アルゴリズム 51－4
(1996. 5. 29)

# 極大パス集合に対する効率的な並列アルゴリズムとその応用

上原 隆平[†], 陳 致中[‡], Xin He[§]

[†] 東京女子大学
[‡] 東京電機大学
[§] State University of New York at Buffalo

**概要:** 与えられた無向グラフに対して、極大パス集合を求める並列アルゴリズムを 2 つ構成した。1 つめは実行時間の期待値が $O(\log n)$ 時間のランダムアルゴリズムで、これは CRCW PRAM で、プロセッサ数は $O(n+m)$ 個である。2 つめは実行時間が $O(\log^2 n)$ 時間の決定性アルゴリズムで、これは EREW PRAM で、プロセッサ数は $O(\Delta^2(n+m)/\log n)$ である。ここで $n$ は頂点数、$m$ は辺の数、$\Delta$ は最大次数である。これらの結果は既知の結果を改善し、また有効グラフ上のアルゴリズムに拡張することができる。次にこの結果を利用して Jiang らによって導入された shortest superstring problem のあるバリエーションに対する $NC$ 近似アルゴリズムを構成した。このアルゴリズムは任意の$\epsilon > 0$ に対して $\frac{1}{3+\epsilon}$ の圧縮率を達成する。

**キーワード:** 並列アルゴリズム, ランダムアルゴリズム, グラフアルゴリズム, 極大パス集合, 近似アルゴリズム, Shortest common superstrings.

# Fast *RNC* and *NC* Algorithms for Finding a Maximal Set of Paths with an Application

Ryuhei Uehara[†], Zhi-Zhong Chen[‡], Xin He[§]

[†] Tokyo Woman's Christian University
[‡] Tokyo Denki University
[§] State University of New York at Buffalo

**Abstract:** We present two parallel algorithms for finding a maximal set of paths in a given undirected graph. The former runs in $O(\log n)$ expected time with $O(n+m)$ processors on a CRCW PRAM. The latter runs in $O(\log^2 n)$ time with $O(\Delta^2(n+m)/\log n)$ processors on an EREW PRAM. The results improve on the best previous ones and can also be extended to digraphs. We then use the results to design an $NC$ approximation algorithm for a variation of the shortest superstring problem introduced by Jiang *et al.* The approximation algorithm achieves a compression ratio of $\frac{1}{3+\epsilon}$ for any $\epsilon > 0$.

**Key words:** Parallel algorithms, Randomized parallel algorithms, Graph algorithms, Maximal path sets, Approximation algorithms, Shortest common superstrings.

## 1 Introduction

The *maximal path set* (MPS) problem is to find, given an undirected graph $G = (V, E)$, a maximal subset $F$ of $E$ such that the subgraph induced by $F$ is a forest in which each connected component is a path. In [3], Chen introduced this problem and showed that parallel algorithms for it can be used to design parallel approximation algorithms for the famous shortest superstring problem (SSP). It is worth mentioning that SSP has been extensively studied due to its important applications in DNA sequencing and data compression [1, 4, 10, 11].

In [3], Chen presented an $NC$ algorithm and an $RNC$ algorithm for the MPS problem. The former runs in $O(\log^3 n)$ time with $O(n+m)$ processors on a CRCW PRAM and the latter runs in $O(\log^2 n)$ expected time with $O(n+m)$ processors on a CRCW PRAM. In this paper, we present two faster parallel algorithms for the problem. Our first algorithm runs in $O(\log n)$ expected time with $O(n+m)$ processors on a CRCW PRAM. This algorithm is faster and more efficient than Chen's $RNC$ algorithm. Our second algorithm runs in $O(\log^2 n)$ time with $O(\Delta^2(n+m)/\log n)$ processors on an EREW PRAM, where $\Delta$ is the maximum degree of the input graph. Compared with Chen's $NC$ algorithm, this algorithm is faster, runs on a weaker computation model, and is more efficient for input graphs of bounded degree.

Our $RNC$ algorithm for the MPS problem has a similar structure to that of Israeli and Itai's $RNC$ algorithm (I&I algorithm, for short) for the maximal matching problem [6]. Namely, given a graph $G$, both the I&I algorithm and our algorithm proceeds in stages; in each stage, their main jobs are to compute a random matching $M$ in a certain subgraph of $G$ and to delete (from $G$) *some* edges incident to the vertices matched by $M$. In the I&I algorithm, the expected number of edges deleted in each stage is a constant fraction of the number of edges in $G$ [6]. However, our algorithm does not have this property. Instead, we define a potential function $\phi$ and prove that in each stage, $\phi(G)$ decreases by a constant fraction *on average*. This is the key for us to obtain the desired time bound. Our $NC$ algorithm for the MPS problem is obtained by *carefully* derandomizing the $RNC$ algorithm. An immediate consequence of the results is that the parallel approximation algorithms for SSP given in [3] can be made faster.

In [7], Jiang *et al.* introduced an interesting variation of SSP. Let $S = \{s_1, \cdots, s_n\}$ be a set of $n$ strings over an alphabet $\Sigma$. A ***superstring-with-flipping*** of $S$ is a string $s$ over $\Sigma$ such that for each $i$, at least one of $s_i$ and its reverse is a substring of $s$. The ***amount of compression*** achieved by a superstring-with-flipping $s$ of $S$ is $|S| - |s|$, where $|s|$ is the length of $s$ and $|S|$ is the total length of the strings in $S$. Define SSP$_F$ to be the following problem: Given a set $S$ of strings, find a shortest superstring-with-flipping of $S$. Like SSP, SSP$_F$ is NP-hard [7] and it is of interest to design approximation algorithms for SSP$_F$. In [7], Jiang *et al.* presented a polynomial-time approximation algorithm for SSP$_F$ that produces a superstring-with-flipping whose length is at most 3 optimal. They also pointed out that there is a greedy algorithm for SSP$_F$ that produces a superstring-with-flipping by which the amount of compression achieved is at least $\frac{1}{2}$ optimal [7]. At present, no better sequential approximation algorithms for SSP$_F$ are known. Also, no parallel approximation algorithm for SSP$_F$ has been given previously. Here, using the ideas in our parallel algorithms for the MPS problem, we give an $NC$ approximation algorithm for SSP$_F$ that produces a superstring-with-flipping by which the amount of compression achieved is at least $\frac{1}{3+\epsilon}$ optimal for any $\epsilon > 0$.

