On the Least Deviant Path in a Graph

William F. Klostermeyer
Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506–6330

ABSTRACT. A *least deviant path* between two vertices in a weighted graph is defined as a path that minimizes the difference between the largest and smallest edge weights on the path. Algorithms are presented to determine the least deviant path. The fastest algorithm runs in $O(|E|^{1.793})$, in the worst case. A type of two-dimensional binary search is used to achieve this running time.

1 Introduction

We show how to efficiently find a path between two vertices, u and v, that minimizes the difference between the largest and smallest edge weights on the path. Such a path is called the *least deviant path* between u and v. For example, a motorist may wish to choose a route to a destination so that his or her speed stays within as narrow a range as possible, where edge weights in the graph would represent speed limits on the highways. As another example, suppose the weights represent temperatures of pipes in a chemical pipeline network. It may be desirable to route chemicals from the source to the destination using a least deviant path so that the chemicals experience minimal variations in temperature. Four algorithms are given, each refining the previous one. The most efficient algorithm runs in $O(|E|m^{0.793})$ time, where m is the number of distinct edge weights in the graph, using a type of two-dimensional binary search. A different type of multi-dimensional binary search was discussed in [1].

Typical algorithms for shortest-path type problems, such as shortest path between two vertices, all-pairs shortest path, or bottleneck shortest path use variants of breadth-first search, depth-first search, Dijkstra's greedy algorithm or dynamic programming [2]. These techniques are difficult to apply to the problem of finding deviant paths, as locally optimal paths do

JCMCC 24 (1997), pp. 147-154

not necessarily lead to globally optimal paths, as shown in Figure 1. In this example, the least deviant path from u to intermediate vertex w contains edges of weight five and four, whereas the least deviant path from w to v contains edges of weight ten and eleven. However, the least deviant path from u to v has edges of weight five, nine and ten.

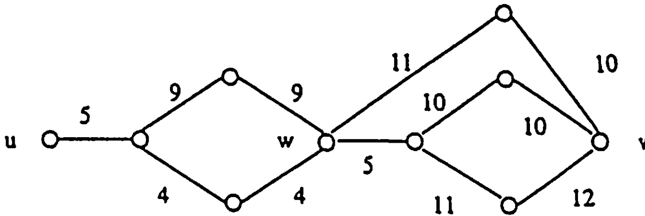


Figure 1. Example of Least Deviant Path

2 The Basic Approach to Least Deviant Path

Input a weighted graph $G = (V, E)$, source vertex u and destination vertex v. G may be directed or undirected; we shall assume the latter in this note. Assume without loss of generality that all edge weights are positive. Define the *deviation* of a path be the difference between the maximum edge weight on the path and the minimum edge weight on the path. The idea is to search for paths between u and v such that each edge weight on the path is in a specified range $[a..b]$. By enumerating all possible combinations of a and b, we will discover a least deviant path. Denote by $\max(P)$ the maximum edge weight on a path P and by $\min(P)$ the minimum edge weight on path P. The algorithm is as follows:

```
1. Sort edges weights in ascending order
2. best := ∞
3. for i := 1 to max_edge_weight in G do
4.        for j := 1 to i do
                test if there exists a path, P, from u to v such that all
                     edge weights in P are in range [j..i];
                if max(P) − min(P) < best then best := max(P) − min(P)
          end;
   end;
```

Unfortunately, the number of iterations of the "for" loops at lines 3 and 4 may be exponential in the length of the algorithm's input. This is the case if some edge weights are very large, since the number of bits needed to represent a number is logarithmic in terms of the number's magnitude.

3 A First Polynomial Time Algorithm for Least Deviant Path

A simple modification of the algorithm in Section 2 results in polynomial time solution. In Section 4 and Section 5 we shall further improve upon the algorithm given below. The key idea is to count the number of distinct edge weights in G, call this m, and map each distinct weight to an integer in the range $1..m$. Now only m^2 combinations of edge weights need to be enumerated, regardless of the magnitude of the weights.

```
1. Sort edges weights in ascending order
2. let there be m ≤ |E| distinct edge weights
3. map weights to [1..m], i.e. the smallest weight is mapped to 1, next
       smallest to 2, and so on
4. Construct G' = (V, E') where E' is same as E except edge weights
       are in [1..m], according to mapping from step 3
5. best := ∞
6. for i := 1 to m do
       for j := 1 to i do
           a) test if there exists a path P' in G' from u to v such
               that all weights in P' are in range [j..i]
           b) if such a path exists then
               find corresponding path P in G
               if max(P) − min(P) < best
               then best := max(P) − min(P)
       end;
   end;
```

We detail step 6a). To find a path whose edge weights are in the range $[j..1]$ can be done using a modified breadth-first search [2] as follows.

```
mark all vertices "white"
Q := {u};                         — queue of vertices
mark u "gray"
while Q not empty do
      w := head(Q);               — remove first vertex in Q
      for each x adjacent to w
            if color(x) = "white" then
                  if weight((w, x)) in [j..1] then
                        color(x) := "gray"
                        add x to Q;        — add x to Q since
      end for;                             — there is a good path
                                           — from u to x
end while;
if v is colored "gray" then return("Yes") else return("No");
```

Suppose a path whose edge weights all are in the range $[j..1]$ exists from u to v. Let one such path be $(u, w_1, w_2, \ldots, w_k, v)$. An easy induction on k suffices to show that v will be colored "gray". On the other hand, if no such path exists, v can never be added to the queue, since each possible path from u to v contains an edge whose weight is not in $[j..1]$. In this case, each path has an edge that fails to satisfy the "if weight((w, x)) in $[j..1]$" statement. It is a simple matter to extend the path testing procedure to return a path rather than just "yes" or "no".

