Project in String Processing Algorithms (Spring 2012)

Topics to be shared Tue 17.1, 14-16, B119, Veli Mäkinen

1. **Dynamic Programming and Aho-Corasick.**

   Generalized edit operation converts a substring into another substring. For example, `abc` $\rightarrow$ `cd`. Let $G = \{(\alpha_1, \beta_1), \ldots, (\alpha_r, \beta_r)\}$ be the set of allowed generalized edit operations, where each $(\alpha_i, \beta_i) \in \Sigma^* \times \Sigma^*$. Let $c : \Sigma^* \times \Sigma^* \rightarrow \{1, 2, \ldots, W\}$ be a cost assigned to each edit operation, denoted $c(\alpha_i, \beta_i)$. The cost $c(L)$ of a list L of edit operations is the sum of the costs of the edit operations in L. *Generalized edit distance* between strings A and B is the minimum total cost of a list of generalized edit operations to convert A from left to right into B.

   For example, let $A =$ `abcabc`, $B =$ `cdcd`, $G = \{($`abc,cd`$), ($`ab,c`$), ($`ca,d`$), ($`bc,cd`$)\}$ with costs $c($`abc,cd`$) = 2$ and $c($`ab,c`$) = c($`ca,d`$) = c($`bc,cd`$) = 1$. Then there are two lists of operations to convert A into B: (i) ($`abc,cd`$), ($`abc,cd`$) with cost 4 and (ii) ($`ab,c`$), ($`ca,d`$), ($`bc,cd`$) with cost 3. Generalized edit distance between A and B is hence 3.

   The $O(mn)$ unit cost edit distance computation for strings of length m and n is easy to extend to generalized edit distance to achieve $O(mn||G||)$ time, where $||G||$ is $\sum_{i=1}^{r}(|\alpha_i| + |\beta_i|)$: Construct an array $D[0 \ldots m, 0 \ldots n]$ such that $D[i, j]$ stores the minimum cost generalized edit distance between $A[1 \ldots i]$ and $B[1 \ldots j]$. The recurrence to compute each $D[i, j]$ is

$$D[i,j] = \min \quad \{ \quad D[i', j'] + c(A[i'+1 \ldots i], B[j'+1 \ldots j]) \\ \mid (A[i'+1 \ldots i], B[j'+1 \ldots j]) \in G\}. \qquad (1)$$

   With initialization $D[0, 0] = 0$, definition $\min\{\emptyset\} = \infty$, usage of suitable evaluation order, and implementation of $(\alpha, \beta) \in G$ as a for-loop, the claimed running time follows to compute $D[m, n]$, i.e. the minimum cost generalized edit distance of A and B.

   This basic algorithm is easy to speed up to $O(mn|G| + ||G||)$ time (assuming constant alphabet size here) by building two Aho-Corasick automata, one for set $\{\alpha_i \mid 1 \leq i \leq r\}$ and one for $\{\beta_i \mid 1 \leq i \leq r\}$, and then feeding A to first automaton and B to second automaton during filling the dynamic programming table. The pseudocode follows. There `Apatterns` denotes set $\{\alpha_i \mid 1 \leq i \leq r\}$ and `Bpatterns` denotes set $\{\beta_i \mid 1 \leq i \leq r\}$.

```
ACautomatonA = new ACautomaton(Apatterns)
ACautomatonB = new ACautomaton(Bpatterns)
// Assume G is a list of string pairs;
// i-th pair is (G[i].first, G[i].second)
// ACautomaton.push(a) reads symbol a and changes state
// ACautomaton.reset() returns the state to the root
//  (equaling empty string)
// ACautomaton.output() prints current accepting states
```

```
//  as subset of indices in G
// intersection(C,D) computes the intersection of C and D
// D[i,j]=infinity in the beginning for all i,j
// A[0]=B[0] contain a special symbol such that push() does nothing
D[0,0] = 0
for i=0 to m do
   ACautomatonA.push(A[i])
   for j=0 to n do
       ACautomatonB.push(B[j])
       for k in intersection(ACautomatonA.output(),
                             ACautomatonB.output()) do
          D[i,j] = min(D[i,j],D[i-|G[k].first|+1,j-|G[k].second|+1]+
                               c(A[i-|G[k].first|+1 ... i],
                                 B[j-|G[k].second|+1 ... j]))
   ACautomatonB.reset()
print "Edit distance between " + A + " and " + B + " is " + D[m,n]
```

The goal in the project is to implement Aho-Corasick automaton and apply it to achieve the above speed up. Experimental comparison should be made on what type of generalized edit operations G the new algorithm wins the basic algorithm.

