1 Minimum Edit Distance

Definition 1 The minimum edit distance (MED) problem is given a character set $\Sigma$ and two sequences of characters $S = \Sigma^*$ and $T = \Sigma^*$ to determine the minimum number of insertions and deletions of single characters required to transform $S$ to $T$.

For example if we started with the sequence

$$S = \langle A, B, C, A, D, A \rangle$$

we could convert it to

$$T = \langle A, B, A, D, C \rangle$$

with 3 edits (delete the C, delete the last A and insert a C). This is the best that can be done so the minimum edit distance is 3.

The MED problem is an important problem that has many applications. For example in version control systems such as git or svn when you update a file and commit it, the system does not store the new version but instead only stores the “difference” from the previous version\(^1\). This is important since often the user is only making small changes and it would be wasteful to store the whole file. Variants of the minimum edit distance problem are use to find this “difference”. Edit distance can also be used to reduce communication costs by only communicating the differences from a previous version. It turns out that edit-distance is also closely related to approximate matching of genome sequences.

One might consider a greedy method that scans the sequence finding the first difference, fixing it and then moving on. Unfortunately no simple greedy method is known to work. The problem is that there can be multiple ways to fix the error—we can either delete the offending character, or insert a new one. In the example above when we get to the $C$ in $S$ we could either delete $C$ or insert an $A$. If we greedily pick the wrong way to fix it, we might not end up with an optimal solution. Again in the example, if you inserted an $A$, then more than two more edits will be required.

However, considering the greedy solution gives a good hint of how to find a correct solution. In particular when we get to the $C$ in our example there were exactly two possible ways to fix it—deleting $C$ or inserting $A$. This leads to the following algorithm.

\(^1\)Alternatively it might store the new version, but use the difference to encode the old version.
**EditDist**$(S, T, i, j)$

1. if $S$ is empty
   2. then return $|T|$
2. if $T$ is empty
   3. then return $|S|$
3. if $S[i] = T[j]$
   4. then return **EditDist**$(S, T, i + 1, j + 1)$
4. else return $1 + \min\{**EditDist**(S, T, i, j + 1), **EditDist**(S, T, i + 1, j)\}$

In the first base case where $T$ is empty we need to delete all of $S$ to generate an empty string requiring $|S|$ insertions. In the second base case where $S$ is empty we need to insert all of $T$, requiring $|T|$ insertions. If neither is empty we can just skip them and make a recursive call on the rest of the sequences. If they are different then we need to consider the two cases. The first case (**EditDist**$(S, T')$) corresponds to inserting the value $t = T[1]$. We pay one edit for the insertion and then need to match up $S$ (which all remains) with the tail of $T$ (we have already matched up the head $t$ with the character we inserted). The second case (**EditDist**(S', T)) corresponds to deleting the value $s = S[1]$. We pay one edit for the deletion and then need to match up the tail of $S$ (the head has been deleted) with all of $T$.

### 1.1 Shared Subproblems

If we ran the code recursively we would end up with an algorithm that takes exponential work. In particular the recursion tree is binary and has a depth that is linear in the size of $S$ and $T$. However, there is significant sharing going on. Let’s draw out this tree:

```
MED({A,B,C},{D,B,C})
 /    \  \\
MED({B,C},{D,B,C}) MED({A,B,C},{B,C})
 /    \  /    \  \\
MED({C},{D,B,C}) MED({B,C},{B,C}) MED({A,B,C},{C})
 /    \  /    \  /    \\
MED({},{D,B,C}) MED({C},{B,C}) MED({C},{C}) MED({B,C},{C}) MED({A,B,C},{})
 /    \  /    \\
MED({},{B,C}) MED({C},{C}) MED({},{}) MED({C},{C}) MED({B,C},{})
 |     |    \\
MED({},{}) MED({},{}) MED({B,C},{}) MED({},{}) MED({},{})
```

Note that there are many nodes that are computed again and again. This is a hallmark of a problem that may be solvable by dynamic programming. What we would like to do is to store these solutions to sub-problems and use them over and over again.
2 Dynamic Programming

Dynamic programming is an algorithmic technique in which (1) one constructs the solution of a larger problem instance by composing solutions to smaller instances, and (2) the solution to each smaller instance can be used in multiple larger instances. Dynamic programming is thus one of the inductive algorithmic techniques we have are covering in this course. Recall that in all the inductive techniques an algorithm relies on putting together smaller parts to create a larger solution. The correctness then follows by induction on problem size. The beauty of such techniques is that the proof of correctness parallels the algorithmic structure.

