

Implementation of a minimal branch-decomposition algorithm for simple 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 Motivation

Some graph optimization problems can be solved efficiently for graphs of small branchwidth.??

which? counting hamiltonian cycles of planar cubic graphs

2 Description of the problem

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

Definition 2.1. THE PLANAR MINIMAL BRANCH DECOMPOSITION PROBLEM

Input: Given a simple undirected connected planar graph G .

Output: A minimal branch decomposition of G .

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

A graph is called **simple** if and only if it has no parallel edges and no self-loops.

A graph is called **undirected** if and only if all its edges can be traversed in both directions.

A graph is called **connected** (or 1-vertex-connected) if and only if there exists a path between any two vertices.

A graph is called **planar** if and only if there is a way to draw the graph in 2 dimensions such that no pair of edges crosses.

A **branch decomposition** B of a graph G is a tree where every edge of G is a leaf in B and every internal vertex of B has exactly 3 neighbors. B is an unrooted binary tree.

Removing any edge $e = \{u, v\}$ of B partitions B into 2 trees B_u and B_v and the intersection of the sets of vertices in the leaves of B_u and B_v is called a middle set. Every edge of B has an associated middle set. The maximal cardinality of any middle set among all middle sets of B is the width of B .

There can be 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 .

Without the constraint of planarity on the input graph, then it is NP-complete to determine whether a general graph G has a branch decomposition of width at most k , when G and k are both considered as inputs to the problem.

Definition 2.2. THE MINIMAL BRANCH DECOMPOSITION PROBLEM

Input: Given a simple undirected 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 called the branch width of G .

3 The algorithm

This section describes the algorithm given by Seymour and Thomas?? by indentifying a set of practical problems and subproblems and how they relate.

Problem 2.2 is the overarching problem, that the algorithm solves, and can be broken down into many smaller subproblems.

Considering a graph G , you can compute a minimal branch decomposition of G from a minimal carving decomposition of the medial of G , by replacing the vertices in leaves of the decomposition with edges, using the mapping between edges and vertices from computing the medial graph. ?? Therefore problem 2.2 break down into problems 3.1, 3.2 and 3.3.

Problem 3.1. Given a minimal carving decomposition of a medial graph of G , output a minimal branch decomposition of G .

Problem 3.2. Given a graph G , output a medial graph and a bijectional mapping between medial nodes and vertex pairs.

Problem 3.3. 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 3.1 is decribed in 4.1.

To solve 3.2 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 of disambiguation.

Informally; The medial graph is a graph where there is a node in the medial for each edge, and an edge between two nodes if their corresponding edges are consecutive in a face of the graph.

Definition 3.4. The medial graph $M(G)$ of a connected plane graph G is a graph with a vertex e^* for each edge e of G and for each face f of G , there's an edge d^* between a pair of vertices e_1^*, e_2^* of $M(G)$ if e_1 and e_2 are consecutive in f .

Computing a medial graph is described in 4.2.

To solve 3.3 ?? gives an contraction algorithm.

Doing a series of edge contractions (contraction of all edges between 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 4.3.

The contraction algorithm depends on a function to compute a contraction and a function to compute the carving width. This is problems 3.5 and 3.6.

Problem 3.5. Given a graph M that might have parallel edges, output a graph resulting from a contraction of all edges between a pair of vertices.

Problem 3.6. Given a plane graph M that might have parallel edges, output the carving width of M .

First consider problem 3.5.

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 an edge- and vertex-contractions, as other definitions might result in self-loops when contracting one of multiple parallel edges.

Definition 3.7. Contraction.

Given an undirected Graph $G = \{V, E\}$ with no self-loops and pair of vertices $u, v \in V$ such that $\{u, v\} \in E$, remove all edges between u and v and update every edge $\{v, w\} \in E$ to be $\{u, w\}$.

Computing a medial graph is described in 4.2.

Now consider problem 3.6.