Recall that the EREW PRAM is the parallel model where the processors operate synchronously and share a common memory, but no two of them are allowed simultaneous access to a memory cell (whether the access is for reading or for writing in that cell). The CRCW PRAM differs from the EREW PRAM in that both simultaneous reading and simultaneous writing to the same cell are allowed; in case of simultaneous writing, the processor with lowest index succeeds.

For lack of space, we will omit the proofs of most facts, lemmas, theorems, and corollaries from this extended abstract.

## 2 The $RNC$ algorithm

In this section, we present an $RNC$ algorithm for the MPS problem for undirected graphs. At the end of this section, we will also mention how to modify it for digraphs.

We start by giving several basic definitions. Let $G$ be an undirected graph. The vertex set and edge set of $G$ are denoted by $V(G)$ and $E(G)$, respectively. The *neighborhood* of a vertex $v$ in $G$, denoted $N_G(v)$, is the set of vertices in $G$ adjacent to $v$; $d_G(v) = |N_G(v)|$ is the *degree* of $v$ in $G$. Vertices of degree 0 are called ***isolated vertices***. For $F \subseteq E(G)$, let $G[F]$ denote the graph $(V(G), F)$. A subset $M$ of $E$ is a *matching* in $G$ if no two edges in $M$ have a common endpoint. A matching is *maximal* if it is not properly included in any other matching. We use $V(M)$ to denote the set of all vertices $v$ such that $v$ is an endpoint of some edge in a matching $M$. By a *path*, we always mean a simple path. Note that a single vertex is considered as a path (of length 0). A set $F$ of edges in $G$ is called a ***path set*** if $G[F]$ is a forest in which each connected component is a path. Intuitively speaking, if $F$ is a path set, then $G[F]$ is a collection of vertex-disjoint paths. A *maximal path set* (MPS) in $G$ is a path set that is not properly contained in another path set. The MPS problem is to find, given $G$, an MPS in $G$.

Throughout this paper, unless stated otherwise, $G$ always denotes the input (undirected) graph, $\Delta$ denotes the maximum degree of $G$, and $n$ and $m$ denote the numbers of vertices and edges in $G$, respectively. As the input representation of $G$, we assume that $V(G) = \{0, 1, \cdots, n-1\}$ and that each vertex has a list of the edges incident to it. Thus, each edge $\{i, j\}$ has two copies - one in the edge list for vertex $i$ and the other in the edge list for vertex $j$.

### 2.1 Description of the algorithm

The top-level structure of our $RNC$ algorithm is described by the following pseudo-code:

```
1: $F := \emptyset$; $G' := G$;
2: **for** each vertex $i \in G'$ **do** $R[i] := i$;
3: **while** $G'$ has at least one edge **do begin**
4:    remove all isolated vertices from $G'$;
5:    $M := FIND\_MATCH(G', R)$;
6:    $F := F \cup M$;
7:   $UPDATE(G', R, M)$
8: **end**;
```

The algorithm maintains an array $R$ for which the following is an *invariant*: For each vertex $i$ in $G'$, $R[i] = i$ if $d_{G[F]}(i) = 0$, and $R[i] = j$ if $d_{G[F]}(i) = 1$, where $j$ is the other vertex of degree 1 in the connected component of $G[F]$ containing $i$. Note that we have the invariant before the first execution of the while-loop of the algorithm.

Let us simply explain what subroutines $FIND\_MATCH$ and $UPDATE$ do. $FIND\_MATCH$ returns a *random* matching $M$ in $G'$ such that each connected component of $G[F \cup M]$ contains at most one edge in $M$. $UPDATE$ updates the array $R$ so that the invariant is kept, and deletes those edges $e$ from $G'$ such that $e \in M$ or $F \cup \{e\}$ is not a path set. Thus, by a simple induction on the number of iterations of the while-loop in the algorithm, we can show the correctness of the algorithm.

*FIND_MATCH* is the heart of our algorithm. Given $G'$ and $R$, *FIND_MATCH* performs the following steps:

**F1.** In parallel, for each $i \in V(G')$, choose a neighbor $t(i)$ at random. Let $L$ be the list of the pairs $(i, t(i))$ for the vertices $i$ in $G'$.

**F2.** In parallel, for each $j \in V(G')$, if there are two or more pairs $(i, t(i))$ in $L$ with $t(i) = j$, then choose one of them arbitrarily and delete the rest from $L$.

**F3.** Let $S$ be the set of those edges $\{i, j\}$ in $G'$ with $(i, j) \in L$ or $(j, i) \in L$. Let $H$ be the graph $(U, S)$, where $U$ is the set of endpoints of edges in $S$. (Comment: For each vertex $i$ in $H$, $d_H(i) = 1$ or $2$.)

**F4.** In parallel, for each $i \in U$, randomly select an edge incident to $i$ in $H$.

**F5.** Set $M'$ to be the set of those edges $e \in S$ such that $e$ was selected by both its endpoints in step F4.

**F6.** For each $i \in V(M')$, select $i$ if $R[i] = i$, and randomly select one of $i$ and $R[i]$ if $i < R[i]$.

**F7.** Set $M$ to be the set of those edges $e \in M'$ such that both endpoints of $e$ were selected in step F6.

**F8.** Return $M$.

It is not difficult to see that $M$ is always a matching such that each connected component of $G[F \cup M]$ contains at most one edge in $M$. Note that steps F1 through F5 have previously been used in the I&I's algorithm for maximal matching [6, 8]. Following [8], we say that a vertex $i$ in $G'$ is *good* if $\sum_{j \in N_{G'}(i)} \frac{1}{d_{G'}(j)} \geq \frac{1}{3}$, and say that an edge in $G'$ is *good* if at least one of its endpoints is good. Then, we have the following lemmas:

**Lemma 2.1** [6, 8] At least half the edges in $G'$ are good.

**Lemma 2.2** [6] For each good vertex $i$ in $G'$, $\Pr[i \in V(M')]$ is no less than a positive constant.

**Lemma 2.3** For all vertices $i$ in $G'$, $\Pr[i \in V(M) \mid i \in V(M')] \geq \frac{1}{4}$.

**Corollary 2.4** For each good vertex $i$ in $G'$, $\Pr[i \in V(M)]$ is no less than a positive constant. Consequently, for each good edge $\{i, j\}$ in $G'$, $\Pr[i \in V(M) \text{ or } j \in V(M)]$ is no less than a positive constant.

