Implementation of a minimum branch-decomposition and branch-width algorithm for planar graphs.

Kristoffer Højelse (krbh)

2 Sep 2024

[1]

[1]Illustration by Dall-E via ChatGPT (GPT-4o) with the prompt "Generate an image of pied piper the rat-catcher along with a computer science graph".

Abstract

Seymour and Thomas give an algorithm, the rat-catching algorithm, for deciding $bw(G) \leq c$ in $O(n^2)$ time, and by using it as a subroutine, an algorithm to compute an optimal branch-decomposition in $O(n^4)$ time. In this paper, I describe an implementation of this algorithm and publish the source code.

1 Introduction

In algorithmic complexity theory, an NP-hard problem has no known algorithm with all 3 of these properties: (1) runs in polynomial time, (2) gives an exact solution, and (3) runs on all instances of the problem.

You can find brute-force algorithms for NP-hard problems that both give an exact solution and run on all instances of the problem, but they run in exponential time.

You can find approximation algorithms for NP-hard problems that both run in polynomial time and run on all instances, but they give an approximate solution.

Finally, you can find parameterized algorithms for NP-hard problems that both run in polynomial time and give an exact solution, but they only run on instances of the problem where some parameter is small. Treewidth is a common such parameter.

Treewidth is a measure of how tree-like a graph is. A graph with a treewidth of 1 is a tree. Larger values are less tree-like.

For some NP-hard problems, if a graph has a small enough treewidth, then a parameterized algorithm can find an exact solution in polynomial time.

Branch width and carving width are related to treewidth, and some NP-hard problems can be solved efficiently for graphs of small branch width.

The treewidth of a graph is at least $b - 1$ and at most $\lfloor \frac{3}{2} b \rfloor - 1$, where b is the branch width.[6]

The carving width is at least $\frac{b}{2}$ and is at most $\Delta(G) \cdot b$, where b is the branch width.[3]

A decomposition is generally a derivative structure that encapsulates some aspect of a graph. Tree-, branch- and carving-width all have related decompositions. These decompositions can be used in conjunction with dynamic programming to design efficient parameterized algorithms. For instance, Pino[2] applies branch decompositions on some NP-hard connectivity problems.

It is NP-complete to determine whether a graph G has a branch-decomposition of width at most k, when G and k are both considered as inputs to the problem.[1]

Informally, a planar graph is a graph that you can draw on a plane surface without any edges crossing. There might be multiple such drawings of a planar graph. A particular such drawing is called a plane graph. A plane graph can be said to divide the plane into regions called *faces*.

The algorithm by Seymour and Thomas[1] computes a minimum branch-decomposition of a planar graph in $O(n^4)$ time.

Bian, Gu, and Zhu[5] describe and benchmark some implementations.

This paper describes an implementation of the algorithm and publishes the source code for anyone to examine, use, and modify.

2 Preliminaries

This section defines the terminology and concepts used in the algorithm.

The algorithm deals with graphs that might have parallel edges. Distinguishing between two parallel edges is crucial. Therefore, the implementation gives labels to both vertices and edges, by encoding a graph as an adjacency list of edge IDs and a map from unique edge IDs to its vertex-pair.

A *graph* G consists of a vertex set $V(G)$, and an edge set $\mathbb{E}(G)$ and a relation ϕ_G, where $V(G) \subset \mathbb{N}^+$ and where $\mathbb{E}(G) \subset \mathbb{N}^+$ and where $\phi_G \colon \mathbb{E}(G) \to \{\{u, v\} \colon u, v \in V(G)\}$.

Note. Normally *graph* refers to a simple graph, usually also without labels, which are normally encoded by an adjacency list or matrix. In this paper, a graph is a multigraph with labels.

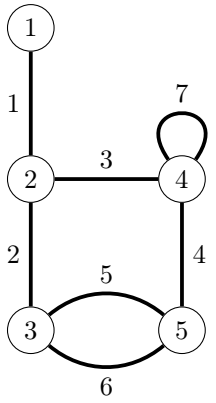
Note. Regarding notation, V and $\mathbb{E}$ are operations on graphs returning the vertex set and edge set respectively.

Let $E(G)$ return a multiset of all vertex-pairs of G; in other words, $E(G) = \{\phi_G(e) \colon e \in \mathbb{E}(G)\}$.

A *drawing* of a graph G is a node-link diagram in which the vertices are represented as disks and the edges are represented as line segments or curves in the Euclidean plane.

Here is a drawing of a graph G and its relation ϕ_G.

G ϕ_G



1: {1, 2}
2: {2, 3}
3: {2, 4}
4: {4, 5}
5: {3, 5}
6: {3, 5}
7: {4, 4}

Two edges $e_1, e_2 \in \mathbb{E}(G)$ are *parallel*, if $\phi_G(e_1) = \phi_G(e_2)$.

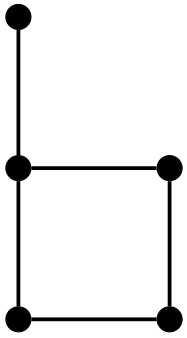
An edge $e_1 \in \mathbb{E}(G)$ is a *parallel edge* if there exists an edge $e_2 \in \mathbb{E}(G)$ such that e_1 and e_2 are parallel. The edge labeled 5 and 6, in the graph above, are examples of parallel edges.

A *self-loop*, is an edge e where $\phi_G(e) = \{u, v\}$ and $u = v$. The edge with label 7, in the graph above, is an example of a self-loop.

A graph G is *loop-less*, if no edge $e \in \mathbb{E}(G)$ is a self-loop.

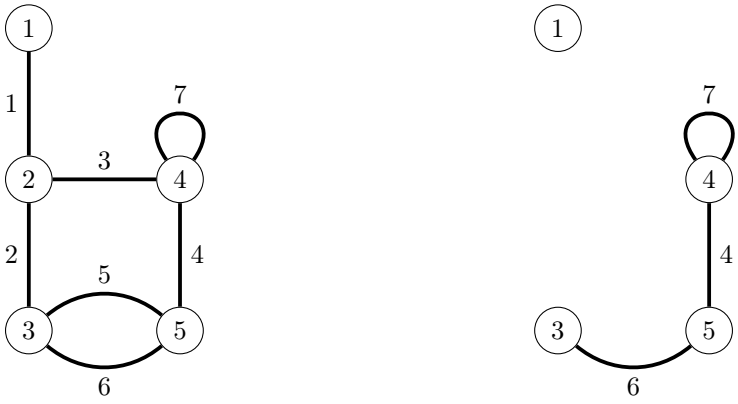
A graph G is *simple*, if it has no parallel edges; in other words, if all elements of $E(G)$ are pair-wise distinct.

A simple graph with no labels shown



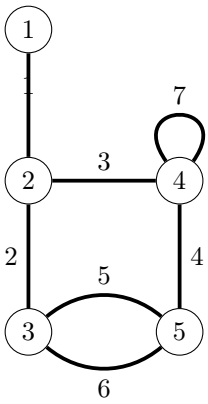
A *subgraph* H of a graph G, is a graph where some vertices and edges might be missing; in other words, is a graph where $V(H) \subseteq V(G)$ and where $\mathbb{E}(H) \subseteq \mathbb{E}(G)$ and where $\forall e \in \mathbb{E}(H), \phi_H(e) = \phi_G(e)$.

G H

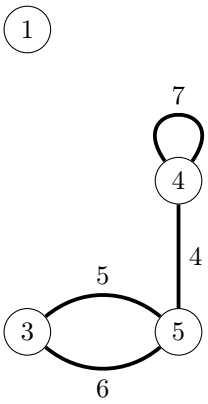


For $A \subseteq V(G)$, we denote by $G[A]$ the subgraph induced by the subset of vertices A; in other words, $G[A]$ is the subgraph where $V(G[A]) = A$ and where $\mathbb{E}(G[A]) = \{e\colon\ e \in \mathbb{E}(G) \wedge |\phi_G(e) \cap A| = 2\}$ and where $\forall e \in E(G[A]), \phi_{G[A]}(e) = \phi_G(e)$.

G A $G[A]$



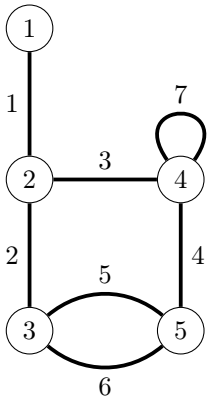
$\{1, 3, 4, 5\}$



A vertex $v \in V(G)$ and an edge $e \in \mathbb{E}(G)$ are *incident* to each other, if $v \in \phi_G(e)$. Furthermore, two distinct edges $e_1, e_2 \in \mathbb{E}(G)$ are incident to each other, if $\phi_G(e_1) \cap \phi_G(e_2) \neq \emptyset$.

The *degree* of a vertex v, denoted $\deg(v)$, is the number of times that an edge is incident to v. A self-loop is incident to the same vertex twice.

G $\deg(v)$



1: 1
2: 3
3: 3
4: 4
5: 3

The *maximum degree* of a graph G, denoted $\Delta(G)$, is the maximum degree of any vertex of G.

$$\Delta(G) = 4$$

A *walk* of a graph G is a list $[v_0, e_1, v_1, ..., e_k, v_k]$ where $v_0, v_1, ..., v_k \in V(G)$ and for $1 \leq i \leq k$, $\phi_G(e_i) = \{v_{i-1}, v_i\}$.

A walk of G

$$[4, 7, 4, 4, 5, 5, 3, 5, 5, 6, 3]$$

The *length* of a walk is the number of edges in the walk.

An *s,t-walk* is a walk where $s = v_0$ and $t = v_k$.

An s, t-walk is *closed*, if $s = t$.

A *path* of a graph G, is a walk such that no vertex is repeated in the list.

A *cycle* of a graph G, is an s, t-walk such that no vertex is repeated in the list except $s = t$.

A graph G is *connected* if there exists a s, t-walk for every pair of distinct vertices $s, t \in V(G)$.

A *component* of a graph, is a connected subgraph.

A *bijection* (or *one-to-one correspondence*) is a relation between two sets such that each element of either set is paired with exactly one element of the other set.

A *plane graph* is a drawing of a graph, such that no edges are crossing.

A graph G is *planar*, if there exists a *plane graph* of G.

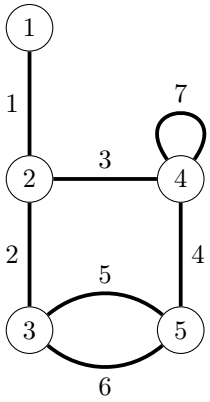
A *rotation system* is an encoding of a graph, in particular, an adjacency list such that the list of neighbors, of every vertex, is sorted in clockwise ordering according to some drawing of G. If you encode a plane graph as a rotation system, you lose the coordinates of the vertices, but you can still recover the faces, to recover an equivalent plane graph.

Definition 2.1. (*Contraction*)

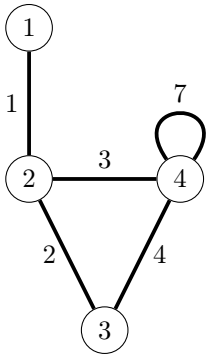
A contraction is an operation that given a graph G and pair of distinct vertices u, v of some edge $\{u, v\} \in E(G)$, removes e if $\phi_G(e) = \{u, v\}$ and makes any edge that is adjacent to v, $\phi_G(e) = \{w, v\}$, adjacent to u instead, $\phi_G(e) = \{w, u\}$, and finally returns the resulting graph.

