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

Kristoffer Høje lse

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 $\phi_G\colon\ \mathbb{E}(G) \to \{\{u, v\}\colon\ u, v \in V(G)\}$.

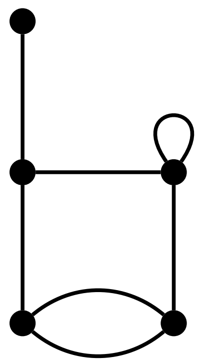
Note. Other authors might instead call this definition an undirected labelled pseudograph, with edges having own identity.

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.

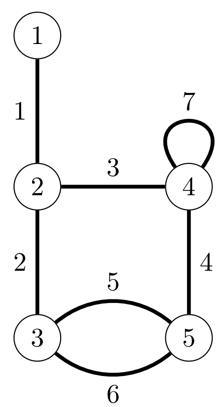
Here is a drawing of a graph G.

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}

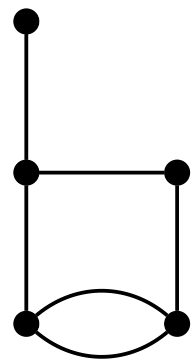
Let $E(G)$ return a multiset of all vertex-pairs of G; in other words, $E(G) = \{\phi_G(e)\colon\ e \in \mathbb{E}(G)\}$.

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

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

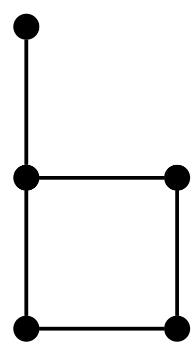
A *multi-graph* is a graph that is loop-less.

A multi-graph G



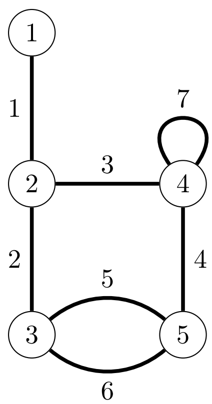
A multi-graph G is *simple*, if it has no parallel edges; in other words, if elements of $E(G)$ are pair-wise distinct.

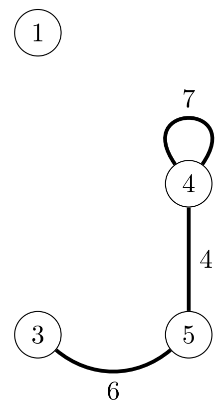
A simple graph G



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

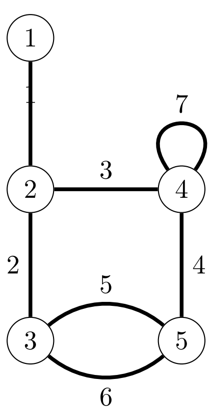
G H



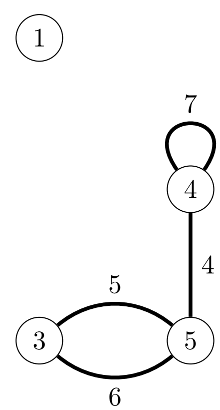


For $A \subseteq V(G)$, we denote by $G[A]$ the subgraph induced by the subset of vertices A; in other words, $G[A]$ is the subgraph where $V(G[A]) = A$ and where $\mathbb{E}(G[A]) = \{e\colon\ e \in \mathbb{E}(G) \wedge |\phi_G(e) \cap A| = 2\}$ and where $\forall e \in E(G[A]), \phi_{G[A]}(e) = \phi_G(e)$.

G A $G[A]$



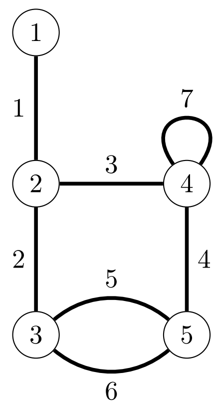
$\{1, 3, 4, 5\}$



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 $\quad\quad$ $\deg(v)$



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

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

A *walk* of a graph G is a list of edges, such that consecutive edges in the list are incident to each other.

An s,t-*walk* is a walk, such that the first edge is incident to the vertex s and such that the last edge is incident to the vertex t.

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

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

A *path* of a graph G, is a walk such that only consecutive edges of the walk are incident.

A *cycle* of a graph G, is a path but where the first and the last edge are incident to each other.

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* is a relation between two sets such that each element of either set is paired with exactly one element of the other set.

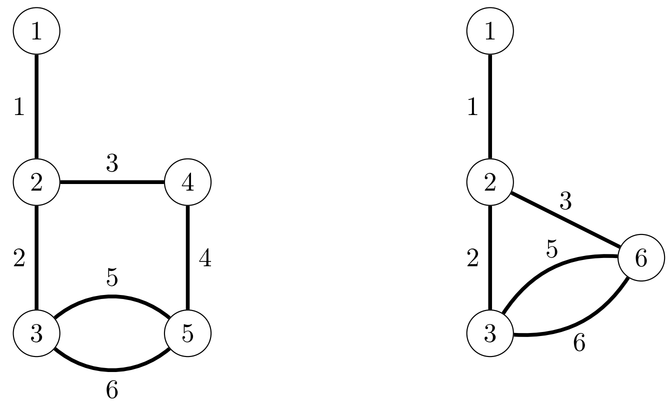
A *plane graph* is a drawing of a graph, such that no edges are crossing.

