Maximum Independent Set and Maximum Induced Matching Problems for Competitive Programming

by

Janet Dean Brock

Honors Thesis

Appalachian State University

Submitted to the Department of Computer Science

and the Honors College

in partial fulfillment of the requirements for the degree of

Bachelor of Science

May 2021

APPROVED BY:

Raghuveer Mohan, Ph.D., Thesis Director

Stephen Hedetniemi, Ph.D., Second Reader

Alice McRae, Ph.D., Departmental Honors Director

Jefford Vahlbusch, Ph.D., Dean, The Honors College

ABSTRACT

Maximum Independent Set and Maximum Induced Matching Problems for Competitive Programming.

(May 2021)

Janet Dean Brock, Appalachian State University

Appalachian State University

Thesis Chairperson: Raghuveer Mohan, Ph.D.

Competitive programming is a growing interest among students, with some students training for years to be competitive in national and international competitions. Competitive programming problems continue to become more complex; yet they are always solvable with skills learned in an undergraduate algorithms class. This makes competitive programming a great way for undergraduates to develop their coding skills and learn complex algorithms. However, there are very few competitive programming problems on particular graph classes, despite the fact that the field of graph theory is rich with complexity and algorithms results for over eighty years. This may be because of the overwhelming amount of graph classes and terminology that students need to be familiar with to understand even the simplest results in graph theory, sometimes overlooking the connection between graph theory and the study of algorithms. Some of these algorithms, like that of computing a maximum independent set (MIS) or a maximum induced matching (MIM) on special graph classes, only require techniques learned in an undergraduate algorithms course. However, in the literature, they are hidden behind results for generalized classes, often using terminology and notation far beyond what undergraduate students are exposed to. Some of these graph theoretic results are either so old that the original papers are hard to find or they are held behind payment gateways from publishers. Therefore, there needs to be substantive work done to improve the expositions of old (and some new) graph theory algorithms that can be solved using topics learned in an undergraduate course. This will allow students in algorithm classes to be exposed to topics in graph theory, while fundamental problems on graphs can be used as excellent motivating examples for topics in algorithms.

In this thesis, to bridge the gap between graph theory and competitive programming, we provide new expositions on computing a MIS and a MIM on five different graph classes – trees, interval graphs, circular-arc graphs, permutation graphs and trapezoid graphs. Using fancy data structures, we provide a novel, and currently the fastest, algorithm for computing a MIM on permutation and trapezoid graphs that runs in only $O((m + n)\log n)$ time, where n is the number of elements in the input, and m is the number of intersections between elements.

As a final contribution, we describe six competitive programming problems, three of which are original, based on intersection graphs, and provide their solutions. We hope that this work inspires future contest writers to write more problems on special graph classes, thereby exposing more of the field of graph theory to undergraduate students, and graph theory researchers to provide newer expositions on published results.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Competitive programming is becoming a growing interest among students of computer science and mathematics. Programming contests offer challenging and exciting problems that often involve extraordinary levels of creative problem solving, and offer excellent motivational examples for topics in algorithms and data structures. These problems are getting more and more challenging every year and students go through a rigorous training program to compete. Some students even start as early as ninth grade as they train and participate in the United States Computing Olympiad (USACO), or other national computing olympiads, and the International Olympiad for Informatics (IOI) [16].

The problems in these contests are solvable with topics learned in an undergraduate algorithms course, even though they require tremendous levels of insight, algorithmic problem solving, abstracting unnecessary details, reducing problems into other known problems, programming proficiency, and software design and testing that includes generating large test cases. Therefore, competitive programming helps develop all the skills in a computer programmer's toolkit.

Some common topics found in programming contest problems include dynamic programming, greedy algorithms, complete and exhaustive search techniques, sorting and searching algorithms, string matching, and common problems in graphs like graph traversals, computing shortest paths, minimum spanning trees and maximum flows. However, there seems to be very few problems on particular graph classes, even though the field of graph theory has been rich with complexity and algorithmic results over the last eighty years. One reason for this is perhaps the overwhelming amount of graph classes and terminology that students need to be familiar with to even understand simple results in graph theory, sometimes overlooking the study of algorithms on graph classes. Some of these algorithms, like that of computing a maximum independent set (MIS) or a maximum inudced matching (MIM) on special graph classes only require techniques learned in an undergraduate algorithms course. However, in the literature, they are hidden behind results for generalized classes often using terminology and notation far beyond what undergraduate students are exposed to. For example, the results for interval graphs are often expressed in terms of chordal or co-comparibility graphs [1], and results for trapezoid graphs are expressed in terms of their higher dimensional generalization, which are the k-trapezoid graphs. Some of these graph theoretic results are so old that the original papers are hard to find or they are held behind payment gateways from publishers.

Therefore, there needs to be substantive work done in improving expositions of old (and some new) graph theory algorithms that can be solved using topics learned in an undergraduate algorithms course. This allows students of algorithms to be exposed to topics in graph theory, while fundamental problems on graphs can be used as excellent motivating examples for topics in algorithms. This connection may inspire future contest writers to write more problems on special graph classes, thereby exposing more of the field of graph theory to undergraduate students.

To our knowledge, the recent work of Do, Pham and Than from Vietnam [5] is the only work that has attempted to bridge this gap. The authors being non-English speakers, we found their paper difficult to understand because of ambiguous statements, grammatical errors, and typographical errors in their pseudocode. This thesis corrects their errors, and offers new expositions of the MIS and MIM problems on five different graph classes – trees, interval graphs, cicrular-arc graphs, permutation graphs and trapezoidal graphs. These results along with their algorithmic techniques are shown in the table below. Wherever applicable, we have provided citations to the original papers in the table.

We also make the following novel contribution. We have developed the fastest algorithm for computing a MIM on permutation and trapezoid graphs. Our algorithm runs in only $\mathcal{O}((m+n)\log n)$ time where n is the number of elements in the input, and m is the number of intersections among elements in the input. Although not part of this thesis, we have been

	Tree Graphs	Circular-Arc Graphs	Interval Graphs	Permutation Graphs	Trapezoid Graphs
MIS	Greedy $O(n)$ [4]	Greedy $O(n \log n)$ [15]	Greedy (Activity Selection) $O(n \log n)$	DP (LIS), BSTs $O(n \log n)$	Sweep Lines, DP (LIS), BSTs $O(n \log n)$ [6]
MIM	DP $O(n)$ [13]	Alpha-redundant, Greedy $O(n \log n)$ [5]	Apha-redundant, Greedy (Activity Selection) $O(n \log n)$ [5]	Sweep Lines, Priority Queues, BSTs, DP (LIS) $O((m+n) \log n)$ (this thesis)	Sweep Lines, Priority Queues, BSTs, DP (LIS) $O((m+n) \log n)$ (this thesis)

Table 1.1: Table of Results

able to extend this algorithm to compute k-induced matchings, which we intend to publish in a forthcoming paper.

As a final contribution, we offer competitive programming problems on intersection graphs. Some of the problems are original, while we also improve the presentation of problems in [5]. We hope that this thesis inspires more contest writers to write more problems on graph theory, and graph theory researchers to write more accessible exposition on important topics in graph theory to undergraduate students.

The rest of the thesis is organized as follows. In the next chapter, we provide the necessary background and terminology used in the rest of the thesis. We also provide a brief literature review of previous results. In Chapter 3 we describe algorithms for MIM and MIS problems on trees, interval graphs, circular-arc graphs, permutation graphs, and trapezoid graphs. In Chapter 4, we discuss six competitive programming problems related to graph theory and describe their solutions. We conclude our thesis in Chapter 5 summarizing the algorithms in this thesis and offering comments on future work.

Chapter 2

Preliminaries

Let $G = (V, E)$ be a graph with vertex set V and edge set E. We let $n = |V|$ be the number of vertices in G, and $m = |E|$ be the number of edges in G. We use small letters u and v to denote vertices and tuples (u, v) to denote edges that connect vertices u and v.

A set $S \subseteq V$ is said to be independent if no two vertices in S are adjacent to each other. The number of vertices in a largest independent set possible in a graph is the independence number, denoted $\alpha(G)$. The set itself is called a maximum independent set (MIS). Somewhat similar to an independent set, a matching is a set of edges $S \subseteq E$ such that no two edges in S share a common vertex. An induced matching is a set of edges $S \subseteq E$ such that no two edges in S are connected by a path of length at most 2. A maximum induced matching (MIM) is the maximum cardinality matching you can make of a graph. Though both of these problems are considered NP-hard for general graphs, they can be solved efficiently on certain graph classes.

There is also a neat trick that reduces the problem of computing a MIM to the problem of computing a MIS. Let $L(G)$ denote the line graph of G; this means the edges of G are vertices in $L(G)$, and two vertices in $L(G)$ are connected if their corresponding edges in G have a vertex in common. Given a graph G, the graph G^k has the same vertex set as G, but two vertices are connected if and only if there is a path of length at most k between them. So the graph $L(G)^2$ connects all vertices in $L(G)$ that are connected by paths of lengths 1 or 2. Additionally, an MIS in $L(G)^2$ corresponds to an MIM on G. This is demonstrated in Figure 2.1 below. This means, if $L(G)^2$ is the same class of graphs as G, then we can compute a MIM on G by using the algorithm to compute a MIS on $L(G)^2$. In this thesis, we will study the MIS and MIM

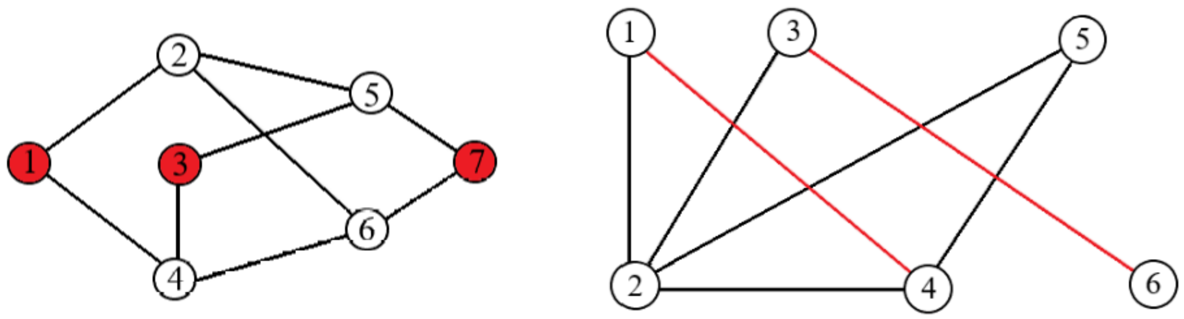

Figure 2.1: Example of a MIS and MIM on a Graph

problems on five different graph classes. We define them in the next section.