Next, let us turn to *UPDATE*. Given $G'$, $R$, and $M$, *UPDATE* performs the following steps:

**U1.** Remove the edges in $M$ from $G'$.

**U2.** In parallel, for each edge $\{i, j\} \in M$, perform the following steps:

**U2.1.** If $R[i] = i$ and $R[j] = j$, then set $R[i] = j$ and $R[j] = i$.

**U2.2.** If $R[i] = i$ and $R[j] = k \neq j$, then first set $R[i] = k$ and $R[k] = i$, next remove $j$ and all its incident edges from $G'$, and finally remove the edge $\{i, k\}$ from $G'$ if it is in $G'$.

**U2.3.** If $R[i] = k \neq i$ and $R[j] = l \neq j$, then first set $R[k] = l$ and $R[l] = k$, next remove $i$, $j$, and all their incident edges from $G'$, and finally remove the edge $\{k, l\}$ from $G'$ if it is in $G'$.

It is easy to verify that *UPDATE* really updates the array $R$ so that the invariant is kept and that *UPDATE* really deletes those edges $e$ from $G'$ such that $e \in M$ or $F \cup \{e\}$ is not a path set.

## 2.2 Complexity analysis

In this subsection, we prove the following theorem:

**Theorem 2.5** The *RNC* algorithm runs in $O(\log n)$ expected time using $O(n + m)$ processors on a CRCW PRAM.

The algorithm uses $O(n+m)$ processors; every vertex and every edge in $G'$ has a processor associated with it. Each processor associated with a vertex (resp., edge) uses one bit of its local memory to remember whether the vertex (resp., edge) has been deleted or not from $G'$.

Clearly, the first three steps of the algorithm takes $O(1)$ time with $O(n)$ processors on an EREW PRAM. We claim that each iteration of the while-loop can be done in $O(1)$ time with $O(n + m)$ processors on a CRCW PRAM. To see the claim, first observe that removing isolated vertices from $G'$ can be done in $O(1)$ time with $O(n+m)$ processors on a CRCW PRAM. According to [6], steps F1 through F5 of *FIND_MATCH* can be done in $O(1)$ time with $O(n + m)$ processors on a CRCW PRAM. Other steps of *FIND_MATCH* use no more resources. Thus, *FIND_MATCH* can be done in $O(1)$ time with $O(n + m)$ processors on a CRCW PRAM. It is also easy to see that *UPDATE* can be done in $O(1)$ time with $O(n + m)$ processors on a CRCW PRAM. This establishes the claim. In the remainder of this subsection, we will show that the expected number of iterations of the while-loop is $O(\log n)$. This together with the claim implies the theorem.

We proceed to the proof of the fact that the expected number of iterations of the while-loop is $O(\log n)$. We use a potential function argument. For a subgraph $\mathcal{G}$ of the input graph $G$ and a path set $\mathcal{F}$ in $G$, define

$$\phi(\mathcal{G}, \mathcal{F}) = \sum_{\text{edge } \{i,j\} \text{ in } \mathcal{G}} (2 - d_{G[\mathcal{F}]}(i))(2 - d_{G[\mathcal{F}]}(j)).$$

For a random variable $X$, let $\mathcal{E}X$ denote the expected value of $X$, and let $\mathcal{E}(X \mid B)$ denote the expected value of $X$ given that event $B$ occurs.

**Lemma 2.6** (Main Lemma). Fix an iteration of the while-loop. Let $G'_b$ and $G'_a$, respectively, be the graph $G'$ before and after the iteration. Similarly, let $F_b$ and $F_a$, respectively, be the path set $F$ before and after the iteration. Then, $\mathcal{E}(\phi(G'_b, F_b) - \phi(G'_a, F_a)) \geq \mathcal{E}(\phi(G'_b, F_b) - \phi(G'_b, F_a)) \geq c \cdot \phi(G'_b, F_b)$ for some constant $c > 0$.

*Proof.* For each edge $e = \{i, j\}$ in $G'_b$, let $X_e = (2 - d_{G[F_b]}(i))(2 - d_{G[F_b]}(j))$, $Y_e = (2 - d_{G[F_a]}(i))(2 - d_{G[F_a]}(j))$, $Z_e = X_e - Y_e$, and $B_e$ be the event that $i \in V(M)$ or $j \in V(M)$. Let the number of edges in $G'_b$ be $m'_b$. Clearly, $\phi(G'_b, F_b) \leq 4m'_b$.

Fix an edge $e = \{i, j\}$ in $G'_b$. We claim that $\mathcal{E}(Z_e \mid B_e) \geq 1$. To see the claim, assume that $i \in V(M)$ or $j \in V(M)$ (i.e., event $B_e$ occurs). According to the values of $d_{G[F_b]}(i)$ and $d_{G[F_b]}(j)$, we have the following four cases:

*Case 1:* $d_{G[F_b]}(i) = d_{G[F_b]}(j) = 0$. Then, we have $X_e = 4$. If both $i \in V(M)$ and $j \in V(M)$, then $Y_e = 1$; otherwise, $Y_e = 2$. Thus, $Z_e \geq 2$.

*Case 2:* $d_{G[F_b]}(i) = 0$ and $d_{G[F_b]}(j) = 1$. Then, we have $X_e = 2$. If $j \in V(M)$, then $Y_e = 0$; otherwise, $i \in V(M)$ and $Y_e = 1$. Thus, $Z_e \geq 1$.

*Case 3:* $d_{G[F_b]}(i) = 1$ and $d_{G[F_b]}(j) = 0$. This case is similar to Case 2.

