Implementation of an optimal branch-decomposition algorithm for planar graphs.

Kristoffer Højelse

February 2024

Abstract

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

1 Motivation

Graph optimization problems may be solved efficiently for graphs of small branchwidth.

2 Description of the problem

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

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 and is associated with the edge e. Every edge of B therefore, has an associated middle set. The maximal cardinality of any middle set among all middle sets of B is the width of the branch decomposition 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.

Definition 2.1. The Minimal Branch Decomposition Problem

Input: Given a simple undirected connected graph G.

Output: A minimal branch decomposition of G.

3 The domain

The algorithm described in this paper solves The Minimal Branch Decomposition Problem with an additional constraint of the input graph being planar.

Some informal definitions of the aforementioned properties of graphs:

1. A graph is called **simple** if and only if it has no parallel edges and no self-loops.
2. A graph is called **undirected** if and only if all its edges can be traversed in both directions.
3. A graph is called **connected** (or 1-vertex-connected) if and only if there exists a path between any two vertices.
4. 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.

Different subroutines of the algorithm that take a graph as argument, assume different properties of the graph.

3.1 Medial graph

A subroutine of the algorithm computes the medial graph of a graph from the class; simple connected planar cubic graphs.

Definition 3.1. The medial graph $M(G)$ of a connected plane graph G with a vertex e^* for each edge e of G and for each face f of G, there's an edge $c*$ between a pair of vertices e_1*, e_2* of $M(G)$ if e_1 and e_2 are consecutive in f.

todo: argue that simple connected planar cubic graphs are a valid substitution for a connected plane.

todo: identify a more succinct description of the class; medial of simple connected planar cubic graphs

An implementation of this subroutine can be found in the appendix 5.2.

3.2 Dual graph

A subroutine of the algorithm computes the dual graph of a graph.

Definition 3.2. The dual graph $G*$ of a planar graph G is a graph with a vertex $f*$ for each face f of G and an edge e^* for each edge e that separates a face f_1 of G and a face f_2 of G.

Corollary 3.3. If multiple edges separate f_1 and f_2 there will be multiple edges between f_1* and f_2*.

Corollary 3.4. If e separates f_1 and f_2 and are the same face, 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 cubic graphs.

I therefore claim, for now without any proof or argument, that;

Claim 3.5. Corollary 3.4 will be irrelevant for any implementation of the algorithm.

An implementation of this subroutine can be found in the appendix 5.3.

3.3 Edge contraction

A subroutine of the algorithm computes the resulting graph from an edge contraction.

This operation is a bit different from the conventional understanding of an edge contraction.

Definition 3.6. Edge 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 $\{u, w\}$.

The resulting graph might have parallel edges but will not have self-loops.

4 Description of the algorithm

The algorithm computes a minimal branch decomposition of a simple connected planar graph G, the width of this branch decomposition is the branch width of G. This section describes a high-level set of problems for the algorithm to tackle.

Problem 4.1. Given a simple connected planar graph G, output a minimal branch decomposition of G.

Problem 4.1 is the overarching problem, that the algorithm solves, and can be broken down into many smaller subproblems.

For the first large step in the algorithm, we use the fact that a minimal carving decomposition of the medial of a graph G can be translated into a minimal branch decomposition of G, by replacing the vertices in leaves of the decompositions with edges, using the mapping between edges and vertices from computing the medial graph.

The argument for why this is true can be found in **??**.

Problem 4.2. Given a graph G and a minimal carving decomposition of a medial graph of G, output a minimal branch decomposition of G.

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

```
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
```

```python

    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)
```

Problem 4.3. Given a graph G, output a medial graph and a mapping from medial nodes to vertex pairs.

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

Therefore if you assume that the adjacency list of G is given such that the neighborhoods are given in clockwise ordering according to some plane embedding of G then a pair of consecutive nodes in a neighborhood are also consecutive edges in some face of G.

medial_graph.py

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

# assume planar graph
# assume clockwise ordering of neighbors
def medial_graph(G_adj: dict[int, list[int]]) -> Graph:
    half_edges = 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(half_edges)])
    node_to_vertexpair = dict([(i+1, e) for i,e in enumerate(half_edges)])

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

    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)
```

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

```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
```

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 reassembled into a minimal carving decomposition of M.

The argument for why this is true can be found in **??**.

> **Problem 4.4.** Given a graph G that might have parallel edges, output a graph resulting from a contraction of all edges between a pair of vertices, preserving clockwise ordering.

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

As the 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.

Creating the neighborhood of c from the neighborhoods of a and b is done by finding the first edge that the adjacency lists of a and b have in common, and then "rotating" the adjacency lists such that a concatenation of the lists preserve the ordering. This is where telling apart parallel edges is very useful.

contraction.py

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

def index_of_first(lst, pred):
        for i, v in enumerate(lst):
                if pred(v):
                        return i
        return None

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

        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

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)
```

I will defer describing how to compute the carving width and focus on the following problem for now.

