

Implementation of a minimal branch-decomposition algorithm for planar graphs.

Kristoffer Højelse

February 2024

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

Some graph optimization problems can be solved efficiently for graphs of small branchwidth.[4]

Pino[2] applies branch decompositions.

Seymour and Thomas[1] give the rat-catching algorithm.

Bian, Gu and Zhu[3] describe and benchmark some implementations.

which? counting Hamiltonian cycles of planar cubic graphs

2 Preliminaries

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

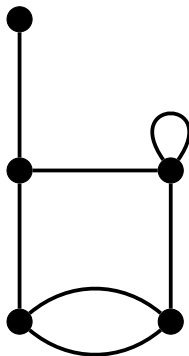
Note. Other authors might instead call this definition an undirected labelled pseudograph, with edges having own identity.

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

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 .

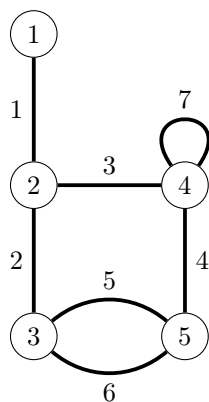
G



Here is a labeled drawing of the same graph G and its function ϕ_G .

G

ϕ_G



- 1: $\{1, 2\}$
- 2: $\{2, 3\}$
- 3: $\{2, 4\}$
- 4: $\{4, 5\}$
- 5: $\{3, 5\}$
- 6: $\{3, 5\}$
- 7: $\{4, 4\}$

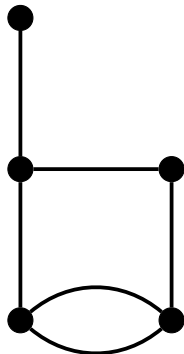
Let $E(G)$ return a multiset of all vertex-pairs of G ; in other words, $E(G) = \{\phi_G(e) : e \in \mathbb{E}(G)\}$.

A *self-loop*, is an edge e where $\phi_G(e) = \{u, v\}$ and $u = v$.

A graph G is *loop-less*, if no edge $e \in \mathbb{E}(G)$ is a self-loop. The edge with label 7, in the graph above, is an example of a self-loop.

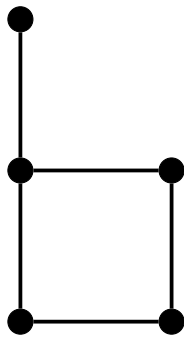
A *multi-graph* is a graph that is loop-less.

A multi-graph G

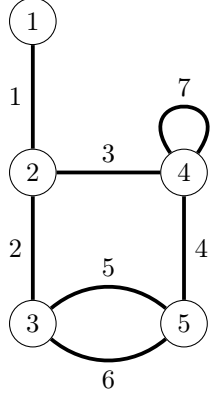
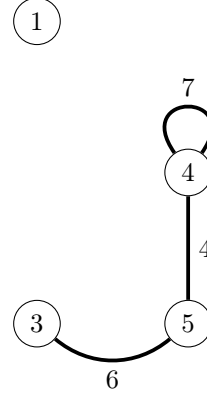


A multi-graph G is *simple*, if it has no parallel edges; in other words, if elements of $E(G)$ are pair-wise distinct.

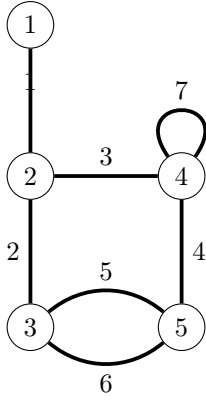
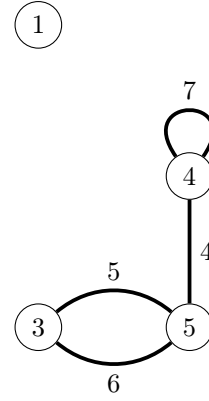
A simple graph G



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 $E(H) \subseteq E(G)$ and where $\forall e \in E(H), \phi_H(e) = \phi_G(e)$.

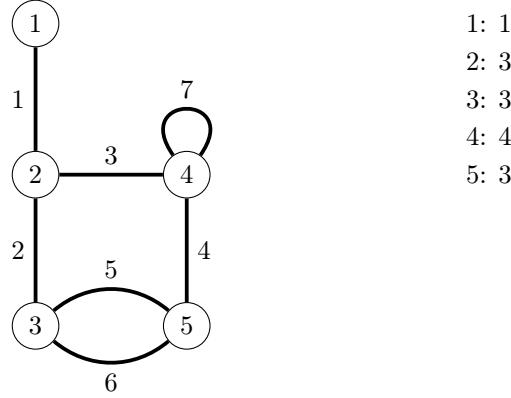
G  H 

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 $\mathbb{E}(G[A]) = \{e : e \in \mathbb{E}(G) \wedge |\phi_G(e) \cap A| = 2\}$ 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 $\deg(v)$ 

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

$$\Delta(G) = 4$$

A *walk* of a graph G is a list $[v_0, e_1, v_1, \dots, e_k, v_k]$ where $v_0, v_1, \dots, v_k \in V(G)$ 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 .