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

Definition 2.1. (*Contraction*)

A contraction is a function that given a multi-graph G and pair of 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.

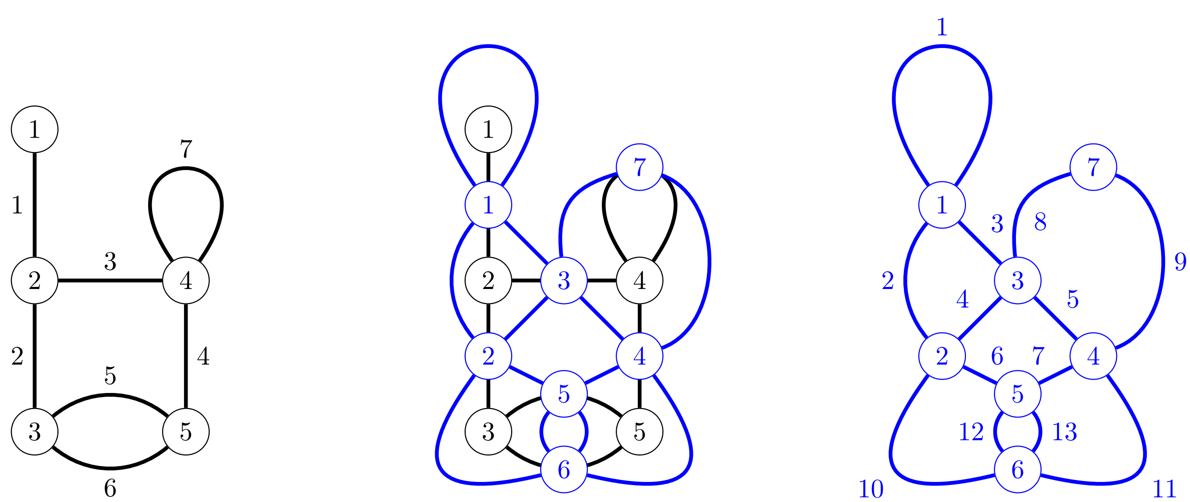
G Contraction of G given vertices 4 and 5



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.

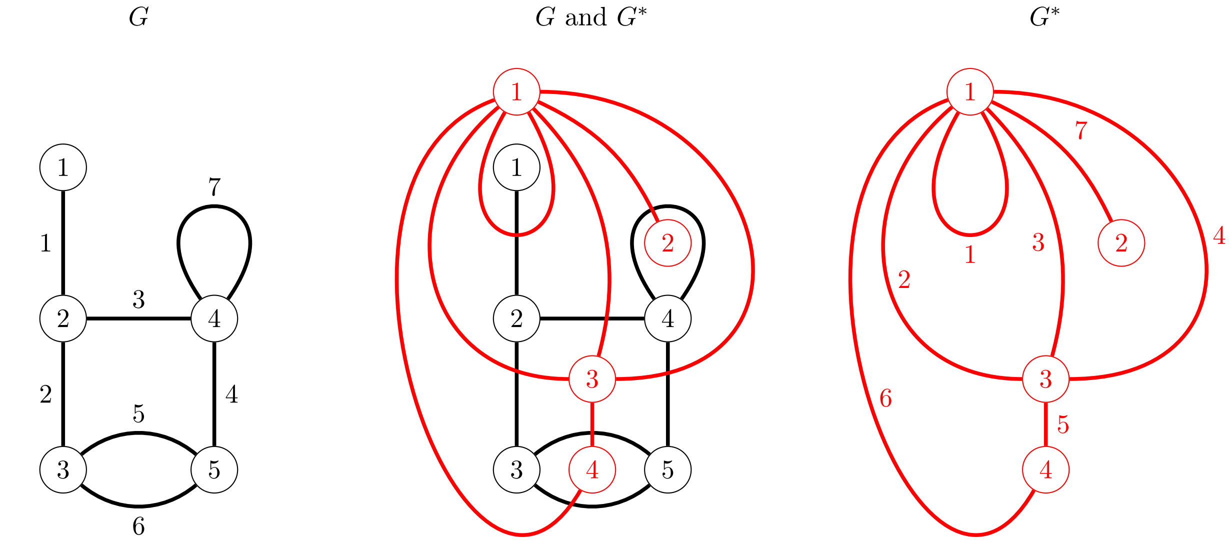
G G and $G^\times$ $G^\times$



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

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.

