Single-Source Shortest Paths in Planar Graphs

Stuart Baker, Ömer Cerrahoğlu, Sebastian Claici

Abstract

We survey advances in single-source shortest paths algorithms on the special case of planar graphs. For the case of non-negative edge weights, we show $O(n \log \log n)$ algorithm and provide intuition for an optimal $O(n)$ algorithm. For arbitrary edge weights, we give a $O(n \log^2 n)$ algorithm and show how to modify it to achieve $O(n \log^2 n / \log \log n)$.

1 Introduction

Finding shortest paths in graphs is one of the oldest combinatorial optimization problems. At least in the sense of path finding, there are references dating back to the early 1800's (reference). The standard algorithm for single-source shortest paths on a graph with no negative edge weights is Dijkstra's algorithm (reference).

While for general graphs the problem has been effectively solved decades ago, for the special case where the graph is planar has seen a number of surprising developments in recent years. We enumerate these here, and detail them in later sections.

For directed graphs where the edge weights are non-negative, the first improvement over $O(n \log n)$ came from Federickson who gave a $O(n\sqrt{\log n})$ bound [1]. This was improved nearly a decade later by Henzinger et al. to linear time [2]. We remark that for general *undirected* graphs with positive edge weights, there are known linear time algorithms for single-source shortest paths in the RAM model [8].

For directed graphs with arbitrary edge weights, there have been recent developments that greatly improve upon a naive $O(n^2)$ Bellman-Ford. In 2010, Klein, Mozes and Weimann gave a $O(n \log^2 n)$ algorithm [4], which was improved to $O(n \log^2 n / \log \log n)$ shortly thereafter [5]. Both algorithms make use of a $O(n \log n)$ multiple-source shortest paths algorithm due to Klein [3].

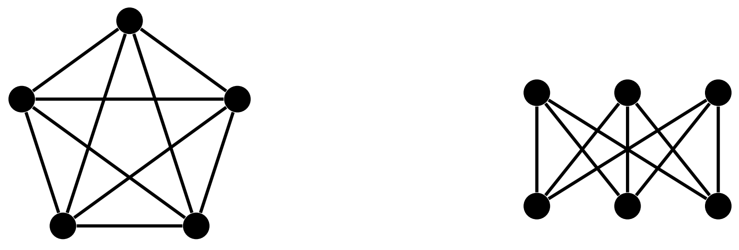

Figure 1: Kuratowski subgraphs. (a) K_5, (b) $K_{3,3}$

2 Background

Later on, we make use of several properties of planar graphs in later sections. We provide background on the required theorems here.

Theorem 2.1 (Jordan curve theorem)**.** *Let C be any closed curve in the plane. Removal of C divides the plane into exactly two connected regions, the "inside" and the "outside" of C.*

An interesting and relevant proof of the Jordan curve theorem that uses the non-planarity of $K_{3,3}$ was given in [7].

Theorem 2.2. *Any n-vertex planar graph with $n \geq 3$ contains no more than $3n - 6$ edges.*

Theorem 2.3 (Kuratowski's theorem)**.** *A graph is planar if and only if it contains neither a complete graph on five vertices, nor a complete bipartite graph on two sets of three vertices as a generalized subgraph.*

An in-depth discussion of Kuratowski's theorem and a short proof are found in [6]. The Kuratowski subgraphs are shown in figure 1.

Theorem 2.4. *Every node v of a planar graph can be embedded on the boundary of the infinite face f_∞.*

Embedding nodes on the outer boundary is accomplished through stereographic projections (figure placeholder).

Finally, specifically for shortest paths algorithms, we use the fact that every planar graph can be transformed by adding $O(n)$ vertices and edges into another planar graph such that every node has indegree and outdegree at most 2.

3 Planar Graph Separators

3.1 Overview

For large planar graphs, a common approach to solving a specific problem is to take a divide-and-conquer approach. By dividing the problem into two or more smaller chunks and solving each subproblem, calculations that are difficult on a large scale can still be done. The core problem in divide-and-conquer problems is finding a way to divide the problem space into smaller spaces and recurse until you reach a subspace that is small enough to allow for effective computation. The results at the lowest level are then rolled back up through the recursion to give the final answer.

