November, 1993
This paper appeared in Utilitas Mathematica 45 (1994), 169-185.

An Algorithm for Finding a Long Path in a Graph

William Kocay* and Pak-Ching Li
Computer Science Department
University of Manitoba
Winnipeg, Manitoba, CANADA, R3T 2N2
e-mail: bkocay@cs.umanitoba.ca

Abstract

An algorithm is described which constructs a long path containing a selected vertex x in a graph G. In hamiltonian graphs, it often finds a hamilton cycle or path. The algorithm uses crossovers of order $k \leq M$, where M is a fixed constant, to build a longer and longer path. The method is based on theoretical methods often used to prove graphs hamiltonian.

1. Crossovers

Let G be a 2-connected undirected simple graph on n vertices. If $u, v \in V(G)$, then $u \to v$ means that u is adjacent to v (and so also $v \to u$). The reader is referred to [4] for other graph-theoretic terminology. In particular, a *trail* in G is a walk in which vertices may be repeated, but not edges. Since G is simple, paths and trails may be represented as sequences of vertices. This uniquely defines their edge-sets. If $(w_0, w_1, \ldots, w_m)$ represents a trail Q, we use Q to denote both the trail itself (ie, a subgraph of G), its sequence of vertices, as well as its set of edges. The usage should always be clear from the context. Let x be a vertex of G. We want to find a long path P in G containing x. Initially, $P = (x)$, a path of length 0. We then extend the path P as follows.

$u := x$; $\quad v := x$
while $\exists w \to u$ such that $w \notin P$ do $P := P + uw$; $u := w$; end
while $\exists w \to v$ such that $w \notin P$ do $P := P + vw$; $v := w$; end

At this point we have a uv-path $P = (u, \ldots, x, \ldots, v)$ such that the endpoints u and v are adjacent only to vertices of P. The length of P is $\ell(P)$, the number of edges in P. The vertices of P are ordered from u to v. If $w \in P$, then w^+ indicates the vertex following w (if $w \neq v$). Similarly w^-

* This work was supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada.

indicates the vertex preceding w (if $w \neq u$). If $x, y \in P$ are such that x precedes y in this ordering, we write $x < y$ on P.

If $u \to v$ then we have a cycle $C = P + uv$. As G is connected, there is a vertex $w \in P$ such that $w \to y$, where $y \notin P$. Hence there exists a longer path $P^* := P - ww^+ + wy$. (The only time no such w exists is when C is a hamilton cycle.) Fig. 1 shows a cycle $C = P + uw - ww^- + vw^-$ formed by a "crossover" pattern.

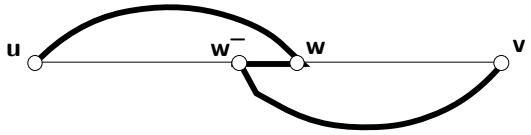

Fig. 1, uv-path P and trail $Q = (u, w, w^-, v)$

Shown in Fig. 2 are some sample cycles C, where $V(C) = V(P)$, such that C contains 3 edges not in P.

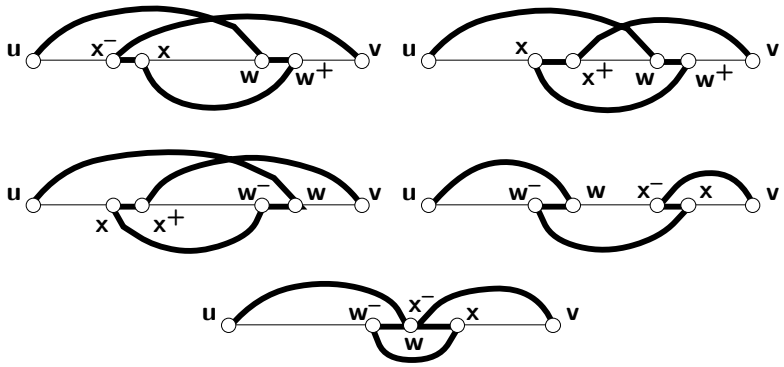

Fig. 2, Crossovers of order 2

1.1 Definition. Let P be a uv-path. A *crossover* Q is a uv-trail such that $V(Q) \subseteq V(P)$ and $C := P \oplus Q$ is a cycle with $V(C) = V(P)$. ($\oplus$ indicates the operation of exclusive-OR, applied to the edges of P and Q.) The *order* of a crossover Q is the number $|P \cap Q|$. A *cross-edge* is any edge $xy \in E(Q) - E(P)$.

