revised, June, 1991

This paper appeared in Discrete Mathematics 101, (1992), pp. 171-188

An Extension of the Multi-Path Algorithm for Finding Hamilton Cycles

William Kocay
Computer Science Department
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2

Abstract

The multi-path algorithm for finding hamilton cycles in a graph G is described in the book by Christofides. It is an intelligent exhaustive search for a hamilton cycle. In this paper we describe how the algorithm can be improved in two ways:

(1) by detecting small separating sets M for which G–M has more than |M| components; and

(2) by detecting bipartitions (X,Y), where |X|<|Y|.

1. Introduction.

Let G be an undirected graph with n vertices V(G), and ε edges E(G). We consider only simple graphs (that is, no loops or multiple edges), so that every edge is uniquely defined by a pair of vertices. If u and v are vertices, then the pair {u,v} is written as uv. It is often convenient to write $u \to v$ if u is adjacent to v, that is uv E(G), and $u \not\to v$ if u is not adjacent to v (cf. Hopcroft and Tarjan [4]). We want to find a hamilton cycle in G, or to determine that G is non-hamiltonian. *We assume throughout that G is a 2-connected graph,* since G cannot be hamiltonian if it is not 2-connected. This problem is NP-complete, so we cannot realistically expect to find a polynomial algorithm. Nevertheless, it is a fundamental question in graph theory to determine whether a graph is hamiltonian, and we would like to do so as efficiently as possible, for as wide a range of graphs as possible. In this article, we describe an algorithm based on the "multi-path" method described in Christophides [3]. It is an exhaustive search of all paths in the graph which may extend to hamilton cycles. In general, it is the non-hamiltonian graphs which are difficult, since we must examine the *entire* search tree, which is equivalent to the problem of finding *all* hamilton cycles of a hamiltonian graph; whereas with hamiltonian graphs, we can stop as soon as we find one cycle. The algorithm described herein is able to recognize, in certain cases, when the current path cannot possibly extend to a hamilton cycle, and this enables us to prune off portions of the search tree, sometimes very large portions. It may be possible to extend the method to include pruning in other instances as well in order to develop a still more efficient algorithm.

2. The Multi-Path Method.

We begin by briefly describing the exhaustive search upon which the algorithm is based. See Christophides [3] for more information. Suppose that we want to determine whether the Petersen graph is hamiltonian. We begin by selecting any vertex u V(G), and any edge uv incident on u. This completes the first step of the search. It is illustrated in Fig. 1, where u=1 and v=2. Write S = { uv }. We are looking for a hamilton cycle C which *must* include the edge uv. In general, any path in G which *must* be included as part of C will be called a *segment* of C. S will be the set of all *segments* of C. We are looking for a hamilton cycle C which must include all the edges of all segments of S. So far S = { uv }.

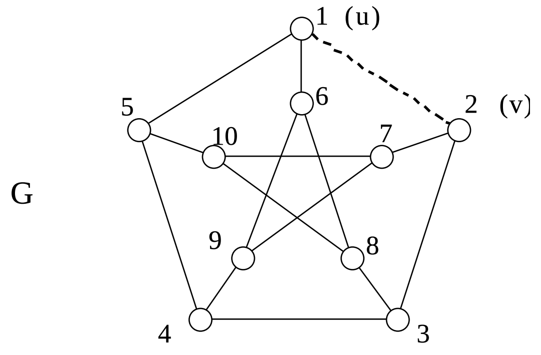


Fig. 1

On the second step of the search, we choose a vertex u, such that u is an endpoint of some segment of S, and we choose some w $\rightarrow$ u. In this example, we have chosen u=2 and w=7. This gives a path $1 \rightarrow 2 \rightarrow 7$ as part of C. Vertex 2 now has two cycle edges, so edge 23 cannot be used in C. Therefore we *delete it from G*. The degree of vertex 3 now decreases to two, so that both remaining edges incident on vertex 3 must be part of C. This gives a path $4 \rightarrow 3 \rightarrow 8$ which must also be part of C, and is illustrated in Fig. 2.

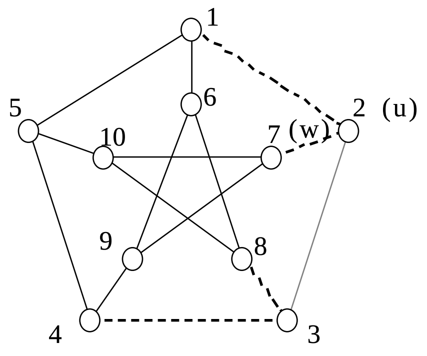


Fig. 2

The paths P_1=(1,2,7) and P_2=(4,3,8) are the new segments of C. S is defined as the set of all segments which are to be a part of C. So now S = { (1,2,7), (4,3,8) }. We can now state the problem to be solved.

HamCycle: Given a graph G and a set S of segments, find a hamilton cycle C containing all segments of S.