In order to tackle planar graphs, there are several important relations that we rely on. If C is some closed curve in the plane, by removing C you can divided the plane into exactly two connected regions: the inside region and the outside region. Additionally, Kuratowski's theorem states that a graph is planar if and only if it does not have any generalized subgraphs that are either a complete graph of five nodes or a complete bipartite graph of two sets of three nodes. From Kuratowski's theorem we know that we can shrink any edge of a planar graph to a single vertex and preserve the planarity. Expanding this idea, we can shrink any subgraph of a planar graph to a single vertex and still have a planar graph.

3.2 Fundamental Cycle Separators

On method for dividing a planar graph into smaller set was developed by Lipton and Tarhan called the fundamental cycle separator. Lipton and Tarjan demonstrated that for any planar graph $G = (V, E)$ on $n = |V|$ vertices, and for any weight function $w : V \to \mathbb{R}^+$, it is possible to partition the nodes in the graph into three sections $A, B, C, \subseteq V$ with the following qualities.

- $w(A), w(B) \leq \alpha \cdot w(V)$ for some $\alpha \in (0, 1)$
- There are no edges between any node in A ($a \in A$) and any other node in B ($b \in B$), ($A \times B \cap E = \emptyset$)
- The size of the separator S is small, $|S| \leq f(n)$, specifically $\frac{2}{3}$

Fro a root vertex r, you can build a spanning tree of the graph G that has depth d. You can then define T^* as the dual tree of the triangulated version of G. From this tree, every non-tree edge e defines a fundamental cycle $C(e)$. Since the depth of T is at most d, we know that $|C(e)| \leq 2d + 1$. However, because the diameter of G may be large, we want to reduce it so that we can constrain $|S| \leq \sqrt{n}$. Central to developing this better partition is the ability to divide the planar graph into levels. Given some n-vertex planar graph G with nonnegative vertex costs, it is possible to partition the vertices of the graph based on their distance from a vertex v. One method for finding this partitioning is to run a breadth-first search from v. Given the partitioning, we can define $L(l)$ as the number of vertices on the level l which is the distance from v. The levels range from 0 to r where r is the maximum distance from v to any vertex in the graph. For the algorithm to work, an additional, empty, level must be added at $r + 1$.

The algorithm is as follows:

1. Find the most costly component in the graph and run a depth-first search from this graph. This calculates the level of each vertex in the graph in provides the level values $(L(l))$ for every l. For the maximum depth r in the level tree, add an additional level at $r + 1$ that contains no vertices. This can be performed in $\mathcal{O}(n)$.
2. Find the level i_0 that contains the median vertex. This is the level where $\sum_{i \leq i_0} |L_i(v)| \geq \frac{n}{2}$ and $\sum_{i \geq i_0} |L_i(v)| \geq \frac{n}{2}$. This can be performed in $\mathcal{O}(n)$.
3. Find the levels $i_- \leq i_0 \leq i_+$. This can be performed in $\mathcal{O}(n)$.
    - Start from the median vertex containing level i_0 and increase i_+ as well as decrease i_- until $|L_{i_-}|, |L_{i_+}| \leq \sqrt{n}$
    - Because each section can only contain half of the vertices, we can use the counting argument to state that $|i_0 - i_-|, |i_+ - i_0| \leq \frac{\sqrt{n}}{2}$
    - At this point we have the separator $|L_{i_-} \cup L_{i_+}| \leq 2\sqrt{n}$, which we can return if some grouping of $L_{<i_-}$, $L_{>i_+}$, and L_{i_-,i_+} is balanced.
4. From a condensed graph $G^{'}$
    - Delete or contract all edges in $L_{\geq i_+}$
    - Contract all edges in $L_{\leq i_-}$ to form a super-vertex v that is connected to all vertices $u \in L_{i_-+1}$
5. From the condensed graph $G^{'}$, create a fundamental cycle separator
    - Build a breadth-first tree from $G^{'}$ from v. This tree will have a depth $|i_+ - i_-| \leq \sqrt{n}$
    - Triangulate the BFS tree.
    - Apply the fundamental cycle separator lemma presented above.
6. Using this separator, return A and B as some combination of $\text{int}(C)$, $\text{ext}(C)$, $L_{<i_-}$, and $L_{>i_+}$. S can be returned as a combination of L_{i_-}, L_{i_+}, and C.