So a crossover of order 0 occurs when $u \to v$. Then $Q = (u, v)$ and $C = P + uv$. There is one crossover of order 1, shown in Fig. 1. There are 5 possible kinds of crossovers of order 2, shown in Fig. 2.

We want to find a crossover because we can then convert P into a cycle $C = P \oplus Q$. We can then find a vertex $w \in C$ such that $w \to y \notin C$. This gives a new path $P^* := P - ww^+ + wy$ that is longer than P. We can now set $P := P^*$, and repeat the process of extending P and looking

for a crossover. Eventually we either find a hamilton path in G, or else we reach a situation where P has no crossover. It is easy to prove that if $\deg(u)+\deg(v) \geq n-1$, for all non-adjacent vertices u and v, then there will always be a crossover of order ≤ 1, and that a hamilton path can always be found in this way. This method was used in the *Groups & Graphs* program [5] to look for long paths. It works very well in graphs with many edges, even if the condition $\deg(u)+\deg(v) \geq n-1$ is not satisfied. But in graphs with few edges, it does not often find a long path. The methods described in this paper provide a means of improving the algorithm substantially.

To find a crossover, we have to do an exhaustive search of all the possibilities. We build a trail Q from u such that $V(Q) \subseteq V(P)$. So Q is always a uw-trail for some $w \in P$. We say the order of the trail Q is $|P \cap Q|$. If at any point, $w \rightarrow v$, then $Q + wv$ will be a crossover iff $P \oplus Q$ is a cycle. A recursive algorithm for finding a crossover is given here. The number of trails from u is typically exponential in n, the number of vertices of G. We therefore choose a maximum order M, and limit the search for crossovers up to that order.

M is a global variable, indicating the maximum allowable order.
P is a global variable, a uv-path.
u and v are global variables, the endpoints of P.
Q is a global variable, a uw-trail, such that $V(Q) \subseteq V(P)$.
CrossOver is a global boolean variable

FindCrossover(w: vertex; k: integer)
{ Extend trail Q from vertex w. The current order of Q is k }
begin
  CrossOver := *false* { assume no crossover will be found }
  if $k > M$ then Exit
  if $w \rightarrow v$ then check if $Q + wv$ is a valid crossover
  if so, set *CrossOver* := *true* and Exit
  for all $x \rightarrow w$, such that $x \in P - Q$, $x \neq w^{\pm}$, and $\deg(x, Q) \leq 2$ do
  begin
    $Q := Q + wx$
    if $x \neq u$ then if $xx^- \notin E(Q)$ then begin
      $Q := Q + xx^-$
      FindCrossover(x^-, $k+1$)
      if *CrossOver* = *true* then Exit
      $Q := Q - xx^-$
    end
    if $x \neq v$ then if if $xx^+ \notin E(Q)$ then begin
      $Q := Q + xx^+$
      FindCrossover(x^+, $k+1$)

```
        if CrossOver = true then Exit
        Q := Q − xx⁺
      end
      Q := Q − wx
    end
  end { FindCrossover }
```

The algorithm is executed by choosing a value for M, initiallizing $Q = (u)$, and calling *FindCrossover*$(u, 0)$. It builds trails Q satisfying the following intersection property.

1.2 Property. Let P be a uv-path, and let $Q = (w_0, w_1, \ldots, w_m)$ be a uw-trail, where $w_0 = u$, $w_m = w$, and $V(Q) \subseteq V(P)$. Then Q has the *intersection property* if for each w_i, where $1 \leq i < m$, exactly one of w_{i+1} and w_{i-1} equals $w_i^{\pm}$.

1.3 Theorem. *If G has a crossover Q of order k, then there exists a crossover $Q' = (w_0, w_1, \ldots, w_m)$ of order k such that $P \oplus Q' = P \oplus Q$, and Q' has the intersection property.*