2.1 Graph Classes

One of the more well known graph classes we will discuss is trees. A tree is an undirected graph in which for every pair of vertices u and v there is a unique path between u and v. Equivalenetly, a tree is an undirected, acyclic connected graph. Since the graph contains no cycles, and it is connected, we have $m = n - 1$. In some trees, the root vertex r need not be specified. In this case, we can always root our graph at an arbitrary vertex r, and determine all parents by traversing the graph from r. Rooting at an arbitrary vertex does not change the optimal solutions of both the MIS and MIM algorithms. So, without loss of generality, we will assume that the input tree is always rooted.

Linear algorithms for computing a MIS on trees were developed by Daykin and Ng [4] and Mitchell [17]. Daykin and Ng also designed an algorithm for finding a MIS on a vertex-weighted tree; as did Cockayne and Hedetniemi [2]. Algorithms for weighted independent sets use a dynamic programming approach. If all vertices have unit weight, then these algorithms find a MIS. However, Daykin and Ng showed that a MIS in an unweighted tree can be computed using a very simple greedy approach. We describe this elegant algorithm in the next chapter. As for the MIM, Fricke and Laskar [9] developed a linear time algorithm that was later simplified by Golumbic and Lewenstein [13], which uses a dynamic programming technique. We also describe this algorithm in the next chapter.

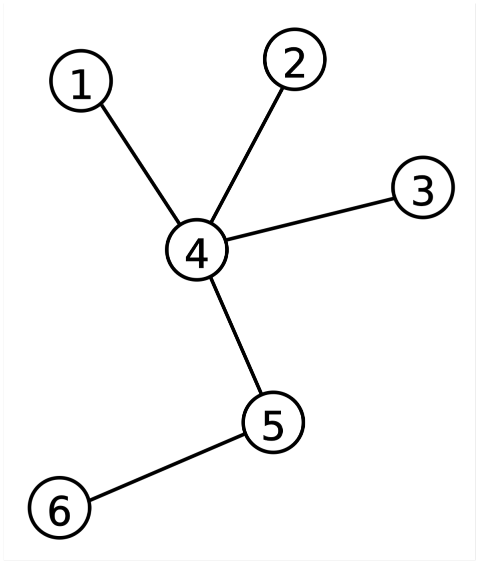

Figure 2.2: A Tree

The other graph classes we will explore in this thesis are all graphs made to represent geometric objects. Sometimes they are also called geometric graphs, where you are given a set of similar geometric objects (like circles, rectangles, intervals, etc.) and a graph is formed with vertices representing each object, and edges corresponding to intersection between pairs of obejcts within the geometric space. Since edges correspond to intersections, these graphs also belong to the class of intersection graphs. If m and n denote the number of edges and vertices in the geometric graph, then n is the number of geometric objects in the input, and m is the number of intersections between paris of objects. For all the algorithms discussed in this thesis, we will assume that we are given as input the geometric representation of the objects, not their graph representation.

An interval graph represents intervals on a line. Each vertex represents a different interval, and if two intervals overlap they are connected by an edge in the graph. For unweighted interval graphs we can use a greedy algorithm to find a MIS; however, we have to sort the intervals by their right endpoints first, so the runtime is $\mathcal{O}(n \log n)$. For weighted interval graphs, Hsiaso and Tang [14] developed an algorithm based on dynamic programming that also runs in $\mathcal{O}(n \log n)$.

The first algorithm to find a MIM on interval graphs was developed by Cameron [1], but it was expressed in terms of chordal graphs. Golumbic and Lewenstein [13] developed an algorithm to find a MIM in $\mathcal{O}(m + n)$ time after the input set of intervals have been sorted by their left endpoints. They found and removed α-redundant vertices, which are vertices in the graph whose removal does not affect the maximum independent set of the graph. Finding these intervals took $\mathcal{O}(m)$ time, so their algorithm can be said to take a total of $\mathcal{O}(m + n \log n)$ time. Do et al. [5] gave a slight improvement that runs in only $O(n)$ time once the intervals are sorted. We will describe their algorithm in in Chapter 3.

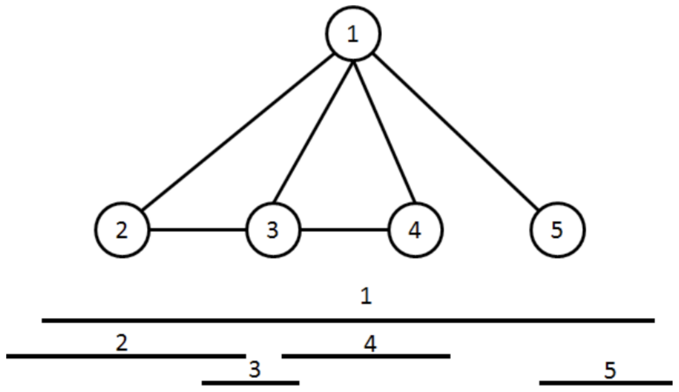

Figure 2.3: An Interval Graph

Similar to interval graphs, circular-arc graphs represent arcs on a circle. Each vertex represents a different arc and if two arcs overlap they are connected by an edge. A MIS on a circular-arc graph can be found with the same algorithm to find a MIS on interval graphs [5]. Gavril [10] also developed a similar polynomial time algorithm, but their paper is hard to access. Hsu and Tai [15] developed a greedy algorithm for finding a MIS on circular-arc graphs that runs in $O(n)$ time once the arcs are sorted by their endpoints in clockwise direction. Their algorithm first finds a circular-arc that must be part of a MIS and determines the rest of the arcs in a greedy fashion. We will expand on this in Chapter 3.

Golumbic and Laskar [12] showed that if G is a circular-arc graph, then $L(G)^2$ is too, so finding a MIS on $L(G)^2$ corresponds to MIM on G. We can then use Golumbic and Hammer's

[11] (and also Do et al. [5]) algorithm to compute a MIS on $L(G)^2$, whose algorithm requires the set of circular-arcs sorted in clockwise direction. Do et al. showed how to use the α-redundant technique to compute a MIM on circular-arc graphs in only $\mathcal{O}(n \log n)$ time. We discuss this in Chapter 3.

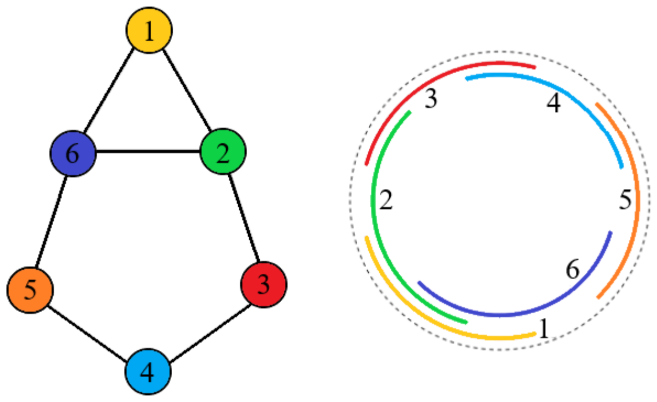

Figure 2.4: A Circular-Arc Graph

Permutation graphs represent a set of numbers before and after a permutation. Visualize the numbers as sitting on two parallel lines, one with numbers before the permutation, and one with the numbers after. Now draw line segments connecting the numbers before the permutation to where they ended up after the permutation; some of these lines will cross each other. A permutation graph represents this set of line segments. Vertices correspond to line segments, and edges correspond to intersections of pairs of these line segments. A MIS on permutation graphs corresponds to computing the longest increasing subsequence of the input permutation, which is a sequence of numbers, not necessarily contiguous that is strictly increasing in the input permutation. This can be computed in only $\mathcal{O}(n \log n)$ time, which we describe in Chapter 3. Permutation graphs are a subset of trapezoid graphs [3], and our novel algorithm for trapezoid graphs computes a MIM on permutation graphs too.

To picture the geographic representation of trapezoid graphs, imagine two parallel lines, where the two parallel sides of the trapezoids intersect these parallel lines. In the figure 2.6 below, we show a set of trapezoids labelled $1, \ldots 8$, and indicate them by labeling their left

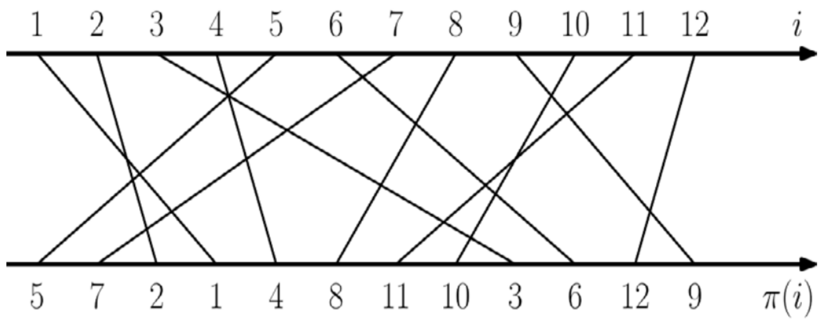
1 2 3 4 5 6 7 8 9 10 11 12 i
5 7 2 1 4 8 11 10 3 6 12 9 $\pi(i)$

Figure 2.5: A Permutation Graph

and right endpoints on the two parallel lines with the same integer. Note that some of these trapezoids can be overlapping. In a trapezoid graph each vertex corresponds to a trapezoid, and edges connect two trapezoids that overlap. Felsner et al. [6] developed an algorithm to find a MIS on a trapezoid graph using a sweep line technique and dynamic programming similar to finding the longest increasing subsequence in only $\mathcal{O}(n \log n)$ time. This algorithm is optimal due to a matching lower bound given by Fredman [8].