3.3 Miller's Algorithm

3.4 Recursive Segmentation

The graph segmentation algorithms presented above can be used to recursively break a graph apart for a divide-and-conquer approach to solving problems. Specifically, the graph can be divided into $\Theta\left(\frac{n}{r}\right)$ regions, each of which have $\mathcal{O}(r)$ vertices and a total of $\mathcal{O}\left(\frac{n}{\sqrt{r}}\right)$ boundary vertices. Miller takes this definition a step further and defines a *suitable* r-division of a planar graph as the r-division that satisfies two characteristics:

1. Each boundary vertex is contained in at most three regions
2. Any region that is not connected consists of connected components, all of which share boundary vertices with exactly the same set of either one or two connected regions.

Starting with the initial graph G, all of the vertices are in the interior region. A separator algorithm can then be applied to the graph with all of the vertex weights set to $\frac{1}{n}$. This will produce three sets: A, B, and C. From these two sets, we can infer two regions that have the vertex sets $A_1 \subseteq A \cup C$ and $A_2 \subseteq B \cup C$. These two vertex sets will have sizes $\alpha n + \mathcal{O}\sqrt{n}$ and $(1-\alpha)n + \mathcal{O}\sqrt{n}$ with $\frac{1}{3} \leq \alpha \leq \frac{2}{3}$. To continue the process, the separator algorithm can be recursively applied to any region that has more than r vertices. The total runtime of this algorithm is $\mathcal{O}\left(n \log\left(\frac{n}{r}\right)\right)$.

However, to ensure that the two qualities that Miller enumerated hold, additional steps must be made. After applying the planar separator algorithm, there are three sets of vertices: A, B, and C. If we say that C' is the set of vertices in C not adjacent to any vertex in $A \cup B$, the we can define $C'' = C - C'$. Next, we need to identify the connected components $A_1, A_3, \ldots, A_q$ in $A \cup B \cup C'$. Using this information, we can remove any vertex v from C'' and insert it into A_i if that vertex is adjacent to a vertex in A_i, but not adjacent to a vertex in A_j for $i \neq j$. This will ensure that a boundary vertex will be in at most three subgraphs. However, this does not ensure that there are at most $\Theta\left(\frac{n}{r}\right)$ connected subgraphs. To ensure that there are no more than $\Theta\left(\frac{n}{r}\right)$ connected subgraphs, we can apply a greedy approach. Sweep through the set of connected regions, join together any two neighboring regions that each have less than $\frac{r}{2}$ vertices.

4 Single source shortest paths with nonnegative edge weights

4.1 Simple algorithm

For a planar graph with nonnegative edge weights, Dijkstra's algorithm runs in $O(n \log n)$ as $m \leq 3n - 6$. It is possible to improve this to $O(n)$. To get there, recall that and r-division of a planar graph is a partition of the graph into $\Theta(n/r)$ regions of size $O(r)$ with boundary size $O(\sqrt{r})$. An r-division of a planar graph can be computed in linear time.

A simple $O(n\sqrt{\log n \log\log n})$ emerges quite beautifully just from the r-division if we set $r = \frac{\log n}{\log\log n}$. The algorithm follows a divide-and-conquer approach in which each region is processed first, followed by a clean-up phase where the results are merged.

```
for all Regions $R$ do
    for all Boundary nodes $v \in R$ do
        Compute SSSP from $v$ in $R$
        Store $(u, v)$ distances for any two boundary nodes $u$, $v$
    end for
end for
```

The first step is to compute the single-source shortest paths for each boundary node in each region R (algorithm 1). We can now replace each region R by a complete graph on R's boundary nodes with shortest paths distances between any two nodes. Call this auxiliary graph G'. The second phase of the algorithm is to compute the SSSP from s in G'. This gives the true shortest paths from s to all the boundary nodes. Finally, we must tidy up by finding the distances from s to the nodes inside each region (algorithm 2).

```
for all Regions $R$ do
    for all Boundary nodes $v \in R$ do
        Set $d(v) = d_{G'}(s, v)$
        Compute SSSP from $v$ in $R$
    end for
end for
```

To analyze the algorithm, we will need a few pieces of information:

- Total number of boundary nodes is $O(\sqrt{r})O(n/r) = O(n/\sqrt{r})$.
- Number of nodes in G' is $O(n/r)O(\sqrt{r}) = O(n/\sqrt{r})$.
- Number of edges in G' is $O(n/r)O(r) = O(n)$.