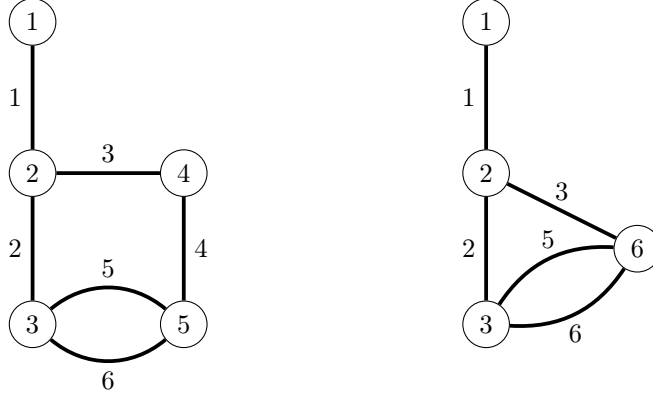
A *rotation system* is an encoding of a graph, in particular, it is an adjacency list such that the neighborhood, around any vertex, is in clockwise ordering according to some plane embedding of G .

Definition 2.1. (*Contraction*)

A contraction is a function that given a multi-graph G and pair of distinct vertices $u, v \in V(G)$ such that $\{u, v\} \in E(G)$, then for all edges $e \in \mathbb{E}(G)$ if $\phi_G(e) = \{u, v\}$ then removes e else if $\phi_G(e) = \{v, w\}$ then $\phi_G(e) = \{u, w\}$, and finally returns the resulting graph.

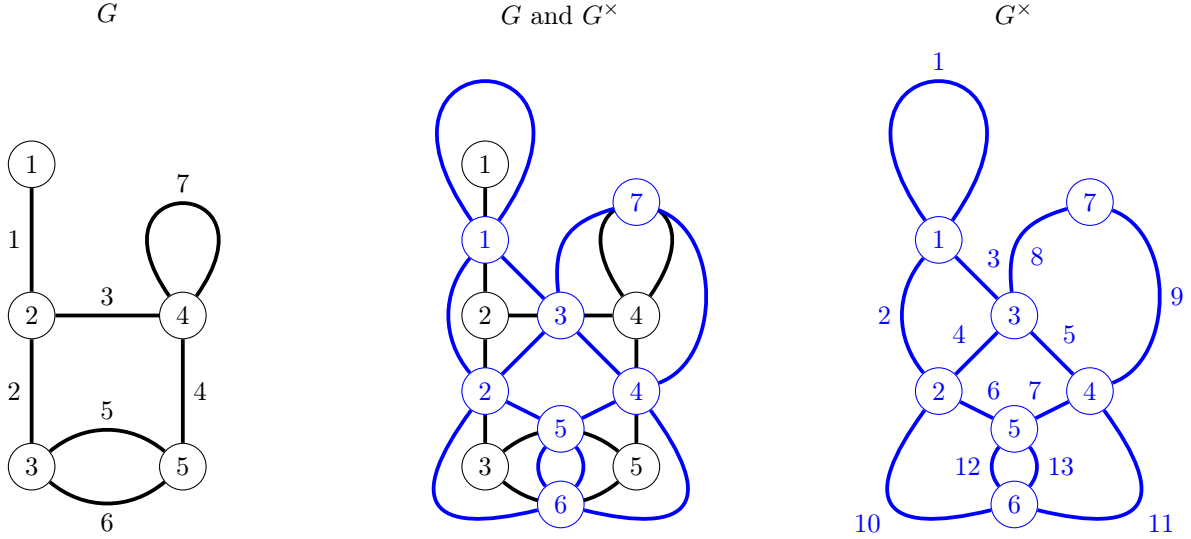
G

Contraction of G given vertices 4 and 5



Definition 2.2. (*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 .

