The Densest k Subgraph Problem in b-Outerplanar Graphs

Sean Gonzales and Theresa Migler(✉)

California Polytechnic State University, San Luis Obispo, CA 93405, USA
tmigler@calpoly.edu

Abstract. We give an exact $O(nk^2)$ algorithm for finding the densest k subgraph in outerplanar graphs. We extend this to an exact $O(nk^2 8^b)$ algorithm for finding the densest k subgraph in b-outerplanar graphs. Often, when there is an exact polynomial time algorithm for a problem on b-outerplanar graphs, this algorithm can be extended to a polynomial time approximation scheme (PTAS) on planar graphs using Baker's technique. We hypothesize that this is not possible for the densest k subgraph problem.

Keywords: Densest k subgraph problem · Density · Outerplanar graphs

1 Introduction

The *density* of a graph is defined to be the ratio of edges to vertices in the graph. More precisely, if an undirected graph $G = (V, E)$ has $|V| = n$ vertices and $|E| = m$ edges, the density of G is $\frac{m}{n}$.

In a network that represents academic collaboration, authors within the densest component of the graph tend to be the most prolific [15]. Dense components in a web graph might correspond to sets of web sites dealing with related topics [10] or link farms [11]. Finding dense subgraphs aids in finding price value motifs [7]. Dense subgraphs can identify communities in social networks [5]. In the field of visualization, finding dense subgraphs allows for better graph compression [4]. Dense subgraphs assist in finding stories and events in micro-blogging streams such as Twitter [1]. Dense subgraphs can be used to discover regulatory motifs in genomic DNA [9], and to find correlated genes [14]. It is therefore interesting to find the dense components of graphs, or *dense subgraphs*. The density of a subgraph induced by a vertex set $S \subseteq V$ is $d(S) = \frac{|E(G[S])|}{|S|}$. Goldberg gave an $O(nm \log n \log(\frac{n^2}{m}))$ algorithm to find a subgraph of maximum density using network flow techniques [12].

Given an undirected graph $G = (V, E)$ and an integer k, the *densest k subgraph problem* is defined as follows: find a subgraph $H \subseteq G$ such that $|V(H)| = k$ and the density of H is maximized. This problem can be shown to be NP-Hard by a reduction from Clique [8]. Since (if $P \neq NP$) there is no polynomial time algorithm for this problem, we restrict our domain. We consider the densest k

H. Cherifi et al. (Eds.): COMPLEX NETWORKS 2019, SCI 881, pp. 116–127, 2020.
https://doi.org/10.1007/978-3-030-36687-2_10

subgraph problem on *planar graphs*, graphs that can be drawn in the plane in such a way that no two edges cross each other. While the complexity of the unconnected densest k subgraph problem on planar graphs is unknown [6], the connected planar densest k subgraph problem is NP-Complete by a reduction from the connected vertex cover problem on planar graphs with maximum degree 4 [13]. Therefore, we further restrict our domain to *outerplanar* and *b-outerplanar graphs*. An *outerplanar graph* (or a *1-outerplanar graph*)is a planar graph with an embedding such that all vertices are on the outer face. A *2-outerplanar graph* is a planar graph with the property that, when the vertices on the unbounded face are removed, the remaining vertices all lie on the unbounded face. A *b-outerplanar graph* is a planar graph with the property that, when the vertices on the unbounded face are removed, the remaining graph is $(b-1)$-*outerplanar*.

In what follows we begin with an exact polynomial time algorithm for the densest k subgraph problem in outerplanar graphs inspired by Baker [2] in Sect. 2. There have been other exact polynomial time algorithms for this problem in outerplanar graphs, but to the authors' knowledge, none that use Baker's technique [3]. We extend this algorithm to b-outerplanar graphs in Sect. 3. Often, when there is an exact polynomial time algorithm for a problem on b-outerplanar graphs, this algorithm can be extended to a polynomial time approximation scheme (PTAS) on planar graphs using Baker's technique. We hypothesize that this is not possible for the densest k subgraph problem in Sect. 4.