Using $r = \frac{\log n}{\log\log n}$, the first phase is bounded above by

$$O\left(n\frac{\sqrt{\log\log n}}{\sqrt{\log n}}\log n\right) = O(n\sqrt{\log n \log\log n}),$$

the second phase is an SSSP in a size $O(n\frac{\sqrt{\log\log n}}{\sqrt{\log n}})$ graph, and thus also $O(n\sqrt{\log n \log\log n})$, while the tidying up is a series of SSSPs in each of the regions, and has the same bound as the first phase—$O(n\sqrt{\log n \log\log n})$. The total time bound ends up $O(n\sqrt{\log n \log\log n})$.

4.2 Recursion

The simple algorithm shown above is an improvement over Dijkstra's algorithm, but one can do better. In fact, we can achieve linear time by recursively subdividing the graph. Without loss of generality, assume the graph is directed, and that each node has at most two incoming and two outgoing edges. We call a region atomic if it contains only one edge uv. A nonatomic region will have as children subregions that are contained within it.

For each region R, we maintain a priority queue $Q(R)$ that stores the subregions of R if R is nonatomic, or the single arc uv is R is atomic. The algorithm ensures that for every region R, the minimum element of $Q(R)$ is the minimum label $d(v)$ over all edges vw in R that remain to be processed.

```
1: Find recursive subdivision $R(G), R(P_i), \ldots, R(uv)$
2: Allocate queue $Q$ for each region
3: $d(v) \leftarrow \infty, \forall v$
4: $d(s) \leftarrow 0$
5: **for all** $sv \in E(G)$ **do**
6:     Update$(R(sv), sv, 0)$
7: **end for**
8: **while** $Q(R(G)).minKey() < \infty$ **do**
9:     Process$(R(G))$
10: **end while**
```

The algorithm repeatedly calls two subprocedures Process and Update to process individual regions and update parent regions. Unlike Dijkstra's algorithm, the work we do in a region is only speculative. Often we cannot afford to fully process a region, and executions are stopped after a fixed number of steps.

By playing around with the number of levels, number of nodes per region per level, and the α_i, we can achieve linear time. To give an intuition, we present here a $O(n \log\log n)$ algorithm and briefly comment on how to change it to get rid of the $\log\log n$ factor.

```
procedure PROCESS
    if $R$ contains only $uv$ then
        if $d(v) > d(u) + c(u,v)$ then
            $d(v) \leftarrow d(u) + c(u,v)$
            for each outgoing edge $vw$ of $v$, call UPDATE($R(vw), vw, d(v)$)
        end if
        $Q(R).updateKey(uv, \infty)$
    else
        repeat
            $R' \leftarrow Q(R).getMin()$
            PROCESS(()$R'$)
            $Q(R).updateKey(R', Q(R').minKey())$
        until $Q(R).minKey()$ is infinity or if repeated $\alpha_{h(R)}$ times
    end if
end procedure
```

```
procedure UPDATE($R, x, k$)
    $Q(R).updateKey(x, k)$
    if $updateKey$ reduced the value of $Q(R).minKey()$ then
        UPDATE($parent(R), R, k$)
    end if
end procedure
```

4.2.1 Correctness

Recall from Dijkstra's algorithm that three properties imply correctness:

1. Initialization: $d(s) = 0$.
2. Minimum length property: $d(v)$ is an upper bound on the s to v distance
3. Edges are relaxed: $d(v) \leq d(u) + c(u,v), \forall uv \in E$

The first property is true at the start of the algorithm, and remains true throughout. The following lemma establishes the second property:

Lemma 4.1. *For each node v, $d(v)$ is an upper bound on the distance from s to v throughout the algorithm.*

Proof. Initially, all labels except $d(s)$ are infinity. The labels only get changed in line 4 of PROCESS, and assuming inductively that the old labels $d(v)$ and $d(u)$ are upper bounds on the distance to u and v, it follows that the new labels will also be upper bounds. □

Similarly, the following three lemmas establish the third property.