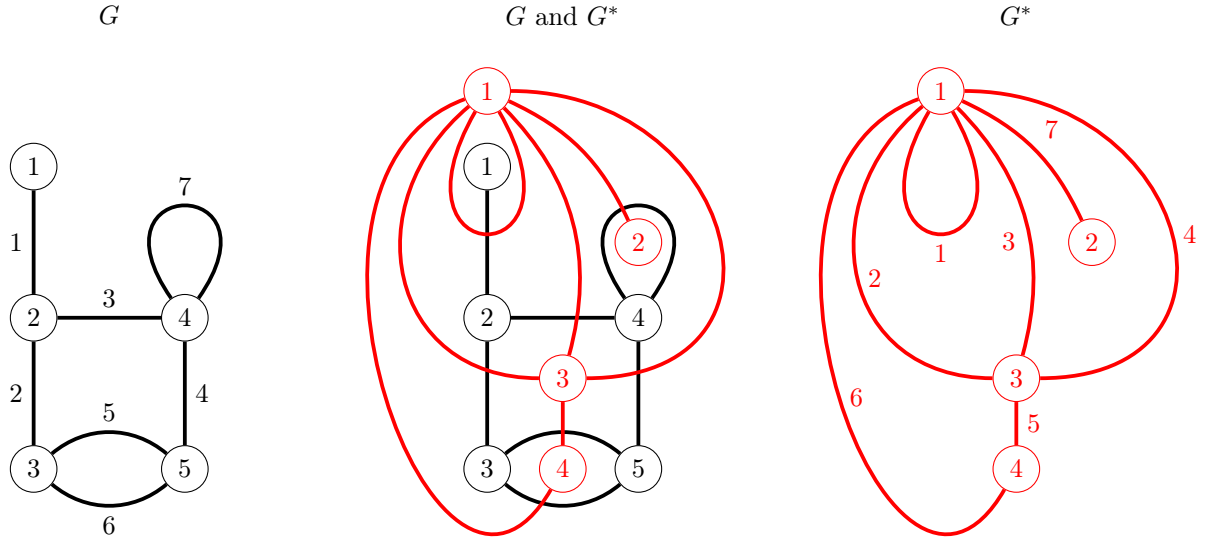


Corollary 2.3. A medial graph is a 4-regular plane graph.

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

Definition 2.4. (*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.

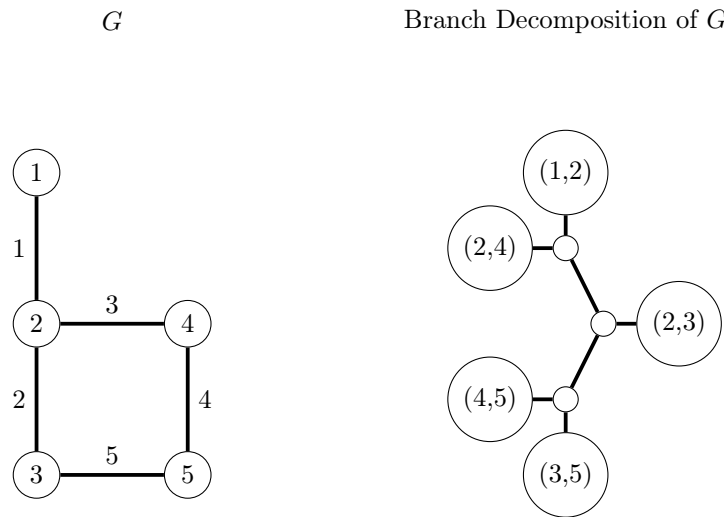
A *leaf* v of a tree T , is a vertex $v \in V(T)$ of degree 1.

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 degree at least 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 $L(P_e) \cap L(Q_e)$ is called

a *middle set* of B_G given e , denoted $Z(B_G, e)$. The maximal cardinality of any middle set of B_G given any e of B_G is the width of B_G ; in other words, the width of B_G is $\max\{|Z(B, e)| : e \in \mathbb{E}(B)\}$.

There might exist many branch decompositions of a graph G .

A *Minimal Branch Decomposition* of G is any branch decomposition of G of minimal width among all branch decompositions of 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 $L(P_e) \cap L(Q_e)$ is called a *middle set* of C_G given e , denoted $Z(C_G, e)$. The maximal cardinality of any middle set of C_G given any e of C_G is the width of C_G ; in other words, the width of C_G is $\max\{|Z(C_G, e)| : e \in \mathbb{E}(C_G)\}$.

There might exist many carving decompositions of a graph G .

A *Minimal Carving Decomposition* of G is any carving decomposition of G of minimal width among all carving decompositions of G .

3 Overview of the algorithm

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

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

To compute the minimal 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 minimal carving decomposition of G^\times .

To compute the carving width of G , the algorithm uses the rat-catching algorithm. By an analogy, the rat-catching algorithm can be described by a game of two players, the rat and rat-catcher, who take turns moving around on G . The game is played to determine, 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 or if the rat can escape indefinitely. Larger k makes the rat-catching game easier for the rat-catcher. The smallest k where the rat can escape indefinitely is the carving width of G .

