence that defines this problem. If we have matrices A_1, A_2, ..., A_n, then we can choose to multiply the first k matrices, the remaining matrices, and then put them together. In other words, we can look at the following:

$$(A_1A_2A_3\dots A_k)(A_{k+1}A_{k+2}\dots A_n)$$

In order to find the optimal way to parenthesize these matrices, we need to find the optimal cost of the left half, the optimal cost of the right half, and the cost of putting them together.
Let D be the array that holds the dimensions (that we defined above). The cost of a matrix chain starting at A_i, ending at A_j, and split at A_k as shown above is

$$C(i,k,j) = C[i,k] + C[k,j] + D[i]D[k]D[j]$$

If we try all possible k, then we can find the optimal value.
This suggests the following algorithm:
First, we transform the list of matrix sizes into a single list containing no duplicate values.
Then, we use the following function to compute the minimum:

```
Function MMult(i,j,mlist,M):
  if i+1 == j:
     return 0
  else if M[i,j] exists:
     return M[i,j]
  else:
    minsplit = infinity
    for k from i+1 to j-1:
      try_split_at_k = mlist[i] * mlist[k] * mlist[j]
                       + MMult(i,k,mlist,M) + MMult(k,j,mlist,M)
      minsplit = min(minsplit, try_split_at_k)

   M[i,j] = minsplit
    return minsplit
```

The Edit Distance between two words

You are given two strings, $A[1..n]$ and $B[1..m]$. Given that the cost of inserting, deleting and replacing a character are C_i, C_d, and C_r respectively, write a Dynamic Programming Algorithm to find the minimum cost of transforming A to B.

Solution:

Let $T(i,j)$ be the minimum cost of transforming $A[1..i]$ into $B[1..j]$. Thus, the required solution would be $T(n,m)$. The recurrence below successfully calculates $T(i,j)$ from values of T that have been calculated earlier. Note that the recurrence is divided into three parts.

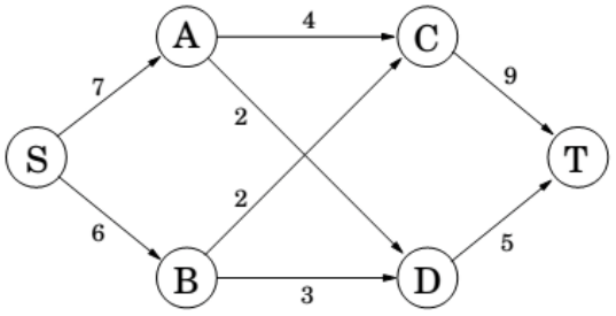
- $A[1..i]$ can be transformed into $B[1..j]$, by first deleting the i-th character in the string. In that case, the minimum cost would be $C_d + T(i-1,j)$.

- $A[1..i]$ can also be transformed into $B[1..j]$, by first transforming it into $B[1..j-1]$ and then inserting the j-th character. In that case, the minimum cost would be $T(i, j-1) + C_i$.
- Finally, if $A[i]$ equals $B[j]$, then the last characters can be ignored, and the minimum cost of transforming $A[1..i]$ into $B[1..j]$ would just be $T(i-1, j-1)$. On the other hand, if the last characters are not equal, then you could replace $A[i]$ to match $B[j]$, and focus on transforming the first $i-1$ characters of A. In that case, the minimum cost would be $T(i-1, j-1) + C_r$.

With the correct base cases, the above recurrence is correct because any sequence of replacements, deletions, and insertions, can be simulated by one of the above choices. This is assuming characters are inserted or replaced only to match a character in B. The base cases are defined as follows. $T(i, 0)$ should be initialized to $C_d \cdot i$ for $i \in (0, 1, \ldots, n)$, and $T(0, j)$ should be initialized to $C_i \cdot j$ for $j \in (1, 2, \ldots, m)$.

Network Flow

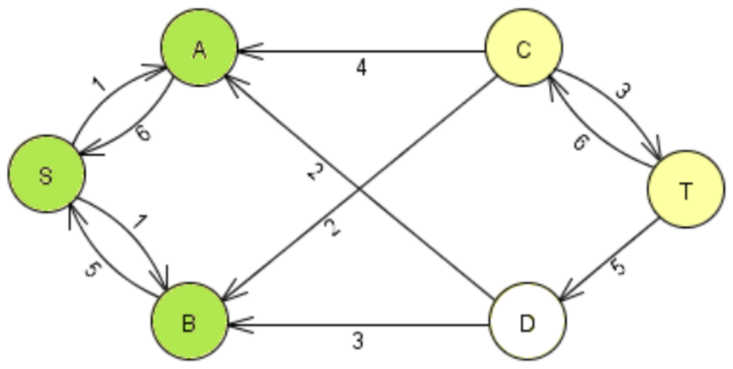
Consider the following flow network. (the numbers associated with the edges are the edge capacities).