Proof. Let $Q = (w_0, w_1, \ldots, w_m)$ be a crossover of order k, where $w_0 = u$ and $w_m = v$. Then $C = P \oplus Q$ is a cycle, so every vertex of C has degree 2. The edges of P and Q can be divided into those of $P - Q$, $Q - P$, and $P \cap Q$. If $w \in C$ then $\deg(w, C) = \deg(w, P) + \deg(w, Q) - 2\deg(w, P \cap Q) = 2$. Therefore $\deg(w, Q) = 2 + 2\deg(w, P \cap Q) - \deg(w, P) \leq 2 + \deg(w, P) \leq 4$. If $\deg(w, Q) = 2$, then $\deg(w, P) = 2$ and $\deg(w, P \cap Q) = 1$. So $w = w_i$ where $1 \leq i < m$, and exactly one of w_{i+1} and w_{i-1} equals $w^{\pm}$. If $\deg(w, Q) = 3$, then $\deg(w, P) = 1$, so that w is one of u or v. If $\deg(w, Q) = 4$, then $\deg(w, P) = 2$ and $\deg(w, P \cap Q) = 2$. Edges ww^+ and ww^- are both in Q. Two vertices w_i and w_j of Q correspond to w. If the theorem is not true, we can assume that one of w_i and w_j, say w_i, has $\{w_{i+1}, w_{i-1}\} = \{w^+, w^-\}$. Write $Q = (w_0, \ldots, w^{\pm}, w_i, w^{\mp}, \ldots, w_j, \ldots, w_m)$. Since $w_i = w_j = w$, we can reverse the portion of Q between w_i and w_j to get a crossover $Q' = (w_0, \ldots, w^{\pm}, w_i, w_{j-1}, \ldots, w^{\mp}, w_i, w_{j+1}, \ldots, w_m)$. Q' has two fewer vertices at which the intersection property does not hold. We can continue until Q is transformed into a crossover of the required type.

1.4 Lemma. *When FindCrossover(w, k) is entered, $Q = (w_0, w_1, \ldots, w_m)$ is a trail of order k from $u = w_0$ to $w = w_m$ satisfying the intersection property.*

Proof. By induction on k. This is true intially when $k = 0$ and $Q = (w_0)$ where $w_0 = u$. When *FindCrossover*(w, k) is entered, it first checks whether $w \to v$. If so, $Q + wv$ will be a crossover of order k iff $P \oplus Q$ is a cycle. Otherwise all $x \to w$ are considered, such that $x \in P - Q$, $x \neq w^{\pm}$ and $\deg(x, Q) \leq 2$. For each such x, Q is extended by the edge wx, so $w_{m+1} = x$.

Notice that $w_{m+1} \neq w^{\pm}$. Then Q is in turn extended by the edges xx^- and xx^+ if possible, and a recursive call is made, increasing k by 1 since the edge $xx^{\pm} \in P \cap Q$. Notice that $w_{m+2} = x^{\pm}$, so that the condition that exactly one of w_{i+1} and w_{i-1} equals $w_i^{\pm}$ is satisfied at the 2 new vertices w_{m+1} and w_{m+2}. The condition $\deg(x, Q) \leq 2$ is required since $\deg(x, Q)$ will be increased by 2 (edges wx and $xx^{\pm}$), and we must always have $\deg(x, Q) \leq 4$. By induction the trails Q constructed satisfy the intersection property.

1.5 Theorem. *FindCrossover will find a crossover iff G contains a crossover of order $k \leq M$.*

Proof. By lemma 1.3 if G has a crossover of order k, then it has a crossover of order k with property 1.2. Since the algorithm recursively builds all trails with this property, it will find a crossover if one exists.

Most of the operations in the algorithm can be implemented quite easily. The difficult one is to check whether $Q + wv$ forms a valid crossover. We must determine whether $C = P \oplus Q$ is a cycle or not. The conditions imposed on Q ensure that $\deg(w, C) = 2$ for every $w \in P$. However it is most often the case that C is a union of several cycles, instead of a single cycle. The most direct way to test Q is to start at u and follow the edges of $P \oplus Q$ until we return to u. If the number of edges traversed is $\ell(P) + 1$, then Q is a crossover. The trouble with this is that it takes up to $\ell(P) + 1$ steps to execute. Most of the trails constructed will not be crossovers, and this means too much time spent checking invalid trails.

1.6 Definition. Let P be a uv-path, and Q a trail from u with property 1.3. A *segment* of P with respect to Q is any connected component of $P - E(Q)$. $S(Q)$ is the set of segments of P wrt Q.

The segments that we will be dealing with are the segments created by the *FindCrossover* algorithm above. Initially when $Q = (u)$, there is only one segment P. In the first crossover of Fig. 2 there are three segments $P_{ux^-}, P_{xw}, P_{w^+v}$, where P_{xy} denotes the sub-path of P from x to y. The number of segments always equals $k + 1$, where k is the order of Q. Since each segment is a sub-path of P, it has 2 endpoints. Each endpoint will usually be connected to another vertex of P by a cross-edge.

