Baker's approximation scheme for planar graphs

Philip Whittington
Seminar: Algorithms on Sparse Graphs
Supervisors: Jan Dreier, Philipp Kuinke
RWTH Aachen University

CONTENTS

I. INTRODUCTION

Graphs are one of the most useful tools to formalise real-world problems. A relatively small number of NP-complete problems on graphs can offer solutions for a vast amount of use cases, which makes studying these problems so important. However, the complexity of NP-complete problems is unknown. All current approaches lead to exponential runtime, which is often impractical, but there is no proof that polynomial runtime cannot be achieved. There are many tricks to deal with this problem, each with their own advantages and disadvantages. In this paper, we will focus on four of the most common ways to do so.

A. Planar graphs

The first strategy we will consider is limiting the structure the problem is to be solved on to a subclass of graphs to derive an algorithm only working on that subclass but with better running time. Probably the most common and well-known subclass of graphs are planar graphs, which often contain sufficient structures to properly represent real-world problems. This paper aims to take a closer look at how to solve multiple NP-complete problems on planar graphs.

B. Approximation algorithms

Another way of evading the complexity of a problem is to approximate the optimal solution. Some problems are completely unsuitable for this strategy, but for many other problems algorithms with a polynomial runtime and a bound on the approximation ratio can be found. A polynomial time approximation scheme (PTAS) is an algorithm which takes the usual input for an NP-complete problem and an additional parameter $\epsilon > 0$ and computes a solution with an approximation ratio of at least $(1 - \epsilon)$, or at most $(1 + \epsilon)$ for minimization problems, in polynomial running time for fixed ϵ. A PTAS is a powerful tool, as it can adapt depending on the importance of a good approximation ratio and complexity.

C. Fixed parameter tractability

Another topic we want to take a look at is fixed parameter tractability. A problem is considered fixed parameter tractable if an algorithm exists which solves the problem in running time $f(k) \cdot n^{O(1)}$ where k is an additional parameter depending on the input, n the size of

the input, in this case the number of nodes, and f an arbitrary function depending only on k. This allows for a deeper analysis of an NP-hard problem by looking at what makes it so hard, and then refining the complexity to allow good running times for instances with a low k. This is especially useful on graphs, as many problems are often easily solved on simple graphs such as forests, and only get hard as the graph becomes more complex. We want to use k-outerplanarity as a way to evaluate the complexity of a graph.

D. Dynamic programming

Dynamic programming is a popular programming paradigm to solve problems that can be divided into subproblems. Mainly, the input is recursively decomposed into smaller instances, until a size is reached such that the solution is obvious. Using some relatively easy formula, the solutions for smaller instances are then, backtracking the recursion, merged to give solutions for their parent instances until the solution for the main instance has been computed.

E. The purpose of this paper

This paper presents a technique using the principles above to solve various NP-complete problems on planar graphs. It is based on the paper 'Approximation Algorithms for NP-Complete Problems on Planar Graphs' by Brenda S. Baker [Baker, 1994] and aims to explain its findings to make it accessible to a wider readership. When parts of the paper are quoted with only minimal or no changes, this is explicitly marked. For purposes of discussion, we will focus on the Maximum Independent Set problem, but we will give an overview how to adapt the technique to other problems in the appendix. Our first focus lies on a simple dynamic program which only works on outerplanar graphs, a relatively simple subclass of planar graphs with nice algorithmic properties. This is followed by an overview how to generalise this algorithm to all planar graphs. To not go beyond the scope of this paper, we will omit most technical details of the algorithm itself and instead focus on its main innovation, which is the decomposition of a planar graph into slices. These slices will then function as a way to decompose the graph so dynamic programming can be applied. Their size will also depend on the k-outerplanarity of a given graph and the technique therefore uses the principle of fixed parameter tractability. This will give us an algorithm to solve k-outerplanar graphs exactly with a running time linear in the number of nodes n, but exponential in k. This algorithm is then incorporated into a polynomial time approximation scheme, solving Maximum Independent Set with approximation ratio $k/k+1$ in time $O(8^k kn)$ where n is the number of nodes and k a freely chosen postive integer.

II. PRELIMINARIES

As mentioned before, we will focus on the Maximum Independent Set problem:
Given an undirected Graph $G = (V, E)$ and a positive integer i, does G contain an independent set of size at least i, that is, a subset $V' \subseteq V$ with size at least i such that no two vertices in V' are joined by an edge in E [Baker, 1994]?

For now, we need just one additional term, which is k-outerplanarity.

A. k-outerplanar graphs

Given a planar embedding E of a planar graph G, we define k-level nodes according to the faces of the embedding. Every node on the exterior face is called an exterior or level 1 node. Exterior edges are defined analogously. The technical definition of level i nodes, $i > 1$, is as follows: A cycle of level i nodes that is an interior face in the subgraph induced by all level i nodes is called a level i face. Consider the graph G_f induced by all nodes inside a level i face. The nodes on the exterior edge of G_f are then called level $i+1$ nodes.
This definition becomes more accessible when looking at an algorithm to compute the level of all nodes. In each step, starting at 1, remove all nodes on the exterior face. Repeat until there are no nodes left. The level of a node is the step in which it was removed. Using an appropriate data structure for the planar embedding, such as the one used by Lipton and Tarjan [Lipton and Tarjan, 1979], this can be done in linear time.
A planar embedding is called k-outerplanar if there are no nodes of level $> k$, which means the algorithm terminates after at most k steps. A planar graph is called k-outerplanar if it has a k-outerplanar embedding. If a graph is 1-outerplanar, we will simply call it outerplanar. The subclass of outerplanar graphs will be of importance to us, as it has some properties which make it easy to develop algorithms solving various problems on this subclass. See Figure 1 for a 3-outerplanar graph. The nodes A to G are of level 1, the nodes a to g are of level 2 and the nodes 1 to 8 are of level 3. Notice that although the nodes 1 to 5 and 6 to 8 are not enclosed by the same face and therefore not connected, they're still of the same level.