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 corner 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, is definitions 3.8 and 3.9.

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

Definition 3.8. 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^* .

Definition 3.9. When the rat-catcher is on some room 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 $e \in E(M)$, is a subgraph of M with $V(Q(M, k, e)) = V(M)$ and

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

Problem 3.10. Given a plane graph M that might have parallel edges, an edge $e \in E(M)$, and noise-level $k \in \mathbb{N}$, output the quiet subgraph $Q(M, k, e)$.

Problem 3.10 depends on a function for computing the dual of a graph. Computing a dual graph is problem 3.11.

Problem 3.11. Given a plane graph $M = \{V, E\}$ that might have parallel edges, output the dual of M .

Definition 3.12. The dual graph M^* of a plane graph M is a graph where a face f of M is a vertex f^* of M^* and an edge e that separates two faces f_1, f_2 of M is an edge e^* of M^* between f_1^* and f_2^* .

Corollary 3.13. If multiple edges separate f_1 and f_2 there will be parallel edges between f_1^* and f_2^* .

Corollary 3.14. If the same face is on both sides of e , then e^* will be a self-loop.

For the algorithm in this paper, the class of graphs that will be given as input is medial graphs of simple undirected connected planar graphs. Therefore I claim 3.15.

Claim 3.15. Corollary 3.14 will be irrelevant for any implementation of the algorithm.

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. $S = \{(f, v) \mid v \in V(M) \wedge f \text{ is a face of } M\}$.

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

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.

4 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 halfedge has ID i then the other halfedge 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         self.adj_edges = adj.copy()
10        next_edgeid = 1
11        for x, ys in self.adj_edges.items():
12            for i, y in enumerate(ys):
13                if x < y:
14                    self.edge_to_vertexpair[next_edgeid] = (x, y)
15                    self.edge_to_vertexpair[-next_edgeid] = (y, x)
16
17                    self.adj_edges[x][i] = next_edgeid
18                    self.adj_edges[y][adj[y].index(x)] = -next_edgeid
19
20                    next_edgeid += 1
21
22    def V(self) -> list[int]:
23        return list(self.adj_edges.keys())
24
25    def E(self) -> list[int]:
26        return list(self.edge_to_vertexpair.keys())
27
28    def N(self, v: int) -> list[int]:
29        return [self.edge_to_vertexpair[e][1] for e in self.adj_edges[v]]
30
31    def adj(self) -> dict[int, list[int]]:
32        return dict([(x, self.N(x)) for x in self.adj_edges.keys()])
33
34    def copy(self):
35        H = Graph()
36        H.adj_edges = self.adj_edges.copy()
37        H.edge_to_vertexpair = self.edge_to_vertexpair.copy()
38        return H
```

4.1 Computing a minimal branch decomposition

Solving problem 3.1.

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

```

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

```

4.2 Computing a medial graph

Solving problem 3.2.

I assume that the input graph G is a planar graph given as 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 orderings 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 links added to the neighborhood of v also in counterclockwise ordering.

medial_graph.py

```

4  # assume planar graph
5  # assume clockwise ordering of neighbors
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

```

4.3 Computing a minimal carving decomposition

Solving problem 3.3.

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 was a result of a contraction into a tuple of the vertex pair that it 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

```

8  # Find a contraction that does not increase the carving width
9  def nonincreasing_cw_contraction(G: Graph, cw1: int) -> tuple:
10     for es in G.E():
11         u, v = G.edge_to_vertexpair[es]
12         G2, w = contraction(G, u, v)
13         cw2 = carving_width(G2)
14         if cw2 <= cw1:
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     return cd

```

4.4 Contraction

Solving problem 3.5.

As the resulting graph is later given as an argument to functions assuming a clockwise or counter-clockwise ordering of vertices, the implementation needs to preserve this invariant when contracting.

As this contraction is a contraction of ALL edges between a pair of vertices, the resulting graph will

not exhibit any self-loops. I suspect reconciling this and the ordering invariant 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 halfedge 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 between 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