Corollary 2.2. The resulting graph of a contraction will not have any new self-loops.

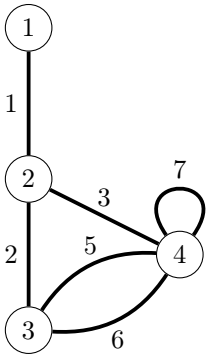
G



Contraction of G
given vertices 3 and 5

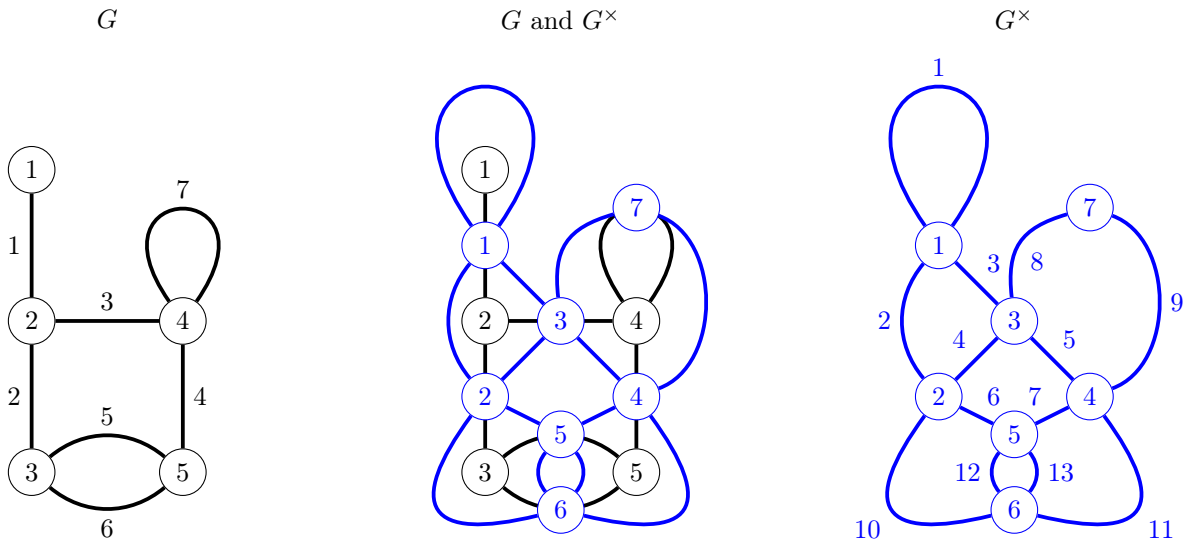


Contraction of G
given vertices 4 and 5



Definition 2.3. (*Medial Graph*)

The medial graph $G^\times$ of a connected plane graph G is a graph such that there is a bijection between $V(G^\times)$ and $\mathbb{E}(G)$ and such that for each face f of G, there's an edge $e^\times \in \mathbb{E}(G^\times)$ incident to a pair of vertices $u^\times, v^\times \in V(G^\times)$ if edges $u, v \in \mathbb{E}(G)$ are consecutive in f.

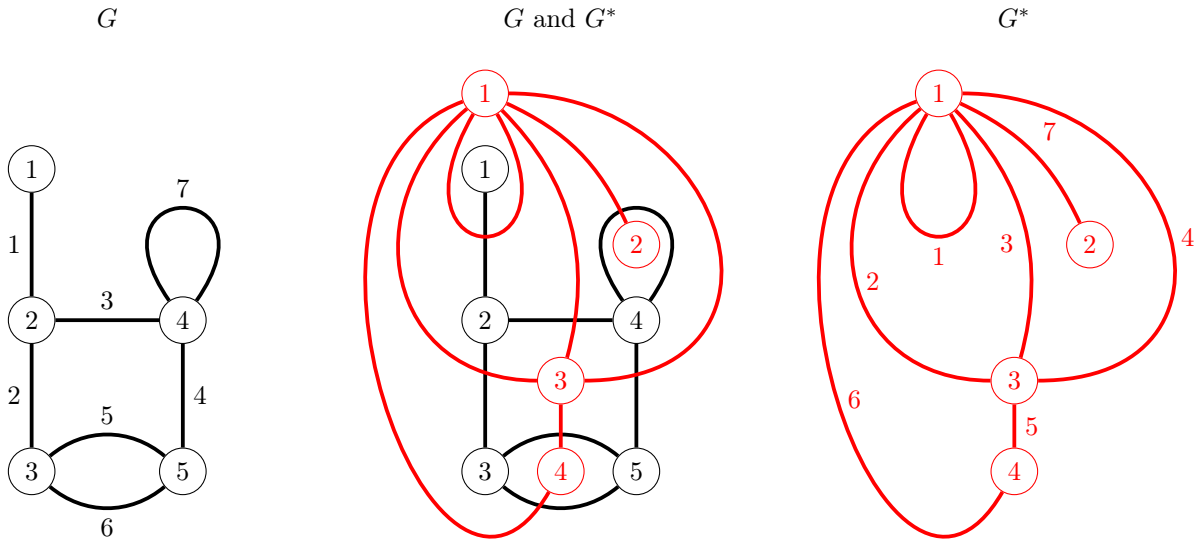


Corollary 2.4. A medial graph is a 4-regular plane graph.

I will refer to vertices and edges of the medial graph as *nodes* and *links* in an attempt at disambiguation.

Definition 2.5. (*Dual Graph*)

The dual graph G^* of a plane graph G is a graph with a bijection between the set of faces of G and $V(G^*)$ and a bijection between $\mathbb{E}(G)$ and $\mathbb{E}(G^*)$ such that an edge $e \in \mathbb{E}(G)$ that separates two faces f_1,f_2 of G is an edge $e^* \in \mathbb{E}(G^*)$ incident to f_1^* and f_2^*.



A *tree* is a connected graph with no cycles.

A *leaf* v of a tree T, is a vertex $v \in V(T)$ of degree 1.

Let the *leaf set* of a tree T, denoted $L(T)$, be the subset of vertices $L(T) \subseteq V(T)$ that are also leaves of T.

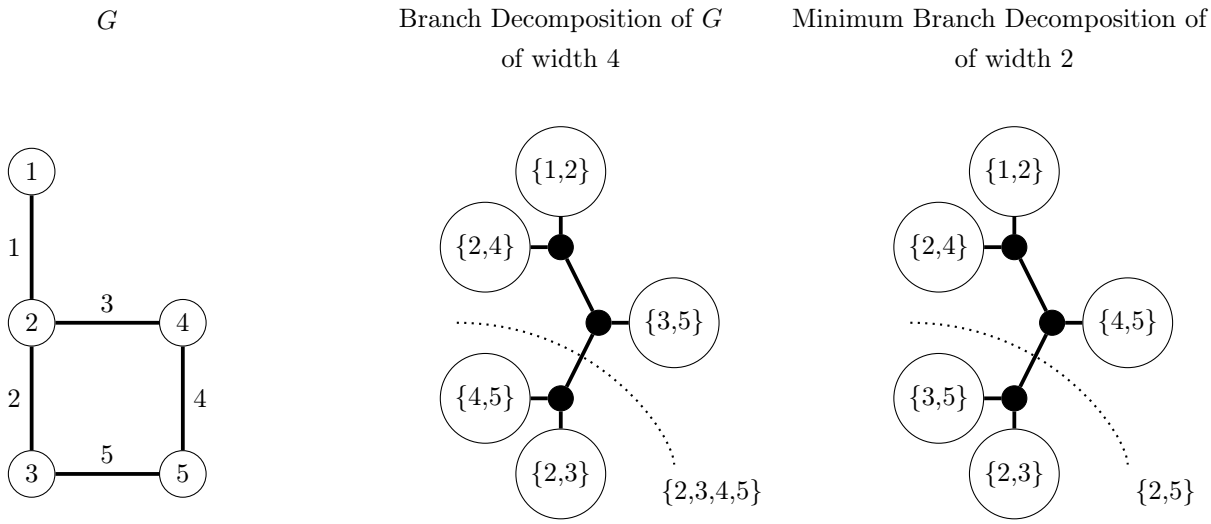
An *internal vertex* v of a tree T, is a vertex $v \in V(T) - L(T)$ that is not a leaf. An internal vertex therefore has at least degree 2.

An *unrooted binary tree* T, is a tree where every internal vertex has degree 3.

A *Branch Decomposition* (B_G, δ_G) of a simple graph G consists of firstly, an unrooted binary tree B_G and secondly a bijection δ_G between $\mathbb{E}(G)$ and $L(B_G)$.

Removing any edge $e \in \mathbb{E}(B_G)$ partitions B_G into 2 trees P_e and Q_e. The set $\cup\{\phi_G(e) : e \in L(P_e)\} \cap \cup\{\phi_G(e) : e \in L(Q_e)\}$ is called a *middle set* of B_G given e, denoted $Z(B_G, e)$. The maximum cardinality of any middle set of B_G given any $e \in \mathbb{E}(B_G)$ is the width of B_G; in other words, the width of B_G is $\max\{|Z(B, e)| : e \in \mathbb{E}(B)\}$.

A *Minimum Branch Decomposition* of G is any branch decomposition of G of minimum width among all branch decompositions of G. Note there might exist many branch decompositions of a graph G, as well as, multiple minimum branch decompositions of a graph G.

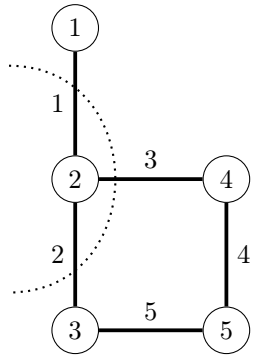


A *Carving Decomposition* (C_G, λ_G) of a simple graph G consists of firstly, an unrooted binary tree C_G and secondly a bijection λ_G between $V(G)$ and $L(C_G)$.

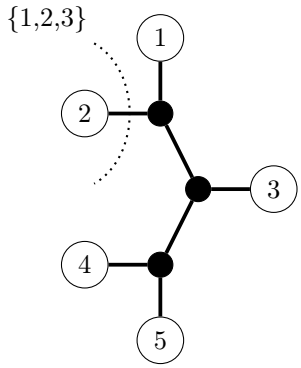
Removing any edge $e \in \mathbb{E}(C_G)$ partitions C_G into 2 trees P_e and Q_e. The set of edges $e \in \mathbb{E}(G)$ that cross from $L(P_e)$ to $L(Q_e)$ is called a *crossing set* of C_G given an $e \in \mathbb{E}(C_G)$, denoted $Y(C_G, e)$. The maximum cardinality of any crossing set of C_G given any $e \in \mathbb{E}(C_G)$ is the width of C_G; in other words, the width of C_G is $\max\{|Y(C, e)| : e \in \mathbb{E}(C)\}$.

A *Minimum Carving Decomposition* of G is any carving decomposition of G of minimum width among all carving decompositions of G.

