March, 1983

HEURISTICS FOR FINDING A MAXIMUM NUMBER OF DISJOINT BOUNDED PATHS

D.Ronen[1] and Y. Perl[1,2]

DCS-TR-126

[1]Department of Mathematics and Computer Science
Bar-Ilan University, Ramat Gan Israel

[2]Department of Computer Science
Rutgers University, New Brunswick, NJ 08903

Department of Computer Science
Rutgers University
New Brunswick, NJ

This work was supported by a grant from the Israel Commission for Basis Research of the Israel Academy of the Science and Humanities and by a grant from the Research Authority, Bar-Ilan University, Ramat-Gan, Israel.

March, 1983

HEURISTICS FOR FINDING A MAXIMUM NUMBER OF DISJOINT BOUNDED PATHS

by

D. Ronen[1] and Y. Perl[1,2]

DCS-TR-126

[1]Department of Mathematics and Computer Science
Bar-Ilan University, Ramat Gan Israel

[2]Department of Computer Science
Rutgers University, New Brunswick, NJ 08903

Department of Computer Science
Rutgers University
New Brunswick, NJ

This work was supported by a grant from the Israel Commission for Basic Research of the Israel Academy of the Science and Humanities and by a grant from the Research Authority, Bar-Ilan University, Ramat-Gan, Israel.

ABSTRACT

We consider the following problem: Given an integer k and a network G with two distinct vertices s and t, find a maximum number of vertex disjoint paths from s to t of length bounded by k.

In a recent work [IPS] it was shown that for length greater than four this problem is NP-Hard. In this paper we present a polynomial heuristic algorithm for the problem for general length. The algorithm is proved to give optimal solution for length less than five. Experiments show very good results for the algorithm.

1. INTRODUCTION

A network is a graph G(V,E) without multiple edges and self-loops. The length of a path in a network is the number of its edges. In the design of communication networks there is a requirement for communication paths of bounded length because of the noise existing in communication lines. Thus it may be required to solve the following problems:

Given an integer k and a network G with two distinct vertices s and t, find a maximum number of vertex disjoint paths from s to t of length bounded by k.

In a recent work [IPS] it was shown that for length greater than four the above problem is NP-Hard [GJ]. For length less than or equal to four, polynomial algorithms (using reductions to known polynomial problems like maximum matching or maximum flow) were given in [IPS].

It is interesting to note that for the related problem of finding a maximum number of disjoint paths of minimum total length, a polynomial algorithm is given by Surballe [S].

In this paper we present a polynomial heuristic algorithm for the above problem for general length. This algorithm incorporates distance considerations into maximum flow techniques. Such a similar incorporation was used by Dinic [D] (see also [ET]) to decrease the complexity of maximum flow techniques. In Dinic's algorithm the length of the augmenting path is bounded. However, in our problem we have to bound the length of each path constructed. Note that more efficient flow techniques, for example Karzanov [K] (see also [E]), Cherkasky [C], Galil [G] and Galil and Naamad [GN], do not help here since we are interested in unit flow in the network.

In the next section a general description of the algorithm is given. It is composed of finding an initial solution and an augmentation algorithm which will be described in section 3. Section 3 also contains the complexity analysis of the algorithm. In section 4 we prove that the algorithm gives an optimal solution for vertex disjoint paths bounded by length 4. As mentioned earlier, for higher length the problem is NP-Hard. This proof intends to give a theoretical basis for the algorithm. Experiments showing very good results for the algorithm are presented in section 5. Some of the proofs are long and are omitted. For the full proofs the reader is referred to [R].

2. GENERAL DESCRIPTION OF THE ALGORITHM

A k-bounded path is a path of length bounded by k. A solution for our problem is a set of disjoint k-bounded paths. An optimal solution is a solution containing such a set of maximum number of paths. The algorithm is combined of two phases.

Phase 1: finding an initial solution.
Phase 2: repeated application of the augmentation algorithm, increasing each time the number of paths in the solution by one until no such improvement is possible.

Finding an initial solution

There are several possible ways to obtain an initial solution. The most trivial of them is to start with an initial empty solution and apply immediately phase 2. However for efficiency we prefer to find an initial solution with the number of paths as large as possible.

A bounding solution is a solution such that no path of bounded length, disjoint to all the paths of the solution, exists in the network. In other words, the network obtained by removing the vertices of the bounding solution contains no k-bounded path. We choose to construct a bounding solution as an initial solution, according to the following algorithm:

Algorithm for constructing a bounding solution of k-bounded paths.

