Report Number: WUCS-86-16

1986-01-01

Approximation Algorithms for the Shortest Common Superstring Problem

Jonathan S. Turner

APPROXIMATION ALGORITHMS FOR THE SHORTEST COMMON SUPERSTRING PROBLEM

Jonathan S. Turner

WUCS-86-16

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

Approximation Algorithms for the Shortest Common Superstring Problem

Jonathan S. Turner
jst@wucs.UUCP

1. Introduction

Let $s_1 = a_1, \ldots, a_r$ and $s_2 = b_1, \ldots, b_s$ be strings over some finite alphabet Σ. We say that s_1 is a *substring* of s_2 if there is an integer $i \in [0, s-r]$ such that $a_j = b_{i+j}$ for $1 \leq j \leq r$. We also say in this case that s_2 is a *superstring* of s_1.

An instance of the *shortest common superstring problem* (SCS) is a list of strings $S = (s_1, \ldots, s_n)$ over a finite alphabet Σ. The object of the problem is to find a minimum length string that is a superstring of every $s_i \in S$. We let $\phi^*(S)$ denote the length of a minimum length superstring.

EXAMPLE: If $S = \{\texttt{egiach}, \texttt{bfgiak}, \texttt{hfdegi}, \texttt{iakhfd}, \texttt{fgiakh}\}$, the string `bfgiakhfdegiach` is a solution of length 15.

We say that a set of strings is *substring free* if no string in the set is a substring of any other. We will generally limit our attention to substring free sets. This involves no loss of generality, since any set of strings has a unique substring free subset which has the same solutions as the original set.

We have presented the problem in the conventional way, with the object being to minimize the solution length. It is useful to consider an alternative viewpoint as well. One can view the object of the problem as being to find an ordering of the strings that maximizes the amount of overlap between consecutive strings. To make this precise we need a few definitions.

Let $s_1 = a_1, \ldots, a_r$ and $s_2 = b_1, \ldots, b_s$ be strings. We define

$$\psi(s_1, s_2) = \max \{k \geq 0 \,|\, a_{r-k+i} = b_i \; 1 \leq i \leq k\}$$

If $\psi(s_1, s_2) = k$ then $s_1 \circ s_2$ is defined to be the string $a_1, \ldots, a_r, b_{k+1}, \ldots, b_s$. We note that if s_1, s_2, s_3 are strings, none of which is a substring of another, then $s_1 \circ (s_2 \circ s_3) = (s_1 \circ s_2) \circ s_3$; that is, the overlapping operation is associative for substring free sets. Consequently, we may write $s_1 \circ s_2 \circ \cdots \circ s_n$ with no ambiguity.

Let π be a permutation on $\{1, \ldots, n\}$. We will usually write π_i for $\pi(i)$. We define

$$\psi_\pi(s_1, \ldots, s_n) = \sum_{i=1}^{n-1} \psi(s_{\pi_i}, s_{\pi_{i+1}})$$

and $\phi_\pi(s_1, \ldots, s_n) = |s_{\pi_1} \circ \cdots \circ s_{\pi_n}|$. Note that for any instance $S = (s_1, \ldots, s_n)$ of SCS,

$$\phi_\pi(S) = \|S\| - \psi_\pi(S)$$

where $\|S\| = \sum_{i=1}^{n} |s_i|$. In particular,

$$\phi^*(S) = \|S\| - \psi^*(S) \quad \text{where} \quad \psi^*(s_1, \ldots, s_n) = \max_{\pi} \psi_\pi(s_1, \ldots, s_n)$$

Hence, we can view the object of the SCS problem as being to find a mapping π that maximizes ψ_π.

Let A be an algorithm for SCS which given a collection of strings $S = (s_1, \ldots, s_n)$ produces a mapping $\pi = \pi_A(S)$. We define $\psi_A(S) = \psi_\pi(S)$ and $\phi_A(S) = \phi_A(S)$.

SCS was shown to be NP-complete by Maier and Storer in [8]. Another, and more elegant proof appears in [3] and [4]. One obvious application for the problem is data compression. Storer and Szymanski [15] for example, consider a fairly general model of data compression which includes SCS as an important special case. See also [9]. Another application is to DNA sequencing. SCS is one of the simplest models for the problem of recovering DNA sequencing information from experimental data [5,12,14]. To our knowledge the only approximation algorithm to be discussed in the literature is a simple greedy algorithm which is treated briefly by Gallant in [4]. Gallant claims that for this algorithm, which we refer to as SGREEDY, $\phi_{\text{SGREEDY}}(S) \leq (3/2)\phi^*(S)$ for all collections of strings S. We show that this is not in fact true by displaying a set of strings S for which $\phi_{\text{SGREEDY}}(S) \approx 2\phi^*(S)$. We have found no worse example problem than this, but have also been unsuccessful in proving an upper bound on the performance of this algorithm in terms of the length measure. On the other hand, we do show that $\psi^*(S) \leq 2\psi_{\text{SGREEDY}}(S)$.

In section 2 we relate SCS to the longest path problem (LPP) in graphs by describing a transformation from SCS to LPP that preserves solution values with respect to the overlap measure. We then construct three approximation algorithms for LPP, two based on matching and the third a greedy heuristic. By virtue of the transformation from SCS, all three are also approximation algorithms for SCS. We show that the greedy heuristic for LPP always produces solutions within a factor of three of the optimum value. In section 3, we show that the instances of LPP that result from our transformation from SCS have a special structure that allows us to obtain a tighter bound. We also describe an efficient implementation of this greedy algorithm for strings using a compact representation of suffix trees. In section 4, we relate SCS to the traveling salesman problem (TSP) by another transformation that preserves solution values, this time with respect to the length measure. The instances of TSP arising from this transformation are asymmetric, but satisfy the triangle inequality. There are no approximation algorithms known for this problem with provably good worst-case performance, nor have we succeeded in finding any. Nevertheless, this transformation means that if such an algorithm is found, it can be used for SCS as well as TSP. If on the other hand, it is turns out that approximating this version of TSP is hard, then any approximation algorithm for SCS, will have to make use of special structural properties present in the instances of TSP that arise from this transformation.

2. SCS and the Longest Path Set Problem

In this section we relate SCS to the *longest path problem* (LPP) in graphs. An instance of the longest path problem is a complete directed graph $G = (V, E)$ with each edge (u, v) having a non-negative integer length $\ell(u, v)$. The *length of a path* p in G is defined to be the sum of the lengths of its edges and is denoted $\lambda_p(G, \ell)$. The object of the longest path problem is to find a Hamiltonian path p (that is a path including every vertex) in G that maximizes $\lambda_p(G, \ell)$. The length of such a longest path is denoted $\lambda^*(G, \ell)$.