We say that (G,S) is hamiltonian if G has a hamilton cycle using all edges of all segments of S. The multipath method is an exhaustive search which can be briefly described as a recursive procedure in Pascal-like pseudo-code.

```
var HamCycle: Boolean  { global, true if a hamilton cycle is discovered }

MultiPath(G: graph;  S: set of segments)
{ search for a hamilton cycle of G containing all segments of S }
{ return as soon as a cycle is discovered }
begin
    choose a vertex u, an endpoint of some segment P   S
    for all w → u do begin
        extend path P to P′ := (P,w)
        compute the new set of segments S′ and the modified graph G′
        { adding the edge uw to P and computing the new  segments  causes  edges
          to be deleted from G.  This may either force a hamilton cycle  C,  force  a
          cycle  of  length  less  than  n,  make  G  disconnected,  or  simply  result  in  a
          new set S′ and graph G′ }
        if  a hamilton cycle C was forced then begin
            HamCycle := true
            return
        end
        if S′ does not contain an invalid configuration  then begin
            { no other cycles were forced, G′ is still connected }
            MultiPath(G′, S′)
            if HamCycle then return
        end
        { at this point, no ham cycle containing S and the edge uw was found,
          now try next vertex w }
    end { for }
    { at this point, all vertices w were tried, but no cycle was found }
    { therefore, the set S does not extend to any hamilton cycle }
end;  { MultiPath }
```

When the new set S′ of segments is computed, it is quite possible for segments of S to be linked together in longer and longer paths, that is the segments often need to be merged.

As with any recursive search procedure, the sequence of recursive calls, and the associated G’s and S’s form a tree. Each node in the tree corresponds to a pair G and S, so that we use (G,S) to represent a node in the tree. The complexity of the algorithm is proportional to the number of nodes in the tree, which is generally exponential in n. If G is hamiltonian, the algorithm stops after finding the first cycle, so that the actual number of nodes searched may be quite small. Indeed, this algorithm is known to work well in practice for most *hamiltonian* graphs (see [3]). It does not perform very well on non-hamiltonian graphs. We describe two improvements which often enable it to recognize that a given set of segments S in G cannot be extended to a hamilton cycle, without actually searching the subtree beneath the node (G,S). This allows one to prune this branch from the tree. In some cases, it can be further deduced that the entire subtree containing (G,S) can also be pruned, and so on.

3. Data Structures for the Segments.

Before describing the improvements mentioned, we first describe the data structures used for the segments. The data structures used to implement any algorithm affect its complexity, so we want the data structure to be as efficient as possible, since it will be used very frequently. In the MultiPath algorithm, we have a number of segments, all of which eventually must be used in a single cycle. The operations which will be performed are:

– determine whether a given vertex u is in a segment, and if so, in which segment;
– find the endpoints of a segment;
– extend a segment P with endpoint u to a segment P′ = (P,w), where $w \rightarrow u$;
– given segments P_1 with endpoint u, and P_2 with endpoint v, add the edge uv, and merge P_1 and P_2 into a single segment. Calculate the new endpoints.

Segments are very much like *merge-find sets* (see [1]), except that each segment has two endpoints, which must be stored. These are called the *left* and *right* endpoints. One way to store them is like merge-find sets – using an array of integer pointers, and a recursive function to follow the pointers to the endpoint, simultaneously compressing the pointer path:

```
PathPt: array[1..n] of integer;
Function Segment(u: integer):  integer;
```

For each vertex x, PathPt[x]=0 if x is not in any segment. If x is in a segment P with left and right endpoints v and u, respectively, then PathPt[x]=–v if x=u, the right endpoint of P; otherwise PathPt[x]>0, being a pointer toward u. See [1] for a fuller description of this data structure. Thus, we can always find u from any x   P by following PathPt[·]:

```
Function Segment(u: integer):  integer;
{ finds which segment x is in, and returns the right endpt }
{ requires global array PathPt }
var z: integer
begin
    if PathPt [x]=0 then Segment := 0 { not in any segment }
    else if PathPt [x]<0 then Segment := x { right endpoint }
    else begin
      z := Segment(PathPt [x])
      PathPt [x] := z { path compression }
      Segment := z
    end
end  { Segment }
```

The right endpoint is given by u := Segment(x) and the left by v := –PathPt[u]. Segments P_1 and P_2 can be merged simply be reassigning the PathPt[·] of their endpoints.

For best efficiency, the merge-find data structure requires that the size of each segment also be stored, so that when merging segments, the smaller is always merged onto the larger.

4. Virtual Edges.

It is convenient to replace each segment P with endpoints u and v by a *virtual* edge uv, that is, a pair which is not currently an edge of G, and which is distinguished from the true edges of G. Intermediate vertices of P are then considered to have been deleted from G. This is because it is the endpoints of segments that are important for the HamCycle problem. This replacement is illustrated for the previous example of the Petersen graph, where the segments (1,2,7) and (4,3,8) are replaced with virtual edges 17 and 48, vertices 2 and 3 having been deleted.

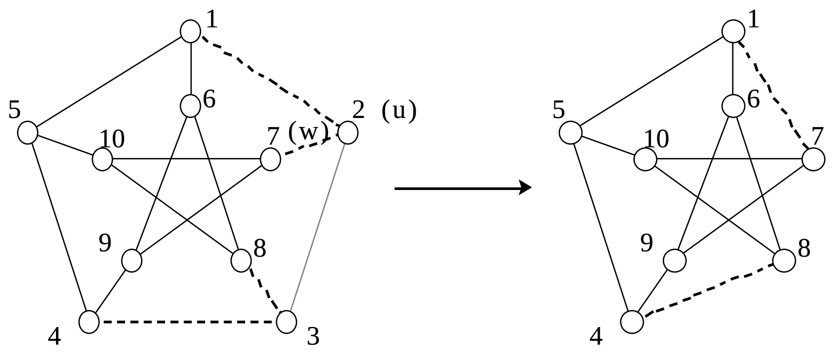


Fig. 3

So, *conceptually* we are replacing each segment with a virtual edge; but in practice, the programming technique considers segments as paths represented by the PathPt[·] array and the function Segment(u). We shall use both terms in describing the algorithm, since they present two different aspects of the problem, and both are useful.