1.7 Definition. The *segment graph* of P with respect to Q is $SG(Q)$. Its vertices are the segments $S(Q)$. Segments $S_i, S_j \in S(Q)$ are adjacent if there is a cross-edge connecting an endpoint of S_i to an endpoint of S_j.

An example of a trail Q and its segment graph $SG(Q)$ is illustrated in Figs. 3 and 4. In Fig. 4, the 2 endpoints of each S_i are also shown.

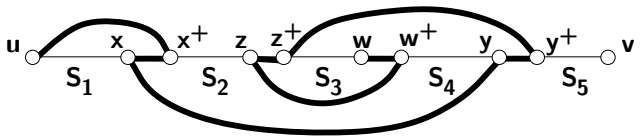

Fig. 3, A uw-trail Q of order 4

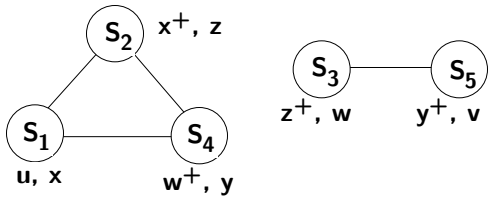

Fig. 4, The segment graph $SG(Q)$

1.8 Lemma. *There is a one-to-one correspondence between the components of $P \oplus Q$ and the components of $SG(Q)$.*

Proof. The segments are the connected components of $P - E(Q)$. Each connected component of $P \oplus Q$ consists of segments connected by cross-edges. In $SG(Q)$ the segments are represented as vertices.

If Q is a uw-trail, the components of $P \oplus Q$ will consist of a path connecting v to w, and possibly several cycles. When *FindCrossover*(w, k) is entered, w is the endpoint of a segment S_i. S_i is the endpoint of a path-component of $SG(Q)$. The other endpoint is a segment containing v. If $w \to v$, then the addition of edge vw creates a cycle. This gives:

1.9 Lemma. *Suppose that $w \to v$ when FindCrossover(w, k) is entered. Then $Q + vw$ is a crossover of order k iff the number of components of $SG(Q)$ is 1.*

So we need to know the number of components of SG. Write N for this number. Vertex w is an endpoint of a segment S_i. If $w \not\to v$, or if $Q + vw$ is not a crossover, a vertex $x \to w$ is selected. Then $x \in S_j$, for some j. Given any vertex x we need to be able to determine the segment S_j containing x, and the component of SG containing S_j. Edges xx^+ and xx^- are in turn added to Q. This splits S_j into two segments. There are three possible situations that can arise.

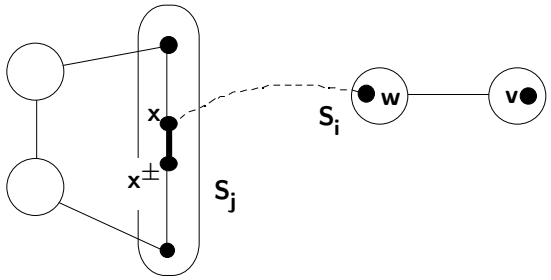

Fig. 5, Case 1, a segment is split

1. S_i and S_j are in different components of SG. N decreases by 1.
   In this case, w and x are in different components of $P \oplus Q$. The cross-edge wx connects the two. Since S_j is in a cycle of SG, x and $x^{\pm}$ will still be connected when the edge $xx^{\pm}$ has been deleted. $x^{\pm}$ becomes the new endpoint of the component containing S_i.

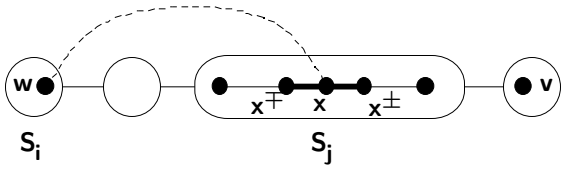

Fig. 6, Cases 2 and 3, a segment is split

2. S_i and S_j are in the same component of SG. N increases by 1.
   In this case, w and x are in the same component of $P \oplus Q$, a path connecting w to v. The path is split into two paths when $xx^{\pm}$ is deleted. If $x^{\pm}$ lies between x and v on this path, S_i will split into a cycle and a path.
3. S_i and S_j are in the same component of SG. N is unchanged.
   This is like case 2 above, except that $x^{\pm}$ lies between w and x on the path. The path component is rerouted but remains a path.