To compute a noisy subgraph of G given a noise level k and a rat-catchers position, the algorithm considers the dual graph of G 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.

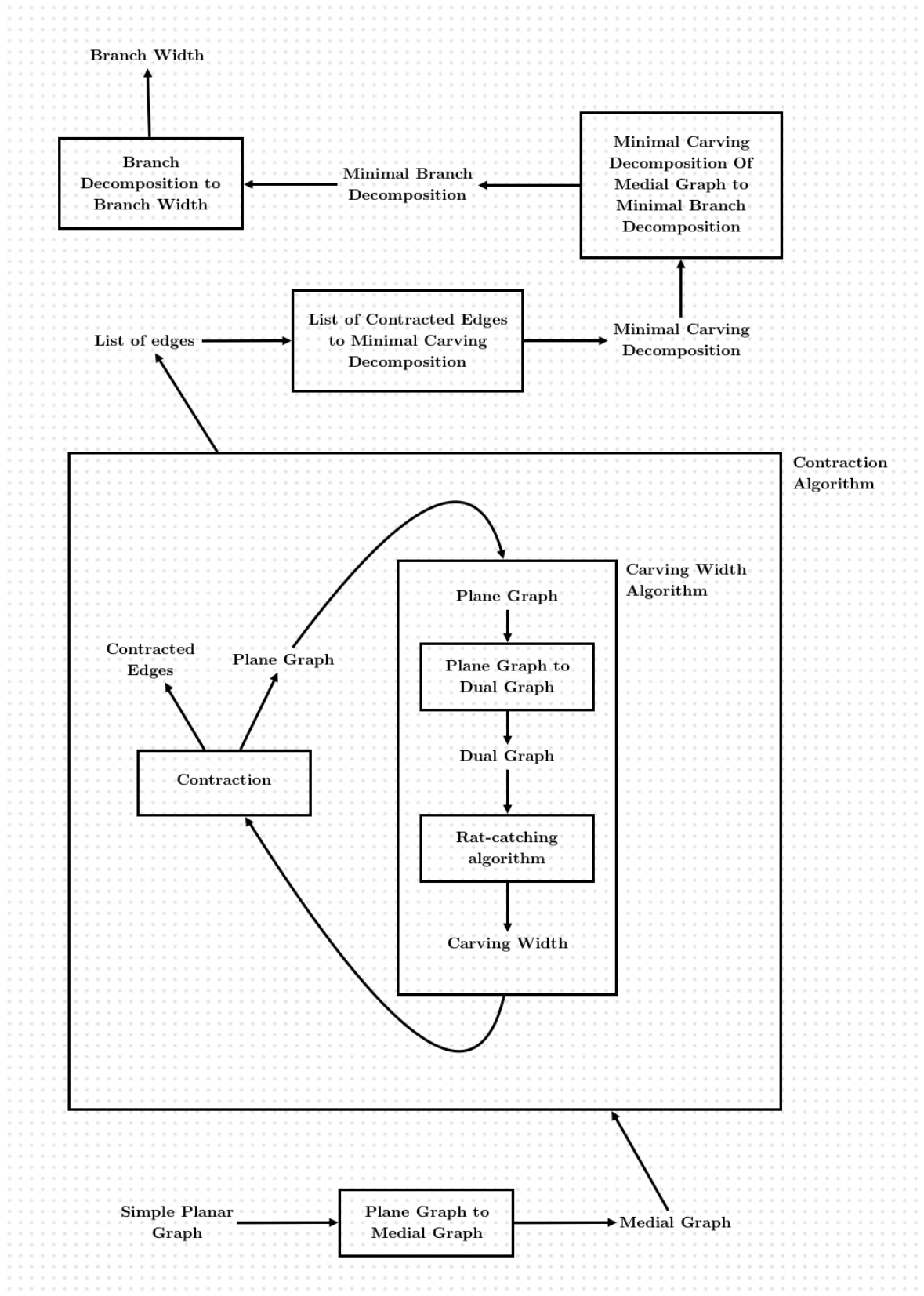


Figure 1: Overview of the algorithm

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 edges and a map from unique edge IDs its vertex-pair.