An *internal vertex* v of a tree T, is a vertex $v \in V(T)$ of degree at least 2.

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

A *Branch Decomposition* (B_G, δ_G) of a graph G consists of firstly, a tree B_G where every internal vertex of B_G has degree 3; in other words, B_G is an unrooted binary tree, 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 $\max\{|Z(B,e)| \colon e \in \mathbb{E}(B)\}$.

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 graph G consists of firstly, a tree 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 $\max\{|Z(C_G,e)| \colon e \in \mathbb{E}(C_G)\}$.

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 Description of the problem

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

> **Definition 3.1.** The Planar Minimal Branch Decomposition Problem
>
> Input: Given a simple connected planar graph G.
>
> Output: A minimal branch decomposition of G.

Here are some informal definitions to unpack the aforementioned properties of graphs.

> **Definition 3.2.** The Minimal Branch Decomposition Problem
>
> Input: Given a simple connected graph G.
>
> Output: A minimal branch decomposition of G.

The algorithm described in this paper solves The Planar Minimal Branch Decomposition Problem, which can be computed in polynomial time.

The width of a minimal branch decomposition of G is called the branch width of G.

4 The algorithm

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 3.2 is the overarching problem, and can be broken down into many smaller subproblems.

Considering a 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. Do this by replacing the vertices in leaves of the carving decomposition with edges, using the mapping between edges and vertices from computing the medial graph;

Therefore problem 3.2 break down into problems 4.1, 4.2 and 4.3.

> **Problem 4.1.** Given a minimal carving decomposition of a medial graph of G, output a minimal branch decomposition of G.

> **Problem 4.2.** Given a graph G, output a medial graph and a bijectional mapping between medial nodes and vertex pairs.

> **Problem 4.3.** Given a plane graph M and function to compute the carving width of a graph, output a minimal carving decomposition of M.

Implementing a function to solve 4.1 is described in 5.1.

To solve 4.2 is a matter of following the definition.

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

Computing a medial graph is described in 5.2.

To solve 4.3 **??** gives a contraction algorithm.

Doing a series of edge contractions (contraction of all edges incident to a pair of vertices) 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.

I will defer describing exactly how to assemble a minimal carving decomposition to 5.3.

The contraction algorithm depends on a function to compute a contraction and a function to compute the carving width. This is problems 4.4 and 4.5.

Problem 4.4. Given a graph M and a pair of vertices $\{u, v\}$, output a graph resulting from a contraction of all edges incident to u and v if u and v are neighbors.

Problem 4.5. Given a plane graph M that might have parallel edges, output the carving width of M.

First consider problem 4.4.

Informally; a contraction is merging two vertices into one and letting incident edges connect to the new vertex. The contraction in this paper differs a bit from conventional definitions of edge- and vertex-contractions, as other definitions might result in self-loops when contracting one of multiple parallel edges.

Computing a medial graph is described in 5.2.

Now consider problem 4.5.

The rat-catching algorithm decides $cw(M) \geq k$ with k being in the positive integers. This is a monotonic boolean space, so you can perform a binary search to find the smallest k where $cw(M) \geq k$ is true.

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 if they can force the rat to be in some wall of the room that they are in, with noise level k, and the rat wins 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 assuming $\Delta(M) < k$ the game is played.

For some noise level and location of the rat-catcher, exactly which edges are noisy and which are quiet are definitions 4.6 and 4.7.

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

Definition 4.6. 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^*.

Definition 4.7. When the rat-catcher is on some room 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, some noise level k and some $e \in \mathbb{E}(M)$, is a subgraph of M with $V(Q(M, k, e)) = V(M)$ and

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

> **Problem 4.8.** 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 4.8 depends on a function for computing the dual of a graph. Computing a dual graph is problem 4.9.

> **Problem 4.9.** 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. $S = \{(f, v) \colon\ v \in V(M) \wedge f \text{ is a face of } M\}$.

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

Computing the quiet subgraph requires the dual graph.

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

You can mark states 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.

5 The implementation

For the upcoming problems one needs to deal with parallel edges and be able to tell them apart, therefore the implementation uses a data structure that encapsulates an adjacency list of edges and a map from unique edge ids its vertexpair.

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

```
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
		self.adj_edges = adj.copy()
		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] for e in self.adj_edges[v]]

    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()
        H.adj_edges = self.adj_edges.copy()
        H.edge_to_vertexpair = self.edge_to_vertexpair.copy()
        return H
```

5.1 Computing a minimal branch decomposition

Solving problem 4.1.