Do you find any natural application where generalized edit distance would be important?

2. **Suffix Trees and Overlap Computation**

   In *fragment assembly* the goal is to reconstruct a *superstring* S given a set $\mathcal{S}$ of its fragments (substrings) as the input. This is possible if the substrings cover all positions in S, as one can deduce S by overlapping the set of fragments in correct order. Finding the correct order is computationally a hard problem (and even the optimization goal is not easy to define), but many heuristics have been proposed that start with the graph defined by overlap-relations between the fragments. This *all-pairs suffix/prefix overlap computation* is the task of determining for all pairs of fragments the length of their longest overlap, i.e. longest suffix of fragment A that is a prefix of fragments B for all pairs A and B in $\mathcal{S}$. Let us refine the problem such that only overlaps longer than a given threshold are outputted. The problem can be solved in optimal running time $O(N+\text{out})$, where N is the overall length of fragments in $\mathcal{S}$ and out is the size of the output, see e.g. [Gus97][pp. 137–8]: The idea is to build a suffix tree for the concatenation of strings in $\mathcal{S}$. Then the leaves corresponding to start of the strings have paths shared with all suffixes and the longest overlaps can be recorded in stacks using one depth-first traversal.

   The goal in the project is to implement the above overlap computation algorithm using suffix trees. One can use any existing suffix tree implementation, but it is highly recommended to use *compressed suffix trees*: `http://www.uni-ulm.de/en/in/institute-of-theoretical-computer-science/research/sdsl.html` or `http://www.cs.helsinki.fi/group/suds/cst/`. These implementations

offer an implicit interface to suffix tree (class with functions to access children, parent, suffix links, etc.); under the hood they have a mixture of compressed data structures, but to use them it is sufficient to know how an explicit suffix tree functions.

For comparison, one could compare the algorithm to the trivial one that tries out all combinations brute force. (Do you know how to find in $O(|A| + |B|)$ time the longest overlap of A and B without using suffix tree?)

It is also possible to extend this topic for a bigger group so that more suffix tree -based algorithms of the same kind as above from Gusfield's book are implemented and tested.

3. **Suffix Trees and Approximate Search**

   Suffix tree provides a way to do various kind of approximate pattern matching faster than scanning the text from left to right. Consider you want to locate all occurrences of pattern P in text T allowing k mismatches. You can build suffix tree for T and then *backtrack* with the pattern on all paths of the suffix tree. Whenever you follow a branch with next character different from that of pattern, you can keep a counter on how many mismatches have been found so far. There is no need to continue on branches where the counter exceeds k. In theory, this approach does not provide a good worst case running time, but on small k it works fast. The advantage of the approach is that it extends to many variants of approximate search. For example, edit distance can be supported by computing the columns of the dynamic programming matrix along the paths, or even faster by simulating Myers' bitparallel algorithm on the paths.

   There are several heuristics to speed up this kind of backtracking search. The goal in this project is to implement some of these heuristics and do an experimental comparison on their efficiency. Possible heuristic speed-ups to try out are the following and their combinations (of course one can invent new ones):

   - If one only wants to find one match, it makes sense to go first greedily to directions having fewer errors rather than depth-first order.
   - Build suffix tree of reverse text and use it to partition the reverse pattern into minimum amount of pieces such that each piece occurs exactly; this can be done greedily by matching exactly the reverse pattern in suffix tree of reverse text until there is no branch to follow, and starting the matching recursively with the rest of the reverse pattern from the root. Let this induce partitioning for the (forward) pattern $P = P^1P^2 \cdots P^p$. Then when backtracking for P in the suffix tree of T with the current position in P being inside piece P^i and number of errors seen so far being k', one knows that if $k' + p - i > k$ there is no need to continue the search in the current branch.
   - Consider 1-mismatch problem. Then an occurrence has this mismatch either in the left half of P or in the right half of P. One can search these cases separately. For the former, use suffix tree of the reverse text and search the reverse pattern with 0 errors to the midpoint, then backtracking with at

most 1-mismatch. For the latter, do the symmetric case with suffix tree of the text. Approach extends to more errors, but then there are always cases where backtracking must be started already from the root. However, when searching only for one occurrence, one can consider these bad cases last.