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

```
1 class Graph:
2     def __init__(self):
3         self.adj_edges: dict[int, list[int]] = dict()
4         self.edge_to_vertexpair: dict[int, tuple[int, int]] = dict()
5         pass
6
7     def from_adj(self, adj: dict[int, list[int]]):
8         # assign edge ids
9         adj_deepcopy = dict([(u, vs.copy()) for u, vs in adj.items()])
10        self.adj_edges = adj_deepcopy
11        next_edgeid = 1
12        for x, ys in self.adj_edges.items():
13            for i, y in enumerate(ys):
14                if x < y:
15                    self.edge_to_vertexpair[next_edgeid] = (x, y)
16                    self.edge_to_vertexpair[-next_edgeid] = (y, x)
17
18                    self.adj_edges[x][i] = next_edgeid
19                    self.adj_edges[y][adj[y].index(x)] = -next_edgeid
20
21                next_edgeid += 1
22
23        def V(self) -> list[int]:
24            return list(self.adj_edges.keys())
25
26        def E(self) -> list[int]:
27            return list(self.edge_to_vertexpair.keys())
28
29        def N(self, v: int) -> list[int]:
30            return [self.edge_to_vertexpair[e][1] if self.edge_to_vertexpair[e][0] == v else
31                    ↪ self.edge_to_vertexpair[e][0] for e in self.adj_edges[v]]
32
33        def adj(self) -> dict[int, list[int]]:
34            return dict([(x, self.N(x)) for x in self.adj_edges.keys()])
```

5 The algorithm

The main computational problem of this paper is THE PLANAR MINIMAL BRANCH DECOMPOSITION PROBLEM.

Definition 5.1. THE PLANAR MINIMAL BRANCH DECOMPOSITION PROBLEM

Input: Given a simple connected planar graph G .

Output: A minimal branch decomposition of G .

The algorithm described in this paper solves THE PLANAR MINIMAL 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 minimal branch decomposition (B_G, δ_G) of G from a minimal 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 vertex-pairs $E(G)$.

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

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

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

branch_decomposition.py

```

6  # Construct a branch decomposition of a graph
7  def branch_decomposition(G_adj: dict[int, list[int]]):
8      # Construct the carving decomposition of the medial graph
9      Gx, node_to_vertpair = medial_graph(G_adj)
10
11      cd = carving_decomposition(Gx.copy())
12
13      # Convert the carving decomposition of M to a branch decomposition of G
14      def decomp(t):
15          if isinstance(t, int):
16              return node_to_vertpair[t]
17          return tuple([decomp(a) for a in t])
18
19      bd = decomp(cd)
20
21      log.add("Branch decomposition: " + str(bd))
22      return bd

```

Here are three sections, one for each subproblem.

5.1 Medial graph

Solving problem 5.2.

Recall the definition 2.2.

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.