So far the inductive approaches we have covered are divide-and-conquer, greedy method, and contraction. In the greedy method and contraction each instance makes use of only a single smaller instance. In the case of greedy algorithms the single instance was one smaller—e.g. solving the shortest superstring problem by merging two strings with the maximum overlap. In the case of contraction it is typically a constant fractions smaller—e.g. solving the scan problem by solving an instance of half the size.

In the case of divide-and-conquer, as with dynamic programming, we made use of multiple smaller instances to solve a single larger instance. However in D&C we have always assumed the solutions are solved independently and hence we have simply added up the work of each of the recursive calls. But what if two instances of size \(k\), for example, both need the solution to the same instance of size \(j < k\).

Although in the simple example sharing the results will only make at most a factor of two difference in work, in general sharing the results of subproblems can make an exponential difference in the work performed. The simplest, albeit not particularly useful, example is in calculating the Fibonacci numbers. As we have seen, one can easily write the recursive algorithm for Fibonacci:

```plaintext
FIB(n)
1 if n ≤ 1
2 then return 1
3 x ← spawn FIB(n – 1)
4 y ← FIB(n – 1)
5 sync
6 return x + y
```
but this will take exponential time in $n$. However, if the results from the instances are somehow shared then the algorithm only requires linear work:

```
  fib(5)
  |  \  
  |   fib(4)
  |    /  |
  fib(3) |  
  |  \  |
  |   fib(2)
  |    /  |
  fib(1) |  
     \  |
      fib(0)
```

It turns out there are many quite practical problems where sharing of sub results is useful and can make a significant differences in the work used to solve a problem. In the course we will go through several of these examples.

### 2.1 DAG structure of DP

With divide-and-conquer the composition of a problem instance in terms of smaller instances is typically described as a tree, and in particular the so called recursion tree. With dynamic programming the composition can instead be viewed as a Directed Acyclic Graph (DAG). Each vertex in the DAG corresponds to a problem instance and each edge goes from an instance of size $j$ to one of size $k > j$—i.e. we direct the edges (arcs) from smaller instances to the larger ones that use them. We direct them this way since the edges can be viewed as representing dependences between the source and destination (i.e. the source has to be calculated before the destination can be). The vertices with no in-edges are the base cases of our induction (instances that can be solved directly), and the final node(s) of the DAG (the vertex with no out-edges) is the instance we are trying to solve. More generally we might actually have multiple final nodes, although this can be converted into a DAG with a single root by adding a new vertex and an edge from each of the previous roots to this single new root.

Abstractly dynamic programming can therefore be best viewed as evaluating a DAG by propagating values from the leaves to the root and performing some calculation at each vertex based on the values of its in-neighbors. Based on this view calculating the work and span of a dynamic program is relatively straightforward, as we know from the DAG model. If it takes constant time

---

$^2$Note that “size” is used in an abstract sense and does not necessarily mean the size (e.g. number of bytes) of the input but rather any measure that can be used for inductive purposes.
to compute each vertex given its predecessors, then the work is the total number of vertices and the span is the length of the longest span.