For both branch- and carving-decompositions, I have chosen a data structure of tuples of 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 a copy of any tuple, but returns a tuple of integers for any integer, using the mapping from medial node to vertex pair.

branch_decomposition.py

```
# Construct a branch decomposition of a graph
def branch_decomposition(G_adj: dict[int, list[int]]):
    # Contruct the carving decomposition of the medial graph
    M, node_to_vertexpair, vertexpair_to_node = medial_graph(G_adj)
    cd = carving_decomposition(M)

    # Convert the carving decomposition of M to a branch decomposition of G
    def decomp(t):
        if isinstance(t, int):
            return node_to_vertexpair[t]
        return tuple([decomp(a) for a in t])

    bd = decomp(cd)
    return bd

```

5.2 Computing a medial graph

Solving problem 4.2.

I assume that the input graph G is a rotation system of a planar graph; an adjacency list such that the neighborhoods are given in clockwise ordering according to some plane embedding of G.

Given this format, any two consecutive edges e_1 and e_2 in some face of G are therefore consecutive vertices in the neighborhood of the vertex that e_1 and e_2 share.

The implementation adds all medial links around some vertex for each vertex in G.

The clockwise ordering of neighborhoods of G becomes counterclockwise ordering of neighborhoods of the medial M. The medial graph of a plane graph is 4-regular **??**. 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.

medial_graph.py

```
4  # assume planar graph
5  # assume G_adj is a rotation system
6  def medial_graph(G_adj: dict[int, list[int]]) -> Graph:
7          vertexpairs = set([tuple(sorted((i, j))) for i in G_adj for j in G_adj[i]])
8
9          vertexpair_to_node = dict([(e, i+1) for i,e in enumerate(vertexpairs)])
10         node_to_vertexpair = dict([(i+1, e) for i,e in enumerate(vertexpairs)])
11
12         medial = dict([(i+1, []) for i in range(len(vertexpairs))])
13
14         for u,vs in G_adj.items():
15                 nodes = [vertexpair_to_node[tuple(sorted((u, v)))] for v in vs]
16                 for i in range(len(nodes)):
17                         medial[nodes[i]].append(nodes[(i-1)%len(nodes)])
18                         medial[nodes[i]].append(nodes[(i+1)%len(nodes)])
19
20         M = Graph()
21         M.from_adj(medial)
22         return M, node_to_vertexpair, vertexpair_to_node
```

5.3 Computing a minimal carving decomposition

Solving problem 4.3.

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 sequence of contracted edges is found and reassembled into a minimal carving decomposition.

The "contraction" function returns a new unique vertex ID, therefore by saving which vertex is a contraction of which vertex pair in the "edges" dictionary, constructing the decomposition is then a matter of recursively expanding any vertices that were a result of a contraction into a tuple of the vertex pair that is was composed of. Repeating this until all only vertices of M remain gives a carving decomposition in Newick-like nested tuple format.

carving_decomposition.py

```
        for es in G.E():
                u, v = G.edge_to_vertexpair[es]
                G2, w = contraction(G, u, v)
                cw2 = carving_width(G2)
                if cw2 <= cw1:
                        return G2, (u, v), cw2, w
        return None, None, None, 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) -> Graph:
        G2 = G.copy()
        cw1 = carving_width(G)
        edges = dict()
        while True:
                G3, uv, cw2, w = nonincreasing_cw_contraction(G2, cw1)
```

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

# Contruct a carving decomposition of a graph
def carving_decomposition(G: Graph) -> tuple:
        G2, edges = gradient_descent_contractions(G)

        # Construct the decomposition from the edges that were contracted

        def decomp(x):
                if x not in edges:
                        return x
                a,b = edges[x]
                return (decomp(a), decomp(b))

        a,b,c = G2.V()
        cd = (decomp(a), decomp(b), decomp(c))
        return cd

if __name__ == "__main__":
```

5.4 Contraction

Solving problem 4.4.

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.

As this contraction is a contraction of 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 luckily 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 reusing a or b as this later makes assembling the carving decomposition easier.

First, update any edges incident to either a or b. Then creating the neighborhood of the new vertex c from the contraction of vertices a and b, is done by firstly finding any shared edge e. In this implementation the first shared edge e in the neighborhood of a. This edge has some ID e and the other half-edge with ID $-e$ will therefore be in the neighborhood of b. Now "rotating" the neighborhoods of a and b such edge e and $-e$ is at index 0 in both lists means that a concatenation of the lists will preserve the ordering around the new vertex c. And finally, remove any edges connecting a and b.

This is where telling apart parallel edges, which the Graph class allows, becomes very useful. Inferring where to stitch together the neighborhoods to preserve the ordering, just from a normal adjacency list, becomes a way harder problem.

contraction.py

```
# assume G might have parallel edges
# assume G do not have self-loops
# assume adjacency list of G has clockwise ordering of neighbors
def contraction(G: Graph, a: int, b: int) -> Graph:
        # copy G
        G1 = G.copy()

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

```

```python
    # 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)

