# Maximal and maximum transitive relation contained in a given binary relation

Sourav Chakraborty[1], Shamik Ghosh[2], Nitesh Jha[1], and Sasanka Roy[1]

[1] Chennai Mathematical Institute, Chennai, India
e-mail: {sourav, nj, sasanka}@cmi.ac.in
[2] Department of Mathematics, Jadavpur University, Kolkata, India
e-mail: sghosh@math.jdvu.ac.in

**Abstract.** We study the problem of finding a *maximal* transitive relation contained in a given binary relation. Given a binary relation of size $m$ defined on a set of size $n$, we present a polynomial time algorithm that finds a maximal transitive sub-relation in time $O(n^2 + nm)$.
We also study the problem of finding a *maximum* transitive relation contained in a binary relation. For the class of triangle-free relations (directed graphs), we present a 0.874-approximation via the problem of maximum directed cut.

## 1 Introduction

All relations considered in this study are binary relations. We represent a relation alternately as a digraph to simplify the presentation at places (see Section 2 for definitions). 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 $\rho$, checking whether $\rho$ is transitive, finding the transitive closure of $\rho$, finding the maximum transitive relation contained in $\rho$, partitioning $\rho$ 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 straightforward 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 it. This problem was proven to be NP-complete in [11]. 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-DICUT problem (see Section 4). MAX-DICUT has well known inapproximability results.

We can also relate it to a problem of optimisation 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 $(i, j)$ in $V$, create a boolean variable $x_{ij}$. Consider the following 3SAT formula.

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

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 $(x_{ij} \vee \overline{x}_{ik} \vee \overline{x}_{kj})$ 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'.

### 1.1 Our Results

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 subrelation in a given relation $\rho$. Our algorithm runs in time $O(n^3)$ where $n$ is the size of set on which the relation $\rho$ 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 $\rho$. To the best of our knowledge, no better algorithm for finding a maximal transitive sub-relation is known. We present the algorithms and the proof of correctness related to the following theorem in Section 3.

**Theorem 1.** *Let $\rho$ be a binary relation on a set of size $n$ and let $m$ be the size of $\rho$. There is an algorithm that given $\rho$, outputs a maximal transitive relation contained in $\rho$, in time $O(n^2 + nm)$.*

In the case of finding a *maximum* transitive relation contained in a binary relation, we present the following results.

**Theorem 2.** *There exists a 0.25-approximation algorithm for obtaining a maximum transitive relation contained in a binary relation.*

**Theorem 3.** *There exists a 0.874-approximation algorithm for finding the maximum transitive relation contained in a triangle-free relation (triangle-free digraph).*

These results have been arrived at by connecting the the problem of maximum transitive 'sub-relations' to the well known problems of MAX-CUT and MAX-DICUT. This has been detailed in Section 4.

### 1.2 Related Results

The transitive property is a fundamental property of binary relations. Various important algorithmic problems with respect to transitive property has been studied and used. One very important and well studied problem is finding the transitive closure of a binary relation $\rho$ (that is the smallest binary relation which contains $\rho$ and is transitive). This problem of finding transitive closure has been studied way back in 1960s. Warshall [9] gave an algorithm to find the transitive closure is time $O(n^3)$, where $n$ is the size of the set on which the binary relation is defined. Using different techniques [5] gave an $O(n^2 + nm)$ algorithms, where $m$ is the number of relations in $\rho$. Modifying the algorithm of Warshall, Nuutila [8] connected the problem of finding transitive closure with matrix multiplication. With the latest knowledge of matrix multiplication ( [4] and [10]) we can compute the transitive closure of a binary relation on $n$ elements using $O(n^{2.37})$ time complexity.

Another important problem connected to transitive property is the finding the transitive reduction of a binary relation. Transitive reduction of a binary relation $\rho$ is the minimal sub-relation whose transitive closure is same as the transitive closure of $\rho$. This was introduce by Aho et al [1] and they also gave the tight complexity bounds. A closely related concept to the transitive reduction is the maximal equivalent graph, introduced by Moyles [7].

Given a binary relation, partitioning it as a union of transitive relations is another very important related problem (see [2]). A plethora of work has been done on this problem in recent times as this problem has found application in biomedical studies.

## 2 Notations

Let $S = \{1, 2, \ldots, n\}$, where $n$ is a natural number. A binary relation $\rho$ on $S$ is a subset of the cross product $S \times S$. We only consider binary relations in this study. Any relation $\rho$ on $S$ can be represented by a $(0, 1)$ matrix $A = (a_{ij})_{n \times n}$ of size $n \times n$, where

$$a_{ij} = \begin{cases} 1, \text{ if } (i, j) \in \rho \\ 0, \text{ otherwise.} \end{cases}$$

Similarly, a relation $\rho$ on $S$ can represented by a directed graph with $S$ as the vertex set and elements of $\rho$ as the arcs of the directed graph.