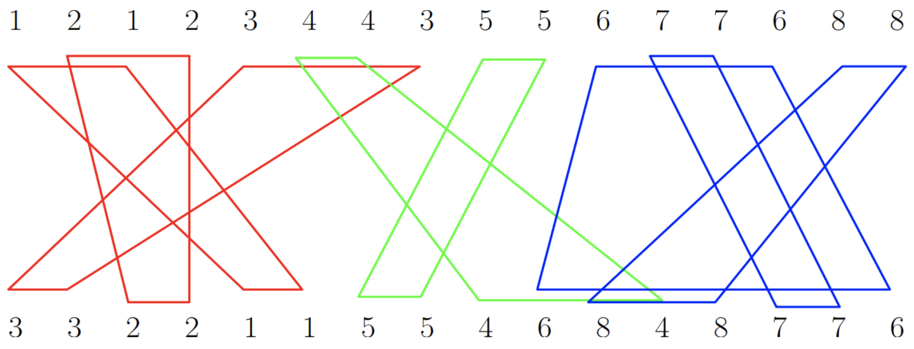

Figure 2.6: A Trapezoid Graph

Golumbic and Lewenstein [13] showed that if G is a trapezoid graph G, then so is $L(G)^2$. So we can transform G to $L(G)^2$ and then use Flesner et al.'s [6] algorithm to compute a MIS. Straightforward implementations could take as much as $\mathcal{O}(m^2 + m \log m)$, which in the worst-case is $\mathcal{O}(n^4)$. Using range query data structures and sweep lines, we can reduce this to only $\mathcal{O}((m+n)\log n)$. We show this in Chapter 3.

2.2 Other Graph Theory Topics in Competitive Programming

We mention that there are few other topics in graph theory that are rather popular in competitive programming. Trees come up quite frequently, with problems revolving around pre and post order traversals, articulation points and bridges, shortest paths on weighted trees (edges or vertices being weighted), finding lowest common ancestors, finding diameters, and other dynamic programming problems on trees. There are also a good number of problems about directed acyclic graphs (DAG) in competitive programming. Some problems include finding a shortest or longest path between two vertices, topological sorting, and counting paths the number of paths between two vertices in a graph.

Though not quite as frequent there are some problems related to bipartite graphs, usually about finding the maximum cardinality bipartite matching which is computed using a maximum flow. There are some, but even fewer problems about Eulerian graphs; usually about finding Eulerian cycles/paths or detecting if a graph is Eulerian. And only recently there have been a very small number of problems related to complete graphs, forests/paths, pseudo-forests/trees, star graphs, and planar graphs. Those related to planar graphs tend to focus on Kuratowski's theorem. Perhaps future work can give a better overview of these other graph theory topics in competitive programming.

Chapter 3

Algorithms

In this chapter, we describe algorithms for computing a maximum independent set (MIS) and a maximum induced matching (MIM) on five different graph classes – trees, interval graphs, circular-arc graphs, permutation graphs and trapezoid graphs. For previous work on these algorithms, please see Chapter 2. The algorithms we describe can be discovered using topics in competitive programming or an undergraduate algorithms course.

3.1 MIS on Trees

On an unweighted and undirected tree $G = (V, E)$, we can use a simple greedy algorithm starting at the leaves of the tree. Include all the leaves of the tree in your set. Then remove all the leaves and their parents from the tree, and repeat the process until no more vertices are left in the tree. Firstly, note that this computes a maximal independent set, since only one t6he set of leaves of the parents of the leaves can be in the MIS.

To prove that this algorithm also finds a maximum independent set, that is a maximal set with largest cardinality, we use a simple greedy exchange argument. Consider an optimal solution OPT that agrees with our greedy solution the most. The vertices in our greedy solution can be thought of being computed in a post-order traversal of a tree. As a base case, a leaf vertex is in the set. As an inductive case, we include vertex u in the set only if all of its children are not in the set, otherwise u is not included in the set. Let u be the first vertex they disagree on; meaning u is in the greedy solution, but not in OPT, and both algorithms agree on u's

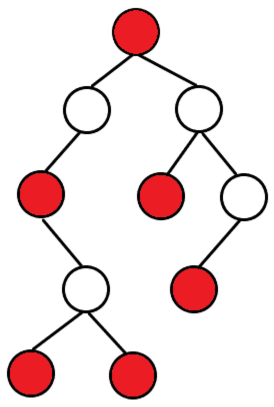
Figure 3.1: MIS on a Tree in Red

subtrees. We know u's children are not in OPT, because it agrees with our greedy solution until u. Note that u's parent p has to be in OPT, otherwise we can improve OPT by adding u to it, and therefore we have a contradiction that OPT is in fact optimal. This means p's children cannot be in OPT, and therefore the size of our greedy solution and OPT in p's subtrees is unaffected – we can simply apply the same exchange argument in p's subtrees and transform vertices in OPT to greedy without affecting its size. This leaves us with p's parent g. OPT cannot choose g while our greedy solution can, which contradicts the fact that OPT was indeed optimal. Therefore our greedy solution is optimal.

Since a MIS can be computed using a simple post-order traversal of the tree, and that computing whether a vertex u is included in the set is only dependent on its degree, the total time is $\sum_{u \in V} degree(u) = \mathcal{O}(n)$.

We now discuss algorithms on computing MIS on intersection graphs. Each of these graphs has a geometric representation as well as a graph representation. We assume that we are given their geometric representations as input, otherwise we may need to transform the graph to its geometric representations, which could be slow. The runtime of the following alorithms are dependent on n and m, where n is the number of objects in the input and m is the number of intersections among these objects.

3.2 MIS on Interval Graphs

The activity selection problem asks us to determine the largest set of disjoint events among a set of events, where each event is modeled as an interval on the number line. It can be easily seen that computing this is exactly a MIS on the corresponding interval graph.

The solution is to use a greedy algorithm. Let S contain all the intervals. We first choose an interval $s \in S$ with the earliest endpoint. Then we remove s from S and add it to our MIS, then remove all other intervals that intersect with s from S. We then repeat the process until no more intervals are left in S. An $\mathcal{O}(n \log n)$ sorting algorithm that sorts the input set of intervals based on the endpoints allows us to pick s easily in every step.

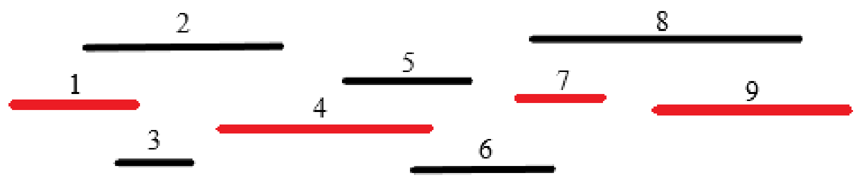

Figure 3.2: MIS on an Interval Graph in Red

To show that this algorithm produces a maximum independent set, first observe that this produces a independent set, since if we choose an interval s, we don't choose any interval that intersects with it. To show that this also produces a MIS, consider an optimal solution OPT that agrees with our greedy solution the most, and let interval u be the first place where they disagree. Let OPT choose v instead of u. If v does not overlap with u, then we can get a better solution by adding u to OPT, thereby contradicting the fact that OPT was optimal. Since u overlaps with v and the endpoint of u finishes earlier than the endpoint of v, we can safely exchange v for u in the optimal solution without affecting the rest of the optimal solution. Therefore our greedy solution is no worse than OPT.

3.3 MIS on Circular-Arc Graphs

Finding a MIS on a circular-arc graph is actually very similar to finding a MIS on an interval graph. Hsu and Tai [15] showed that once we find one arc that can appear in a MIS, we can determine the rest of the arcs in the set by using the same greedy algorithm for interval graphs.

To determine such an arc, they built a graph as follows. For each arc x in the input, we let $l(x)$ and $r(x)$ be its two endpoints, and that the arc goes from $l(x)$ to $r(x)$ in clockwise direction. Let $next(x)$ be the arc y that does not overlap with x, and whose right endpoint $r(y)$ ends earliest in clockwise order after $r(x)$. That is, when we apply the greedy algorithm from interval graphs, y will be the next interval in an independent set that also contains x. Also consider the set of arcs $x = x_1, x_2, \ldots, x_k = z$, where $next(x_i) = x_{i+1}$. Using the same greedy exchange argument for interval graphs, this set of arcs is the largest independent set that includes arc x. Therefore, in order to find a MIS for the entire set of arcs, we only need to determine a "good" arc, an arc that is guaranteed to be in a MIS, and then apply the $next(.)$ function iteratively to determine the rest of the arcs in the set. We construct a graph with vertex set corresponding to arcs, and each arc x has exactly one edge that points to $next(x)$. Therefore, every vertex of this graph has outdegree 1, and must have at least one directed cycle.

Hsu and Tai [15] claim that a good arc is always part of a directed cycle. They prove this by contradiction. We omit details of this proof, and leave it as an exercise for interested readers. To implement the *next* function, we make use of a doubly linked list or a circular array of arcs sorted by their endpoints (say array a), and an array of arcs c where each arc keeps track of its left and right endpoints in a (the left endpoint will point to its successor). Then, we can monotonically scan through a using two pointers i and j starting from an arbitrary arc. Fixing i at some index which corresponds to some arc x, we advance j until we find arc $y = next(x)$. Let the left endpoint of arc y point to index j' in a. Then all arcs between j' and j are overlapping with i, and must also overlap with all arcs between i and j'. Therefore as we advance i, it is safe to continue searching from j. Since at each step we are either advancing i or j, it takes only $\mathcal{O}(n)$ time to determine $next(.)$ for every arc in the input.

Once we find the next arc for every arc, the rest of the algorithm is simple – determine an arc x that forms a directed cycle, then find the rest of the independent set that includes

x. Both of these steps take only $\mathcal{O}(n)$ time. However, we need to sort the arcs by their right endpoints. So the entire algorithm takes $\mathcal{O}(n \log n)$.