Depending on the size of the group, one can concetrate on k-mismatches problem or extend the approach for k-errors. Check previous assignment for pointers to existing (compressed) suffix tree implementations that can be used as a basis.

4. **Suffix Arrays, LCPs, and Maximal Exact Matches**

Maximal Exact Matches (MEM) is the problem of computing for two strings A and B tuples (i, j, k, l) such that $A[i, j] = B[k, l]$ and $j - i \geq K$, but $A[i-1, j] \neq B[k-1, l]$ and $A[i, j+1] \neq B[k, l+1]$, where K is a threshold parameter. This problem can be solved in the optimal time $O(|A|+|B|+\text{out})$, where out is the size of the output. There are many solutions achieving the optimal time e.g. using suffix trees or enhanced suffix arrays. One such (easy to implement) solution is described below and similar alternatives can be found in the literature.

Build suffix array SA for $c = A\#B$, where $\#$ is a symbol not occurring in A or B. Mark in a bitvector I the suffixes of A and B in the order of SA: $I[i] = 1$ iff $SA[i]$ is a suffix of B. Build also the LCP-array recording at $LCP[i]$ the length of the longest common prefix of suffixes $C[SA[i]\ldots]$ and $C[SA[i-1]\ldots]$. For i such that $I[i] = 0$ all indexes $j > i$ such that $I[j] = 1$ and $RMQ(i, j) = \min_{i'=i+1}^{j} LCP[i'] \geq K$ have the property that suffixes $C[SA[i]\ldots]$ and $C[SA[j]\ldots]$ share a prefix at least of length K, the first suffix is from A, and second from B. Each pair (i, j) satisfying the above give an exact match tuple $(SA[i], SA[j] - |A| - 1, SA[i] + RMQ(i, j) - 1, SA[j] - |A| + RMQ(i, j) - 2)$. However, these tuples may not all be maximal: It can happen that $(SA[i] - 1, SA[j] - |A| - 2, SA[i] + RMQ(i, j) - 1, SA[j] - |A| + RMQ(i, j) - 2)$ is also an exact match. This happens iff $C[SA[i] - 1] = C[SA[j] - 1]$. To skip efficiently all non-maximal tuples, let R be a bitvector such that $R[i] = 1$ iff $C[SA[i] - 1] \neq C[SA[i - 1] - 1]$. Now, if for some (i, j) we have $C[SA[i] - 1] = C[SA[j] - 1]$, we can set $j \leftarrow succ(R, j + 1)$, where $succ(X, x)$ gives the first 1-bit in $X[x\ldots]$. Then $j \leftarrow succ(I, j)$ gives the next candidate for (i, j). Assume that $RMQ(i, j)$ and $succ(X, x)$ can be calculated in constant time. Then the sketched algorithm is optimal, as at most every second tuple visited for fixed i is not maximal. Moreover, all maximal exact matches are considered by using a symmetric solution for $j < i$.

The following pseudocode gives a slightly more engineered solution that avoids the use of $RMQ(i, j)$. This would not be a bottleneck for the theoretical running time, as there exist data structures that can be build in $O(|A| + |B|)$ time and answer $RMQ(i, j)$ in constant time. Queries $succ(X, x)$ are easy to support in constant time by building a cumulative array, but there also exist $o(|A| + |B|)$ bits data structures that achieve constant time query time.[1]

```
Let SA, LCP, I, R, and K be as above
```