In this paper we do not distinguish between a relation and its matrix representation or its directed graph representation. So for a given relation $\rho$, if $(i, j) \in \rho$, we sometimes refer to it as the arc $(i, j)$ being present and sometimes as the adjacency matrix entry $\rho_{ij} = 1$.

If $\rho$ is a binary relation on $S$ then the size of $\rho$ (denoted by $m$) is the number of arcs in the directed graph corresponding to $\rho$. In other words, it is the number of pairs $(i, j) \in S \times S$ such that $(i, j) \in \rho$.

If $\rho$ is a binary relation on $S$ we say $\rho'$ is contained in $\rho$ (or is a sub-relation) if for all $i, j \in S$, $(i, j) \in \rho'$ implies $(i, j) \in \rho$.

**Definition 1.** *A binary relation $\rho$ on $S$ is called* transitive *if for all $a, b, c \in S$, $(a, b) \in \rho, (b, c) \in \rho$ implies $(a, c) \in \rho$.*

For a binary relation $\rho$ on $S$ a sub-relation $\alpha$ is said to be a *maximal transitive* relation contained in $\rho$ if there does not exist any transitive relation $\beta$ such that $\alpha$ is strictly contained in $\beta$ and $\beta$ is contained in $\rho$. A *maximum transitive* relation contained in $\rho$ is a largest relation contained in $\rho$.

## 3 Maximal transitive relation finding algorithms

We first present an algorithm which finds a maximal transitive relation contained in a given binary relation in $O(n^3)$ and then we improve it to obtain another algorithm for this with time complexity $O(n^2 + mn)$.

### 3.1 $O(n^3)$ algorithm for finding maximal transitive sub-relation

**Theorem 4.** ***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 2 and the maximality of $T$ is proved in Lemma 3.

**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 ≠ j and a_ik = 0 then
6  |  |  |  |  |  set a_jk = 0
7  |  |  |  |  end
8  |  |  |  |  if k ≠ i and a_kj = 0 then
9  |  |  |  |  |  set a_ki = 0
10 |  |  |  |  end
11 |  |  |  end
12 |  |  end
13 |  end
14 end
15 return A
```

### 3.2 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 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 1.* 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 of the most important lemma for the proof of correctness is the following:

**Lemma 1.** *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 1, $r \geq i$. From point number 5 in Remark 1 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.

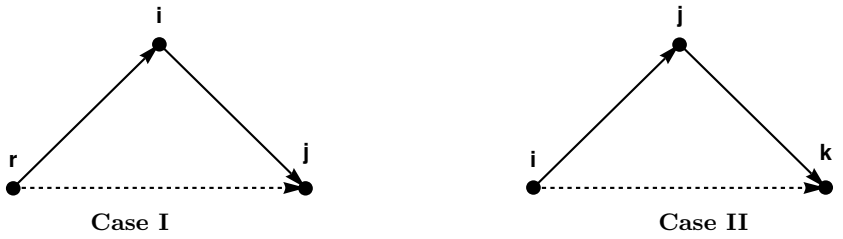


**Fig. 1.** Diagrams of the two cases for Lemma 1

**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 1(left)).

Since no arc is ever created by the algorithm (point 1 in Remark 1), $a_{ri}$ 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 1(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 $r_1 \neq j$.

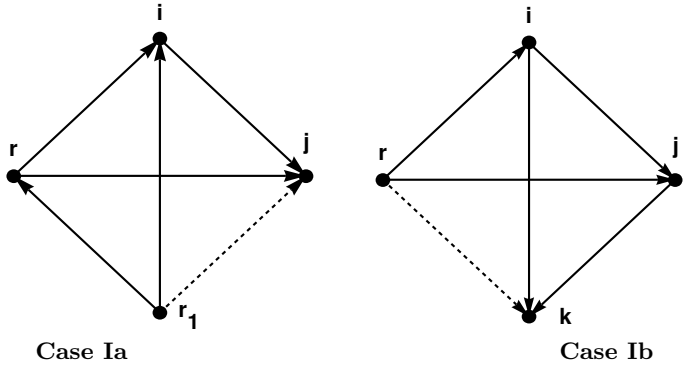


**Fig. 2.** Diagram for subcases of Case 1 for Lemma 1

**Case Ia: ($r_1 \neq j$)**

By Remark 1(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 $(r, i)$ is 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 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 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 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 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 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 1(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.

The second lemma we need is the following. Because of limitation of space we move the proof to the appendix.

**Lemma 2.** *The matrix $T$ output by the **Algorithm 1** is transitive.*

Using Lemma 2 and Lemma 1 we can finally prove the correctness of the algorithm. Unfortunately because of the shortage of space we have to push the proof of the following lemma to the appendix.

**Lemma 3.** *The matrix $T$ output by the **Algorithm 1** is a maximal transitive relation contained in $A$.*

### 3.3 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 5.** *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 4. We calculate only the time complexity of the algorithm and it is given by

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

**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 = ∅
3  |  for each s ← 1 to n, j ≠ i do
4  |  |  if a_ij = 1 then
5  |  |  |  Include j in B_i
6  |  |  end
7  |  end
8  |  for each j ∈ B_i do
9  |  |  for each k = 1 to n do
10 |  |  |  if k ≠ j and a_ik = 0 then
11 |  |  |  |  Make a_jk = 0
12 |  |  |  end
13 |  |  |  if k ≠ i and a_kj = 0 then
14 |  |  |  |  Make a_ki = 0
15 |  |  |  end
16 |  |  end
17 |  end
18 end
```