Let G' = G.

while there exists a k-bounded path from s to t in G',

do; let d be the length of a shortest path from s to t in G'.

> Apply a Breadth First Search (BFS) to construct a subgraph G'' of G' containing exactly the edges contained in any path of length d from s to t in G'. Apply a maximum flow technique to find a set P(d) of maximum number of disjoint paths of length d from s to t in G''. Delete the vertices of the paths of P(d) from G'.

end;

The solution P is the union of all sets P(d) for all $d \leq k$. It is easy to show that P is a bounding solution of k-bounded paths. The complexity of finding the set P(d) of paths is $O(|V|^{1/2}|E|)$ [IPS]. Thus the complexity of finding the bounding solution is $O(k|V|^{1/2}|E|)$.

Phase 2 of the algorithm consists of repeated applications of the augmentation algorithm described in the next section.

3. THE AUGMENTATION ALGORITHM

Before presenting the detailed description of the algorithm, we will explain the techniques used by the algorithm. Given a solution with f disjoint k-bounded paths, we will show how to improve the solution (if possible) and obtain f+1 k-bounded paths. A vertex is free if it is not contained in any path of the solution. The algorithm applies a Depth First Search (DFS) starting from the vertex s and searching for the vertex t. The DFS prefers free vertices over non-free vertices, such that the non-free neighbours of a vertex are considered only after considering the free neighbours. In order to bound the path generated by the Depth First Search, every free vertex visited by the DFS is marked by its distance from s along the search path. A vertex may be visited by the DFS only if its present distance from s is less than the previous distance from s. Thus each free vertex is

marked by the shortest distance from s obtained up to this stage of the DFS. Hence, the algorithm works according to the assumption (sometimes wrong) that if in the previous visit the search path failed to reach t it will fail again when the search path at this stage is not shorter than the previous search path.

The DFS is performed as long as the last vertex in the search path has a free neighbour and as long as the search path to this neighbour is k-bounded. Such a neighbour is called a free reachable neighbour. If the search path reaches t (t is always considered free), an improvement in the solution is achieved.

Assume a search path P1 = s, v(1),v(2),. . .,v(l) such that all free neighbours of v(l) were already considered.

Let P2 = s, w(1),w(2),. . .,w(j),. . .,w(m), t be a k-bounded path in the solution. Let w(j) be a neighbour of v(l) such that the distance of w(j) from t plus the distance of v(l) from s is less than k.

The algorithm performs a match between v(l) and w(j) and produces:

1. a new k-bounded path P1 = s, w(1),v(2),. . .,v(l),w(j),w(j+1),. . .,w(m), t and
2. a new search path P2 = s, w(1),. . .,w(j-1).

The DFS will continue using the new search path P2.

As stated, the DFS is performed using free vertices only. If the last vertex in the search path does not have free reachable neighbours, a match will be done with one of the non-free neighbours. It is possible to perform a number of matches in a sequence. A check is made to ensure that a path does not match to itself. Also to prevent endless rematching among a group of paths, a forbidden list is attached to each vertex. The forbidden list of a vertex w(j) contains triplets of the form (v,DIST(v),PATH-ID(v)). Each triplet in the forbidden list forbids a rematch at w(j) from the vertex v, when v is at a distance DIST(v) from s and participates in a path whose identification is PATH-ID(v). Each path is uniquely defined by its exit from s. (see Figure 1.)

A match of v(l) and w(j) adds two triplets to w(j)'s forbidden list:

1. (v(l),DIST(v(l)),PATH-ID(l))
2. (w(j-1),DIST(w(j-1)),PATH-ID(w(j-1))).

The second triplet is necessary to prevent an immediate rematch of the new search path, thus returning to the paths before the match. Once a match is done, it is necessary to backtrack along the new search path from w(j) to w(j-1). This is necessary to keep the paths vertex disjoint.