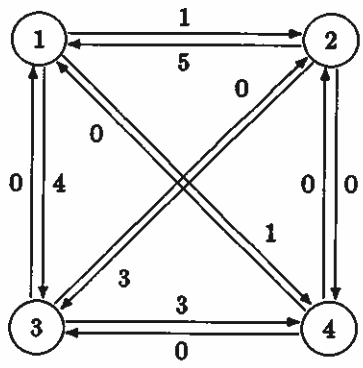

Figure 1: Example of transformation from SCS to LPP ($S = \{\texttt{cbadef}, \texttt{fcbade}, \texttt{adefcd}, \texttt{fcadfb}\}$)

Let $S = (s_1, \ldots, s_n)$ be an instance of SCS. We define LPP(S) to be an instance (G, ℓ) of LPP with

$$\begin{aligned} V &= \{u_1, \ldots, u_n\} \qquad E = V \times V \\ \ell(u_i, u_j) &= \psi(s_i, s_j) \qquad 1 \leq i, j \leq n,\ i \neq j \end{aligned}$$

An example of this transformation is shown in Figure 1.

Let π be a permutation on $\{1, \ldots, n\}$. We can view π as defining a Hamiltonian path $u_{\pi_1}, \ldots, u_{\pi_n}$ in G. We let $\lambda_\pi(G, \ell)$ denote the length of this path. We now state a trivial, but useful theorem.

THEOREM 2.1. *Let $S = (s_1, \ldots, s_n)$ be an instance of* SCS, *$(G, \ell) =$* LPP*(S) and let π be any permutation on $\{1, \ldots, n\}$. $\lambda_\pi(G, \ell) = \psi_\pi(S)$. In particular, $\lambda^*(G, \ell) = \psi^*(S)$.*

The theorem implies that any approximation algorithm for LPP is an approximation algorithm for SCS with respect to the overlap measure. In the remainder of this section, we present three simple approximation algorithms for LPP.

2.1. Matching Algorithm

A *matching* in a graph $G = (V, E)$ is a set of edges, no two of which share a common vertex. A maximum matching in a graph with edge lengths $\ell(e)$ is a matching M that maximizes $\ell(M)$. We define $\mu^*(G, \ell) = \max_M \ell(M)$ to be the value of a maximum matching. There are algorithms for finding maximum matchings having running times of $O(n^3)$ (where $n = |V|$) [16].

Our first algorithm for LPP is based on the observation that any matching for an instance (G, ℓ) of LPP can be extended to a path (since G is assumed to be complete) and a maximum matching must have total length at least half that of a longest path. (Recall that we are restricting attention to non-negative weights.)

THEOREM 2.2. *If $(G = (V, E), \ell)$ is an instance of* LPP *then $\lambda^*(G, \ell) \leq 2\mu^*(G, \ell)$.*

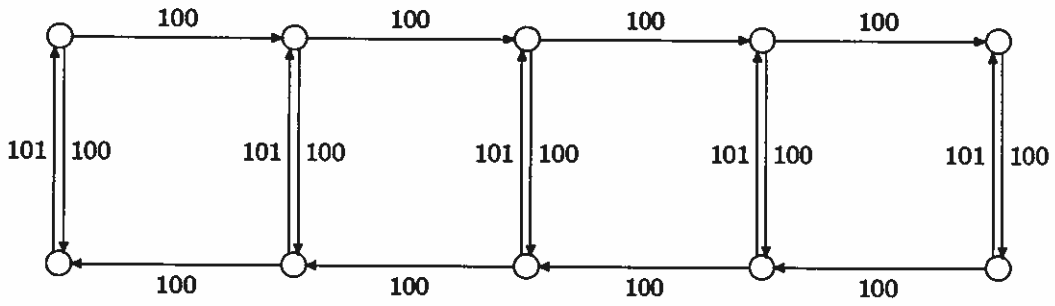

Figure 2: Worst-Case Example for Theorem 2.2

```
function edgeset MATCH(digraph G = (V, E), edgelengths ℓ)
    edgeset P, M;
    P := ∅;
    do E ≠ ∅ →
        M := MAXMATCH(G, ℓ);
        P := P ∪ M;
        for (u, v) ∈ M →
            Delete from G, all edges of the form (u, x) or (y, v);
            Collapse u and v into a single vertex;
        rof;
    od;
    return P;
end;
```

Figure 3: Matching Algorithm for LPP

Proof. Let P be a set of edges defining any Hamiltonian path. Let Q be obtained by taking alternate edges from P and let $R = P - Q$. Both Q and R are matchings. The sum of the lengths of the edges in Q is $\leq \mu^*(G, \ell)$. Similarly, the sum of the lengths of the edges in R is $\leq \mu^*(G, \ell)$. Hence, $\ell(P) \leq 2\mu^*(G, \ell)$ and since this holds for all paths P, it follows that $\lambda^*(G, \ell) \leq 2\mu^*(G, \ell)$. □

Remark. There are instances (G, ℓ) of LPP for which $\lambda^*(G, \ell)$ approaches $2\mu^*(G, \ell)$. Figure 2 shows a graph for which $\lambda^*(G, \ell) = 900$ and $\mu^*(G, \ell) = 505$. (The edges not explicitly shown have length 0.) The example is easily extended to give graphs for which the ratio λ^*/μ^* is arbitrarily close to 2.

Theorem 2.2 provides the basis for our first approximation algorithm shown in Figure 3. The procedure MATCH starts by finding a maximum matching in G, then removes edges that are ruled out by the selected edges, collapses the selected edges into single vertices and then repeats the process on the new graph. To see that the algorithm does construct a Hamiltonian path, note the following: (1) the edge eliminations ensure that the set P never contains two edges leaving a common vertex or entering a common vertex, (2) the collapsing of edges into single vertices prevents creation of cycles and (3) since the original graph is assumed to be complete, the algorithm will halt only when a complete Hamiltonian path has been constructed. An example illustrating the operation of the algorithm is given in Figure 4.

Theorem 2.2 implies that $\psi^*(G, \ell) \leq 2\psi_{\text{MATCH}}(G, \ell)$ for any instance (G, ℓ) of LPP. This can't be improved, as can be seen by considering the operation of MATCH on the graph in Figure 2.