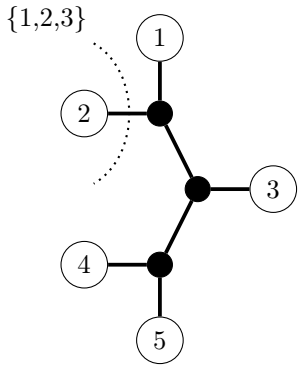
G



Carving Decomposition of G of width 3



Minimum Carving Decomposition of G of width 3



3 Overview of the algorithm

Given a simple connected planar graph G, the algorithm computes a minimum branch decomposition of G.

To compute a minimum branch decomposition of G, the algorithm first computes the medial graph $G^\times$ of G, secondly computes a minimum carving decomposition of $G^\times$, and finally computes a minimum branch decomposition of G from the minimum carving decomposition of $G^\times$.

To compute the minimum carving decomposition of $G^\times$, the algorithm uses a contraction algorithm that finds a series of contractions that does not increase the carving width and then assembles the series of contractions into a minimum carving decomposition of $G^\times$.

To compute the carving width of a graph H, the algorithm uses the rat-catching algorithm. By analogy, the rat-catching algorithm can be described as a game of two players, the rat, and the rat-catcher, who take turns moving around on H. The game is played to determine, if the rat can escape indefinitely or if the rat-catcher can corner the rat by making noise and thereby scaring away the rat from some subgraph, the size of which is dependent on k. Larger k makes the rat-catching game easier for the rat-catcher and vice versa. The smallest k where the rat can escape indefinitely is the carving width of H.

To compute a noisy subgraph of H given a noise level k and a rat-catcher position, the algorithm considers the dual graph of H and finds the set of dual edges reachable by a closed walk of length at most $k-1$, the dual of these edges is the noisy subgraph.

All of these subroutines are described in more detail in the following sections.

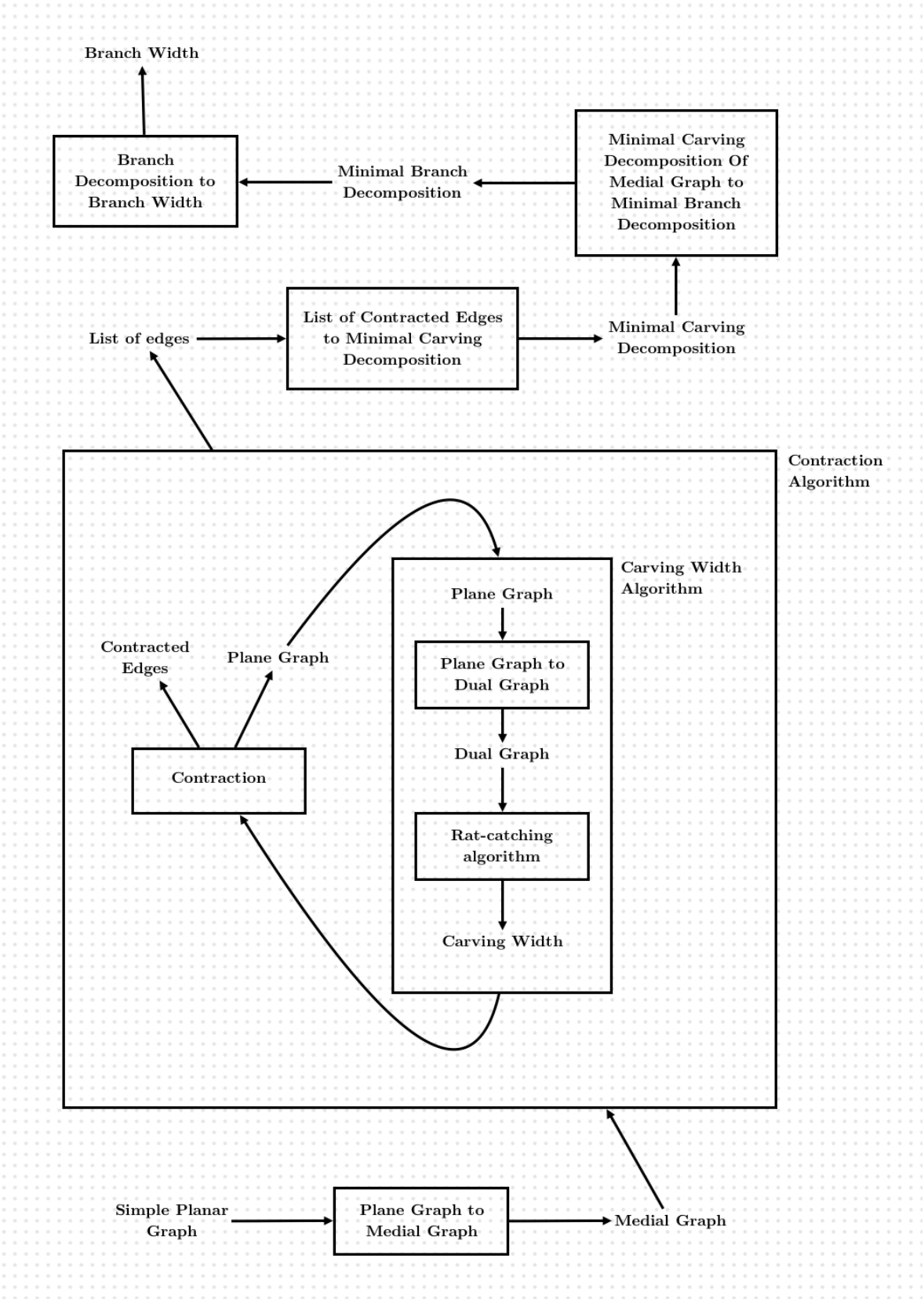


Figure 1: Overview of the algorithm

4 Data Structures and Implementation Considerations

For many of the subroutines, the algorithm needs to deal with parallel edges and be able to tell them apart. Therefore the implementation encodes a graph as an adjacency list of edge IDs and a map from unique edge IDs its vertex-pair.

A *half-edge* is one of the two directed edges that make up an undirected edge in an adjacency list of edge IDs.

I have chosen to assign IDs such that if one half-edge has ID i then the other half-edge has ID $-i$, therefore the absolute value $|i|$ uniquely identifies an undirected edge.

graph.py

```python
class Graph:
        def __init__(self):
                self.adj_edges: dict[int, list[int]] = dict()
                self.edge_to_vertexpair: dict[int, tuple[int, int]] = dict()
                pass

        def from_adj(self, adj: dict[int, list[int]]):
                # assign edge ids
                adj_deepcopy = dict([(u, vs.copy()) for u, vs in adj.items()])
                self.adj_edges = adj_deepcopy
                next_edgeid = 1
                for x, ys in self.adj_edges.items():
                        for i,y in enumerate(ys):
                                if x < y:
                                        self.edge_to_vertexpair[next_edgeid] = (x, y)
                                        self.edge_to_vertexpair[-next_edgeid] = (y, x)

                                        self.adj_edges[x][i] = next_edgeid
                                        self.adj_edges[y][adj[y].index(x)] = -next_edgeid

                                        next_edgeid += 1

        def V(self) -> list[int]:
                return list(self.adj_edges.keys())

        def E(self) -> list[int]:
                return list(self.edge_to_vertexpair.keys())

        def N(self, v: int) -> list[int]:
                return [self.edge_to_vertexpair[e][1] if self.edge_to_vertexpair[e][0] == v else
                ↪  self.edge_to_vertexpair[e][0] for e in self.adj_edges[v]]

        def rename_vertex(self, old: int, new: int):
                self.adj_edges[new] = self.adj_edges.pop(old)
```

Rotation systems are of type `Graph` but the `adj_edges` list is sorted in clockwise ordering according to some drawing of the graph.

Carving-decompositions are adjacency lists `dict[int, list[int]]` .

Branch-decompositions are adjacency lists with either integers or integer pairs as vertices `dict[int, list[Union[int, tuple[int, int]]]]` .

5 The algorithm

The main computational problem of this paper is The Planar Minimum Branch Decomposition Problem.

Definition 5.1. The Planar Minimum Branch Decomposition Problem

Input: Given a simple connected planar graph G.

Output: A minimum branch decomposition of G.

The algorithm described in this paper solves The Planar Minimum Branch Decomposition Problem in polynomial time.

This section describes the algorithm given by Seymour and Thomas[1] by identifying a set of practical problems and subproblems and how they relate.

Problem 5.1 is the overarching problem. It can be broken down into many smaller subproblems.

Considering a plane graph G, you can compute a minimum branch decomposition (B_G, δ_G) of G from a minimum carving decomposition $(C_{G^\times}, \lambda_{G^\times})$ of the medial graph $G^\times$ of G.

Therefore problem 5.1 breaks down into subproblems 5.2, 5.3, and 5.4.

Problem 5.2. Given a plane graph G, output a medial graph $G^\times$, along with a bijectional relation between medial nodes $V(G^\times)$ and edges $\mathbb{E}(G)$.

Problem 5.3. Given a plane graph M, output a minimum carving decomposition of M.

Problem 5.4. Given a minimum carving decomposition of a medial graph of G, output a minimum branch decomposition of G.

With subroutines for all three problems, the implementation obtains a branch decomposition like so.

branch_decomposition.py

```
19
20  # Construct a branch decomposition of a graph
21  def branch_decomposition(G: Graph) -> dict[int, list[Union[int, tuple[int, int]]]]:
22          Gx, node_to_vertexpair = medial_graph(G)
23          cd = carving_decomposition(Gx.copy())
24          bd = carving_decomposition_to_branch_decomposition(cd, node_to_vertexpair)
25          print(adj_to_str(bd))
26          return bd
27
```

The next 3 subsections describe the implementation of the subroutines for the problems 5.2, 5.3, and 5.4.

5.1 Medial graph

Solving problem 5.2.

I assume that the input graph G is a rotation system. Given this format, any two consecutive edges w and v in some face of G are therefore consecutive vertices in the neighborhood of the vertex a that w and v share.

A medial graph is 4-regular; every node has degree 4. Exploiting this property, the implementation iterates over the faces of G, and for each pair of clockwise consecutive edges w and v in some face, adds a link between the nodes corresponding to w and v in the medial graph.

From the perspective of a medial node $v^{\times}$, the first two links are added to its neighborhood in counterclockwise ordering, after processing the first face, and later the final two links are added in counterclockwise ordering, after processing the second face. Therefore the medial graph M is a rotation system.

medial_graph.py

```
# assume G is planar
# assume G_adj is a rotation system
# guarantee an edge of G have the same id as its corresponding medial node
# guarantee M is a rotation system
def medial_graph(G: Graph):
        M = Graph()
        M.adj_edges = dict((abs(n), []) for n in G.edge_to_vertexpair.keys())

        # Find faces
        edges = [e for e in G.E()]
        faces = []
        while len(edges) > 0:
                e = edges.pop()
                next_e = e
                face = [e]
                while True:
                        u,v = G.edge_to_vertexpair[next_e]
                        idx = G.adj_edges[v].index(-next_e)
                        next_e = G.adj_edges[v][(idx-1)%len(G.adj_edges[v])]
                        if (next_e == e):
                                break
                        edges.remove(next_e)
                        face.append(next_e)
                faces.append(face)

        # Add edges to medial graph
        next_linkid = 1
        for face in faces:
                for i in range(len(face)):
                        n1 = abs(face[i])
                        n2 = abs(face[(i+1)%len(face)])
                        M.adj_edges[n1].append(next_linkid)
                        M.adj_edges[n2].append(-next_linkid)
                        M.edge_to_vertexpair[next_linkid] = (n1, n2)
                        M.edge_to_vertexpair[-next_linkid] = (n2, n1)
                        next_linkid += 1
                n1 = abs(face[0])
                e1 = M.adj_edges[n1].pop()
                e2 = M.adj_edges[n1].pop()
                M.adj_edges[n1].append(e1)
                M.adj_edges[n1].append(e2)

        # Make clockwise ordering of neighbors
        for v,es in M.adj_edges.items():
                M.adj_edges[v] = list(reversed(es))

        return M, dict([(k,v) for k,v in G.edge_to_vertexpair.items() if k > 0])

if __name__ == "__main__":
        G = Graph()
        G.from_lmg(read_lmg_from_stdin())
```

5.2 Minimum Branch Decomposition from Minimum Carving Decomposition

Solving problem 5.4.