If, at any intermediate stage, the last vertex in the search path, v(l), has no free reachable nor matchable neighbours, the algorithm backtracks along the search path and the DFS continues from v(l-1). If while backtracking along the search path, the algorithm returns to s, the last match made is declared unsuccessful. It is then necessary to return to the paths

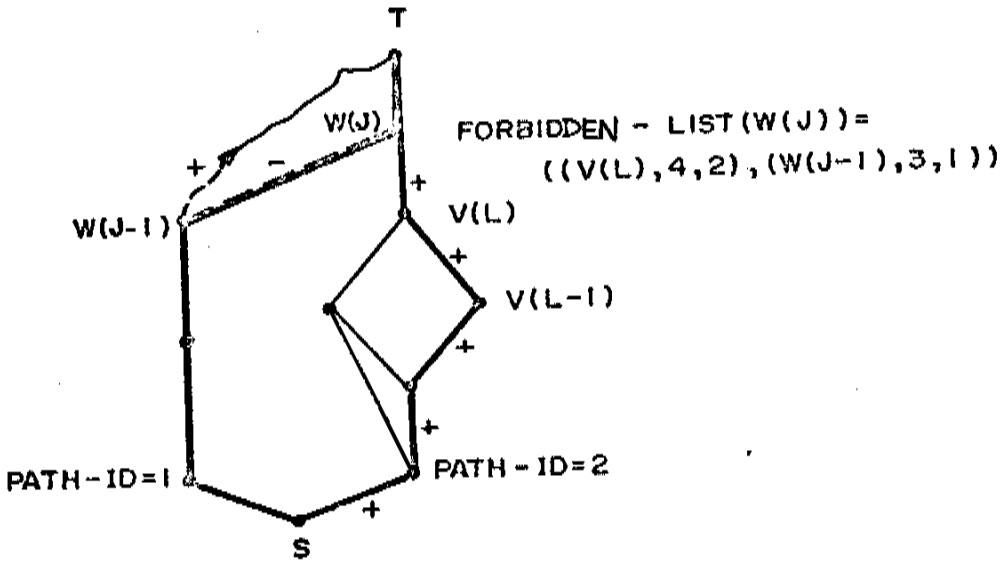
T
W(J)
FORBIDDEN - LIST(W(J))=
((V(L),4,2),(W(J-1),3,1))
+
-
+
V(L)
W(J-1)
+
V(L-1)
+
+
PATH-ID=1
PATH-ID=2
+
S

FIGURE 1.

proceeding this unsuccessful match. In order to be able to back track from a match, it is vital to save the history of each vertex. This is done by attaching to each vertex a stack. When a match is made, the new information assigned to the vertices participating in the match is added to the appropriate stacks. The vertex being matched is also saved in a match stack. The necessary information added to the stacks is:

1–path identification 2–predecessor 3–successor 4–distance from s 5–distance from t 6–match–id number creating this information.

In order to backtrack from a match it is necessary to pop out the last match and all vertex information relating to this match. If, while backtracking along the search path, the algorithm returns to s and there is no match in the match stack, the algorithm simply exits s throughout another neighbour. All the free neighbours of s that were visited and marked by 1 will not participate in any search path. The algorithm ends when all free neighbours of s are marked by a distance of 1, and all non–free neighbours contain s in their forbidden list.

The algorithm.

```
initially assign
  DIST(v)←k+1 for v = 1,2,3,. . .,|V|
  DIST(s)←0
  v←s
  let d = 0 be the depth of the match stack
  let P(d) be the given set of disjoint k–bounded paths.

DFS:
                  while DIST(v)<k
Condition (1):    do; If there is a free neighbour w of v marked by
                          DIST(w) such that DIST(w) > DIST(v)+1
                          then  do; Add w to the search path.
                                    Mark w by DIST(v)+1.
                                    v←w.
                                    If v = t then return. NOTE: a new path is found.
                                end;

                           else go to MATCH. NOTE: condition 1 is not satisfied
                           for all free neighbours of v.

                    end;

                  Go to BACKTRACK ALONG SEARCH PATH.
                  NOTE: the search path is not k–bounded.

MATCH:

Condition (2): If there is a non–free neighbour w of v such that the
                  triplet (v,DIST(v),PATH–ID(v) ) is not in w's forbidden list
                  and such that DIST(v) + the distance of w from t < k
```

then do; Let z be the predecessor of w. Add the
triplet (v,dist(v),path-id(v)) and the triplet
(z,dist(z),path-id(z)) to w's forbidden list.
Store w in the match stack.
Let d be the depth of the match stack.
Perform a match between v and w thus obtaining a
new set, P(d) of k-bounded paths.
v←z.
Go to DFS.

end;

BACKTRACK ALONG SEARCH PATH:

NOTE: at this stage v has no neighbours satisfying condition (1) or (2).
x←predecessor of v.
Delete v from the search path.
If x = s then do; v←x.
Go to DFS.
end;

BACKTRACK FROM LAST MATCH:

NOTE: at this stage the algorithm has backtracked to s.