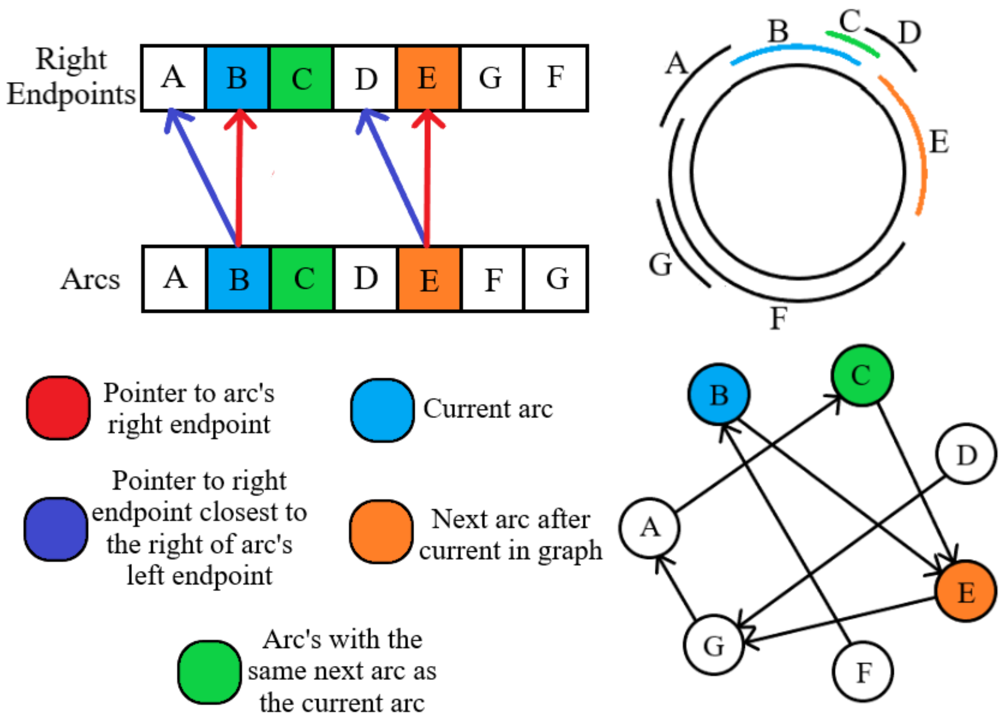
Right Endpoints
A
B
C
D
E
G
F
Arcs
A
B
C
D
E
F
G
Pointer to arc's right endpoint
Pointer to right endpoint closest to the right of arc's left endpoint
Current arc
Next arc after current in graph
Arc's with the same next arc as the current arc
A
B
C
D
E
F
G
A
B
C
D
E
F
G

Figure 3.3: Creating the Graph from the Arcs after Sorting Them

3.4 MIS on Permutation Graphs

We are given as input a permutation π of the numbers $1, \ldots, n$. As a small note, even if we are just given an array of comparable objects as input instead of the numbers $1, \ldots, n$, we can re-label each object with the index it would appear in sorted order. We can imagine two parallel lines A and B with line segments connecting i on the first line to $\pi(i)$ on the second line. It can be seen that a set of disjoint line segments has to form an increasing subsequence in the input unsorted permutation. This is because during the "sorting" process, each object x in the permutation will only cross paths with other objects that are larger than x to the left of x, and smaller than x to the right of x. The longest such increasing subsequence (LIS) therefore gives a MIS on a permutation graph. Note that a subsequence need not contain contiguous elements in the input array.

We can compute a LIS using a dynamic program. Let the array L represent our state space, where $L[i]$ represents the length of the longest increasing subsequence that ends at i. As a base case, we have $L[0] = 0$. Starting from index 1 that represents the first element in the input, we can compute $L[i] = (\max\{L[j]\} + 1) : \forall\ 0 \leq j < i, A[j] < A[i]$, where A is the input array. The optimal solution can end at any index, therefore $\texttt{OPT} = \max_{j=1}^{n} L[j]$. Since there are $\mathcal{O}(n)$ subproblems and each one takes $\mathcal{O}(n)$ time, the total time is $\mathcal{O}(n^2)$.

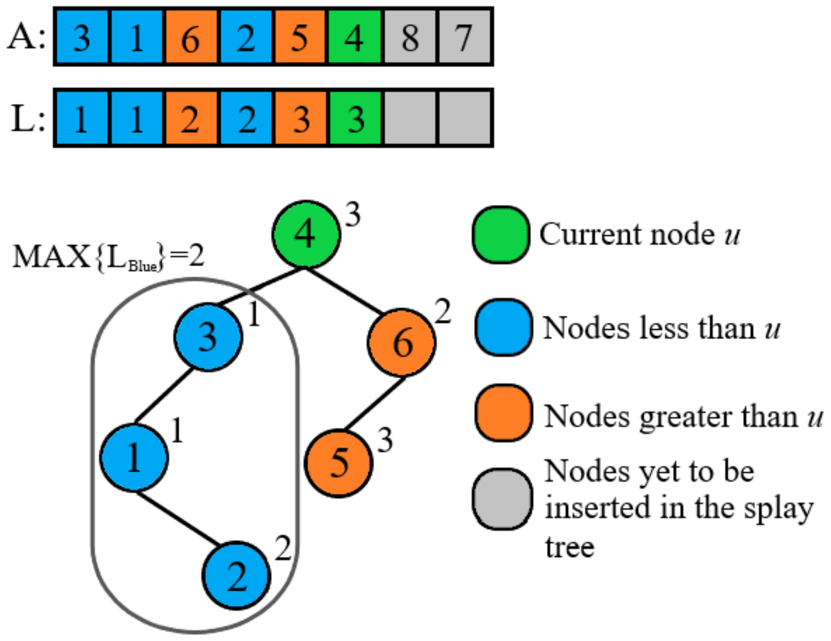

Figure 3.4: Inserting vertices in a Permutation Graph into a Splay Tree

However, we can speed up computing the maximum over all set of solutions j quickly by using a balanced binary search tree. We explain this speed up using splay trees, as they give a very nice way to visualize the process. The splay tree will be keyed on the input array of elements, and we process the input array from left to right. Let $A[i] = x$ in the i^{th} step. We first insert x into the splay tree, and the tree adjusts itself to bring this element to the root. This means everything in x's left subtree will be less than x and everything in x's right subtree will be at least as large as x. Each vertex $u = A[j]$ will be augmented with $L[j]$ and $\max(u)$ which is the maximum over all $L[j]$s in the subtree rooted at vertex u. Since the splay tree's balancing mechanism is based on rotations, we can keep these up to date as we splay and insert new elements in the tree. Once x is splayed to the root, we let its left child be u. Then we can compute $L[i] = \max(u) + 1$, and $\max(x) = \max\{L[i], \max(v)\}$, where v is x's right subtree. Both of these operations take only constant time after x is splayed to the root, which takes $\mathcal{O}(\log n)$ amortized time. The total time is thus, only $\mathcal{O}(n \log n)$.

3.5 MIS on Trapezoid Graphs

Felsner et al. [6] showed a different representation of the trapezoids called as the box representation. An algorithms to compute a MIS is much easier using this representation since we can use a sweep line technique. All the trapezoids in the input have their two parallel sides along two horizontal parallel lines A and B. We can imagine these lines as the x and y axes. Let the two vertices of a trapezoid x that lie on line A be $l(x)$ and $r(x)$, and the two vertices that lie on line B be $b(x)$ and $t(x)$. The point $l(x)$ and $r(x)$ determine the left and right edges of the box, and the points $b(x)$ and $t(x)$ determine its bottom and top edges. Therefore, in the box representation, each trapezoid is only characterized by two points – its left bottom point $lb(x) = (l(x), b(x))$, and its right top point $rt(x) = (r(x), t(x))$. It is easy to see that this transformation from the trapezoid representation to the box representation takes only $\mathcal{O}(n)$ time.

Now, we define an operation on points in this space. Two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$ are comparable if one dominates the other. That is, we say $p < q$ iff $p_x < q_x$ and $p_y < q_y$. Extending this notion to boxes, we have $x < y$ iff $rt(x) < lb(y)$. That is the top right vertex of box x is dominated by the bottom left point of box y. It can be seen that two trapezoids are disjoint iff one dominates the other in the box representation. Therefore, the longest increasing subsequence (LIS) of boxes gives a MIS on trapezoid graphs.

The same algorithm described above for Permutation graphs works with some changes. Firstly, the boxes are processed in the order they appear from left to right. Notice that the main information of the boxes are their bottom left and top right points. Therefore, we can

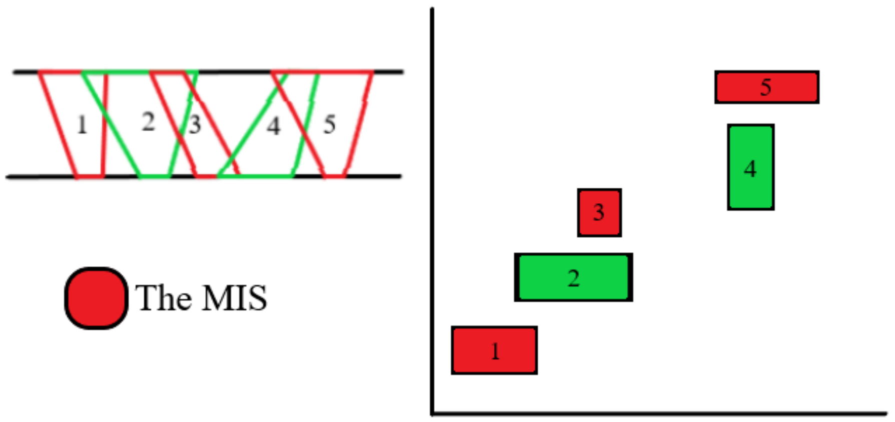

Figure 3.5: A Trapezoid Graph's Trapezoid Representation on the Left, and its Corresponding Box Representation on The Right with Indicated MIS

sort all the "interesting points" based on their x co-ordinates, then scan this set of points in order. Visually, we can imagine a line that sweeps the scene from left to right. Secondly, the splay tree is keyed on the y co-ordinates of the points, and we only store the top right points. This means that whenever the sweep line hits the top right point of any box, we insert this point into the splay tree. Whenever the sweep line hits the bottom left point of any box, we query the data structure to compute $L[i]$, the longest increasing subsequence that ends at box i. For this step, since $s = A[i] = (s_x, s_y)$ is not in the tree when computing $L[i]$, we can splay its successor element $successor(x)$, which is the smallest element in the tree that is larger than x. The entire process only takes $\mathcal{O}(n \log n)$ time.