Therefore we can keep track of the number of components of SG as follows. Initially, $N = 1$. w is in segment S_i, which is part of a path component of SG. When $x \to w$ is selected, we determine which segment S_j contains x. We then determine which component S_j is in. If S_i and S_j are in different components then $N := N - 1$. If they are in the same component, we need to distinguish cases 2 and 3. If case 2 applies, we then execute $N := N + 1$.

The use of the segment graph allows us to determine immediately whether $Q+vw$ is a crossover or not. There is some extra overhead involved,

since the segment graph must be constructed. We represent the path P as a linked list of nodes.

```
NodePtr = ^Node
Node = record
        Pt: Integer   (a point x on the path)
        NextNode: NodePtr   (the next node x+ on the path)
        PrevNode: NodePtr   (the previous node x- on the path)
    end
```

A segment is represented by a data structure containing the following items.

```
SegmentPtr = ^Segment
Segment = record
        LeftPt, RightPt: Integer   (left & right endpoints of segment)
        LeftSeg: SegmentPtr   (adjacent segment on left)
        RightSeg: SegmentPtr   (adjacent segment on right)
        LeftAdjPt: Integer   (adjacent pt in segment on left)
        RightAdjPt: Integer   (adjacent pt in segment on right)
    end
```

The operations which must be perfomed on the segments are as follows.

- Given any vertex x, we need to be able to determine quickly which segment S_j contains x.

  This can be accomplished by keeping the segments in a binary tree. Each segment has a left endpoint and a right endpoint, which defines an ordering on the segments. The number of segments equals $k+1$, where k is the order of Q. Since this is limited to a previously chosen maximum M, searching the tree will be fast.
- Given any segment S_i we need to be able to determine which component of SG contains S_i.

  This can be accomplished by keeping a representative segment for each component, and using a "merge-find" array [1] of pointers toward the representative.
- If S_i and S_j are in the same component, we neeed to distinguish between cases 2 and 3.

  One way to do this is to follow the path in SG containing $x^{\pm}$ until a segment is encountered whose endpoint is either w or v. The path-component of SG corresponds to a vw-path in $P \oplus Q$; however we do not follow the path in $P \oplus Q$, since its length is $O(\ell(P))$. The length of the path-component of SG is limited by the number of segments, which is $\leq M$. Therefore this will be fast.

There are a number of other details which must be considered when the algorithm is programmed. These will become apparent to anyone programming it.

Some results of the implementation of the algorithm to find crossovers up to order $M = 10$ are listed below. Results are shown for the Horton graph and the Grinberg graph (see [4] for a description of these graphs). These are non-hamiltonian 3-regular graphs on 92 and 46 vertices, respectively. The tables show the number of edges in the longest path P that was found, starting from vertex x, and the time taken. The numbering of the vertices is arbitrary. Its only significance is that $\ell(P)$ depends on the starting vertex. The program ran under System 7.1 on a Macintosh IIfx with 20 Meg of RAM. 1 tick = $\frac{1}{60}$ sec.

x	# ticks	$\ell(P)$
1	11	40
2	10	42
3	10	41
4	15	42
5	13	42
18	10	51
19	11	67
20	11	46
21	11	59
40	11	42
41	10	42
42	10	36
43	10	44
44	10	44
56	11	42
57	10	36
58	10	30
59	10	36
66	11	66
67	12	61
68	12	64
69	12	51
88	10	60
89	12	42
90	11	58
91	11	42
92	10	31

Horton graph, 92 vertices

x	# ticks	$\ell(P)$
1	14	45
2	18	45
3	11	45
4	14	45
5	14	45
6	13	45
16	7	29
17	9	45
18	11	45
19	17	45
20	11	45
40	14	45
41	7	29
42	6	29
43	13	45

Grinberg graph, 46 vertices

Notice that for the Grinberg graph, a hamilton path was found in most cases, even though each vertex has degree 3 only. For the Horton graph, which has a hamilton path, the longest path found was usually much too short. The next section describes a method of improving the performance.