If d > 0 then do; w←the vertex at the top of the match stack.
v ← predecessor of w.
Let P(d)←P(d-1).
NOTE: return to the paths before the last match.
Pop out w from match stack.
Go to DFS.

end;

TERMINATE:

If s has a neighbour satisfying condition (1) or (2) then go to DFS, else terminate the algorithm with no improvement in the solution.

Time complexity of the algorithm.

We calculate separately the time complexity of the DFS part and that of the matching part of the algorithm.

At the beginning of the DFS, all free vertices are marked by k+1 as the bound for the distance from s. A visit to a free vertex is possible only if the present distance from s is less than the previous distance. It is therefore possible to scan the edges at most k times. Thus, the DFS will need at most $O(k|E|)$ time to find one more path.

In finding one more path, it is possible to perform a match at a vertex if the forbidden list allows it. The number of matches performed at a vertex is equal to half the number

of the elements in the forbidden list. Each element is of the form (v,DIST(v),PATH-ID(v)). The number of neighbours of a vertex is DEGREE(v). Each such neighbour can be marked at most k times. Thus at most k different values of DIST(v) are possible. The number of disjoint paths possible in the network is certainly not more than $|V|/k$ since the paths a k-bounded. Thus PATH-ID may have at most $|V|/k$ different values. The number of matches possible is therefore $SUM(DEGREE(v)k|V|/k)=|V|SUM(DEGREE(v))=|V||E|$. Following the match, the algorithm assigns new values to the vertices participating in the new bounded path. There are at most k vertices in the bounded path, thus the matches performed require $O(k|V||E|)$ time.

The time complexity for finding one more path is the sum of the time complexity for the DFS part and that of the matching part. It is therefore $O(k|E| + k|V||E|=O(k|V||E|)$.

In order to get the complexity of the algorithm we have to multiply the above complexity by the number of applications of the augmentation algorithm, which is $|V|/k$ since this is the maximum number of possible disjoint k-bounded path in the network. Hence, the complexity of the algorithm is bounded by $O(|V|^2|E|)$.

However, in this analysis we did not use the fact that we first construct a bounding solution. If the maximum number of disjoint k-bounded paths in the network is q and p disjoint k-bounded paths are found in the bounding solution, then the complexity of the algorithm is bounded by $O((q-p)k|V||E|)$. Note that the complexity of finding a bounding solution is $O(k|V|^{1/2}|E|)$.

The space complexity of the algorithm.

Because of the many matches and backtracks performed by the algorithm while searching for a new path, it is necessary to save the history of every vertex. A stack is assigned to each vertex. Whenever a change occurs to a vertex, the old information is pushed down into the stack and the new information is added at the top of the stack. When backtracking, this information is poped out of the stack and we are left with the previous information assigned to the vertex. Clearly the space requirement is linear with the number of triplets in the forbidden lists and the depth of the stacks of all vertices.

As in the case of the time complexity, we shall distinguish between the space complexity of the DFS part and that of the matching part.

Every visit to a free vertex brings to an allocation of one storage cell necessary to add the vertex to the search path. The number of possible visits to a vertex is k, thus the space complexity for scanning free vertices is $O(k|V|)$.

Each match brings to an allocation of k storage cells necessary for defining the new k-bounded path. The number of matches is at most $O(|V||E|)$, thus the space complexity for performing matches is $O(k|V||E|)$. The total space complexity for finding one more bounded path is $O(k|V| + k|V||E|)=O(k|V||E|)$.

Once a path is found all storage structures (stacks, lists etc.) are initialized and thus this is the space complexity for finding any number of paths.

4. THEORETICAL BASIS FOR THE ALGORITHM

Theorem 1:

All the paths found during the operation of the algorithm are vertex disjoint and k-bounded. The proof is technical and is omitted (see [R]).

As is expected from the NP-Hardness proof in [IPS] our algorithm does not necessarily provide optimal solution even for length five. For example see figure 2.
Because the algorithm is quite complicated, it is difficult to prove a guaranty for the approximation obtained, or any result about the probability of obtaining the optimum.

However we shall prove in Theorem 2 that for length 4 the algorithm provides an optimal solution. This proof is intended to give some theoretical insight of the algorithm, though for length greater than four it is not optimal. For length three the initial bounding solution is optimal (see [IPS]).

