
FINDING TRANSITIVE SUBGRAPHS AND COUNTING POPULAR
MATCHINGS

By

NITESH JHA

*A thesis submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy*

to

Chennai Mathematical Institute

July 2017



Plot No-H1, SIPCOT IT Park,
Kelambakkam 603103
Tamilnadu, India

DECLARATION

I declare that this thesis titled, "Finding Transitive Subgraphs and Counting Popular Matchings" submitted by me for the degree of Doctor of Philosophy is the record of work carried out by me during the period from August 2010 to August 2016 under the guidance of Prof. Sourav Chakraborty. This work has not formed the basis for the award of any degree, diploma, associateship, fellowship, titles in this or any other university or other similar institution of higher learning.

Nitesh Jha

July 2017
Chennai Mathematical Institute
Plot H1, SIPCOT IT Park, Siruseri,
Kelambakkam 603103
India

CERTIFICATE

This is to certify that the Ph.D. thesis submitted by Nitesh Jha to Chennai Mathematical Institute, titled "Finding Transitive Subgraphs and Counting Popular Matchings" is a record of bona fide research work done during the period August 2010 to August 2016 under my guidance and supervision. The research work presented in this thesis has not formed the basis for the award of any degree, diploma, associateship, fellowship, titles in this institute or any other university or institution of higher learning. It is further certified that the thesis represents independent work by the candidate and collaboration when existed was necessitated by the nature and scope of problems dealt with.

Sourav Chakraborty
(Thesis supervisor)

July 2017
Chennai Mathematical Institute
Plot H1, SIPCOT IT Park, Siruseri,
Kelambakkam 603103
India

Abstract

Optimization problems on graphs are central to theoretical computer science and have many practical applications. Of special interest are optimization problems that are computationally hard. Since NP-hard optimization problems aren't expected to be solved efficiently, many approaches have emerged in tackling them. Obtaining *approximate* solutions to these NP-hard problems is a central challenge in this field. When it's not easy to obtain a deterministic algorithm for approximation, we use randomized algorithms. While NP-hardness captures the complexity of problems that have a 'yes' or 'no' answer, the class #P-hard encompasses problems where one counts the number of solutions itself. Obtaining randomized approximation algorithms is also the main goal in the field of counting problems.

Another approach is to give an *exact* algorithm with some parameter of the problem fixed. The goal here is to design algorithms which run in time polynomial in parameter which is assumed to be fixed. Such problems are called fixed-parameter tractable.

The first part of the thesis is concerned with optimization problems arising out of transitivity in binary relations. The binary relation is naturally viewed as a directed graph and the associated optimization problems can then be posed in terms of graph theoretic problems on the underlying graph. The transitivity structure in a binary relation is a fundamental object that has a rich history in multiple areas of mathematics and computer science. Since transitivity is a desired structure, it is approached in multiple ways. Two most common are transitive closures and transitive subgraphs. The problems can then be posed in the form of an optimal or approximate solution. Problems have also been studied under the notion of *distance* from a transitive structure.

We present an algorithm that, given a directed graph on n vertices and m arcs outputs a maximal transitive subgraph in time $O(n^2 + nm)$. In the case of Maximum Transitive Subgraph (MTS) problem, we give a 0.25-approximation for the general problem and a 0.874-approximation for the triangle-free case (underlying graph being triangle-free). We also give an upper bound on the size of MTS being $m/4 + cm^{4/5}$ for some $c > 0$. Further, we give exact algorithms for the MTS problem in different settings. We prove that the MTS problem is Fixed-parameter tractable when parameterized by treewidth.

In the later part of the thesis, we study the problem of *Popular Matchings*. Here our goal is to count the number of popular matchings in a given instance. We give some hardness and approximation results for the same.

Acknowledgements

I thank my advisor Sourav Chakraborty for being always there for me. My research journey started with him and we continue to collaborate on many of our shared interests. His guidance in matters both technical and non-technical has been of immense help. I was very lucky to get him as my advisor and later as a friend.

I have a special mention for my friend and collaborator Rupam Acharyya. I spent the earlier part of my PhD in constant discussions with him and those are some of the best memories of my time here. I must also mention my wonderful collaborators Shamik Ghosh and Sasanka Roy. Working with them has been a complete joy.

I thank the computer science faculty at CMI and IMSc from whom I have learned immensely but I must mention Samir Datta and K V Subrahmanyam in particular. They were always available to me whenever I needed any help. I also want to thank Partha Mukhopadhyay for being one of the best teachers I have ever met.

I want to thank CMI for funding my studies and providing the grants for research activities. The CMI administration is extremely efficient and open, something worth emulating everywhere.

In my stay here, I have met so many wonderful graduate students at CMI and IMSc. Many of you are now my lifelong friends! This space is too small to list your kindnesses. I thank you all.

*Nitesh Jha
Chennai,
December 2016.*

Contents

Declaration of Authorship	iii
Certificate	v
Abstract	vii
Acknowledgements	ix
Contents	ix
List of Figures	xiii
1 Introduction	1
I Transitivity in Relations	7
2 Introduction to Transitivity	9
3 Maximal Transitive Subgraph	11
3.1 Maximal transitive relation finding algorithms	11
3.1.1 $O(n^3)$ algorithm for finding maximal transitive sub-relation	12
3.1.2 Proof of Correctness of Algorithm 1	12
3.1.3 Better running time analysis of Algorithm 1	19
4 Maximum Transitive Relation	21
4.1 Approximation	21
4.2 Maximum directed cuts in triangle free graphs	23
4.3 Upper Bound on Directed Max-Cut	24
5 Exact Algorithms for Maximum Transitive Subgraph Problem	27
5.1 Introduction	27
5.2 MTS in Trees	28
5.3 MTS in Bounded Treewidth Graphs: First Attempt	31
5.4 MTS is FPT Parameterised by Treewidth	34
5.5 Conclusion	38

II Popular Matchings	39
6 Introduction to Popular Matchings	41
7 Counting in House Allocation Problem with Ties	45
8 Counting in Capacitated House Allocation Problem	49
8.1 Switching Graph Characterization of CHA	50
8.2 Hardness of Counting	54
9 Conclusion	57
9.1 Transitivity	57
9.2 Popular Matching	58
Bibliography	59

List of Figures

- 3.1 Diagrams of the two cases for Lemma 3.4 14
- 3.2 Diagram for subcases of Case 1 for Lemma 3.4 15
- 3.3 Diagram for Lemma 3.5 17
- 3.4 Diagrams of the two cases for Lemma 3.6 18

- 5.1 Computation of $MTS^+(T_{e_i})$ 'down' edge case: we only include the MTS of rooted trees in the dashed rectangles 29
- 5.2 A high level description of Algorithm 3 33

- 7.1 Gallai-Edmonds decomposition of the first-choice graph of G 46
- 7.2 Reduction to a perfect-matching instance by extending the Gallai-Edmonds decomposition of G_1 48

List of Algorithms

1	Finding a maximal transitive sub-relation	12
2	Finding a maximal transitive sub-relation	20
3	MTS(G, w): Maximum Transitive Subgraph of weighted digraph G	32
4	MTS(G): Maximum Transitive Subgraph of digraph G	35

Chapter 1

Introduction

Optimization problems on graphs are central to theoretical computer science and have many practical applications. Of special interest are optimization problems that are computationally hard. Since NP-hard optimization problems aren't expected to be solved efficiently, many approaches have emerged in tackling them. Obtaining *approximate* solutions to these NP-hard problems is a central challenge in this field. When it's not easy to obtain a deterministic algorithm for approximation, we use randomized algorithms. While NP-hardness captures the complexity of problems that have a 'yes' or 'no' answer, the class #P-hard encompasses problems where one counts the number of solutions itself. Obtaining randomized approximation algorithms is also the main goal in the field of counting problems. An important class of such algorithms is *fully polynomial-time randomized approximation scheme* or FPRAS. Jerrum and Sinclair [JS89] gave the first such algorithm for approximately-counting the number of perfect matchings in a graph (of high average degree). Later Jerrum et al [JSV01] generalized this to graph with vertices of any degrees. This problem is equivalent to the problem of approximating the permanent.

Another approach is to give an *exact* algorithm with some parameter of the problem fixed. The goal here is to design algorithms which run in time polynomial in parameter which is assumed to be fixed. Such problems are called fixed-parameter tractable. First systematic study of parameterized complexity was done by Downey and Fellows [DF99]. More recent account of the field can be found in the texts [FG06, Nie06, CFK⁺15].

The first part of the thesis is concerned with optimization problems arising out of transitivity in binary relations. The binary relation is naturally viewed as a directed graph and the associated optimization problems can then be posed in terms of graph theoretic problems on the underlying graph. The transitivity structure in a binary relation is a fundamental object that has a rich history in multiple areas of mathematics and computer science. Since transitivity is a desired structure, it is approached in multiple ways. Two most common are

transitive closures and transitive subgraphs. The problems can then be posed in the form of an optimal or approximate solution. Problems have also been studied under the notion of *distance* from a transitive structure.

In the later part of the thesis, we study the problem of *Popular Matchings*. Here our goal is to count the number of popular matchings in a given instance. We give some hardness and approximation results for the same.

Transitivity

Given a directed graph $G = (V, E)$, a subgraph S of G is called transitive if for every $a, b, c \in V$, if S contains the edges $a \rightarrow b$ and $b \rightarrow c$, then it also contains the edge $a \rightarrow c$. Our goal is to maximize the number of edges present in a transitive subgraph. For this we consider two questions – maximal and maximum transitive subgraphs.

Maximal Transitive Subgraph

In [CGJR15], we consider the problem of *maximal transitive subgraph* – output a transitive subgraph of maximal size (in terms of number of edges) contained in a given directed graph. Let's consider two related problems first. Let G be a directed graph with n vertices and m edges. Given a transitive subgraph S of G , can we add any more edges to S and still maintain transitivity? We can check the maximality of S in time $O(n^{w+1})$ using a standard algorithm (where $O(n^w)$ is the complexity of multiplying two $n \times n$ matrices.) A related problem is – given a transitive subgraph S of G , compute a maximal transitive subgraph of G that contains S . The naive algorithm takes $O(n^{w+2})$ time.

In [CGJR15], we give an algorithm that computes a maximal transitive subgraph in $O(n^2 + nm)$ time. The interesting part of our algorithm is that we avoid checking for maximality explicitly but output is still maximal. This is the first such algorithm that improves upon the standard techniques which have a complexity of $O(n^{w+1})$.

Maximum Transitive Subgraph

We then study the *maximum transitive subgraph* (MTS) problem – compute a transitive subgraph of largest size contained in a given directed graph. This problem was proven to be NP-complete by Yannakakis in [Yan78]. We start by studying approximation algorithms for this problem.

The MTS problem is a generalization of well-studied hard problems. For the class of triangle-free graphs, the problem of finding a maximum transitive subgraph in a directed graph is the same problem as the MAX-DICUT problem. MAX-DICUT has well known hardness and inapproximability results.

Approximation: In [CGJR15], we give a simple 0.25-approximation algorithm of obtaining an MTS in a general graph. For the case where the underlying undirected graph is triangle free, we give a 0.874-approximation for the MTS problem. The idea there is to look at the related problem of directed maximum cuts in the same graph. To the best of our knowledge, no approximation algorithms are present in the literature which present any ratio better than 0.25.

Upper Bound: Another interesting questions is how large the MTS can theoretically be. We study this problem in [CJ16b]. In a triangle-free (underlying undirected) graph, we know that there is a one-to-one correspondence between directed-cuts and transitive subgraphs. We prove that in triangle free graphs with m edges, any directed cut is of size at most $m/4 + cm^{4/5}$ for some $c > 0$. This gives the same bound for the size of an MTS. This also shows that the approach of finding MTS approximations via bipartite subgraphs can't have better constant approximation ratio than $1/4$.

Parameterization: Further we investigate the *parameterized complexity* of the MTS problem in [CJ16a]. A parameterization of a problem assigns an integer k to each input instance I and we say that a the problem is *fixed-parameter tractable* if there is an algorithm that solves the problem in time $f(k) \cdot |I|^{O(1)}$. Here, f is any computable function.

Arnborg et al. [ALS88] showed that the problem of MTS is fixed parameter tractable. They give an alternate proof of Courcelle's theorem [Cou90] and express the MTS problem in *Extended Monadic Second Order*, thus giving a meta-algorithm for the problem. This algorithm is not explicit and f is known to be only a computable function.

The MTS problem has been studied in a more general setting as the TRANSITIVITY EDITING problem where the goal is to compute the minimum number of edge insertions or deletions in order to make the input digraph transitive. Weller et al [WKNU12] prove its NP-hardness and give a fixed-parameter algorithm that runs in time $O(2.57^k + n^3)$ for an n -vertex digraph if k edge modifications are sufficient to make the digraph transitive. This result also applies to the case where only edge deletions are allowed – the MTS problem. Our result differs from [WKNU12] because we parameterize by treewidth.

Raman et al [RS07] show that the problem of deciding whether a directed graph has a transitive induced subgraph of size k is fixed-parameter tractable.

We first give a poly-time algorithm for computing an MTS in a directed tree. We generalize this algorithm. For a given directed graph with treewidth at most k , we give an algorithm which runs in time $O(n^{k^2})$. The idea here is to recursively use separators and combine the solutions of the two parts. We improve this algorithm by performing dynamic programming on a tree-decomposition of the input and present an algorithm that runs in time $O(4^{k^2} n^2)$.

Popular Matchings

A *popular matching problem* instance I comprises a set \mathcal{A} of *agents* and a set \mathcal{H} of *houses*. Each agent a in \mathcal{A} ranks (numbers) a subset of houses in \mathcal{H} (lower rank specify higher preference). The ordered list of houses ranked by $a \in \mathcal{A}$ is called a 's *preference list*. For an agent a , let E_a be the set of pairs (a, h) such that the house h appears on a 's preference list. Define $E = \cup_{a \in \mathcal{A}} E_a$. The problem instance I is then represented by a bipartite graph $G = (\mathcal{A} \cup \mathcal{H}, E)$ and a preference list for each $a \in \mathcal{A}$. A matching M of I is a matching of the bipartite graph G . We use $M(a)$ to denote the house assigned to agent a in M and $M(h)$ to denote the agent that is assigned house h in M . An agent *prefers* a matching M to a matching M' if (i) a is matched in M and unmatched in M' , or (ii) a is matched in both M and M' but a prefers the house $M(a)$ to $M'(a)$. Let $\phi(M, M')$ denote the number of agents that prefer M to M' . We say M is *more popular than* M' if $\phi(M, M') > \phi(M', M)$, and denote it by $M \succ M'$. A matching M is called *popular* if there exists no matching M' such that $M' \succ M$.