2 An Algorithm to Find the Densest k Subgraph Problem in Outerplanar Graphs

Baker presented a dynamic programming algorithm for the independent set problem on outerplanar graphs [2]. We modify this dynamic program to an algorithm for the densest k subgraph problem on outerplanar graphs. Suppose we are given an *outerplanar graph*, a planar graph with an embedding such that all vertices are on the outer face. We will define a rooted labelled tree, T, that will correspond to the given outerplanar graph. This tree construction was given by Baker [2]. We repeat it here for completeness.

Suppose we are given an outerplanar graph, G. If G has any bridges (an edge whose removal disconnects the graph), (x, y), add an additional edge (x, y). Then the bridge may be treated as a face. We will call edges *exterior* if they lie on the outer face and *interior* otherwise.

Now we build the tree, T: First we suppose that G has no cutpoints (vertices whose removal disconnects the graph). For a description of the tree construction on an outerplanar graph with cutpoints, see [2]. We place a tree vertex in each interior face (call these *interior tree vertices*) and on each exterior edge (call these *exterior tree vertices*). We add a tree edge between each interior tree vertex and the interior tree vertices of adjacent faces. We also add a tree edge between each interior tree vertex and any exterior tree vertices with edges adjacent to the face. For a very simple example, see Fig. 1.

We may choose any interior tree vertex to be the root of T. We may also choose which child of this root will be the leftmost child. These two choices

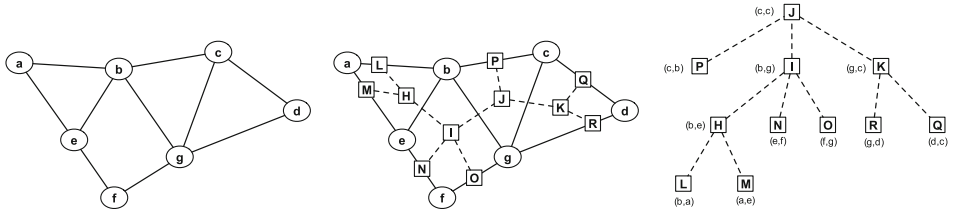

Fig. 1. The left figure is an example outerplanar graph, G. The middle figure shows the construction of T (with square vertices and dashed edges) from G. The right figure shows the labeling of T after making the choice to have vertex J as the root with P as J's leftmost child.

determine the ordering of all remaining vertices. We label the vertices of T recursively. Label each exterior tree vertex in the tree with the exterior edge that it is on. Label each interior tree vertex with the first and last vertices of its children's labels.

After constructing Baker's tree (as described above), we give our original dynamic program. We fill in a table for each vertex, v, in T (with label (x, y)). The table will hold the maximum number of edges for a subgraph on k vertices (k ranging from 0 to the size of the subgraph for the subtree rooted at v) depending on whether or not x and y are in the set. For example, consider the leaf vertex, L, in T representing edge (b, a) in G:

$Table(L) = T_{(b,a)} =$

b	a	$k=0$	$k=1$	$k=2$
0	0	0	$\emptyset$	$\emptyset$
0	1	$\emptyset$	0	$\emptyset$
1	0	$\emptyset$	0	$\emptyset$
1	1	$\emptyset$	$\emptyset$	1

This table is undefined for many entries, for example, the entry where $b = 0$, $a = 1$, and $k = 2$ is undefined because there is no way to have a subgraph on 2 vertices when there is only one vertex to select from. For the entry where $b = 1$, $a = 1$, and $k = 2$, we obtain a value of 1 because there is an edge between b and a. The table will be identical for each leaf in T.

Now we will fill out a table for a tree vertex with exactly two children. This table is calculated by *merging* the tables for its two children, as described below. Consider H with label (b, e):

$Table(H) = T_{(b,e)} =$

b	e	$k=0$	$k=1$	$k=2$	$k=3$
0	0	0	0	$\emptyset$	$\emptyset$
0	1	$\emptyset$	0	1	$\emptyset$
1	0	$\emptyset$	0	1	$\emptyset$
1	1	$\emptyset$	$\emptyset$	1	3

We now give pseudocode for how to *merge* two sibling tables, $T_{(x,y)}$ and $T_{(y,z)}$ (creating $T_{(x,z)}$). For the table of a vertex with label $\mathcal{L}$, $\mathcal{K}$ will be the minimum of k (the input k for the densest k subgraph problem) and the number of vertices in the subtree represented by label $\mathcal{L}$.