    # create neighborhood of c
    def index_of_first(lst, pred):
        for i, v in enumerate(lst):
            if pred(v):
                return i
        return None

    index_of_first_shared_edge = index_of_first(G1.adj_edges[a], lambda e: G1.edge_to_vertexpair[e][0] == c and G1.edge_to_vertexpair[e][1] == c)
    first_shared_edge = G1.adj_edges[a][index_of_first_shared_edge]

    idx1 = G1.adj_edges[a].index(first_shared_edge)
    rotated_Ga = G1.adj_edges[a][idx1:] + G1.adj_edges[a][:idx1]

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

    G1.adj_edges[c] = rotated_Ga + rotated_Gb

    # remove self-loops on c
    G1.adj_edges[c] = [e for e in G1.adj_edges[c] if not (G1.edge_to_vertexpair[e][0] == G1.edge_to_vertexpair[e][1] == c)]
    G1.edge_to_vertexpair = dict([(k,v) for k,v in G1.edge_to_vertexpair.items() if not (v[0] == v[1] == c)])

    # remove a and b
    del G1.adj_edges[a]
    del G1.adj_edges[b]

    return G1, c
```

5.5 Carving width and the rat cathing algorithm

Solving problem 4.5

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

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

```

carving_width.py

```
    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 states
        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 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
```

5.6 Quiet subgraph

Solving problem 4.8.

Using definition 4.6: 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^*.

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 5.1. The shortest closed walk that includes both e_1^* and e_2^* has the same length as 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

```
        # 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 = []
        unseen = set(quiet_subgraph.keys())

        while len(unseen) > 0:
                v = unseen.pop()
                component = [v]
                stack = [v]
                while len(stack) > 0:
                        v = stack.pop()
                        for e1 in quiet_subgraph[v]:
                                u,v = G.edge_to_vertexpair[e1]
                                if v in unseen:
                                        unseen.remove(v)
                                        stack.append(v)
                                        component.append(v)
                components.append(component)

        return components
```

5.7 Dual graph

Solving problem 4.9

No other path of the implementation needs the assumption that the dual is planar, therefore no clockwise or counterclockwise ordering of the neighborhoods of the adjacency list is needed.

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, by following the edges that are just next to each other in the ordered neighborhoods.

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

dual_graph.py

```
        while True:
                u,v = G.edge_to_vertexpair[next_e]
```

dual_graph.py

```
# Assume G is a rotation system
def dual_graph(G: Graph) -> Graph:
        edges = [e for e in G.E()]

        D = Graph()
        edge_to_link = dict()
        link_to_edge = dict()
        node_to_face = dict() # nodeid to edgeid list
        edge_to_node = dict() # half-edge to the faceid/node to its either left/right

        # Find faces
        next_nodeid = -1
        while edges:
                e = edges.pop()
                next_e = e
                edge_to_node[e] = next_nodeid
                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)
                edge_to_node[next_e] = next_nodeid
            node_to_face[next_nodeid] = face
            next_nodeid -= 1

    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(list(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)
```

6 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: ().

7 Appendix

branch_decomposition.py

```python
from parse_graph import parse_text_to_adj
from medial_graph import medial_graph
from carving_decomposition import carving_decomposition

# Construct a branch decomposition of a graph
def branch_decomposition(G_adj: dict[int, list[int]]):
	# Contruct the carving decomposition of the medial graph
	M, node_to_vertexpair, vertexpair_to_node = medial_graph(G_adj)
	cd = carving_decomposition(M)

	# Convert the carving decomposition of M to a branch decomposition of G
	def decomp(t):
		if isinstance(t, int):
			return node_to_vertexpair[t]
		return tuple([decomp(a) for a in t])

	bd = decomp(cd)
	return bd

if __name__ == "__main__":
	adj = parse_text_to_adj()
	bd = branch_decomposition(adj)
	print(bd)
```