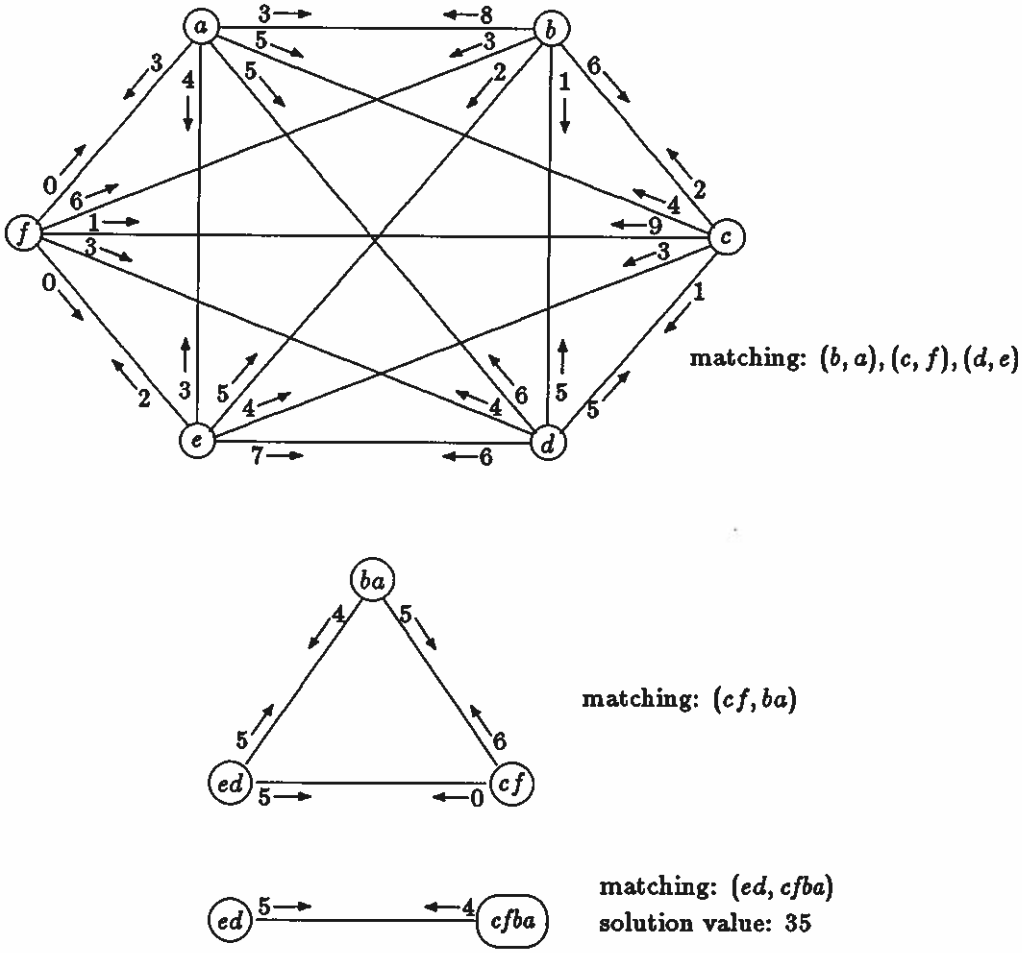

Figure 4: Example of Algorithm MATCH

The running time of MATCH is determined primarily by the matching algorithm used. Assuming a matching algorithm that runs in $O(n^3)$ time, we get a running time of $O(n^3 \log n)$ for MATCH.

2.2. Directed Matching Algorithm

A *directed matching* in a digraph $G = (V, E)$ is a set of edges, no two of which enter a common vertex and no two of which leave a common vertex. In other words, it is a subgraph of G comprising a collection of disjoint paths and cycles. A maximum directed matching in a graph G with edge lengths $\ell(e)$ is a directed matching M that maximizes $\ell(M)$. We define $\delta^*(G, \ell) = \max_M \ell(M)$ to be the value of a maximum directed matching (where in this case, M ranges over all directed matchings of G). There are algorithms for finding a maximum directed matchings having running times of $O(n^{5/2})$ [16].

Given any matching M, let M^- be a subset of M obtained by discarding a least cost edge from each cycle in M. Our next algorithm for LPP is based on the observation contained in the next theorem.

```
function edgeset DIMATCH(digraph G = (V, E), edgelengths ℓ)
    edgeset P, M;
    P := ∅;
    do E ≠ ∅ →
        M := MAXDIMATCH(G, ℓ);
        M⁻ := M − one least cost edge from each cycle of M;
        P := P ∪ M⁻;
        for each path (u_1, ..., u_r) ∈ M⁻ →
            Delete from G, all edges of the form (u_i, x), 2 ≤ i ≤ r;
            Delete from G, all edges of the form (x, u_i), 1 ≤ i ≤ r − 1;
            Delete from G the edge (u_r, u_1), if present;
            Collapse the path into a single vertex;
        rof;
    od;
    return P;
end;
```

Figure 5: Directed Matching Algorithm for LPP

THEOREM 2.3. *Let $(G = (V, E), \ell)$ be an instance of* LPP, *let M be a maximum directed matching of G and let k be the minimum number of edges in any cycle defined by M. $\lambda^*(G, \ell) \leq \frac{k}{k-1}\ell(M^-)$. In particular, $\lambda^*(G, \ell) \leq 2\ell(M^-)$.*

Proof. Let P be a set of edges defining a path and let M be a maximum directed matching. Notice that P is a directed matching and hence $\ell(P) \leq \ell(M)$. Let C be a cycle in M with h edges and let C^- be a path obtained by discarding a minimum length edge from C.

$$\ell(C) \leq \frac{h}{h-1}\ell(C^-) \leq \frac{k}{k-1}\ell(C^-)$$

Also, for every path $R \in M$, $\ell(R) \leq \frac{k}{k-1}\ell(R)$. Summing over all paths and cycles in M yields $\ell(M) \leq \frac{k}{k-1}\ell(M^-)$. Since this is true for all paths P and since $\ell(P) \leq \ell(M)$, $\lambda^*(G, \ell) \leq \frac{k}{k-1}\ell(M^-)$. □

Remark. There are instances (G, ℓ) of LPP for which $\lambda^*(G, \ell)$ approaches $2\ell(M^-)$. Consider for example, the graph shown in Figure 2. For this graph $\lambda^*(G, \ell) = 900$ and the optimum directed matching consists of five cycles each having two edges and length 201. When the cycles are broken, we have $\ell(M^-) = 500$. The example is easily extended to give graphs for which the ratio $\lambda^*(G, \ell)/\ell(M^-)$ is arbitrarily close to 2.

We note that $\delta^*(G, \ell) \geq \lambda^*(G, \ell)$. Hence, it provides a measure of how close a given solution is to optimal. We expect that the solutions obtained by breaking cycles will often be much closer to optimal than the bound in the theorem implies.

Theorem 2.3 provides the basis for our next approximation algorithm for LPP, shown in Figure 5. This algorithm constructs a maximum directed matching M in G, then breaks all the cycles in M and constructs a new graph in which the paths of M correspond to vertices. It then proceeds by finding a maximum directed matching in the new graph, continuing in this fashion until a Hamiltonian path in the original graph has been found. To verify that the algorithm does construct a Hamiltonian path, it suffices to note the following: (1) the edge eliminations ensure that the set P never contains

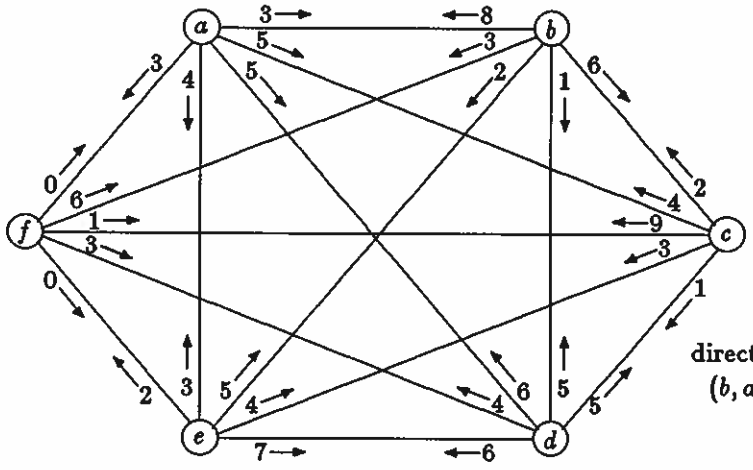

Figure 6: Example of Algorithm DIMATCH

two edges leaving a common vertex or entering a common vertex, (2) cycles formed are explicitly broken and the broken edges removed from the graph and (3) since the original graph is assumed to be complete, the algorithm will halt only when a complete Hamiltonian path has been constructed. An example illustrating the operation of the algorithm is given in Figure 6.