III. AN APPROXIMATION ALGORITHM FOR PLANAR GRAPHS

We will use this new notion of k-outerplanarity to construct a polynomial time approximation scheme for maximum independent set on planar graphs, which can be easily adapted to other NP-complete problems. The goal is an algorithm that, for a freely chosen but fixed k, solves

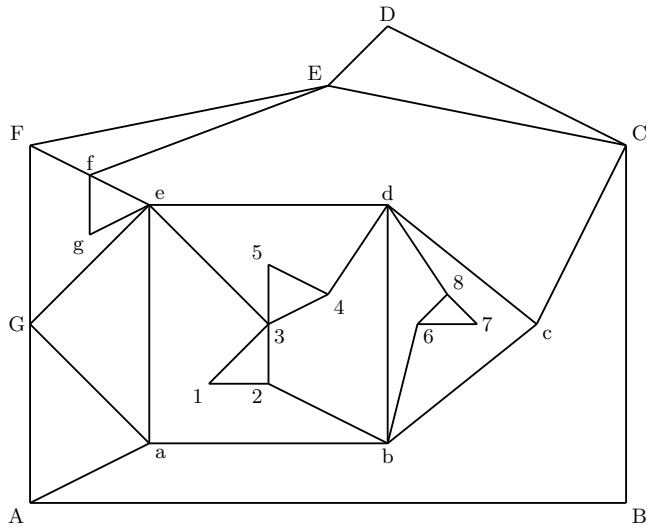

Figure 1. A planar embedding of a 3-outerplanar graph [Baker, 1994].

maximum independent set with an approximation ratio of $k/k+1$ and a linear complexity with respect to the number of nodes n.
Basically, we want to exactly solve maximum independent on disjunct subgraphs that are k-outerplanar and ignore some nodes in the graph to merge the subgraphs without violating the conditions of the maximum independent set. The first part of this sentence is formalised in this theorem, which we will prove later:

Theorem 1 [Baker, 1994] *Let k be a positive integer. Given a k-outerplanar embedding of a k-outerplanar graph G, an optimal solution for maximum independent set can be obtained in time $O(8^k n)$, where n is the number of nodes.*

Given a positive integer k and a planar graph G, the algorithm works as follows:

1. Generate a planar embedding of G using the linear-time algorithm of Hopcroft and Tarjan [Hopcroft and Tarjan, 1974] and compute the level of every node.
2. For each $i, 0 \leq i \leq k$, do the following:
   (a) Remove all nodes where the nodes level mod $(k+1)$ equals i, splitting the graph into components with an (already computed) k-outerplanar embedding.
   (b) Use Theorem 1 to exactly solve maximum independent set on all those components and take the union of the solutions. This is possible as the components are disjunct and no edges can exist between two components.
3. Take the best solution for all i as the solution for the entire graph.

To see that this gives a solution with approximation ratio $k/k+1$, consider the optimal solution S_{OPT}. For some i as defined above, at most $1/k+1$ of the nodes in S_{OPT} are at a level congruent to i. As the other components are solved exactly, the union of their solutions has at least the size of S_{OPT} minus those at most $1/k+1 \cdot |S_{OPT}|$ nodes, leading to the desired approximation ratio. This leads to the second theorem:

Theorem 2 [Baker, 1994] *For fixed k, there is an $O(8^k kn)$-time algorithm for maximum independent set that achieves a solution of size at least $k/k+1$ optimal for general planar graphs. Choosing $k = \lceil c \log\log n \rceil$, where c is a constant, yields an approximation algorithm that runs in time $O(n(\log n)^{3c} \log\log n)$ and achieves a solution of size at least $\lceil c \log\log n \rceil / (1+\lceil c \log\log n \rceil)$ optimal. In each case, n is the number of nodes in the graph.*

Theorem 2 formalises exactly what this paper aims to prove. To prove it correct, we need to prove Theorem 1.

IV. AN EXACT ALGORITHM FOR OUTERPLANAR GRAPHS

First, we will just consider the special case of outerplanar graphs. Although these are relatively easy to solve, the procedure to solve k-outerplanar graphs is quite similar. For that reason, we will analyse this algorithm quite in depth while keeping in mind how to adapt any step to solve k-outerplanar graphs.
The algorithm is based on the principle of dynamic programming. By recursively merging solutions for subgraphs of G, we construct an optimal solution for the whole graph. To keep complexity low, we need to put some effort into the choice of subgraphs, so that each merger takes constant time.