An example of the algorithm is shown in Figure 2. In Figure 2a), the least deviant path from u to v is the path labeled with edge weights (50,51,52,50,51,53). Note that this path will not be found in G' (shown in Figure 2b) by the algorithm until the iteration when $i = 5$ and $j = 4$, at which point it will be recognized at the least deviant path (found so far) at step 6b). Of course, subsequent iterations on this input graph will not find a less deviant path.

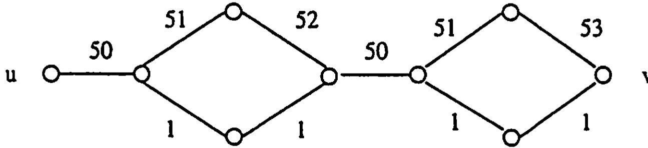


Figure 2a.

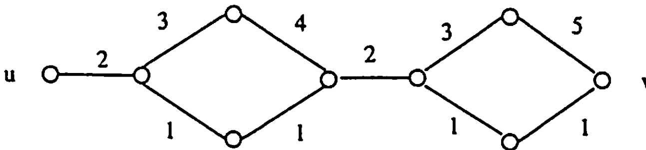


Figure 2b.

Figure 2. Computing the Least Deviant Path

The running time of the algorithm is: $O(|E| \log |E|)$ to sort; $O(|E|)$ for steps 2–3. We henceforth assume that $|E| \geq |V|$, else the problem is not interesting (e.g. trees or disconnected graphs). The remaining analysis is: $O(|E|)$ for steps 4 and 5; and $O(|E|^3)$ for step 6 since step 6a) can be done in $O(|E|)$ time and 6b) can be done in $O(|V|)$ time. So the total running time is $O(|E|^3)$ in the worst case. If the number of distinct edge weights is $O(|E|)$ the actual running time is more precisely stated as $O(m^2|E|)$.

We prove the correctness of the algorithm by way of contradiction:

Proof of correctness: Suppose some other path Z has less deviation than P. Let Z have minimum weight min and maximum weight max. So $\max - \min < \max(P) - \min(P)$. Let max be the xth largest distinct edge weight and min the kth largest edge weight, $x \geq k \geq 0$. At step 6 of our algorithm with $i = x$ and $j = k$, we would have discovered a path with deviation no more than that of Z. Hence the.proof. □

4 A Faster Algorithm

A particularly slow step in the algorithm of Section 3 is step 6 – the nested "for" loops that lead to an $O(m^2)$ factor in the running time. Suppose we discover there is a path having all edge weights in the range [3..7]. Then there is no need to perform tests on the ranges [1..7] or [2..7]. Likewise, if we discover that there is no path having all edge weights in the range [3..7], there is no need to perform tests on the ranges [4..7], [5..7], [6..7], or [7..7]. We may therefore use a binary search to more efficiently search through the j indices. This is done as follows:

```
for i := 1 to m do
  /* now do a binary search in the range [1..i] */
     lo : = 1;
     hi := i;
     while lo ≤ hi do
          mid := (lo + hi)/2;
          a) test if there exists a path P' in G' from u to v such
                 that all weights in P' are in range [mid..i]
          b) if such a path exists
                 i) lo := mid + 1
                 ii) find corresponding path P in G;
                    if max(P) − min(P) < best
                    then best := max(P) − min(P)
          else
                 hi := mid − 1;
     end while;
  end for;
```

This improves the running time to $O(m|E|\log m)$, which is $O(|E|^2 \log |E|)$ in the worst case.

5 The Final Algorithm

We further refine the idea of Section 4 to get an algorithm that runs in $O(|E|\log|E| + m^{0.793}(|E|)$, where 0.793 is used to approximate $\log_4 3$. The limitation of the algorithm of Section 4 was that it only optimized the

searching along the j index. We solve this problem using a type of two-dimensional binary search. Once the edges have been sorted by weight and mapped to integers in the range $[1..m]$, construct an $m \times m$ table, T with Boolean entries. Table entry $T[b, a]$ indicates whether a path can be found that has maximum edge weight at most b and minimum edge weight at least a. Initially all values in T are undefined.