The implementation identifies the leaf vertices and maps them to vertex-pairs according to the relation

`node_to_vertexpair` from the computation of the medial graph.

branch_decomposition.py

```
6
7  def carving_decomposition_to_branch_decomposition(
8          cd: dict[int, list[int]],
9          node_to_vertexpair: dict[int, tuple[int, int]]
10 ) -> dict[int, list[Union[int, tuple[int, int]]]]:
11
12         leafs = [u for u,vs in cd.items() if len(vs) == 1]
13
14         bd = dict()
15         for u,vs in cd.items():
16                 bd[u if u not in leafs else node_to_vertexpair[u]] = [v if v not in leafs else
                   ↪  node_to_vertexpair[v] for v in vs]
17
```

5.3 Minimum Carving Decomposition

Solving problem 5.3.

By doing a series of edge contractions on a graph M, where the carving width does not increase, until 3 vertices remain, then the series of contracted edges along with the 3 vertices, can be assembled into a minimum carving decomposition of M.

The implementation finds a nonincreasing contraction by doing a linear search over every edge. No consideration has yet been given to any potential clever orderings of the edges that might improve the running time.

The `contraction` function, given a pair of vertices u, v, returns a new unique vertex ID w (instead of reusing u as definition 2.1 suggests), therefore by keeping track of which vertex is a contraction of which vertex-pair, in a dictionary, constructing the decomposition is then a matter of recursively looking up vertices in the dictionary. Repeating this until only vertices of M remain gives a carving decomposition.

carving_decomposition.py

```
7  # Find a contraction that does not increase the carving width
8  def nonincreasing_cw_contraction(G: Graph, cw1: int) -> Union[tuple[Graph, tuple[int, int], int,
   ↪  int], None]:
9          for e in G.E():
10                 u, v = G.edge_to_vertexpair[e]
11                 if u == v:
12                         continue
13                 G2, w = contraction(G, u, v)
14                 cw2 = carving_width(G2)
15                 if cw2 <= cw1:
16                         return G2, (u, v), cw2, w
17         return None
18
19 # Contract edges that do not increase the carving width
20 # until only 3 vertices remain.
21 # Return the resulting graph and the edges that were contracted
22 def gradient_descent_contractions(G: Graph) -> tuple[Graph, dict[int, tuple[int, int]]]:
23         G2 = G.copy()
24         cw1 = carving_width(G)
25         edges = dict()
26         while True:
27                 if len(G2.V()) <= 3:
28                         return G2, edges
29                 res = nonincreasing_cw_contraction(G2, cw1)
30                 if res is not None:
```

```
                        G3, uv, cw2, w = res
                        if len(G3.V()) >= 3:
                                G2 = G3
                                cw1 = cw2
                                edges[w] = uv

# Contruct a carving decomposition of a graph
def carving_decomposition(G: Graph) -> dict[int, list[int]]:
        G2, edges = gradient_descent_contractions(G)

        cd = defaultdict(list)

        # return trivial carving decompositions
        if len(G2.V()) == 1:
                cd[G2.V()[0]] = []
                return cd

        if len(G2.V()) == 2:
                a,b = G2.V()
                cd[a].append(b)
                cd[b].append(a)
                return cd

        # connect the 3 vertices to a new internal vertex and expand
        a,b,c = G2.V()
        d = max(list(edges.keys()) + G2.V()) + 1
        cd[d] = [a,b,c]
        cd[a].append(d)
        cd[b].append(d)
        cd[c].append(d)

        def expand(v):
                if v in edges:
                        a,b = edges[v]
                        cd[v].append(a)
                        cd[v].append(b)
                        cd[a].append(v)
                        cd[b].append(v)
                        if a in edges:
                                expand(a)
                        if b in edges:
                                expand(b)

        for v in G2.V():
                expand(v)

        cd = dict(cd)

        return cd
```

The contraction algorithm depends on a function to compute a contraction and a function to compute the carving width of a graph. These are referred to as problems 5.5 and 5.6.

Problem 5.5. Given a graph M and a pair of vertices $\{u, v\}$, output the graph resulting from a contraction.

Problem 5.6. Given a plane graph M that might have parallel edges, output the carving width of M.

5.3.1 Contraction

Solving problem 5.5.

The implementation needs to preserve the rotation system, as the resulting graph is later given as an argument to functions assuming a rotation system of a planar graph.

Recall the definition 2.1. As this contraction removes all edges connecting a pair of vertices, the resulting graph will not create any new self-loops. I suspect reconciling this and the rotation system could be difficult, but in this context, it is irrelevant.

For a contraction of vertices a and b, I have chosen to create a new vertex ID c (instead of, as the definition suggests, reusing either a or b), as this makes the implementation, for assembling the carving decomposition, simpler.

First, let any edges incident to a or b be incident to c instead. Then creating the neighborhood of the new vertex c is done by firstly finding any shared edge e; an edge that was in the neighborhood of both a and b. This edge has some ID e and the other half-edge, ID $-e$, will therefore be in the neighborhood of b. Now *rotating* the lists representing the neighborhoods of a and b such edge e and $-e$ is at index 0, in their respective lists, means that a concatenation of the lists will preserve the clockwise ordering around the new vertex c. And finally, remove any edges connecting a and b.

This is where telling apart two edges that are parallel, by having edge IDs, becomes very useful. Inferring where to stitch together the neighborhoods to preserve the clockwise ordering, solely from an adjacency list, is not a trivial problem.

contraction.py

```
2
3  # assume G might have parallel edges and self-loops
4  # assume adjacency list of G has clockwise ordering of neighbors
5  # assume a != b
6  def contraction(G: Graph, a: int, b: int) -> tuple[Graph, int]:
7          # copy G
8          G1 = G.copy()
9
10         # create new vertex c
11         c = max(G1.adj_edges.keys()) + 1
12
13         # find edges connecting a and b
14         edges_a_b = []
15         for e in G1.adj_edges[a] + G1.adj_edges[b]:
16                 u,v = G1.edge_to_vertexpair[e]
17                 if (u == a and v == b) or (u == b and v == a):
18                         edges_a_b.append(e)
19
20         # create neighborhood of c while preserving clockwise ordering
21         idx1 = G1.adj_edges[a].index(edges_a_b[0])
22         rotated_Ga = G1.adj_edges[a][idx1:] + G1.adj_edges[a][:idx1]
23
24         idx2 = G1.adj_edges[b].index(-edges_a_b[0])
25         rotated_Gb = G1.adj_edges[b][idx2:] + G1.adj_edges[b][:idx2]
26
27         G1.adj_edges[c] = rotated_Ga + rotated_Gb
28
29         # let every edge incident to a or b be incident to c instead
30         for e in G1.E():
31                 u,v = G1.edge_to_vertexpair[e]
32                 if u == a or u == b:
33                         G1.edge_to_vertexpair[e] = (c, v)
34                 u,v = G1.edge_to_vertexpair[e]
35                 if v == a or v == b:
36                         G1.edge_to_vertexpair[e] = (u, c)
37
38         # remove all edges connecting a and b
39         G1.adj_edges[c] = [e for e in G1.adj_edges[c] if e not in edges_a_b]
40         G1.edge_to_vertexpair = dict([(e,uv) for e,uv in G1.edge_to_vertexpair.items() if e not in
           ↪  edges_a_b])
41
42         # remove a and b
43         del G1.adj_edges[a]
44         del G1.adj_edges[b]
45
46         return G1, c
```

5.3.2 Carving Width

Solving problem 5.6.

The rat-catching algorithm decides whether $cw(M) \geq k$ with $k \in \mathbb{N}^+$. The boolean results of this algorithm for $k = \{1, 2, 3, ...\}$ are monotonic, so you can perform a binary or linear search to find the smallest k where $cw(M) \geq k$ is true.

carving_width.py

```
5  def carving_width(G: Graph) -> int:
6          D, edge_to_link, link_to_edge, node_to_face, edge_to_node = dual_graph(G)
```

carving_width.py

```
130        def binary_search_cw():
131                l = 0
132                r = 1
133                while True:
134                        if rat_wins(r):
135
136                                l = r
137                                r *= 2
138                        else:
139                                break
140                m = l
141                while l < r:
142                        m = int(math.ceil((l + r) / 2))
143                        if rat_wins(m):
144                                l = m
145                        else:
146                                r = m - 1
147                return l
148
149        def linear_search_cw():
150                k = 0
151                while rat_wins(k):
152                        k += 1
153                return k - 1
154
155        cw = binary_search_cw()
156        return cw
```

6 The Rat-Catching Algorithm

The rat-catching algorithm decides $cw(M) \geq k$ with $k \in \mathbb{N}^+$.

The rat-catching algorithm can be described as a game of two players, the rat and rat-catcher. Considering a graph M, the edges of a face can be thought of as walls of a room and vertices as the corners of some rooms. The rat moves from corner to corner along the walls and the rat-catcher moves from room to room through some wall. The rat-catcher can force the rat away from some walls by making noise. A round of this game is played with some noise level k. The rat-catcher wins the round if they can force the rat to be in some wall of the room that they are in, and the rat wins the round if there is a strategy whereby the rat can escape indefinitely.

Additionally, if $\Delta(M) \geq k$ then the rat wins. The argument for why this is true is glossed over in **??**.

So if $\Delta(M) < k$, then the game is played to determine an outcome, otherwise the rat is said to win.

We have arrived at the crux of the algorithm. Does the rat win for some integer k?

For some noise level and location of the rat-catcher, exactly which edges are noisy and which are quiet are definitions 6.1 and 6.2.

Definition 6.1. When the rat-catcher is on some edge e_1, then edge e_2 is noisy iff. there is a closed walk of length scrictly less than k containing e_1^* and e_2^* in the dual M^*.

An edge e is called quiet iff. e is not noisy.

Definition 6.2. When the rat-catcher is in some face f, then edge e is noisy iff. there is a closed walk of length scrictly less than k containing f^* and e^* in the dual M^*.