Algorithm 1. Merge$((T_{(x,y)}, T_{(y,z)}, \mathcal{K}))$

```
for Each (b_x, b_z) in {(0,0),(0,1),(1,0),(1,1)} do
    values = []
    for Each k from 0 to K do
        for Each b_y in {0,1} do
            for Each k_x from b_x + b_y to k do
                k_z = k − k_x + b_y
                if x == z AND (b_x AND b_y) then
                    k_z++
                value = T_(x,y)(b_x, b_y, k_x) + T_(y,z)(b_y, b_z, k_z)
                if x ≠ z AND (b_x AND b_z) AND (x, z) is an edge then
                    value++
                if value ≠ ∅ then
                    Add value to values
    if values is not empty then
        T_(x,z)(b_x, b_z, k) = max(values)
    else
        T_(x,z)(b_x, b_z, k) = ∅
```

To find the solution for the densest k subgraph problem, we look to the table for the root, in the case of our example, the table for J. We take the maximum value in the table for the column corresponding to k.

$Table(J) = T_{(c,c)} =$

c	c	$k=0$	$k=1$	$k=2$	$k=3$	$k=4$	$k=5$	$k=6$	$k=7$
0	0	0	0	1	3	4	6	7	$\emptyset$
0	1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	0	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	1	$\emptyset$	0	1	3	4	6	8	10

To find all intermediate tables, please see the extended version of the paper.

The following two lemmas give the proof of correctness of the dynamic programming algorithm.

Lemma 1. *For each leaf vertex, X (with label (a,b)), in the tree, $Table(X) = T_{(a,b)}$ correctly gives the maximum number of edges in the subgraph corresponding to X with the constrained number of vertices (each k value).*

Proof. Each leaf vertex, X, in the tree corresponds to a single edge, (a,b), in the original graph. There are only three possible values of k in this case, zero, one, or two.

If there are zero vertices included, the maximum number of edges is zero. However if we try to include either vertex a or b or both this would be absurd (because we are including zero vertices), so these entries should be undefined.

If one vertex is to be included, again it would be absurd to try to include zero or two vertices, so these entries are undefined. The only feasible possibilities are including only a for zero edges or only b for zero edges.

If two vertices are to be included, it would be absurd to include zero or only one vertex, so these entries are undefined. The only feasible possibility is to include both a and b and since there is an edge between a and b, we get one edge. □

Lemma 2. *For each vertex, V, in the tree $Table(V)$ correctly gives the maximum number of edges in the subgraph corresponding to V with the constrained number of vertices (each k value).*

Proof. If V is a leaf vertex, then Lemma 1 gives us that $Table(V)$ is correct. So, we may suppose that V is not a leaf vertex and inductively that the tables for the children of V are correct. Let $X_1, X_2, \ldots, X_{m-1}$ denote the children of V, and let the child X_i have label (x_i, x_{i+1}). This means that V has label (x_1, x_m). The table for V is then calculated by merging the tables for each X_i; that is, we first merge $Table(X_1) = T_{(x_1,x_2)}$ and $Table(X_2) = T_{(x_2,x_3)}$ to get a table $T_{(x_1,x_3)}$, then we merge $T_{(x_1,x_3)}$ with $Table(X_3) = T_{(x_3,x_4)}$ to get $T_{(x_1,x_4)}$ and so on, ultimately obtaining the table $T_{(x_1,x_m)}$. We claim that a call to *merge* on the tables for two consecutive tree vertices X and Y with labels (x, y) and (y, z), respectively, results in a correct table for the union of the subgraphs corresponding to X and Y, as well as the edge (x, z) (if it is an edge). If the claim is true, then the result of merging the tables of each child X_i must result in the correct table for V.