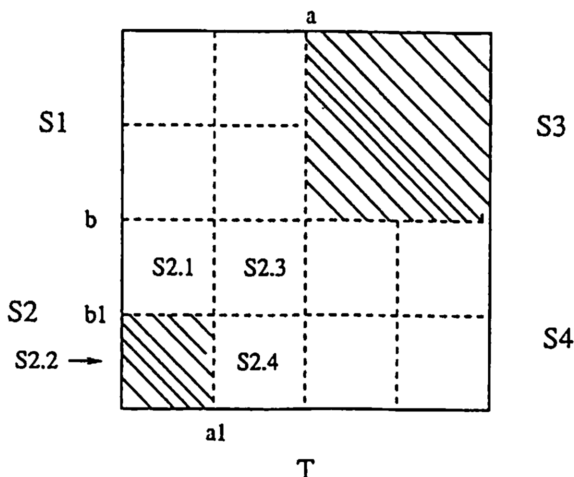
A queue of subtables will be used as a data structure. Initially the queue contains T. Observe that when we execute line 6a of the basic algorithm (with a and b used in place of i and j):

> test if there exists a path P' in G' from u to v such that all weights in P' are in range $[a..b]$

we actually learn a great deal of information. Suppose such a path does exist. Then it is superfluous to test for any paths having edge weights in the range $[c..d]$ where both $c \leq a$ and $d \geq b$. On the other hand suppose such a path does not exist. Then it is superfluous test for any paths having edge weights in the range $[e..f]$ where both $e \geq a$ and $f \leq b$. Note that path tests for table positions with $b < a$ may trivially answer "no such path exists" in $O(1)$ time.

The two-dimensional binary search proceeds as follows. Remove suitable S from the head of the queue. Suppose S is equal to $T[k..l, p..q]$. We maintain the original indices from T in all subtables. That is, S's entries are indexed by $[k..l, p..q]$. Locate the middle element of the suitable: position $S[(k+l)/2, (p+q)/2]$. Now test if there exists a path P' in G' from u to v such that all weights in P' are in range $[(k+l)/2..(p+q)/2]$. Based on the outcome of this test, from our observation above, we can eliminate at least one-fourth of the entries of S from future consideration. In other words, one fourth of the elements of S (and the corresponding positions in T) may be inferred from the results of the test just conducted. An example is shown in Figure 3. The shaded areas in Figure 3 represent entries of T that are known – either from a direct test or by inference. We then partition S into four disjoint subtables as follows:

$$S1 := S[k..(k+l)/2, p..(p+q)/2]$$
$$S2 := S[(k+l)/2..l.p..(p+q)/2]$$
$$S3 := S[k..(k+l)/2, (p+q)/2..q]$$
$$S4 := S[(k+l)/2..l, (p+q)/2..q]$$



Search Computation:
1. Path test at $T[a.b]$ was negative so Section 3 is discarded
2. Path test at $S2[a1, b1]$ was positive so S2.2 is discarded
3. Queuc now equals: S1, S4, S2.1, S2.3, S2.4

Figure 3. Recursive Example

To be more precise, if the outcome of the test if positive (i.e., a path is found), $S2$ should actually be defined to be $S[\lfloor (k+1)/2 \rfloor ..1, p..\lceil (p+q)/2 \rceil]$. And if the outcome of the test is negative, $S3$ should be defined to be $S3 :=$ $S[k..\lceil (k+1)/2 \rceil, \lfloor (p+q)/2 \rfloor ..q]$. If the outcome of the test were positive, add subtables $S1$, $S3$, and $S4$ to the queue, otherwise add subtables $S1$, $S2$, and $S4$ to the queue. Of course, if S were simply a one-dimensional table (i.e., $k = l$ or $p = q$) we simply partition S into two subtables. If S were contained a single entry (i.e., $k = l$ and $p = q$), we perform the test and do not add any subtables to the queue.

The reason we must add three subtables to the queue, in the worst case, is because the table indices do not correspond to edge weights from the original graph, G. For example, consider a graph with a path having edge weights in the range [1..3] and another path with edge weights in the range [2..5], where the range bounds are weights in G'. But it may be that these translate to paths in G with weights (1,100,101) and (100,101,102,103), respectively.

It is now easy to write a recurrence relation to describe the running time of step 6 of the algorithm. In so doing we assume the worst case: that each subtable is reduced in size by exactly one fourth. Of course, each iteration of the loops at step 6 requires $O(|V| + |E|)$ time in the worst case; for

simplicity call that term c in the recurrence since it is not a function of the recursion. Let R be the running time required for a graph with m distinct edge weights.

$$R(1) = c$$
$$R(m) = 3R(m/4) + c$$

A solution to the recurrence is $O(m^{**} \log_4 3)$, which can be shown using the Master Theorem [2]. The careful reader will observe that the table T and its subtables do not actually need to be built, rather we can keep the indices bounding the dimensions of each table/subtable in the queue. In this way each subtable can be represented with at most four integers. Thus an efficient implementation can be made to run in $O(|E| \log |E| + m^{0.793}(|E|))$ time. Of course, depending of m, this ranges from $O(|E| \log |E|)$ in the best case to $O(|E|^{1.793})$ if the number of distinct edge weights approaches $|E|$.

Acknowledgements

The author wishes to thank Professor George Trapp for his helpful discussions and comments and the anonymous referee for their valuable comments.

References

[1] D. Dobkin and R. Lipton, Multidimensional Searching Problem, *SIAM J. Comput.*, 5 (1976), 181–186.

[2] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1992.