The popular matching problem was introduced by Gärdenfors in [Gär75] as a variation of the stable marriage problem¹ [GS62]. The idea of popular matching has been studied extensively in various settings in recent times [AIKM07, SM10, MI11, Mah06, KMN11, McCo8, Nas14], mostly in the context where only one side has preference of the other side but the other side has no preference at all. We will also focus on this setting. Much of the earlier work focuses on finding efficient algorithms to output a popular matching, if one exists. Problems have also been studied where both sides have preferences [BIM10, Kav14].

The problem of counting the number of "solutions" to a combinatorial question falls into the complexity class #P. An area of interest that has recently gathered a certain amount of attention is the problem of counting stable matchings in graphs. The Gale-Shapely algorithm [GS62] gives a simple and efficient algorithm to output a stable matching, but counting

¹Given n men and n women, where each person has ranked all members of the opposite sex in order of preference, marry the men and women together such that there are no two people of opposite sex who would both rather have each other than their current partners. When there are no such pairs of people, the set of marriages is deemed stable.

them was proved to be #P-hard by Irving and Leather in [IL86]. Bhatnagar et al. [BGR08] showed that the random walks on the *stable marriage lattice* are slowly mixing, even in very restricted versions of the problem. Chebolu et al. [CGM10] give further evidence towards the conjecture that there may not exist an FPRAS at all for this problem.

We look at generalizations of the standard version - preferences with ties and houses with capacities. In the case where preferences could have ties, it is already known that the counting version is #P-hard as shown by Nasre in [Nas14]. We give an FPRAS for this problem. In the case where houses have capacities², we prove that the counting version is #P-hard. While the FPRAS for the case of ties is achieved via a reduction to a well known algorithm, the #P-hardness for the capacitated case is involved, making it the more interesting setting of the problem.

²A house could have multiple rooms which can be occupied by different people, some of whom may have a different preference for the same house.

Part I

Transitivity in Relations

Chapter 2

Introduction to Transitivity

Transitivity is a fundamental property of relations. Given the importance of relations and the transitivity property, it is not surprising that various related problems have been studied in detail and have found widespread application in different fields of study.

Some of the fundamental problems related to transitivity that have been long studied are - given a relation ρ , checking whether ρ is transitive, finding the transitive closure of ρ , finding the maximum transitive relation contained in ρ , partitioning ρ into smallest number of transitive relations. Various algorithms have been proposed for these problems and some hardness results have also been proved.

In this paper, we study two related problems on transitivity. First - given a relation, obtain a *maximal* transitive relation contained in it. It is straight-forward to see that this can be solved in poly-time, hence our goal is to do this as efficiently as possible. Second - given a relation, obtain a *maximum* transitive relation contained in it. This problem was proven to be NP-complete in [Yan78]. Here our approach is to find approximate solutions.

The problem of finding a *maximum* transitive relation contained in a given relation is a generalisation of well-studied hard problems. For the class of triangle-free graphs, the problem of finding a maximum transitive subgraph in a directed graph is the same problem as the MAX-DICUT¹ problem. MAX-DICUT has well known inapproximability results.

We can also relate it to a problem² of optimization on a 3SAT instance. We look at the relation as a directed graph $G = (V, E)$, where $|V| = n$. For every pair for distinct vertices

¹Given a directed graph G and an arc weight function $w : E(G) \rightarrow \mathbb{R}_+$, the maximum directed cut problem (MAX-DICUT) is that of finding a directed cut with maximum total weight.

²This is different from MAX-3SAT problem

(i, j) in V , create a boolean variable x_{ij} . Consider the following 3SAT formula.

$$C = \bigwedge_{1 \leq i < j < k \leq n} (x_{ij} \vee \bar{x}_{ik} \vee \bar{x}_{kj})$$

Let C' be a formula derived from C such that any literal with variable x_{ij} is removed if $(i, j) \notin E$. It is easy to see that a solution to C' represents a subgraph of G . Specifically, a solution to C' is also transitive. To see this, observe that for every triplet (i, j, k) , if a clause $(x_{ij} \vee \bar{x}_{ik} \vee \bar{x}_{kj})$ is satisfied, then either the edge (i, j) is included or at least one of the edges (i, k) or (k, j) is excluded. To get the maximum transitive subgraph, the solution must maximize the number of variables set to 1. To conclude, the maximum transitive subgraph problem is same as the problem of finding a satisfying solution to a 3SAT formula that also maximizes the number of variables assigned the value 'true'.

Chapter 3

Maximal Transitive Subgraph

3.1 Maximal transitive relation finding algorithms

The usual greedy algorithm for finding a maximal substructure - satisfying a given property \mathcal{P} - starts with the empty set and incrementally grows the substructure while maintaining the property \mathcal{P} . Finally it ends when the set becomes maximal. Thus checking for maximality is a subroutine for the usual greedy algorithm.

In the case of finding a maximal transitive relation contained in a given relation the usual greedy algorithm takes $O(n^5)$ time, where n is the size of the set on which the binary relation is defined. Using matrix multiplication as a subroutine for checking maximality one can improve the running time of the greedy algorithm to $O(n^{\omega+2})$, where the matrix-multiplication of two $n \times n$ matrices takes $O(n^\omega)$ time.

The other greedy approach for finding a maximal substructure could be to start with an object \mathcal{O} and slowly shrink the object until it satisfies the property \mathcal{P} . Unfortunately, this technique may not yield a maximal substructure - the maximality may not be satisfied at the end.

In this paper we design an algorithm, for finding a maximal transitive sub-relation in a given relation ρ . Our algorithm runs in time $O(n^3)$ where n is the size of set on which the relation ρ is defined. Our algorithm does not use any subroutine for checking for maximality. In fact the best known algorithm for checking maximality in this case has running time $O(n^{\omega+1})$ which is clearly more than the running time of our algorithm.

Instead our algorithm follows the approach of the second kind of greedy algorithms discussed above. Given the fact that usually this approach does not guarantee that the output is maximal, we have to make some clever modification. The algorithm as such is simple but the key is the proof of correctness, which is quite involved.

In fact we present an algorithm that runs in time $O(nm + n^2)$ where, n is the size of the set on which the relation is defined and m is the size of the relation ρ . To the best of our knowledge, no better algorithm for finding a maximal transitive sub-relation is known.

3.1.1 $O(n^3)$ algorithm for finding maximal transitive sub-relation

Algorithm 1: Finding a maximal transitive sub-relation

Input : An $n \times n$ matrix $A = (a_{ij})$ representing a relation.

Output: A matrix $T = (t_{ij})$ which is a maximal transitive sub-relation contained in A .

```

1 for i ← 1 to n do
2   for j ← 1 to n, j ≠ i do
3     if  $a_{ij} = 1$  then
4       for k = 1 to n do
5         if  $k \neq j$  and  $a_{ik} = 0$  then
6           set  $a_{jk} = 0$ 
7         end
8         if  $k \neq i$  and  $a_{kj} = 0$  then
9           set  $a_{ki} = 0$ 
10        end
11       end
12     end
13   end
14 end
15  $T \leftarrow A$ 
16 return  $T$ 

```

Theorem 3.1. *Algorithm 1 correctly finds a maximal transitive sub-relation in a given relation in time $O(n^3)$.*

Proof. It is easy to see that the time complexity of the algorithm is $O(n^3)$. For the proof of correctness, all we need to prove is that the output T of the algorithm is transitive and maximal. The transitivity of the output T is proved in Lemma 3.5 and the maximality of T is proved in Lemma 3.6. \square

3.1.2 Proof of Correctness of Algorithm 1

Before we prove the correctness of Algorithm 1, let us make some simple observations about the algorithm. In this section we will treat the binary relation on a set S as a directed graph with vertex set S . So the Algorithm 1 takes a directed graph A on n vertices (labelled 1 to n) and outputs a directed transitive subgraph T that is maximal, that is, one cannot add arcs

from G to T to obtain a bigger transitive graph. In the algorithm, note that changing an entry a_{ij} from 1 to 0 implies deletion of the arc (i, j) .

Definition 3.2. At any stage of the Algorithm 1 we say the arc (a, b) is visited if at some earlier stage of the algorithm when $i = a$ in Line 1 and $j = b$ in Line 2 we had $a_{ij} = 1$.

Remark 3.3. We first note the following obvious but important facts of the Algorithm 1

- (1) No new arc is created during the algorithm because it never changes an entry a_{ij} in the matrix A from 0 to 1. It only deletes arcs.
- (2) Line 1, 2 and 3 of the algorithm implies that the algorithm visits the arcs one by one (in a particular order). And while visiting an arc it decides whether or not to delete some arcs.
- (3) Since in Line 1 the i increases from 1 to n so the algorithm first visits the arcs starting from vertex 1 and then the arcs starting from vertex 2 and then the arcs starting from vertex 3 and so on.
- (4) Arcs are deleted only in Line 6 and Line 9 in the algorithm.
- (5) While the for loop in Line 1 is in the i -th iteration (that is when the algorithm is visiting an arc starting at i) no arc starting from the i is deleted. In Line 6 only arcs starting from j are deleted and $j \neq i$ from Line 2. And in Line 9 only arcs ending in i are deleted.
- (6) In Line 2 the condition $j \neq i$ is given just for ease of understanding the algorithm. As such even if the condition was not there the algorithm would have the same output because if $j = i$ in Line 2 and the algorithm pass line 3 (that is $a_{ii} = 1$) then Line 6 would read as "if $a_{ik} = 0$ write $a_{ik} = 0$ " and Line 9 would read as "if $a_{ki} = 0$ write $a_{ki} = 0$ ", both of which are no action statement.
- (7) Similarly, in Line 5 the condition $k = j$ is given just for ease of understanding of the algorithm. If the condition was not there even then the algorithm would have produced the same result because from Line 3 we already have $a_{ij} = 1$ and thus if $k = j$ then $a_{ik} = a_{ij} \neq 0$.
- (8) Similarly, the condition $k \neq i$ in Line 9 has no particular role in the algorithm.

One important lemma for the proof of correctness is the following:

Lemma 3.4. *An arc once visited in Algorithm 1 cannot be deleted later on.*

Proof. Let us prove by contradiction.

Suppose at a certain point in the algorithm's run the arc (i, j) has already been visited, and then when the algorithm is visiting some other arc starting from vertex r the algorithm decides to delete the arc (i, j) .

If such an arc (i, j) which is deleted after being visited exists then there must a first one also. Without loss of generality we can assume that the arc (i, j) is the first such arc: that is when the algorithm decides to delete the arc (i, j) no other arc that has been visited by the algorithm has been deleted.

By point number 3 in Remark 3.3, $r \geq i$. From point number 5 in Remark 3.3 we can say that $r \neq i$. So we have $r > i$.

We now consider two cases depending on whether the algorithm decides to delete the arc (i, j) is Line 6 or Line 9.



FIGURE 3.1: Diagrams of the two cases for Lemma 3.4

Case I. Suppose (i, j) is deleted in Line 6, when the algorithm was visiting an arc starting from vertex r . Since the algorithm is deleting (i, j) in Line 6 so from Line 3 and Line 5 we have, at that stage, $a_{ri} = 1$ and $a_{rj} = 0$ (just like in Figure 3.1(left)).

Since no arc is ever created by the algorithm (point 1 in Remark 3.3), a_{ri} was 1 when the arc (i, j) was visited. So at the stage when the algorithm was visiting arc (i, j) , a_{rj} must be 1, otherwise (r, i) would be deleted by Line 9. Thus (r, j) was deleted after visiting the arc (i, j) and but before time (i, j) is being deleted.

By Remark 3.3(5), (r, j) cannot be deleted when visiting an arc starting from r . So (r, j) must have been deleted when visiting an arc starting from vertex r_1 and $r_1 < r$.

We now split this case into two cases depending on whether $r_1 = j$ or $r_1 \neq j$.

Case Ia: ($r_1 \neq j$)

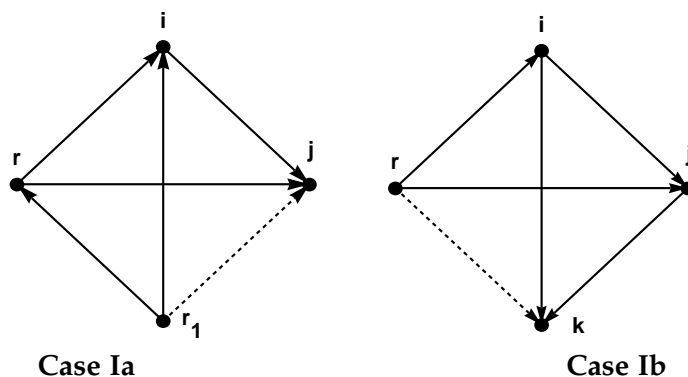


FIGURE 3.2: Diagram for subcases of Case 1 for Lemma 3.4

By Remark 3.3(5) we know at the set of arcs starting from vertex r_1 must have remained unchanged during the r_1 -th iteration of Line 1.

But since in the r_1 -th iteration of Line 1 the arc (r,j) was deleted so (r_1,r) must have been present while (r_1,j) was absent. Also if $a_{r_1,i} = 0$ when visiting the arc (r_1,r) , the algorithm would have found $a_{r_1,r} = 1$ and $a_{r_1,i} = 0$ and in that case would have deleted (r,i) in Line 6. That would contradict that fact the the arc (r,i) was present when the arc (i,j) was being deleted. Thus at the start of the r_1 -th iteration of Line 1 the situation would have been like in Figure 3.2(left).

But in that case, when visiting (r_1,i) the algorithm would have found $a_{r_1,i} = 1$ and $a_{r_1,j} = 0$ and then would have deleted the arc (i,j) . But by assumption the arc (i,j) is deleted when visiting arc (r,i) and not an arc starting at r_1 . So we get a contradiction. And thus if $s \neq j$ we have a contradiction.

Case Ib: $(r_1 = j)$

Let the arc (r,j) be deleted when the algorithm was visiting the arc (r_1,k) (that is (j,k)) for some k . Since the arc (j,k) is deleted after the arc (i,j) is visited and before the arc (r,i) is visited, so $i < j < r$.