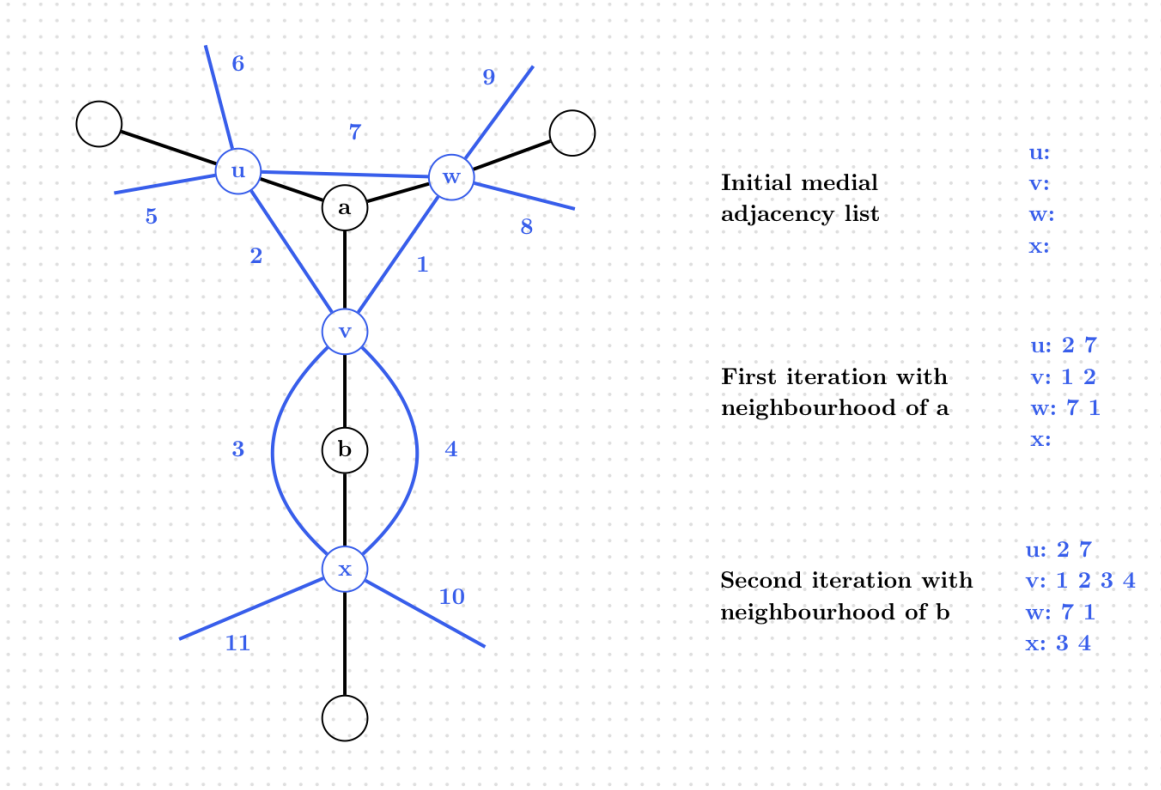


Figure 2: Two iterations of the algorithm computing a medial rotation system

A medial graph is 4-regular; every node has degree 4. From the perspective of some medial node v , in some single iteration of the loop on line 14, two links are added to the neighborhood of v in counterclockwise ordering, and later the two other links are added to the neighborhood of v also in counterclockwise ordering. Therefore the medial graph M is a rotation system.

medial_graph.py

```

5  # assume G is simple
6  # assume G is planar
7  # assume G_adj is a rotation system
8  # guarantee M is a rotation system
9  def medial_graph(G_adj: dict[int, list[int]]) -> Graph:
10     vertexpairs = set([tuple(sorted((i, j))) for i in G_adj for j in G_adj[i]])
11
12     vertexpair_to_node = dict([(e, i+1) for i,e in enumerate(vertexpairs)])
13     node_to_vertexpair = dict([(i+1, e) for i,e in enumerate(vertexpairs)])
14
15     medial = dict([(i+1, []) for i in range(len(vertexpairs))])
16
17     for u,vs in G_adj.items():
18         nodes = [vertexpair_to_node[tuple(sorted((u, v)))] for v in vs]
19         for i in range(len(nodes)):
20             medial[nodes[i]].append(nodes[(i-1)%len(nodes)])
21             medial[nodes[i]].append(nodes[(i+1)%len(nodes)])
22
23     M = Graph()
24     M.from_adj(medial)
25     # log.add("Medial graph: " + str(M))
26     # log.add("Node to vertexpair: " + str(node_to_vertexpair))
27     return M, node_to_vertexpair

```

5.2 Minimal Branch Decomposition from Minimal Carving Decomposition

Solving problem 5.4.

For both branch- and carving-decompositions, I have chosen a data structure of tuples of either more tuples or integers. This has a straightforward translation to the Newick Tree Format, a concise notation for tree structures.

To then solve the above-mentioned problem, the implementation recursively returns either returns a copy of any tuple it encounters or returns a tuple of integers for any integer in encounters, using the relation `node_to_vertexpair`.

This effectively replaces the leaf nodes of the carving decomposition with vertex-pairs for the branch decomposition.

`branch_decomposition.py`

```
5
6 # Construct a branch decomposition of a graph
7 def branch_decomposition(G_adj: dict[int, list[int]]):
8     # Construct the carving decomposition of the medial graph
9     Gx, node_to_vertexpair = medial_graph(G_adj)
10
11     cd = carving_decomposition(Gx.copy())
12
13     # Convert the carving decomposition of M to a branch decomposition of G
14     def decomp(t):
15         if isinstance(t, int):
16             return node_to_vertexpair[t]
17         return tuple([decomp(a) for a in t])
18
19     bd = decomp(cd)
```

5.3 Minimal Carving Decomposition

To solve 5.3 ?? gives a contraction algorithm.

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 three vertices, can be assembled into a minimal 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 either u or v), therefore by keeping track of which vertex is a contraction of which vertex-pair, in say 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 in the aforementioned Newick-like nested tuple format.

`carving_decomposition.py`

```
7 # Find a contraction that does not increase the carving width
8 def nonincreasing_cw_contraction(G: Graph, cw1: int) -> tuple:
9     for e in G.E():
10         u, v = G.edge_to_vertexpair[e]
11         G2, w = contraction(G, u, v)
12         cw2 = carving_width(G2)
13         if cw2 <= cw1:
14             log.add(f"Graph after contracting edge {e}: \n{str(G2)}")
```

```

15         return G2, (u, v), cw2, w
16     return None, None, None, None
17
18     # Contract edges that do not increase the carving width
19     # until only 3 vertices remain.
20     # Return the resulting graph and the edges that were contracted
21     def gradient_descent_contractions(G: Graph) -> Graph:
22         G2 = G.copy()
23         cw1 = carving_width(G)
24         edges = dict()
25         while True:
26             G3, uv, cw2, w = nonincreasing_cw_contraction(G2, cw1)
27             if G3 is not None and len(G3.V()) >= 3:
28                 G2 = G3
29                 cw1 = cw2
30                 edges[w] = uv
31             if len(G2.V()) == 3:
32                 return G2, edges
33
34     # Construct a carving decomposition of a graph
35     def carving_decomposition(G: Graph) -> tuple:
36         G2, edges = gradient_descent_contractions(G)
37
38         # Construct the decomposition from the edges that were contracted
39
40         def decomp(x):
41             if x not in edges:
42                 return x
43             a, b = edges[x]
44             return (decomp(a), decomp(b))
45
46         a, b, c = G2.V()
47         cd = (decomp(a), decomp(b), decomp(c))
48
49         log.add("Carving decomposition: " + str(cd))
50         return cd