Theorem 2.3 implies that $\psi^*(G,\ell) \leq 2\psi_{\text{DIMATCH}}(G,\ell)$ for any instance (G,ℓ) of LPP. This can't be improved, as can be seen by considering the operation of MATCH on the graph in Figure 2. The running time of DIMATCH is determined primarily by the directed matching algorithm used. Assuming an algorithm that runs in $O(n^{5/2})$ time, we get a running time of $O(n^{5/2}\log n)$ for DIMATCH.

DIMATCH is essentially an adaptation of an algorithm for the asymmetric traveling salesman problem (TSP) described by Karp in [6]. Karp's algorithm has poor worst-case performance for TSP, but performs well in a probabilistic sense for instances in which inter-city distances are selected uniformly on the interval [0, 1]. We have simply adapted his algorithm to the longest path problem (simplifying it slightly in the process), and observed that its worst-case performance is provably good in this context.

2.3. Greedy algorithm

The algorithms considered above are both fairly complicated and time consuming because they require the calculation of maximum weighted matchings. Another algorithm that is worth considering is the simple greedy algorithm that scans the edges in non-increasing order of length and selects an edge (u,v) if it has not previously selected an edge of the form (u,x) or (y,v) and if the collection of paths constructed so far does not include a path from v to u. On the graph in Figure 6, this

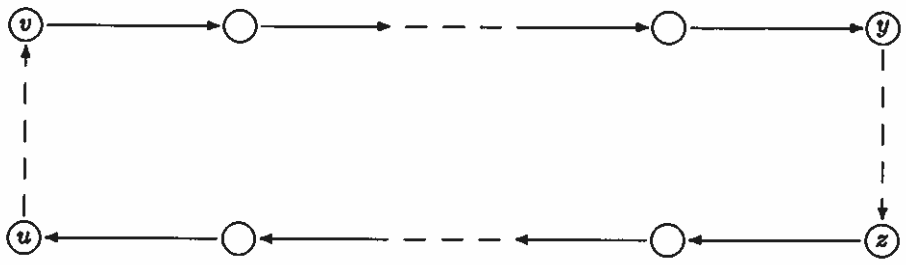

Figure 7: Illustration for Theorem 2.4

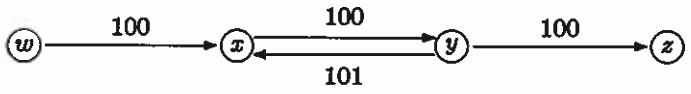

Figure 8: Worst-Case example for PGREEDY

algorithm selects the edges $(c,f),(b,a),(e,d),(f,b),(d,c)$ in that order. The next theorem gives a worst-case bound on the performance of the greedy algorithm.

THEOREM 2.4. *If (G,ℓ) is an instance of* SCS *then* $\lambda^*(G,\ell) \leq 3\lambda_{\text{PGREEDY}}(G,\ell)$.

proof. Let F be the set of edges in some optimum solution to (G,ℓ). Let $H = \{h_1,\ldots,h_s\}$ be the set of edges chosen by the greedy algorithm in the order in which they were selected (that is, h_1 was selected first, h_2 second, and so forth).

We say an edge is *permissible* at some stage of the execution of the algorithm if its selection hasn't been precluded by earlier selections. Define H_i to be the set of edges which are permissible just before h_i is selected, but not permissible after h_i is selected.

Let $h_i = (u,v)$ and consider the situation just before h_i is selected by the greedy algorithm. At this point, u is the last vertex of some path constructed by the algorithm and v is the first vertex of some path (one or both paths may contain just a single vertex). Let z be the first vertex on the path containing u and let y be the last vertex on the path containing v as shown in Figure 7.

If there is an edge from y to z that is permissible before the selection of h_i then that edge is a member of H_i. All other members of H_i have the form (u,x) or the form (x,v).

Next, note that $\ell(h_i) = \max\{\ell(e) \mid e \in H_i\}$ and that $(H_1,\ldots,H_s)$ is a partition of E. Finally note that for $i \in [1,s]$, $|F \cap H_i| \leq 3$. This yields the theorem. □

Figure 8 gives an example graph showing that the bound of Thereom 2.4 cannot be improved. (The edges not shown have length 0.) PGREEDY finds a solution of length 101, while the optimal solution has length 300. Figure 9 is a sketch of an implementation of the greedy algorithm. It bears a strong resemblance to Kruskal's minimum spanning tree algorithm and uses the disjoint set data structure described in [16] to partition the vertex set into subsets corresponding to the paths making up partial solutions. Initially the disjoint set structure d contains n singleton sets, each containing a distinct vertex. As the algorithm proceeds, sets are combined to denote the merging of the paths in the partial solution. The operation d.FIND(u) returns the name of the set containing the vertex

```
function edgeset PGREEDY(digraph G = (V, E), edgelengths ℓ)
    mapping in, out : V ↦ bit;
    disjoint_sets d;
    for u ∈ V → in(u), out(u) := false; d.MAKESET(u); rof;
    Sort E from longest to shortest;
    for (u, v) ∈ E →
        if not out(u) and not in(v) and d.FIND(u) ≠ d.FIND(v) →
            S := S ∪ {(u, v)};
            out(u), in(v) := true;
            d.LINK(d.FIND(u), d.FIND(v));
        fi;
    rof;
    return S;
end
```

Figure 9: Greedy Algorithm for LPP

u and the operation d.LINK() combines two sets and returns the name of the resulting set. The two bit vectors *in* and *out*, are indexed by vertex and set to **true** when an edge in or out of the specified vertex has been included in the solution set S. The running time for this implementation is $O(n^2 \log n)$.

3. A Greedy Algorithm for SCS

The greedy algorithm for the longest path problem can be restated for SCS as follows. Given a non-empty set of strings S, repeat the following step until S contains just one string.

> Select a pair of strings $s_1, s_2 \in S$ that maximizes $\psi(s_1, s_2)$. Remove s_1 and s_2 from S, replacing them with $s_1 \circ s_2$.