3.6 MIM on Trees

Now we explore the problem of computing a MIM in each of these five classes of graphs. For trees, the algorithm by Golombic and Luwenstien [13] uses dynamic programming as it processes vertices in a post-order traversal. As we traverse through the graph we will augment each vertex u with three things: $y(u)$, which is a MIM on the entire subtree rooted at u such that no edge in the matching is connected to u, $z(u)$, which is a MIM on the entire subtree rooted at u such that no edge in the matching is connected to u or any of u's children, and $x(u)$, which is a MIM on the entire subtree rooted at u that may or may not have an edge connected to u.

As a base case, if u is a leaf, we have $x(u) = 0$, $y(u) = 0$, and $z(u) = 0$. For the inductive case, we have $y(u) = \sum_{v \in c(u)} x(v)$, where $c(u)$ is the set containing all the children of u. This is true because $x(v)$ can include any edges connecting u's children, but not connecting to u itself. We also have $z(u) = \sum_{v \in c(u)} y(v)$. This again is true because $y(v)$ will not contain any edges connecting to u nor any of u's children. Lastly, to compute $x(u)$, we have $x(u) = \max\left\{ y(u), \max_{v \in c(u)} \left\{ 1 + z(v) + \sum_{w \in c(u), w \neq v} x(w) \right\} \right\}$. That is, $x(u)$ is the larger of two quantities, one in which no edge is connected to u, in which case the answer is $y(u)$, and the other in which u is connected to one child v, in which case we add up one for the edge (u, v), $z(v)$ and add up the best solutions from the rest of the subtrees of u. Notice that $\sum_{w \in c(u), w \neq v} x(w) = y(u) - x(v)$, therefore all quantities can be computed in time proportional to the number of edges of u. So the total time is $\sum_{u} \mathit{degree}(u) = \mathcal{O}(n)$. The final answer is present in $x(r)$, where r is the root of the entire tree.

3.7 MIM on Interval and Circular-Arc Graphs

Recall from Chapter 2 that computing a MIM on a graph G is equivalent to computing a MIS on the corresponding graph $L(G)^2$, where $L(G)$ is the line graph of G. We have the following theorems.

Theorem 1 *(Cameron [1]) If G is an interval graph, then so is $L(G)^2$.*

Theorem 2 *(Golumbic, Laskar [12]) If G is a circular-arc graph, then so is $L(G)^2$.*

Theorem 3 *(Golumbic, Lewenstein [13]) If G is an trapezoid graph, then so is $L(G)^2$.*

So, for each of the above classes of graphs, we can compute a MIS on $L(G)^2$ using algorithms described earlier in the chapter. A small caveat is in producing the set of geometric objects representing $L(G)^2$. In a straightforward fashion, this takes $\mathcal{O}(n^2)$ time, and this is the worst-case. However, we can show how to compute the respective geometric representation in time proportional to both n and m.

To understand why for an interval graph G, $L(G)^2$ is also an interval graph, consider two intersecting intervals u, v in G. In $L(G)^2$, there is a vertex that corresponds to this intersection, we simply denote this using the edge notation (u, v). There is an edge from (u, v) to (u, w) since both vertices share a common vertex u, and edges between (u, v) and (w, x) in $L(G)^2$ if v intersects w for four intervals u, v, w and x in G. These define paths of length two in $L(G)$. In order to get the geometric representation of intervals in $L(G)^2$, we simply take the union of intersecting intervals over all pairs of intervals u and v. This is true for circular-arc graphs too.

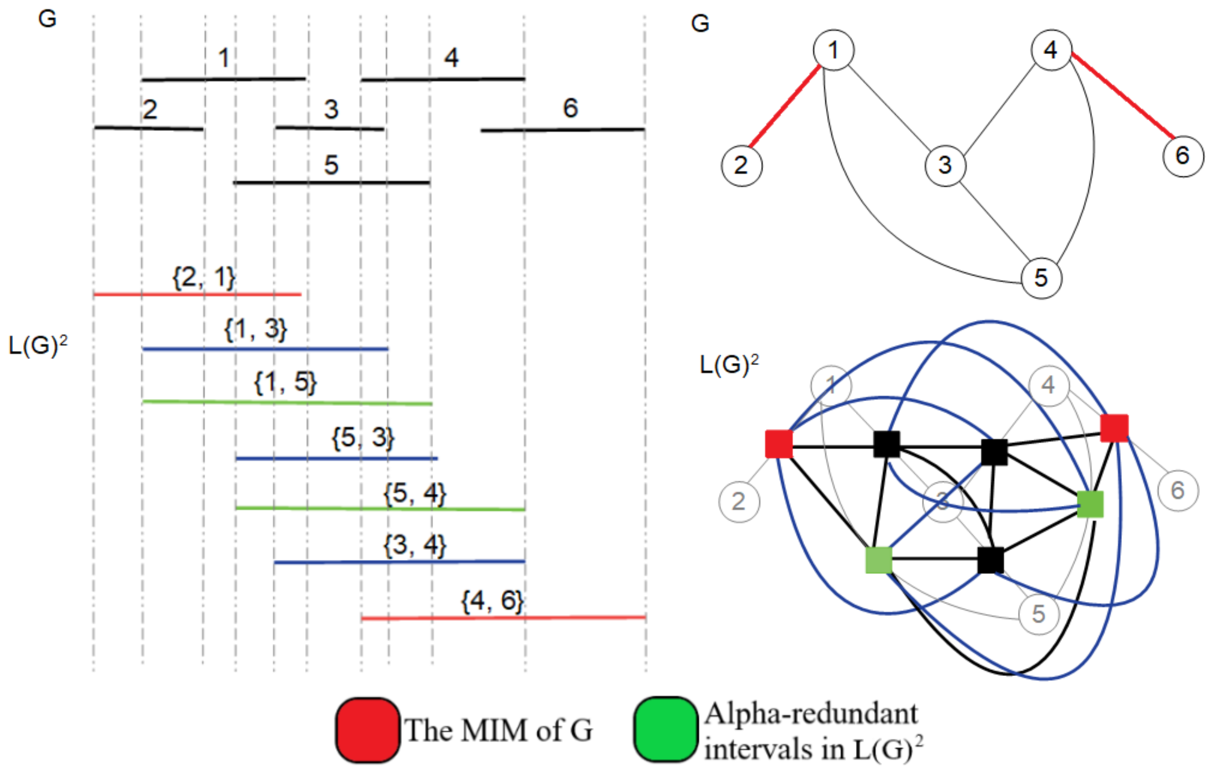

Figure 3.6: A MIM on G Corresponds to MIS on $L(G)^2$. Showing α-Redundant Intervals in Green, and a MIM in Red

It turns out that we do not need the entire set of intersecting intervals (or arcs). We can identify intervals (or arcs) whose removal does not change the MIS. These are called α-redundant intervals. Therefore, for both interval and circular-arc graphs, we use the α-redundant technique described in [5] to remove α-redundant intervals (or arcs), and then use the greedy algorithm described earlier for computing a MIS on the reduced set of intersecting intervals to compute a MIM. Removing the set of α-redundant intervals (or arcs) takes only $\mathcal{O}(n)$ time, so the entire algorithm to compute a MIM on interval and circular-arc graphs is actually $\mathcal{O}(n \log n)$ due to the need to sort the intervals (or arcs).

To find α-redundant intervals, we only need for each interval (or arc) u, its "optimal" interval (or arc) o_u. In the reduced set of intervals S, we have only $u \cup o_u$. All other intervals intersecting with u are removed as they are α-redundant, since they all end later than $u \cup o_u$. The same greedy exchange argument from earlier holds. The following pseudocode determines the set of intervals S that are not α-redundant, which serves as input to the MIS algorithm described before. Here, we denote the left and right endpoints of an interval u as $l(u)$ and $r(u)$ respectively.

```
input  : Set of intervals $V$ sorted based on starting points
output: Array $o$ where $o_u$ is the optimal interval of $u$

$o_u \leftarrow$ null for every $u \in V$
Stack $T = \phi$
for $v \in V$ do
    while $T \neq \phi$ do
        $u \leftarrow T$.top
        if $r(v) < r(u)$ then
            // $v$ is completely contained in $u$
            // So $v$ is the optimal interval of $u$
            $o_u \leftarrow v$
        $T$.pop()
    end
    if $T \neq \phi$ and $l(v) \leq r(T$.top$)$ then
        // $v$ is likely the new optimal interval of $u$
        $o_u \leftarrow v$
    end
    $T$.push($v$)
end
```

Algorithm 1: Finding Optimal Interval of Every Interval $u \in V$

To understand the above psuedocode, we have some interval u on top of the stack, and v is the next interval processed in sorted order. If v starts after u finishes, then no other interval can be overlapping with u, and therefore we can pop u. If v is completeley contained inside u, then no other interval can be an optimal interval of u. Therefore we update u's optimal interval to v and pop u. Otherwise, currently u's optimal interval is v, and we update this but do not pop u as u could have a better optimal interval that is explored later. That is why we put u on the stack. Note that any later interval that overlaps with u also overlaps with v, so when v gets popped off, we have the chance to update u's optimal interval as well.

The above α-redundant algorithm is applicable for circular-arcs as well. However, we need to start from the smallest arc. So as a pre-processing step, we need to find the smallest arc u, let $l(u)$ be the origin as we sort all points from here in clockwise direction. For both interval and circular-arc graphs, this sorting step is the bottleneck, therefore it takes $\mathcal{O}(n \log n)$ time to find a MIM on both of these graph classes.

3.8 MIM on Permutation and Trapezoid Graphs

We first note that permutation graphs are a subset of trapezoid graphs. This is because, each line segment x connecting i and $\pi(i)$ can be transformed into a trapezoid with $l(x) = r(x) = i$ and $b(x) = t(x) = \pi(i)$. In the geometric representation of the trapezoids, these will be single points instead of boxes. This transformation takes only linear time. We can then apply the algorithm for computing a MIM on trapezoid graphs as explained below.

From Theorem 3, we have if graph G is a trapezoid graph, then so is $L(G)^2$. As mentioned previously, straightforward transformations of the trapezoids in G to $L(G)^2$ could take $\mathcal{O}(n^2)$. But we can make the running time dependent on m and n.