Lemma 4.2. *If an edge uv is inactive then it is relaxed.*

Proof. The lemma holds before the first call to PROCESS as every node but s has label infinity and outgoing edges from s are active. Edges are deactivated only in line 7 of PROCESS which occurs only after the edge has been relaxed.

Note that it is possible that an edge becomes unrelaxed after changes to the labels of its endpoints. This can occur for a call to UPDATE, but line 2 of UPDATE changes the key of the edge, making it active again. □

Lemma 4.3. *The key of an active edge uv is $d(u)$ (except during lines $3-6$ of* PROCESS*).*

Proof. Whenever a label $d(u)$ is assigned a value k, UPDATE$(R(uv), uv, k)$ is called for each outgoing edge uv, and the key of uv is updated to k. □

Lemma 4.4. *For any region R that is not an ancestor of the current region, the key associated with R in $Q(parent(R))$ is the minimum key of $Q(R)$.*

Proof. Whenever the minimum key of a queue $Q(R)$ is changed in line 2 of UPDATE, the recursive call in line 4 ensures that the key associated with R in the parent of R is also changed. □

From lemma 4.4, the following corollary follows:

Corollary 4.4.1. *For any region R that is not an ancestor of the current region,*

$$Q(R).minKey() = \min\{d(v) | uv \text{ is a pending edge contained in } R\}$$

When the algorithm terminates, the minimum key of the priority queue associated with the entire graph will be infinity, and by corollary 4.4.1 all edges will be relaxed.

4.2.2 Analysis

A recursive subdivision of a graph induces a hierarchy of levels. We will say that edges are at level 0, and levels increase up to the region containing the entire graph.

We show that dividing the graph into $O(n/\log^4 n)$ regions of size $O(\log^4 n)$ with boundaries of size $O(\log^2 n)$ and setting $\alpha_1 = \log n$ and $\alpha_2 = 1$ yields an $O(n \log\log n)$ single source shortest paths algorithm. Note that the division is simply an r-division with $r = \log^4 n$. There are 3 levels in this division: the edges, the regions of the r-division, and the whole graph.

We restate the algorithm:

1. Select the region containing the lowest labeled node that has active outgoing edges in the region.
2. Repeat $\log n$ times: Select the lowest labeled node v in the current region that has active outgoing edges in the region. Relax and deactive its outgoing edges vw in that region. For each of the other endpoints w of these edges, if relaxing the edge vw resulted in decreasing the label of w, then activate the outgoing edges of w.

The majority of the time is spent in invocations of PROCESS. We say that an invocation of PROCESS on region R is *truncated* if $Q(R).minKey()$ is infinity at the end of the invocation. All level 0 invocations are truncated. The crux of the analysis relies on the following *charging invariant* (the proof of which is in appendix A):

> For any pair (R, v) of region R and entry node v, there is an invocation B of PROCESS such that all invocations charging to (R, v) are descendants of B (or B itself).

Intuitively, the *charging invariant* says that the number of charges to any pair is small, so the number of truncated invocations will be small.

Let us first consider the number of pairs (R, v) on each level to which we can charge truncated invocations to (recall that v is an entry node into R):

- If R has level 0, then R is charged by at most one level 0 invocation. There are $O(n)$ pairs (R, v) on level 0, and thus $O(n)$ chargers.
- If R has level 1, the pair is charged by at most one level 1 invocation and at most $\alpha_1 = \log n$ level 0 invocations. There are $O(n/\log^4 n) \cdot O(\log^2 n)$ pairs on level 1.
- If R has level 2, then the pair is charged by at most one level 2 invocation, at most $\alpha_2 = 1$ level 1 invocations, and at most $\alpha_2\alpha_1 = \log n$ level 0 invocations. There is only one pair (R, v) on level 2, namely (R_G, s).

Let s_i be the total number of invocations at level i (truncated and non-truncated), and t_i be the number of truncated chargers at level i.

All level 0 invocations are truncated, thus

$$s_0 = O(n)$$

Each non-truncated level j invocation results in α_j invocations at level $j - 1$. Hence the number of level 1 invocations is

$$s_1 \leq s_0/\alpha_1 + t_1 = O(n/\log n) + O(n/\log^2 n).$$

Similarly, the total number of level 2 invocations is

$$s_2 \leq s_1/\alpha_2 + 1 = s_1 + 1 = O(n/\log n).$$

Let's look at the time spent per invocation at each level. We bound the time required per queue operation; since at level i there are α_i calls to lower levels, and thus α_i queue operations, this gives us a time bound per invocation.