*Case 4:* $d_{G[F_b]}(i) = d_{G[F_b]}(j) = 1$. Then, we have $X_e = 1$ and $Y_e = 0$. Thus, $Z_e = 1$.

Since one of the four cases must occur, we always have $Z_e \geq 1$ whenever event $B_e$ occurs. This implies that $\mathcal{E}(Z_e \mid B_e) \geq 1$, establishing the claim. From the claim, it follows that $\mathcal{E}(Z_e) \geq \mathcal{E}(Z_e \mid B_e)\Pr[B_e] \geq \Pr[B_e]$. Thus, if $e$ is good, then by Corollary 2.4, $\mathcal{E}(Z_e) \geq \Pr[B_e] \geq c'$ for some constant $c' > 0$. Combining this with the fact that $G'_a$ is a subgraph of $G'_b$, we now have

$$\begin{aligned}
&\mathcal{E}(\phi(G'_b, F_b) - \phi(G'_a, F_a)) \\
&\geq \mathcal{E}(\phi(G'_b, F_b) - \phi(G'_b, F_a)) \\
&= \mathcal{E}\Big(\sum_{\text{edge } e \text{ in } G'_b} X_e - \sum_{\text{edge } e \text{ in } G'_b} Y_e\Big) \\
&= \mathcal{E}\Big(\sum_{\text{edge } e \text{ in } G'_b} Z_e\Big) \\
&= \sum_{\text{edge } e \text{ in } G'_b} \mathcal{E}Z_e \\
&\geq \sum_{\text{good edge } e \text{ in } G'_b} \mathcal{E}Z_e \\
&\geq \sum_{\text{good edge } e \text{ in } G'_b} c' \\
&\geq c'm'_b/2.
\end{aligned}$$

The last inequality follows from Lemma 2.1. On the other hand, we have $\phi(G'_b, F_b) \leq 4m'_b$. Thus, $\mathcal{E}(\phi(G'_b, F_b) - \phi(G'_a, F_a)) \geq c'm'_b/2 \geq \frac{c'}{8}\phi(G'_b, F_b)$. This completes the proof. ∎

Note that $\phi(G', \emptyset) = 4m$ and that the while-loop is iterated until $\phi(G', F) < 1$. Thus, by Lemma 2.6 above and Theorem 1.3 in [9], we immediately have that the expected number of iterations of the while-loop is at most $\int_1^{4m} \frac{1}{cx} dx = O(\log n)$. This completes the proof of the theorem.

## 2.3 Extension to digraphs

We start by giving several basic definitions. Let $D$ be a digraph. The vertex set and arc set of $D$ are denoted by $V(D)$ and $A(D)$, respectively. For a subset $M$ of $A(D)$, we use $V(M)$ to denote the set of all vertices $v$ such that $v$ is the tail or head of some arc in $M$. The *underlying graph* of $D$ is the undirected graph $(V(D), E)$, where $E$ consists of those edges $\{u, v\}$ with $(u, v) \in A(D)$ or $(v, u) \in A(D)$. The ***tail*** and ***head*** of an arc $(u, v)$ are $u$ and $v$, respectively. The ***indegree*** (resp., ***outdegree***) of a vertex $u$ in $D$ is the number of arcs with head (resp., tail) $u$ in $D$ and is denoted by $d_D^-(u)$ (resp., $d_D^+(u)$). The ***total degree*** of a vertex $u$ is $d_D^-(u) + d_D^+(u)$ and is denoted by $d_D(u)$. Vertices of total degree 0 are called ***isolated vertices***. For $B \subseteq A(D)$, let $D[B]$ denote the digraph $(V(D), B)$. Hereafter, a path in $D$ always means a simple directed path. Note that a single vertex is considered as a path (of length 0). A set $B$ of arcs in $D$ is called a ***directed path set*** (DPS) if $D[B]$ is an acyclic digraph in which the indegree and outdegree of each vertex are both at most 1. Intuitively speaking, if $B$ is a DPS, then $D[B]$ is a collection of vertex-disjoint paths. A ***maximal directed path set*** (MDPS) in $D$ is a DPS that is not properly contained in another DPS.

Throughout this subsection, $D$ always denotes the input digraph, and $n$ and $m$ denote the numbers of vertices and arcs in $D$, respectively. As the input representation of $D$, we assume that $V(D) = \{0, 1, \cdots, n-1\}$ and that each vertex $i$ has two lists; one of the lists consists of all arcs with tail $i$ and the other consists of all arcs with head $i$.

The top-level structure of our $RNC$ algorithm for finding an MDPS in a given digraph $D$ is described by the following pseudo-code:

```
1:  B := ∅; D' := D;
2:  for each i ∈ V(D') do R[i] := i;
3:  while D' has at least one arc do begin
4:      remove all isolated vertices from D';
5:      G' := underlying graph of D';
6:      M := FIND_MATCH(G', R);
7:      M' := {(i,j) ∈ A(D) | {i,j} ∈ M} −
        {(i,j) ∈ A(D) | {i,j} ∈ M, (j,i) ∈ A(D),
        and i > j};
8:      B := B ∪ M';
9:      D_UPDATE(D', R, M')
10: end;
```

The algorithm maintains an array $R$ for which the following is an *invariant*: For each $i \in V(D')$, $R[i] = i$ if $d_{D[B]}(i) = 0$, and $R[i] = j$ if $d_{D[B]}(i) = 1$, where $j \neq i$ is the unique vertex satisfying that

$d_{D[B]}(j) = 1$ and that there is a directed path either from $i$ to $j$ or from $j$ to $i$ in $D[B]$. Note that we have the invariant before the first execution of the while-loop of the algorithm.