Table $T_{(x,z)}$ should have a row for each pair of values b_x and b_z, and each entry in the row corresponds to a particular value of k. So, fix b_x, b_z, and k. The entry $T_{(x,z)}(b_x, b_z, k)$ should contain the maximum number of edges possible over all subgraphs of k vertices that may contain x and z (depending on the values of b_x and b_z), where this subgraph is contained in the union described above. The *merge* procedure checks for the cases when we include y or do not include y in the subgraph by setting the bit b_y. So, fix b_y. The procedure then checks the egde count in every possible subgraph by varying the values of k_x and k_z, which are the k values we use to lookup edge counts in the tables $T_{(x,y)}$ and $T_{(y,z)}$, respectively. The idea is that we can check all possible subgraphs of size k containing the included vertices by checking subgraphs corresponding to X and Y such that the number of vertices in both subgraphs sums to k. In order to ensure that the values of k_x and k_z do in fact result in a correctly sized subgraph union, the procedure first sets k_x to be some value between $b_x + b_y$ and k (the lower bound is due to the fact that k_x is a vertex count, and b_x and b_y effectively count whether x and y are included). The procedure then sets k_z to be $k - k_x + b + y$; adding in b_y avoids double counting the inclusion of y (if it is included). However, if $x = z$ and $b_x = b_z = 1$ (i.e. x and z are both included in the subgraph), we have double counted the vertex x, hence the procedure increments the value of k_z to allow another vertex to be included in the union. The procedure then saves the value $T_{(x,y)}(b_x, b_y, k_x) + T_{(y,z)}(b_y, b_z, k_z)$ and increments it if (x, z) is an edge and both vertices are included (this is because each individual table does not account for the edge). Once all such values are computed, the procedure sets $T_{(x,z)}(b_x, b_z, k)$ to be the maximum value (if all values were undefined, the table entry is set to be undefined as well). Since each value corresponds to an edge

count of the corresponding subgraph and all subgraphs were checked, the table entry is thus the maximum number of edges. Therefore, the *merge* procedure results in a correct table for the corresponding subgraph, whence the table for V must also be correct. □

The running time for this dynamic program on outerplanar graphs is $O(nk^2)$: There is a table for each tree vertex and the number of tree vertices is the number of edges in the graph (plus 1 for the root which doesn't correspond to an edge). Also, since our graph is planar the number of edges is linear in the number of vertices. It takes $O(k^2)$ time to fill in each table.

3 An Algorithm to Find the Densest k Subgraph Problem in b-Outerplanar Graphs

We now lay out the dynamic programming solution for the densest k subgraph problem in b-outerplanar graphs. We define trees, labels, and slices similarly to Baker [2]. We assume that the given graph is connected. We define *levels*. Level 1 vertices form the outer face of the b-outerplanar graph. Level 2 vertices form the outer face if all level 1 vertices are removed, and so on. Additionally, we assume that the level i vertices contained in a level $i-1$ face form a connected subgraph called a *level i component*. If this is not the case, we may add fake edges that will simply be ignored when we calculate table values. Throughout this section, we will use the 3-outerplanar graph in Fig. 2 as a running example.

We construct a triangulation of our graph. We will use the triangulation to construct trees and slices. Any triangulation will do, and one can be constructed in linear time by scanning the vertices in levels i and $i+1$ in parallel for each $i = 1, 2, \ldots, k-1$. A triangulation for the graph in Fig. 2 is given in Fig. 3.

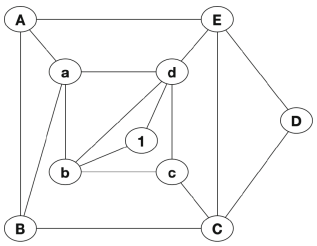

Fig. 2. 3-outerplanar example

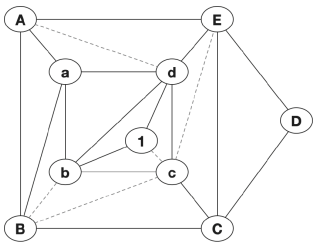

Fig. 3. Triangulation of example

Tree Construction. Since a level i component is an outerplanar graph, we can construct a tree for each level i component where the vertices of the tree represent interior faces and exterior edges of the component, just as in the previous section. However, we need to restrict how the roots and leftmost children are chosen for trees of level i components, where $i > 1$. For details, see Baker [2]. An example is given in Fig. 4.