A *quiet subgraph* $Q(M, k, e)$, for some graph M and some noise level k and some edge $e \in \mathbb{E}(M)$, is a subgraph of M with the vertex set $V(Q(M, k, e)) = V(M)$ and the edge set

$\mathbb{E}(Q(M, k, e)) = \{e_1 : \text{ every closed walk of } M^* \text{ containing } e_1^* \text{ and } e_2^* \text{ has length at least } k\}$

Problem 6.3. Given a plane graph M that might have parallel edges, an edge $e \in \mathbb{E}(M)$, and noise level $k \in \mathbb{N}^+$, output the quiet subgraph $Q(M, k, e)$.

Problem 6.3 depends on a function for computing the dual of a graph. Computing a dual graph is problem 6.4.

Problem 6.4. Given a plane graph $M = \{V, E\}$ that might have parallel edges, output the dual of M.

The game states and possible moves, for some graph M and some noise level k, can be described as a graph $H(M, k)$.

Let $F(M)$ be the set of faces of M.

Let S be every possible state when the rat-catcher is in a face, some of which might be losing states. $S = \{(f, v) : v \in V(M) \wedge f \text{ is a face of } M\}$.

Let T be every possible state when the rat-catcher is on an edge. $T = \{(e, C) : e \in \mathbb{E}(M) \wedge C \text{ is a component of } Q(M, k, e)\}$.

With the graph of possible moves H, the only missing piece of the rat-catching algorithm is how to determine the outcome.

You can mark vertices of the graph H that are losing states, and then repeatedly mark any state that leads to a losing state, until either every state is marked or no more states can be marked. If every state is marked then the rat-catcher wins, otherwise the rat wins.

Solving problem 5.6.

The vertices of the game state graph H are initialized by computing the elements of T and S, while edges of H are not explicitly kept in any data structure, but instead checked while playing the game.

Losing states (the tuples $(f, v) \in S$ where $v \in f$) are marked as losing.

The outcome of the game is computed by marking states as losing.

Considering a state $(e, C) \in T$, if all (f, v) where $v \in V(C)$ is losing then (e, C) is losing.

Considering a state (f, v), if there exists a state (e, C) that is losing where $e \in f$ and $v \in V(C)$ then (f, v) is losing.

carving_width.py

```
        # Assume |V(G)| >= 2
        # Return True
        # iff. carving-width >= k
        # iff. rat has a winning escape strategy with noise-level k
        def rat_wins(k: int) -> bool:
                if len(G.V()) < 2:
                        return False

                if max([len(G.N(v)) for v in G.V()]) >= k:
                        return True

                # Set up the game graph
                halfedges = edge_to_link.keys()

                T = set([(e, tuple(C)) for e in halfedges for C in quiet_components(e, k)])
                S = set([(f, v) for f in node_to_face.keys() for v in G.V()])

                # Set up the initial losing states
                losing_T = set()
                losing_S = set()

                for (f, v) in S:
                        if v in flatten([G.edge_to_vertexpair[e] for e in node_to_face[f]]):
                                losing_S.add((f,v))

                if len(T) == len(losing_T) or len(S) == len(losing_S):
                        return False

                # Play the game
                while True:
                        new_deletion = False

                        for (e, C) in T:
                                if all([(edge_to_node[e], v) in losing_S for v in C]):
                                        if (e, C) not in losing_T:
                                                new_deletion = True
                                                losing_T.add((e, C))

                        for (e, C) in losing_T:
                                f1 = edge_to_node[e]
                                f2 = edge_to_node[-e]
                                for (f, v) in [(f1, v) for v in C] + [(f2, v) for v in C]:
                                        if (f, v) not in losing_S:
                                                new_deletion = True
                                                losing_S.add((f, v))

                        if len(T) == len(losing_T) or len(S) == len(losing_S):
                                return False
                        elif not new_deletion:
                                return True
```

6.1 Quiet subgraph

Solving problem 6.3.

Recall the definition of a noisy edge 6.1.

Let s_1 and t_1 be the vertex-pair for the link e_1^* and let s_2 and t_2 be the vertex-pair for the link e_2^*.

Claim 6.5. The shortest closed walk that includes both e_1^* and e_2^* is the minimum of either

$$d(s_1, s_2) + d(t_1, t_2) + 2$$

or

$$d(s_1, t_1) + d(s_2, t_2) + 2$$

. Where $d(u, v)$ is the length of the shortest u, v-path.

The single source shortest distances can then be computed using a breadth-first approach.

Using the mapping from links to edges, and the fact that an edge e is called quiet iff. e is not noisy, the quiet edges can be obtained by applying the map.

Computing the quiet subgraph and the components thereof is done with a depth-first search approach.

The edges of the components are irrelevant for the rest of the algorithm, so only a list of vertices is returned for each component.

carving_width.py

```
        D, edge_to_link, link_to_edge, node_to_face, edge_to_node = dual_graph(G)

        # If the rat-catcher is on edge e1, then edge e2 is noisy iff there is
        # a closed walk of length scrictly less than k containing e1* and e2* in the dual G*.

        def noisy_links(l: int, k: int) -> set[int]:
                s,t = D.edge_to_vertexpair[l]
                links = link_to_edge.keys()

                def dists(n: int) -> dict[int, int]:
                        dist = {v: -1 for v in D.V()}
                        dist[n] = 0
                        queue = [n]
                        while len(queue) > 0:
                                v = queue.pop(0)
                                for y in D.N(v):
                                        if dist[y] == -1:
                                                dist[y] = dist[v] + 1
                                                queue.append(y)
                        return dist

                dist_s = dists(s)
                dist_t = dists(t)

                noisy = []
                for l1 in links:
                        u,v = D.edge_to_vertexpair[l1]
                        if min(
                                dist_s[u] + dist_t[v] + 2,
                                dist_s[v] + dist_t[u] + 2
                        ) < k:
                                noisy.append(l1)

                return set([abs(e) for e in noisy])

        def quiet_links(l: int, k: int) -> set[int]:
                links = set([abs(e) for e in D.E()])
                return links - noisy_links(l, k)

        def quiet_edges(e: int, k: int) -> set[int]:
                return set([abs(link_to_edge[l]) for l in quiet_links(edge_to_link[e], k)])

        def quiet_components(e: int, k: int) -> list[list[int]]:
                edges = quiet_edges(e, k)

                quiet_subgraph = {v: [] for v in G.V()}
                for e1 in edges:
                        u,v = G.edge_to_vertexpair[e1]
                        quiet_subgraph[u].append(e1)
                        quiet_subgraph[v].append(-e1)

                components = []
```

```
58                 unseen = set(quiet_subgraph.keys())
59
60                 while len(unseen) > 0:
61                         v = unseen.pop()
62                         component = [v]
63                         stack = [v]
64                         while len(stack) > 0:
65                                 v = stack.pop()
66                                 for e1 in quiet_subgraph[v]:
67                                         u,v = G.edge_to_vertexpair[e1]
68                                         if v in unseen:
69                                                 unseen.remove(v)
70                                                 stack.append(v)
71                                                 component.append(v)
72                         components.append(component)
73
74                 return components
```

6.2 Dual graph

Solving problem 6.4

No other path of the implementation needs the assumption that the dual is planar, therefore the output doesn't need to be a rotation system. This simplifies the implementation.

The dual has a vertex for each face of the input graph. The faces are found by selecting an unmarked half-edge, then marking all the edges of the face it belongs to, and repeating this until all half-edges are marked.

The next halfedge e_{i+1} after the current halfedge $e_i = \{u, v\}$ is the edge just before $-e_i$ in the neighborhood list for vertex v.

dual_graph.py

```
4  def dual_graph(G: Graph) -> tuple[Graph, dict[int, int], dict[int, int], dict[int, list[int]],
   ↪  dict[int, int]]:
5          edges = [e for e in G.E()]
6
7          D = Graph()
8          edge_to_link: dict[int, int] = dict()
9          link_to_edge: dict[int, int] = dict()
10         node_to_face: dict[int, list[int]] = dict()  # nodeid to edgeid list
11         edge_to_node: dict[int, int] = dict()        # half-edge to the faceid/node to its either
           ↪  left/right
12
13         # Find faces
14         next_nodeid = -1
15         while edges:
16                 e = edges.pop()
17                 next_e = e
18                 edge_to_node[e] = next_nodeid
19                 face = [e]
20                 while True:
21                         u,v = G.edge_to_vertexpair[next_e]
22                         idx = G.adj_edges[v].index(-next_e)
23                         next_e = G.adj_edges[v][(idx-1)%len(G.adj_edges[v])]
24                         if (next_e == e):
25                                 break
26                         edges.remove(next_e)
27                         face.append(next_e)
28                         edge_to_node[next_e] = next_nodeid
29                 node_to_face[next_nodeid] = face
30                 next_nodeid -= 1
31
```

```
        for i in node_to_face.keys():
                D.adj_edges[i] = []

        # Add edges to dual graph
        next_linkid = 1
        for i,f1 in node_to_face.items():
                for j,f2 in node_to_face.items():
                        if i < j:
                                common_edges = set(set(map(abs, f1))).intersection(set(map(abs, f2)))
                                for e in common_edges:
                                        D.edge_to_vertexpair[next_linkid] = (i, j)
                                        D.edge_to_vertexpair[-next_linkid] = (j, i)
                                        edge_to_link[e] = next_linkid
                                        link_to_edge[next_linkid] = e
                                        edge_to_link[-e] = -next_linkid
                                        link_to_edge[-next_linkid] = -e
                                        D.adj_edges[i].append(next_linkid)
                                        D.adj_edges[j].append(-next_linkid)
                                        next_linkid += 1

        # todo make edge_to_link and link_to_edge redundant
        # by nameing edges and links the same

        return D, edge_to_link, link_to_edge, node_to_face, edge_to_node

```

7 Results

The algorithm has been implemented in Python 3.12.2. The source code is available on GitHub [2].

The implementation has been tested against two sets of graphs, using various methods.

The first set of graphs are random cubic planar graphs. The second set of graphs are small simple planar graphs generated by *Plantri*[3][4].

The random cubic planar graphs is a set of 23 graphs, that were generated by a script, created by Andreas Björklund. See appendix 11.12 for the generator script. The graphs are available on GitHub [5]. They also created a branch width algorithm that was cross-validated with the implementation from this paper. The two algorithms agreed on every value. See appendix 11.9 for the script.

The second set of small simple planar graphs was generated by Plantri, which exhaustively generated all 2228 simple planar graphs on 2 to 7 vertices. Testing the implementation on these graphs, the results are shown in figure 2. The scripts for generating, processing, and testing this set of graphs are in appendix 11.13, 11.14, and 11.10. The graphs are available on GitHub [6]

Fomin and Thilikos[4] find a new theoretical upper bound on the branch width of planar graphs, $bw(G) \leq 4.5 \cdot \sqrt{n}$, where n is the number of vertices in the graph.

Some of the graphs are suspected to be cause the implementation to go into an infinite loop, as the branch width is not computed within a reasonable time. So the implementation is suspectedly not correct for all planar graphs.