```

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

```


4.5 Carving width and the rat cathing algorithm

Solving problem 3.6

The vertices of game state graph H is initialized by computing the elements of T and S , while edges of H are not explicitly kept in any datastructure, 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) which is losing where $e \in f$ and $v \in V(C)$ then (f, v) is losing.

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)
9
```

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

```

4.6 Quiet subgraph

Solving problem 3.10.

Using definition 3.8: 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^* .

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

Claim 4.1. 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 path from u to v .

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

Using the mapping from links of the dual 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 connected components thereof is done with a depth-first search approach.

The edges of the connected components are irrelevant for the rest of the algorithm, so only a list of vertices are returned for each connected 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             return dist
28
29         dist_s = dists(s)

```

```

30         dist_t = dists(t)
31
32     noisy = []
33     for l1 in links:
34         u,v = D.edge_to_vertexpair[l1]
35         if min(
36             dist_s[u] + dist_t[v] + 2,
37             dist_s[v] + dist_t[u] + 2
38         ) < k:
39             noisy.append(l1)
40
41     return set([abs(e) for e in noisy])
42
43 def quiet_links(l: int, k: int) -> set[int]:
44     links = set([abs(e) for e in D.E()])
45     return links - noisy_links(l, k)
46
47 def quiet_edges(e: int, k: int) -> set[int]:
48     return set([abs(link_to_edge[l]) for l in quiet_links(edge_to_link[e], k)])
49
50 def quiet_components(e: int, k: int) -> list[list[int]]:
51     edges = quiet_edges(e, k)
52
53     quiet_subgraph = {v: [] for v in G.V()}
54     for e1 in edges:
55         u,v = G.edge_to_vertexpair[e1]
56         quiet_subgraph[u].append(e1)
57         quiet_subgraph[v].append(-e1)
58
59     components = []
60     unseen = set(quiet_subgraph.keys())
61
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

```

4.7 Dual graph

Solving problem 3.11

No other path of the implementation need 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 halfedge, and then marking all the edges of the face it belongs to, by following the edges which are just next to eachother 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

```

21         idx = G.adj_edges[v].index(-next_e)
22         next_e = G.adj_edges[v][(idx-1)%len(G.adj_edges[v])]

```

dual_graph.py

```

4  def dual_graph(G: Graph) -> Graph:
5      edges = [e for e in G.E()]
6
7      D = Graph()
8      edge_to_link = dict()
9      link_to_edge = dict()
10     node_to_face = dict()
11     edge_to_node = dict()
12
13     next_nodeid = -1
14     while edges:
15         e = edges.pop()
16         next_e = e
17         edge_to_node[e] = next_nodeid
18         face = [e]
19         while True:
20             u,v = G.edge_to_vertexpair[next_e]
21             idx = G.adj_edges[v].index(-next_e)
22             next_e = G.adj_edges[v][(idx-1)%len(G.adj_edges[v])]
23             if (next_e == e):
24                 break
25             edges.remove(next_e)
26             face.append(next_e)
27             edge_to_node[next_e] = next_nodeid
28             node_to_face[next_nodeid] = face
29             next_nodeid -= 1
30
31     for i in node_to_face.keys():
32         D.adj_edges[i] = []
33
34     next_linkid = 1
35     for i,f1 in node_to_face.items():
36         for j,f2 in node_to_face.items():
37             if i < j:
38                 common_edges = set(list(map(abs, f1))).intersection(set(map(abs, f2)))
39                 for e in common_edges:
40                     D.edge_to_vertexpair[next_linkid] = (i, j)
41                     D.edge_to_vertexpair[-next_linkid] = (j, i)
42                     edge_to_link[e] = next_linkid
43                     link_to_edge[next_linkid] = e
44                     edge_to_link[-e] = -next_linkid
45                     link_to_edge[-next_linkid] = -e
46                     D.adj_edges[i].append(next_linkid)
47                     D.adj_edges[j].append(-next_linkid)
48                     next_linkid += 1
49
50     return D, edge_to_link, link_to_edge, node_to_face, edge_to_node