Slice Construction. The dynamic programming solution follows a divide and conquer paradigm, where we divide the graph into so-called *slices* and calculate the tables for each slice before merging them together. Each vertex in each tree (and hence each interior face and exterior edge in each component of the graph) corresponds to a particular slice. The idea is to first define left and right *boundaries* for each tree vertex, and then define their slices by taking the induced subgraph of all vertices and edges that exist between the boundaries. The analogy here is that one can obtain a slice of pie by first deciding where the two cut lines will be (the left and right boundaries), and then the slice will be the pie that exists between your cuts. The full construction of slices is given in the full version of this paper.

In Fig. 4, each tree vertex has its left boundary vertices to the left and its right boundary vertices to the right (note that the right boundary vertices for a tree vertex are the same as the left boundary vertices of the next tree vertex).

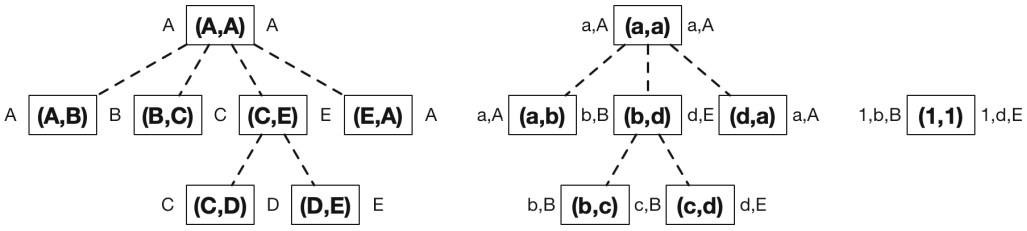

Fig. 4. Trees with slice boundaries

Dynamic Program. In this section, we detail the dynamic program that solves the densest k subgraph problem for b-outerplanar graphs. The dynamic program is given by the procedures *table, adjust, merge, extend,* and *contract*. Note that *adjust, merge, extend,* and *contract* are original to this paper whereas the *table* procedure is given by Baker [2]. The pseudocode for these procedures can be found in the extended version of the paper. The program constructs a table for each slice. The table for a level i slice consists of 2^{2i} entries: one entry for each subset of the boundary vertices (the left and right boundaries for a level i slice contain exactly i vertices each, for a total of $2i$ boundary vertices). An entry contains a number for each value of $k' = 0 \ldots k$, where this number is the maximum number of edges over all subgraphs of the slice of exactly k' vertices that contain the corresponding subset of the boundary vertices.

The main procedure of the program is *table* (find in extended version), which takes as input a tree vertex $v = (x, y)$. This procedure contains four conditional branches. The first branch handles the case when v represents a face f that does not enclose a level $i+1$ component. In this case, the procedure makes a recursive call to *table* on each child of v and merges the resulting tables together.

The second conditional branch handles the case when v represents a face f that encloses a level $i + 1$ component C. In this case, the procedure makes a recursive call to *table* on the tree vertex that represents C. The resulting table

is then passed to the *contract* function, which turns a level $i+1$ table into a level i table by removing the level $i+1$ boundary vertices from the table, and the contracted table is then returned.

The third conditional branch handles the case when v is a level 1 leaf. In this case, the procedure returns a template table that works for all level 1 leaf vertices, since any level 1 leaf represents a level 1 exterior edge of the graph.

The fourth conditional branch is slightly more complicated. This branch handles the case when v is a level $i > 1$ leaf vertex. The idea is to break up the slice of v into subslices, compute the table for an initial subslice, and *extend* this table by *merging* the tables for the subslices clockwise and counterclockwise from the initial subslice. These subslices have their own respective subboundaries. In the case that the slice of v is simply a line of vertices, no subslices can be created, so the procedure will return the table for the whole slice by passing the vertex to the *create* procedure, which effectively applies brute force to create the table for the subslice determined by the second parameter p (this is explained in more detail below).

In the case that the slice of v is not just a line, we can create tables for subslices. Since we are dealing with a planar graph, there exists a level $i-1$ vertex z_p such that all level $i-1$ vertices other than z_p that are adjacent to x are clockwise from z_p, and all level $i-1$ vertices other than z_p that are adjacent to y are counterclockwise from z_p. Here, z_p is the only level $i-1$ vertex in slice(v) that can be adjacent to both x and y (although it might not be adjacent to either). So, we construct an initial level i table for the subslice corresponding to z_p using *create*, and then we make as many calls as necessary to the *merge* and *extend* procedures (described below) to extend the table on one side with subslices constructed from the vertices adjacent to x, and on the other side with subslices constructed from the vertices adjacent to y.