[2] https://github.com/hojelse/thesis
[3] http://users.cecs.anu.edu.au/~bdm/plantri/
[4] https://github.com/mishun/plantri/tree/master
[5] https://github.com/hojelse/thesis/tree/main/code/graphs/random-planar-cubic
[6] https://github.com/hojelse/thesis/tree/main/code/graphs/simple-planar

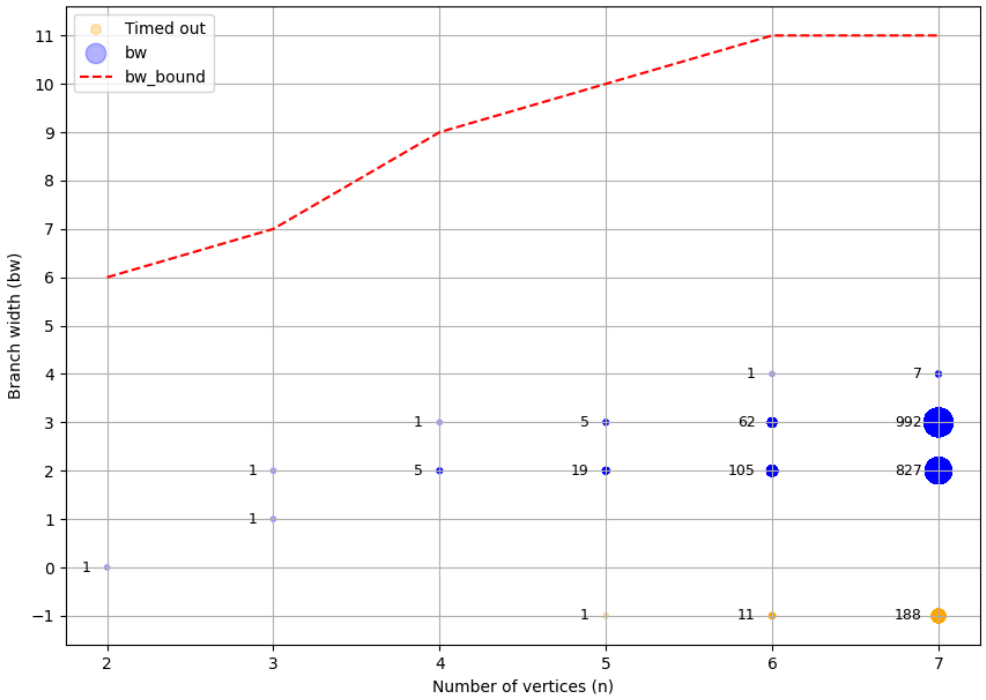


Figure 2: Branch width of planar graphs generated by Plantri. The x-axis is the number of vertices in the graph, and the y-axis is the branch width. The red line is $\lfloor 4.5 \cdot \sqrt{n} \rfloor$. The blue dots are the branch width and the number directly next to it is the number of the graph with the same unique (n, bw) pair.

On the same set of graphs, the implementation was tested against itself by permuting the vertex labels of input graphs, which should not change the branch width. This was not done exhaustively, but the results were as expected.

A tool, *grafer*, was developed for visualizing and editing planar graphs and used for debugging during this thesis `https://github.com/hojelse/grafer`.

8 Future work

The implementation was only tested on cubic planar graphs until late in the process of writing this thesis, as there were plans to use this branch-decomposition algorithm as a subroutine in a larger algorithm. As seen in the results, the implementation does not behave nicely for all planar graphs. Debugging the implementation on the simple planar graphs generated by Plantri that fail is a good next step.

9 Conclusion

This paper presents a clear and accessible implementation of the rat-catching algorithm for planar graphs, based on the work of Seymour and Thomas. The implementation follows the original algo-

rithm and is supported by a framework for testing correctness to ensure its reliability. The results regrettably show that there are still bugs to eliminate. The source code provided alongside this paper is intended as a practical resource for those working on algorithm design involving branch- and carving decompositions. I hope this work will be useful for students, researchers, or practitioners looking to learn about, use, or develop a branch-decomposition algorithm by examining, using, or modifying the code provided alongside this paper. `https://github.com/hojelse/thesis`.

10 References

References

[1] “Call Routing and The Ratcatcher”. In: ().

[2] “Cut and Count Representative Sets on Branch Decompositions”. In: ().

[3] “Eppstein, David (2018), "The effect of planarization on width", Journal of Graph Algorithms and Applications, 22 (3): 461-481”. In: ().

[4] “Fomin, Fedor V., and Dimitrios M. Thilikos. "New upper bounds on the decomposability of planar graphs." Journal of Graph Theory 51.1 (2006): 53-81.” In: ().

[5] “Practical algorithms for branch-decompositions of planar graphs”. In: ().

[6] “Robertson and Seymour 1991, Theorem 5.1, p. 168.” In: ().

11 Appendix

11.1

branch_decomposition.py

```python
from typing import Union
from Graph import Graph, read_lmg_from_stdin
from medial_graph import medial_graph
from carving_decomposition import carving_decomposition
from util import adj_to_str

def carving_decomposition_to_branch_decomposition(
        cd: dict[int, list[int]],
        node_to_vertexpair: dict[int, tuple[int, int]]
) -> dict[int, list[Union[int, tuple[int, int]]]]:

        leafs = [u for u,vs in cd.items() if len(vs) == 1]

        bd = dict()
        for u,vs in cd.items():
                bd[u if u not in leafs else node_to_vertexpair[u]] = [v if v not in leafs else
                ↪  node_to_vertexpair[v] for v in vs]

        return bd

# Construct a branch decomposition of a graph
def branch_decomposition(G: Graph) -> dict[int, list[Union[int, tuple[int, int]]]]:
        Gx, node_to_vertexpair = medial_graph(G)
        cd = carving_decomposition(Gx.copy())
        bd = carving_decomposition_to_branch_decomposition(cd, node_to_vertexpair)
        print(adj_to_str(bd))
        return bd

if __name__ == "__main__":
        G = Graph()
        G.from_lmg(read_lmg_from_stdin())
        bd = branch_decomposition(G)
        print("bd", bd)
```

11.2

branch_width.py

```python
from typing import Union
from Graph import Graph
from branch_decomposition import branch_decomposition
from util import adj_from_stdin

def branch_width_of_branch_decomposition(bd: dict[int, list[Union[int, tuple[int, int]]]]) -> int:
        # Get the vertex set of the leafs of the subtree of x (not y)
        def leafs_set(x, y):
                leafs = set()
                visited = set([y])
                stack = [x]
                while stack:
                        v = stack.pop()
                        if isinstance(v, tuple):
                                leafs.update(set(v))
                                continue
                        if v not in visited:
                                visited.add(v)
                                for w in bd[v]:
```

```
                                        stack.append(w)
                return leafs

        # Find the maximal width of any middle set
        width = 0
        for x,ys in bd.items():
                for y in ys:
                        a = leafs_set(x, y)
                        b = leafs_set(y, x)
                        middle_set = len(a.intersection(b))
                        width = max(width, middle_set)

        return width

def branch_width(G: Graph) -> int:
        bd = branch_decomposition(G)
        bw = branch_width_of_branch_decomposition(bd)

        return bw

if __name__ == "__main__":
        # G = Graph()
        # G.from_lmg(read_lmg_from_stdin())
        # bw = branch_width(G)
        # print("bw", bw)

        adj = adj_from_stdin()
        G = Graph()
        G.from_adj(adj)
        bw = branch_width(G)
        print(bw)
```

11.3

carving_decomposition.py

```
from collections import defaultdict
from typing import Union
from carving_width import carving_width
from contraction import contraction
from Graph import Graph, read_lmg_from_stdin

# Find a contraction that does not increase the carving width
def nonincreasing_cw_contraction(G: Graph, cw1: int) -> Union[tuple[Graph, tuple[int, int], int,
↪  int], None]:
        for e in G.E():
                u, v = G.edge_to_vertexpair[e]
                if u == v:
                        continue
                G2, w = contraction(G, u, v)
                cw2 = carving_width(G2)
                if cw2 <= cw1:
                        return G2, (u, v), cw2, w
        return None

# Contract edges that do not increase the carving width
# until only 3 vertices remain.
# Return the resulting graph and the edges that were contracted
def gradient_descent_contractions(G: Graph) -> tuple[Graph, dict[int, tuple[int, int]]]:
        G2 = G.copy()
        cw1 = carving_width(G)
        edges = dict()
        while True:
                if len(G2.V()) <= 3:
                        return G2, edges
```

```
                res = nonincreasing_cw_contraction(G2, cw1)
                if res is not None:
                        G3, uv, cw2, w = res
                        if len(G3.V()) >= 3:
                                G2 = G3
                                cw1 = cw2
                                edges[w] = uv

# Contruct a carving decomposition of a graph
def carving_decomposition(G: Graph) -> dict[int, list[int]]:
        G2, edges = gradient_descent_contractions(G)

        cd = defaultdict(list)

        # return trivial carving decompositions
        if len(G2.V()) == 1:
                cd[G2.V()[0]] = []
                return cd

        if len(G2.V()) == 2:
                a,b = G2.V()
                cd[a].append(b)
                cd[b].append(a)
                return cd

        # connect the 3 vertices to a new internal vertex and expand
        a,b,c = G2.V()
        d = max(list(edges.keys()) + G2.V()) + 1
        cd[d] = [a,b,c]
        cd[a].append(d)
        cd[b].append(d)
        cd[c].append(d)

        def expand(v):
                if v in edges:
                        a,b = edges[v]
                        cd[v].append(a)
                        cd[v].append(b)
                        cd[a].append(v)
                        cd[b].append(v)
                        if a in edges:
                                expand(a)
                        if b in edges:
                                expand(b)

        for v in G2.V():
                expand(v)

        cd = dict(cd)

        return cd

if __name__ == "__main__":
        G = Graph()
        G.from_lmg(read_lmg_from_stdin())
        cd = carving_decomposition(G)
        print(cd)
```