A. Constructing the tree $\overline{G}$ representing G

To understand how we need to choose the subgraphs, we first have to find an easy way to merge optimal solutions for two graphs. Notice that an optimal solution for the subgraph is not necessarily subset of an optimal solution for the merged subgraph or the entire graph. This is because nodes of the subgraph adjacent to nodes outside the subgraph are additionally restrained. For each such node we need to consider both cases, the node is part of the solution or it is not. The number of cases is exponential to the number of such nodes, so we want to keep those at a low constant. Those nodes are called boundary nodes, as their assignment clearly induces the optimal assignment of all other nodes of the subgraph.
There is a relatively easy decomposition of any outerplanar graph G such that every subgraph has two boundary nodes. One way to see this is noticing that any connected outerplanar graph (with at least four nodes) can be divided into two components by removing two

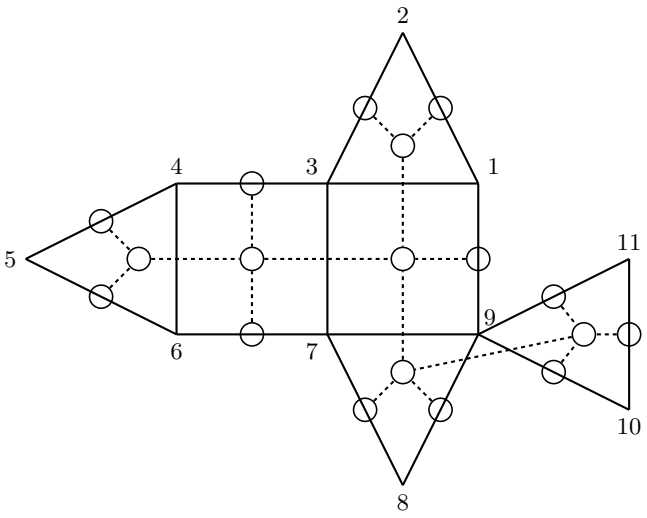

Figure 2. A tree for an outerplanar graph [Baker, 1994].

nodes. It is based on the idea of going a counterclockwise walk around the exterior edges of G and then taking shortcuts. The decomposition can be visualized by a tree $\overline{G}$, which we will now construct.

1. Remove each bridge, that is, an edge whose removal disconnects the graph, by adding a second edge between the same nodes, making it a face.
2. Construct a graph vertex for every interior face and every exterior edge.
3. Draw edges between every edge vertex and the vertex of the face the edge is contained in, and between every two vertices of faces that share an edge.
4. There might be nodes that need special attention, whose removal disconnects the graph. Call those nodes cutpoints. As we removed bridges, a cutpoint is a node where at least two faces meet without sharing an edge, therefore disconnecting the tree for G. Draw an edge between two vertices of faces that both contain the cutpoint and are part of different components until $\overline{G}$ is connected.

See Figure 2 for an example of a graph and its tree. Note that node 9 is a cutpoint. The connection to the face vertex for the triangle $7-8-9$ was chosen arbitrarily; the square $1-3-7-9$ would have proven equally fit.

B. Labelling $\overline{G}$ to represent a walk around G

For now, we have constructed a tree representing the structure of G. In order to extract an ordering of subgraphs and merges, we need to give this tree an ordering by choosing a root and therefore give direction to the tree. As the optimal solutions for edges are basically

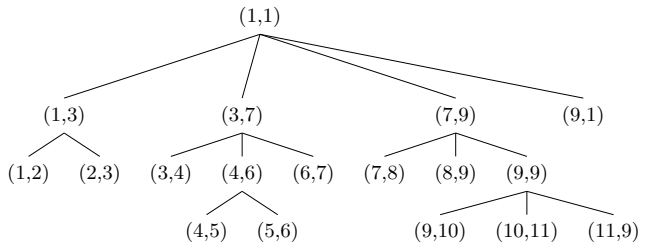

Figure 3. The tree from Figure 2 ordered and labelled [Baker, 1994].

given by definition and any edge can trivially be bound by two nodes as it only contains those two, it makes sense to choose all edge vertices as the leaves of $\overline{G}$. Choose any face vertex to be the root of the tree, any adjacent vertex to be the first child. The ordering of any face vertices' children is now unambigously induced by a counterclockwise walk on the edges of the face.

To better understand how this tree induces an ordering of subgraphs, we label the vertices. Each label is a two-tuple of two not necessarily different nodes from G, which are the boundaries of the subgraph represented by the vertex. This subgraph consists of all nodes on a counterclockwise walk from the first to the second boundary node, where the first boundary node is the first of the two nodes to appear in a counterclockwise walk from the right node. The leaves of the subtree of any vertex are exactly the edges on this walk. The inner vertices represent the inner edges (or cutpoints) that act as shortcuts on this walk. Every time a solution is merged, such a shortcut is taken, symbolizing that all nodes that are not part of the walk anymore have been dealt with.

First, label each edge vertex with the two endpoints of the edge. Then label each face vertex with the left label of its first and the right label of its last child.

See Figure 3 for the labeled tree of the graph G, where the face vertex for the square $1-3-7-9$ was chosen as the root, and the face vertex for the triangle $1-2-3$ as the first child, which means the walk starts at the node 1. The vertex $(9,9)$ represents the cutpoint at node 9.