Now consider the stage when the arc (j,k) is visited by the algorithm. If arc (i,j) is not present at that time then the arc (i,j) would have been deleted which would contradict the assumption that the arc (i,j) is deleted when the algorithm was visiting (r,i) . So just before the stage when the algorithm was visiting arc (j,k) the situation would have been like in Figure 3.2(right).

So the arc (i,k) was present when the algorithm was visiting the arc (j,k) . But since $i < j$ so the arc (i,k) must have been visited already. By the minimality condition that (i,j) is the first arc that is visited and then deleted and since the arc (i,j) is deleted when visiting arc

(r, i) , so when the algorithm just started visiting the arc (r, i) the arc (i, k) must be present. Also at that stage the arc (r, k) was absent as it was absent when visiting the arc (j, k) and $j < r$. So when the algorithm just started to visit (r, i) the situation would have been like in Figure 3.2(right)) except the arc (r, j) would also have been missing.

When the algorithm was visiting the arc (j, k) the arc (r, k) was not there. But when the algorithm visited the arc (i, j) the arc (r, k) must have been there, else the arc (r, i) would have been deleted at that stage, which would contradict our assumption that (i, j) was deleted when visiting (r, i) . So the arc (r, k) must have been deleted after the arc (i, k) was visited but before the arc (j, k) was visited.

If the arc is deleted when visiting some arc starting with k then it means that $i < k < j$. Now consider the stage when the algorithm was visiting (r, i) . As described earlier the situation would have been like in Figure 3.2(right)) except the arc (r, j) would also have been missing. Since $k < j$ so the algorithm would have deleted (i, k) before it deleted (i, j) . And since the algorithm has also visited (i, k) earlier so this contradicts the the minimality condition of (i, j) being the first visited arc to be deleted.

The other case being the arc deleted when visiting the some arc ending in r , say (t, r) , where $i < t < j$. Thus during the t -th iteration of Line 1 the arcs (t, r) is present while the arc (t, k) is absent. Now, since in the t -th iteration the arc (r, j) is not deleted thus it means that the arc (t, j) was present during the t -th iteration of Line 1. But in that case since arcs (t, j) and (j, k) are present while (t, k) is not present the algorithm would have deleted the arc (j, k) in the t -th iteration of Line 1, this contradicts the assumption that the arc (r, j) is deleted in the j -th iteration of Line 1 when visiting the arc (j, k) .

Thus the arc (i, j) cannot be deleted by the algorithm in Line 6 when visiting an arc starting from r .

Case II. Suppose (i, j) is deleted in Line 9, when the algorithm was visiting an arc starting from vertex r . In this case $j = r$. And since $r > i$ so $j > i$. Say the arc (i, j) is deleted when visiting arc (j, k) , for some vertex k . Since the algorithm is deleting (i, j) in Line 9 so from Line 3 and Line 8 we have, at that stage, $a_{jk} = 1$ and $a_{ik} = 0$ (cf. Figure 3.1(left)).

Now if a_{ik} was 0 when the algorithm visited the arc (i, j) then the algorithm would have found $a_{ik} = 0$ and $a_{i,j} = 1$ and in that case would have deleted the arc (j, k) in Line 6. That would give a contradiction as in a later stage of the algorithm (in particular in the j -th iteration of Line 1, with $j > i$) the arc (j, k) is present. So when the arc (i, j) was visited the arc (i, k) was present.

Since by Remark 3.3(5) the arc (i, k) cannot be deleted in the i th iteration of Line 1, so the arc (i, j) must have been visited in the i -th iteration of Line 1 and must have been deleted by the algorithm at a later time but before the arc (i, j) is deleted. But this would contradict the minimality of the arc (i, j) .

Hence even in this case also we get a contradiction. So this completes the proof. \square

The second lemma we need is the following.

Lemma 3.5. *The matrix T output by the Algorithm 1 is transitive.*

Proof. Suppose $t_{ij} = 1 = t_{jk}$. By Remark 3.3(1) no arc is created. So at all stages and in particular, at the initial stage $a_{ij} = a_{jk} = 1$. Suppose $a_{ik} = 0$ at the initial stage. Then when the algorithm visited (i, j) or (j, i) (whichever comes first), the arc (j, k) or (i, j) (respectively) will be deleted for the lack of the arc (i, k) , as $a_{ij} = a_{jk} = 1$ throughout (cf. Figure 3.3).

Thus suppose the arc (i, k) is deleted at some stage, say, r -th iteration of Line 1. Now $r > i, j$ for otherwise the arc (i, k) would be deleted before the i -th or j -th iteration of Line 1. And in that case in the i -th or j -th iteration of Line 1 (depending on which of i and j is smaller) either (j, k) or (i, j) would be deleted. And then at the end at least one of t_{ij} and t_{ik} must be 0.

But then the arc (i, k) is deleted during the i -th iteration of Line 1 (as $i < r$). Since no arc is deleted once it is visited by Lemma 3.4, we have $t_{ik} = 1$. Therefore T is transitive. \square

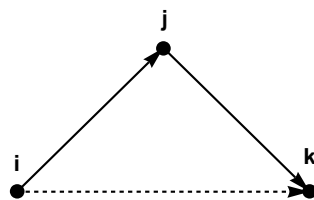


FIGURE 3.3: Diagram for Lemma 3.5

Using Lemma 3.5 and Lemma 3.4 we can finally prove the correctness of the algorithm.

Lemma 3.6. *The matrix T output by the Algorithm 1 is a maximal transitive relation contained in A .*

Proof. Now if T is not a maximal transitive sub-relation then there must be some arc (say (a, b)) such that the transitive closure of $T \cup \{(a, b)\}$ is also contained in A .

Now by Lemma 3.4, an arc once visited can never be deleted. Also the algorithm is visiting every undeleted arc. Thus T is the collection of visited arcs and these arcs are present at every stage of the algorithm.

Thus, every arc in the transitive closure of $T \cup \{(a, b)\}$ that is not in T must have been deleted in some iteration of Line 1. Let (i, j) be the first arc to be deleted among all the arcs that are in the of transitive closure of $T \cup \{(a, b)\}$ but not in T .

Clearly the transitive closure of $T \cup \{(i, j)\}$ is also contained in A , and all the arcs in the transitive closure of $T \cup \{(i, j)\}$ either is never deleted or is deleted after the arc (i, j) is deleted. Suppose the arc (i, j) is deleted in the r -th iteration of Line 1. We have $r \neq i$ by Remark 3.3(5) and by Lemma 3.4 we have $r < i$.

We now consider two cases depending on whether r is j or not.

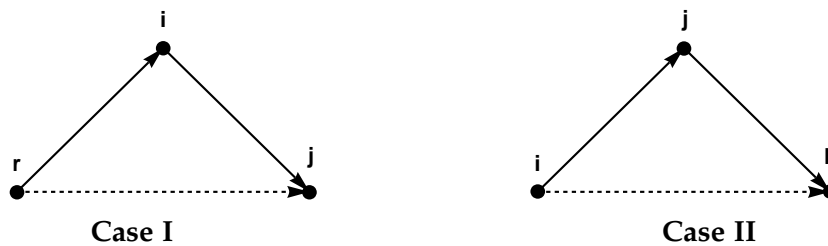


FIGURE 3.4: Diagrams of the two cases for Lemma 3.6

Case I: $r \neq j$

In this case, since the arc (i, j) was deleted in the r -iteration of Line 1, the arc (i, j) must have been deleted when the algorithm was visiting the arc (r, i) . So at the stage when the arc (i, j) was deleted, the arc (r, j) must not have been there (else the algorithm wouldn't have deleted the arc (i, j)).

If $a_{rj} = 0$ in A , then $t_{rj} = 0$ (by Remark 3.3(1)). But by Lemma 3.4 $t_{ri} = 1$ as the arc (r, i) is being visited. So $T \cup \{(i, j)\}$ is not transitive (cf. Figure 3.4(left)), and the transitive closure of $T \cup \{(i, j)\}$ must contain the arc (r, j) . Thus $a_{rj} = 1$ in A , but the arc (r, j) is deleted in some stage of the algorithm but before the visit of the r -th iteration of Line 1, say, at r_1 -th iteration of Line 1, with $r_1 < r$.

Thus the arc (r, j) is in the transitive closure of $T \cup \{(i, j)\}$ and it got deleted before the deletion of arc (i, j) . This is a contradiction to the fact that the arc (i, j) was the first arc to be deleted. So when $r \neq j$ we have a contradiction.

Case II: $r = j$

In this case, since the arc (i, j) was deleted in the j -iteration of Line 1, the arc (i, j) must have been deleted when the algorithm was visiting some arc (j, k) , for some vertex k . So at the stage when the arc (i, j) was deleted, the arc (i, k) must not have been there (else the algorithm wouldn't have deleted the arc (i, j)).

If $a_{ik} = 0$ in A , then $t_{ik} = 0$ (by Remark 3.3(1)). But by Lemma 3.4 $t_{jk} = 1$ as the arc (j, k) is being visited. So $T \cup \{(i, j)\}$ is not transitive (cf. Figure 3.4(right)), and the transitive closure of $T \cup \{(i, j)\}$ must contain the arc (i, k) . Thus $a_{ik} = 1$ in A , but the arc (i, k) is deleted in some stage of the algorithm but before the visit of the j -th iteration of Line 1, say, at r_1 -th iteration of Line 1, with $r_1 < j$.

Thus the arc (i, k) is in the transitive closure of $T \cup \{(i, j)\}$ and it got deleted before the deletion of arc (i, j) . This is a contradiction to the fact that the arc (i, j) was the first arc to be deleted. So when $r = j$ we have a contradiction.

Since in both the case we face a contradiction so we have that the output T is a maximal transitive relation contained in A . \square

3.1.3 Better running time analysis of Algorithm 1

If we do a better analysis of the running time of the Algorithm 1 we can see that the algorithm has running time $O(n^2 + nm)$. To see it more formally consider a new pseudocode of the algorithm that we present as Algorithm 2. It is not hard to see that both the algorithms are basically same.

Theorem 3.7. *Algorithm 2 correctly finds a maximal transitive relation contained in a given binary relation in $O(n^2 + mn)$, where m is the number of 1's in A .*

Proof. The proof for correctness is same as in Theorem 3.1. We calculate only the time complexity of the algorithm and it is given by

$$\begin{aligned} & \sum_{i=1}^n (n + k_i n), \text{ (where } k_i \text{ is the number of 1's in the } i^{\text{th}} \text{ row)} \\ &= n^2 + n \sum_{i=1}^n k_i = n^2 + mn. \end{aligned}$$

\square

Algorithm 2: Finding a maximal transitive sub-relation

Input : An $n \times n$ matrix $A = (a_{ij})$ representing a binary relation.**Output**: A matrix $T = (t_{ij})$ which is a maximal transitive relation contained in the given binary relation A .

```

1  for i ← 1 to n do
2  | Initialize  $B_i = \emptyset$ 
3  | for each  $s \leftarrow 1$  to  $n, j \neq i$  do
4  | | if  $a_{ij} = 1$  then
5  | | | Include  $j$  in  $B_i$ 
6  | | end
7  | end
8  | for each  $j \in B_i$  do
9  | | for each  $k = 1$  to  $n$  do
10 | | | if  $k \neq j$  and  $a_{ik} = 0$  then
11 | | | | Make  $a_{jk} = 0$ 
12 | | | end
13 | | | if  $k \neq i$  and  $a_{kj} = 0$  then
14 | | | | Make  $a_{ki} = 0$ 
15 | | | end
16 | | end
17 | end
18 end

```

Chapter 4

Maximum Transitive Relation

In this chapter, we continue the study of transitive structures in a directed graph. Our goal is to compute the largest transitive subgraph (in terms of number of edges) contained within a digraph.

4.1 Approximation

In this section, we study the problem of obtaining a maximum transitive relation contained in a binary relation. We will be using the notation of directed graphs for binary relations. As before, let's assume that input directed graph has m edges. Denote by $UG(D)$ the underlying graph of digraph D .

First, we state a well known result from graph theory.

Lemma 4.1. *There exists a bipartite subgraph of size $m/2$ in any undirected graph with m edges.*

Obtaining such a bipartite graph deterministically in poly-time is a folklore result. We outline a proof here. Let $G = (V, E)$ be an undirected graph. Let's make an arbitrary partitioning $V = X \uplus Y$. For any vertex $v \in X$ such that $d_X(v) > d_Y(v)$, moving v from X to Y will increase the size of $E(X, Y)$. We keep doing this switching operation on vertices in both X and Y until it is not possible anymore. This process has to stop within m steps as $|E(X, Y)|$ can not keep on increasing infinitely. At this point, we estimate $|E(X, Y)|$.

$$|E(X, Y)| = \frac{1}{2} \left(\sum_{u \in X} d_Y(u) + \sum_{v \in Y} d_X(v) \right) \geq \frac{1}{2} \left(\sum_{u \in X} d(u)/2 + \sum_{v \in Y} d(v)/2 \right) = m/2$$

This gives the following.

Theorem 4.2. *There exists a poly-time algorithm to obtain an $m/4$ sized transitive subgraph in any directed graph D with m edges. This gives a $1/4$ -approximation algorithm for maximum transitive subgraph problem.*

Proof. From Lemma 4.1, we get a bipartite subgraph of $UG(D)$ of size at least $m/2$. Now consider the original orientations on this bipartite subgraph. We collect all the edges in the direction that has more number of edges. This set of arcs is of size at least $m/4$ and is transitive as there are no directed paths of length two in the set. \square

The obvious question is – given a digraph with m edges, is there a transitive subgraph of size tm such that $t > 1/4$? We claim that this is not possible. We prove the following theorem in [CJ16b].

Theorem 4.3. *For every m , there exists a digraph D with m edges such that $UG(D)$ is triangle-free and the size of any directed cut in D is at most $m/4 + cm^{4/5}$ for some $c > 0$.*

We observe in Lemma 4.5 that in a digraph D such that $UG(D)$ is triangle-free, there is a one-to-one correspondence between directed cuts and transitive subgraphs in a digraph. Hence, obtaining a transitive subgraph of size better than $m/4$ (in the constant multiple) would contradict this theorem - since this would break the upper bound on the size of any directed cut. We return to proving Theorem 4.3 in the next section.

In order to improve upon the approximation factor, we focus on the class of *triangle-free* directed graphs.

Let G be a graph and U, V be a partition of the vertex set of H . A *directed cut* (U, V) is the set of edges with a starting in U and ending point in V . The MAX-DICUT problem is the problem of obtaining a largest directed cut in a graph. This is NP-hard. [LLZ02] gives an approximation algorithm for the MAX-DICUT problem.