2. Inner Vertices.

Let P be a uv-path in G, and let $H = G - V(P)$. H is a subgraph of G. We call the vertices of H the *inner vertices* of G with respect to P. H consists of a number of connected components $H_1, H_2, \ldots, H_k$, where $k \geq 1$. If $x, y \in V(P)$, then P_{xy} denotes the xy-subpath of P. Suppose that some component H_i contains vertices a and b and let R_{ab} denote an ab-path in H_i. If $\exists x, y \in V(P)$ such that $a \to x$, $b \to y$, and $\ell(R_{ab}) + 2 > \ell(P_{xy})$, then we can construct a path longer than P by replacing P_{xy} with the path $(x, a, \ldots, b, y)$ through H. Any path $R_{ab} \subseteq H$ connecting $x, y \in P$ is called an *xy-bypass.*

The components of H can easily be found by either a breadth-first search or depth-first search, and both of these search methods will construct a spanning tree for each H_i. Let T_i be a spanning tree of H_i. Then T_i contains a unique ab-path R_{ab}, for all $a, b \in V(H_i)$. Since we are looking for a long path, and a DFS tends to construct longer paths than a BFS, we use a DFS. Given any a and b, we need to be able to determine quickly the length of R_{ab}, and to find this path if it is to replace P_{xy}.

Let r_0 denote the root of a DF-tree T. The edges of T are called *tree-edges.* For each $v \in V(T)$, define

$D(v)$ = the DF-number of v, the order in which the vertices are visited;
$a(v)$ = the parent of v in T;
$r(v)$ = the rank of v, its distance in T from r_0;

If we start at any vertex $v \in T$ and successively construct $v_0 = v$, $v_1 = a(v_0)$, $v_2 = a(v_1)$, etc., we eventually reach the root r_0. Let T_v denote this path $(v_0, v_1, \ldots, r_0)$. We then further define

$b(v)$ = the *branch point* of v. This is defined as follows:
$b(r_0) = r_0$. If $v \neq r_0$, let $u = a(v)$. Then

$$b(v) = \begin{cases} b(u), & \text{if } uv \text{ is the first tree-edge incident on } u \\ u, & \text{otherwise.} \end{cases}$$

$s(v)$ = the *stem* of the branch containing v; namely $s(v)$ is the unique vertex $w \in T_v$ such that $a(w) = b(v)$. $s(r_0)$ is not defined.

These quantities are easily computed by the DFS. *DFCount* is a global counter initially set to 0. $D(v)$ is initially set to 0 for all v. $r(r_0)$ and $a(r_0)$ are intiallized to 0, and $b(r_0)$ is initiallized to r_0.

```
DFS(u)
begin
  DFCount := DFCount + 1
  D(u) = DFCount
  FirstTime := true    { indicates first tree-edge incident on u }
  for each v → u do if D(v) = 0 then begin
    a(v) := u;   r(v) := r(u) + 1
    if FirstTime then b(v) := b(u) else b(v) := u
    s(u) := v { temporary assignment for the branch to be searched }
    DFS(v)
    FirstTime := false
  end
  s(u) := s(b(u))
end   { DFS }
```

The *leaves* of T are those those vertices with no descendants. Write $L(T)$ for the set of leaves of T. For each $v \in V(T)$, P_v denotes the path in T from v to $b(v)$.

2.1 Lemma. *$\{P_w \mid w \in L(T)\}$ is a partition of T into paths.*
Proof. By induction on $|L(T)|$. If $|L(T)| = 1$, then T consists of a path from $v \in L(T)$ to $r_0 = b(v)$. Assume the result is true when $|L(T)| \leq k$, and suppose that T has $k+1$ leaves. Let v be the last leaf of T visited by the DFS. Let $u = b(v)$. P_v is a path from v to u. Then $T' = T - P_v$ is a tree with k leaves, such that $L(T') = L(T) - \{v\}$. Since the result holds for T', we have $\{P_w \mid w \in L(T)\} = \{P_w \mid w \in L(T')\} \cup \{P_v\}$ is a partition of T into paths.

Given these data structures, we can compute $\ell(R_{ab})$ given any $a, b \in H_i$.

```
w := a;   z := b;   ℓ := 0
while b(w) ≠ b(z) do begin
  if D(b(w)) < D(b(z)) then   ℓ := ℓ + r(z) − r(b(z));   z := b(z)
  else   ℓ := ℓ + r(w) − r(b(w));   w := b(w)
end
if s(w) = s(z) then ℓ(R_ab) := ℓ + |r(w) − r(z)|
else ℓ(R_ab) := ℓ + r(w) + r(z) − 2r(b(w))
```

2.2 Lemma. *The above statements correctly compute* $\ell(R_{ab})$.

Proof. If w and z have the same branch point, then either they are in the same branch of T at $b(w)$, in which case $s(w) = s(z)$, or else they are in different branches, in which case $s(w) \neq s(z)$. If they are in the same branch, then either $P_w \subseteq P_z$ or $P_z \subseteq P_w$. The distance between w and z on this path is $|r(w) - r(z)|$. If they are in different branches, then P_w and P_z are both paths to $b(w)$, of lengths $r(w) - r(b(w))$ and $r(z) - r(b(z))$. Therefore $\ell(R_{wz}) = r(w) + r(z) - 2r(b(w))$.