```

The contraction algorithm depends on a function to compute a contraction and a function to compute the carving width of a graph. This is 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.

As the resulting graph is later given as an argument to functions assuming a rotation system of a planar graph, the implementation needs to preserve this invariant when contracting.

Recall the definition 2.1. As this contraction removes all edges connecting a pair of vertices, the resulting graph will not exhibit any 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 way less practical.

contraction.py

```

4  # assume G might have parallel edges
5  # assume G do not have self-loops
6  # assume adjacency list of G has clockwise ordering of neighbors
7  def contraction(G: Graph, a: int, b: int) -> Graph:
8      # copy G
9      G1 = G.copy()
10
11     # create new vertex c
12     c = max(G1.adj_edges.keys()) + 1
13
14     # let every edge incident to a or b be incident to c instead
15     for e in G1.E():
16         u,v = G1.edge_to_vertexpair[e]
17         if u == a or u == b:
18             G1.edge_to_vertexpair[e] = (c, v)
19         u,v = G1.edge_to_vertexpair[e]
20         if v == a or v == b:
21             G1.edge_to_vertexpair[e] = (u, c)
22
23     # create neighborhood of c
24     def index_of_first(lst, pred):
25         for i, v in enumerate(lst):
26             if pred(v):
27                 return i
28         return None
29
30     index_of_first_shared_edge = index_of_first(G1.adj_edges[a], lambda e:
31     ↪ G1.edge_to_vertexpair[e][0] == c and G1.edge_to_vertexpair[e][1] == c)
32     first_shared_edge = G1.adj_edges[a][index_of_first_shared_edge]
33
34     idx1 = G1.adj_edges[a].index(first_shared_edge)
35     rotated_Ga = G1.adj_edges[a][idx1:] + G1.adj_edges[a][:idx1]
36
37     idx2 = G1.adj_edges[b].index(-first_shared_edge)
38     rotated_Gb = G1.adj_edges[b][idx2:] + G1.adj_edges[b][:idx2]
39
40     G1.adj_edges[c] = rotated_Ga + rotated_Gb
41
42     # remove self-loops on c
43     G1.adj_edges[c] = [e for e in G1.adj_edges[c] if not (G1.edge_to_vertexpair[e][0] ==
44     ↪ G1.edge_to_vertexpair[e][1] == c)]
45     G1.edge_to_vertexpair = dict([(k,v) for k,v in G1.edge_to_vertexpair.items() if not (v[0] ==
46     ↪ v[1] == c)])
47
48     # remove a and b
49     del G1.adj_edges[a]
50     del G1.adj_edges[b]
51
52     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 $k = \{1, 2, 3, \dots\}$ 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

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

carving_width.py

```
132 def binary_search_cw():
133     l = 0
134     r = 1
135     while True:
136         if rat_wins(r):
137             l = r
138             r *= 2
139         else:
140             break
141     m = 1
142     while l < r:
143         m = int(math.ceil((l + r) / 2))
144         if rat_wins(m):
145             l = m
146         else:
147             r = m - 1
148     return l
149
150 def linear_search_cw():
151     k = 0
152     while rat_wins(k):
153         k += 1
154     return k - 1
155
156 cw = binary_search_cw()
157 return cw
158
```

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 ???. This is discussed in section ???.

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 strictly 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 strictly less than k containing f^* and e^* in the dual M^* .

A quiet subgraph $Q(M, k, e)$, for some graph M , some noise level k and some $e \in \mathbb{E}(M)$, 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 states/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 tuple $(e, C) \in T$, if all (f, v) where $v \in V(C)$ is losing then (e, C) is losing.

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

carving_width.py

```

78     def flatten(xss):
79         return set([x for xs in xss for x in xs])
80

```