Theorem 4.4 (see [LLZ02]). *There exists a 0.874 -approximation algorithm for the MAX-DICUT problem.*

As a corollary of Lemma ??, we have the following.

Lemma 4.5. *In a digraph D such that $UG(D)$ is triangle-free, there is a bijection between directed cuts of D and the set of transitive subgraph of D .*

Proof. First we prove that every transitive subgraph T of D is also a directed cut. Since $UG(D)$ is triangle-free, T does not have any directed 2-path. Hence no vertex in D has both a head-edge and a tail-edge from the set T . We place all the vertices at the tails in one set,

and all the vertices on the heads in another set. This partitioning along with the edge set T defines a directed-cut of D .

To see how every directed-cut C is also a transitive, notice that there are no directed 2-paths in C . \square

This implies that finding the maximum transitive subgraph is same as the MAX-DICUT problem for digraphs D with $UG(D)$ being triangle free.

Theorem 4.6. *There exists a 0.874-approximation algorithm for finding the maximum transitive subgraph in a digraph D such that $UG(D)$ is triangle-free.*

4.2 Maximum directed cuts in triangle free graphs

The *max-cut problem* is an extensively well studied problem both in terms of finding good approximation algorithms and estimating its bounds combinatorially. Both its undirected and directed versions are NP-complete. Here we give an upper bound on the size of *directed max-cut* using the probabilistic method.

The following notation is borrowed from [ABG⁺07]. Let G be an undirected graph and U, V be a partition of the vertex set of G . A *cut* (U, V) is the set of edges with one endpoint in U and other endpoint in V . Call $e(U, V)$ the size of cut (U, V) . Define

$$f(G) = \max_{(U,V)} e(U, V) \quad \text{and,} \quad f(m) = \min_{G:|E(G)|=m} f(G)$$

Finding a max-cut was proved to be NP-complete in [GJS76]. Goemans and Williamson give a semidefinite programming based algorithm in [GW95] to achieve an approximation ratio of 0.878. Under the Unique Games Conjecture, this is the best possible [KKMO07]. But a 0.5-approximation algorithm is straight forward - randomly put each vertex in U or V , leading to an expected cut size of $m/2$. Hence, $f(m) \geq m/2$. Various bounds have been proposed for $f(m)$, most notably in [EFPS88, Alo96]. Following is an upper bound for $f(m)$ in triangle free graph.

Theorem 4.7 (Alon [Alo96]). *There exists a constant $c' > 0$ such that for every m there exists a triangle-free graph G with m edges satisfying $f(G) \leq m/2 + c'm^{4/5}$.*

Let H be a directed graph and U, V be a partition of vertex set of H . A *cut* of H is similarly defined as before. A *directed cut* (U, V) is the set of edges with starting point in U and ending

point in V . Call $e(U, V)$ the size of cut (U, V) . Define

$$g(H) = \max_{(U, V)} e(U, V) \quad \text{and,} \quad g(m) = \min_{H: |E(H)|=m} g(H)$$

Finding a directed cut of maximum size is NP-complete (via a simple reduction from the max-cut problem). [GW95] gave a 0.796 approximation for this problem. Again, a 0.25 approximation is simple, given the 0.5-approximation of max-cut. Since it is easy to find a cut of size $m/2$ in undirected graphs and a directed cut of size $m/4$ in directed graphs, an obvious question is how much better one can do as a fraction of m . Alon proved in [Alo96] that the factor $1/2$ can not be improved for max-cuts. We prove that the factor $1/4$ can't be improved for directed max-cuts.

4.3 Upper Bound on Directed Max-Cut

In this section we prove the following bound. For any m , there exists a directed graph with m edges such that for some $c > 0$,

$$g(H) \leq m/4 + cm^{4/5}$$

The proof idea is as follows. From Theorem 4.7, we know that for every m there exists an undirected graph with m edges, all whose cuts are bounded by $m/2 + o(m)$ in size. For any given m in our case, we start with the undirected graph of Theorem 4.7 satisfying the above bound. We orient this graph uniformly at random. We then prove that every cut of size more than $m/4$ will be highly balanced, in the sense that - the cut will have almost the same number of edges going from left to right and right to left. We formalise these ideas below.

We define a notion of balanced cuts of a directed graph and balanced directed graphs.

Definition 4.8 (δ -balanced cut). For a directed graph, consider a cut (U, V) . The cut is δ -balanced if

$$|e(U, V) - e(V, U)| \leq \delta \left(\frac{e(U, V) + e(V, U)}{2} \right)$$

Definition 4.9 ((k, δ) -balanced graph). A directed graph H is (k, δ) -balanced if every cut of H of size at least k is δ -balanced.

Lemma 4.10. For any $m, \delta > 0$ and $k \leq m$, there exists a directed graph H on n vertices and m edges such that H is (k, δ) -balanced if $n < k\delta^2/6$.

Proof. For the given m , we start with an undirected graph G satisfying the condition in Theorem 4.7. We orient the edges of G uniformly at random and independently and call it H .

Let $C = (U, V)$ be a cut in the undirected graph G of size at least k . We first calculate the probability (over the random orientations of G) that C is not δ -balanced in H .

$$\begin{aligned} P[C \text{ is not } \delta\text{-balanced}] &= P[|e(U, V) - e(V, U)| > \delta(e(U, V) + e(V, U))/2] & (4.1) \\ &= 2P[e(U, V) > (1 + \delta)|C|/2] & (4.2) \end{aligned}$$

For each edge e_i in the cut (U, V) , define a random variable X_i as follows,

$$X_i = \begin{cases} 1 & \text{if } e_i \text{ is directed from } U \text{ to } V \\ 0 & \text{otherwise} \end{cases}$$

X_i 's are i.i.d. random variables with probability $1/2$. Then, $e(U, V) = \sum_{e_i \in (U, V)} X_i$ with mean $|C|/2$. We apply the standard Chernoff bound to get an upper bound for the probability in Equation (4.2),

$$\begin{aligned} P[C \text{ is not } \delta\text{-balanced}] &\leq 2 \exp(-\delta^2|C|/6) \\ &\leq 2 \exp(-\delta^2k/6) \end{aligned}$$

We now calculate the probability that the graph H is (k, δ) -balanced.

$$\begin{aligned} &P[H \text{ is } (k, \delta)\text{-balanced}] \\ &= 1 - P[\text{there exists a cut } C \text{ in } H \text{ of size at least } k \text{ which is not } \delta \text{ balanced}] \\ &= 1 - P \left[\bigcup_{\text{cut } C, |C| \geq k} C \text{ is not } \delta\text{-balanced} \right] \\ &\geq 1 - 2^n (2 \exp(-\delta^2k/6)) \\ &> 0, \quad \text{if } n < k\delta^2/6 \end{aligned}$$

□

The following lemma gives us a directed graph H , such that any cut of size at least $m/4$ in H is 'well' balanced. This result is used in proving the final theorem.

Lemma 4.11. *For any m , there exists a directed graph H with m edges such that H is $(m/4, \alpha/m^{1/5})$ -balanced for some $\alpha > 0$.*

Proof. By choosing $k = m/4$ and $\delta = \alpha/m^{1/5}$ in Lemma 4.10, we get H if

$$\begin{aligned} n &\leq (m/4)(\alpha/m^{1/5})^2/6 \\ \text{or, } m &\geq (24/\alpha^2)^{5/3}n^{5/3} \end{aligned}$$

The counterexample in Theorem 4.7 requires that $m = (1/8 + o(1))n^{5/3}$. Hence we need $\alpha \geq \sqrt{24}/(1/8 + o(1))^{3/10}$. \square

We now prove our main claim.

Theorem 4.12. *For every m , there exists a digraph D with m edges such that $\text{UG}(D)$ is triangle-free and the size of any directed cut in D is at most $m/4 + cm^{4/5}$ for some $c > 0$.*

Proof. For the given m , Lemma 4.11 gives a graph H that is $(m/4, \alpha/m^{1/5})$ -balanced, which would imply that every cut of size at least $m/4$ is $\alpha/m^{1/5}$ -balanced. Consider any cut (U, V) in H . We have,

$$\begin{aligned} |e(U, V) - e(V, U)| &\leq \alpha/m^{1/5}|(U, V)| \\ |e(U, V)|, |e(V, U)| &\leq |(U, V)|/2 + (\alpha/m^{1/5})|(U, V)|/4 \\ &\leq (m/2 + c'm^{4/5})/2 + (\alpha/4m^{1/5})m \\ &\leq m/4 + (c'/2 + \alpha/4)m^{4/5} \end{aligned}$$

In the second last inequality we use the fact that $|(U, V)| \leq m/2 + c'm^{4/5}$ from Theorem 4.7. This completes the proof with the choice of $c = (c'/2 + \alpha/4)$. \square

Chapter 5

Exact Algorithms for Maximum Transitive Subgraph Problem

5.1 Introduction

Our main goal in this chapter is to understand the *parameterized complexity* of the MTS problem. A parameterization of a problem assigns an integer k to each input instance I and we say that a the problem is *fixed-parameter tractable* if there is an algorithm that solves the problem in time $f(k) \cdot |I|^{O(1)}$. Here, f is any computable function. First systematic study of parameterized complexity was done by Downey and Fellows [DF99]. More recent account of the field can be found in the texts [FG06, Nie06, CFK⁺15].

The MTS problem has been studied in a more general setting as the TRANSITIVITY EDITING problem where the goal is to compute the minimum number of edge insertions or deletions in order to make the input digraph transitive. Weller et al [WKNU12] prove its NP-hardness and give a fixed-parameter algorithm that runs in time $O(2.57^k + n^3)$ for an n -vertex digraph if k edge modifications are sufficient to make the digraph transitive. This result also applies to the case where only edge deletions are allowed – the MTS problem. Our result differs from [WKNU12] because we parameterize by treewidth.

Raman et al [RS07] show that the problem of deciding whether a directed graph has a transitive induced subgraph of size k is fixed-parameter tractable.

In Section 5.2, we give a poly-time algorithm for computing an MTS in a directed tree. In Section 5.3, we give our first generalisation. For a given directed graph with treewidth at most k , we give an algorithm which runs in time $O(n^{k^2})$. The idea here is to recursively use separators and combine the solutions of the two parts. We improve this algorithm in Section

5.4, where we show that this problem is fixed-parameter tractable when parameterized by *treewidth*. The main result is stated below.

Theorem 5.1. *There exists an algorithm that runs in time $O(4^{k^2} n^2)$ to output an MTS for a graph with treewidth k .*

Notation

For any set S and $x \in S$, define $S - x = S \setminus \{x\}$. Let $G = (V, E)$ be a given directed graph. For $v \in V, e \in E$, define $G - v$ to be the graph obtained by removing the vertex v from G and $G \setminus e$ represents the graph obtained by deleting the edge e from G . The notation $F \subseteq G$ defines a subgraph F of G . For any subgraph F of G , $V(F)$ defines the vertex set of F and $E(F)$ defines the edge set of F . For $U \subseteq V$, $G(U)$ defines the induced subgraph on U .

For $A, B \subseteq V$, define $E(A, B) = \{u \rightarrow v : u \in A, v \in B\}$ and $\mathcal{E}(A, B) = E(A, B) \cup E(B, A)$. In the context of transitivity, we say that the two-path $u \rightarrow v \rightarrow w$ is *complete* if $u \rightarrow w \in E$. If $u \rightarrow w \notin E$, the two-path is called *incomplete*.

5.2 MTS in Trees

In the following discussion, an edge-rooted tree is a rooted tree in which the root has only one child. We identify the root of an edge-rooted tree with a root edge e and denote the tree as T_e .

The problem is to compute an MTS of a given directed tree T (the underlying undirected graph is a tree). We can root the tree at a vertex which has only outgoing edges. Let this root be r and e_1, \dots, e_l denote the outgoing edges from r . Denote the edge-rooted trees at r by T_{e_1}, \dots, T_{e_l} .

Note that the T_{e_i} 's are completely independent of each other and hence their MTS can be computed independently.

Since there are only outgoing edges from the root r ,

$$\text{MTS}(T) = \bigcup_{i \in [l]} \text{MTS}(T_{e_i})$$

We now describe how to compute $\text{MTS}(T_{e_i})$. We can divide the set of transitive subgraphs of T_{e_i} into two subsets and compute their maximums:

1. $MTS^+(T_{e_i})$: maximum over transitive subgraphs that include the edge e_i ,
2. $MTS^-(T_{e_i})$: maximum over transitive subgraphs that exclude the edge e_i .

Then,

$$MTS(T_{e_i}) = \max\{MTS^+(T_{e_i}), MTS^-(T_{e_i})\}$$

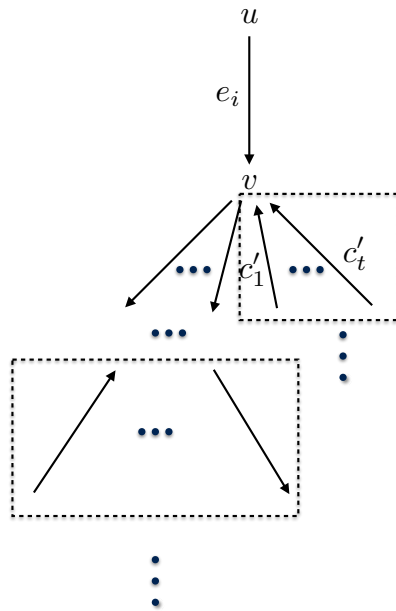
The maximum here is over the size of the transitive sets.

Computing $MTS^+(T_{e_i})$

This is further divided into two cases.

Case 1: e_i is a 'down' edge

FIGURE 5.1: Computation of $MTS^+(T_{e_i})$ 'down' edge case: we only include the MTS of rooted trees in the dashed rectangles



Let e_i be the edge $u \rightarrow v$ such that the tree T_{e_i} is rooted at the vertex u . Let the vertex v has outgoing edges c_1, \dots, c_s and incoming edges c'_1, \dots, c'_t . Then,

$$MTS^+(T_{e_i}) = \left[\bigcup_{i \in [s]} MTS^-(T_{c_i}) \right] \cup \left[\bigcup_{i \in [t]} MTS(T_{c'_i}) \right]$$

The reason we do not include the edges c_i in the first union in this calculation is because this will lead to a possible inclusion of 2-path (whose first edge is e_i) in the MTS set.

Case 2: e_i is an ‘up’ edge

Here $e_i = v \rightarrow u$ is in ‘upward’ direction of the rooted tree T_{e_i} . The algorithm is symmetrical to Case 1.