C. The dynamic program computing the optimal solution

The approach of computing the optimal solution is now clearly given by the tree $\overline{G}$, so we will now focus on the technical details. We will use dynamic programming on the tree to construct a table with four entries for every tree vertex. We need four entries as we need to encode all combinations for the two boundary nodes whether they are in the maximum independent set or not. Those will be adressed by two bits for the two boundary nodes. Note that at least one entry will always be undefined, as the two boundary nodes will always be either adjacent, so that it is impossible for

both to be in the set, or the same node, so that it is impossible for just one of them to be in the set. The other entries will contain the size of the optimal solution for this assignment of the boundary nodes in the current subgraph. In particular, the two defined entries on the root, which is always labelled (x, x), contain the sizes of optimal solutions for the entire graph depending on whether x ist part of the set.

```
table(v)
  if v is a level 1 leaf with label (x,y)
    return a table representing (x,y)
  else T=table(u), where u is the leftmost
     child of v
    for each other child c of v from left to
       right
      T= merge(T,table(c))
    return adjust(T)
```

Again, we start by computing tables for the edges, as these are the base cases of our recursion. The entries for any edge will be $0 - 1 - 1 - undefined$. Similar to a post-order traversal, the values for all face vertices will then be computed by backtracking the recursion. The tables of all children are merged to form the new table for the parent vertex, whose boundaries are, as per definition of its label, the left boundary of the first and the right boundary of the last child. It also holds that the right boundary of a child is equal to the left boundary of its successor (if it exists) so we can always perform a merge.

The procedure `merge` itself works as follows: For two already computed tables T_1 and T_2 of solutions bound by (x, y) and (y, z), therefore sharing a boundary, construct a table for (x, z) by computing a value for every combination of assignments of x and z. To do so, take the maximum over $T_1(x, 0) + T_2(0, z)$ (the new solution does not use y) and $T_1(x, 1) + T_2(1, z) - 1$ (the new solution does use y). Note that you need to subtract 1 in the second formula to avoid counting y twice, as it is part of both solutions. This can be generalised to taking the maximum over all possible assignments of y, which currently are only 0 and 1, of the formula $T_1(x, y) + T_2(y, z) - |y|_1$ to generate the entry $T(x, z)$ for some boundaries x, z. $|y|_1$ encodes the number of ones in y, which again are currently only 0 and 1.

When this has been done for all children, we have computed a table for the boundary of the parent vertex. As the two boundary nodes of the parent vertex must be adjacent or the same node, we need the procedure `adjust`. It takes care of setting some entries to undefined due to the solution being incorrect otherwise. Calling `table` on the root of $\overline{G}$ results in a table with entries for $(0, 0)$ and $(1, 1)$, and the larger of these values is the size of the maximum independent set of G. By backtracking the decisions made during each merge, the solution itself can be found easily.

D. Conclusion and adaptability

Let us summarize the algorithm we constructed so far: Using dynamic programming, we recursively computed solutions for growing subgraphs until we achieved the solution for the entire graph. To do so, we defined the tree $\overline{G}$ to an outerplanar graph G, which gives the exact order of merge operations. Then we computed tables for every vertex of the tree starting at the leaves, which simply encoded some edges.

We found that it is possible to bind any subgraph using just two boundary nodes when it comes to outerplanar graphs. This is the only prerequisite for our algorithm that does not hold for k-outerplanar graphs, which means we need to change our choice of subgraphs. Changing the subgraphs will of course also lead to changes in the technical details of the dynamic programming. However, the main idea will stay the same, so we will not go into its details. Instead, we want to focus on the main difference, the subgraphs and their boundaries. Until now, the subgraphs simply consisted of edges, faces, or unions over faces such that there was only one face adjacent to some other face that was not part of the subgraph. Now, this needs to become more involved, which is why we will look at the concept of slices.

V. ON THE NOTION OF SLICES

A. The basic idea

Let us take a look at the properties of k-outerplanar graphs that might come in useful when defining a new type of subgraph. For one, they are planar, so choosing a path between two exterior nodes as a boundary will always split the graph in two. Of course, to keep complexity low, we want to choose two nodes so that the length of the path between them is as short as possible, at best somewhat bounded. That is essentially what we did with outerplanar graphs, but finding exterior nodes with short paths between them was really easy. Now we must make do with the looser restriction of a k-outerplanar graph.

Consider some node of level i. Although a path from this node to an exterior node may be arbitrarily long, we can always find $i - 1$ nodes, one of every level from 1 to i such that a path using exactly those nodes would not hurt the planarity constraint. This requires additional edges, which will be used in the construction of the slices, but not in the computation of the optimal solution. By combining the paths of two nodes of level i that share an edge or are the same, we get a boundary of $2i$ nodes, where the left boundary consists of the path from some exterior edge to the first node and the right boundary consists of the path from some exterior node to the second node. This means we will now think of boundaries as two vectors of nodes rather than just two nodes.

To better understand this idea, consider this analogy:

Think of the entire graph as a pie, and you want to cut out a slice of some size. What you would do is make a first cut pointing to the middle and not longer than the pie's radius. Then you'd make a second cut ending exactly where your first cut ended and again not longer than the pie's radius, leading to a slice. This is exactly what we are doing here, with the left boundary representing your first and the right boundary representing your second cut.

B. Constructing trees for k-outerplanar graphs

We will again make use of our concept of trees describing the structure of the original graph. Each level i component seperated is outerplanar, so we use the same strategy to construct a tree representing a counterclockwise walk around the exterior edges of the component. This of course generates a seperate tree for each component, which will not be linked, but constructed depending on the trees for the enclosing component. The construction of the slices will then make sure that the subgraphs are correctly merged.
The only difference is the choice of the root and leftmost child for the trees of the level i components, which depends on the root of the enclosing component. Consider a level i component C enclosed by a level $i-1$ face f labelled (x, y). Note that the labels are still chosen in respect to the walk on the exterior edges of the component and therefore still consist of two nodes, not two vectors of nodes like the boundaries, although we use (x, y) for both. By scanning their nodes in parallel, construct a triangulation between those two in linear time. See Figure 4 for such a triangulation. The root (z, z) for $\overline{C}$ is chosen depending on (x, y). If $x = y$, then z is any node adjacent to x, otherwise it is the node adjacent to both x and y in the triangulation. The leftmost child is then the first edge counterclockwise from (z, x), which unambiguosly defines the rest of the tree as done before.