If $b(w) \neq b(z)$, then the branch point with the larger DF-number, say $b(z)$, is chosen. The path R_{wz} contains P_z as a subpath. Its length is $r(z) - r(b(z))$. This is added to the cumulative length ℓ.

The path R_{ab} can be easily constructed.

```
w := a;   z := b
while w ≠ z do
  if D(w) < D(z) then z := a(z)
  else w := a(w)
```

The algorithm to find a long path is now altered as follows. First, crossovers are used to find a path P as long as possible with no crossover of order $\leq M$. Then the inner vertices are used to create a longer path. For each component H_i, a spanning tree T_i is constructed. As T_i is built, a linked list A_i is formed consisting of all vertices $y \in P$ such that y is adjacent to some $b \in H_i$. The vertex b is saved in the node of A_i corresponding to y.

```
for each x ∈ P do
  for each a → x do if a ∉ P then begin
    find i such that a ∈ H_i
    if T_i has not been built then construct T_i and the list A_i
    for each y ∈ A_i do begin
      suppose that y → b ∈ H_i
      if ℓ(R_ab) + 2 > ℓ(P_xy) then
        replace P_xy with (x, a, ..., b, y) and Exit
    end
  end
```

If a sufficiently long bypass R_{ab} is found, then P is replaced by a longer path. This changes the inner vertices $H = G - V(P)$ so that the spanning trees T_i and lists A_i must be discarded. The new path P may now have a crossover, so the algorithm is repeated in order to search for a crossover again. If no bypass is found this does not mean that one does not exist. There are too many paths in H to search them all. Therefore the algorithm constructs a single spanning tree T_i for each H_i and uses only the paths of T_i.

There is another technique involving the inner vertices that can also be used to construct a longer path P. Let R_{ab} be a path in H, where $x \to a$, $y \to b$, and $x, y \in P$. If $x^- \to y^-$ or $x^+ \to y^+$, then a longer path can be found as illustrated in Fig. 7.

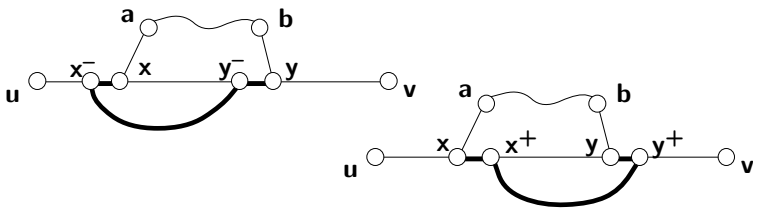

Fig. 7, Bypass crossovers

2.3 Definition. Let R_{ab} be an xy-bypass for the uv-path P. A *bypass crossover* Q wrt R_{ab} is an xy-trail such that $V(Q) \subseteq V(P)$ and $P \oplus Q \oplus R_{ab}$ is a uv-path. The *order* of Q is $|P \cap Q|$.

The examples of Fig. 7 have bypass crossovers (x, x^-, y^-, y) and (x, x^+, y^+, y) of order 2. As before we pick a maximum order M' and search for crossovers of order $\leq M'$. The algorithm *FindBypassCrossover*(w, k) is similar to *FindCrossover*. An xw-trail Q with an intersection property similar to 1.2 is constructed. We build the segment graph $SG(Q)$ whose vertices are the components of $P \oplus Q$. We do not use the edges R_{ab} except for rerouting the path once the bypass crossover is found. The components of SG consist now of two paths, and possibly some cycles. The components of $P \oplus Q$ include a ux-path and a yv-path. At each iteration, the vertices w^+ and w^- are considered. If one of these equals y, there exists an xy-trail. If the number of components $N = 2$, then $Q + yw$ is a valid bypass crossover. Otherwise recursion is used up to $k = M'$.