*D_UPDATE* updates the array $R$ so that the invariant is kept, and deletes those arcs $e$ from $D'$ such that $D \in M'$ or $D' \cup \{e\}$ is not a DPS. More precisely, given $D'$, $R$, and $M'$, *D_UPDATE* performs the following steps:

**D1.** Remove the arcs in $M'$ from $D'$.

**D2.** In parallel, for each arc $(i, j) \in M'$, perform the following steps:

- **D2.1.** If $R[i] = i$ and $R[j] = j$, then set $R[i] = j$ and $R[j] = i$, remove all arcs with tail $i$ or head $j$ from $D'$, and remove the arc $(j, i)$ from $D'$ if it is in $D'$.
- **D2.2.** If $R[i] = i$ and $R[j] = k \neq j$, then set $R[i] = k$ and $R[k] = i$, remove $j$ and all its incident arcs from $D'$, remove all arcs with tail $i$ from $D'$, and remove the arc $(k, i)$ from $D'$ if it is in $D'$.
- **D2.3.** If $R[i] = k \neq i$ and $R[j] = j$, then set $R[k] = j$ and $R[j] = k$, remove $i$ and all its incident arcs from $D'$, remove all arcs with head $j$ from $D'$, and remove the arc $(j, k)$ from $D'$ if it is in $D'$.
- **D2.4.** If $R[i] = k \neq i$ and $R[j] = l \neq j$, then set $R[k] = l$ and $R[l] = k$, remove $i$, $j$, and all their incident arcs from $D'$, and remove the arc $(l, k)$ from $D'$ if it is in $D'$.

We say that a vertex $i$ in $D'$ is *good* if it is good in $G'$ (the underlying graph of $D'$), and say that an arc in $D'$ is *good* if its tail or head is good.

**Lemma 2.7** At least one third of the arcs in $D'$ are good.

From Corollary 2.4, it is easy to see the following lemma:

**Lemma 2.8** For each good arc $(i, j)$ in $D'$, $\Pr[i \in V(M')$ or $j \in V(M')]$ is no less than a positive constant.

To prove that the expected number of iterations of the while-loop is $O(\log n)$, we need to modify the potential function in the last subsection as follows: For a subgraph $\mathcal{D}$ of the input digraph $D$ and a DPS $\mathcal{B}$ in $D$, define

$$\begin{aligned}\phi(\mathcal{D}, \mathcal{B}) &= \sum_{\text{arc } (i,j) \text{ in } \mathcal{D}} (1 - d^{+}_{D[\mathcal{B}]}(i))(1 - d^{-}_{D[\mathcal{B}]}(j)) \\ &\qquad (2 - d_{D[\mathcal{B}]}(i))(2 - d_{D[\mathcal{B}]}(j)).\end{aligned}$$

Then, using Lemma 2.7 and Lemma 2.8, we can show the following lemma by a similar proof to that of Lemma 2.6:

**Lemma 2.9** Fix an iteration of the while-loop. Let $D'_b$ and $D'_a$, respectively, be the digraph $D'$ before and after the iteration. Similarly, let $B_b$ and $B_a$, respectively, be the DPS $B$ before and after the iteration. Then, $\mathcal{E}(\phi(D'_b, B_b) - \phi(D'_a, B_a)) \geq \mathcal{E}(\phi(D'_b, B_b) - \phi(D'_b, B_a)) \geq c \cdot \phi(D'_b, B_b)$ for some constant $c > 0$.

Note that $\phi(D', \emptyset) = 4m$ and that the while-loop is iterated until $\phi(D', B) < 1$. Thus, by Lemma 2.9 above and Theorem 1.3 in [9], we immediately have that the expected number of iterations of the while-loop is at most $\int_1^{4m} \frac{1}{cx} dx = O(\log n)$. From this, it is not difficult to see that the *RNC* algorithm runs in $O(\log n)$ expected time using $O(n+m)$ processors on a CRCW PRAM. Therefore, we have:

**Theorem 2.10** An MDPS can be computed in $O(\log n)$ expected time with $O(n + m)$ processors on a CRCW PRAM.

The following corollary will be used later:

**Corollary 2.11** Given a digraph $D$ and a DPS $F$ in $D$, an MDPS $B$ in $D$ with $F \subseteq B$ can be found in $O(\log n)$ expected time with $O(n+m)$ processors on a CRCW PRAM.

## 3 The *NC* algorithm

In this section, we obtain an *NC* algorithm for the MPS problem by carefully derandomizing the *RNC* algorithm in section 2.1. Recall that the *RNC* algorithm consumes random bits only in steps F1, F4, and F6 of *FIND_MATCH*. Our first step toward derandomizing the algorithm is to make these steps consume a small number of random bits. More precisely speaking, we modify *FIND_MATCH* as follows:

**F1'.** Randomly choose $x$ and $y$ such that $0 \leq x, y \leq q - 1$, where $q$ is a (previously computed) prime with $2\Delta \leq q \leq 4\Delta$. In parallel, for each $i \in V(G')$, set $t(i)$ to be the $j$-th neighbor of $i$ in $G'$ if there is some (unique) $j$ with $(j-1)\lfloor \frac{q}{d_{G'}(i)} \rfloor \leq (x + iy) \bmod q \leq j\lfloor \frac{q}{d_{G'}(i)} \rfloor - 1$; otherwise, let $t(i)$ be undefined. Further set $L$ to be the list of all pairs $(i, t(i))$ such that $i \in V(G')$ and $t(i)$ is not undefined.

**F2'.** Same as step F2 above.

**F3'.** Same as step F3 above.

**F4'.** In parallel, for each connected component $C$ of $H$ that is a cycle, delete an arbitrary edge in $C$ from $H$. Next, partition the edges in $H$ into two matchings $M_1$ and $M_2$.

**F5'.** Randomly set $M'$ to be one of $M_1$ and $M_2$.

**F6'.** In parallel, for each connected component $C$ of $G[F \cup M']$ that is a cycle, select an arbitrary edge in $E(C) \cap M'$. Let $M_3$ be the set of the selected edges, and $M_4 = M' - M_3$.

**F7'.** Randomly set $M$ to be one of $M_3$ and $M_4$.

Note that the input parameters to the modified *FIND_MATCH* are $G'$ and $F$. That is, we do not use the array $R$ any more. Accordingly, *UPDATE* can be modified to consist of the following single step:

**U1'.** Remove from $G'$ all edges $e$ such that $e \in M$ or $F \cup \{e\}$ is not a path set.

It should be easy to see that even if we modify *FIND_MATCH* and *UPDATE* as above, the resulting *RNC* algorithm is still correct. Next, we want to show that the expected number of iterations of the while-loop in the modified *RNC* algorithm is still $O(\log n)$. To this end, first note that steps F1' through F4' have previously been used in [2], where the following lemma was shown:

**Lemma 3.1** [2] For each good vertex $i$ in $G'$, $\Pr[i \in U] \geq \frac{1}{18}$. (Recall that $U$ is the vertex set of $H$.)