C. The formal definition

1. A first approach

Let us go into the technical details of slices. For every vertex of every level i tree we constructed as in the subsection above, we want to build a slice with two boundary vectors containing i boundary nodes each. Note that the slice can include nodes of higher level, but as they will be surrounded by some face in our slice, we do not need seperate boundary nodes for these. In particular, every slice for a face will always contain all nodes enclosed by the face, and the slices for these are computed by recursing on the root of the tree of the enclosed component.
Slices will be defined recursively. Again, we will follow the order given by the trees in the construction. The recursion starts at the root of the level 1 component, as

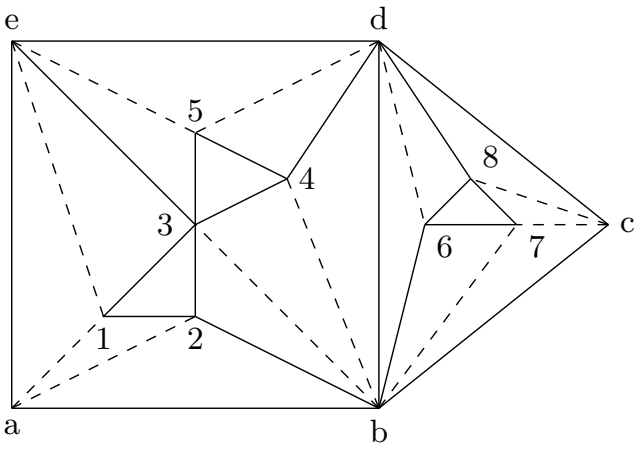

Figure 4. Triangulating regions between levels 2 and 3 for the graph of Figure 1. Edges added in the triangulation are shown as dashed lines [Baker, 1994].

the dynamic program will also start there, and ends only at the level 1 leaves. As noted before, the transition to a higher level will be taken care of by the slices for vertices of enclosing faces. However, as the recursion does only end on the level 1 leaves, we also need a way to get back from higher to lower levels. This does also make sense considering that we want the boundaries to run from level 1 to i, so we do always need nodes from all the lower levels in every slice. Considering this keyword ‚always', it makes sense to integrate this connection into the leafs, as their slices will be part of the nodes for inner slices. These thoughts lead to the following informal definition: Let v be a tree vertex labeled (x, y). Remember that tree vertices are not labeled with vectors, but two single nodes.

1. If v represents a level i face with no enclosed nodes, $i \geq 1$, its slice is the union of the slices of its children, plus (x, y).

2. If v represents a level i face enclosing a level $i+1$ component C, its slice is that of the root of the tree for C plus (x, y). (However, as noted above, the boundaries only run from level i to level 1 instead of from level $i+1$ to 1.)

3. If v represents a level 1 edge, its slice is the subgraph consisting of (x, y).

4. if v represents a level i edge, $i > 1$, then its slice includes (x, y), edges from x and y to level $i-1$ nodes, and the slices computed recursively for appropriate level $i-1$ trees. Here, ‚appropriate' is determined by slice boundaries placed along edges in a triangulation of the region between level $i-1$ and i [Baker, 1994].

Points 1 and 3 are pretty intuitive and would already be enough to derive the strategy for outerplanar graphs, so

we will look at the peculiar points 2 and 4. The second point states that the slice for a face enclosing some component includes the slice of said component, which is, considering our forethoughts, no surprise. However, it does not include the slices of the children of the tree vertex. Instead, the slices for the leaves of the tree of the enclosed take care of these, and we will have to take a closer look at which leaves take care of which children. This is done so the slices always contain nodes from all levels up to the level of the slice, as we mentioned before. So how exactly do we assign the which leaves contain the slices of which children? It all depends on the boundaries implied by the assignment. We want to choose the boundaries in a way so that leaves can easily be merged, so the right boundary of a leaf should be equal to the left boundary of the next one. Also, all children must be considered and there may be no edges crossing slice boundaries.

2. Computing slices across levels

We can assume by induction that using the left boundary of some level $i-1$ vertex and the right boundary of some other level i vertex, which is to the right of the first vertex considering the tree from the level i component or simply the same vertex, yields a correct boundary if the hole between their boundaries is properly filled. Therefore, we will produce a function assigning two level $i-1$ vertices to each level i vertex which fills said hole in the boundary. The triangulation used for defining the trees will be recycled here, as it gives us an idea which pairs of nodes would be suitable partners in a boundary.

For this function, we first need to define dividing points. Let C be a level i component enclosed by a level $i-1$ face f whose tree vertex is labeled (x, y). Let $TRI(C, f)$ be the triangulation of the region between C and f already constructed in defining the trees. For any pair of successive edges $(x_1, x_2), (x_2, x_3)$ in a counterclockwise walk around the exterior edges of C, there is at least one node y of f that is adjacent to x_2 in $TRI(C, f)$ such that the edges $(x_2, x_1), (x_2, y), (x_2, x_3)$ occur in counterclockwise order around x_2. Call such a node a dividing point for (x_1, x_2) and (x_2, x_3)) [Baker, 1994].