We refer to this algorithm as SGREEDY. Gallant [4] claims that $\phi_{\text{SGREEDY}}(S) \leq (3/2)\phi^*(S)$. This is not in fact true, as can be seen by considering the set of strings

$$S = \{\texttt{abcbcbcbcb}, \texttt{cbcbcbcbc}, \texttt{bcbcbcbcbd}\}$$

for which $\phi_{\text{SGREEDY}}(S) = 20 > (3/2)\phi^*(S) = 19.5$. One can easily generalize this example to show that there is no constant $c < 2$ for which $\phi_{\text{SGREEDY}}(S) \leq c\phi^*(S)$. We currently do not know if there is some constant $c > 2$ for which $\phi_{\text{SGREEDY}}(S) \leq c\phi^*(S)$.

On the other hand, Theorem 2.4 allows us to conclude that $\psi^*(S) \leq 3\psi_{\text{SGREEDY}}(S)$. In fact, we can improve the constant factor to 2 by noting that the instances of LPP that arise from the transformation from SCS have a special structure which is described in the following lemma.

LEMMA 3.1. *Let S be any set of strings and let $(G, \ell) = \text{LPP}(S)$. If $\{w, x, y, z\} \subseteq V$ with $\ell(w, y) = \max\{\ell(w, y), \ell(w, z), \ell(x, y), \ell(x, z)\}$ then $\ell(w, y) + \ell(x, z) \geq \ell(w, z) + \ell(x, y)$.*

proof. Identify w, x, y, z with the corresponding strings in S and let $\alpha = \ell(w, y)$, $\beta = \ell(x, z)$, $\gamma = \ell(w, z)$, $\delta = \ell(x, y)$. Note that if $\alpha \geq \gamma + \delta$ the result follows immediately. We will assume therefore that $\alpha \leq \gamma + \delta$.

We define some notation for designating substrings. If $s = a_1 \dots a_r$ is a string, $s[i]$ denotes the symbol a_i if $i > 0$ and a_{r+i+1} if $i < 0$. The notation $s[i, j]$ denotes the substring $s[i] \dots s[j]$.

By definition of G,

$$w[-\alpha, -1] = y[1, \alpha] \quad (1)$$
$$w[-\gamma, -1] = z[1, \gamma] \quad (2)$$
$$x[-\delta, -1] = y[1, \delta] \quad (3)$$

From this we find

$$\begin{aligned} z[1, \gamma + \delta - \alpha] &= w[-\gamma, \delta - \alpha - 1] && \text{from (2) and } \alpha \leq \gamma + \delta \\ &= y[\alpha - \gamma + 1, \delta] && \text{from (1)} \\ &= x[\alpha - \gamma - \delta, -1] && \text{from (3)} \end{aligned}$$

Hence, $\beta = \psi(x, z) \geq \gamma + \delta - \alpha$. □

THEOREM 3.1. *Let S be any set of strings. $\psi^* \leq 2\psi_{\text{SGREEDY}}(S)$.*

Proof. Let $(G, \ell) = \text{LPP}(S)$. Let $H = \{h_1, \dots, h_s\}$ be the set of edges chosen by the greedy algorithm in the order in which they were selected (that is, h_1 was selected first, h_2 second, and so forth). Define λ_i^* to be the the length of a longest path set on the subgraph of G defined by the set of edges that are still permissible after the selection of h_i. We show that for $i \in [1, s]$, $\lambda_{i-1}^* \leq 2\ell(h_i) + \lambda_i^*$. This in turn, implies the theorem.

Let $h_i = (w, y)$ and let X be a longest path in the subgraph of G defined by the set of edges that are permissible just before the selection of h_i (so $\ell(X) = \lambda_{i-1}^*$). By definition of the greedy algorithm, $\ell(w, y) = \max\{\ell(e) \mid e \in X\}$. At most three edges of X are not permissible after h_i is selected. If at most two become impermissible, then clearly $\lambda_i^* \geq \lambda_{i-1}^* - 2\ell(w, y)$ as desired.

If three edges become impermissible then one must have the form (w, z) with $z \neq y$, another the form (x, y) with $x \neq w$ and the third one, e joins the last vertex on the path containing y with the first vertex on the path containing w. This means that $X \cup \{h_1, \dots, h_{i-1}\}$ contains a path from x to z, which means that (x, z) is permissible *after* (w, y) is selected. By Lemma 3.1, $\ell(w, y) + \ell(x, z) \geq \ell(w, z) + \ell(x, y)$. Consequently,

$$2\ell(w, y) + \ell(X \cup \{(x, z)\} - \{e, (x, y), (w, z)\}) \geq \ell(X)$$

which implies $\lambda_i^* \geq \lambda_{i-1}^* - 2\ell(w, y)$. □

The bound given by Theorem 3.1 cannot be improved as can be seen by considering the set of strings mentioned at the beginning of this section. The improvement obtained for the greedy algorithm on strings raises the question of whether or not the bounds for the other approximation treated in section 2 can be improved. It turns out that they cannot. If we define

$$\begin{aligned} S = \{ \quad & \mathtt{a}^k\mathtt{x}\mathtt{b}^k, \quad \mathtt{b}^k\mathtt{x}\mathtt{c}^k, \quad \mathtt{c}^k\mathtt{x}\mathtt{d}^k, \quad \mathtt{d}^k\mathtt{x}\mathtt{e}^k, \quad \mathtt{e}^k\mathtt{x}\mathtt{f}^k, \\ & \mathtt{b}^{k-1}\mathtt{x}\mathtt{a}^k\mathtt{x}, \quad \mathtt{c}^{k-1}\mathtt{x}\mathtt{b}^k\mathtt{x}, \quad \mathtt{d}^{k-1}\mathtt{x}\mathtt{c}^k\mathtt{x}, \quad \mathtt{e}^{k-1}\mathtt{x}\mathtt{d}^k\mathtt{x}, \quad \mathtt{f}^k\mathtt{x}\mathtt{e}^k\mathtt{x}\} \end{aligned}$$

and $(G, \ell) = \text{LPP}(S)$, we find that $\lambda^*(G, \ell) = 9k$, $\lambda_{\text{MATCH}}(G, \ell) = \lambda_{\text{DIMATCH}} = 5(k + 1)$. The example can be extended to make the ratios $\lambda^*/\lambda_{\text{MATCH}}$ and $\lambda^*/\lambda_{\text{DIMATCH}}$ arbitrarily close to 2.

A naive implementation of SGREEDY takes at least quadratic time. A similar running time is obtained if one uses the transformation to LPP and then uses PGREEDY. A linear running time can be obtained however by making use of an appropriate data structure. For our purposes, a *suffix tree* T is an abstract data type representing a collection of strings $S = \{s_1, \dots, s_n\}$ on which the following operations are defined.