We first find the set of trapezoids in $L(G)$. This corresponds to all pairs of intersecting trapezoids in G. We use the box representation and a sweep line algorithm. We can sweep the set of boxes from left to right. When the sweep line hits the left edge, we insert that trapezoid into a set data structure (say S), and when the sweep line hits the right edge, we remove the corresponding trapezoid from S. So at each step, the set S contains the set of active trapezoids. We compute intersections for a trapezoid x at both its left and right edges. When the sweep line hits x's left (or right) edge, the set S contains the set of active intervals that intersect with x. We can simply iterate over the set S and create trapezoids (x, y) by "unioning" them, that is, we create a new trapezoid (x, y) where the left vertex is at $l(x, y) = \min\{l(x), l(y)\}$, the right vertex at $r(x, y) = \max\{r(x), r(y)\}$, top vertex at $t(x, y) = \min\{t(x), t(y)\}$, and bottom vertex at $b(x, y) = \max\{b(x), b(y)\}$. This actually covers all types of intersections, because each intersection between (x, y) is explored when the sweep line hits the left (or right) edge of x, or y, or both. So, each intersection is only explored at most $4m$ times. We do the same algorithm by sweeping a line from top to bottom and now each intersection is found at most $8m$ times.

Exploring all of these intersections and creating the set of intersecting trapezoids takes only $\mathcal{O}(m)$ time in total after sorting. So the entire algorithm takes $\mathcal{O}(m + n \log n)$ time.

In order to compute the square of a trapezoid graph, we first have the following theorem as a corollary from Flotow [7].

Theorem 4 *(Flotow [7]) If G is a trapezoid graph, then so is G^2.*

Therefore, given a trapezoid representation of G, we would like to compute a trapezoid representation of G^2. This will be the set of boxes where each box x in G is replaced with x' in G^2, where x' is the union of all boxes intersecting with x. First, notice that there is a one-to-one correspondence between vertices in G and vertices in G^2 – each vertex x is replaced with x'. Therefore, if two boxes x and y intersect in G, their corresponding boxes x' and y' also intersect. These are edges in G. But G^2 also contains edges in G that are connected by paths of length 2. Let box x intersect y and y intersects z in G. Since both x' and z' contain y, we have an edge from x' and z' in G^2.

To find the set of boxes in G^2, we can use the same sweep line algorithm described above with the some changes. We implement the set S using a min-priority queue keyed on the lower edge of the boxes. As we sweep from left to right, when we hit a box x at either left or right edges, we can compute $b(x') = \min_{y \in \{S \cup \{x\}\}} \{b(y)\}$. Moreover, we also update for each box $y \in S$, $b(y') = \min\{b(y'), b(x)\}$. Similarly, using a max-priority queue keyed on the top edges, we can compute $t(x')$ for all boxes. Also, sweeping from bottom to top gives us, and keying on left and right edges gives us $l(x')$ and $r(x')$ respectively. This process takes the same time as the above algorithm, which is $\mathcal{O}(m + n \log n)$.

The above algorithm can be sped up using a range query data structure like a splay tree [1]. As mentioned previously, a splay tree can find an aggregate statistic over elements in a subtree. Let us consider the problem of finding $b(x')$. For each box x in G, we have a range of values determined by $l(x)$ and $r(x)$, which correspond to the the left and right edges respectively. Therefore we can sort all the $2n$ left and right edges, and build a splay tree keyed on the x axis of the edges. Each edge corresponding to a box x is augmented with $b(x)$. Each

[1] Other binary search trees also work. Segment trees may be the simplest in terms of implementation, but we describe using splay trees as this is more intuitive and similar to the speed up of the longest increasing subsequence problem described earlier.

subtree u in the splay tree is also augmented with $\min(u) = b(y)$ over all elements y in u's subtree. Now, for each box x in the input, we perform a range minimum query between $l(x)$ and $r(x)$, which returns the minimum (or the bottom edge) of all boxes contained within the desired range. This is done by splaying $l(x)$ first, and then $r(x)$ making sure it knocks $l(x)$, thereby isolating everything in the desired range in the red subtree (see Figure 3.7). Note that there could be two entries for the same box in the range if both its edges are contained within it. Since we only want the minimum over all boxes in the range, this does not affect the minimum. Therefore, for each box x, we can compute $b(x')$ in only $\mathcal{O}\log n)$ time. Similarly we can compute the other three edges of x'. The total time is thus $\mathcal{O}(n \log n)$.

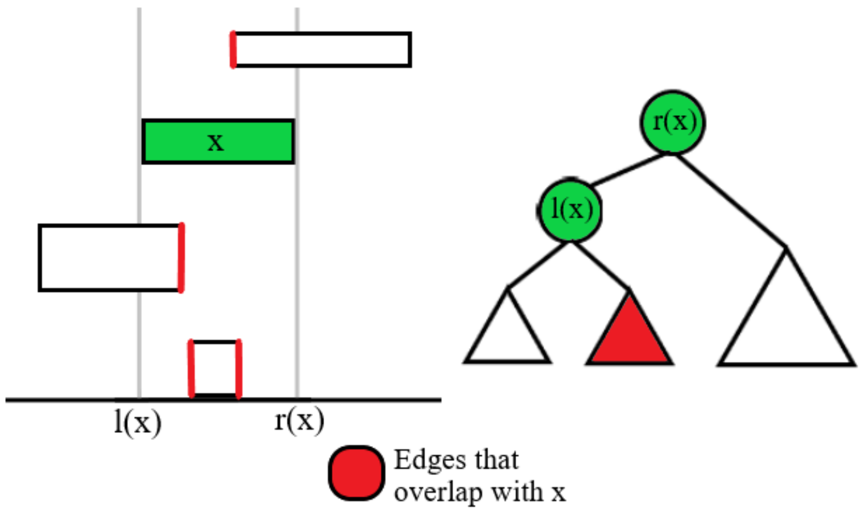

Figure 3.7: A Splay Tree of Left and Right Edges. Subtree in Red Contains Some Boxes That Overlap with Box X

There is a small caveat in using splay trees. It captures all intersecting boxes y with x, where either $l(y)$ or $r(y)$ or both are within the desired range (between $l(x)$ and $r(x)$). But it does not include boxes where both are outside the range yet intersecting with x. This happens when $l(y) < l(x)$ and $r(y) > r(x)$. The algorithm described above that uses a sweep line and priority queues can be used. Here, we do not need to update all the elements in the subtree, but only need to update $b(x') = \min_{y \in \{S \cup \{x\}\}} \{b(y)\}$, which can be computed in only $\mathcal{O}(\log n)$ time using a priority queue. Combining both ideas of priority queues and splay trees, the entire algorithm to compute trapezoids in G^2 is only $\mathcal{O}(n \log n)$.

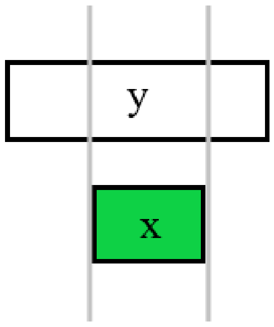

Figure 3.8: An Example where Box y will not be in the Desired Subtree when Querying x's Intersections in the Splay Tree

To summarize our algorithm to compute a MIM on a trapezoid graph G, we first compute its trapezoid representation in $L(G)$ in $\mathcal{O}(m + n \log n)$. Then we obtain its trapezoid representation in $L(G)^2$ in $\mathcal{O}(m \log m)$ because the number of vertices in $L(G)$ is m. Now, we can use the algorithm for computing a MIS in $L(G)^2$. The total time is thus $\mathcal{O}(m \log m + n \log n) = \mathcal{O}(m \log n^2 + n \log n) = \mathcal{O}(m \log n + n \log n) = \mathcal{O}((m + n) \log n)$. This is only $\mathcal{O}(n^2 \log n)$ in the worst-case, and is currently the fastest algorithm for computing a MIM on both permutation and trapezoid graphs.

Chapter 4

Competitive Programming Problems

In this chapter, we describe a few competitive programming problems on intersection graphs. Problems 1, 2 and 4 are original, while the others are slightly adapted from Do et al. [5]. We greatly improve on their presentation.

4.1 Gifts Giving

The employees of a company are planning two events to socialize and bond with one another before the end of the year holidays – one to celebrate Thanksgiving, and the other to celebrate Christmas. There are lots of gifts being exchanged during these times. In order to facilitate stronger bonds between workers and their immediate bosses in the organization, a group of employees are selected to gift their immediate bosses or subordinates during Thanksgiving, while they will receive gifts from them during the Christmas event. Each person in the company answers to only one immediate boss, but a boss may have multiple subordinates. Every person has an immediate boss, except the owner of the company.

In order to keep these events exciting, it was decided during the Thanksgiving event that some employees will be the gift givers, while others will be the receivers (the receivers will return the favor during Christmas). Every employee is in exactly one of the two groups – so if a person received a gift, then this person will not be giving any gift and vice-versa during the Thanksgiving event. Note that a person may receive multiple gifts, but each gift giver only gifts one gift during the Thanksgiving event (this means that people who received multiple gifts will have to reciprocate giving multiple gifts during Christmas). Since we like gifts, please determine the largest number of gifts that are given during the Thanksgiving event, along with a description of the employees that are the gift givers.

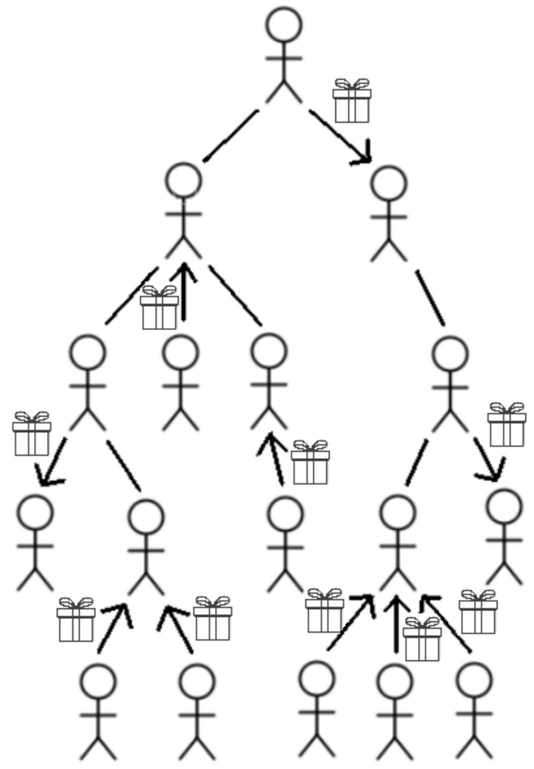

Figure 4.1: An Example Matching of Gifts

Input Format

The first line of the input is the integer $n, n \leq 10^6$, the number of employees in the company. Person 1 is the designated owner of the company and the only person who does not have an immediate boss. All other employees are labeled $2, \ldots, n$. Then several lines follow that describe the employee structure. Specifically, each line contains two integers x and y where employee x is the immediate boss of y.