More generally, if computing each vertex is itself a more complex parallel program, we can associate with each vertex a work and span required for that vertex. The overall work is then simply the sum of the work across the vertices. The overall span is the longest path in the DAG where the path length is the sum of the spans of the vertices along that path. Many dynamic programs have significant parallelism although some do not. For example consider the following DAG in which the work and span for each vertex is given.

```
(5,2)  (11,3)
  o ------>o---->
   ^      \
      \
(3,1) / v
  o -----o---o---o (1,1)
    
(2,2) (4,1)
```

This does $5 + 11 + 3 + 2 + 4 + 1 = 26$ units of work and has a span of $1 + 2 + 3 + 1 = 7$.

The challenging part of developing an algorithm for a problem based on dynamic programming is figuring out what DAG to use. The best way to do this, of course, is to think inductively—how can I solve an instance of a problem by composing the solution to smaller instances? Once an inductive solution is formulated you can think about whether the solutions can be shared and how much savings can be achieved by sharing. As with all algorithmic techniques, being able to come up with solutions takes practice. We note however that most problems that can be tackled with dynamic programming solutions are either optimization or decision problems. An optimization problem is one in which we are trying to find a solution that optimizes some criteria (e.g. finding a shortest path, or finding the longest contiguous subsequence sum). A decision problem is one in which we are trying to find if a solution exists.

## 2.2 Analysis of Work of Edit Distance

We can now place an upper bound on the number of vertices in our DAG for edit distance by bounding the number of distinct arguments. There can be at most $|S| + 1$ possible values of the first argument since in recursive calls we only use suffixes of the original $S$ and there are only $|S| + 1$ such suffixes (including the empty and complete suffixes). Similarly there can be at most $|T| + 1$ possible values for the second argument. Therefore the total number of possible distinct arguments to $\text{EDITDIST}$ on original strings $S$ and $T$ is $(|T| + 1)(|S| + 1) = O(|S||T|)$.

All together this gives us

$$W(\text{EDITDIST}(S, T)) = O(|S||T|)$$
2.3 Programming DP: Top Down and Bottom Up

Although abstractly dynamic programming can be viewed as a DAG, in practice we need to implement (code) the dynamic program. There are two common ways to do this, which are referred to as the *top-down* and *bottom-up* approaches. The top-down approach uses recursion as in divide-and-conquer but remembers solutions so if the algorithm tries to solve the same instance many times only the first call does the work and the rest just look up the solution. Storing solutions for reuse is called memoization. The bottom-up approach starts at the leaves of the DAG and typically processes the DAG in some form of level order traversal—for example, by processing all problems of size 1 and then 2 and then 3, and so on. Each approach has its advantages and disadvantages. Using the top-down approach (divide-and-conquer with memoization) can be quite elegant and can be more efficient in certain situations by evaluating fewer instances. Using the bottom-up approach (level order traversal of the DAG) can be easier to parallelize and also more space efficient. *It is important, however, to remember to first formulate the problem abstractly in terms of the inductive structure, then think about it in terms of how substructure is shared in a DAG, and only then worry about coding strategies.*

**Top-Down Dynamic Programming**

The top-down approach is based on running the recursive code basically as is but generating a mapping from input argument to solution as we proceed. This way when we come across the same argument a second time we can just look up the solution. This is called memoization, and the table used to map the arguments to solutions is called a memo table.

The tricky part of memoization is sometimes checking for equality of arguments since the arguments might not be simple values such as integers. In our examples so far, we could have passed entire sequences as arguments, but then if we compare the whole sequence element by element, but that would be too expensive. You might think that we can do it by comparing “pointers” to the values, but this does not work since the sequences can be created separately so even though the values are equal there could be two copies of the same value in different locations and comparing pointers would say they are not equal and we would fail to recognize that we have already solved this instance.

To get around this problem the top-down approach typically requires that the user create integer surrogates that represent the input values\(^3\). The property of these integers is that there has to be a 1-to-1 correspondence between the integers and the argument values—therefore if the integers match, the arguments match. For our edit-distance problem, how do we do this? We can just store a table indexed by \((i, j)\) values we used in our code and modify our code so that if the cell already contains the solution, then we don’t recompute it.