We assume that we have a graph G and a non-empty set of segments S. As described above, every segment P with endpoints u and v can also be considered a virtual edge uv. The multi-path algorithm selects a segment P S, and picks the right endpoint u of P. All vertices w→u are then tried. Edge uw is removed from G and replaced with a virtual edge. This makes u incident on two virtual edges, vu and uw, so that u must be deleted and replaced with a virtual edge vw. This is done by the procedure that computes the new segments. Now when u is about to be deleted, all remaining true edges ux are first deleted. This causes deg(x) to decrease by one. If deg(x) becomes equal to two, then the remaining edges at x, say xy and xz, must both be used in the hamilton cycle C. Consequently, xy and xz must both be deleted and replaced by a single virtual edge yz. There are thus two reasons which may cause a vertex to be deleted (that is, two types of vertices which may be deleted):

(I) During the course of the algorithm, vertex u is found to be incident with two *virtual* edges, say vu and uw. The remaining edges at u must be deleted. Vertex u is deleted, and a new virtual edge vw replaces the previous two.

(II) Upon deleting an edge ux, vertex x is found to have degree two. The remaining edges at x, say yx and xz must be used in C. Delete x and replace edges yx and xz with a new virtual edge yz.

Notice that these two conditions are mutually dependent, for deleting a type I vertex can create type II vertices, and vice versa. Therefore the vertices to be deleted are kept on a queue, and processed in turn. It is possible for an edge ux to be deleted when x is already a known type II vertex on the queue, waiting to be processed, thereby causing deg(x) to decrease to one. In this case, there is no hamilton cycle C with given segments S, so the algorithm immediately returns, indicating that the segments are invalid. Processing these vertices can also force a hamilton cycle. The easiest way to detect this is to store the total number of edges contained in the segments. If this value reaches n, the number of vertices, then a hamilton cycle has been forced. When a vertex is placed on the queue, we intend this to mean that the counter *Q*Size is simultaneously incremented.

```
var  InvalidSegments: Boolean { global, equals true if the segments are invalid }

ComputeSegments(u: integer)
{ vertex u is incident on two virtual edges. Delete it and compute the new
segments.  Vertices to be deleted are collected on a queue }
var Scan$Q$: array of integer  { used as a queue of vertices }
     $Q$Size: integer; { number of vertices on the Scan$Q$ }
begin
    InvalidSegments := true  { initially assume the segments will be invalid }
    { place u on Scan$Q$,u is a type I vertex }
    Scan$Q$[1] := u
    $Q$Size := 1
    k := 1  { counter }
    repeat
        x := Scan$Q$[k]  { select $k^{th}$ vertex from queue }
        { x is to be deleted }
        if x is of type I then begin
            { delete all incident edges at x }
            for all true edges xy do begin
                delete edge xy
                if deg(y)=2 then begin
                  place y on Scan$Q$ and increment  $Q$Size
                  mark y type II
                end
                else if deg(y)<=1 then return  { segments are invalid }
            end
        end
        else begin
            { x is of type II,  say $x \rightarrow y,z$ }
            replace xy and yz with the virtual edge yz
            if a hamilton cycle is forced here then begin
                HamCycle := true
                return
            end
            if a small cycle is forced here then return { segments are invalid }
            if y or z is incident on another virtual edge, then put them on ScanQ
        end
        k := k + 1
    until k>QSize
    InvalidSegments := false { at this point the segments may still extend to C }
end; { ComputeSegments }
```

Let m be the number of type I vertices deleted by ComputeSegments. It is very easy to modify the algorithm so that this number is computed as it proceeds. Notice that $m \geq 1$, since the initial vertex u on the queue is always of type I. It is the value m which is used to improve the multi-path method.

Given a graph G with segments S and edge uw, ComputeSegments(u) will produce a new graph G′ with segments S′. The important fact about this is the following:

4.1 Theorem. G has a hamilton cycle using all the segments of S and the edge uw if and only if (G′, S′) is hamiltonian.

Proof. If G′ has a hamilton cycle C using the segments S′, it is clear that C is also a hamilton cycle of G, because of the equivalence of segments and virtual edges. If G′ has no hamilton cycle using S′, then since the virtual edges forced by ComputeSegments(u) must necessarily belong to a hamilton cycle of G using all the segments of S, neither can G have such a hamilton cycle.

Therefore should (G′, S′) be non-hamiltonian, we know that (G, S) also is (that is, when we require the cycle to use the virtual edges and the edge uw). Notice that the virtual edges of S always form a *matching* in G, namely, each vertex can be incident on *at most one* virtual edge, since whenever a vertex x is found to be incident on two virtual edges, x is deleted, and the two edges are replaced with a single virtual edge.

Deleting vertices tends to disconnect a graph, and so the connectivity of G is important.