Problem 4.5. Given a graph G and function to compute the carving width of a graph, output a minimal carving decomposition of G.

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.

carving_decomposition.py

```
from carving_width import carving_width
from contraction import contraction
from parse_graph import parse_text_to_adj, adj_to_text
from dual_graph import dual_graph
from medial_graph import medial_graph
from Graph import Graph

# Find a contraction that does not increase the carving width
def nonincreasing_cw_contraction(G: Graph, cw1):
        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, e, cw2, w = nonincreasing_cw_contraction(G2, cw1)
                if G3 is not None and len(G3.V()) >= 3:
                        G2 = G3
                        cw1 = cw2
                        edges[w] = e
                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__":
        adj = parse_text_to_adj()
        cd = carving_decomposition(adj)
        print(cd)
```

Problem 4.6. Given a graph M that might have parallel edges, output the carving width of M.

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 proposition $cw(M) \geq k$ is true if and only if $\Delta(M) \geq k$ or the rat can evade the rat-catcher indefinitely, with noise-level k, in a particular game based on M.

carving_width.py

```
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)

        # When the rat-catcher is on edge e, edge f is noisy iff there is
        # a closed walk of length scrictly less than k containing e* and f* in G* .
        # Return the un-noisy subgraph.
        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)

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

                for x,ys in D.adj().items():
```

```
                        blah[x] = ys

                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
                edge_set = edge_to_link.keys()

                Te = set([(e, tuple(C)) for e in edge_set for C in quiet_components(e, k)])
                Sr = set([(r, v) for r in node_to_face.keys() for v in G.V()])

                # Set up the losing states
                losing_eC = set()
                losing_rv = set()

                for (r, v) in Sr:
                        if v in flatten([G.edge_to_vertexpair[e] for e in node_to_face[r]]):
                                losing_rv.add((r,v))

                if len(Te) == len(losing_eC) or len(Sr) == len(losing_rv):
                        return False

                # Play the game
                while True:
                        new_deletion = False

                        for (e, C) in Te:
                                if all([(edge_to_node[e], v) in losing_rv for v in C]):
                                        if (e, C) not in losing_eC:
                                                new_deletion = True
                                                losing_eC.add((e, C))

                        for (e, C) in losing_eC:
                                r1 = edge_to_node[e]
                                r2 = edge_to_node[-e]
                                for (r, v) in [(r1, v) for v in C] + [(r2, v) for v in C]:
                                        if (r, v) not in losing_rv:
                                                new_deletion = True
                                                losing_rv.add((r, v))

```

```
            if len(Te) == len(losing_eC) or len(Sr) == len(losing_rv):
                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)
```

Problem 4.7. Given a graph G that might have parallel edges, output the dual of G.

5 Appendix.

5.1 Count Hamiltonian Cycles with Brute Force

count_hamcyc_brute_force.py

```
import itertools
from parse_graph import parse_text_to_adj

G: dict[int, list[int]] = parse_text_to_adj()
vertex_set = G.keys()
N = len(vertex_set)

def valid(cycle: list[int]) -> bool:
    for i in range(0, N-1):
        if not cycle[i+1] in G[cycle[i]]:
            return False
    if not cycle[0] in G[cycle[-1]]:
        return False
    return True

```

```
def count_ham_cyc() -> int:
        cycles = itertools.permutations(vertex_set)
        count = 0
        for cycle in cycles:
                if valid(cycle):
                        count += 1
        return count//(2*N)

print(count_ham_cyc())
```

5.2 Medial graph

medial_graph.py

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

# assume planar graph
# assume clockwise ordering of neighbors
def medial_graph(G_adj: dict[int, list[int]]) -> Graph:
        half_edges = 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(half_edges)])
        node_to_vertexpair = dict([(i+1, e) for i,e in enumerate(half_edges)])

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

        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)
```

5.3 Dual graph

dual_graph.py

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

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()
        edge_to_node = dict()

        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] = []

    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)
                    next_linkid += 1

    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)
```