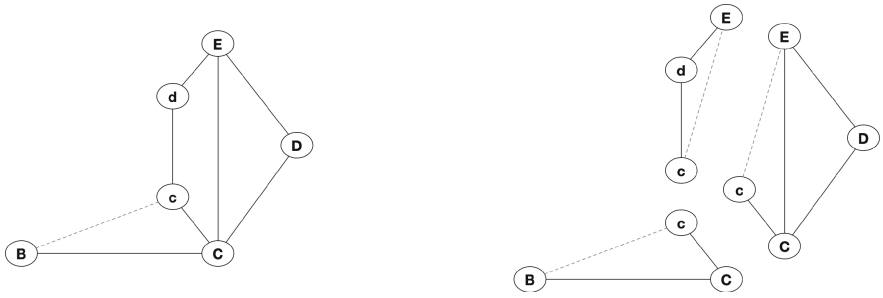

Fig. 5. The slice for (c, d) is split into subslices for computing tables.

For example, Fig. 5 shows how we split slice$((c, d))$ into subslices. The algorithm finds that E is a level 1 vertex such that the level 1 vertices adjacent to c are clockwise from E and the level 1 vertices adjacent to d are counterclockwise to E (of which there are none). A call to *create* constructs the table for the initial subslice with subboundaries c, E and d, E. We then merge the other subslices in

a clockwise fashion, first merging the table for the subslice with subboundaries c, C and c, E, and then merging the table for the subslice with subboundaries c, C and c, B.

The *adjust* procedure, (see extended version for pseudocode), takes as input a table T, which represents a slice with left boundary L and right boundary R. Let x and y be the highest level boundary vertices in L and R, respectively. This procedure checks if $x \neq y$, and if so, adds 1 to the table entry where x and y are both included. Unlike in Baker's original procedure, the case when $x = y$ is not handled in this procedure and is instead handled in *merge*.

The *merge* procedure, given in Fig. 6 in Sect. A of the Appendix, takes as input two tables T_1 and T_2 such that the right boundary of the slice that T_1 represents is the same as the left boundary of the slice that T_2 represents. Let the left and right boundaries of the slice that T_1 represents be L and M, and let the left and right boundaries of the slice that T_2 represents be M and R. The resulting table T returned by *merge* represents the slice with left and right boundaries L and R. This procedure constructs T by creating an entry for each subset A of $L \cup R$. As stated earlier, each entry contains a number for each value of k', where k' ranges from 0 to the number of vertices in the union of the two slices represented by T_1 and T_2, and where the number is the maximum number of edges over all subgraphs of exactly k' vertices containing A. Each of these individual subgraphs contains a different subset B of M.

The *contract* procedure, (see extended version for pseudocode), changes a level $i+1$ table T into a level i table T'. Here, T represents the slice for (z, z), where (z, z) is the root of a tree corresponding to a level $i+1$ component C contained in a level i face f, and T' is the table for the slice of vertex(f). Let $S = \text{slice}((z, z))$ and let $S' = \text{slice}(\text{vertex}(f))$. Let the left and right boundaries of S' be L and R, respectively. Then the left and right boundaries of S are of the form z, L and z, R respectively. For each subset of $L \cup R$ and for each value of k', T contains two numbers: one that includes z and one that does not include z. So, for each subset A of $L \cup R$ and each value of k', we set $T(A, k')$ equal to the larger of these two numbers.

The *create* procedure takes as input a leaf vertex $v = (x, y)$ in a tree that corresponds to a level $i+1$ component enclosed by a face f, and a number $p \leq t+1$, where the children of vertex(f) are $u_1, u_2, \ldots, u_t$. This procedure simply applies brute force to create the table for the subgraph containing the edge (x, y), the subgraph induced by the left boundary of u_p if $p \leq t$ or the right boundary of u_{p-1} if $p = t+1$, and any edges from x or y to the level i vertex of this boundary.