```

5 Appendix.

branch_decomposition.py

```

1  from parse_graph import parse_text_to_adj
2  from medial_graph import medial_graph
3  from carving_decomposition import carving_decomposition
4
5  # Construct a branch decomposition of a graph

```

```

6 def branch_decomposition(G_adj: dict[int, list[int]]):
7     # Construct the carving decomposition of the medial graph
8     M, node_to_vertpair, vertpair_to_node = medial_graph(G_adj)
9     cd = carving_decomposition(M)
10
11     # Convert the carving decomposition of M to a branch decomposition of G
12
13     def decomp(t):
14         if isinstance(t, int):
15             return node_to_vertpair[t]
16         return tuple([decomp(a) for a in t])
17
18     bd = decomp(cd)
19     return bd
20
21 if __name__ == "__main__":
22     adj = parse_text_to_adj()
23     bd = branch_decomposition(adj)
24     print(bd)

```

branch_width_brute_force.py

```

1 from parse_graph import parse_graph_to_adj
2
3 # The branchwidth of G is the minimum width of any of its branch-decompositions.
4 def branch_width(G):
5     Ts = branch_decompositions(G)
6     min_T = min(Ts, key=width_of_branch_decomposition)
7     return width_of_branch_decomposition(min_T)
8
9 # A branch-decomposition of a graph G is a tree T such that:
10 # - The leafs of T are the edges of G.
11 # - The internal nodes of T have 3 neighbors.
12 def branch_decompositions(G):
13     leaves = [f"{chr(64 + i)}{chr(64 + j)}" for (i, j) in edges(G)]
14     trees = enumerate_trees(leaves)
15     return [tree.to_adj() for tree in trees]
16
17 # The width of a branch-decomposition T is the maximum width of any of its e-separations.
18 def width_of_branch_decomposition(T):
19     return max(width_of_e_seperation(T, e) for e in edges(T))
20
21 def edges(T):
22     edge_set = set()
23     for v in T:
24         for w in T[v]:
25             edge_set.add(tuple(sorted((v, w))))
26     return edge_set
27
28 # The width of an e-separation is the number of vertices of G that appear in both T1 and T2.
29 def width_of_e_seperation(T, e):
30     S1 = leafs_of(T, e[0], e[1])
31     S2 = leafs_of(T, e[1], e[0])
32     return len(set(S1).intersection(S2))
33
34 def leafs_of(T, s, x):
35     seen = set([x])
36     leafs = []
37     stack = [s]
38     while stack:
39         v = stack.pop()
40         if v in seen:
41             continue
42         seen.add(v)
43         if "internal" not in v:
44             leafs.extend(list(v))
45         stack.extend(T[v])
46     return leafs
47