We will show that for a given network with f vertex disjoint 4-bounded paths, the algorithm will find f+1 such paths if f is not the maximum number of vertex disjoint 4-bounded paths in the network. The symmetric difference of two sets A and B is $(A-B)\cup(B-A)=A\cup B-A\cap B$. For any set of f+1 such paths it is possible to obtain a symmetric difference with the set of the f given paths. The symmetric difference produces a signed alternating path from s to t consisting of edges in forward and backward direction [S] (see also [F]).

Let us choose a set of f+1 paths which produces a shortest alternating path. Let this shortest alternating path be AP=x,v(1),v(2),v(3),. . .,v(n),t.

There may be repetition of vertices in the alternating path. To every appearance of a vertex in the current induced bounded path, we attach its distance from s and t along the path induced by the alternating path. The distance of a vertex v from s will be presented by SD(v) and from t by TD(v).

In the general case, it is possible that a vertex which appears more than once in the alternating path will be marked by different distances. For example, Figure 2 shows the alternating path AP=s,1,2,3,4,8,6,5,s,11,12,7,8,9,10,t. In this path, vertex 8 appears twice as v(5) and as v(12), yet SD(v(5))=3 and SD(v(12))=2 since the corresponding induced paths through vertex 8 are s,5,6,8,9,10,t and s,7,8,9,10,t, respectively.

For the problem of length four, the general structure of the network is as described in Figure 3.

SN is the set containing all the neighbours of s. Similarly TN is the set containing all the

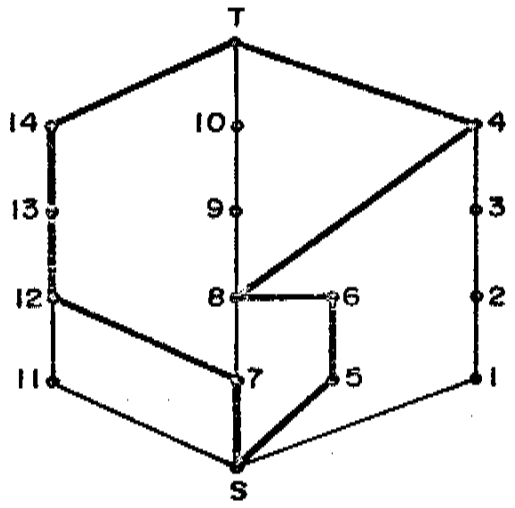

FIGURE 2

neighbours of t. The set A is the intersection of SN and TN. The set B contains all the vertices which are neither in SN nor in TN. We present a number of lemmas which enable us to prove the validity of the algorithm for length four.

Lemma 1:

For length 4 it is sufficient to consider networks where the intersection of SN and TN is empty and no edge connects two vertices of SN or two vertices of TN.

Proof: It follows from the following two facts: Every vertex v in the intersection of SN and TN appears in a path s,v,t. Every path containing an edge between two vertices of SN(or TN) can be shortened without using new vertices.■

When given a network, let us omit the set A of vertices and all edges connecting two vertices of SN or two vertices of TN. Any optimal solution will contain the $|A|$ paths of length 2 which are omitted.

In the following we will consider only networks satisfying lemma 1. Their structure is described in Figure 4.
For these networks we prove the following properties of the algorithm which are necessary for the proof of Theorem 2.

Property 1:

All vertices which are marked by the algorithm are marked by:

SD(v)=1 and TD(v)=2 or TD(v)=3 if v is in SN,
TD(v)=1 and SD(v)=2 or SD(v)=3 if v is in TN,
SD(v)=2 and TD(v)=2 if v is in B.

Let us assume that there is a vertex v in SN such that SD(v)>1. By Lemma 1, in SN there are no two vertices connected to each other, thus SD(v)≠2. SN does not contain t therefore SD(v)≠4, and thus we are left with SD(v)=3. On the other hand by Lemma 1, vertices in SN are not connected to t, therefore TD(v)>1. If so, SD(v)+TD(v)>4 contradicting Theorem 1 which states that the algorithm finds disjoint 4-bounded paths.

In a similar way we can prove the rest of the facts of property 1.■

Property 2:

All vertices which appear in an alternating path more than once, are in B.

Property 3:

In the shortest alternating path there is no j, $0<j<n+1$, such that v(j)=s.