From this lemma and the modified *FIND_MATCH*, it is easy to see that Corollary 2.4 still holds. This in turn implies that Lemma 2.6 still holds. Thus, in a given iteration, $\phi(G', F)$ decreases by a constant fraction on average. Now, we are ready to show our *NC* algorithm:

**ALGORITHM** *MAX_PATH_SET*

**Input:** An undirected graph $G$.

**Output:** An MPS $F$ in $G$.

**Initialization:** Set $F = \emptyset$ and $G' = G$.

```
1. Compute the maximum degree $\Delta$ of $G$ and find a prime $q$ with $2\Delta \leq q \leq 4\Delta$.
2. While $G'$ has at least one edge, perform the following steps:
    2.1. Remove all isolated vertices from $G'$.
    2.2. In parallel, for each $(x, y, b_1, b_2)$ with $0 \leq x, y \leq q-1$ and $b_1, b_2 \in \{0, 1\}$, perform the following steps:
        2.2.1. Same as step F1' above except that the first sentence is deleted.
        2.2.2. Same as step F2' above.
        2.2.3. Same as step F3' above.
        2.2.4. Same as step F4' above.
        2.2.5. If $b_1 = 0$, then set $M' = M_1$; otherwise, set $M' = M_2$.
        2.2.6. Same as step F6' above.
        2.2.7. If $b_2 = 0$, then set $M_{x,y,b_1,b_2} = M_3$; otherwise, set $M_{x,y,b_1,b_2} = M_4$.
        2.2.8. Set $m_{x,y,b_1,b_2} = \phi(G', F) - \phi(G'_{x,y,b_1,b_2}, F \cup M_{x,y,b_1,b_2})$, where $G'_{x,y,b_1,b_2}$ is the graph obtained from $G'$ by removing all edges $e$ such that $e \in M_{x,y,b_1,b_2}$ or $F \cup M_{x,y,b_1,b_2} \cup \{e\}$ is not a path set.
    2.3. Among the quadruples $(x, y, b_1, b_2)$ with $0 \leq x, y \leq q-1$ and $b_1, b_2 \in \{0, 1\}$, find a quadruple $(x, y, b_1, b_2)$ such that $m_{x,y,b_1,b_2}$ is maximized.
    2.4. Add the edges in $M_{x,y,b_1,b_2}$ to $F$.
    2.5. Remove from $G'$ all edges $e$ such that $e \in M_{x,y,b_1,b_2}$ or $F \cup \{e\}$ is not a path set.
3. Output $M$.
```

It is clear that *MAX_PATH_SET* always finds an MPS in $G$. We next analyze its complexity. Step 1 can be implemented in $O(\log^2 n)$ time with $O(\Delta^2(n+m)/\log n)$ processors on an EREW PRAM [2]. Step 2.1 can be simply done in $O(\log n)$ time with $O((n+m)/\log n)$ processors on an EREW PRAM. According to [5], the connected components in a planar graph can be computed in $O(\log n)$ time with $O((n+m)/\log n)$ processors on an EREW PRAM. Using this fact, it is not hard to see that steps 2.2.1 through 2.2.8 can be done in $O(\log n)$ time with $O((n+m)/\log n)$ processors on an EREW PRAM. Thus, step 2.2 can be done in $O(\log n)$ time with $O(\Delta^2(n+m)/\log n)$ processors on an EREW PRAM. Clearly, step 2.3 can be implemented in $O(\log \Delta)$ time with $O(\Delta^2/\log \Delta)$ processors on an EREW PRAM. The implementation of step 2.4 is trivial. Let us consider how to implement step 2.5. First observe that $F \cup \{e\}$ is not a path set if and only if either the two endpoints of $e$ are in the same connected component of $G[F]$ or at least one of the two endpoints of $e$ has degree 2 in $G[F]$. Thus, to implement step 2.5, we compute the connected component of $G[F]$ and the degrees of the vertices in $G[F]$ in $O(\log n)$ time with $O((n+m)/\log n)$ processors on an EREW PRAM. After this, we can find out all those edges $e$ such that $F \cup \{e\}$ is not a path set in $O(\log n)$ time with $O((n+m)/\log n)$ processors on an EREW PRAM. Hence, each iteration of the while-loop in *MAX_PATH_SET* takes $O(\log n)$ time with $O(\Delta^2(n+m)/\log n)$ processors on an EREW PRAM. On the other hand, by Lemma 2.6, the while-loop in *MAX_PATH_SET* is iterated at most $O(\log n)$ times. Therefore, *MAX_PATH_SET* runs in $O(\log^2 n)$ time with $O(\Delta^2(n+m)/\log n)$ processors on an EREW PRAM. This establishes the following theorem:

**Theorem 3.2** An MPS in a given undirected graph can be found in $O(\log^2 n)$ time with $O(\Delta^2(n+m)/\log n)$ processors on an EREW PRAM.

Similarly, we can prove the following theorem:

**Theorem 3.3** An MDPS in a given digraph can be found in $O(\log^2 n)$ time with $O(\Delta^2(n+m)/\log n)$ processors on an EREW PRAM.

**Corollary 3.4** Given a digraph $D$ and a DPS $F$ in $D$, an MDPS $B$ in $D$ with $F \subseteq B$ can be found in $O(\log^2 n)$ time with $O(\Delta^2(n+m)/\log n)$ processors on an EREW PRAM.

An immediate consequence of Corollary 3.4 is that Algorithm 5 in [3] can be made faster.

## 4 An application to shortest superstrings with flipping

For a string $s$, let $s^R$ denote the reverse of $s$ and $|s|$ denote the length of $s$. Let $s$ and $t$ be two distinct strings, and let $v$ be the longest string such that

$s = uv$ and $t = vw$ for some non-empty strings $u$ and $w$. $|v|$ is called the *overlap* between $s$ and $t$ and is denoted by $ov(s,t)$. By $s \circ t$, we denote the string $uvw$.