```

```

48 # Enumerate trees, https://github.com/fedeoliv/Rosalind-Problems/blob/master/eubt.py
49 # solving https://rosalind.info/problems/eubt/
50 class Node():
51     def __init__(self, name):
52         self.name = name
53
54     def __str__(self):
55         if self.name is not None:
56             return self.name
57         else:
58             return "internal_{}".format(id(self))
59
60 class Edge():
61     def __init__(self, node1, node2):
62         self.nodes = [node1, node2]
63
64     def __str__(self):
65         return "{}--{}".format(*self.nodes)
66
67 class Tree():
68     def __init__(self, nodes=[], edges=[]):
69         self.nodes = nodes
70         self.edges = edges
71
72     def __str__(self):
73         return "tree_{} edges: {}".format(id(self), [str(x) for x in self.edges])
74
75     def copy(self):
76         node_conversion = {node: Node(node.name) for node in self.nodes}
77         new_nodes = list(node_conversion.values())
78         new_edges = [Edge(node_conversion[edge.nodes[0]], node_conversion[edge.nodes[1]]) for
79             ↪ edge in self.edges]
80
81         new_tree = Tree(new_nodes, new_edges)
82         return new_tree
83
84     def to_adj(self):
85         adj = {}
86         for node in self.nodes:
87             adj[str(node)] = []
88         for edge in self.edges:
89             node1, node2 = edge.nodes
90             adj[str(node1)].append(str(node2))
91             adj[str(node2)].append(str(node1))
92         return adj
93
94 def enumerate_trees(leaves):
95     assert(len(leaves) > 1)
96
97     if len(leaves) == 2:
98         n1, n2 = leaves
99         t = Tree()
100         t.nodes = [Node(n1), Node(n2)]
101         t.edges = [Edge(t.nodes[0], t.nodes[1])]
102         return [t]
103     elif len(leaves) > 2:
104         # get the smaller tree first
105         old_trees = enumerate_trees(leaves[:-1])
106         new_leaf_name = leaves[-1]
107         new_trees = []
108
109         # find the ways to add the new leaf
110         for old_tree in old_trees:
111             for i in range(len(old_tree.edges)):
112                 new_tree = old_tree.copy()
113                 edge_to_split = new_tree.edges[i]
114                 old_node1, old_node2 = edge_to_split.nodes
115
116                 # get rid of the old edge
117                 new_tree.edges.remove(edge_to_split)

```

```

117
118         # add a new internal node
119         internal = Node(None)
120         new_tree.nodes.append(internal)
121
122         # add the new leaf
123         new_leaf = Node(new_leaf_name)
124         new_tree.nodes.append(new_leaf)
125
126         # make the three new edges
127         new_tree.edges.append(Edge(old_node1, internal))
128         new_tree.edges.append(Edge(old_node2, internal))
129         new_tree.edges.append(Edge(new_leaf, internal))
130
131         # put this new tree in the list
132         new_trees.append(new_tree)
133
134     return new_trees
135
136 adj = parse_graph_to_adj()
137
138 print(branch_width(adj))

```

branch_width.py

```

1  from branch_decomposition import branch_decomposition
2  from parse_graph import parse_text_to_adj, adj_to_text
3
4  def branch_width_of_branch_decomposition(bd):
5      # Create an adjacency list from the branch decomposition
6      T_adj = dict()
7      def aux(subtree, depth, name):
8          if len(subtree) == 2 and isinstance(subtree[0], int) and isinstance(subtree[1], int):
9              T_adj[subtree] = []
10             return subtree
11         else:
12             T_adj[name] = []
13             for i,a in enumerate(subtree):
14                 child_name = aux(a, depth+1, name+str(i))
15                 T_adj[name].append(child_name)
16                 T_adj[child_name].append(name)
17             return name
18         aux(bd, 0, "i0")
19
20     # Get the vertex set of the leafs of the subtree of x (not y)
21     def leafs_set(x, y):
22         leafs = set()
23         visited = set([y])
24         stack = [x]
25         while stack:
26             v = stack.pop()
27             if isinstance(v, tuple):
28                 leafs.update(set(v))
29                 continue
30             if v not in visited:
31                 visited.add(v)
32                 for w in T_adj[v]:
33                     stack.append(w)
34         return leafs
35
36     # Find the maximal width of any middle set
37     width = 0
38     for x,ys in T_adj.items():
39         for y in ys:
40             a = leafs_set(x, y)
41             b = leafs_set(y, x)
42             middle_set = len(a.intersection(b))
43             width = max(width, middle_set)
44

```

```

45         return width
46
47 def branch_width(adj: dict[int, list[int]]):
48     bd = branch_decomposition(adj)
49     return branch_width_of_branch_decomposition(bd)
50
51 if __name__ == "__main__":
52     adj = parse_text_to_adj()
53     bd = branch_decomposition(adj)
54     bw = branch_width_of_branch_decomposition(bd)
55     print("bw", bw)