Lastly, the *extend* procedure, given in Fig. 7 in Sect. A of the Appendix, takes as input a level $i+1$ vertex z and a table T representing a level i slice, and produces a table T' for a level $i+1$ slice. Let L and R be the boundaries for the level i slice represented by T. The boundaries for the new slice will be $L \cup \{z\}$ and $R \cup \{z\}$. For each subset A of $L \cup R$ and each value of k', the new table T' contains two values: one that includes z and one that does not include z. The entries $T'(A, k')$ which do not include z can simply be set to their original values in T. For the entries $T'(A \cup \{z\}, k')$ that do include z, we first check

that $T(A, k'-1)$ is not undefined. If this is the case, we set $T'(A \cup \{z\}, k')$ as $T(A, k'-1)$ plus the number of edges between z and every vertex in A.

We claim that calling the above algorithm on the root of the level 1 tree results in a correct table for the slice of the root and that this slice is actually the entire graph. Since the level 1 root is of the form (x, x), its left and right boundaries are both equal to x. Thus, the table for this root has exactly 4 rows, and two of these rows are invalid since they attempt to include one copy of x and not the other (which is nonsensical). The two remaining rows have numbers corresponding to the maxmimum number of edges in subgraphs of size exactly k' for $k' = 0, \ldots, k$. Taking the maximum between the two numbers corresponding to when $k' = k$ gives us the size of the densest k subgraph.

A proof of correctness and analysis of the $O(k^2 8^b n)$ running time can be found in the extended version.

4 Polynomial Time Approximation Scheme and Future Work

When searching for a polynomial time approximation scheme (PTAS) for planar graph problems, one often attempts to use Baker's technique. For this technique, we assume that we have a dynamic programming solution to the given problem in b-outerplanar graphs. This technique works as follows: Given a planar graph G and a positive number ϵ, let $b = \frac{1}{\epsilon}$. Perform a breadth-first search on G to obtain a BFS tree T, and number the levels of T starting from the root, which is level 0. For each $i = 0, 1, \ldots, b-1$, let G_i be the subgraph of G induced by the vertices on the levels of T that are congruent to i modulo b. G_i is likely disconnected. Let the connected components of G_i be $G_{i,0}, G_{i,1}, \ldots$. Since each $G_{i,j}$ is $(b-1)$-outerplanar by construction, we may run the given dynamic program on each $G_{i,j}$ and combine the solutions over all j to obtain a solution S_i for the graph G_i. We then take the maximum S_i, denoted S, as our approximate solution.

We hypothesize that this technique will not work for the densest k subgraph problem on planar graphs. The reason is that by having a potentially large number of disconnected components, the approximate solution cannot be guaranteed to be within the bound given by ϵ. Suppose we have an approximate solution S for the densest k subgraph problem on some planar graph G. Note that S is the exact solution for the graph G_i for some i, meaning S does not account for any vertices on the levels of T which are congruent to i modulo b. While S could still be very dense, it is possible that most (if not all) of the edges in G are between vertices in different levels of T. This allows for the possibility that no matter which S_i is chosen as the maximum, each graph G_i is missing too many edges to closely approximate the optimal solution. For future work, it would be of great interest for one to prove that such a construction is impossible using Baker's technique.

Acknowledgements. We would like to express our sincere thanks to Samuel Chase for his collaboration on our initial explorations of finding a PTAS for the densest k subgraph problem. We would like to thank our reviewer who pointed us to previous work on this problem [3].

A Pseudocode Selections

procedure MERGE(T_1, T_2)
  let T be an initially empty table;
  let L and M be the left and right boundaries of the slice that T_1 represents;
  let M and R be the left and right boundaries of the slice that T_2 represents;
  for each subset A of $L \cup R$ **do**
    for each $k' = 0, \ldots, \max_{T_1} k' + \max_{T_2} k' - |M|$ **do**
      let V be an initially empty list;
      for each subset B of M **do**
        let $n = 0$;
        let x and y be the top level vertices in L and R, respectively;
        if $x = y$ and x and y are both in A **then**
          let $n = 1$;
        for each k_1, k_2 satisfying $k_1 + k_2 - |B| - n = k'$ **do**
          let m be the number of edges between vertices in B;
          let $v = T_1((A \cap L) \cup B, k_1) + T_2((A \cap R) \cup B, k_2) - m$;
          if v is not undefined **then**
            add v to V;
        if V is not empty **then**
          let $T(A, k') = \max_v V$;
        else
          let $T(A, k')$ be undefined;
  return T;