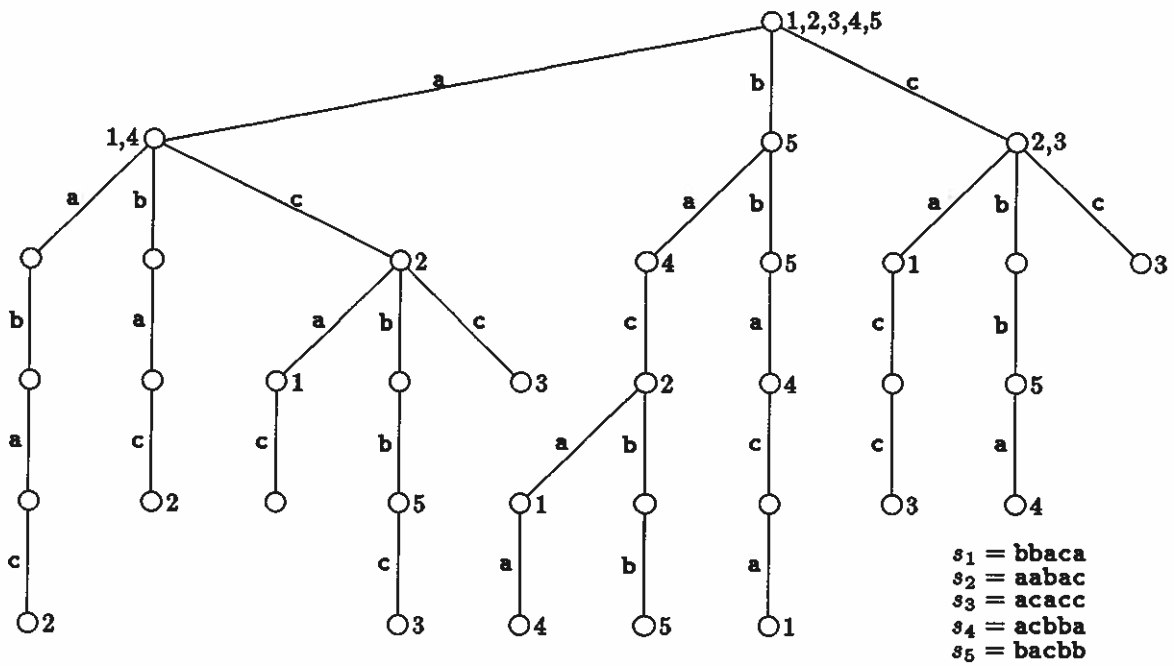

Figure 10: Example of Trie Representation for Suffix Tree

T.SUFFIX_TREE($S = \{s_1, \ldots, s_n\}$) Initialize T to represent the strings in S. This operation may only be performed once.

T.LOOKUP(integer i, j) Returns a pair $[\ell, k]$, where ℓ is the length of the longest prefix of s_i which is also a suffix of some string in $S - \{s_j\}$ and s_k is one such string.

T.DELETE(integer i) Removes s_i from the set of strings represented.

The obvious implementation of a suffix tree is a *trie* (see [1]) containing an entry for every suffix of every string in the set. An example of this representation is shown in Figure 10. The lists of integers next to some of the nodes are the indices of strings with suffixes ending at that point. This representation doesn't quite satisfy our needs, as the size of the trie and hence the time required to construct it is $\Omega(m^2)$ in the worst-case. A more compact representation can be obtained by labeling edges with strings rather than single characters. This allows us to eliminate many nodes with single children and results in a representation that requires $O(m)$ space and that can be constructed in $O(m)$ time, as described by McCreight [10]. See also [2,11]. (Actually, McCreight defines a suffix tree to contain suffixes of a single string rather than a collection of strings. Our variant requires only minor modifications to McCreight's method.) An example of this compact representation of suffix trees is shown in Figure 11

We perform deletion in suffix trees using lazy deletion. That is, to delete a string s_i, we simply mark it deleted in an auxiliary bit vector maintained for this purpose. When a lookup operation is performed, we perform a probe in the tree to find the longest match. Let u be the node at which the probe terminates. The list of matching strings at u is scanned and any that are marked deleted are removed from the list. If this makes the list empty and u has no children, then u is removed from the tree. If no acceptable match can be found in the list, the search continues at the parent of u.

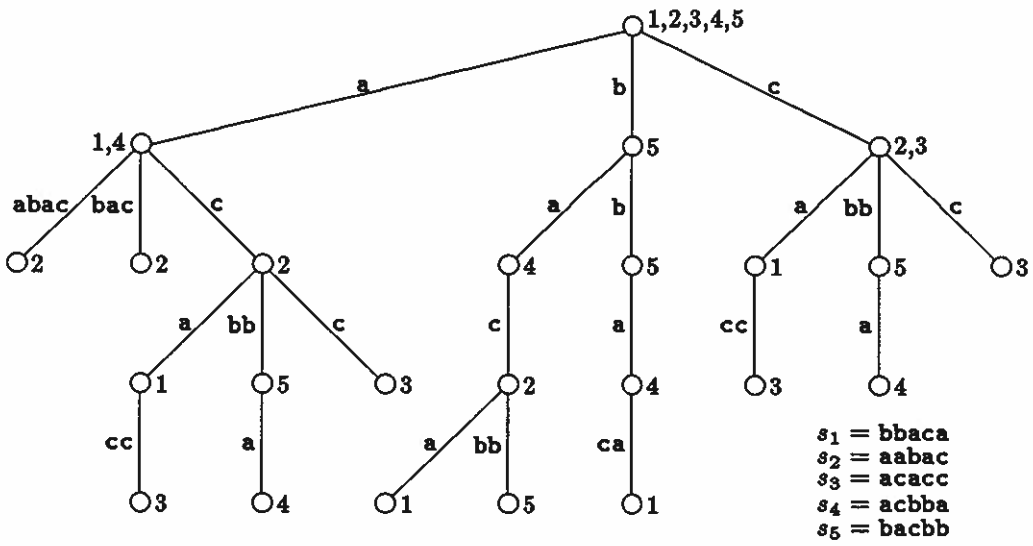

Figure 11: Example of Compacted Trie Representation

The time required for a single lookup operation may well exceed the length of the string being searched for. However, any excess time is spent deleting list entries. Since there are initially $m+n$ list entries in the whole tree, the time spent on any sequence of lookups is $O(m)$ plus the sum of the lengths of all strings being searched for.

We say that a sequence of lookup and delete operations is *monotonic* if for every i, j, k with $j \neq k$, whenever the sequence contains the operation T.LOOKUP(i, j) and and later on the operation T.LOOKUP(i, k) it contains the operation T.DELETE(j) between the other two.

We can speed up a monotonic sequence of operations, by maintaining for each string, a pointer to the node where the most recent lookup for that string ended. This allows us to avoid the initial probe of the tree when we perform a lookup operation. Instead, we use the pointer to go straight to the node where the last probe ended, and search up from that node if necessary. In this way, we can perform a monotonic sequence of r operations in $O(m+r)$ time.