branch_width_brute_force.py

```python
from parse_graph import parse_graph_to_adj

# The branchwidth of G is the minimum width of any of its branch-decompositions.
def branch_width(G):
	Ts = branch_decompositions(G)
	min_T = min(Ts, key=width_of_branch_decomposition)
	return width_of_branch_decomposition(min_T)

# A branch-decomposition of a graph G is a tree T such that:
# - The leafs of T are the edges of G.
# - The internal nodes of T have 3 neighbors.
def branch_decompositions(G):
	leaves = [f"{chr(64 + i)}{chr(64 + j)}" for (i, j) in edges(G)]
	trees = enumerate_trees(leaves)
	return [tree.to_adj() for tree in trees]

# The width of a branch-decomposition T is the maximum width of any of its e-separations.
def width_of_branch_decomposition(T):
	return max(width_of_e_seperation(T, e) for e in edges(T))

def edges(T):
	edge_set = set()
	for v in T:
		for w in T[v]:
			edge_set.add(tuple(sorted((v, w))))
	return edge_set

# The width of an e-separation is the number of vertices of G that appear in both T1 and T2.
def width_of_e_seperation(T, e):
	S1 = leafs_of(T, e[0], e[1])
	S2 = leafs_of(T, e[1], e[0])
	return len(set(S1).intersection(S2))

def leafs_of(T, s, x):
	seen = set([x])
	leafs = []
	stack = [s]
	while stack:
```

```python
        v = stack.pop()
        if v in seen:
            continue
        seen.add(v)
        if "internal" not in v:
            leafs.extend(list(v))
        stack.extend(T[v])
    return leafs

# Enumerate trees, https://github.com/fedeoliv/Rosalind-Problems/blob/master/eubt.py
# solving https://rosalind.info/problems/eubt/
class Node():
    def __init__(self, name):
        self.name = name

    def __str__(self):
        if self.name is not None:
            return self.name
        else:
            return "internal_{}".format(id(self))

class Edge():
    def __init__(self, node1, node2):
        self.nodes = [node1, node2]

    def __str__(self):
        return "{}--{}".format(*self.nodes)

class Tree():
    def __init__(self, nodes=[], edges=[]):
        self.nodes = nodes
        self.edges = edges

    def __str__(self):
        return "tree_{} edges: {}".format(id(self), [str(x) for x in self.edges])

    def copy(self):
        node_conversion = {node: Node(node.name) for node in self.nodes}
        new_nodes = list(node_conversion.values())
        new_edges = [Edge(node_conversion[edge.nodes[0]], node_conversion[edge.nodes[1]]) for
        ↪  edge in self.edges]

        new_tree = Tree(new_nodes, new_edges)
        return new_tree

    def to_adj(self):
        adj = {}
        for node in self.nodes:
            adj[str(node)] = []
        for edge in self.edges:
            node1, node2 = edge.nodes
            adj[str(node1)].append(str(node2))
            adj[str(node2)].append(str(node1))
        return adj

def enumerate_trees(leaves):
    assert(len(leaves) > 1)

    if len(leaves) == 2:
        n1, n2 = leaves
        t = Tree()
        t.nodes = [Node(n1), Node(n2)]
        t.edges = [Edge(t.nodes[0], t.nodes[1])]
        return [t]
    elif len(leaves) > 2:
        # get the smaller tree first
        old_trees = enumerate_trees(leaves[:-1])
        new_leaf_name = leaves[-1]
        new_trees = []

```

```python
        # find the ways to add the new leaf
        for old_tree in old_trees:
            for i in range(len(old_tree.edges)):
                new_tree = old_tree.copy()
                edge_to_split = new_tree.edges[i]
                old_node1, old_node2 = edge_to_split.nodes

                # get rid of the old edge
                new_tree.edges.remove(edge_to_split)

                # add a new internal node
                internal = Node(None)
                new_tree.nodes.append(internal)

                # add the new leaf
                new_leaf = Node(new_leaf_name)
                new_tree.nodes.append(new_leaf)

                # make the three new edges
                new_tree.edges.append(Edge(old_node1, internal))
                new_tree.edges.append(Edge(old_node2, internal))
                new_tree.edges.append(Edge(new_leaf, internal))

                # put this new tree in the list
                new_trees.append(new_tree)

        return new_trees

adj = parse_graph_to_adj()

print(branch_width(adj))
```

branch_width.py

```python
from branch_decomposition import branch_decomposition
from parse_graph import parse_text_to_adj, adj_to_text

def branch_width_of_branch_decomposition(bd):
    # Create an adjacency list from the branch decomposition
    T_adj = dict()
    def aux(subtree, depth, name):
        if len(subtree) == 2 and isinstance(subtree[0], int) and isinstance(subtree[1], int):
            T_adj[subtree] = []
            return subtree
        else:
            T_adj[name] = []
            for i,a in enumerate(subtree):
                child_name = aux(a, depth+1, name+str(i))
                T_adj[name].append(child_name)
                T_adj[child_name].append(name)
            return name
    aux(bd, 0, "i0")

    # 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 T_adj[v]:
                    stack.append(w)
        return leafs

```

```python
        # Find the maximal width of any middle set
        width = 0
        for x,ys in T_adj.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(adj: dict[int, list[int]]):
        bd = branch_decomposition(adj)
        return branch_width_of_branch_decomposition(bd)

if __name__ == "__main__":
        adj = parse_text_to_adj()
        bd = branch_decomposition(adj)
        bw = branch_width_of_branch_decomposition(bd)
        print("bw", bw)
```