Computing $MTS^-(T_{e_i})$

Here, the direction of e_i is not important. Let the tree T_{e_i} be rooted at u and v be the other vertex of edge e_i . Let the vertex v has outgoing edges c_1, \dots, c_s and incoming edges c'_1, \dots, c'_t . Let,

$$M_1 = \left[\bigcup_{i \in [s]} MTS(T_{c_i}) \right] \cup \left[\bigcup_{i \in [t]} MTS^-(T_{c'_i}) \right]$$

$$M_2 = \left[\bigcup_{i \in [s]} MTS^-(T_{c_i}) \right] \cup \left[\bigcup_{i \in [t]} MTS(T_{c'_i}) \right]$$

Then, we have,

$$MTS^-(T_{e_i}) = \max\{M_1, M_2\}$$

The base cases are defined naturally.

Dynamic programming over edge-rooted trees

Though we defined the solution using a top-down approach, we observe that the subproblems are calculated every time there is a call of type $T(e_i)$. The same subproblem is called many times as part of computation of other subproblems. To avoid this, we do the actual computation in bottom up manner. For each edge-rooted tree T_e , we keep in memory the set $MTS(T_e)$. The order of computation of is as follows. We do a breadth-first search at root vertex r . We compute all the $MTS(T_e)$ for edges at the largest level first. These sets are just the edges themselves. In next stage, we decrease the level by 1. We go on doing this until we compute the $MTS(T_e)$ values for edges at the level 0.

Complexity: We work with edge-rooted trees in a reverse BFS order, where each edge is considered at most three times. This is because an edge can appear in computations at three different levels – its own level, as a child-edge or as a grandchild edge. This gives a liner-time algorithm for MTS in a tree.

5.3 MTS in Bounded Treewidth Graphs: First Attempt

We try to generalize the idea used in case of trees to compute the MTS for graphs. In particular, we want to compute the MTS for weighted-directed graphs whose underlying undirected graph has bounded treewidth. We allow weights here because it allows us to artificially force the selection of any transitive subset in a solution set. What we are effectively solving here is the problem of finding the MTS containing a given transitive set.

In the case of trees, at every step, we use a vertex that separates a graph into two (or more) disjoint subgraphs. We could then apply the algorithm recursively on these subgraphs and combine the results to compute the MTS for the current tree. For this, we will need the following lemma which ensures that small treewidth implies small balanced separators. We apply the separator idea on the underlying undirected graph of our input.

Lemma 5.2. *If G is a graph with treewidth at most d , then we can find a $1/2$ -balanced separator S of G in polynomial time, such that $|S| \leq d + 1$.*

The proof of Lemma 5.2 can be found in Lemma 7.19 in [CFK⁺15]. Let $G = (V, E)$ be a graph with treewidth at most $d - 1$. Then we can find a separator S that separates the graph into vertex sets L and R (left and right sets) such that, $V = S \uplus L \uplus R$, $|S| \leq d$, and $|L|, |R| \leq |V|/2$. We need the following subgraph notation to define a useful structure which we use in our algorithm. For $A, S \subseteq V$ such that $A \cap S = \emptyset$, T be a transitive subgraph of $E(S)$ and (I, O, U, Y) be vertex partitioning of S , define $G(A, S, T, (I, O, U, Y))$ to be a subgraph of G such that,

- all the edges in $E(A)$ are included,
- only the edges in T are included from $E(S)$,
- for the edges between the sets A and S , all the edges in $E(A, I), E(O, A), E(A, U), E(U, A)$ are included and no edges between A and Y are included.

Here, the sets intuitively mean the following: O – only outgoing edges, I – only incoming edges, U – completely unrestricted and Y – no edges allowed. We describe Algorithm 3 in detail now. The high level idea is described in Figure 5.2. After we compute the balanced-separator S , we work separately on $L \cup S$ and $R \cup S$ and combine the MTS calculated from these two subproblems to compute the MTS for the main problem. In order to be able to combine the two solutions, we force the transitive set from S to be same in both the solutions. We go over every possible subset T of S and recurse on both left and right side such that the solution from both the sides must contain exactly the set T from the edges on set S . To do this, we assign the weight ∞ to the edges of T in the recursive calls.

Algorithm 3: $\text{MTS}(G, w)$: Maximum Transitive Subgraph of weighted digraph G

Input : A directed graph $G = (V, E)$ with edge weights $w : E \rightarrow \mathbb{N}^+$ such that the underlying undirected graph of G has treewidth at most $d - 1$

Output: An MTS of G

```

1  $\max \leftarrow \sum_{e \in E} w(e)$ 
2 compute a 1/2-separator  $S$  of  $G$  of size at most  $d$ , with components  $L$  and  $R$ 
3  $M \leftarrow \phi$ 
4 foreach transitive  $T \subseteq E(S)$  do
5    $w' \leftarrow w$ 
6   foreach  $e \in T$  do
7      $w'(e) \leftarrow \max$ 
8   end
9   foreach partition  $(I, O, U, Y)$  of  $S$  do
10    foreach  $U' \subseteq U$  do
11       $M' \leftarrow \text{MTS}(G(L, S, T, (I, O, U', Y \cup (U \setminus U'))), w')$ 
12         $\cup \text{MTS}(G(R, S, T, (I, O, U \setminus U', Y \cup U')), w')$ 
13      if  $w(M') > w(M)$  then
14         $M \leftarrow M'$ 
15      end
16    end
17  end
18 end
19 return  $M$ 

```

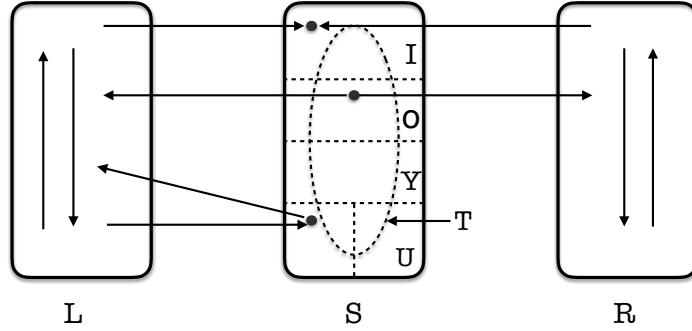
Notice that combining a solution from left and right (even with a common set T in the separator) may bring in discrepancies in transitivity. For example, a two-path $a \rightarrow b \rightarrow c$ with $a \in L, b \in S, c \in R$ may creep in. There can be no *completing edge* $a \rightarrow c$ since vertices a and c are separated by set S . To take care of this, we partition S into four parts (I, O, U, Y) . The subgraph that we pass as argument to the recursive calls has the useful features, such as: only incoming edges are present on vertices in set I . This allows us to combine the results from left and right as no two-paths can have a vertex of I at its center. Similarly we restrict only outgoing edges from the vertices in O . We restrict the vertices in Y to have no edges since a solution may not use all the vertices in S . Finally, we allow a part U' of U to have all the *original* edges on one side and have no edges on the other side. One can think of U' as being *unrestricted* on one side and restricted to have no participation on the other side.

Theorem 5.3. *Given (G, w) , algorithm $\text{MTS}(G, w)$ outputs an MTS of G .*

Proof. We prove it by induction on the number of vertices. For base case, we consider graphs on 3 vertices. It is straightforward to see that the statement is true in this case.

Let H be an MTS of G . Consider any separator S that separates the graph G into L and R (vertex sets). Let $T' = H \cap G(S)$. Notice that $(H \cap G(L)) \cup T'$ is an MTS for $G(L \cup S)$.

FIGURE 5.2: A high level description of Algorithm 3



Define partition $(I', O', U' = (U'_1, U'_2), Y')$ of S based on the edges in $\mathcal{E}(S, V(H) \cap L)$ and $\mathcal{E}(S, V(H) \cap R)$ and our definition of such a partitioning earlier. Here U'_1 is unrestricted on the left side and U'_2 is unrestricted on the right side.

Now since Algorithm 3 loops over all partitions $(I, O, U = (U_1, U_2), Y)$ of S , it will also hit the particular partition $(I', O', U' = (U'_1, U'_2), Y')$ at one point. At this point a call to $\text{MTS}(G(L, S, T, (I', O', U'_1, Y'), w))$ is made. Using induction, $\text{MTS}(G(L, S, T, (I', O', U'_1, Y'), w))$ returns the MTS of the graph $G(L, S, T, (I', O', U'_1, Y'))$. This would mean that

$$w(\text{MTS}(G(L, S, T, (I', O', U'_1, Y'), w))) = w(H \cap G(L)) \cup T'$$

A similar argument can be given for,

$$w(\text{MTS}(G(R, S, T, (I', O', U'_2, Y'), w))) = w((H \cap G(R)) \cup T')$$

Since we combine the MTS for $G(L \cup S)$ and $G(R \cup S)$ via the same transitive set T' in the both the cases, we conclude that,

$$w(\text{MTS}(G, w)) = w(H)$$

□

We now consider the complexity of Algorithm 3. From Lemma 5.2, we can assume that each partition is of size at most half of original number of vertices. For each $T \subseteq E(S)$ (at most 2^{k^2}), for each 4-partition of S (at most 4^k), for each $U' \subseteq U$ (at most 2^k), we make two calls of each size at most $T(n/2)$ and combine the solutions in time $O(n^2)$. This give the recurrence: $T(n) \leq 2^{k^2} 4^k 2^k (T(n/2) + O(n^2))$. Solving this, we get a running time of $O(n^{k^2})$.

5.4 MTS is FPT Parameterised by Treewidth

For a given digraph with treewidth at most k , we give an algorithm, for finding MTS, which runs in time $O(4^{k^2}n^2)$. To keep the presentation simple, we work on unweighted digraphs, though the same ideas can be used to obtain an MTS for weighted digraphs.

We first introduce the basics of tree decomposition of an undirected graph $G = (V, E)$. We borrow the notations from [CFK⁺15]. Let $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ be a tree decomposition, where T is a tree whose every node t is assigned a vertex subset $X_t \subseteq V(G)$, called a bag, such that the following three conditions hold:

1. $\cup_{t \in V(T)} X_t = V(G)$.
2. \forall edge $(u, v) \in E(G), \exists t \in T$ with $u, v \in X_t$.
3. $\forall u \in V(G)$, the set $T_u = \{t \in V(T) : u \in X_t\}$ induces a connected subtree of T .

Further, we use what is called a *nice* tree decomposition $\mathcal{T} = (T, \{X_t\}_{t \in V(T)})$ of G . Such a decomposition has the following properties.

1. The leaf and the root nodes are empty.
2. For every non-leaf node t is one of the following types:
 - (a) Introduce: t has exactly one child t' with $X_t = X_{t'} \cup \{u\}$ for some $u \notin X_{t'}$
 - (b) Forget: t has exactly one child t' with $X_t = X_{t'} \setminus \{u\}$ for some $u \in X_{t'}$
 - (c) Join: t has two children t' and t'' with $X_t = X_{t'} = X_{t''}$.

We perform a bottom up dynamic programming on \mathcal{T} starting from the leaves and ending at the root. We describe the calculations performed at a node of any type (categorised above) using the computation performed at the children nodes. For any node $t \in \mathcal{T}$, denote by V_t the union of the bags associated with all the nodes in the subtree rooted at t , including X_t . Let G_t define the induced graph on V_t .

We define a table entry $m[t, F, I, O, U, Y]$ for each node $t \in \mathcal{T}$, for each transitive subgraph $F \subseteq E(X_t)$ and for each partition (I, O, U, Y) of X_t . The entry $M = m[t, F, I, O, U, Y]$ contains the MTS on $G(V_t)$ with the restriction that $G(X_t) \cap M = F$ and in M ,

- no edges from the set $E(I, V_t \setminus X_t)$ are allowed,
- no edges from the set $E(V_t \setminus X_t, O)$ are allowed,

- any edge from the set $\mathcal{E}(V_t \setminus X_t, U)$ is allowed, and
- no edges from the set $\mathcal{E}(V_t \setminus X_t, Y)$ are allowed.

Vertices in U are said to be *unrestricted*. This partitioning has been defined to support the *Join* operation at a join node in the tree decomposition. The idea is same as before - avoid two-paths across separated partitions and avoid *incomplete* two-paths in the separator.

We present a rough sketch of the algorithm first.

Algorithm 4: MTS(G): Maximum Transitive Subgraph of digraph G

Input : A directed graph $G = (V, E)$

Output: An MTS of G

```

1 T ← a nice tree decomposition of G
2 m[t, *, *, *, *, *] ←  $\phi$ 
3 // perform a bottom-up dynamic programming on t
4 foreach t in T in bottom-up queue do
5   if t is root then
6     | output m[t,  $\phi$ ,  $\phi$ ,  $\phi$ ,  $\phi$ ,  $\phi$ ]
7     | exit
8   end
9   if t is 'leaf' then
10    | m[t,  $\phi$ ,  $\phi$ ,  $\phi$ ,  $\phi$ ,  $\phi$ ] =  $\phi$ 
11  end
12  if t is 'introduce' then
13    | Update m[.] according to Introduce algorithm
14  end
15  if t is 'forget' then
16    | Update m[.] according to Forget algorithm
17  end
18  if t is 'join' then
19    | Update m[.] according to Join algorithm
20  end
21 end

```

Leaf node: The only valid cell entry here is $m[t, \phi, \phi, \phi, \phi, \phi] = \phi$.

Introduce node: Suppose node t has child node t' such that $X_t = X_{t'} \cup \{v\}$. For any given partition (I, O, U, Y) of X_t and $F \subseteq G(X_t)$, we need to compute $m[t, F, I, O, U, Y]$.

First notice that the introduced vertex v can have edges to only the vertices in the set $X_{t'}$. Also, by definition, $\text{MTS}(G_t) \cap G(X_t) = F$. Applying both these conditions together, if v is

not in $V(F)$, no edge incident on v can be included in the MTS of G_t . Hence, for $v \notin V(F)$,

$$m[t, F, I, O, U, Y] = m[t', F, I - v, O - v, U - v, Y - v]$$

Now we consider the case where $v \in V(F)$. For a vertex $r \in V(G)$ and $X \subseteq E(G)$, define the in-neighbours of r in X as $N_X^i(r) = \{s : s \rightarrow r \in X\}$ and the out-neighbours of r in X as $N_X^o(r) = \{s : r \rightarrow s \in X\}$. Define $N_X(r) = N_X^i(r) \cup N_X^o(r)$.