[1] Such techniques are covered in the Data Compression Techniques -course.

```
// recompute LCP-array to store RMQ(i,succ(I,i+1))
minlcp=|A|+|B|+2
for i=|A|+|B|+1 downto 1 do
   temp = LCP[i]
   LCP[i] = minlcp
   if I[i]=1 then minlcp = temp
   else  minlcp = min(minlcp,temp)
// compute LCP-array for I[i]=1 to store RMQ(i,succ(I,succ(R,i+1))))
LCPR = copy of LCP
minlcp=|A|+|B|+2
for i=|A|+|B|+1 downto 1 do
   if I[i]=1 then LCPR[i]=minlcp
   if R[i]=1 then minlcp = LCPR[i]
   else minlcp = min(minlcp,LCPR[i])
// collect maximal exact matches
for i such that I[i]=0 do
   j = succ(I,i+1)
   minlcp = LCP[i]
   while minlcp>=K do
       if C[SA[i]-1]!=C[SA[j]-1] then
          print (SA[i]-1,SA[j]-|A|-2,SA[i]+minlcp-1,SA[j]-|A|+minlcp-2)
          minlcp = min(minlcp,LCP[j])
          j = succ(I,j+1)
       else
          minlcp = min(minlcp,LCPR[j])
          j = succ(I,succ(R,j+1))
// Do the analogous computation for the symmetric case j<i
```

The goal in the project is to implement the above algorithm (or some other that uses enhanced suffix arrays). You can use existing suffix array and LCP array construction algorithms to start with. You can compare the implementation to a non-optimal practical solution:

```
// use original LCP
// collect maximal exact matches
for i such that I[i]=0 do
   minlcp = |A|+|B|+2
   for j=i+1 to |A|+|B|+1 do
      minlcp = min(LCP[i],minlcp)
      if minlcp>=K and C[SA[i]-1]!=C[SA[j]-1] then
          print (SA[i]-1,SA[j]-|A|-2,SA[i]+minlcp-1,SA[j]-|A|+minlcp-2)
      else if minlcp<K then break
// Do the analogous computation for the symmetric case j<i
```

Do you find any instances where the theoretically optimal algorithm is faster in practice than the trivial solution above?

For the poster, look up in what kind of applications MEMs are used.

5. **Distributed Suffix Sorting**

   Consider the case you have p processors and access to a shared fast storage space (ideally shared RAM). The task is to sort suffixes of a string (i.e. construct suffix array) as fast as possible exploiting distributed computation. One way to proceed is to use *pivots* and either *bucket sorting* or *doubling algorithm* a.k.a. *Karp-Miller-Rosenberg naming technique*.

   The idea to first choose $p+1$ strings (pivots) $S^0, S^1, \ldots, S^p$ such that suffixes of a string $T = t_1 t_2 \ldots t_n$ listed in lexicographic order could be partitioned into p almost equal size blocks with the property that block i suffixes are larger than S^{i-1} and smaller than S^i. This is a difficult combinatorial problem to solve in distributed manner (without resorting to sorting leading to a chicken and egg problem). Here we assume that pivots are given, or they can be trivially constructed assuming T is sampled from uniform distribution.

   Now prosessor assigned to pivot S^i can concentrate in sorting suffixes that lie between S^{i-1} and S^i. Bucket sort could be used here with the modification that each prosessor ignores buckets that are strictly out of the pivot range. When all prosessors have all their buckets shrinked to size one, sorting is done, and one can concatenate the results in the pivot order to produce the suffix array. Even with the perfect choice of pivots the worst case running time of distributed bucket suffix sort is $O(n^2)$ time; consider text $T = c^n$.

   We can do better using the *doubling algorithm*. To see how that algorithm extends to the distributed case, let us consider i-th prosessor after j-th doubling step. It stores the array $A^i[0 \ldots n_i - 1]$ with $A^i[k] = (\mathtt{rank}^i_k, x^i_k)$, where $\mathtt{rank}^i_k$, $1 \leq i < n_i - 1$, is the lexicographic rank of substring $T[x^i_k \ldots x^i_k + 2^j - 1]$ among substrings of the same length lying in the same pivot interval. Values $A^i[0] = (\mathtt{rank}^i_0, x^i_0)$ and $A^i[n_i - 1] = (\mathtt{rank}^i_{n_i-1}, x^i_{n_i-1})$ correspond to pivots S^{i-1} and S^i, respectively. For this to work, we need to concatenate the pivots to the end of the text: $T \leftarrow t_1 t_2 \ldots t_n \# S^0 \# S^1 \# \ldots S^p \#$, where $\#$ is smaller than other alphabet symbols.