Basically, every node of the face adjacent to x_2 is a dividing point for the two exterior edges including x_2, except when there is a cutpoint. A cutpoint has more than two exterior edges, so we can't simply assign nodes to their dividing points for the boundary and need the more complicated definition above. Again, see Figure 4 for examples of dividing points. a and b are dividing points for the edges $(1, 2)$ and $(2, 3)$. e is a cutpoint for the edges $(5, 3)$ and $(3, 1)$, but b is not as 3 is a cutpoint.

We will now define the functions assigning boundaries to all level i vertices, which we will call LB and RB. These functions will map the vertices of $\overline{C}$ on the numbers 1 to r where r is the number of children of the tree vertex corresponding to f. The definition is as follows:

1. Let the leaves of $\overline{C}$ be $v_1, v_2, ..., v_t$ from left to right, and let v_j have label (x_j, x_{j+1}), for $1 \leq j \leq t$. Let the children of $vertex(f)$ be $z_1, z_2, ..., z_r$ from left to right, where z_j has labeled (y_j, y_{j+1}) for $1 \leq j \leq r$. Define $LB(v_1) = 1$ and $RB(v_t) = r + 1$. For $1 < j \leq t$, define $LB(v_j) = q$ if q is the least $p \geq LB(v_{j-1})$ for which y_p is a dividing point for (x_{j-1}, x_j) and $x_j, x_{j+1})$. For $1 \leq j < t$, define $RB(v_j) = LB(v_{j+1})$.

2. If v is a face vertex of $\overline{C}$, with leftmost child c_L and rightmost child c_R, define $LB(v) = LB(c_L)$ and $RB(v) = RB(c_R)$ [Baker, 1994].

Remember that we want to do a counterclockwise walk around the edges of f, but right now we do not want to consider the edge corresponding to the faces label as its slice has already been taken care of, so we make sure to start exactly to the right of it and end exactly to the left by setting the values for $LB(v_1)$ and $RB(v_t)$ accordingly. To each leaf, a value is assigned which represents the leftmost dividing point, with the additional condition that the value is not lower than the value of the left neighbour leaf, which is needed to ensure that the boundaries do not include the face represented by the parent vertex. Finally, for all nodes except the last, RB is simply set to be equal to the successors LB so that the boundaries match and we can later perform merges in the dynamic programming. This definition achieves two key points. First, the values of LB and RB are non-decreasing for sibling vertices from left to right, which makes sure that the assignment will not build slices with edges crossing the boundary. Second, for each vertex v, it holds that $LB(v) \leq RB(b)$ so that the boundaries for a single vertex are correct. The exact translation for a vertex v labeled (x, y) from values for LB and RB to the boundaries works as follows:

1. If v is a level 1 leaf, the left boundary of $slice(v)$ is x and the right boundary is y.

2. Suppose v is a vertex of a tree for a level i component enclosed by the level $i-1$ face f. Let s be the number of children of $vertex(f)$. In the following, define the right boundary of the 0th child of $vertex(f)$ to be same as the left boundary of the first child of $vertex(f)$, and the left boundary of the $(s+1)$th child to be the same as the right boundary of the sth child. If $LB(v) = q$, the left boundary of $slice(v)$ is x plus the left boundary of the slice of the qth child of $vertex(f)$. If $RB(v) = t$, the right boundary of $slice(v)$ is y plus the right boundary of the slice of the $t-1$th child of $vertex(f)$ [Baker, 1994].

3. The final definition

This gives us the formal definition of slices:
Let v be a level i vertex, $i \geq 1$, with label (x, y). Again, if a vertex u has s children, define the left boundary of the $(s+1)$th child to be the same as the right boundary of the sth child.

1. If v is a face vertex, and $face(v)$ encloses no level $i+1$ nodes, then $slice(v)$ is the union of the slices of the children of v, together with (x, y) if (x, y) is an edge (i.e., $x \neq y$).

2. If v is a face vertex and $face(v)$ encloses a level $i+1$ component, then $slice(v)$ is the subgraph containing $slice(root(\overline{C}))$ plus (x, y) if (x, y) is an edge.

3. If v is a level 1 leaf, then $slice(v)$ is the subgraph consisting of (x, y).

4. Suppose v is a level i leaf, $i > 1$. Suppose the enclosing face is f, and $vertex(f)$ has children $u_j, 1 \leq j \leq t$, where u_j has label (z_j, z_{j+1}). If $LB(v) \neq RB(v)$, then $slice(v)$ is the subgraph containing (x, y), any edges from x or y to z_j, for $LB(v) \leq j \leq RB(v)$, and $slice(u_j)$, for $LB(v) \leq j \leq RB(v)$. If $LB(v) = RB(v) = r$, then $slice(v)$ is the subgraph containing (x, y), any edges from x or y to z_r, the left boundary of $slice(u_r)$, and any edges between boundary nodes of successive levels `[Baker, 1994]`.

This definition encorporates all of our thoughts collected before. See Figure 5 for the slices of the level 3 vertices $(6, 7), (7, 8), (8, 6)$ in Figure 1. As you can see, the boundaries align so that the slices can be merged easily. For example, the slice of the vertex labelled (b, d), which represents the face enclosing the level 3 component, can now easily be computed by merging on the right boundary of $(6, 7)$, which is $7, b, A$ and equal to the left boundary of $(7, 8)$, and merging the resulting slice with the slice for $(8, 6)$, then finally adding the edge (b, d).