5. The Connectivity of G.

In order to have a hamilton cycle, G must obviously be 2-connected, since a hamilton cycle cannot have a cut-vertex. A 2-connected graph may have a pair of vertices {u, v}, such that G – {u, v} is disconnected. Any set of vertices M V(G) such that G – M is disconnected is called a *separating set*(see [4]). It was pointed out by Rubin [5] that a 2-connected graph which is not 3-connected could be decomposed into its 3-connected components. that is, a set of maximal 3-connected subgraphs. See Tutte

[6] for a precise definition of 3-connected components, Hopcroft and Tarjan [4] for information on how to do this decomposition. This gives a collection of 3-connected graphs, polygons, and “bonds”, $G_1, G_2, \ldots, G_t$, in which each separating set {u,v} has been replaced with one or more virtual edges. Then G is hamiltonian if and only if each G_i has a hamilton cycle using all its virtual edges, and a hamilton cycle C in G can be constructed from cycles $C_1, C_2, \ldots, C_t$ in $G_1, G_2, \ldots, G_t$ by fitting the smaller cycles together using their virtual edges. However, there is a twofold disadvantage to using 3-connected components:
`

(1) If G is already 3-connected, it is of no help. Most graphs in which a hamilton cycle is wanted are 3-connected.
(2) After 3-connected components, one would need 4-connected components, then 5-connected components, and so on, if the method is to help substantially. (It is quite easy to construct 3-connected graphs which are non-hamiltonian because of a separating set of size 4 or more — see below).

The graph shown in Fig. 4 is non-hamiltonian because the shaded set M of 3 vertices is a separating set for which G – M has 4 components. The following lemma is from Bondy and Murty [2]. It will be used by the algorithm to reject non-hamiltonian graphs.

5.1 Lemma. Let M be a separating set of G. Then G is non-hamiltonian if the number of components of G – M is greater than $|M|$.
Proof. Let C be a hamilton cycle in G. Then C – M has at most $|M|$ components for any set M V(G). Therefore G – M can have at most $|M|$ components.

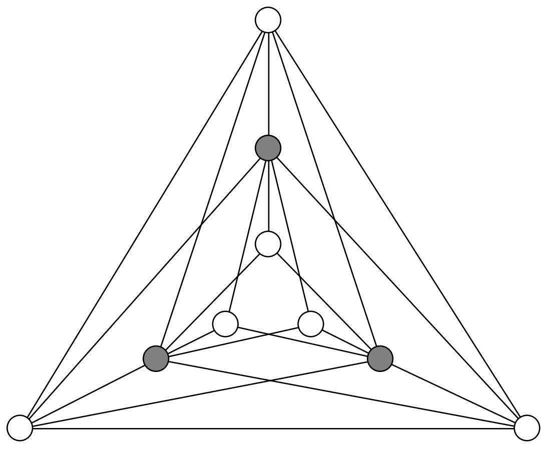


Fig. 4

5.2 Theorem. Let G have virtual edges S, let u be an endpoint of a virtual edge uv S, and let edge uw of G be selected. Let G′ and S′ be constructed by ComputeSegments(u), where M′ is the set of type I vertices that are deleted. If G′ has a separating set U, then M := M′ U is a separating set of G; G–M has at least as many connected components as G′– U has.

Proof. The proof is by induction on m=|M′|. If m=1, then only vertex u was of type I. There may have been one or more type II vertices adjacent to u. This is illustrated in Fig. 5, where edges wu and uv are to be deleted and replaced with the single virtual edge vw; and edges yx and xz are to be deleted and replaced with yz.

Suppose first that m=1; then only vertex u was of type I. There may have been one or more type II vertices adjacent to u. This is illustrated in Fig. 5, where edges wu and uv are to be deleted and replaced with the single virtual edge vw; and edges yx and xz are to be deleted and replaced with yz.

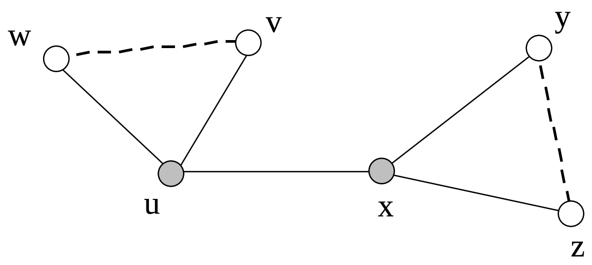


Fig. 5

In general, suppose that m≥1, and let N be the set of type II vertices deleted by the algorithm. Notice that in G – (M′ U), all vertices of N have degree two, and that they subdivide virtual edges of G′.

Suppose that G′ has a separating set U. In G′, y and z are joined by a virtual edge. Consider G – u. Since G is 2-connected, G – u is connected. In it, y and z are joined by the path $y \rightarrow x \rightarrow z$. Since x has degree 2 in G – u, the deletion of the type II vertices like x and their replacement by virtual edges like yz cannot affect the connectivity of G – u. It follows that U is also a separating set of G – u, so that M=U {u} is a separating set of G. Since the edge vw is *added* to G – u in forming G′, the number of components of G – M is at least as big as the number of components of G′ – U. So the result holds when m=1.

If m>1, there is a sequence of type I vertices u_1, u_2, …, u_m that are deleted. Let G″ and S″ be the graph and segments obtained by ComputeSegments(u_1) at that point when u_m reaches the head of the queue. At this point, only u_1, u_2, …, u_{m-1} and possibly some type II vertices have been deleted. The remaining steps computed are equivalent to calling ComputeSegments(u_m) with the graph corresponding to (G″,S″). By the induction hypothesis, we can assume that the result holds for G and G″, namely, that if G″ has a separating set U″, then U″ {u_1, u_2, …, u_{m-1}} is a separating set of G whose deletion gives at least as many components as G″ – U″. There is only 1 type I vertex u_m remaining to be deleted from G″ in forming G′, so if G′ has a separating set U, then U″=U {u_m} is a spearating set of G″, giving at least as many components as G – U. It follows that M=U {u_1, u_2, …, u_m} is a separating set for G such that G – M has at least as many components as G′ – U. By induction the result holds for all m.