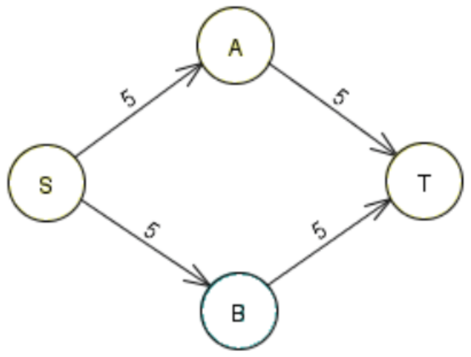


1. Find the maximum flow f and a minimum cut.
2. Draw the final residual graph G_f (along with its edge capacities). In this residual network, mark the vertices reachable from S and the vertices from which T is reachable.
3. An edge of a network is called a *bottleneck* edge if increasing its capacity results in an increase in the maximum flow. List all bottleneck edges in the above network.
4. Give a very simple example (containing at most four vertices) of a network which has no bottleneck edges.
5. Give an efficient algorithm to identify all bottleneck edges in a network. (Hint: Execute a known network flow algorithm (e.g., Ford-Fulkerson), and then examine the residual graph.)

Solution:

1. The max flow value is 11. A min cut is $L = \{S, A, B\}, R = \{C, D, T\}$.
2. The residual graph is shown below. The vertices reachable from S are A, B and vertices from which T is reachable is C.
3. The edges (A, C) and (B, C) are bottleneck edges.
4. The following graph has no bottleneck edges.





5. Run the network flow algorithm and find the final residual network. For each edge (u, v) in the original graph, check in the residual network whether the vertex u of the edge is reachable from S and the vertex v of the edge can reach T. We can use a depth first search algorithm to check the reachability. If the paths from S to u and v to T exist, then edge (u, v) is a bottleneck edge. This is true because if we increase the capacity of the edge (u, v), then we get an edge (u, v) in the residual network. Using this edge, we can find an augmenting path from S to T resulting in an increase of flow value.

   Proof of correctness:

   We know that the existence of a path between vertices u and v in the residual graph indicates that flow can be augmented along that path. i.e., if vertex v is reachable from u, then the flow from u to v can be augmented. We use this property in this proof.

   Consider any edge (u, v) found by the algorithm such that u is reachable from s and t is reachable from v. Since the algorithm only checks for those edges that were present in the original graph, we know that edge (u, v) had some initial capacity that is now being fully used (as the edge does not exist in the residual graph). By the property stated earlier, there is some augmenting flow possible from s to u and v to t. Thus, increasing the capacity of edge (u, v) would introduce a new edge (u, v) in the residual graph with some non-zero capacity, making t reachable from s through edge (u, v). This new path from s to t is an augmenting flow, along which flow can be increased. Thus, (u, v) is a bottleneck edge.

   Consider the case when either u is not reachable from s or t is not reachable from v, but edge (u, v) is being used at maximum capacity. In this case, increasing the capacity of edge (u, v) will not increase flow from s to t, and hence (u, v) is not a bottleneck edge and is not found by the algorithm.

Thus, the algorithm correctly finds only and all bottleneck edges in the graph.

Running time:

We can find all the vertices reachable from S in the residual graph in $O(|V| + |E|)$ time by a traversal algorithm like DFS. To find all the vertices that can reach T, we need only reverse the graph, which takes $(|V| + |E|)$ time, and then find all vertices reachable from T in the reversed graph, which also takes $O(|V| + |E|)$ time. Then checking each edge (u, v) to find those edges where u is in the set of vertices reachable from S and v is in the set of vertices that can reach T takes $O(|E|)$ time.

Therefore, this algorithm takes $O(|V| + |E|)$, i.e., linear time, overall, in addition to the running time of the network flow algorithm for generating the final residual graph.

The Escape Problem (Network Flow)

We define the *Escape Problem* as follows. We are given a directed graph $G = (V, E)$ (picture a network of roads). A certain collection of nodes $X \subset V$ are designated as populated nodes, and a certain other collection $S \subset V$ are designated as safe nodes. (Assume that X and S are disjoint.) In case of an emergency, we want evacuation routes from the populated nodes to the safe nodes. A set of evacuation routes is defined as a set of paths in G so that (i) each node in X is the tail of one path, (ii) the last node on each path lies in S, and (iii) the paths do not share any edges. Such a set of paths gives a way for the occupants of the populated nodes to "escape" to S, without overly congesting any edge in G.

1. Given G, X, and S, show how to decide in polynomial time whether such a set of evacuation routes exists.

2. Suppose we have exactly the same problem as in (1), but we want to enforce an even stronger version of the "no congestion" condition (iii). Thus we change (iii) to say "the paths do not share any nodes.". With this new condition, show how to decide in polynomial time whether such a set of evacuation routes exists.

3. Provide an example digraph G, and define X, and S, such that the answer is yes to the question in (1) but no to the question in (2).

Solution:

1. Add a source node s to the graph G. From s, draw an edge to every node in X. This ensures that condition (i) is satisfied. Assign a capacity of 1 to every edge currently in the graph. This ensures that condition (iii) is satisfied. Now add a sink node t to G. From every node in S, add an edge to t. This ensures that condition (ii) is satisfied. Assign each of the newly added edges a capacity of $|X|$. This ensures that any number of paths can end at a safe node.

2. First follow the steps in above solution. Then transform each vertex v in $V - s, t - X - S$ as follows: replace v with v_{in} and v_{out} such that all the edges arriving at v now arrive at v_{in}, and all the edges leaving v, now leave from v_{out}. Also, add an edge from v_{in} to v_{out} and assign this edge a capacity of 1. This ensures that only one path can go through v.