Property 4:

The vertex v(1) is the only free vertex from SN which appears in an alternating path. The

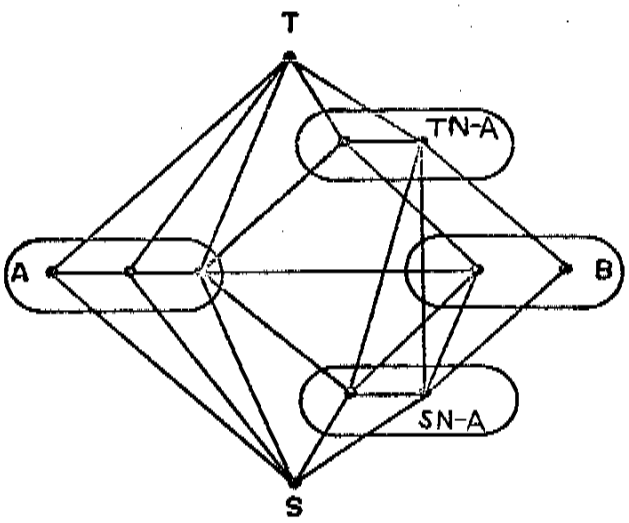

FIGURE 3.

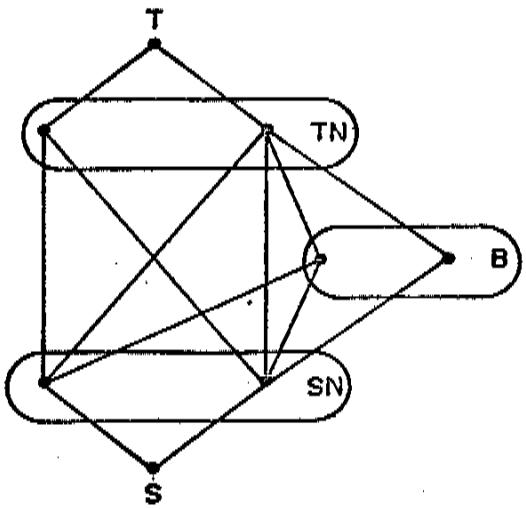

FIGURE 4.

vertex v(n) is the only free vertex from TN which appears in an alternating path.

The proofs of these three properties are very long and complicated and are omitted. The interested reader is referred to [R]. Theorem 2 uses directly only property 4 whose proof uses the previous properties.

Theorem 2:

The augmentation algorithm finds an alternating path for 4-bounded paths.

Proof:

Let AP=s,v(1),v(2),. . .,v(n),t be a shortest alternating path. Let SP=s,w(1),w(2),. . .,w(l) be a search path of the algorithm. Let D(i) represent the distance of w(i) from s along the search path.

The search path of the algorithm exits s through one of it's neighbours. If it reaches t then the algorithm finds an alternating path and the solution is improved. However, if the search path fails to reach t, it will exit s through another neighbour. If the search path continues to fail to reach t, it will eventually exit s through it's neighbour v(1). The vertex v(1) is a free vertex which participates in the shortest alternating path, AP, such that $D(1)+TD(v(1))\leq 4$. Thus we can assume that the search path of the algorithm intersects AP at a vertex $x=v(j)=w(i)$, for some i and j, $0<i\leq l$, $0<j\leq n$, and $D(i)+TD(v(j))\leq 4$. We shall prove by induction that if the intersection is at vertex v(j) or any other vertex following v(j) in the alternating path then the algorithm reaches t. Let i be the first index such that w(i)=x. Similarly, let j be the first vertex such that v(j)=x. The proof is by induction on j (along the alternating path starting from n).

If j=n we get from Property 4 that v(j)=v(n) is the only free neighbour of t participating in the alternating path. During the Depth First Search, the algorithm checks if the last vertex in the search path is a neighbour of t, thus if $D(i)+TD(v(n))\leq 4$ then the algorithm reaches t with the alternating path s,w(1),w(2),. . .,w(i-1),v(n),t.

Assume that if the intersection of the two paths is at vertex v(j) or any other vertex following v(j) in AP and the condition $D(i)+TD(v(j))\leq 4$ is satisfied, than the algorithm reaches t. Assume that the search path of the algorithm intersects AP at v(j-1) at a distance D(h) such that $D(h)+TD(v(j-1))\leq 4$.