The first modification that we make to the MultiPath algorithm is, upon computing G', to determine whether it has a cut-vertex, in other words, a separating set U such that $G' - U$ has $>|U|$ components. This can be done efficiently by a depth-first search (DFS) in $O(\varepsilon)$ steps. If a cut-vertex a is found, the algorithm counts the number k of blocks of G' containing a. This can be done with a simple modification of a basic DFS. For k is just one plus the number of times a is discovered to be a cut-vertex. Having computed k and m, we then know that if $m+1<k$, then G is non-hamiltonian. The remaining portion of the search tree associated with the node (G,S) can be ignored.

6. Bipartite graphs.

Let G be bipartite with bipartition (X,Y). If $|X|\neq|Y|$ then G cannot be hamiltonian. This can be improved upon slightly, as follows.

6.1 Lemma. Let (G', S') be obtained by ComputeSegments(u) from (G, S) with the deletion of m type I vertices. Let G' be bipartite with bipartition (X,Y), where $|X|<|Y|$. If $|X|+m<|Y|$ then G is non-hamiltonian.

Proof: Suppose first that $m=1$ and that $|X|+1<|Y|$. G consists of X-vertices, Y-vertices, and deleted vertices. Suppose that C is a hamilton cycle in G. No two X-vertices or two Y-vertices can be adjacent on C. Therefore between every pair of consecutive Y-vertices on C, there must be either an X-vertex or a deleted vertex. Consider now how the vertices were deleted in constructing G' from G. A type I vertex u was deleted. Consequently the degree of a number of vertices (perhaps none) adjacent to u became equal to 2, so that they were deleted as type II vertices, and replaced with virtual edges. Let x be the last such type II vertex deleted and let yz be the virtual edge replacing it. Since G' is bipartite, y and z are in opposite sides of the bipartition (see Fig. 6). We can restore G' to G by reinserting, in reverse order, the type II vertices deleted. They all become vertices of degree 2 on existing virtual edges of G'. We then join u to each of these vertices, and then to some X or Y-vertices.

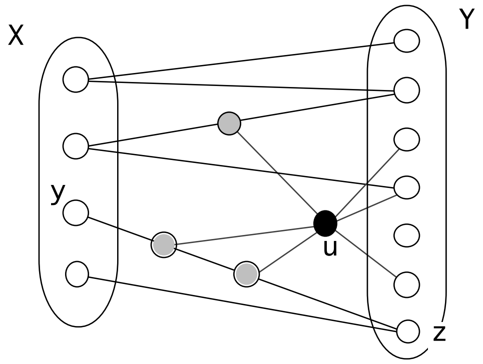


Fig. 6, Type II vertices are lightly shaded

Since $|X|+1<|Y|$, the cycle C must use u to separate two Y-vertices. The type II vertices cannot be used to separate Y-vertices, so that G is in fact non-hamiltonian. Notice that this means that G has no hamilton cycle, whether using its virtual edges S or not.

Now suppose that $m>1$. Suppose that G has hamilton cycle C. Let $G=G_0$, G_1, ...,

$G_m=G'$ be the sequence of graphs constructed by ComputeSegments(u), such that in forming G_{i+1} from G_i, exactly one type I vertex u_i, and its consequent type II vertices are deleted. Using the argument above, G_{m-1} has the form shown in Fig. 6, and is non-hamiltonian. Since G_{m-1} is no longer bipartite, we cannot directly use an induction argument. However, we can continue reinserting, in reverse order, the vertices deleted. Continue doing this until we reach G_{m-2}. The type II vertices are inserted on existing virtual edges of G_{m-1}, until finally u_{m-2} is replaced, adjacent to each type II vertex just added, and perhaps to vertices of X and Y as well. Since $|X|+m<|Y|$, the cycle C must use u_{m-2} to separate two Y-vertices. Therefore the edges joining it to the type II vertices just reinserted are not used on C, and so can all be deleted, thereby making these vertices of degree 2. Since they merely subdivide several edges of the non-hamiltonian graph G_{m-1}, they cannot be used to separate Y-vertices on C. We must therefore use also u_{m-1} to separate Y-vertices. However $|X|+m<|Y|$, so that G_{m-2} is also non-hamiltonian. We can repeat this argument, eventually using all m type I vertices to separate Y-vertices on C, until we finally reach $G=G_0$, which is therefore non-hamiltonian.

G′ is formed from G by deleting certain vertices and adding virtual edges. Form a graph G″ from G′ by removing all virtual edges. G″ may be bipartite even when G′ is not.

6.2 Theorem. Let G, G′, and G″ be as above. Let G″ be bipartite with bipartition (X,Y), where $|X|\neq|Y|$. Suppose the induced subgraphs G′[X] and G′[Y] of G′ contain ε_x and ε_y virtual edges, respectively. If $|X|-\varepsilon_x<|Y|-\varepsilon_y$ then (G′, S′) is non-hamiltonian. If $|X|+m-\varepsilon_x<|Y|-\varepsilon_y$ then G is non-hamiltonian.