Let $S = \{s_1, s_2, \cdots, s_n\}$ be a set of strings over some alphabet $\Sigma$. Define $S^R = \{s_1^R, ..., s_n^R\}$ and $|S| = \sum_{i=1}^{n} |s_i|$. A *superstring-with-flipping* of $S$ is a string $s$ over $\Sigma$ such that for each $i$, at least one of $s_i$ and $s_i^R$ is a substring of $s$. In the sequel, a superstring-with-flipping is simply called a superstring; this should be distinguished with the usual definition of a superstring in the literature. The *amount of compression* achieved by a superstring $s$ is $|S| - |s|$. Let $opt_{com}(S)$ denote the maximum amount of compression achievable by a superstring of $S$. W.l.o.g., we assume that the set $S \cup S^R$ is *substring free*, i.e., no string in $S \cup S^R$ is a substring of any other. For simplicity of explanation, we assume that no string in $S$ is a palindrome. At the end of this section, we will point out why this assumption is not essential to our results.

The *overlap graph* of $S$ is a weighted digraph $OG(S) = (S \cup S^R, A, ov)$, where $A = \{(s,t) \mid s,t \in S \cup S^R,\ s \neq t, \text{ and } s \neq t^R\}$ and each arc $(s,t)$ has weight $ov(s,t)$. For a subgraph $D$ of $OG(S)$, the *weight* of $D$ is the total weight of the arcs in $D$ and is denoted by $ov(D)$. The *mate* of a vertex $s$ in $OG(S)$ is $s^R$. Similarly, the *mate* of an arc $e = (s,t)$ in $OG(S)$ is $(t^R, s^R)$ and is denoted by $e^R$. Note that $e$ and $e^R$ have the same weight. A (directed) path $P$ in $OG(S)$ is said to be *legal* if for every string $s \in S$, at most one of $s$ and its mate $s^R$ is on $P$. The *mate* of a legal path $P$ in $OG(S)$ is the path consists of the mates of the arcs on $P$, and is denoted by $P^R$. Note that $ov(P) = ov(P^R)$. For a legal path $P = s, t, ..., u$ in $OG(S)$, we call $s \circ t \circ \cdots \circ u$ the *string associated with* $P$. Note that the string associated with $P$ is a superstring of the strings $s$, $t, \cdots, u$ and has length $(|s| + |t| + \cdots + |u|) - ov(P)$. A legal path $P$ in $OG(S)$ is said to be *Hamiltonian* if for each string $s$ in $S$, either $s$ or its mate $s^R$ is on $P$. A *two-path cover* of $OG(S)$ is a subgraph consisting of a Hamiltonian legal path and its mate. We denote by $opt_{cov}(S)$ the weight of a maximum-weight two-path cover of $OG(S)$. Then, we have the following fact immediately:

**Fact 1** $opt_{cov}(S) = 2 \cdot opt_{com}(S)$. Moreover, the amount of compression achieved by the string associated with a Hamiltonian legal path $P$ in $OG(S)$ is $ov(P)$. Especially, the amount of compression achieved by the string associated with a maximum-weight Hamiltonian legal path is $opt_{com}(S)$.

Recall that a directed path set (DPS) in a digraph $D = (V_D, A_D)$ is a subset $B$ of $A_D$ such that the digraph $(V_D, B)$ is an acyclic digraph in which the indegree and outdegree of each vertex are both at most 1. Consider the following simple algorithm for finding a two-path cover of $OG(S)$ with large weight:

**Algorithm** *GREEDY*

**Input:** $OG(S) = (S \cup S^R, A, ov)$.

**1.** Initialize $B$ to be the empty set.

**2.** While the digraph $(S \cup S^R, B)$ is not a two-path cover of $OG(S)$, perform the following: Add to $B$ the maximum-weight arc $e$ and its mate $e^R$ such that $B \cup \{e, e^R\}$ is a DPS in $OG(S)$ but $B \cup \{f, f^R\}$ is not a DPS in $OG(S)$ for all arcs $f$ with $ov(e) < ov(f)$.

**3.** Output the two-path cover $(S \cup S^R, B)$.

The following fact was implicitly mentioned in [7]:

**Fact 2** [7] Let $P$ be one of the two paths in the two-path cover output by *GREEDY*. Then, the amount of compression achieved by the string associated with $P$ is at least $\frac{opt_{com}(S)}{2}$.

**Lemma 4.1** Suppose that the weights on the arcs in $OG(S)$ are modified in a manner such that each arc has the same weight as its mate. Let $OG'(S) = (S \cup S^R, A, w)$ be the resulting digraph. Then, the two-path cover $(S \cup S^R, B)$ output by *GREEDY* on input $OG'(S)$ has weight $\geq \frac{w(C_{max})}{3}$, where $C_{max}$ is a maximum-weight two-path cover of $OG'(S)$.

Our next goal is to parallelize *GREEDY*. To reach this goal, we need to prove several lemmas. First, several definitions are in order. An unweighted subgraph $D$ of $OG(S)$ is said to be *legal* if the mate of each vertex and each arc in $D$ is also contained in $D$. A DPS $B$ in $D$ is said to be *legal* if the mate of each arc in $B$ is also contained in $B$. A *maximal* legal DPS in $D$ is a legal DPS that is not properly contained in another legal DPS. We want to design a parallel algorithm for computing a maximal legal DPS in a legal unweighted subgraph $D$ of $OG(S)$. To this end, we modify the *RNC* algorithm in subsection 2.3 as follows. The new input is a legal unweighted subgraph $D$ of $OG(S)$ and lines 8 and 9 therein are replaced with the following five lines:

```
8': for each $s_i \in S$, randomly select one of $s_i$ and
      its mate;
9': $M_1'' := \{e \in M' \mid$ both the tail and head of $e$
      were selected in the last step$\}$;
10': $M'' := M_1'' \cup \{e \mid e$ is in $D'$ and $e^R$ is in $M_1''\}$;
11': $B := B \cup M''$;
12': $D\_UPDATE(D', R, M'')$;
```

The crucial point is that for each vertex (resp., arc) in $D'$, the vertex (resp., arc) and its mate must be removed from $D'$ in the same call of procedure *D_UPDATE*. Moreover, from Lemma 2.8 and lines 8' through 12', it is easy to see the following lemma:

**Lemma 4.2** For each good arc $(i,j)$ in $D'$, $\Pr[i \in V(M'')$ or $j \in V(M'')]$ is no less than a positive constant.