carving_decomposition.py

```python
from carving_width import carving_width
from contraction import contraction
from parse_graph import parse_text_to_adj, adj_to_text
from Graph import Graph

# Find a contraction that does not increase the carving width
def nonincreasing_cw_contraction(G: Graph, cw1: int) -> tuple:
        for es in G.E():
                u, v = G.edge_to_vertexpair[es]
                G2, w = contraction(G, u, v)
                cw2 = carving_width(G2)
                if cw2 <= cw1:
                        return G2, (u, v), cw2, w
        return None, None, None, 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) -> Graph:
        G2 = G.copy()
        cw1 = carving_width(G)
        edges = dict()
        while True:
                G3, uv, cw2, w = nonincreasing_cw_contraction(G2, cw1)
                if G3 is not None and len(G3.V()) >= 3:
                        G2 = G3
                        cw1 = cw2
                        edges[w] = uv
                if len(G2.V()) == 3:
                        return G2, edges

# Contruct a carving decomposition of a graph
def carving_decomposition(G: Graph) -> tuple:
        G2, edges = gradient_descent_contractions(G)

        # Construct the decomposition from the edges that were contracted

        def decomp(x):
                if x not in edges:
                        return x
                a,b = edges[x]
                return (decomp(a), decomp(b))

        a,b,c = G2.V()
        cd = (decomp(a), decomp(b), decomp(c))
        return cd
```

```python

if __name__ == "__main__":
        adj = parse_text_to_adj()
        cd = carving_decomposition(adj)
        print(cd)
```

carving_width_brute_force.py

```python
from parse_graph import parse_graph_to_adj

G = parse_graph_to_adj()
vertex_set = set(G.keys())

# carving width = minimum carving decomposition width
def carving_width(G: dict[int, list[int]]):
        return min([decomposition_width(d) for d in decompositions_partitions(vertex_set)])

# decomposition width = maximum partition width
def decomposition_width(d):
        return max([partition_width(G, part) for part in d])

def decompositions_partitions(xs: set[int]) -> list[list[tuple[set[int], set[int]]]]:
        if len(xs) == 1:
                return [[(set(xs), vertex_set-set(xs))]]
        parts = []
        for (A, B) in partitions(xs):
                for dA in decompositions_partitions(A):
                        for dB in decompositions_partitions(B):
                                parts.append([(A, vertex_set-A), (B, vertex_set-B), *dA, *dB])
        return parts

# def decompositions(xs: set[int]):
#         if len(xs) == 1:
#                 return list(xs)
#         decomps = []
#         for (A, B) in partitions(xs):
#                 for dA in decompositions(A):
#                         for dB in decompositions(B):
#                                 decomps.append([dA, dB])
#         return decomps

# partition width = number of edges in G crossing the partition
partition_width_cache = dict()
def partition_width(G, partition: tuple[set[int], set[int]]):
        (A, B) = partition
        t_AB = (tuple(A), tuple(B))
        t_BA = (tuple(B), tuple(A))

        if (t_AB) in partition_width_cache: return partition_width_cache[t_AB]
        if (t_BA) in partition_width_cache: return partition_width_cache[t_BA]

        w = len([(u, v) for u in A for v in B if v in G[u]])

        partition_width_cache[t_AB] = w
        partition_width_cache[t_BA] = w

        return w

def partitions(s):
        s = list(s)
        x = len(s)
        for i in range(1, (1<<x)//2):
                A = set([s[j] for j in range(x) if (i & (1 << j))])
                B = set(s) - A
                yield (A, B)

print(carving_width(G))
```

carving_width.py

```python
import math

from Graph import Graph
from parse_graph import adj_to_text, adj_to_text_2, parse_text_to_adj
from dual_graph import dual_graph

def carving_width(G: Graph) -> int:
        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 = []
                unseen = set(quiet_subgraph.keys())

                while len(unseen) > 0:
                        v = unseen.pop()
                        component = [v]
                        stack = [v]
                        while len(stack) > 0:
                                v = stack.pop()
                                for e1 in quiet_subgraph[v]:
                                        u,v = G.edge_to_vertexpair[e1]
```

```
                    if v in unseen:
                        unseen.remove(v)
                        stack.append(v)
                        component.append(v)
            components.append(component)

        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 states
        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 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__":
    adj = parse_text_to_adj()

    G = Graph()
    G.from_adj(adj)
    cw = carving_width(G)
    print("cw", cw)
```