This analysis assumes that the symbol alphabet is small enough that it is reasonable to use a vector of pointers to children in each node, indexed by the first symbol of the strings labeling the edges. If a large alphabet is needed, a hash table may be used. Another option is to use a variant on Sleator and Tarjan's *lexicographic splay tree* [13]. With this representation, the time required to perform a sequence of operations is $O(m)$, plus sum of the lengths of the strings being searched for, plus $O(\log m)$ per operation. For a monotonic sequence of r operations the time is $O(m + r \log m)$.

An efficient implementation of the greedy algorithm for strings is shown in Figure 12. The algorithm uses several data structures in addition to the suffix tree T. The disjoint set data structure d partitions S into sets of strings belonging to a single path. The mappings *left*(i) and *right*(i) give the left and right neighbors of i in the solution constructed so far. A value of 0 means that no neighbor is defined. The solution is returned in these vectors. The mapping *rightend*() is used to determine which string is the rightmost one in the path containing a given string. In particular *rightend*(d.FIND(i)) is the index of the rightmost string in the path containing s_i.

The heap data structure h is used to determine which pair of strings should be combined next. Each string is entered in h with the *key* being the length of the best match for h. As the algorithm

```
procedure SGREEDY(string_set S = (s_1, ..., s_n), mapping left, right : [1,n] ↦ [0,n])
    integer i, j, ℓ;
    mapping rightend : [1,n] ↦ [1,n];
    mapping key : [1,n] ↦ integer;
    disjoint_sets d; heap h; suffix_tree T;
    T.SUFFIX_TREE(S);
    for i ∈ [1,n] →
        left(i), right(i) := 0; rightend(i) := i;
        [key(i), j] := T.LOOKUP(i, i);
        d.MAKESET(i); h.INSERT(i);
    rof;
    do |h| > 1 →
        i := h.FINDMAX();
        [ℓ, j] := T.LOOKUP(i, rightend(d.FIND(i)));
        if ℓ = key(i) →
            left(i), right(j) := j, i;
            T.DELETE(j); h.DELETE(i);
            rightend(d.LINK(d.FIND(i), d.FIND(j))) := rightend(d.find(i));
         | l < key(i) →
            key(i) := ℓ; h.SIFTDOWN(i);
        fi;
    od;
end
```

Figure 12: Greedy Algorithm for SCS

proceeds, certain matches become unavailable and the values of $key()$ may become invalid. Consequently, whenever a string s_i is selected from h, a new lookup operation is performed in T. If the result of that operation is a match of the same length as $key(i)$, the strings are combined. If the lookup results in a shorter match, the value of $key(i)$ is changed and the position of s_i in the heap is adjusted to reflect the new value. Note that a string is deleted from the heap once it is successfully matched with another string on its left end. Similarly, a string is deleted from the suffix tree once it is matched with a string on the right.

The running time of the algorithm is dominated by the operations on the various data structures within the main loop. The number of iterations of the main loop is $O(m)$ in the worst case. Since the heap operations are $O(\log n)$ per operation, the total time spent on the heap operations is $O(m \log n)$. The time required for the disjoint set operations is $O(m\alpha(m,n)) = O(m \log n)$. Since the sequence of operations on the suffix tree is monotonic, the time needed for the suffix tree operations is $O(m)$ assuming a small alphabet and $O(m \log m)$ assuming a large alphabet and the use of lexicographic splay trees.

4. SCS and the Traveling Salesman Problem

In this section we relate SCS to the path version of the *traveling salesman problem* (TSP). An instance of the traveling salesman problem is a list of cities $C = (c_1, \ldots, c_n)$ with a distance $d(c_i, c_j)$ between

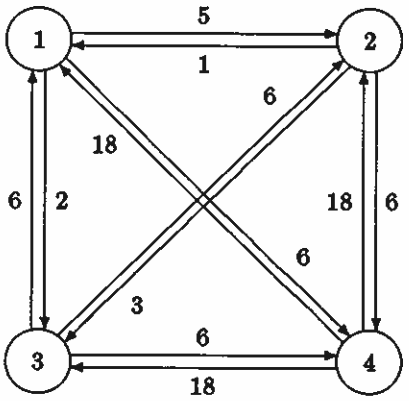

Figure 13: Example of transformation from SCS to TSP($S = \{\texttt{cbadef}, \texttt{fcbade}, \texttt{adefcd}\}$)

each pair of cities. The object is to find a permutation π on $\{1, \ldots, n\}$. that minimizes

$$\theta_\pi(C, d) = \sum_{i=1}^{n-1} d(c_{\pi_i}, c_{\pi_{i+1}})$$

We define $\theta^*(C, d) = \min_\pi \theta_\pi(C, d)$.

Let $S = (s_1, \ldots, s_n)$ be an instance of SCS. We define TSP$(s_1, \ldots, s_n)$ to be an instance (C, d) of TSP with $C = (c_1, \ldots, c_n, c_{n+1})$ and

$$d(c_i, c_j) = \begin{cases} |s_i| - \psi(s_i, s_j) & 1 \le i \le n \quad 1 \le j \le n \quad i \ne j \\ |s_i| & 1 \le i \le n \quad j = n+1 \\ \|S\| & i = n+1 \quad 1 \le j \le n \end{cases}$$

An example of this transformation is given in Figure 13. Note that in general, if π satisfies $\theta_\pi(C, d) = \theta^*(C, d)$ then $\pi_{n+1} = c_{n+1}$.

THEOREM 4.1. *Let $S = (s_1, \ldots, s_n)$ be an instance of* SCS, *$(C, d) =$* TSP*(S) and let π be a permutation on $\{1, \ldots, n, n+1\}$ to $(c_1, \ldots, c_n, c_{n+1})$ for which $\pi_{n+1} = n+1$. Then $\theta_\pi(C, d) = \phi_{\pi'}(S)$, where π' is the restriction of π to $\{1, \ldots, n\}$. In particular, $\theta^*(C, d) = \phi^*(S)$.*

proof.

$$\begin{aligned} \theta_\pi(C, d) &= \sum_{i=1}^{n} d(c_{\pi_i}, c_{\pi_{i+1}}) = \left[\sum_{i=1}^{n-1} |s_{\pi_i}| - \psi(s_{\pi_i}, s_{\pi_{i+1}})\right] + |s_{\pi_n}| \\ &= \left[\sum_{i=1}^{n} |s_{\pi_i}|\right] - \left[\sum_{i=1}^{n-1} \psi(s_{\pi_i}, s_{\pi_{i+1}})\right] = \|S\| - \psi_{\pi'}(S) = \phi_{\pi'}(S) \end{aligned}$$

$\theta^*(C, d) = \phi^*(S)$ follows from the observation that any optimum solution π for (C, d) must have $\pi_{n+1} = c_{n+1}$. □

Theorem 4.1 implies that any good approximation algorithm for this version of the traveling salesman problem is a good approximation algorithm for SCS as well. The particular instances of TSP constructed by the transformation defined above have some special properties. First, they may be asymmetric; that is $d(c_i, c_j)$ need not equal $d(c_j, c_i)$. The next theorem shows that they obey the so-called *triangle inequality*.