Proof: G″ is non-hamiltonian, since $|X|\neq|Y|$. Suppose first that m=1, and let u be the type I vertex deleted from G. G′ can be obtained from G″ by replacing the deleted virtual edges. If C′ were a hamilton cycle of G′ using all virtual edges of S′, then X and Y-vertices must alternate on C′, except for the ε_x virtual edges of G′[X] and the ε_y virtual edges of G′[Y], for which X-vertices are adjacent, and Y-vertices are adjacent, respectively. But since $|X|-\varepsilon_x<|Y|-\varepsilon_y$, there is no such cycle C′, so (G′, S′) is non-hamiltonian.

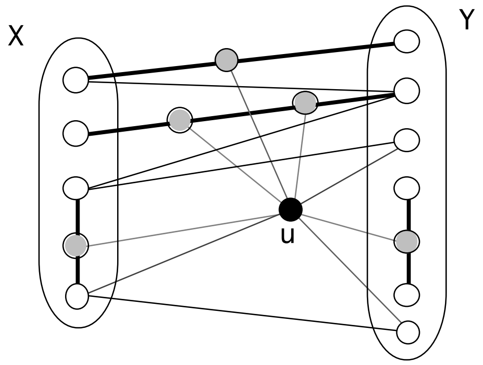


Fig. 7, Type II vertices are lightly shaded, virtual edges are bold.

We then obtain G from G′ by re-inserting the type II vertices on the virtual edges, and re-attaching u. So G consists of X-vertices, Y-vertices, type II vertices, and u. If C were a hamilton cycle of G, then u will be adjacent to 2 vertices on C. If neither of these is a type II vertex, then all the type II vertices must necessarily use their remaining 2 edges. This is almost equivalent to finding a hamilton cycle of G′ using all virtual edges, except that u is now allowed to separate two Y-vertices. But since $|X|+1-\varepsilon_x<|Y|-\varepsilon_y$ there can be no hamilton cycle like this. Otherwise, u will be adjacent on C to at least one type II vertex. The vertices along C alternate X-vertices and Y-vertices, except that several type II vertices will be interspersed amongst them. If z is a type II vertex not adjacent to u on C, then the remaining 2 edges at z must be used in C. If z connects two X-vertices x_1 and x_2, this has the effect of contracting x_1 and x_2 into a single vertex, effectively reducing |X| by 1. Similarly, if z connects two Y-vertices this has the effect of reducing |Y| by 1. This is where the terms $|X|-\varepsilon_x$ and $|Y|-\varepsilon_y$ arise. Since $|X|-\varepsilon_x<|Y|-\varepsilon_y$ there can be a hamilton cycle only if u can be used to prevent vertices like x_1 and x_2 from "contracting into a single vertex", that is, from being adjacent on C. This can be accomplished only if one or both of the vertices adjacent to u on C are type II vertices in X. Let $u\rightarrow z$, where z is a type II vertex connecting x_1 and x_2 in X. Now x_1 and x_2 need no longer be adjacent on C, so that we can use $|X|-\varepsilon_x+1$ when counting adjacent X-vertices on C. However $|X|+1-\varepsilon_x<|Y|-\varepsilon_y$ so this is not in itself sufficient. It follows that the other u-adjacency on C must also be to a type II vertex in X. But then we find that u itself is used to separate two X-vertices on C, which has the effect of making u a Y-vertex; thus we can again subtract one from $|X|$ (or else add one to $|Y|$). We thus find that G is non-hamiltonian if $|X|+m-\varepsilon_x<|Y|-\varepsilon_y$, when m=1.

Now suppose that m>1. As in Theorem 6.1, let $G=G_0, G_1, \ldots, G_m=G'$ be the sequence of graphs constructed by ComputeSegments(u), such that in forming G_{i+1} from G_i, exactly one type I vertex u_i, and its consequent type II vertices are deleted. We cannot use a direct induction proof since the intermediate graphs do not satisfy all the conditions of the theorem. However, using the argument above, we can say that G_{m-1} has the form shown in Fig. 7. We can re-insert the type II vertices deleted, in reverse order, until we come to G_{m-2} by re-attaching the type I vertex u_{m-2} deleted. There can be a hamilton cycle C only if u_{m-2} can be used to prevent X-vertices from being adjacent on C. But the argument above shows that at most one adjacency can be prevented in this way, for each u_i. Continuing up to G, we find that since $|X|+m-\varepsilon_x<|Y|-\varepsilon_y$, it is not possible for G to have a hamilton cycle in G.