- At level 0, there is only one item in the queue, so operations take constant time, and there are $\alpha_0 = 0$ calls to lower levels.
- At level 1, queues are of size $O(log^4 n)$, so queue operations take $O(\log\log n)$ time. There are $\alpha_1 = \log n$ calls to lower levels, for a total of $O(\log n \log\log n)$.
- At level 2, queues are of size $O(n/\log^4 n)$, so queue operations take $O(\log n)$ time, and there are $\alpha_2 = 1$ calls to lower levels, for a total of $O(\log n)$.

To get our total time bounds, we multiply the time per invocation with the number of invocations at each level, and add everything up.

$$\begin{aligned}\text{Total} &= O(\log n)\cdot O(n/\log n) + O(\log n \log\log n)\cdot O(n/\log n) + O(1)\cdot O(n)\\ &= O(n\log\log n)\end{aligned}$$

The information is summarized in table 1.

Table 1: Time required for PROCESS calls.

Level	Calls	Time per invocation	No. (R, v) pairs	No. invocations	Total time
2	1	$O(\log n)$	1	$O(n/\log n)$	$O(n)$
1	$\log n$	$O(\log n\log\log n)$	$O(n/\log^2 n)$	$O(n/\log n)$	$O(n\log\log n)$
0	0	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Total					$O(n\log\log n)$

We have bounded the time for PROCESS, but we still have to worry about the calls to UPDATE. Fortunately, an $O(n\log\log n)$ bound for UPDATE follows quickly. We can assume that the graph has in- and out-degree at most 2. First note that each call to UPDATE starts at level 0, so there are $O(n)$ calls to worry about. If the call stops on or before level 1, then all we need to do is update a key in a queue of size $\log^4 n$. The total time for all calls to UPDATE that stop at or before level 1 is $O(n\log\log n)$.

Otherwise, recall that $O(n/\log n)$ level 0 invocations of PROCESS are charged to each level 1 or level 2 region. Their calls to UPDATE take $O((n/\log n)\log n) = O(n)$ time. The remaining cases are level 0 invocations of PROCESS that are charged to themselves. We must show that at most $O(n/\log n)$ of these invocations result in a call to UPDATE that reaches the top level. We associate each level 0 invocation of PROCESS that is charged to region $R(uv)$ with the node v whose label is being updated. Recall that v has in-degree at most 2, so there are only 2 level 0 invocations associated with a given node. If we reach the top level, it must be the case that v was a boundary node for a level 1 region for otherwise it could not have been the minimum key. Since there are only $O(n/\log^2 n)$ boundary nodes on level 1, the total time needed for calls to UPDATE in this case is $O((n/\log^2 n)\log n) = O(n/\log n)$.

5 Single source shortest paths with arbitrary edge weights

References

[1] Greg N Federickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16(6):1004–1022, 1987.

[2] Monika R Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian. Faster shortest-path algorithms for planar graphs. *journal of computer and system sciences*, 55(1):3–23, 1997.

[3] Philip N Klein. Multiple-source shortest paths in planar graphs. In *SODA*, volume 5, pages 146–155, 2005.

[4] Philip N Klein, Shay Mozes, and Oren Weimann. Shortest paths in directed planar graphs with negative lengths: A linear-space $O(n\log^2 n)$-time algorithm. *ACM Transactions on Algorithms (TALG)*, 6(2):30, 2010.

[5] Shay Mozes and Christian Wulff-Nilsen. Shortest paths in planar graphs with real lengths in $O(n\log^2 n/\log\log n)$ time. In *Algorithms–ESA 2010*, pages 206–217. Springer, 2010.

[6] Carsten Thomassen. Kuratowski's theorem. *Journal of Graph Theory*, 5(3):225–241, 1981.

[7] Carsten Thomassen. The Jordan-Schonflies theorem and the classification of surface. *American Mathematical Monthly*, pages 116–131, 1992.

[8] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM (JACM)*, 46(3):362–394, 1999.