Consider the set $N_{X_t}(v) \setminus N_F(v)$. These neighbours of v in X_t , wherever they may lie in the partition (I, O, U, Y) , can be kept as is in their designated partitions for recursion. The argument is as follows. Consider $u \in N_{X_t}(v) \setminus N_F(v)$. We want to check if any edge through u breaks transitivity. If $u \notin V(F)$, then u does not interact with any other vertex in X_t by definition and hence transitivity is maintained as before. If $u \in V(F)$, any edge in F incident on vertex u is already a part of a transitive set since F is transitive by definition.

We now deal with the set $N_F(v)$. Consider a vertex $u \in N_F(v)$. Following useful cases arise.

1. $u \in I \cap N_F^i(v)$: Here, an edge passing through u may break the transitivity. Such a vertex u must be removed from I and placed in O .
2. $u \in O \cap N_F^o(v)$: This is similar to the last case. We should move u from O to I .
3. $u \in U \cap N_F(v)$: Since the edges $\mathcal{E}(u, V_t \setminus X_t)$ are unrestricted to participate in an MTS, transitivity may break in two ways. Incomplete two-path of the form $r \rightarrow u \rightarrow v$ or $v \rightarrow u \rightarrow r$ where $r \in V_t \setminus X_t$ may result. We should disallow these cases.
4. $u \in Y \cap N_F(v)$: This case is fine as $\mathcal{E}(u, V_t \setminus X_t) = \emptyset$.

We incorporate all these restrictions in the following computation.

$$\begin{aligned} F' &= F - v \\ I' &= (I \setminus N_F^i(v)) \cup (O \cap N_F^o(v)) \cup (U \cap N_F^o(v)) \\ O' &= (O \setminus N_F^o(v)) \cup (I \cap N_F^i(v)) \cup (U \cap N_F^i(v)) \\ U' &= U \setminus N_F(v) \\ Y' &= Y \end{aligned}$$

The update method is then $m[t, F, I, O, U, Y] = m[t', F', I', O', U', Y'] \cup F$.

Forget Node: Suppose node t has child t' such that $X_t = X_{t'} \setminus \{v\}$. We update the current entry as follows:

$$m[t, F, I, O, U, Y] = \max m[t', F', I', O', U', Y']$$

where the maximum is over the following conditions:

$$\begin{aligned} F'|_{X_t} &= F \\ I' &= I, O' = O, Y' = Y \\ U' &= U \cup \{v\} \end{aligned}$$

Here, the transitive set F' is allowed to include the vertex v resulting in the condition $F'|_{X_t} = F$. We also allow v to have unrestricted edges since this effectively covers all the cases - only incoming edges on v , or only outgoing edges from v , or the case where v has both incoming and outgoing edges.

Join Node: Suppose node t has children t_1 and t_2 such that $X_t = X_{t_1} = X_{t_2}$. Define arbitrary partitions (to be fixed below) $X_{t_1} = I' \uplus O' \uplus U' \uplus Y'$ and $X_{t_2} = I'' \uplus O'' \uplus U'' \uplus Y''$. We have the following rule for updating the current entry:

$$m[t, F, I, O, U, Y] = \max(m[t_1, F, I', O', U', Y'] \cup m[t_2, F, I'', O'', U'', Y''])$$

under the restriction that:

$$\begin{aligned} I' \supseteq I, I'' \supseteq I \\ O' \supseteq O, O'' \supseteq O \end{aligned}$$

For each vertex $v \in U$, one of the following is true:

- $v \in I' \cap I''$
- $v \in O' \cap O''$
- $v \in U' \cap Y''$
- $v \in Y' \cap U''$

Here, we keep the F same in both t_1 and t_2 as required. In order to join at any vertex v in I , we demand such a vertex must be present in both I' and I'' but we also allow these sets to be larger. This is required as this leaves the possibility of a larger combination while transitivity is still maintained. A similar restriction is employed on O' and O'' .

The vertices v in U are unrestricted but we need to be careful while using unrestricted vertices in the join operation. Such a vertex should only be allowed to be unrestricted on one side but completely isolated on the other side. This gives us the possibilities $v \in U' \cap Y''$

or $v \in Y' \cap U''$. But this restriction forbids the possibility of edges being used on both sides of v . Such a case could occur if v uses only incoming (or outgoing) edges on both the sides. To accommodate this, we have the options of $v \in I' \cap I''$ or $v \in O' \cap O''$. Finally, we take the maximum over all the legitimate join operations.

We now estimate the running time of our algorithm. Assuming the input graph has treewidth k , each node X_t is of size at most $k + 1$. The number of partitions of type (I, O, U, Y) of X_t is at most 2^{k+4} . The number of transitive subgraphs F of X_t is at most 2^{k^2} . A single update of $m[\cdot]$ at any node can be done in at most n^2 steps. So a simple upper bound to the time complexity is $4^{k^2} n^2$.

5.5 Conclusion

In this chapter, we have continued the systematic study of computing a Maximum Transitive Subgraph of a given directed graph addressed recently in [CGJR15]. We show that this problem is fixed-parameter tractable when parameterized by treewidth. In particular, we give an algorithm that runs in time $O(4^{k^2} n^2)$ to output an MTS for a graph with treewidth k . An immediate question that arises is – whether we can reduce the exponent k^2 to $O(k)$.

Another interesting question that we have not addressed here is a lower bound for this problem. It would be interesting to arrive at any lower bound under the standard assumption of ETH.

Part II

Popular Matchings

Chapter 6

Introduction to Popular Matchings

A *popular matching problem* instance I comprises a set \mathcal{A} of *agents* and a set \mathcal{H} of *houses*. Each agent a in \mathcal{A} ranks (numbers) a subset of houses in \mathcal{H} (lower rank specify higher preference). The ordered list of houses ranked by $a \in \mathcal{A}$ is called a 's *preference list*. For an agent a , let E_a be the set of pairs (a, h) such that the house h appears on a 's preference list. Define $E = \cup_{a \in \mathcal{A}} E_a$. The problem instance I is then represented by a bipartite graph $G = (\mathcal{A} \cup \mathcal{H}, E)$. A matching M of I is a matching of the bipartite graph G . We use $M(a)$ to denote the house assigned to agent a in M and $M(h)$ to denote the agent that is assigned house h in M . An agent *prefers* a matching M to a matching M' if (i) a is matched in M and unmatched in M' , or (ii) a is matched in both M and M' but a prefers the house $M(a)$ to $M'(a)$. Let $\phi(M, M')$ denote the number of agents that prefer M to M' . We say M is *more popular than* M' if $\phi(M, M') > \phi(M', M)$, and denote it by $M \succ M'$. A matching M is called *popular* if there exists no matching M' such that $M' \succ M$.

The popular matching problem was introduced in [Gär75] as a variation of the stable marriage problem [GS62]. The idea of popular matching has been studied extensively in various settings in recent times [AIKM07, SM10, MI11, Maho6, KMN11, McCo8, Nas14], mostly in the context where only one side has preference of the other side but the other side has no preference at all. We will also focus on this setting. Much of the earlier work focuses on finding efficient algorithms to output a popular matching, if one exists.

The problem of counting the number of "solutions" to a combinatorial question falls into the complexity class #P. An area of interest that has recently gathered a certain amount of attention is the problem of counting stable matchings in graphs. The Gale-Shapely algorithm [GS62] gives a simple and efficient algorithm to output a stable matching, but counting them was proved to be #P-hard in [IL86]. Bhatnagar, Greenberg and Randall [BGRo8] showed that the random walks on the *stable marriage lattice* are slowly mixing, even in very restricted

versions of the problem. [CGM10] gives further evidence towards the conjecture that there may not exist an FPRAS at all for this problem.

Our motivation for this study is largely due to the similarity of structures between stable matchings and popular matchings. The one-sided preferences list setting does not have stability defined. However, in the case where both sides have preference lists (with no ties), both stable and popular matchings are defined. It is also well known that a stable matching is popular and in fact a minimum cardinality popular matching.

The interest is further fueled by the existence of a linear time algorithm to exactly count the number of popular matchings in the standard setting [MI11]. We look at generalizations of the standard version - preferences with ties and houses with capacities. In the case where preferences could have ties, it is already known that the counting version is #P-hard [Nas14]. We give an FPRAS for this problem. In the case where houses have capacities, we prove that the counting version is #P-hard. While the FPRAS for the case of ties is achieved via a reduction to a well known algorithm, the #P-hardness for the capacitated case is more involved, making it the more interesting setting of the problem.

We now formally describe the different variants of the popular matching problem (borrowing the notation from [SM10]) and also describe our results alongside.

House Allocation problem (HA) These are the instances $G = (\mathcal{A} \cup \mathcal{H}, E)$ where the preference list of each agent $a \in \mathcal{A}$ is a linear order. Let $n = |\mathcal{A}| + |\mathcal{H}|$ and $m = |E|$. In [AIKM07], Abraham et al. give a complete characterization of popular matchings in an HA instance, using which they give an $O(m + n)$ time algorithm to check if the instance admits a popular matching and to obtain the largest such matching, if one exists. The question of counting popular matchings was first addressed in [MI11], where McDermid et al. give a new characterization by introducing a powerful structure called the *switching graph* of an instance. The switching graph encodes all the popular matchings via *switching paths* and *switching cycles*. Using this structure, they give a linear time algorithm to count the number of popular matchings.

House Allocation problem with Ties (HAT) An instance $G = (\mathcal{A} \cup \mathcal{H}, E)$ of HAT can have applicants whose preference list contains ties. For example, the preference list of an agent could be $[h_3, (h_1, h_4), h_2]$, meaning, house h_3 gets rank 1, houses h_1 and h_4 get a tied rank 2 and house h_2 gets the rank 3. A characterization for popular matchings in HAT was given in [AIKM07]. The characterization is used to give an $O(\sqrt{n}m)$ time algorithm to solve the maximum cardinality popular matching problem. We outline their characterization briefly in Chapter 7 where we consider the problem of counting popular matchings in HAT. In

[Nas14], Nasre gives a proof of #P-hardness of this problem. We give an FPRAS for this problem by reducing it to the problem of counting perfect matchings in a bipartite graph.

Capacitated House Allocation Problem (CHA) A popular matching instance in CHA has a *capacity* c_i associated with each house $h_i \in \mathcal{H}$, allowing at most c_i agents to be matched to house h_i . The preference list of each agent is strictly ordered. A characterization for popular matchings in CHA was given in [SM10], along with an algorithm to find the largest popular matching (if one exists) in time $O(\sqrt{C}n_1 + m)$, where $n_1 = |\mathcal{A}|$, $m = |E|$ and C is the total capacity of the houses. In Chapter 8, we consider the problem counting popular matchings in CHA. We give a switching graph characterization of popular matchings in CHA. This is similar to the switching graph characterization for HA in [MI11]. Our construction is also motivated from [Nas14], which gives a switching graph characterization of HAT. We use our characterization to prove that it is #P-Complete to compute the number of popular matchings in CHA.

Remark: A natural reduction exists from a CHA instance $G = (\mathcal{A} \cup \mathcal{H}, E)$ to an HAT instance. The reduction is as follows. Treat each house $h_i \in \mathcal{H}$ with capacity c as c different houses h_i^1, \dots, h_i^c of unit capacity, which are always tied together and appear together wherever h_i appears in any agent's preference list. Let the HAT instance thus obtained be G' . It is clear that every popular matching of G is a popular matching of G' . Hence, for example, an algorithm which finds a maximum cardinality popular matching for HAT can be used to find a maximum cardinality popular matching for the CHA instance G . In the context of counting, it is important to note that one popular matching of G may translate to many popular matchings in G' . It is not clear if there is a useful map between these two sets that may help in obtaining either hardness or algorithmic results for counting problems.

Chapter 7

Counting in House Allocation Problem with Ties

In this chapter, we consider the problem of counting the number of popular matchings in House Allocation problem with Ties (HAT). We first describe the characterization given in [AIKM07] here using similar notations. Let $G = (\mathcal{A} \cup \mathcal{H}, E)$ be an HAT instance. For any agent $a \in \mathcal{A}$, let $f(a)$ denote the set of first choices of a . For any house $h \in \mathcal{H}$, define $f(h) := \{a \in \mathcal{A}, f(a) = h\}$. A house h for which $f(h) \neq \emptyset$ is called an f -house. To simplify the definitions, we add a unique last-resort house $l(a)$ with lowest priority for each agent $a \in \mathcal{A}$. This forces every popular matching to be an applicant complete matching.

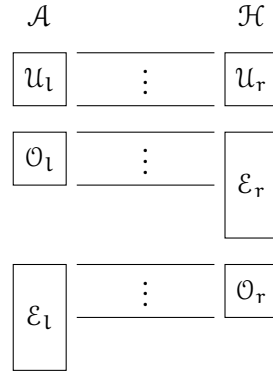
Definition 7.1. (Section 3.1 in [AIKM07]). The **first choice graph** of G is defined to be $G_1 = (\mathcal{A} \cup \mathcal{H}, E_1)$, where E_1 is the set of all rank one edges.

Lemma 7.2. (Lemma 3.1 in [AIKM07]). *If M is a popular matching of G , then $M \cap E_1$ is a maximum matching of G_1 .*

Let M_1 be any maximum matching of G_1 . The matching M_1 can be used to identify the houses h that are always matched to an agent in the set $f(h)$. In this direction, we observe that M_1 defines a partition of the vertices $\mathcal{A} \cup \mathcal{H}$ into three disjoint sets - *even*, *odd* and *unreachable*: a vertex is *even* (resp. *odd*) if there is an even (resp. odd) length alternating path from an unmatched vertex (with respect to M_1) to v ; a vertex v is *unreachable* if there is no alternating path from an unmatched vertex to v . Denote the sets *even*, *odd* and *unreachable* by \mathcal{E} , \mathcal{O} and \mathcal{U} respectively. The following is a well-known theorem in matching theory [LP86].

Lemma 7.3 (Gallai-Edmonds Decomposition). *Let G_1 and M_1 define the partition \mathcal{E} , \mathcal{O} and \mathcal{U} as above. Then,*

- (a) *The sets \mathcal{E} , \mathcal{O} and \mathcal{U} are pairwise disjoint, and every maximum matching in G_1 partitions the vertices of G_1 into the same partition of even, odd and unreachable vertices.*


 FIGURE 7.1: Gallai-Edmonds decomposition of the first-choice graph of G

- (b) In any maximum matching of G_1 , every vertex in \mathcal{U} is matched with another vertex in \mathcal{U} , and every vertex in \mathcal{O} is matched with some vertex in \mathcal{E} . No maximum matching contains an edge between a vertex in \mathcal{O} and a vertex in $\mathcal{O} \cup \mathcal{U}$. The size of a maximum matching is $|\mathcal{O}| + |\mathcal{U}|/2$.
- (c) G_1 contains no edge connecting a vertex in \mathcal{E} with a vertex in \mathcal{U} .