The complete algorithm can now be summarised as follows. x is any given vertex of a graph G.

```
LongPath(x)
{ construct a long path containing vertex x }
begin
  u := x;   v := x;   P := (x)
  extend P from u and v
  while P is not a hamilton path do begin
    Q := (u)
    FindCrossover(u, 0)
    if a crossover Q was found then begin
      extend P to a cycle C := P ⊕ Q
      find w ∈ C such that w → y ∉ P
      P := C + wy − ww+;   u := w+;   v := y;   go to 1
    end
    construct the components H1, ..., Hk of H := G − V(P)
    for each x ∈ P do
      for each a → x such that a ∉ P do begin
        find i such that a ∈ Hi
        if Ti does not yet exist, then construct Ti and Ai
        for each y ∈ Ai do begin
          y → b ∈ Hi;   Rab ⊆ Hi is an ab-path
          if b > a on P then begin
            if ℓ(Rab) + 2 > ℓ(Pxy) then
              reroute P via Rab and go to 1
            Q := (x)
            FindBypassCrossover(x, 0)
            if a crossover Q was found then
              set P := P ⊕ Q ⊕ Rab and go to 1
          end
        end
      end
    at this point, P cannot be extended further ⇒ Exit
  1: extend P from u and v
  end   { while }
end   { LongPath }
```

The results of using the inner vertices and bypass crossovers is illustrated for the same two graphs as before, the Horton graph and Grinberg graph. The tables following are for crossovers of order ≤ 6 and bypass crossovers of order ≤ 2. The tests were done on the same computer in the same configuration. The use of the inner vertices makes a noticeable difference in performance. Timings were done for a number of other graphs, too. The results are similar. The length of the path found can be improved slightly for some starting vertices x by increasing the maximum orders M

and M'. However this requires a substantial increase in execution time, and does not seem worthwhile.

x	# ticks	$\ell(P)$
1	20	90
2	23	86
3	19	86
4	25	90
5	18	88
18	58	89
19	28	83
20	36	88
21	21	92
40	43	88
41	21	88
42	23	86
43	27	86
44	28	86
56	22	88
57	26	86
58	31	88
59	23	86
66	52	92
67	35	86
68	43	90
69	32	86
88	25	84
89	43	88
90	31	84
91	42	88
92	23	89

Horton graph, 92 vertices

x	# ticks	$\ell(P)$
1	14	45
2	18	45
3	11	45
4	14	45
5	14	45
6	13	45
16	10	43
17	9	45
18	11	45

19	17	45
20	11	45
33	9	43
35	11	43
40	14	45
41	9	41
42	8	44
43	13	45

Grinberg graph, 46 vertices

Complexity

Suppose that G is a r-regular graph. The algorithm *FindCrossover*(w, k) considers all $x \to w$. Then *FindCrossover*$(x^+, k{+}1)$ and *FindCrossover*$(x^-, k{+}1)$ may be called recursively. This builds all trails Q from u with property 1.2. If Q has order k, the length of Q is $2k + 1$. The number of such trails is at most r of order 0, $2r^2$ of order 1, $4r^3$ of order 2, etc., giving at most $\sum_{k=0}^{M}(2r)^k r = O((2r)^{M+1})$ trails in total. If M is fixed as a reasonably small constant, this will be an acceptable polynomial in r.

The *FindCrossover* algorithm is presented as a depth-first search. Given the uw-trail Q, it picks $x \to w$ and tries to extend $Q + wx + xx^+$ to a crossover up to order M before any other vertices adjacent to w are considered. That is, it may overlook a crossover of order 2 in order to find one of order 5. The algorithm can be made to run faster by programming it instead as a breadth-first search. However the program will be longer. We have chosen the depth-first search here to make the presentation simpler.

3. Conclusion

Crossovers are very effective in constructing a long path in a graph. This algorithm is based on theoretical techniques used to prove graphs hamiltonian. See [2,3], for example. The algorithm often finds a hamilton path even when the theoretical conditions needed to prove G hamiltonian don't apply. A very similar algorithm can be constructed to find a long cycle in G, rather than a long path.

Question. Given a path P of length ℓ in a 2-connected graph G with n vertices, where $\ell < n - 1$, and given values M and M', what is the maximum number of edges that G can have if there is no crossover of order $\leq M$, no bypass extending P, and no bypass crossover of order $\leq M'$? Give a construction for such graphs.

References

1. A. Aho, J. Hopcroft, and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Toronto, 1974.
2. Douglas Bauer and Edward Schmeichel, "A simple proof of an extension of Ore's theorem", Ars Combinatoria 24B (1987), 93-99.
3. D. Bauer, A. Morgana, and E.F. Schmeichel, "A short proof of a theorem of Jung", Discrete Math. 79 (1990), 147-152.
4. J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, American Elsevier Publishing, New York, 1976.
5. William Kocay, "Groups & Graphs, a MacIntosh application for graph theory", Journal of Combinatorial Mathematics and Combinatorial Computing 3 (1988), 195-206.