By this lemma and the discussions in subsection 2.3, we have:

**Lemma 4.3** Given a legal unweighted subgraph $D$ of $OG(S)$ and a legal DPS $F$ in $D$, a maximal legal DPS $B$ in $D$ with $F \subseteq B$ can be found in

$O(\log n)$ expected time with $O(n+m)$ processors on a CRCW PRAM, where $m$ is the number of arcs in $D$.

To decrease the number of random bits used in line 8' above, we further replace lines 8', 9',and 10' above with the following three lines:

```
8": use $M'$ to construct an undirected graph $K = (M', E_K)$, where $E_K$ consists of those edges $\{e_1, e_2\}$ such that the head or tail of $e_1$ is the mate of the head or tail of $e_2$;
9": 3-color the vertices of $K$ to partition $M'$ into three independent sets $M'_1$, $M'_2$, and $M'_3$;
10": randomly set $M''$ to be one of $M'_1$, $M'_2$, and $M'_3$;
```

It is easy to construct $K$ and 3-color it in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM. Moreover, even if lines 8' through 10' are replaced with lines 8" through 10", Lemma 4.2 still holds. This together with the discussions in section 3 implies the following lemma:

**Lemma 4.4** Given a legal unweighted subgraph $D$ of $OG(S)$ and a legal DPS $F$ in $D$, a maximal legal DPS $B$ in $D$ with $F \subseteq B$ can be found in $O(\log^2 n)$ time with $O(n^2(n+m)/\log n)$ processors on an EREW PRAM, where $m$ is the number of arcs in $D$.

Now, we are ready to present a parallelized version of *GREEDY*. This algorithm is similar to Algorithm 5 in [3].

**Algorithm** *PAR_GREEDY*
**Input:** $OG(S) = (S \cup S^R, A, ov)$.

```
1. Let $c = 1 + \frac{\epsilon}{3}$. In parallel, for each arc $e \in A$, set $lev(e) = \lceil \log_c ov(e) \rceil$ if $ov(e) > 1$, and set $lev(e) = 0$ otherwise.
2. Compute $MaxLev = \max\{lev(e) \mid e \in A\}$.
3. Set $B = \emptyset$ and $CurLev = MaxLev$.
4. While $CurLev \geq 0$, perform the following steps:
    4.1. Construct an unweighted digraph $D = (S \cup S^R, E)$ by setting $E = B \cup \{e \in A \mid lev(e) = CurLev\}$.
    4.2. Compute a maximal legal DPS $F$ in $D$ with $B \subseteq F$ and then update $B$ to be $F$.
    4.3. Decrease $CurLev$ by 1.
5. Output the digraph $(S \cup S^R, B)$.
```

**Lemma 4.5** Algorithm *PAR_GREEDY* finds a two-path cover of $OG(S)$ with weight at least $\frac{opt_{cov}(S)}{3+\epsilon}$.

**Theorem 4.6** There is a parallel approximation algorithm that produces a superstring of a given set $S$ of $n$ strings by which the amount of compression achieved is at least $\frac{1}{3+\epsilon}$ optimal for any $\epsilon > 0$. It runs in $O(\log n \cdot \log_{1+\epsilon/3} |S|)$ expected time with $O(|S|^2)$ processors on a CRCW PRAM or in $O(\log^2 n \cdot \log_{1+\epsilon/3} |S|)$ (deterministic) time with $O(|S|^2 + n^4/\log n)$ processors on an EREW PRAM.

Finally, we explain how to remove the assumption that no string in $S$ is a palindrome. Suppose some strings in $S$ are palindromes. We redefine $OG(S)$ as follows. For each string $s$ in $S$, we introduce two vertices one of which corresponds to $s$ and the other corresponds to $s^R$. That is, we treat $s$ and $s^R$ as different *vertices* in $OG(S)$, even if $s$ is a palindrome. The edges of $OG(S)$ and their weights are defined as before. Redefining $OG(S)$ in this way, we give no influence on the above discussions.

## References

[1] A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis, Linear approximation of shortest superstrings, *23rd STOC*, 1991, pp. 328-336.

[2] Z.-Z. Chen, A fast and efficient *NC* algorithm for maximal matching, *Inf. Proc. Lett.*, **55** (1995), 303-307.

[3] Z.-Z. Chen, *NC* algorithms for finding a maximal set of paths with application to compressing strings, *22nd ICALP*, LNCS vol. 944, Springer-Verlag, 1995, pp. 99-110; journal version to appear in *Theoretical Computer Science*.

[4] A. Czumaj, L. Gasieniec, M. Piotrow, and W. Rytter, Parallel and sequential approximation of shortest superstrings, *4th SWAT*, LNCS vol. 824, Springer-Verlag, 1994, pp. 95-106.

[5] H. Gazit, Optimal EREW parallel algorithms for connectivity, ear decomposition and st-numbering of planar graphs, *Proc. of the 5th IEEE International Parallel Processing Symposium*, 1991, pp. 84-91.

[6] A. Israeli and A. Itai, A fast and simple randomized parallel algorithm for maximal matching, *Inf. Proc. Lett.*, **22** (1986), 77-80.

[7] T. Jiang, M. Li, and D.-z. Du, A note on shortest superstrings with flipping, *Inf. Proc. Lett.*, **44** (1992), 195-199.

[8] D.C. Kozen, *The Design and Analysis of Algorithms*, Springer, New York, 1992.

[9] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[10] S. Rao Kosaraju, J.K. Park, and C. Stein, Long tours and short superstrings, *35th FOCS*, 1994, pp. 166-177.

[11] S.-H. Teng and F. Yao, Approximating shortest superstrings, *34th FOCS*, 1993, pp. 158-165.

[12] J.-S. Turner, Approximation algorithms for the shortest common superstring problem, *Inf. and Comp.*, **83** (1989), 1-20.