There are two possibilities to add v(j-1) to the search path:

1: by making a match with it in case v(j) participates in some bounded path.
2: by the Depth First Search in case v(j) is free.

In case 1, it is necessary to backtrack after the match, in order to keep the induced bounded paths vertex disjoint. The algorithm backtracks to v(j). At v(j-1) we had $D(h)+TD(j-1)\leq 4$, thus at v(j) we get $D(i)=D(h)-1$ and $TD(v(j))=TD(v(j-1))+1$ thus $D(i)+TD(v(j))\leq 4$. By the induction assumption we reach t.

In case 2, the algorithm adds v(j−1) to the search path during the DFS phase. The algorithm continues the DFS using one of the free neighbours of v(j−1). If after l steps we reach some vertex v(m), m>j, and D(l)+TD(v(m))≤4 then by the induction assumption the algorithm finds an alternating path from s to t. If D(l)+TD(v(m))>4 then the algorithm backtracks from v(m), and v(m) is left free. The DFS continues with some other neighbour of v(j−1) and reach v(j). At v(j−1) we had D(h)+TD(v(j−1))≤4. Therefore at v(j) we get D(i)=D(h)+1 and TD(v(j)=TD(v(j−1))−1. Thus we get D(i)+TD(v(j))≤4. Let us show that the algorithm is able to reach v(j). There are two possibilities to add v(j) to the search path:

1: by DFS in case v(j) is free.
2: by making a match with it in case v(j) participates in some bounded path.

In case 1, v(j) is a free vertex. The algorithm checks that the present distance from s along the search path is less than the previous distance. The vertex v(j) is either marked by k+1 (has never been visited), or it has been visited, but the condition D(i)+TD(v(j))≤4 did not hold, or else by the induction assumption the search path of the algorithm would have reached t.

In case 2, the vertex v(j) participates in some path. Since v(j)'s forbidden list does not contain v(j−1) we are able to perform a match between w(i) and v(j). Assume otherwise, that v(j)'s forbidden list contains a triplet of the form (v(j−1)=w(i−1),D(i−1), PATH−ID). The 4−bounded path induced by the match which creats this triplet is of the form s,w(1),w(2),. . .,v(j−1),v(j),. . .,t. However, since w(1) uniquely identifies PATH−ID and the paths are 4−bounded, the path must be of the form s,w(1),v(j−1),v(j),t. By the induction assumption the algorithm should have reached t following this match, and no rematch is necessary. Hence v(j)'s forbidden list does not contain v(j−1) and a match is possible at v(j).

In both cases, once v(j) is reached and D(i)+TD(v(j))≤4 holds, by the induction assumption we reach t.■.

5. EXPERIMENTAL RESULTS

In the experiment, we test networks with an initial bounding solution which is probably not optimal. This possibility exists when the initial solution is of less paths than the maximum number of disjoining paths (not necessarily k−bounded) in the network. The maximum number of disjoint paths in the network (without length constraint) is obtained by Dinic's algorithm [D].

In our experiment we choose random graphs consisting of 50 vertices and a varying number of edges. The edges are undirected (or more precisely, each undirected edge consists of two opposite directed edges). For each length and graph density a number of graphs are generated. From the random graphs generated, we choose only those for which the number of paths in the initial solution is less than the maximum number of paths without length constraints.

In these cases the augmentation algorithm is applied. We count the number of times an improvement is made on the initial solution. We also count the number of times when the improved solution is equal to the maximum number of disjoint paths obtained by Dinic's algorithm. In these last cases we are certain that the improved solution is optimal. In the other cases, it is difficult to say whether the solution is optimal since the optimum is not known. In order to get a better evaluation of the augmentation algorithm, we apply it a number of times in different directions. We first apply the algorithm from s to t on the network containing the initial solution. We apply the algorithm a second time from t to s on the network containing the improved solution from the previous application. We then apply the algorithm a third time from s to t starting with an initial empty solution and a forth time from t to s on the solution obtained in the third time. Similarly, we exchange the role of s and t and apply the algorithm a fifth and sixth time from t to s and from s to t starting again with an initial empty solution. It is assumed that if the solution is not optimal, at least one such application will give a solution with more paths indicating a non-optimal solution. The following table presents the experimental results.