However, another problem with top-down programming is that you may have races, although

---

\(^3\) other simple types would also work.
they are benign. If two recursive calls simultaneously want to calculate some cell in the memo
table they will both compute the same value. However, this is still a race. In addition, even if we
don’t care about the race, we can still do a lot of repeated computation due to this problem. There
are ways to get around this issue using complicated programming paradigms and systems, but that
is outside the scope of this class.

**Bottom-Up Dynamic Programming**

We will now consider a different technique for implementing dynamic programs. Instead of sim-
ulating the recursive structure, which starts at the root of the DAG, it starts at the leaves of the
DAG and fills in the results in some order that is consistent with the DAG—i.e. for all edges \((u, v)\)
it always calculates the value at a vertex \(u\) before working on \(v\). Because of this all values will be
already calculated when they are needed.

The simplest way to implement bottom-up dynamic programming is to do some form of sys-
tematic traversal of a DAG and it is therefore useful to understand the structure of the DAG. For
every example lets consider the structure of the DAG for minimum edit distance. For example lets con-
sider the two strings \(S = tcat\) and \(T = atc\). We can draw the DAG as follows where all the
edges go down and to the right.

```
   t  c  a  t

   1  2  3  4 = i
     |   |   |
   a 1---o---o o---o
       |   |   |
   t 2   o o---o   o
       |   |   |
   c 3---o o----o----o
       |   |   |
     = j
```

The numbers represent the \(i\) and the \(j\) for that position in the string. Consider \(EDITDIST(2, 1)\). The
characters \(S_2\) and \(T_1\) are not equal so the recursive calls are to \(EDITDIST(3, 1)\) and \(EDITDIST(2, 2)\).
This corresponds to the vertex to the right and the one below. Now if we consider \(EDITDIST(4, 2)\) the
characters \(S_4\) and \(T_2\) are equal so there is only one recursive call. This corresponds to the vertex
diagonally below and to the right. This tells us quite a bit about the DAG. In particular it tells us
that it is safe to process the vertices by first traversing the last row from right to left, and then the
second row, and so on. It is also safe to traverse the last column from bottom to top and then the
second and so on. In all cases, the work is \(\Theta(n^2)\).
2.4 How to Parallelize this Algorithm?

There are many ways of parallelizing this code. You can divide this matrix into 4 matrices, fill in the top-left matrix, then do the middle two matrices in parallel and then do the bottom right matrix. The recurrence for span is $S(n) = 3S(n/2) + \Theta(1) = \Theta(n \lg^2 3)$. Can we get a better span? Well how about if we divided into 9 submatrices? we get $S(n) = 5S(n/3) + \Theta(1) = \Theta(n \lg^3 5)$, which is a little bit better.

In fact it is safe to process the diagonals in the / direction from top left moving to the bottom right. In this case each diagonal can be done in parallel. What is the work and span if we want to do each diagonal in parallel? There are $2n - 1$ diagonals, and the $i$th diagonal has $i$ element is $i \leq n$ and $i - n$ elements if $i > n$. For each diagonal, we can do a parallel-for loop to calculate each element in parallel, but the $i + 1$st diagonal must be computed after the $i$th diagonal.

Therefore,

$$S(n) = \sum_{i=1}^{n} \lg i + \sum_{i=1}^{n-1} \lg i$$

$$= \Theta(n \lg n)$$

Can we do even better? Therefore, perhaps instead of computing each diagonal in parallel, we can divide our table into blocks and compute these blocks. Say there are $k$ blocks and each block is of size $n/k$. Then the span is

$$S(n) = (2k - 1)S(n/k) + \sum_{i=1}^{k} \lg i + \sum_{i=1}^{k-1} \lg i$$

$$= (2k - 1)S(n/k) + \Theta(k \lg k)$$

$$= (2n/ \lg n - 1)S(\lg n) + \Theta(n)$$

if we substitute $k = n/ \lg n$ in order to make the last term $\Theta(n)$. We can now draw a recursion tree to solve the recurrence and get the solution $\Theta(n 2^{\lg^* n})$. This is the best solution I can come up with.