11.4

carving_width.py

```
import math
from Graph import Graph, read_lmg_from_stdin
from dual_graph import dual_graph
```

```
4
5   def carving_width(G: Graph) -> int:
6           D, edge_to_link, link_to_edge, node_to_face, edge_to_node = dual_graph(G)
7
8           # If the rat-catcher is on edge e1, then edge e2 is noisy iff there is
9           # a closed walk of length scrictly less than k containing e1* and e2* in the dual G*.
10
11          def noisy_links(l: int, k: int) -> set[int]:
12                  s,t = D.edge_to_vertexpair[l]
13                  links = link_to_edge.keys()
14
15                  def dists(n: int) -> dict[int, int]:
16                          dist = {v: -1 for v in D.V()}
17                          dist[n] = 0
18                          queue = [n]
19                          while len(queue) > 0:
20                                  v = queue.pop(0)
21                                  for y in D.N(v):
22                                          if dist[y] == -1:
23                                                  dist[y] = dist[v] + 1
24                                                  queue.append(y)
25                          return dist
26
27                  dist_s = dists(s)
28                  dist_t = dists(t)
29
30                  noisy = []
31                  for l1 in links:
32                          u,v = D.edge_to_vertexpair[l1]
33                          if min(
34                                  dist_s[u] + dist_t[v] + 2,
35                                  dist_s[v] + dist_t[u] + 2
36                          ) < k:
37                                  noisy.append(l1)
38
39                  return set([abs(e) for e in noisy])
40
41          def quiet_links(l: int, k: int) -> set[int]:
42                  links = set([abs(e) for e in D.E()])
43                  return links - noisy_links(l, k)
44
45          def quiet_edges(e: int, k: int) -> set[int]:
46                  return set([abs(link_to_edge[l]) for l in quiet_links(edge_to_link[e], k)])
47
48          def quiet_components(e: int, k: int) -> list[list[int]]:
49                  edges = quiet_edges(e, k)
50
51                  quiet_subgraph = {v: [] for v in G.V()}
52                  for e1 in edges:
53                          u,v = G.edge_to_vertexpair[e1]
54                          quiet_subgraph[u].append(e1)
55                          quiet_subgraph[v].append(-e1)
56
57                  components = []
58                  unseen = set(quiet_subgraph.keys())
59
60                  while len(unseen) > 0:
61                          v = unseen.pop()
62                          component = [v]
63                          stack = [v]
64                          while len(stack) > 0:
65                                  v = stack.pop()
66                                  for e1 in quiet_subgraph[v]:
67                                          u,v = G.edge_to_vertexpair[e1]
68                                          if v in unseen:
69                                                  unseen.remove(v)
70                                                  stack.append(v)
71                                                  component.append(v)
72                          components.append(component)
73
```

```
                return components

        def flatten(xss):
                return set([x for xs in xss for x in xs])

        # Assume |V(G)| >= 2
        # Return True
        # iff. carving-width >= k
        # iff. rat has a winning escape strategy with noise-level k
        def rat_wins(k: int) -> bool:
                if len(G.V()) < 2:
                        return False

                if max([len(G.N(v)) for v in G.V()]) >= k:
                        return True

                # Set up the game graph
                halfedges = edge_to_link.keys()

                T = set([(e, tuple(C)) for e in halfedges for C in quiet_components(e, k)])
                S = set([(f, v) for f in node_to_face.keys() for v in G.V()])

                # Set up the initial losing states
                losing_T = set()
                losing_S = set()

                for (f, v) in S:
                        if v in flatten([G.edge_to_vertexpair[e] for e in node_to_face[f]]):
                                losing_S.add((f,v))

                if len(T) == len(losing_T) or len(S) == len(losing_S):
                        return False

                # Play the game
                while True:
                        new_deletion = False

                        for (e, C) in T:
                                if all([(edge_to_node[e], v) in losing_S for v in C]):
                                        if (e, C) not in losing_T:
                                                new_deletion = True
                                                losing_T.add((e, C))

                        for (e, C) in losing_T:
                                f1 = edge_to_node[e]
                                f2 = edge_to_node[-e]
                                for (f, v) in [(f1, v) for v in C] + [(f2, v) for v in C]:
                                        if (f, v) not in losing_S:
                                                new_deletion = True
                                                losing_S.add((f, v))

                        if len(T) == len(losing_T) or len(S) == len(losing_S):
                                return False
                        elif not new_deletion:
                                return True

        def binary_search_cw():
                l = 0
                r = 1
                while True:
                        if rat_wins(r):

                                l = r
                                r *= 2
                        else:
                                break
                m = l
                while l < r:
                        m = int(math.ceil((l + r) / 2))
                        if rat_wins(m):
```

```
                                l = m
                        else:
                                r = m - 1
                return l

        def linear_search_cw():
                k = 0
                while rat_wins(k):
                        k += 1
                return k - 1

        cw = binary_search_cw()
        return cw

if __name__ == "__main__":
        G = Graph()
        G.from_lmg(read_lmg_from_stdin())
        cw = carving_width(G)
        print("cw", cw)
```

11.5

contraction.py

```
from Graph import Graph, read_lmg_from_stdin

# assume G might have parallel edges and self-loops
# assume adjacency list of G has clockwise ordering of neighbors
# assume a != b
def contraction(G: Graph, a: int, b: int) -> tuple[Graph, int]:
        # copy G
        G1 = G.copy()

        # create new vertex c
        c = max(G1.adj_edges.keys()) + 1

        # find edges connecting a and b
        edges_a_b = []
        for e in G1.adj_edges[a] + G1.adj_edges[b]:
                u,v = G1.edge_to_vertexpair[e]
                if (u == a and v == b) or (u == b and v == a):
                        edges_a_b.append(e)

        # create neighborhood of c while preserving clockwise ordering
        idx1 = G1.adj_edges[a].index(edges_a_b[0])
        rotated_Ga = G1.adj_edges[a][idx1:] + G1.adj_edges[a][:idx1]

        idx2 = G1.adj_edges[b].index(-edges_a_b[0])
        rotated_Gb = G1.adj_edges[b][idx2:] + G1.adj_edges[b][:idx2]

        G1.adj_edges[c] = rotated_Ga + rotated_Gb

        # let every edge incident to a or b be incident to c instead
        for e in G1.E():
                u,v = G1.edge_to_vertexpair[e]
                if u == a or u == b:
                        G1.edge_to_vertexpair[e] = (c, v)
                u,v = G1.edge_to_vertexpair[e]
                if v == a or v == b:
                        G1.edge_to_vertexpair[e] = (u, c)

        # remove all edges connecting a and b
        G1.adj_edges[c] = [e for e in G1.adj_edges[c] if e not in edges_a_b]
        G1.edge_to_vertexpair = dict([(e,uv) for e,uv in G1.edge_to_vertexpair.items() if e not in
        ↪  edges_a_b])
```

```
41
42          # remove a and b
43          del G1.adj_edges[a]
44          del G1.adj_edges[b]
45
46          return G1, c
47
48  if __name__ == "__main__":
49          a,b = map(int, input().split())
50          G = Graph()
51          G.from_lmg(read_lmg_from_stdin())
52          G1, c = contraction(G, a, b)
53          print(G1)
54          print("c", c)
```

11.6

dual_graph.py

```
1   from Graph import Graph, read_lmg_from_stdin
2
3   # Assume G is a rotation system
4   def dual_graph(G: Graph) -> tuple[Graph, dict[int, int], dict[int, int], dict[int, list[int]],
    ↪  dict[int, int]]:
5           edges = [e for e in G.E()]
6
7           D = Graph()
8           edge_to_link: dict[int, int] = dict()
9           link_to_edge: dict[int, int] = dict()
10          node_to_face: dict[int, list[int]] = dict()  # nodeid to edgeid list
11          edge_to_node: dict[int, int] = dict()        # half-edge to the faceid/node to its either
            ↪  left/right
12
13          # Find faces
14          next_nodeid = -1
15          while edges:
16                  e = edges.pop()
17                  next_e = e
18                  edge_to_node[e] = next_nodeid
19                  face = [e]
20                  while True:
21                          u,v = G.edge_to_vertexpair[next_e]
22                          idx = G.adj_edges[v].index(-next_e)
23                          next_e = G.adj_edges[v][(idx-1)%len(G.adj_edges[v])]
24                          if (next_e == e):
25                                  break
26                          edges.remove(next_e)
27                          face.append(next_e)
28                          edge_to_node[next_e] = next_nodeid
29                  node_to_face[next_nodeid] = face
30                  next_nodeid -= 1
31
32          for i in node_to_face.keys():
33                  D.adj_edges[i] = []
34
35          # Add edges to dual graph
36          next_linkid = 1
37          for i,f1 in node_to_face.items():
38                  for j,f2 in node_to_face.items():
39                          if i < j:
40                                  common_edges = set(set(map(abs, f1))).intersection(set(map(abs, f2)))
41                                  for e in common_edges:
42                                          D.edge_to_vertexpair[next_linkid] = (i, j)
43                                          D.edge_to_vertexpair[-next_linkid] = (j, i)
44                                          edge_to_link[e] = next_linkid
```

```
                                        link_to_edge[next_linkid] = e
                                        edge_to_link[-e] = -next_linkid
                                        link_to_edge[-next_linkid] = -e
                                        D.adj_edges[i].append(next_linkid)
                                        D.adj_edges[j].append(-next_linkid)
                                        next_linkid += 1

        # todo make edge_to_link and link_to_edge redundant
        # by nameing edges and links the same

        return D, edge_to_link, link_to_edge, node_to_face, edge_to_node

if __name__ == "__main__":
        G = Graph()
        G.from_lmg(read_lmg_from_stdin())
        D, edge_to_link, link_to_edge, node_to_face, edge_to_node = dual_graph(G)
        print_adj(D.adj())
        print("edge_to_link", edge_to_link)
        print("link_to_edge", link_to_edge)
        print("node_to_face", node_to_face)
        print("edge_to_node", edge_to_node)
```

11.7

Graph.py

```
class Graph:
        def __init__(self):
                self.adj_edges: dict[int, list[int]] = dict()
                self.edge_to_vertexpair: dict[int, tuple[int, int]] = dict()
                pass

        def from_adj(self, adj: dict[int, list[int]]):
                # assign edge ids
                adj_deepcopy = dict([(u, vs.copy()) for u, vs in adj.items()])
                self.adj_edges = adj_deepcopy
                next_edgeid = 1
                for x, ys in self.adj_edges.items():
                        for i,y in enumerate(ys):
                                if x < y:
                                        self.edge_to_vertexpair[next_edgeid] = (x, y)
                                        self.edge_to_vertexpair[-next_edgeid] = (y, x)

                                        self.adj_edges[x][i] = next_edgeid
                                        self.adj_edges[y][adj[y].index(x)] = -next_edgeid

                                        next_edgeid += 1

        def V(self) -> list[int]:
                return list(self.adj_edges.keys())

        def E(self) -> list[int]:
                return list(self.edge_to_vertexpair.keys())

        def N(self, v: int) -> list[int]:
                return [self.edge_to_vertexpair[e][1] if self.edge_to_vertexpair[e][0] == v else
                ↪  self.edge_to_vertexpair[e][0] for e in self.adj_edges[v]]

        def rename_vertex(self, old: int, new: int):
                self.adj_edges[new] = self.adj_edges.pop(old)
                for e in self.E():
                        u,v = self.edge_to_vertexpair[e]
                        if u == old:
                                self.edge_to_vertexpair[e] = (new, v)
                        if v == old:
```

```
                                self.edge_to_vertexpair[e] = (u, new)

        def rename_edge(self, old: int, new: int):
                self.edge_to_vertexpair[new] = self.edge_to_vertexpair.pop(old)
                for u,vs in self.adj_edges.items():
                        if old in vs:
                                self.adj_edges[u][vs.index(old)] = new

        def adj(self) -> dict[int, list[int]]:
                return dict([(x, self.N(x)) for x in self.adj_edges.keys()])

        def copy(self):
                H = Graph()

                adj_edges_deepcopy = dict([(u, vs.copy()) for u, vs in self.adj_edges.items()])
                edge_to_vertexpair_deepcopy = dict([(e, t) for e, t in
                ↪  self.edge_to_vertexpair.items()])

                H.adj_edges = adj_edges_deepcopy
                H.edge_to_vertexpair = edge_to_vertexpair_deepcopy
                return H

        def from_lmg(self, input_string: str):
                lines = input_string.strip().split('\n')
                N, M = map(int, lines[0].split())

                for line in lines[1:N+1]:
                        x, *ys = map(int, line.split())
                        self.adj_edges[x] = ys

                for line in lines[N+1:]:
                        e, u, v = map(int, line.split())
                        self.edge_to_vertexpair[e] = (u, v)

        def to_lmg(self) -> str:
                s = str(len(self.adj_edges)) + " " + str(len(self.edge_to_vertexpair.keys())) + "\n"
                for x, ys in self.adj_edges.items():
                        s += str(x) + " " + " ".join(map(lambda y: str(y), ys)) + "\n"
                for e, (u, v) in self.edge_to_vertexpair.items():
                        s += str(e) + " " + str(u) + " " + str(v) + "\n"
                return s

        def __str__(self):
                return self.to_lmg()

def read_lmg_from_stdin() -> str:
        N,M = map(int, input().split())
        stdin_multiline_string = f"{N} {M}\n"
        for _ in range(N+M):
                stdin_multiline_string += input() + "\n"
        return stdin_multiline_string

def read_lmg_from_file(filename: str) -> str:
        with open(filename) as f:
                return f.read()

if __name__ == "__main__":
        G = Graph()
        G.from_lmg(read_lmg_from_stdin())
        print(G)
```