```

carving_decomposition.py

```

1  from carving_width import carving_width
2  from contraction import contraction
3  from parse_graph import parse_text_to_adj, adj_to_text
4  from dual_graph import dual_graph
5  from medial_graph import medial_graph
6  from Graph import Graph
7
8  # Find a contraction that does not increase the carving width
9  def nonincreasing_cw_contraction(G: Graph, cw1: int) -> tuple:
10     for es in G.E():
11         u, v = G.edge_to_vertexpair[es]
12         G2, w = contraction(G, u, v)
13         cw2 = carving_width(G2)
14         if cw2 <= cw1:
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 # Contract 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     return cd
49
50 if __name__ == "__main__":
51     adj = parse_text_to_adj()
52     cd = carving_decomposition(adj)
53     print(cd)

```


carving_width_brute_force.py

```

1  from parse_graph import parse_graph_to_adj
2
3  G = parse_graph_to_adj()
4  vertex_set = set(G.keys())
5
6  # carving width = minimum carving decomposition width
7  def carving_width(G: dict[int, list[int]]):
8      return min([decomposition_width(d) for d in decompositions_partitions(vertex_set)])
9
10 # decomposition width = maximum partition width
11 def decomposition_width(d):
12     return max([partition_width(G, part) for part in d])
13
14 def decompositions_partitions(xs: set[int]) -> list[list[tuple[set[int], set[int]]]]:
15     if len(xs) == 1:
16         return [[(set(xs), vertex_set-set(xs))]]
17     parts = []
18     for (A, B) in partitions(xs):
19         for dA in decompositions_partitions(A):
20             for dB in decompositions_partitions(B):
21                 parts.append([(A, vertex_set-A), (B, vertex_set-B), *dA, *dB])
22     return parts
23
24 # def decompositions(xs: set[int]):
25 #     if len(xs) == 1:
26 #         return list(xs)
27 #     decomp = []
28 #     for (A, B) in partitions(xs):
29 #         for dA in decompositions(A):
30 #             for dB in decompositions(B):
31 #                 decomp.append([dA, dB])
32 #     return decomp
33
34 # partition width = number of edges in G crossing the partition
35 partition_width_cache = dict()
36 def partition_width(G, partition: tuple[set[int], set[int]]):
37     (A, B) = partition
38     t_AB = (tuple(A), tuple(B))
39     t_BA = (tuple(B), tuple(A))
40
41     if (t_AB) in partition_width_cache: return partition_width_cache[t_AB]
42     if (t_BA) in partition_width_cache: return partition_width_cache[t_BA]
43
44     w = len([(u, v) for u in A for v in B if v in G[u]])
45
46     partition_width_cache[t_AB] = w
47     partition_width_cache[t_BA] = w
48
49     return w
50
51 def partitions(s):
52     s = list(s)
53     x = len(s)
54     for i in range(1, (1<<x)//2):
55         A = set([s[j] for j in range(x) if (i & (1 << j))])
56         B = set(s) - A
57         yield (A, B)
58
59 print(carving_width(G))

```

carving_width.py

```

1  import math
2
3  from Graph import Graph
4  from parse_graph import adj_to_text, adj_to_text_2, parse_text_to_adj
5  from dual_graph import dual_graph

```

```

6
7 def carving_width(G: Graph) -> int:
8     D, edge_to_link, link_to_edge, node_to_face, edge_to_node = dual_graph(G)
9
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
63        while len(unseen) > 0:
64            v = unseen.pop()
65            component = [v]
66            stack = [v]
67            while len(stack) > 0:
68                v = stack.pop()
69                for e1 in quiet_subgraph[v]:
70                    u,v = G.edge_to_vertexpair[e1]
71                    if v in unseen:
72                        unseen.remove(v)
73                        stack.append(v)
74                        component.append(v)
75
76            components.append(component)