```

81     # Assume |V(G)| >= 2
82     # Return True
83     # iff. carving-width >= k
84     # iff. rat has a winning escape strategy with noise-level k
85     def rat_wins(k: int) -> bool:
86         if len(G.V()) < 2:
87             return False
88
89         if max([len(G.N(v)) for v in G.V()]) >= k:
90             return True
91
92         # Set up the game states
93         halfedges = edge_to_link.keys()
94
95         T = set([(e, tuple(C)) for e in halfedges for C in quiet_components(e, k)])
96         S = set([(f, v) for f in node_to_face.keys() for v in G.V()])
97
98         # Set up the losing states
99         losing_T = set()
100        losing_S = set()
101
102        for (f, v) in S:
103            if v in flatten([G.edge_to_vertexpair[e] for e in node_to_face[f]]):
104                losing_S.add((f,v))
105
106        if len(T) == len(losing_T) or len(S) == len(losing_S):
107            return False
108
109        # Play the game
110        while True:
111            new_deletion = False
112
113            for (e, C) in T:
114                if all([(edge_to_node[e], v) in losing_S for v in C]):
115                    if (e, C) not in losing_T:
116                        new_deletion = True
117                        losing_T.add((e, C))
118
119            for (e, C) in losing_T:
120                f1 = edge_to_node[e]
121                f2 = edge_to_node[-e]
122                for (f, v) in [(f1, v) for v in C] + [(f2, v) for v in C]:
123                    if (f, v) not in losing_S:
124                        new_deletion = True
125                        losing_S.add((f, v))
126
127            if len(T) == len(losing_T) or len(S) == len(losing_S):
128                return False
129            elif not new_deletion:
130                return True

```

6.0.1 Quiet subgraph

Solving problem 6.3.

Recall the definition 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 in the natural way.

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

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

```

62         while len(unseen) > 0:
63             v = unseen.pop()
64             component = [v]
65             stack = [v]
66             while len(stack) > 0:
67                 v = stack.pop()
68                 for e1 in quiet_subgraph[v]:
69                     u,v = G.edge_to_vertexpair[e1]
70                     if v in unseen:
71                         unseen.remove(v)
72                         stack.append(v)
73                         component.append(v)
74             components.append(component)
75
76     return components

```

6.0.2 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, and then marking all the edges of the face it belongs to.

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     # Assume G is a rotation system
5     def dual_graph(G: Graph) -> Graph:
6         edges = [e for e in G.E()]
7
8         D = Graph()
9         edge_to_link = dict()
10        link_to_edge = dict()
11        node_to_face = dict() # nodeid to edgeid list
12        edge_to_node = dict() # half-edge to the faceid/node to its either left/right
13
14        # Find faces
15        next_nodeid = -1
16        while edges:
17            e = edges.pop()
18            next_e = e
19            edge_to_node[e] = next_nodeid
20            face = [e]
21            while True:
22                u,v = G.edge_to_vertexpair[next_e]
23                idx = G.adj_edges[v].index(-next_e)
24                next_e = G.adj_edges[v][(idx-1)%len(G.adj_edges[v])]
25                if (next_e == e):
26                    break
27                edges.remove(next_e)
28                face.append(next_e)
29                edge_to_node[next_e] = next_nodeid
30            node_to_face[next_nodeid] = face
31            next_nodeid -= 1
32
33        for i in node_to_face.keys():
34            D.adj_edges[i] = []
35
36        # Add edges to dual graph
37        next_linkid = 1

```

```

38     for i,f1 in node_to_face.items():
39         for j,f2 in node_to_face.items():
40             if i < j:
41                 common_edges = set(list(map(abs, f1))).intersection(set(map(abs, f2)))
42                 for e in common_edges:
43                     D.edge_to_vertexpair[next_linkid] = (i, j)
44                     D.edge_to_vertexpair[-next_linkid] = (j, i)
45                     edge_to_link[e] = next_linkid
46                     link_to_edge[next_linkid] = e
47                     edge_to_link[-e] = -next_linkid
48                     link_to_edge[-next_linkid] = -e
49                     D.adj_edges[i].append(next_linkid)
50                     D.adj_edges[j].append(-next_linkid)
51                     next_linkid += 1
52
53     # todo make edge_to_link and link_to_edge redundant
54     # by nameing edges and links the same
55
56     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 at <https://github.com/hojelse/thesis>.

The implementation has been tested on a set of graphs, and the results are as expected. [more](#)

8 References

References

- [1] “Call Routing and The Ratcatcher”. In: ().
- [2] “Cut and Count Representative Sets on Branch Decompositions”. In: ().
- [3] “Practical algorithms for branch-decompositions of planar graphs”. In: ().
- [4] “Solving connectivity problems parameterized by treewidth in single exponential time (cut and count)”. In: ().

9 Appendix