D. An overview of the dynamic program

With slices now fully defined, we can start adapting the rest of the algorithm, which mainly breaks down to the dynamic program. Again, we want to compute tables for subgraphs, in this case, smaller slices, and then merge the solutions. The tables will contain the size of the optimal solution for the given assignment of the boundary nodes. As a slice is being constructed according to the definition above, its table will be computed at the same time. This means the order of merges is clearly given by the construction rules of slices.
The dynamic program also uses other subroutines, for example a modified `adjust`, to deal with single edges being incorporated into the graph or changing the boundaries to the right size so merges can be performed, but is still

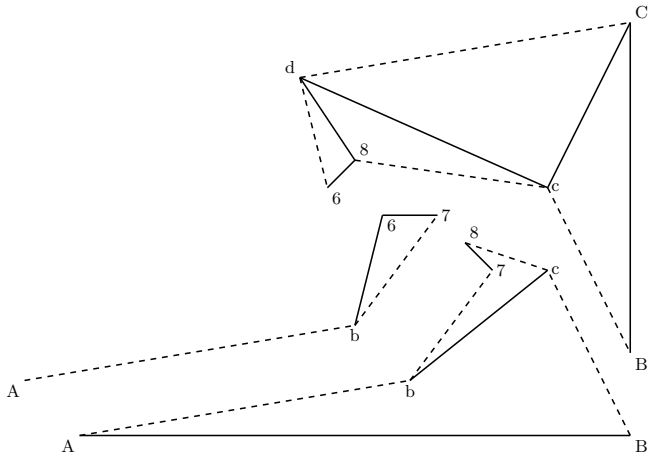

Figure 5. Three slices for the graph of Figure 1. Missing edges between boundary nodes are shown as dashed lines `[Baker, 1994]`.

based on the `merge` routine. For that reason, we will now take a deeper look at `merge`.
As before, `merge` merges two tables T_1 and T_2 for two subgraphs, now level i slices S_1 and S_2, which share a boundary y. Let the boundary of S_1 be (x, y) and the boundary of S_2 be (y, z). The resulting table corresponds to a slice with boundary (x, z). We can again use the formula $T_1(x, y)+T_2(y, z)-|y|_1$ used for outerplanar graphs. For every combination of x and z, the maximum over all assignments of y must be computed. Remember that x,y and z are vectors containing i nodes, so that there are 2^i possible assignments for each of them. Again, the number of nodes that are part of the optimal solution and the merged boundary y must be subtracted to avoid being counted twice.
This operation also dominates the running time of the algorithm. A table for each combination of x and z must be computed, which leads to $2^i \cdot 2^i$ tables. Each computation must consider 2^i possible assignments of y. For a k-outerplanar graph, i is bounded from above by k as i is the level of the considered vertex. A merge is called at most once for every face vertex, as its face or edge is then incorporated into the already solved part of the graph and does not need to be incorporated again. The number of vertices is linear in the number of edges, which is, in planar graphs, linear in the number of nodes. This leads to a total running time of $O(2^k \cdot 2^k \cdot 2^k \cdot n) = O(8^k n)$.

VI. ADAPTING THE ALGORITHM TO SOLVE OTHER PROBLEMS

This section focuses on how to adapt the final algorithm to solve other NP-complete problems exactly. We will focus on three well-known problems, namely Minimum Vertex Cover, Minimum Edge Dominating Set, and Three-Coloring. However, a number of additional problems can also be solved using this technique, but we will

only consider those mentioned above as they are sufficient to give an idea how to adapt the algorithm. When applicable, we will also show how to adapt the approximation algorithm.
Generally speaking, the same decomposition into slices can be used for exactly solving every of these problems, which makes this technique quite powerful as it is easily adaptable. The only difference lies in the tables used in the dynamic programming. For Maximum Independent Set, the highest possible number of nodes was computed for a slice, which now has to change. This also implies that the `merge` operation needs a slight adaptation aswell, but it will still work on slices sharing a boundary and compute new solutions for every combination of boundaries. The boundary nodes will stay the same, however their interpretation might also need modification. Before, we encoded if a boundary node was part of the Maximum Independent Set, which does not work when considering a problem which is not based on nodes or uses more than two possible assignments to a node.

A. Minimum Vertex Cover

We will start with Minimum Vertex Cover, as it is a very similar problem:
Given a graph $G = (V, E)$ and a positive integer $K \leq |V|$, is there a vertex cover of size K or less for G, that is, a subset $V' \subseteq V$ with $|V'| \leq K$ such that for each edge $(u, v) \in E$ at least one of u and v belongs to V' [Garey and Johnson, 1979]? We only need to change the details in the bookkeeping. Each table entry contains the size of the Minimum Vertex Cover for the given slice, and the boundary nodes are assigned whether or not they are part of the Minimum Vertex Cover. The `merge` operation must also be adapted to compute a minimal instead of a maximal value. The approximation works a bit differently. The graph is again split into subgraphs of a chosen outerplanarity, but this time the edge levels are considered twice instead of being left out. Formally, this works as follows. For each $i, 0 \leq i < k$, do the following:
Take the graph of all nodes of level $jk + i$ to $(j + 1)k + i$ and compute the optimal solution for all $j \geq 0$, then take the union of these solutions as the final solution.
For at least one i this gives a solution with an approximation ratio of at most $(k+1)/k$. The reasoning is again similar to Maximum Independent Set. For some i, all levels congruent to i will contain at most $|S|/k$ nodes of an optimal solution S. The instance using these levels as edge levels counts at most$|S|/k$ nodes twice and solves the rest of the graph optimally, leading to the desired approximation ratio.