```

```

76         return components
77
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
131
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
142        m = 1
143        while l < r:
144            m = int(math.ceil((l + r) / 2))
145            if rat_wins(m):
146                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
159 if __name__ == "__main__":
160     adj = parse_text_to_adj()
161
162     G = Graph()
163     G.from_adj(adj)
164     cw = carving_width(G)
165     print("cw", cw)

```

contraction.py

```

1  from Graph import Graph
2  from parse_graph import adj_to_text, parse_text_to_adj
3
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)])

```

```

44
45     # remove a and b
46     del G1.adj_edges[a]
47     del G1.adj_edges[b]
48
49     return G1, c
50
51 if __name__ == "__main__":
52     a,b = map(int, input().split())
53     adj = parse_text_to_adj()
54
55     G = Graph()
56     G.from_adj(adj)
57     G1, c = contraction(G, a, b)
58     adj_to_text(G1.adj())
59     print("c", c)

```

dual_graph.py

```

1  from Graph import Graph
2  from parse_graph import adj_to_text, parse_text_to_adj
3
4  def dual_graph(G: Graph) -> Graph:
5      edges = [e for e in G.E()]
6
7      D = Graph()
8      edge_to_link = dict()
9      link_to_edge = dict()
10     node_to_face = dict()
11     edge_to_node = dict()
12
13     next_nodeid = -1
14     while edges:
15         e = edges.pop()
16         next_e = e
17         edge_to_node[e] = next_nodeid
18         face = [e]
19         while True:
20             u,v = G.edge_to_vertexpair[next_e]
21             idx = G.adj_edges[v].index(-next_e)
22             next_e = G.adj_edges[v][(idx-1)%len(G.adj_edges[v])]
23             if (next_e == e):
24                 break
25             edges.remove(next_e)
26             face.append(next_e)
27             edge_to_node[next_e] = next_nodeid
28         node_to_face[next_nodeid] = face
29         next_nodeid -= 1
30
31     for i in node_to_face.keys():
32         D.adj_edges[i] = []
33
34     next_linkid = 1
35     for i,f1 in node_to_face.items():
36         for j,f2 in node_to_face.items():
37             if i < j:
38                 common_edges = set(list(map(abs, f1))).intersection(set(map(abs, f2)))
39                 for e in common_edges:
40                     D.edge_to_vertexpair[next_linkid] = (i, j)
41                     D.edge_to_vertexpair[-next_linkid] = (j, i)
42                     edge_to_link[e] = next_linkid
43                     link_to_edge[next_linkid] = e
44                     edge_to_link[-e] = -next_linkid
45                     link_to_edge[-next_linkid] = -e
46                     D.adj_edges[i].append(next_linkid)
47                     D.adj_edges[j].append(-next_linkid)
48                     next_linkid += 1
49
50     return D, edge_to_link, link_to_edge, node_to_face, edge_to_node

```

```

51
52 if __name__ == "__main__":
53     adj = parse_text_to_adj()
54     G = Graph()
55     G.from_adj(adj)
56     D, edge_to_link, link_to_edge, node_to_face, edge_to_node = dual_graph(G)
57     adj_to_text(D.adj())
58     print("edge_to_link", edge_to_link)
59     print("link_to_edge", link_to_edge)
60     print("node_to_face", node_to_face)
61     print("edge_to_node", edge_to_node)