Now the DFS which finds the cut-vertices of G can simultaneously determine whether G″ is bipartite, and compute $|X|$, $|Y|$, ε_x, and ε_y. Although performing a DFS for each recursive step in the MultiPath algorithm is expensive if there are many nodes in the search tree, it can compensate for itself by reducing the size of the search tree (see the comments at the end of the paper). Since it is the quantities $|X|-\varepsilon_x$ and $|Y|-\varepsilon_y$ that are important, the DFS calculates them directly as *XSide* and *YSide* by subtracting one whenever an induced virtual edge is discovered. The rough outline of the DFS is shown below. Initially *Bipartite* is set equal to true, and *CutVertex* to false.

```
{ global variables }
var HamCycle, CutVertex, Bipartite: Boolean
    XSide, YSide: Integer  { $|X|-\varepsilon_x$ and $|Y|-\varepsilon_y$ }
    LowPt[u]: array used to find cut-vertices

DFS(u: vertex)
{ depth-first search from u,  upon entry, we know whether u is in X or Y }
begin
    mark u “searched”
    for all non-virtual edges $v \to u$ do
        if v is not searched then begin
            if Bipartite then begin
                if u   X then add v to Y
                else add v to X
            end
            DFS(v)
            if CutVertex then return { no need to complete the DFS }
            use LowPt[u] to check whether u is a cut-vertex
        end
        else begin { v is already searched }
            if Bipartite then if u,v   X then Bipartite := false
                     else if u,v   Y then Bipartite := false
            update LowPt[u]
        end
    end  { for }
    { the virtual edges form a matching, there is at most one incident on u }
    if u is incident on a virtual edge uv then begin
        { be careful not to count these edges twice }
        if u,v   X then XSide := XSide – 1
        else if u,v   Y then YSide := YSide – 1
    end
end  {  DFS  }
```

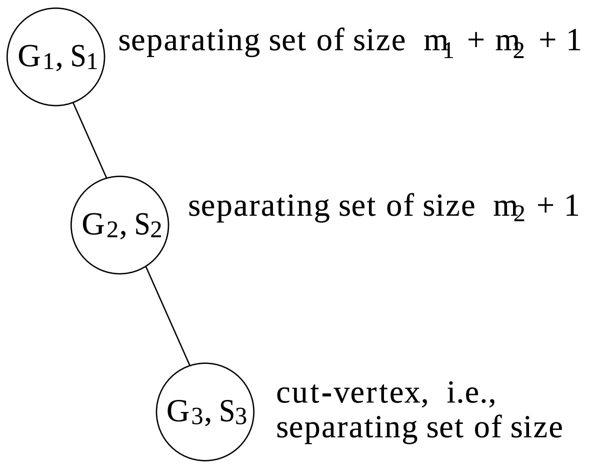
7. The Extended MultiPath Algorithm.

We are now in a position to put these ideas together into a single algorithm. Upon each recursive call, a vertex u of G is chosen incident on a virtual edge uv. A vertex $w \to u$ is chosen to extend the path represented by uv, and ComputeSegments(u) is called, which produces a modified graph G′. The virtual edges of G′ now form a matching. The number m of type I vertices deleted is saved. A DFS is performed in the graph which determines whether G′ has a cut-vertex, and whether G′ less its virtual edges is bipartite. The values k, $|X|-\varepsilon_x$ and $|Y|-\varepsilon_y$ are computed. If these computations do not indicate that G′ is non-hamiltonian, a recursive call to MultiPath is made for the graph G′. Consider a situation in which a sequence of recursive calls is made, proceeding from some point in the search from a graph G_1 from which m_1 type I vertices were deleted to give a graph G_2 from which m_2 type I vertices were deleted to give G_3. from which m_3 type I vertices were deleted, etc., until a graph G_j is reached. The following diagram illustrates this when j=3.

7.1 Theorem. Suppose that G_j is separable, with cut-vertex a, and that $m_{j-1}+1<k$, where k is the number of components of $G_j - a$. Then G_{j-1} is non-hamiltonian. If, further, $m_{j-2}+m_{j-1}+1<k$ then G_{j-2} is also non-hamiltonian, and so on: all G_i are non-hamiltonian for which $m_i+m_{i+1}\ldots+m_{j-1}+1<k$.

Proof. G_j has a cut-vertex, so that G_{j-1} has a separating set M of size $m_{j-1}+1$, such that $G_{j-1} - M$ has at least k components, where $k>m_{j-1}+1$. Therefore G_{j-1} is non-hamiltonian no matter what its set of segments S_{j-1} should be. G_{j-1} was formed from G_{j-2} by deleting type I and II vertices and replacing them with virtual edges. Therefore G_{j-2} has a separating set of size $m_{j-2}+m_{j-1}+1$. If $m_{j-2}+m_{j-1}+1<k$ then G_{j-2} is non-hamiltonian no matter what its set of segments S_{j-2} should be. Similarly, all G_i for which $m_i+m_{i+1}\ldots+m_{j-1}+1<k$ are non-hamiltonian.

Therefore in such cases, the algorithm can return up the search tree from (G_j,S_j) through (G_{j-1},S_{j-1}) and (G_{j-2},S_{j-2}), and so on, until the size of the separating set finally exceeds the value k.



G_1, S_1
separating set of size $m_1 + m_2 + 1$
G_2, S_2
separating set of size $m_2 + 1$
G_3, S_3
cut-vertex, i.e.,
separating set of size

Fig. 6

A similar situation holds if G_j is found to be bipartite:

7.2 Theorem. Suppose that G_j is bipartite with bipartition (X,Y), with x<y, where $x=|X|-\varepsilon_x$ and $y=|Y|-\varepsilon_y$. If $x+m_{j-1}<y$, then G_{j-1} is non-hamiltonian. If, further, $x+m_{j-2}+m_{j-1}<y$, then G_{j-2} is also non-hamiltonian, and so on: all G_i are non-hamiltonian for which $x+m_i+m_{i+1}\ldots+m_{j-1}<y$.