   Before proceeding to the next step, we (virtually) produce a global rank table $A[0 \ldots N]$ as follows:

$$A[K] = (r_K, x^i_k), \tag{2}$$

   where i is such that $\sum_{i'=1}^{i-1} n_{i'} \leq K < \sum_{i'=1}^{i} n_{i'}$, $k = K - \sum_{i'=1}^{i-1} n_{i'}$, and

$$\begin{aligned} r_K &= r_{K-1} \text{ if } \mathtt{rank}^i_k = \mathtt{rank}^i_{k-1} \text{ and } k > 0, \\ r_K &= r_{K-1} + 1 \text{ if } \mathtt{rank}^i_k \neq \mathtt{rank}^i_{k-1} \text{ and } k > 0, \text{ and} \\ r_K &= r_{K-1} \text{ otherwise.} \end{aligned}$$

   Notice that the last case is $k = 0$, and the rank is not incremented because previous block ends with the same pivot as the next one starts, so they must have the same rank. Notice also that each prosessor can construct its part of the table A almost independently; largest rank of each prosessor is known after sorting and it is sufficient that each prosessor sends that value to all the other prosessors.

Finally for each $0 \leq K \leq N$ table $R[1 \ldots |T|]$ is filled by applying $R[x] = r$ for $A[K] = (x, r)$. This writing can be done in distributed manner, assuming a computation model where if array cell is locked for writing, the prosess does not wait for the lock to be released but continues; several processors can write to the same cell x, but they all must have the same rank, so the first writing is sufficient.

The next $((j+1)$-th) doubling step for prosessor i works as follows. Set $A^i[k] = ((R[x^i_k], R[x^i_k + 2^{j+1} - 1]), x^i_k)$ and sort the array by the values $(R[x^i_k], R[x^i_k + 2^{j+1} - 1])$, and replace the values with the ranks of the pairs. Here omitting the special cases of indexes exceeding $|T|$. Once the array is sorted, the head of the array where ranks are below the pivot S^{i-1} and tail of the array where ranks are above S^i are cut off.

This process is repeated on all processors i at most $O(\log n)$ times until all ranks are unique. With perfect choice of pivots, the running time of distributed doubling algorithm is $O(n + N \log^2 n/p)$, where $n \leq N \leq np$ is the size of the global rank table after first round (again $T = c^n$ is the worst case). This bound assumes comparison sort is used for sorting the pairs; better bounds can be achieved using other models of computation. When assuming T is sampled from uniform distribution, we have $N \leq n\lceil p/\sigma \rceil$ (for each character there is the same amount of pivots).

Although these algorithms do not give any theoretical time improvement over the linear time suffix sorting algorithms, the benefit is that they can work in smaller shared memory. Distributed bucket sorting works already completely without shared memory, and distributed doubling sort can be modified so by splitting array R in p equal size parts; each prosessor handles its own part. The complication is that to access values $R[x^i_k]$ and $R[x^i_k + 2^{j+1} - 1]$ one needs to do all-pairs message passing. However, this does not increase the running time of the algorithm significantly as one round of message passing takes $O(p^2 + N/p)$ time (with the perfect pivot assumption). This is required to be done $O(\log n)$ times.

The goal in the project is to implement either the simpler distributed bucket suffix sort working in cluster environment (ukko001.hpc.cs.helsinki.fi-ukko240.hpc.cs.helsinki.fi) with shared file system, or the distributed doubling algorithm using threads and shared memory. Experimental comparison should be made against a serial implementation of (linear time) suffix sorting.[2]

References

[Gus97] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.

[2]If one is familiar with *MapReduce* framework, it is possible to implement the distributed doubling algorithm with all-pairs message passing using it. Our `ukko`-cluster has *disco* (`http://discoproject.org/`) implementation of MapReduce installed if someone wants to try out this approach.