We show the decomposition of G_1 in Figure 7.1, where we look at the bipartitions of \mathcal{U} , \mathcal{O} , and \mathcal{E} into their left and right parts, denoted by subscripts l and r respectively. Since G_1 only contained edges resulting from first-choices, every house in \mathcal{U}_r and \mathcal{O}_r is an f -house. From Lemma 7.3, each such house $h \in \mathcal{U}_r \cup \mathcal{O}_r$ is matched with an agent in $f(h)$ in every maximum matching of G_1 , and correspondingly in every popular matching of G (Lemma 7.2).

For each agent a , define $s(a)$ to be a 's most preferred house(s) in \mathcal{E}_r . Note that $s(a)$ always exists after the inclusion of last-resort houses $l(a)$. The following is proved in [AIKM07].

Lemma 7.4. (Lemma 3.5 in [AIKM07]). *A matching M is popular in G if and only if*

1. $M \cap E_1$ is a maximum matching of G_1 , and
2. for each applicant a , $M(a) \in f(a) \cup s(a)$.

The following hardness result is from [Nas14].

Lemma 7.5. (Theorem 3 in [Nas14]). *Counting the number of popular matchings in HAT is #P-hard.*

We now give an FPRAS for counting the number of popular matchings in the case of ties. As before, let $G = (\mathcal{A} \cup \mathcal{H}, E)$ be our HAT instance. We assume that G admits at least one popular matching (this can be tested using the characterization). We reduce our problem to the problem of counting perfect matchings in a bipartite graph. We start with the first-choice graph G_1 of G , and perform a Gallai-Edmonds decomposition of G_1 using any maximum

matching of G_1 . In order to get a perfect matching instance, we extend the structure obtained from Gallai-Edmonds decomposition described in Figure 7.1. Let \mathcal{F} be the set of f -houses and \mathcal{S} be the set of s -houses. We make use of the following observations in the decomposition.

- Every agent in \mathcal{U}_l and \mathcal{O}_l gets one of their first-choice houses in every popular matching.
- \mathcal{E}_r can be further partitioned into useful sets. Define \mathcal{E}_r^f to be the set of f -houses but not s -houses, \mathcal{E}_r^s to be the set of s -houses but not f -houses, $\mathcal{E}_r^{f/s}$ to be the set of houses which are both f -houses and s -houses and finally define \mathcal{E}_r^* to be the set of houses that are neither f -houses or s -houses.
- \mathcal{O}_l can only match with houses in $\mathcal{E}_r^f \cup \mathcal{E}_r^{f/s}$ in every popular matching.

These observations are described in Figure 7.2(a).

Next, we observe that every agent in \mathcal{E}_l that is already not matched to a house in \mathcal{O}_r , must match to a house in $\mathcal{E}_r^s \cup \mathcal{E}_r^{f/s}$. We facilitate this by adding all edges $(a, s(a))$ for each agent in \mathcal{E}_l . Finally, we add a set of dummy agent vertices \mathcal{D} on the left side to balance the bipartition. The size of \mathcal{D} is $|\mathcal{A}| - (|\mathcal{F}| - |\mathcal{E}_r^*|)$. This difference is non-negative since we assumed every agent has a last-resort house. We make the bipartition $(\mathcal{D}, \mathcal{E}_r^f \cup \mathcal{E}_r^{f/s} \cup \mathcal{E}_r^s)$ a complete bipartite graph by adding the appropriate edges. This allows us to move from one popular matching to another by switching between first and second-choices and, among second choices of agents. Finally, we remove set \mathcal{E}_r^* from the right side. The new structure is described in Figure 7.2(b). Denote the new graph by G' . By an application of Hall's theorem, we note that G' admits a perfect matching.

Lemma 7.6. *The number of popular matchings in G is $|\mathcal{D}|!$ times the number of perfect matchings in G' .*

Proof. Consider a perfect matching M of G' . Let the matching M' be obtained by removing from M all the edges coming out of the set \mathcal{D} . Observe that $M' \cap E_1$ is a maximum matching of G_1 . This is because the sets \mathcal{U}_l , \mathcal{O}_l and \mathcal{O}_r are always matched in M' (or else M would not be a perfect matching of G') and that the size of a maximum matching in G_1 is $(|\mathcal{U}_l| + |\mathcal{O}_l| + |\mathcal{O}_r|)$ by Lemma 7.3. Also, each agent in \mathcal{A} is matched to either a house in \mathcal{F} or in \mathcal{S} by the construction of graph G' . Using Lemma 7.4, we conclude that M is a popular matching of G . Finally, observe that every popular matching in M in G can be augmented to a perfect matching of G' by adding exactly $|\mathcal{D}|$ edges. This follows again from Lemma 7.3 and Lemma 7.4. \square

We now make use of the following result of Jerrum et al. from [JSV01].

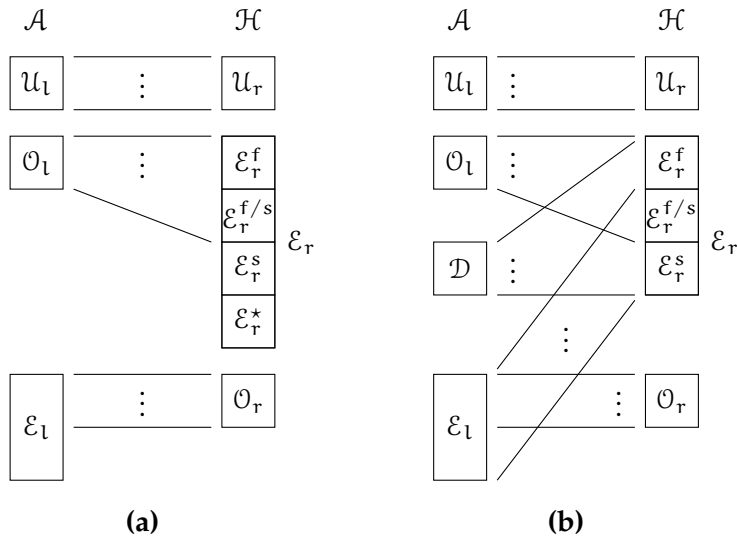


FIGURE 7.2: Reduction to a perfect-matching instance by extending the Gallai-Edmonds decomposition of G_1 .

Lemma 7.7. (Theorem 1.1 in [JSV01]). *There exists an FPRAS for the problem of counting number of perfect matchings a bipartite graph.*

From Lemma 7.6 and Lemma 7.7, we have the following.

Theorem 7.8. *There exists an FPRAS for counting the number of popular matchings in the House Allocation problem with Ties.*

Chapter 8

Counting in Capacitated House Allocation Problem

In this chapter, we consider the structure of popular matchings in Capacitated House Allocation problem (CHA). A CHA instance I consists of agents \mathcal{A} and houses \mathcal{H} . Let $|\mathcal{A}| = n$ and $|\mathcal{H}| = m$. Let $c : \mathcal{H} \rightarrow \mathbb{Z}_{>0}$ be the capacity function for houses. Each agent orders a subset of the houses in a strict order creating its *preference list*. The preference list of $a_i \in \mathcal{A}$ defines a set of edges E_i from a_i to houses in \mathcal{H} . Define $E = \cup_{i \in [n]} E_i$. The problem instance I can then be represented by a bipartite graph $G = (\mathcal{A} \cup \mathcal{H}, E)$.

For the instance I , a *matching* M is a subset of E such that each agent appears in at most one edge in M and each house h appears in at most $c(h)$ edges in M . The definitions of *more popular than* relationship between two matchings and *popular matching* is same as described earlier in Chapter 6.

We now outline a characterization of popular matchings in CHA from [SM10]. As before, denote by $f(a)$ the first choice of an agent $a \in \mathcal{A}$. A house which is the first choice of at least one agent is called an *f-house*. For each house $h \in \mathcal{H}$, define $f(h) = \{a \in \mathcal{A}, f(a) = h\}$. For each agent $a \in \mathcal{A}$, we add a unique last-resort house $l(a)$ with least priority and capacity 1.

Lemma 8.1. (Lemma 1 in [SM10]) *If M is a popular matching then for each f-house h , $|M(h) \cap f(h)| = \min\{c(h), |f(h)|\}$.*

For each agent $a \in \mathcal{A}$, define $s(a)$ to be the highest ranked house h on a 's preference list such that one of the following is true:

- h is not an f-house, or,
- h is an f-house but $h \neq f(a)$ and $|f(h)| < c(h)$.

Notice that $s(a)$ always exists after the inclusion of last-resort houses $l(a)$. The following lemma gives the characterization of popular matchings in G .

Lemma 8.2. (Theorem 1 in [SM10]) *A matching M is popular if and only if*

1. *for every f -house $h \in \mathcal{H}$,*
 - *if $|f(h)| \leq c(h)$, then every agent in $f(h)$ is matched to the house h ,*
 - *else, house h is matched to exactly $c(h)$ agents, all belonging to $f(h)$,*
2. *M is an agent complete matching such that for each agent $a \in \mathcal{A}$, $M(a) \in \{f(a), s(a)\}$.*

8.1 Switching Graph Characterization of CHA

We now give a *switching graph* characterization of popular matchings for instances from this class. Our results are motivated from similar characterizations for HA in [MI11] and for HAT in [Nas14]. A switching graph for an instance allows us to move from one popular matching to another by making well defined walks on the switching graph.

Consider a popular matching M of an instance G of CHA. The switching graph of G with respect to M is a directed weighted graph $G_M = (\mathcal{H}, E_M)$, with the edge set E_M defined as follows. For every agent $a \in \mathcal{A}$,

- add a directed edge from $M(a)$ to $\{f(a), s(a)\} \setminus M(a)$,
- if $M(a) = f(a)$, assign a weight of -1 on this edge, otherwise assign a weight of $+1$.

Associated with the switching graph G_M , we have an *unsaturation degree* function $u_M : \mathcal{H} \rightarrow \mathbb{Z}_{\geq 0}$, defined $u_M(h) = c(h) - |M(h)|$. A vertex h is called *saturated* if its unsaturation degree is 0, i.e. $u_M(h) = 0$. If $u_M(h) > 0$, h is called *unsaturated*. We make use of the following terminology in the foregoing discussion. We now describe some useful properties of the switching graph G_M .

▷ *Property 1:* Each vertex h can have out-degree at most $c(h)$.

Proof. A house h has a maximum capacity $c(h)$, it can only get matched to at most $c(h)$ agents. □

▷ *Property 2:* Let M and M' be two different popular matchings in G and let G_M and $G_{M'}$ denote the switching graphs respectively. For any vertex house h , the number of -1 outgoing edges from h is invariant across G_M and $G_{M'}$. The number of $+1$ incoming

edges on h is also invariant across G_M and $G_{M'}$.

Proof. From Lemma 8.1, in any popular matching, each f -house h is matched to exactly $\min\{f(h), c(h)\}$ agents and this is also the number of outgoing edges with weight -1 . A similar argument can be made for $+1$ weighted incoming edges. \square

▷ *Property 3:* No $+1$ weighted edge can end at an unsaturated vertex.

Proof. If a $+1$ weighted edge is incident on a vertex h , this means that the house h is an f -house for some agent a that is still not matched to it in M . But if h is unsaturated then it still has some unused capacity. The matching M' obtained by just promoting a to h is popular than M , which is a contradiction. \square

▷ *Property 4:* There can be no incoming -1 weighted edge on a saturated vertex if all its outgoing edges have weight -1 .

Proof. A -1 weighted edge on a vertex h implies that the house h is an s -house for some agent a . But if h is saturated with all outgoing edges having a weight of -1 , then all the capacity of h has been used up by agents who had h as their first choice. But by definition, h can not be an s -house for any other agent. \square

▷ *Property 5:* For a given vertex h , if there exists at least one $+1$ weighted incoming edge, then all outgoing edges are of weight -1 and there can be no -1 weighted incoming edge on h .

Proof. Let agent a correspond to any $+1$ weighted incoming edge. Suppose h has an outgoing $+1$ edge ending at a vertex h' and agent a' corresponds to this edge. We can promote agents a and a' to their first choices and demote any agent which is assigned house h' . This leads to a matching more popular than M . Hence all outgoing edges from h must be of weight -1 . Further, Property 3 and Property 4 together imply that there can be no incoming edge on h of weight -1 . \square

Switching Moves

We now describe the operation on the switching graph which takes us from one popular matching to another. We make use of the following terminology with reference to the switching graph G_M . Note that the term “path” (“cycle”) implies a “directed path” (“directed cycle”). A “ $+1$ edge” (“ -1 edge”) means an “edge with weight $+1$ ” (“edge with weight -1 ”).

- A path is called an *alternating path* if it starts with a $+1$ edge, ends at a -1 edge and alternates between $+1$ and -1 edges.
- A *switching path* is an alternating path that ends at an unsaturated vertex.

- A *switching cycle* is an even length cycle of alternating -1 and $+1$ weighted edges.
- A *switching set* is a union of edge-disjoint switching cycles and switching paths, such that at most k switching paths end a vertex of unsaturation degree k .
- A *switching move* is an operation on G_M by a switching set S in which, for every edge e in S , we reversed the direction of e and flip the weight of e ($+1 \leftrightarrow -1$).

Observe that every switching graph inherently implies a matching (in the context of CHA) of G .

Let $G_M = (\mathcal{H}, E_M)$ and $G_{M'} = (\mathcal{H}, E_{M'})$ be the switching graphs associated with popular matchings M and M' of the CHA instance $G = (\mathcal{A} \cup \mathcal{H}, E)$. Observe that the underlying undirected graph of G_M and $G_{M'}$ are same. We have the following.

Theorem 8.3. *Let S be the set of edges in G_M that get reversed in $G_{M'}$. Then, S is a switching set for G_M .*

We prove this algorithmically in stages.