contraction.py

```
from Graph import Graph
from parse_graph import adj_to_text, parse_text_to_adj

# assume G might have parallel edges
# assume G do not have self-loops
# assume adjacency list of G has clockwise ordering of neighbors
def contraction(G: Graph, a: int, b: int) -> Graph:
    # copy G
    G1 = G.copy()

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

    # 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)

    # create neighborhood of c
    def index_of_first(lst, pred):
        for i, v in enumerate(lst):
            if pred(v):
                return i
        return None

    index_of_first_shared_edge = index_of_first(G1.adj_edges[a], lambda e:
    ↪  G1.edge_to_vertexpair[e][0] == c and G1.edge_to_vertexpair[e][1] == c)
    first_shared_edge = G1.adj_edges[a][index_of_first_shared_edge]

    idx1 = G1.adj_edges[a].index(first_shared_edge)
    rotated_Ga = G1.adj_edges[a][idx1:] + G1.adj_edges[a][:idx1]

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

    G1.adj_edges[c] = rotated_Ga + rotated_Gb
```

```python

        # remove self-loops on c
        G1.adj_edges[c] = [e for e in G1.adj_edges[c] if not (G1.edge_to_vertexpair[e][0] == G1.edge_to_vertexpair[e][1] == c)]
        G1.edge_to_vertexpair = dict([(k,v) for k,v in G1.edge_to_vertexpair.items() if not (v[0] == v[1] == c)])

        # remove a and b
        del G1.adj_edges[a]
        del G1.adj_edges[b]

        return G1, c

if __name__ == "__main__":
        a,b = map(int, input().split())
        adj = parse_text_to_adj()

        G = Graph()
        G.from_adj(adj)
        G1, c = contraction(G, a, b)
        adj_to_text(G1.adj())
        print("c", c)
```

dual_graph.py

```python
from Graph import Graph
from parse_graph import adj_to_text, parse_text_to_adj

# Assume G is a rotation system
def dual_graph(G: Graph) -> Graph:
        edges = [e for e in G.E()]

        D = Graph()
        edge_to_link = dict()
        link_to_edge = dict()
        node_to_face = dict() # nodeid to edgeid list
        edge_to_node = dict() # half-edge to the faceid/node to its either left/right

        # Find faces
        next_nodeid = -1
        while edges:
                e = edges.pop()
                next_e = e
                edge_to_node[e] = next_nodeid
                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)
                        edge_to_node[next_e] = next_nodeid
                node_to_face[next_nodeid] = face
                next_nodeid -= 1

        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(list(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)
```

```python
                    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__":
    adj = parse_text_to_adj()
    G = Graph()
    G.from_adj(adj)
    D, edge_to_link, link_to_edge, node_to_face, edge_to_node = dual_graph(G)
    adj_to_text(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)
```

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
        self.adj_edges = adj.copy()
        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] for e in self.adj_edges[v]]

    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()
        H.adj_edges = self.adj_edges.copy()
        H.edge_to_vertexpair = self.edge_to_vertexpair.copy()
        return H
```

medial_graph.py

```python
from Graph import Graph
from parse_graph import adj_to_text, parse_text_to_adj

# assume planar graph
# assume G_adj is a rotation system
def medial_graph(G_adj: dict[int, list[int]]) -> Graph:
        vertexpairs = set([tuple(sorted((i, j))) for i in G_adj for j in G_adj[i]])

        vertexpair_to_node = dict([(e, i+1) for i,e in enumerate(vertexpairs)])
        node_to_vertexpair = dict([(i+1, e) for i,e in enumerate(vertexpairs)])

        medial = dict([(i+1, []) for i in range(len(vertexpairs))])

        for u,vs in G_adj.items():
                nodes = [vertexpair_to_node[tuple(sorted((u, v)))] for v in vs]
                for i in range(len(nodes)):
                        medial[nodes[i]].append(nodes[(i-1)%len(nodes)])
                        medial[nodes[i]].append(nodes[(i+1)%len(nodes)])

        M = Graph()
        M.from_adj(medial)
        return M, node_to_vertexpair, vertexpair_to_node

if __name__ == "__main__":
        adj = parse_text_to_adj()
        M, node_to_vertexpair, vertexpair_to_node = medial_graph(adj)
        adj_to_text(M.adj())
        print("node_to_vertexpair", node_to_vertexpair)
        print("vertexpair_to_node", vertexpair_to_node)
```

parse_graph.py

```python
import sys

def parse_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 parse_text_to_adj() -> 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_text_2(adj):
        s = str(len(adj)) + "\n"
        for v,xs in adj.items():
                s += str(v) + " " + " ".join(map(str, xs)) + "\n"
        return s
```

```python

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")
```

parse_newick.py

```python
def tokenize_newick(s: str) -> list:
	tokens = []
	token = ''
	for c in s:
		if c.isnumeric():
			token += c
		else:
			if len(token) > 0:
				tokens.append(token)
			token = ''
		if c != ' ':
			pass
		if c == '(':
			tokens.append('(')
		if c == ')':
			tokens.append(')')
		if c == ',':
			tokens.append(',')
	return tokens

def rec(tokens: list[str], i: int) -> tuple:
	if tokens[0].isnumeric():
		return tokens[1:], int(tokens[0])

	tail0, t0 = rec(tokens[1:], i+1)

	tail1, t1 = rec(tail0[1:], i+1)

	if i == 0:
		tail2, t2 = rec(tail1[1:], i+1)
		return (tail2[1:], (t0, t1, t2))

	return (tail1[1:], (t0, t1))

def parse_newick(s: str) -> tuple:
	tokens = tokenize_newick(s)
	return rec(tokens, 0)[1]

nw = parse_newick(input())
print(nw)
```