Sample Input

```
17

1 2

1 3

2 4

2 5

2 6

3 7

4 8

4 9

6 10

7 11

7 12

9 13

9 14

11 15

11 16

11 17
```

Output Format

Please print the total number of gifts g that are exchanged during the Thanksgiving event on the first line. On the second line, please print a space separated list of employees that are the gift givers. Please sort this list in descending order. Note that there may be multiple solutions, so you only need to print one.

Sample Output

```
 10

 17 16 15 14 13 10 7 5 4 1
```

Solution

To solve this problem we convert the input into a tree and find a MIS on it. This is because gift givers can not also receive gifts, so gift givers must be an independent set.

4.2 Relaxed Scheduling

Janet Brock is attending a theme park with many attractions – which includes games, rides, water-rides, activities like swimming, trekking, camping, movies and theater, and lots of places for eating and shopping. Janet is given a schedule of all events, where each event is determined by its starting and ending times. All times are measured in seconds since the start of the day. She can only choose to attend some of these events as some events overlapping, and she wants to participate in events from its start to end. Given the schedule of events, please determine the largest number of events she can attend.

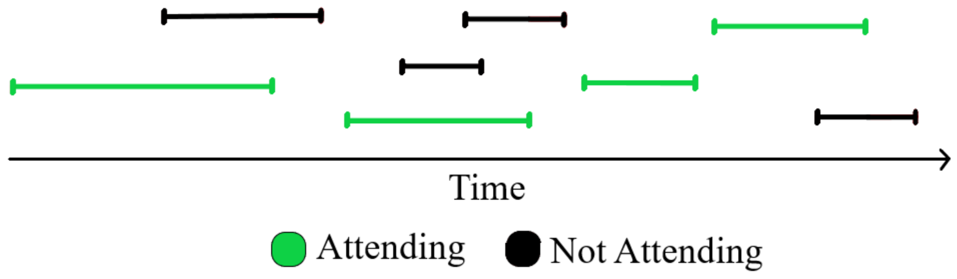

Figure 4.2: An Example Activity Schedule

Input Format

The first line of input contains the integer $n, n \leq 10^6$, the number of events. Then n lines describe each event, where each line contains only two integers a and b, $a < b$ which are the starting and ending times respectively for an event.

Sample Input

```
8
0 10
5 12
13 18
15 17
```

```
16 19

20 23

24 28

26 29
```

Output Format

Please print a single integer that is the largest number of events Janet can attend.

Sample Output

```
4
```

Solution

To find the largest number of events Janet should attend, we simply find the cardinality of a MIS on the input interval graph.

4.3 Circular Arc

You are given a circle defined by its center (x_c, y_c) and radius r, and n lines, where the i^{th} line is determined by the equation $a_i x + b_i y + c_i = 0$. All of these line segments either intersect the circle at two different points A and B, which forms a secant AB, or intersects at one point and is tangent to the circle. If a line cuts the circle at two points, the smaller arc AB formed by the points is called the characteristic arc of that line. It is guaranteed that no line passes through the center of the circle.

To examine the relationship between the arcs, Janet Brock builds a simple undirected graph $G = (V, E)$, where each vertex of V corresponds to a characteristic arc, and there is an edge between two vertices (u, v) if the corresponding characteristic arcs overlap (see figure below). A path of length d between two edges e and f is the sequence of edges $[e, g_1, \ldots, g_d, f]$, where each two adjacent edges in the sequence contains a common vertex. Therefore one can "walk" from the edge e by following the edges in the sequence to edge f. If there is no path between edges e and f, their path length is set to $+\infty$. Janet would like to find a largest set of edges $E' \subseteq E$ such that the distance between any two edges in E' is at least 2.

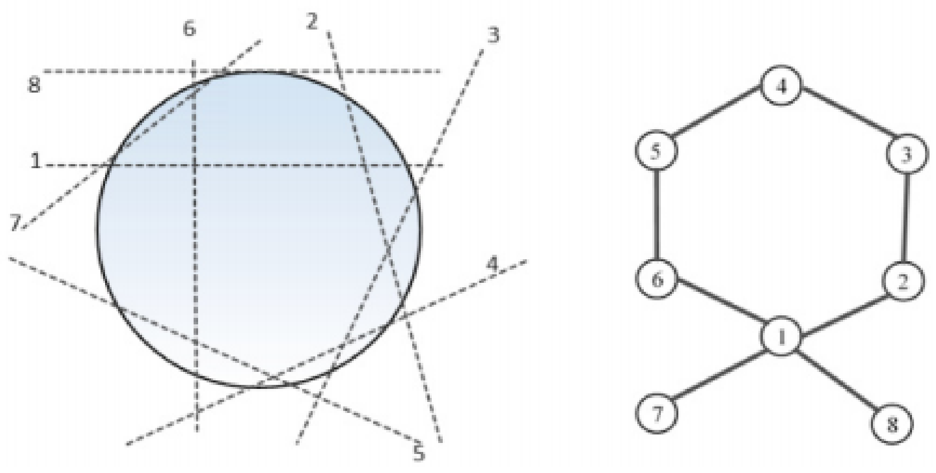

Figure 4.3: An Example of The Arcs and Corresponding Circular-Arc Graph from Do et al. [5]

Input Format

The first line of input contains 4 integers $n, n \leq 10^6$, x_c, y_c and r as described above. Then n lines follow each describing a line, where line i contains three integers a_i, b_i and c_i. It is guaranteed that each of these lines touches the circle at either two points or at one point.

Sample Input

```
8 0 0 10

0 -1 6

-12 -3 105

18 -4 -160

1 -2 -18

-1 -4 -36

-20 -1 -60

2 -3 33

0 -1 10
```

Output Format

Please output a single line that is the largest set of edges in the graph described above such that the distance between two edges in the set is at least 2.

Sample Output

```
2
```

Solution

The problem description clearly states that the solution is to find a MIM on a circular-arc graph. Note that we do not need to construct the graph, instead can find the set of arcs from the input lines, and compute a MIM from the arcs.

4.4 Trapezoid Shooting

Janet Brock is playing her favorite shooting game where objects in the shape of trapezoids zip through the screen from right to left, and the objective is to shoot as many of these as possible. Specifically, there are two parallel horizontal lines A and B, and each trapezoid contains the two parallel edges on these two lines. So each trapezoid is described using four integers $(u, v, w, z), u < v, w < z$ where u and v are the x co-ordinates of the two vertices on the bottom line A, and w and z are the x co-ordinates of the two vertices on the top line B. It is assumed that all of these integers are positive, and the y axis is the leftmost edge of the screen.

Some of these trapezoids may be overlapping, and therefore lie behind one another. When Janet shoots at a trapezoid, she always hits the one at the front. This knocks out not only the trapezoid she hits, but also all trapezoids that intersect with it (and therefore lie behind it). Note that this is true even if the two trapezoids are intersecting at only a very tiny portion, and that she only shoots at whatever trapezoid is at the front.

Janet hacks into the computer code of the program and knows the positions of all the trapezoids in advance. She is skillful enough in the game to bring the desired trapezoids to the front before shooting them. Janet likes to shoot at the largest number of trapezoids. Please determine the largest number of shots she will need to make in order to knock down all the trapezoids.

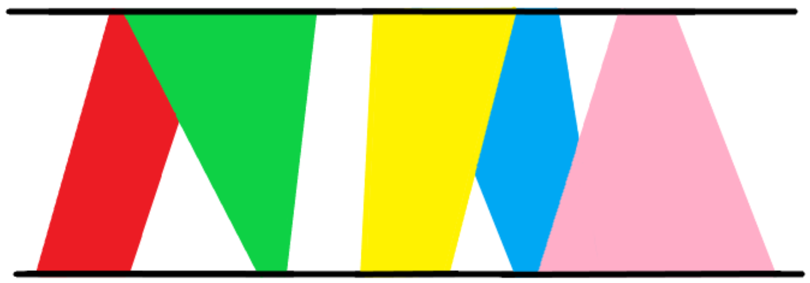

Figure 4.4: An Example of The Overlapping Trapezoids in the Game

Input Format

The first line of input contains a single integer $n, n \leq 10^5$. We will assume that the parallel line A is at $y = 0$ and the line B is at $y = 10$ which are the bottom and top parts of the screen. Next n integers follow that each describe a trapezoid. Line i contains four integers u_i, v_i, w_i and z_i which are the four co-ordinates of the i^{th} trapezoid as described above.

Sample Input

```
5

2 4 1 3

3 6 4 5

7 10 6 8

7 11 9 12

12 14 10 16
```

Output Format

Please print a single integer that is the largest number of trapezoids she will need to shoot in order to knock down all the trapezoids in the game.

Sample Output

```
3
```

Solution

To find the maximum number of trapezoids Janet can shoot we need to find a MIS on the trapezoid graph. This is because if the trapezoids are overlapping they are connected in the graph, and if two trapezoids overlap when we shoot one it knocks over the other. So the largest set of unconnected trapezoids is the highest number of trapezoids Janet can shoot.

4.5 Navigating a River

Janet Brock is trying to navigate a river on her boat. The river is determined by two parallel lines $x = a$ and $x = b$, $a < b, a, b \in Z$. There are n obstacles nailed on the river, where obstacle i is nailed at integer co-ordinates (x_i, y_i). The boat is modeled as a circle with some diameter, and therefore when maneuvering between two obstacles A and B, the diameter of the boat has to be at most the Euclidean distance between A and B. The boat starts from the southernmost point defined by the parallel line $y = c$ and needs to reach the northernmost point defined by the line $y = d$. Janet would love to take her pets on this boat ride, and in order to fit all of her pets on the boat, she would like to find out the largest diameter D of a boat she will need in order to successfully be able to maneuver the entire river without being stuck in-between two obstacles.