Lemma 8.4. *Every directed cycle in S is a switching cycle of G_M .*

Proof. Let C be any cycle in S . From Property 5 of switching graphs, we know that no vertex in C can have an incoming edge and an outgoing edge of same weight $+1$. Similarly, since S is the set of edges in G_M which have opposite directions and opposite weights in $G_{M'}$, we observe that S can not contain any vertex with incoming and outgoing edges both having weight -1 (again from Property 5). This forces the weights of cycle C to alternate between $+1$ and -1 . Moreover, this alternation forces the cycle to be of even length. \square

At this stage we apply the following algorithm to the set S .

Reduction(S):

1. while (there exists a switching cycle C in S):

 let $S := S \setminus C$

2. while (S is non-empty):

 (a) find a maximal path P in S which alternates between weights $+1$ and -1

 (b) let $S := S \setminus P$

At the end of every iteration of the while loop in Step 1, Lemma 8.4 still holds true. We now prove a very crucial invariant of the while loop in Step 2.

Lemma 8.5. *In every iteration of the while loop in Step 2 of the algorithm Reduction, the longest path in step 2(a) is a switching path for G_M .*

Proof. Let us denote the stages of the run of algorithm Reduction by t . Initially, at $t = 0$, before any of the while loops run, S is exactly the difference of edges in E_M and E'_M . Let the while loop in Step 1 runs t_1 times and the while loop in Step 2 runs t_2 times.

Let the current stage be $t = t_1 + i$. Let P be the maximal path in step 2(a) at this stage. We show that P starts with an edge of weight $+1$. For contradiction, let (h_i, h_j) be an edge of weight -1 and that this is the first edge of path P . Let a_{ij} be the agent associated with the edge (h_i, h_j) .

The Property 5 of switching sets precludes any incoming edge of weight -1 on the vertex h_i . Hence, no switching path could have ended at h_i at any stage $t < t_1 + i$. Similarly, no switching cycle with an incoming edge -1 was incident on h_i at an earlier stage.

Let us assume that there were r cycles that were incident at h_i at $t = 0$. At stage $t = t_1 + i$, let the number of outgoing -1 edges be m . Hence at $t = 0$, h_i had r incoming $+1$ edges and $r + m$ outgoing -1 edges. But this would also imply that at $t = 0$, h_i had $r + m$ incoming $+1$ edges in $G_{M'}$. This contradicts Property 2, requiring the number of incoming $+1$ edges to be constant in the switching graphs corresponding to different popular matchings.

A similar argument can be made for the fact that the path P can only end at an edge with weight -1 and that P ends at an unsaturated vertex. \square

The following theorem establishes the characterization for popular matchings in CHA.

Theorem 8.6. *If G_M is the switching graph of the CHA instance G with respect to a popular matching M , then*

- (i) *every switching move on G_M generates another popular matching, and*
- (ii) *every popular matching of G can be generated by a switching move on M .*

Proof.

- (i) We verify that the new matching generated by applying a switching move on G_M satisfies the characterization in Lemma 8.2. Call the new switching graph $G_{M'}$ and the associated matching M' . First, observe that M' is indeed an agent complete matching since $G_{M'}$ still has a directed edge for each agent in \mathcal{A} . Next, each agent a is still matched to $f(a)$ or $s(a)$ as the switching move either reverses an edge of G_M or leaves it as it is. Finally, for each house h , $f(h) \subseteq M'(h)$ if $|f(h)| < c(h)$ and $|M'(h)| = c(h)$ with $M'(h) \subseteq f(h)$ otherwise. This is true because $|M'(h)| = |M(h)|$, by the definition of switching moves.

(ii) This is implied by Theorem 8.3.

□

8.2 Hardness of Counting

In this section we prove the #P-hardness of counting popular matchings in CHA. We reduce the problem of counting the number of matchings in a bipartite graph to our problem.

Let $G = (A \cup B, E)$ be a bipartite matching instance in which we want to count the number of matchings. From G we create a CHA instance I such that the number of popular matchings of I is same as the number of matchings of G .

Observe that a description of a switching graph gives the following information about its instance:

- the set of agents \mathcal{A} ,
- for each agent $a \in \mathcal{A}$, it gives $f(a)$ and $s(a)$, and
- for each s -house or f -house h , the unsaturation degree gives the capacity $c(h)$.

Using this information, we can create the description of the instance I so that it meets our requirement. For simplicity, we assume G to be connected (as isolated vertices do not affect the count). We orient all the edges of G from A to B and call the directed graph $G' = (A \cup B, E')$. Using G' , we construct a graph S , which will be the switching graph.

Let $|A| = n_1$, $|B| = n_2$ and $|E'| = m$. S is constructed by augmenting G' . We keep all the vertices and edges of G' in S and assign each edge a weight of -1 . Further, for each vertex $u \in A$, add a copy u' and add a directed edge from u' to u , and assign a weight of $+1$ to the edge. Call the new set of vertices A' . The sets A' and B contain s -houses and the set A contains f -houses. We label every vertex in A' and A as *saturated* and for each vertex v in B , we label v as *unsaturated* with *unsaturation degree* 1. Hence, the switching graph S has $2n_1 + n_2$ vertices and $n_1 + m$ edges.

The CHA instance I corresponding to the switching graph S has $2n_1 + n_2$ houses and $n_1 + m$ agents. Each agent has a preference list of length 2 that is naturally defined by the weight of edges in S .

Let the popular matching represented by S be M_ϕ . This corresponds to the empty matching of G . Every non-empty matching of G can be obtained by a switching move on S . We make this more explicit in the following theorem.

Theorem 8.7. *The number of matchings in G is same as the number of popular matchings in I .*

Proof. We prove this by showing that each matching in G corresponds to a unique set of edge disjoint switching paths in the switching graph S of I .

Consider a matching M of G and let $(u, v) \in M$. We look at the length 2 directed path in S that is obtained by extending (u, v) in the reverse direction: $u' \rightarrow u \rightarrow v$ with $u' \in A'$. It's easy to see that this is a switching path for I . Moreover, the set of switching paths obtained from any matching of G forms a valid switching set (as every pair of such paths arising from a matching are always edge disjoint).

For the converse, observe that S can only have switching paths of length 2 and it has no switching cycles. An edge disjoint set of such paths corresponds to a matching of G . By the definition of S , it's easy to see every matching in M can be obtained by a switching set of S . □

Chapter 9

Conclusion

9.1 Transitivity

We have presented an algorithm that given a directed graph on n vertices and m arcs outputs a maximal transitive sub-graph in time $O(n^2 + nm)$. This is the first algorithm for finding maximal transitive subgraph that we know of, that does better than the usual greedy algorithm. Although it might be the case that this is an optimal algorithm, we are unable to prove a lower bound for this problem.

There are many related problems for which one might expect similar kind of algorithm - that is $O(n^3)$ time algorithm that does better than the usual greedy algorithm. We would like to present them as open problems:

1. Given a directed graph G on n vertices and a transitive subgraph H of G , check if H is a maximal transitive subgraph of G .
2. Given a directed graph G on n vertices and a subgraph H of G , find a maximal transitive subgraph of G that also contains H .

Obviously an algorithm for the second problem would also give an algorithm for the first problem.

In the case of Maximum Transitive Subgraph (MTS) problem, we give a 0.25-approximation for the general problem and 0.874-approximation for the triangle-free case (underlying graph being triangle-free). The obvious question that arises here is that of improving these approximation ratios. We also give an upper bound on the size of MTS being $m/4 + cm^{4/5}$ for some $c > 0$. Since this bound is achieved via showing upper bounds to max-cut, it

further shows that the approach of using max-cut approximation can't have a better constant approximation ratio than $1/4$.

9.2 Popular Matching

We looked at the different versions of the popular matching problem from a counting and approximation perspective. Specifically, we look at the cases of preferences with ties and houses with capacities. In the case where preferences could have ties, it was already known that the counting version is #P-hard. We complemented this by obtaining an FPRAS for this problem.

In the case where houses have capacities, we prove that the counting version is #P-hard. The question of obtaining an FPRAS for this case still remains open. Such an algorithm will complete the picture of approximate-counting in the different settings of popular matching problems.

Bibliography

- [ABG⁺07] Noga Alon, Béla Bollobás, András Gyárfás, Jenö Lehel, and Alex D. Scott. Maximum directed cuts in acyclic digraphs. *Journal of Graph Theory*, 55(1):1–13, 2007.
- [AIKM07] David J. Abraham, Robert W. Irving, Telikepalli Kavitha, and Kurt Mehlhorn. Popular matchings. *SIAM J. Comput.*, 37(4):1030–1045, 2007.
- [Alo96] Noga Alon. Bipartite subgraphs. *Combinatorica*, 16(3):301–311, 1996.
- [ALS88] Stefan Arnborg, Jens Lagergren, and Detlef Seese. Problems easy for tree-decomposable graphs (extended abstract). In *Automata, Languages and Programming, 15th International Colloquium, ICALP88, Tampere, Finland, July 11-15, 1988, Proceedings*, pages 38–51, 1988.
- [BGR08] Nayantara Bhatnagar, Sam Greenberg, and Dana Randall. Sampling stable marriages: why spouse-swapping won’t work. In *SODA*, pages 1223–1232, 2008.
- [BIM10] Péter Biró, Robert W. Irving, and David Manlove. Popular matchings in the marriage and roommates problems. In *Algorithms and Complexity, 7th International Conference, CIAC 2010, Rome, Italy, May 26-28, 2010. Proceedings*, pages 97–108, 2010.
- [CFK⁺15] Marek Cygan, Fedor V. Fomin, Lukasz Kowalik, Daniel Lokshtanov, Dániel Marx, Marcin Pilipczuk, Michal Pilipczuk, and Saket Saurabh. *Parameterized Algorithms*. Springer, 2015.
- [CGJR15] Sourav Chakraborty, Shamik Ghosh, Nitesh Jha, and Sasanka Roy. Maximal and maximum transitive relation contained in a given binary relation. In *Computing and Combinatorics - 21st International Conference, COCOON 2015, Beijing, China, August 4-6, 2015, Proceedings*, pages 587–600, 2015.
- [CGM10] Prasad Chebolu, Leslie Ann Goldberg, and Russell A. Martin. The complexity of approximately counting stable matchings. In *APPROX-RANDOM*, pages 81–94, 2010.

- [CJ16a] Sourav Chakraborty and Nitesh Jha. Exact algorithms for maximum transitive subgraph problem. submitted, 2016.
- [CJ16b] Sourav Chakraborty and Nitesh Jha. On the size of maximum directed cuts in triangle free graphs. submitted, 2016.
- [Cou90] Bruno Courcelle. The monadic second-order logic of graphs. i. recognizable sets of finite graphs. *Inf. Comput.*, 85(1):12–75, 1990.
- [DF99] Rodney G. Downey and Michael R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.
- [EFPS88] Paul Erdős, Ralph J. Faudree, János Pach, and Joel H. Spencer. How to make a graph bipartite. *J. Comb. Theory, Ser. B*, 45(1):86–98, 1988.
- [FG06] Jörg Flum and Martin Grohe. *Parameterized Complexity Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.
- [Gär75] Peter Gärdenfors. Match making: assignments based on bilateral preferences. *Behavioral Science*, 20(3):166–173, 1975.
- [GJS76] M. R. Garey, David S. Johnson, and Larry J. Stockmeyer. Some simplified np-complete graph problems. *Theor. Comput. Sci.*, 1(3):237–267, 1976.
- [GS62] David Gale and Lloyd S Shapley. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.
- [GW95] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, 1995.
- [IL86] Robert W. Irving and Paul Leather. The complexity of counting stable marriages. *SIAM J. Comput.*, 15(3):655–667, 1986.
- [JS89] Mark Jerrum and Alistair Sinclair. Approximating the permanent. *SIAM J. Comput.*, 18(6):1149–1178, 1989.
- [JSV01] Mark Jerrum, Alistair Sinclair, and Eric Vigoda. A polynomial-time approximation algorithm for the permanent of a matrix with non-negative entries. In *STOC*, pages 712–721, 2001.
- [Kav14] Telikepalli Kavitha. A size-popularity tradeoff in the stable marriage problem. *SIAM J. Comput.*, 43(1):52–71, 2014.
- [KKMO07] Subhash Khot, Guy Kindler, Elchanan Mossel, and Ryan O’Donnell. Optimal inapproximability results for MAX-CUT and other 2-variable csps? *SIAM J. Comput.*, 37(1):319–357, 2007.

- [KMN11] Telikepalli Kavitha, Julián Mestre, and Meghana Nasre. Popular mixed matchings. *Theor. Comput. Sci.*, 412(24):2679–2690, 2011.
- [LLZ02] Michael Lewin, Dror Livnat, and Uri Zwick. Improved rounding techniques for the MAX 2-sat and MAX DI-CUT problems. In *Integer Programming and Combinatorial Optimization, 9th International IPCO Conference, Cambridge, MA, USA, May 27-29, 2002, Proceedings*, pages 67–82, 2002.
- [LP86] L. Lovász and M. D. Plummer. *Matching theory*, volume 121 of *North-Holland Mathematics Studies*. North-Holland Publishing Co., Amsterdam, 1986. *Annals of Discrete Mathematics*, 29.
- [Mah06] Mohammad Mahdian. Random popular matchings. In *ACM Conference on Electronic Commerce*, pages 238–242, 2006.
- [McCo8] Richard Matthew McCutchen. The least-unpopularity-factor and least-unpopularity-margin criteria for matching problems with one-sided preferences. In *LATIN*, pages 593–604, 2008.
- [MI11] Eric McDermid and Robert W. Irving. Popular matchings: structure and algorithms. *J. Comb. Optim.*, 22(3):339–358, 2011.
- [Nas14] Meghana Nasre. Popular matchings: Structure and strategic issues. *SIAM J. Discrete Math.*, 28(3):1423–1448, 2014.
- [Nie06] Rolf Niedermeier. *Invitation to fixed-parameter algorithms*. Oxford University Press, 2006.
- [RS07] Venkatesh Raman and Somnath Sikdar. Parameterized complexity of the induced subgraph problem in directed graphs. *Inf. Process. Lett.*, 104(3):79–85, 2007.
- [SM10] Colin T. Sng and David Manlove. Popular matchings in the weighted capacitated house allocation problem. *J. Discrete Algorithms*, 8(2):102–116, 2010.
- [WKNU12] Mathias Weller, Christian Komusiewicz, Rolf Niedermeier, and Johannes Uhlmann. On making directed graphs transitive. *J. Comput. Syst. Sci.*, 78(2):559–574, 2012.
- [Yan78] Mihalis Yannakakis. Node- and edge-deletion np-complete problems. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing, May 1-3, 1978, San Diego, California, USA*, pages 253–264, 1978.