THEOREM 4.2. *Let* $S = (s_1, \ldots, s_n)$ *be an instance of* SCS *and let* $(C, d) = \text{TSP}(S)$. *For all* $c_i, c_j, c_k \in C$, $d(c_i, c_k) \leq d(c_i, c_j) + d(c_j, c_k)$.

Proof. There are several cases to consider. If $i, k < j = n + 1$, $d(c_i, c_k) \leq \|S\| = d(c_j, c_k)$ and the result follows immediately. Similarly, if $i, j < k = n + 1$, $d(c_i, c_j) \leq |s_i| = d(c_i, c_k)$ and if $j, k < i = n + 1$, $d(c_i, c_j) = d(c_i, c_k)$. This leaves the case where $i, j, k < n$. For convenience, let $\alpha = d(c_i, c_j)$ and $\beta = d(c_j, c_k)$ and note that

$$s_i[\alpha + 1, |s_i|] = s_j[1, |s_i| - \alpha] \quad \text{and} \quad s_j[\beta + 1, |s_j|] = s_k[1, |s_k| - \beta]$$

Note that if $|s_i| \leq \alpha + \beta$, we're done. Therefore, assume $|s_i| > \alpha + \beta$. This implies that

$$s_i[\alpha + \beta + 1, |s_i|] = s_k[1, |s_i| - (\alpha + \beta)]$$

and that $d(c_i, c_j) \leq \alpha + \beta$. □

There exist efficient approximation algorithms for the symmetric version of the traveling salesman problem with triangle inequality that produce solutions within a factor of (3/2) of optimal. When the triangle inequality doesn't hold, finding good approximate solution is as difficult as finding optimum solutions (see [7]). For the asymmetric version with triangle inequality however, little is known. There are no known approximation algorithms that are both efficient and have good worst-case performance (nor have we found any), but the approximation problem has not been shown to be hard. Consequently, the relationship between SCS and TSP has yet to yield any immediately useful results. The relationship does imply some consequences if the status of either problem is resolved in the future. If good approximation algorithms are found for TSP, they may be applied to SCS. If the approximation problem for SCS is shown to be hard, then the approximation problem for TSP must be hard. It also may be that good approximation algorithms discovered for SCS, could be adapted to TSP, although this does not necessarily follow. Finally, a proof that TSP is hard to approximate would not imply that SCS is hard to approximate, but it would imply that approximation algorithms for SCS would have to exploit special structural properties present in the instances of TSP that result from our transformation. We close by mentioning one such property that may prove useful.

LEMMA 4.1. *Let* S *be any set of strings and let* $(C, d) = \text{TSP}(S)$. *If* $\{w, x, y, z\} \subset C$ *with* $d(w, y) = \min d(w, y), d(w, z), d(x, y), d(x, z)$ *then* $d(w, y) + d(x, z) \leq d(w, z) + d(x, y)$.

We refer to this as the *quadrilateral inequality* and note that it is analagous to the property described in Lemma 3.1. As the proof is similar, we omit it.

5. Closing Remarks

We have related the shortest common matching string problem to two other problems, using transformations that preserve solution values. These two transformations reflect different ways of viewing SCS. We have used the transformations to gain insight into the problem of approximating SCS and have discovered several algorithms that have provably good performance with respect to the overlap

measure. The best of these is the string version of the greedy algorithm for which we have described an efficient implementation using suffix trees.

While we have shown that the string version of the greedy algorithm has good worst-case performance with respect to the overlap measure, we cannot determine its performance with respect to the length measure. We know that it can be off by as much as a factor of two with respect to the length measure, but we don't know if it can be worse than this. One open problem then, is to resolve this issue.

Although, we have been unable to make use of the relationship between SCS and TSP to advantage, we feel that it may yet prove useful. More generally, we think that the use of transformations that preserve solution values can be used to extend the application of known approximation algorithms to new domains. A methodical development of such transformations could provide many useful results.

Another worthwhile line of investigation for future research is to study the probable performance of the various approximation algorithms using appropriate probability models. It appears likely for example, that the directed matching algorithm for the longest path problem performs much better than its worst-case bound would indicate for a wide class of natural probability models. Similarly, one would expect the greedy algorithm to perform well in a probabilistic sense for many useful probability models.

References

[1] Aho, Alfred V., John E. Hopcroft and Jeffery D. Ullman. "Data Structures and Algorithms," Addison-Wesley, 1982.

[2] Chen, M. T. and J. I. Seiferas. "Efficient and Elegant Subword-Tree Construction," University of Rochester, Computer Science Department, technical report TR 129, 12/83.

[3] Gallant, John K., David Maier, James A. Storer. "On Finding Minimal Length Superstrings," *Journal of Computer and System Sciences*, vol. 20, no. 1, 50–58, 2/80.

[4] Gallant, John K. "String Compression Algorithms," Ph.D. Dissertation, Princeton University, Department of Electrical Engineering and Computer Science, June 1982.

[5] Gingeras, T. R., J. P. Milao, D. Sciaky and R. J. Roberts. "Computer Programs for the Assembly of DNA Sequences," *Nucleic Acids Research*, vol. 7, 1979, 529–545.

[6] Karp, Richard M. "A Patching Algorithm for the Nonsymmetric Traveling Salesman Problem," *SIAM Journal on Computing*, vol. 8, no. 4, 561–573, 11/79.

[7] Lawler, E. L., J. K. Lenstra and A. H. G. Rinnooy Kan (editors). "The Traveling Salesman Problem," John Wiley and Sons, Ltd, 1986.

[8] Maier, David and James A. Storer. "A Note on the Complexity of the Superstring Problem," Princeton University Technical Report 233, Department of Electrical Engineering and Computer Science, October 1977.

[9] Mayne, A. and E. B. James. "Information Compression by Factorising Common Strings," *The Computer Journal* 18, 1975, 157–160.

[10] McCreight, Edward M. "A Space-Economical Suffix Tree Construction Algorithm," *Journal of the ACM*, vol. 23, 4/76, 262–272.

[11] Rodeh, M., V. R. Pratt and S. Even. "Linear Algorithm for Data Compression via String Matching," *Journal of the ACM*, vol. 28, 1/81, 16–24.

[12] Shapiro, M. B. "An Algorithm for Reconstructing Protein and RNA Sequences," *Journal of the ACM*, vol. 14, 1967, 720–731.

[13] Sleator, Daniel D. and Robert E. Tarjan. "Self-Adjusting Binary Search Trees," *Journal of the ACM*, vol. 32, 7/85, 652–686.

[14] Stefik, M. "Inferring DNA Structures From Segmentation Data," *Artificial Intelligence*, vol. 11, 1978, 85–114.

[15] Storer, James A. and Thomas G. Szymanski. "Data Compression via Textual Substitution," *Journal of the ACM*, vol. 29, 10/82, 928–951.

[16] Tarjan, Robert E. "Data Structures and Network Algorithms," Society for Industrial and Applied Mathematics, 1983.