```

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         self.adj_edges = adj.copy()
10        next_edgeid = 1
11        for x, ys in self.adj_edges.items():
12            for i, y in enumerate(ys):
13                if x < y:
14                    self.edge_to_vertexpair[next_edgeid] = (x, y)
15                    self.edge_to_vertexpair[-next_edgeid] = (y, x)
16
17                    self.adj_edges[x][i] = next_edgeid
18                    self.adj_edges[y][adj[y].index(x)] = -next_edgeid
19
20                next_edgeid += 1
21
22    def V(self) -> list[int]:
23        return list(self.adj_edges.keys())
24
25    def E(self) -> list[int]:
26        return list(self.edge_to_vertexpair.keys())
27
28    def N(self, v: int) -> list[int]:
29        return [self.edge_to_vertexpair[e][1] for e in self.adj_edges[v]]
30
31    def adj(self) -> dict[int, list[int]]:
32        return dict([(x, self.N(x)) for x in self.adj_edges.keys()])
33
34    def copy(self):
35        H = Graph()
36        H.adj_edges = self.adj_edges.copy()
37        H.edge_to_vertexpair = self.edge_to_vertexpair.copy()
38        return H

```

medial_graph.py

```

1 from Graph import Graph
2 from parse_graph import adj_to_text, parse_text_to_adj
3
4 # assume planar graph
5 # assume clockwise ordering of neighbors
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
23
24     if __name__ == "__main__":
25         adj = parse_text_to_adj()
26         M, node_to_vertexpair, vertexpair_to_node = medial_graph(adj)
27         adj_to_text(M.adj())
28         print("node_to_vertexpair", node_to_vertexpair)
29         print("vertexpair_to_node", vertexpair_to_node)

```

parse_graph.py

```

1  import sys
2
3  def parse_bin_to_adj() -> dict[int, list[int]]:
4      adj = dict()
5      n = ord(sys.stdin.buffer.read(1))
6      for i in range(1, n+1):
7          adj[i] = []
8
9      i = 1
10     while i <= n:
11         x = ord(sys.stdin.buffer.read(1))
12         if x == 0:
13             i += 1
14             continue
15         adj[i].append(x)
16     return adj
17
18 def parse_text_to_adj() -> dict[int, list[int]]:
19     adj = dict()
20     n = int(input())
21     for _ in range(n):
22         ys = list(map(int, input().split()))
23         x = ys[0]
24         adj[x] = []
25         for y in ys[1:]:
26             adj[x].append(y)
27     return adj
28
29 def adj_to_text(adj):
30     print(len(adj))
31     for v,xs in adj.items():
32         print(v, *xs)
33
34 def adj_to_text_2(adj):
35     s = str(len(adj)) + "\n"
36     for v,xs in adj.items():
37         s += str(v) + " " + " ".join(map(str, xs)) + "\n"
38     return s
39
40 def adj_to_nx(adj):
41     G = nx.MultiDiGraph()
42     for v,xs in adj.items():
43         G.add_node(v)
44         for x in xs:
45             G.add_edge(v, x)
46     return G
47
48 def adj_to_bytes(adj):
49     print(chr(len(adj)), end="")
50     for v,xs in adj.items():
51         print("".join(map(chr, [*xs])), end="\x00")

```

parse_newick.py

```
1 def tokenize_newick(s: str) -> list:
2     tokens = []
3     token = ''
4     for c in s:
5         if c.isnumeric():
6             token += c
7         else:
8             if len(token) > 0:
9                 tokens.append(token)
10                token = ''
11            if c != ' ':
12                pass
13            if c == '(':
14                tokens.append('(')
15            if c == ')':
16                tokens.append(')')
17            if c == ',':
18                tokens.append(',')
19    return tokens
20
21 def rec(tokens: list[str], i: int) -> tuple:
22     if tokens[0].isnumeric():
23         return tokens[1:], int(tokens[0])
24
25     tail0, t0 = rec(tokens[1:], i+1)
26
27     tail1, t1 = rec(tail0[1:], i+1)
28
29     if i == 0:
30         tail2, t2 = rec(tail1[1:], i+1)
31         return (tail2[1:], (t0, t1, t2))
32
33     return (tail1[1:], (t0, t1))
34
35 def parse_newick(s: str) -> tuple:
36     tokens = tokenize_newick(s)
37     return rec(tokens, 0)[1]
38
39 nw = parse_newick(input())
40 print(nw)
```