Of the 9012 cases generated, the algorithm for the initial bounding solution is optimal in 8690 cases (96%). In the 322 other case the augmentation algorithm gives sure optimal solutions in 126 cases bringing the total sure optimal solutions to 98%. We find only two cases for which the augmentation algorithm certainly does not give an optimal solution. This occurs once for k=6 and once for k=7 when applying the augmentation algorithm back and forth from s to t and from t to s. Each application gives a different result. In view of all the tests that were performed we assume that the augmentation algorithm gives optimal solutions in all the other cases. There is no way to check this assumption in reasonable time.

From the table we see that as the length increases, the initial solution is optimal in more cases. However in the other cases, the augmentation algorithm gives also optimal solutions. In other words, when the length constraint is not severe, it is easy to find an initial optimal solution because of the tendency of the algorithm for the initial solution to find shortest paths. This phenomenon also occurs when the graph density increases. It becomes more and more difficult to find an initial solution which is not optimal for large length and high density. In view of these results, we conclude that the algorithm is very good especially if it is applied back and forth in both directions.

|V|=50 |E|=	200	400	600	800	1000	1500	2000	total
k=3								
graphs generated	40	32	64	176	1088	961	1095	3456
graphs tested	39	25	25	25	23	6	6	149
improved solutions	0	0	0	0	0	0	0	0
sure optimal solutions	39	25	25	25	23	6	6	149
k=4								
graphs generated	70	193	567	315				1145
graphs tested	25	25	10	5				65
improved solutions	2	23	10	4				39
sure optimal solutions	1	23	10	4				38
k=5								
graphs generated	186	958						1144
graphs tested	25	10						35
improved solutions	14	10						24
sure optimal solutions	14	10						24
k=6								
graphs generated	347	17						364
graphs tested	25	2						27
improved solutions	20	2						22
sure optimal solutions	20	2						22
non-optimal solutions	1							1
k=7								
graphs generated	481							481
graphs tested	25							25
improved solutions	21							21
sure optimal solutions	21							21
non-optimal solutions	1							1
k=8								
graphs generated	2422							2422
graphs tested	21							21
improved solutions	21							21
sure optimal solutions	21							21
total								
graphs generated	3546	1200	631	491	1088	961	1095	9012
graphs tested	160	62	35	30	23	6	6	322
improved solutions	78	35	10	4	0	0	0	127
sure optimal solutions	77	35	10	4	0	0	0	126
non-optimal solutions	2							2

Fifth line for k=6, k=7 and total shows non-optimal solutions.

REFERENCES

[C] : B.V. Cherkasky, "Algorithm for Construction of Maximal Flow in Networks with Complexity of $O(V^2E^{1/2})$ Operations", Math. Methods of Solution of Economical Problems (1977) 117-125 (In Russian).

[D] : E.A. Dinic, "Algorithm for Solution of a Problem of Maximum Flow in a Network with Power Estimations", Soviet Math. Dokl., 11(1970) 1277-1280.

[E] : S. Even, " The Max Flow Algorithm of Dinic and Karzanov: An exposition", Information Technology, J. Moneta(ed.), North-Holland Publishing Company (1978) 233-237.

[ET] : S. Even and R.E. Tarjan, "Network Flow and Testing Graph Connectivity", SIAM J. Comput. 4(1975) 507-518.

[F] : I.T. Frisch, "An Algorithm for Vertex-Pair Connectivity", Inter. J. Control 6(1967) 579-593.

[G] : Z. Galil, "A new Algorithm for the Maximal Flow Problem", Proc. 19th Symp. on Foundations of Computer Science (1978) 231-245.

[GJ] : M.R. Garry and D.S. Johnson, "Computers and Intractability; A Guide to the Theory of NP-Completeness", Freeman, San Francisco, 1979.

[GN] : Z. Galil and A. Naamad, "Network Flow and Generalized Path Compression", Proc. 11th Annual ACM Symp. on Theory of Computing (1979) 13-26.

[IPS] : A. Itai, Y. Perl and Y. Shiloah, "The Complexity of Finding Maximum Disjoint Paths with Length Constraints", Networks, 12(1982) 277-286.

[K] : A.V. Karzanov, "Determining the Maximal Flow in a Network by the Method of Preflows", Soviet Math. Dokl., 15(1974) 434-437.

[R] : D. Ronen, "A Heuristic Algorithm for Finding a Maximum Number of Disjoint Bounded Paths in a Network", M.Sc. Thesis, Department of Mathematics and Computer Science, Bar-Ilan Univ.

[S] : J.W. Surballe, "Disjoint Paths in a Network", Networks, 4(1974) 125-145.