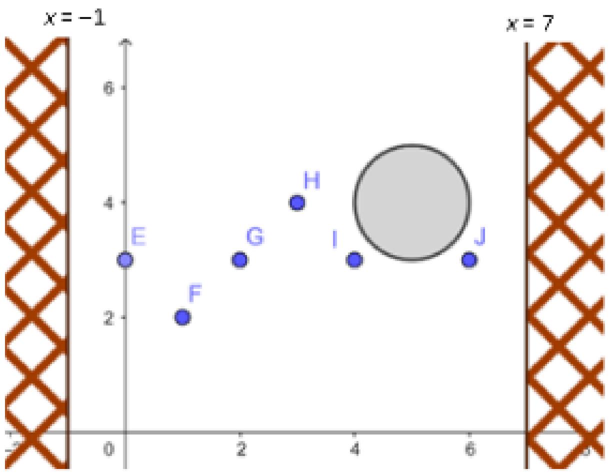

Figure 4.5: An Example of The River with the Boat and Obstacles in It from Do et al. [5]

Input Format

The first line contains four positive integers a, b, c, d and $n, n \leq 10^5$ as described above. We have $a < b$ and $c < d$. Next, n lines follow that each describe an obstacle, where obstacle i is given by two integers x_i and y_i.

Sample Input

```
1 7 0 5 6

0 3

1 2

2 3

3 4

4 3

6 3
```

Output Format

Please print a single integer D that is the largest diameter of the circular boat she will need to maneuver the entire river from the bottom to the top.

Sample Output

```
1
```

Solution

First observe that we can transform the problem of maneuvering a circle that represents the boat and pointed obstacles, to the problem of maneuvering a point with circled obstacles, by changing each obstacle to a circle with some diameter d. We can now construct a graph G where each node is an obstacle, and edges connect obstacles that intersect. That is, this is an intersection graph of circles. Notice that if two obstacles x and y are connected, then the boat cannot maneuver between them. We also add two special nodes l and r to represent the left and right banks, and have edges from them to obstacles that intersect with them.

We can use sweep line techniques to build this graph efficiently. If there is a path in the graph from l to r, the diameter of the ship is too big and Janet will not be able to cross. We can use depth first search from node l for this. To find the largest diameter of the ship that can maneuver the river, we binary search for it between the values 0 and the width of the river.

4.6 Building

For aesthetic reasons, and for attracting tourists, the city in which Janet Brock lives has buildings in nice rectangular shapes where each side is always parallel to the two co-ordinate axes. Therefore, each building i is represented by a 4-tuple (l_i, r_i, b_i, t_i) where l_i and r_i define the left and right edges respectively, and b_i and t_i define the bottom and top edges respectively. Note that l_i and r_i are parallel to the y-axis while b_i and t_i are parallel to the x axis. A building u is adjacent to another building v if the intersection of their sides is non empty. There is a short scenic walkway between pairs of adjacent buildings. After going on several of these walkways, Janet realizes that some walkways between buildings u and v are "unique". A walkway between buildings u and v is unique if after going from u to v, there is no way to come back to u without going back the same walkway. Please determine all such unique walkways.

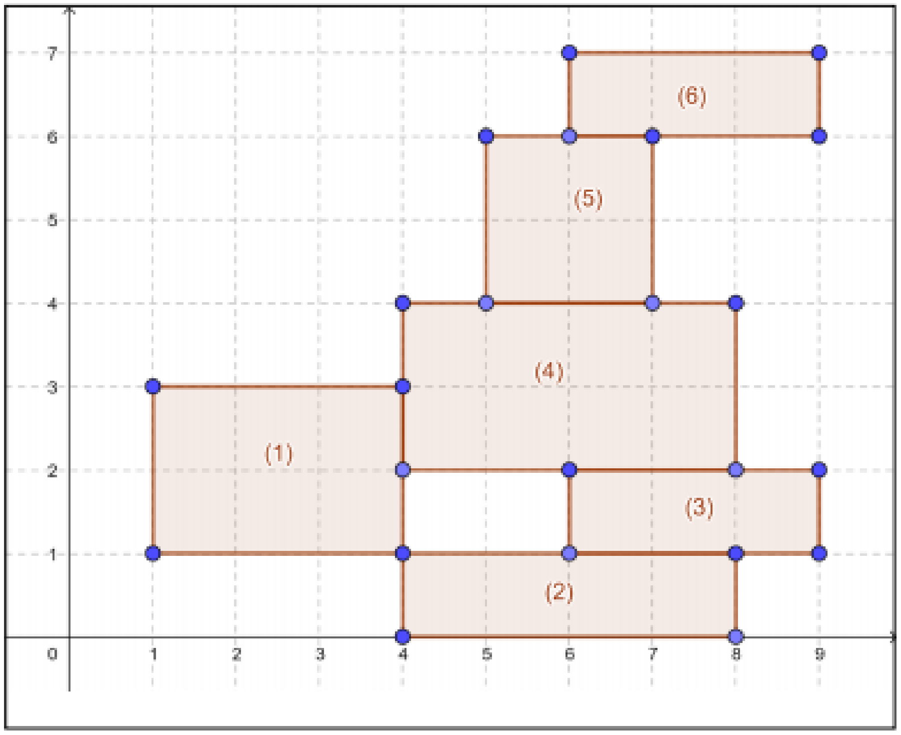

Figure 4.6: An Example of the Buildings on a Grid from Do et al. [5]

Input Format

The first line contains a single integer n that is the number of buildings in the city. Next, n lines follow that each describe a building. The i^{th} such line contains four integers l_i, r_i, b_i and t_i. All buildings are numbered 1 to n.

Sample Input

```
6
1 4 1 3
4 8 0 1
6 9 1 2
4 8 2 4
5 7 4 6
6 9 6 7
```

Output Format

Print the number of unique walkways x on the first line. Next x lines follow that describe the unique walkway between two buildings. The i^{th} such line contains the two buildings u_i and v_i such that $u_i < v_i$. Please order these walkways based on u_i's and if they are equal, then based on v_i's. A unique walkway between a pair of buildings is printed only once in the output.

Sample Output

```
2
4 5
5 6
```

Solution

It can be seen that this graph is an intersection graph of axially aligned rectangles. We can construct this graph using sweep line techniques. A unique walkway between buildings x and y corresponds to a bridge in the graph, which is an edge whose removal splits the graph into two pieces. To find all bridges, we can use the popular algorithm of Tarjan that is based on modified depth-first search.

Chapter 5

Conclusion

There is much material for competitive programming in graph theory. In this paper we were able to explore ten algorithms for finding a MIS and MIM on five special graph classes – trees, interval graphs, circular-arc graphs, permutation graphs, and trapezoid graphs. Although algorithms for computing a MIM on trapezoid graphs are known, we use advanced data structures to provide a clear runtime of $\mathcal{O}((m+n)\log n)$. Our algorithm can be extended to find a k-induced matching on both permutation and trapezoid graphs. Moreover, we can also generalize this for graphs in higher dimensions, like that of k-trapezoid and k-circular trapezoid graphs. We also presented six competitive programming problems based on geometric intersection graphs, three of which are original.

Future work could expand upon this paper to include more graph classes such as comparability graphs, co-comparability graphs, and chordal graphs. Future work could also expand upon the graph problems and include problems such as finding a maximum cardinality clique (MC), a minimum clique cover (MCC), and graph coloring. More competitive programming problems could also be written; not just related to the graph theory topics in this paper, but related to any and all areas of graph theory.

Competitive programming is growing rapidly and graph theory is rich with material for contest problems. However, there is substantive work needed to be done to make these topics more accessible to undergraduate students, and students training for competitive programming. We hope this work inspires more contest writers to write more problems on specific graph classes, and graph theory researchers to write more accessible exposition of graph theory topics to undergraduate students, thereby bridging the gap between competitive programming and graph theory. We plan to write more exposition on these topics ourselves in the future.

Bibliography

[1] K. Cameron, Induced matchings. Discrete Appl. Math. 24 (1989), 97-102.

[2] E. J. Cockayne, S. T. Hedetniemi, A linear algorithm for the maximum weight of an independent set in a tree. Proc. Seventh S. E. Conf. on Combinatorics, Graph Theory and Computing, Util. Math., Winnipeg, 1976, 217-228.

[3] I. Dagan, M. C. Golumbic, R. Y. Pinter, Trapezoid graphs and their coloring. *Discrete Appl. Math.* 21 (1988), 35-46.

[4] D. E. Daykin, C. P. Ng, Algorithms for generalized stability numbers of tree graphs. *J. Austral. Math. Soc.* 6 (1966), 89-100.

[5] P. T. Do, B. T. Pham, V. C. Than, Latest algorithms on particular graph classes. Olympiads in Informatics 14 (2020), 21-35.

[6] S. Felsner, R. Muller, L. Wernisch, Trapezoid graphs and generalizations, geometry and algorithms. Cornell Family Papers, 1997.

[7] Carsten Flotow, On Powers of M-trapezoid Graphs, *Discret. Appl. Math* 63 (2) 1995, 187-192.

[8] M. Fredman, On computing the length of longest increasing subsequences. Discret. Math 1 (1975), 29-35.

[9] G. Fricke, R. C. Laskar, Strong matchings on trees. Congr. Numer. 89 (1992), 239-243.

[10] F. Gavril, Algorithms on circular-arc graphs. Networks 4 (1974), 357-369.

[11] M. C. Golumbic and P. L. Hammer, Stability in Circular Arc Graphs, J. Algorithms 9 (3) 1998, 314-320

[12] M. C. Golumbic, R. C. Laskar, Irredundancy in circular arc graphs. Discrete Appl. Math. 4 (1993), 79-89.

[13] M. C. Golumbic, M. Lewenstein, New results on induced matchings. Discrete Appl. Math. 101 (2000), 157-165.

[14] Hsiao and Tang and Chang, An Efficient Algorithm for Finding a Maximum Weight 2-Independent Set on Interval Graphs, IPL: Information Processing Letters 43 (1992).

[15] W. Hsu and K. Tsai, Linear Time Algorithms on Circular-Arc Graphs, Inf. Process. Lett. 40 (3) 1991, 123-129.

[16] The International Olypiad for Informatics, `https://ioinformatics.org/?r=301`.

[17] S. L. Mitchell, *Linear Algorithms on Trees and Maximal Outerplanar Graphs: Design, Complexity Analysis and Data Structures Study*, Ph.D. thesis, University of Virginia, 1977.