11.8

medial_graph.py

```python
from Graph import Graph, read_lmg_from_stdin

# assume G is simple
# assume G is planar
# assume G_adj is a rotation system
# guarantee an edge of G have the same id as its corresponding medial node
# guarantee M is a rotation system
def medial_graph(G: Graph):
        M = Graph()
        M.adj_edges = dict((abs(n), []) for n in G.edge_to_vertexpair.keys())

        # Find faces
        edges = [e for e in G.E()]
        faces = []
        while len(edges) > 0:
                e = edges.pop()
                next_e = e
                face = [e]
                while True:
                        u,v = G.edge_to_vertexpair[next_e]
                        idx = G.adj_edges[v].index(-next_e)
                        next_e = G.adj_edges[v][(idx-1)%len(G.adj_edges[v])]
                        if (next_e == e):
                                break
                        edges.remove(next_e)
                        face.append(next_e)
                faces.append(face)

        # Add edges to medial graph
        next_linkid = 1
        for face in faces:
                for i in range(len(face)):
                        n1 = abs(face[i])
                        n2 = abs(face[(i+1)%len(face)])
                        M.adj_edges[n1].append(next_linkid)
                        M.adj_edges[n2].append(-next_linkid)
                        M.edge_to_vertexpair[next_linkid] = (n1, n2)
                        M.edge_to_vertexpair[-next_linkid] = (n2, n1)
                        next_linkid += 1
                n1 = abs(face[0])
                e1 = M.adj_edges[n1].pop()
                e2 = M.adj_edges[n1].pop()
                M.adj_edges[n1].append(e1)
                M.adj_edges[n1].append(e2)

        # Make clockwise ordering of neighbors
        for v,es in M.adj_edges.items():
                M.adj_edges[v] = list(reversed(es))

        return M, dict([(k,v) for k,v in G.edge_to_vertexpair.items() if k > 0])

if __name__ == "__main__":
        G = Graph()
        G.from_lmg(read_lmg_from_stdin())
        print(medial_graph(G))
```

11.9

test-cubic-cross-validate.py

```python
import subprocess
import threading

expected = {
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{10}-5.in": "3",
```

```
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{14}-5.in": "3",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{18}-5.in": "4",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{22}-5.in": "4",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{26}-5.in": "4",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{30}-5.in": "4",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{34}-5.in": "4",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{38}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{42}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{46}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{50}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{54}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{58}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{62}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{66}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{70}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{74}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{78}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{82}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{86}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{90}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{94}-5.in": "5",
        f"python3 branch_width.py < ./graphs/random-planar-cubic/{98}-5.in": "6"
}

# Function to run a command
def run_command(cmd):
        process = subprocess.Popen(cmd, shell=True, stdout=subprocess.PIPE, stderr=subprocess.PIPE,
        ↪  text=True)
        stdout, stderr = process.communicate()
        if stdout:
                if stdout.strip() == expected[cmd]:
                        print(f"Branch_width for {cmd}: {stdout.strip()} matches the expected output:
                        ↪  {expected[cmd]}.")
                else:
                        print(f"Branch_width for {cmd}: {stdout.strip()} does not match the expected
                        ↪  output: {expected[cmd]}")
        if stderr:
                print(f"Error in {cmd}: {stderr}")

# Run the commands asynchronously
threads = []
for cmd in expected.keys():
        thread = threading.Thread(target=run_command, args=(cmd,))
        threads.append(thread)
        thread.start()

# Wait for all threads to complete
for thread in threads:
        thread.join()
```

11.10

test-planar-graphs.py

```
import math
import os
import subprocess
from subprocess import PIPE
import time

planar_graphs_dir = "./graphs/planar2/"
program = "branch_width.py"

files = os.listdir(planar_graphs_dir)
files.sort()
```

```

def run_command(file):
        process = None
        try:
                command = f"python3.12 {program} < {planar_graphs_dir}{file}"
                process = subprocess.Popen(command, shell=True, stdout=PIPE, stderr=PIPE)
                stdout, stderr = process.communicate(timeout=2)
                if stderr:
                        return f"Error encountered in {file}: {stderr.decode().strip()}"
                return stdout.decode().strip()
        except subprocess.TimeoutExpired:
                if process:
                        process.terminate()
                        try:
                                process.wait(timeout=2)
                        except subprocess.TimeoutExpired:
                                process.kill()
                return f"Timeout in {file}"
        except Exception as e:
                if process:
                        process.terminate()
                        try:
                                process.wait(timeout=5)
                        except subprocess.TimeoutExpired:
                                process.kill()
                return f"Exception in {file}: {e}"

for file in files:
        if file <= "007-001957.in":
                continue
        st = time.time()
        response = run_command(file)
        elapsed_time = time.time() - st

        formatted_time = time.strftime("%H:%M:%S", time.gmtime(elapsed_time)) + f".{int(elapsed_time
        ↪  * 1000) % 1000:03d}"
        n = int(file.split('-')[0])
        bw_bound = math.floor(4.5 * math.sqrt(n))

        print(f"{file};{n};{response.lstrip().strip()};{bw_bound};{formatted_time}")
```

11.11

util.py

```

import sys

def read_bin_to_adj() -> dict[int, list[int]]:
        adj = dict()
        n = ord(sys.stdin.buffer.read(1))
        for i in range(1, n+1):
                adj[i] = []
        i = 1
        while i <= n:
                x = ord(sys.stdin.buffer.read(1))
                if x == 0:
                        i += 1
                        continue
                adj[i].append(x)
        return adj

def adj_from_stdin() -> dict[int, list[int]]:
        adj = dict()
        n = int(input())
        for _ in range(n):
```

```
                ys = list(map(int, input().split()))
                x = ys[0]
                adj[x] = []
                for y in ys[1:]:
                        adj[x].append(y)
        return adj

def adj_to_text(adj):
        print(len(adj))
        for v,xs in adj.items():
                print(v, *xs)

def adj_to_str(adj):
        s = str(len(adj)) + "\n"
        for v,xs in adj.items():
                s += str(v) + " " + " ".join(map(str, xs)) + "\n"
        return s

def adj_to_nx(adj):
        G = nx.MultiDiGraph()
        for v,xs in adj.items():
                G.add_node(v)
                for x in xs:
                        G.add_edge(v, x)
        return G

def adj_to_bytes(adj):
        print(chr(len(adj)), end="")
        for v,xs in adj.items():
                print("".join(map(chr, [*xs])), end="\x00")
```

11.12

graphs/gen-random-planar-cubic.py

```
# Author: Andreas Björklund
from random import random, seed
import sys

if len(sys.argv)>2:
  NVERTICES=int(sys.argv[1])
  RSEED=int(sys.argv[2])
  seed(RSEED)
elif len(sys.argv)>1:
  NVERTICES=int(sys.argv[1])
else:
  NVERTICES=6

if NVERTICES%2==1:
  sys.stderr.write("Number of vertices must be even\n")
  exit(1)
if NVERTICES<4:
  sys.stderr.write("Number of vertices must be larger than 4\n")
  exit(1)

DEPTH=(NVERTICES-4)//2
def start_instance():
  # Original graph K_4
  #   0
  #
  #   3
  #1     2
  C=[[2,3,1], [0,3,2], [1,3,0], [0,2,1]]
  return C

```

```
# Generate random cubic 3-connected planar graph
def random_instance(C):

  # Triangle
  D=[[0,4,5],[1,5,3],[2,3,4]]

  # Single Bridge
  # 041
  # 253
  F=[[0,1,5],[2,4,3]]

  B = C[:]

  for rounds in range(1):
    n=len(B)
    who = int(random()*n)

    nx = int(random()*3)
    cycle=[who]
    while B[who][nx]!=cycle[0]:
      cycle.append(B[who][nx])
      ix=[x for x in range(3) if B[B[who][nx]][x]==who]
      ix=(ix[0]+2)%3
      who = B[who][nx]
      nx = ix

    m=len(cycle)
    p = int(random()*(m-3))+2;
    T = cycle[:2] + cycle[p:p-2:-1]
    G = B[:]
    H = [(list(map(sum, zip(x, [n-4,n-4,n-4])))) for x in F]
    for i in range(4):
      for ix2 in range(3):
        if B[T[i]][ix2]==T[i^1]:
          G[T[i]]= (G[T[i]][:ix2]+[n+(i // 2)]+G[T[i]][ix2+1:])
      for j in range(len(F)):
        for k in range(3):
          if F[j][k]==i:
            H[j] = (H[j][:k]+[T[i]]+H[j][k+1:])
    B=G+H

  return B

def gen_instance():
  C=start_instance()
  for i in range(DEPTH):
    C=random_instance(C)
  return C

C=gen_instance()
# for i in range(len(C)):
#   print(" ".join(map(str,C[i])))

D=dict([x+1, list(map(lambda x: x+1, C[x]))] for x in range(len(C)))
print(len(D.items()))
for x,ys in D.items():
        print(x, " ".join(map(str,ys)))
```

11.13

graphs/gen-planar.py

```
import os

# ns = [
```

```
#         2, 3, 4, 5, 6,
# ]
ns = [
        7
]

for n in ns:
        os.system(f"./plantri.bin -pm1c1 {n} | python3 process_planar_code.py planar2")
```

11.14

graphs/process_planar_code.py

```
import sys
import os

def read_bin_to_adj() -> dict[int, list[int]]:
        adj = dict()
        n = ord(sys.stdin.buffer.read(1))
        for i in range(1, n+1):
                adj[i] = []
        i = 1
        while i <= n:
                x = ord(sys.stdin.buffer.read(1))
                if x == 0:
                        i += 1
                        continue
                adj[i].append(x)
        return adj

def adj_to_str(adj):
        s = str(len(adj)) + "\n"
        for v,xs in adj.items():
                s += str(v) + " " + " ".join(map(str, xs)) + "\n"
        return s

# skip header
byte_count = 0
while byte_count < 15:
        byte_s = sys.stdin.buffer.read(1)
        if not byte_s:
                break
        byte_count += 1

# read graphs
graph_idx = 1
while True:
        adj = read_bin_to_adj()

        folder_name = sys.argv[1]

        # create file if not exists
        if not os.path.exists(f"./{folder_name}"):
                os.makedirs(f"./{folder_name}")

        with open(f"./{folder_name}/{len(adj.keys()):03}-{graph_idx:06}.in", "w") as outfile:
                outfile.write(adj_to_str(adj))
        graph_idx += 1
```

11.15