Fig. 6. The *merge* procedure.

procedure EXTEND(z, T)
  let T' be a table that is initialized with every entry in T;
  let L and R be the boundaries for the slice represented by T;
  for each subset A of $L \cup R$ **do**
    for each $k' = 0, \ldots, \max_T k'$ **do**
      if $T(A, k' - 1)$ is not undefined **then**
        let m be the number of edges between z and every vertex in A;
        let $T'(A \cup \{z\}, k') = T(A, k' - 1) + m$;
      else
        let $T'(A \cup \{z\}, k')$ be undefined;
  return T';

Fig. 7. The *extend* procedure.

References

1. Angel, A., Sarkas, N., Koudas, N., Srivastava, D.: Dense subgraph maintenance under streaming edge weight updates for real-time story identification. Proc. VLDB Endow. **5**(6), 574–585 (2012)
2. Baker, B.S.: Approximation algorithms for NP-complete problems on planar graphs. J. ACM **41**, 153–180 (1994)
3. Bourgeois, N., Giannakos, A., Lucarelli, G., Milis, I., Paschos, V.T.: Exact and approximation algorithms for densest k-subgraph. In: WALCOM: Algorithms and Computation, pp. 114–125. Springer, Heidelberg (2013)
4. Buehrer, G., Chellapilla, K.: A scalable pattern mining approach to web graph compression with communities. In: Proceedings of the 2008 International Conference on Web Search and Data Mining, WSDM '08, pp. 95–106. ACM, New York, NY, USA (2008)
5. Chen, J., Saad, Y.: Dense subgraph extraction with application to community detection. IEEE Trans. Knowl. Data Eng. **24**(7), 1216–1230 (2010)
6. Corneil, D.G., Perl, Y.: Clustering and domination in perfect graphs. Discret. Appl. Math. **9**(1), 27–39 (1984)
7. Du, X., Jin, R., Ding, L., Lee, V.E., Thornton Jr, J.H.: Migration motif: a spatial - temporal pattern mining approach for financial markets. In: Proceedings of the 15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD '09, pp. 1135–1144. ACM, New York, NY, USA (2009)
8. Feige, U., Kortsarz, G., Peleg, D.: The dense k-subgraph problem. Algorithmica **29**, 2001 (1999)
9. Fratkin, E., Naughton, B.T., Brutlag, D.L., Batzoglou, S.: MotifCut: regulatory motifs finding with maximum density subgraphs. Bioinformatics **22**, 150–157 (2006)
10. Gibson, D., Kleinberg, J., Raghavan, P.: Inferring web communities from link topology. In: Proceedings of the Ninth ACM Conference on Hypertext and Hypermedia : Links, Objects, Time and Space—Structure in Hypermedia Systems: Links, Objects, Time and Space—Structure in Hypermedia Systems, HYPERTEXT '98, pp. 225–234. ACM, New York, NY, USA (1998)
11. Gibson, D., Kumar, R., Tomkins, A.: Discovering large dense subgraphs in massive graphs. In: Proceedings of the 31st International Conference on Very Large Data Bases, VLDB '05, pp. 721–732. VLDB Endowment (2005)
12. Goldberg, A.V.: Finding a maximum density subgraph. Technical report, University of California at Berkeley, Berkeley, CA, USA (1984)
13. Mark Keil, J., Brecht, T.B.: The complexity of clustering in planar graphs. J. Comb. Math. Comb. Comput. **9**, 155–159 (1991)
14. Langston, M.A., Lin, L., Peng, X., Baldwin, N.E., Symons, C.T., Zhang, B., Snoddy, J.R.: A combinatorial approach to the analysis of differential gene expression data: the use of graph algorithms for disease prediction and screening. In: Methods of Microarray Data Analysis IV, pp. 223–238. Springer (2005)
15. Newman, M.E.J.: Fast algorithm for detecting community structure in networks. Phys. Rev. E **69**, 066133 (2004)