B. Minimum Edge Dominating Set

A problem which also changes the assignments to the boundaries, but is otherwise quite similar especially to Minimum Vertex Cover is Minimum Edge Dominating Set:
Given a graph $G = (V, E)$ and a positive integer $K \leq |V|$, is there a set $E' \subseteq E$ of K or fewer edges such that every edge in E shares at least one endpoint with some edge in E' [Garey and Johnson, 1979]? This problem now focuses on minimising a number of edges, so the tables will encode the size of the Minimum Edge Dominating Set for the corresponding slice, subject to which boundary nodes are endpoints of edges in that set. The subroutines also need to be slighty adapted to cover edges instead of nodes.
The approximation algorithm works similar to the algorithm for Minimum Vertex Cover, as the graph is split into the same subgraphs.

C. Three-Coloring

Lastly, we will look at the Three-Coloring problem, as it shows major differences to the other problems and can't be approximated:
Given a graph $G = (V, E)$, is G 3-colorable, that is, can the nodes be colored with three different colors such that adjacent nodes are always assigned different colors [Garey and Johnson, 1979]? Coloring a node with three colors is an assignment with three different states for each node, instead of the two possible assignments (yes/no) we used before. So, for a boundary of size k, all combinations of the three possible assignments must be considered, which leads to 3^{2k} table entries. The table entries encode whether, given the assignment of the boundary nodes, the slice can be three-colored. A merger then takes time $O(3^{3k})$.
Approximating this problem in the way we've done before is not possible, as we can't just leave out nodes or consider them twice. It also makes no sense to approximate the chromatic number, as planar graphs are always four-colorable. Therefore we are not able to construct a PTAS, but a way to solve Three-Coloring on planar graphs with low k-outerplanarity is still found.

VII. CONCLUSION

A. Summary

We have taken a look at various NP-complete problems on graphs and described an algorithm solving them exactly. By constructing trees describing the structure of a k-outerplanar graph and building slices for the tree vertices, we have found a decomposition of the graph suitable for dynamic programming. Slices are used to partition the graph. Their main feature is their boundary, whose size is bounded by the level of the vertex the slice belongs to. This greatly improves complexity, as the decomposition is the most runtime consuming part of the algorithm. By changing details in the dynamic

program, we are able to adapt the algorithm for different NP-complete problems while using the same decomposition.
When the problems proved suitable for approximation, we were also able to use the given algorithm in a polynomial time approximation scheme splitting a given graph into k-outerplanar graphs, solving these exactly and merging the solutions with an error of only $1/k$.

B. Evaluation

We again focus on Maximum Independent Set for purposes of discussion. The algorithm by Brenda S. Baker gives, for fixed k, an exact solution in linear time. In general, a running time of $O(8^k n)$ is obtained. Note that for graphs of outerplanarity $k \in O(log(n))$, this defines a class of planar graphs for which Maximum Independent Set is solvable in polynomial time. While the complexity is quite impressive, the algorithm itself is rather complicated, which is its major weakness. The construction of a single slice can occur according to four different types of tree vertices, and often involves other slices. In particular, slices can depend on other slices of the same level, one level above or one level below, which leads to a recursion switching levels more than $2k$ times. Newer algorithms, for example such based on treewidth, are much simpler, but still achieve the same result [Bodlaender and Koster, 2008]. Treewidth is quite closely related to k-outerplanarity, as both notions characterize the structure of the graph. However, an algorithm based on trees rather than conjoined cyles is predestined to be more straightforward and have less edge cases. The k-outerplanarity of a graph is also bound by its treewidth [Bodlaender, 1998], again showing how closely the two topics are related.
Another notable feature of Baker's algorithm is its usability in a polynomial time approximation scheme with an approximation ratio of scheme to converge towards optimal while not losing its polynomial complexity for increasing n. As treewidth and k-outerplanarity are related, the same approximation can also be used for exact algorithms based on treewidth [Bodlaender and Koster, 2008]. Finally it should be mentioned that Baker's algorithm can easily be adapted to deal with other problems than Maximum Independent Set as seen in Section VI.
Overall, Baker invented a very powerful algorithm offering broad applicabilty, low running times and good results, but also a quite complicated construction.

VIII. REFERENCES

BAKER, BRENDA S. 1994. Appoximation Algorithms for NP-Complete Problems on Planar Graphs, *J. ACM 41, January 1994, pp 153-180*

LIPTON, R.J. AND TARJAN, R.E. 1979. A seperator theorem for planar graphs, *SLAM J. Appl. Math. 36, 2, 177-189.*

HOPCROFT, J. AND TARJAN, R. 1974. Efficient planarity testing, *J. ACM 21, pp 549-568*

GAREY, M.R. AND JOHNSON, D.S. 1979. Computers and Intractability, A Guide To the Theory of NP-Completeness, *W.H. Freeman and Co., San Francisco, Calif.*

BODLAENDER, HANS L. AND KOSTER, ARIE M. C. A. 2008. Combinatorial Optimization on Graphs of Bounded Treewidth *Comput. J. 51, May 2008, pp 255-269*

BODLAENDER, HANS L. 1998. A partial k-arboretum of graphs with bounded treewidth *J. TCS 209, December 1998, pp 1-45*