Proof. G_{j-1} is non-hamiltonian no matter what its set of segments S_{j-1} is, by Theorem 6.2. In going from G_{j-1} to G_{j-2}, there are m_{j-2} additional type I vertices available to separate the Y-vertices. Since $x+m_{j-2}+m_{j-1}<y$, G_{j-2} is also non-hamiltonian no matter what its set of segments S_{j-2} is, and so on. Similarly, all G_i for which $x+m_i+m_{i+1}\ldots+m_{j-1}<y$ are non-hamiltonian.

In such cases, the algorithm can return up the search tree from (G_j,S_j) through

(G_{j-1},S_{j-1}) and (G_{j-2},S_{j-2}), and so on, until the value $x+m_{j-1}+m_{j-2}+\ldots$ finally exceeds y. Thus in certain cases, it is possible to prune substantial portions of the search tree.

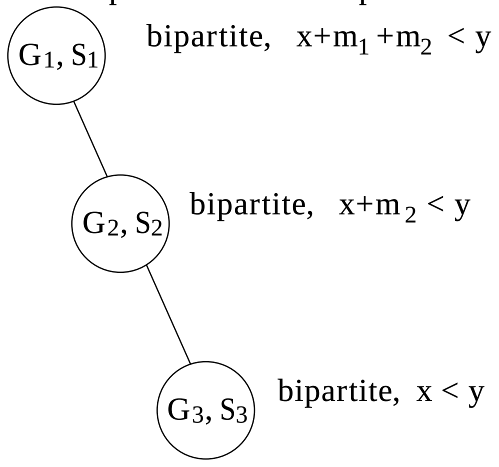


Fig. 7

An outline of the complete algorithm now follows.

```
MultiPath(u: vertex)
{ search for a hamilton cycle of G containing all segments of S }
{ return as soon as a cycle is discovered }
{ vertex u has been already chosen as the right endpoint of some segment }
begin
    for all w → u do begin
        make uw into a virtual edge { u is now a type I vertex }
        ComputeSegments(u) { computes the value m }
        if HamCycle then return  { a hamilton cycle was forced }
        if InvalidSegments then goto 10  { edge uw cannot be used }
        { at this point the segments are OK }
        select a vertex u, the right endpoint of some segment
        CutVertex := false;  Bipartite := true  { initial values for DFS }
        DFS(u)  { computes k, XSide, YSide;  sets flags CutVertex, Bipartite }
        if CutVertex then begin
            MSize := 1  { size of separating set }
            return
        end
        if Bipartite then begin
            if XSide<YSide then return
            { otherwise the bipartite condition is of no help }
            Bipartite := false
        end
        MultiPath(u)  { try to extend the segments into a hamilton cycle }
        if HamCycle then return { ham cycle already found }
        if CutVertex then begin
            MSize := MSize + m
            if MSize<k then return  { see Theorem 7.1 }
            { otherwise MSize is too big – turn CutVertex flag off }
            CutVertex := false
        end
        if Bipartite then begin
            XSide := XSide + m
            if XSide<YSide then return  { see Theorem 7.2 }
            { otherwise XSide is too big – turn Bipartite flag off }
            Bipartite := false
        end
    10: { at this point, no ham cycle containing S and the edge uw was found,
          now try next vertex w }
    end  { for }
end  { MultiPath }
```

If the extended multipath algorithm finds a small separating set, or if it detects a bipartite graph, then there is a large improvement in the running time. However there is currently no guarantee that it will find a separating set or bipartition, even if there is one. This depends on the order in which the virtual edges uv and vertices $w \rightarrow u$ are selected. Currently I have programmed it so as to choose u as the right segment endpoint with the highest possible degree. In this way the largest possible number of edges will be deleted when the path $v \rightarrow u \rightarrow w$ is replaced with a virtual edge.

In summary, if G is hamiltonian, it is likely that the multipath algorithm will find a hamilton cycle fairly quickly (see [3]). There are no good necessary and sufficient conditions known characterising hamiltonicity. This is because the HamCycle problem is NP-complete. Two conditions which force a graph to be non-hamiltonian and which are easy to compute are:

(1) a small separating set M for which G – M has > $|M|$ components;
(2) a bipartition (X,Y) for which $|X| < |Y|$.

These two conditions are really the extremal cases of the same condition, since if G is bipartite, then G–X has |Y| components. Bipartite graphs tend not to have small separating sets, and graphs with small separating sets tend not to be bipartite. The middle ground consists of those graphs which are neither bipartite nor have small separating sets. It would be nice to prove that these graphs tend to be hamiltonian.

Acknowledgement: This work was supported by an operating grant from the Natural Sciences and Engineering Research Council of Canada. I would also like to thank Professor Ron Read for the use of his data-base of papers on graph algorithms.

I would like to thank the referees for their careful job of refereeing the paper, and the many detailed comments for improvements that they made.

References.

1. A. Aho, J. Hopcroft and J. Ullman, *The Design and Analysis of Computer Algorithms*, Addison Wesley, Toronto, 1974.
2. J.A. Bondy and U.S.R. Murty, *Graph Theory with Applications*, American Elsevier, New York, 1976.
3. N. Christophides, *Graph Theory, An Algorithmic Approach*, Academic Press, New York, 1975.
4. J. Hopcroft and R. Tarjan, Dividing a graph into triconnected components, SIAM Journal of Computing 2, 1973, pp. 135-158.
5. F. Rubin, A search procedure for hamilton paths and circuits, JACM 21, 1974, pp. 576-580.
6. W.T. Tutte, *The Connectivity of Graphs*, University of Toronto Press, Toronto, 1966.