## 4 Maximum Transitive Relation

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. In the following discussion, a 'graph' is a directed graph unless otherwise stated.

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

**Lemma 4.** *There exists a bipartite subgraph of size $m/2$ in any graph with $m$ edges.*

Obtaining such a bipartite graph deterministically in poly-time is a folklore result. Observe that every bipartite graph is transitive. Given a cut of size $k$ in a directed graph, we can always obtain a transitive subgraph of size at least $k/2$ by considering all the edges in one direction - the direction which has more number of edges. This gives the following.

**Theorem 6.** *There exists a poly-time algorithm to obtain an $m/4$ sized transitive subgraph in any directed graph. This gives a $1/4$-approximation algorithm for maximum transitive subgraph problem.*

The obvious question is - can we do better than $m/4$? We claim that the constant $1/4$ can not be improved in poly-time. For this we consider the class of triangle-free graphs. From a recent result [3], we have the following result.

**Theorem 7.** *For every $m$, there exists a triangle-free graph with $m$ edges in which the size of any directed cut is at most $m/4 + cm^{4/5}$.*

Obtaining a transitive subgraph of size better than $m/4$ (in the constant multiple) would contradict the theorem - since we could input the counterexample triangle-free directed graph and improve our cut size.

In order to improve upon the approximation factor, we focus on the class of *triangle-free* directed graphs. First we make the following simple observation about triangle-free directed graphs.

**Lemma 5.** *In any transitive subgraph of a triangle-free graph, there are no directed paths of length two.*

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. [6] gives an approximation algorithm for the MAX-DICUT problem.

**Theorem 8 (see [6]).** *There exists a 0.874-approximation algorithm for the MAX-DICUT problem.*

As a corollary of Lemma 5, we have the following.

**Lemma 6.** *In a triangle-free graph, every directed cut is also a transitive subgraph.*

This implies that finding the maximum transitive subgraph is same as the MAX-DICUT problem for triangle free graphs.

**Theorem 9.** *There exists a 0.874-approximation algorithm for finding the maximum transitive subgraph in a triangle-free graph.*

## 5 Conclusion

We have presented an algorithm that given a directed graph on $n$ vertices and $m$ arcs outputs a maximal transitive sub-graph is 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.

## References


[1] Alfred V. Aho, M. R. Garey, and Jeffrey D. Ullman. The transitive reduction of a directed graph. *SIAM J. Comput.*, 1(2):131–137, 1972.

[2] FernandoL. Alvarado, Alex Pothen, and Robert Schreiber. Highly parallel sparse triangular solution. In Alan George, JohnR. Gilbert, and JosephW.H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, volume 56 of *The IMA Volumes in Mathematics and its Applications*, pages 141–157. Springer New York, 1993.

[3] Sourav Chakraborty and Nitesh Jha. On the size of maximum directed cuts in triangle free graphs. unpublished, 2015.

[4] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

[5] Paul Walton Purdom Jr. A transitive closure algorithm. *BIT*, 10:76–94, 1970.

[6] 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.

[7] Dennis M. Moyles and Gerald L. Thompson. An algorithm for finding a minimum equivalent graph of a digraph. *J. ACM*, 16(3):455–460, 1969.

[8] Esko Nuutila. Efficient transitive closure computation in large digraphs. *Acta Polytechnica Scandinavia: Math. Comput. Eng.*, 74:1–124, July 1995.

[9] Stephen Warshall. A theorem on boolean matrices. *J. ACM*, 9(1):11–12, 1962.

[10] Virginia Vassilevska Williams. Multiplying matrices faster than coppersmith-winograd. In *Proceedings of the 44th Symposium on Theory of Computing Conference, STOC 2012, New York, NY, USA, May 19 - 22, 2012*, pages 887–898, 2012.

[11] 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.

## Appendix

### Proof of Lemma 2

**Lemma 2.** *The matrix $T$ output by the **Algorithm 1** is transitive.*

*Proof.* Suppose $t_{ij} = 1 = t_{jk}$. By Remark 1(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).

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

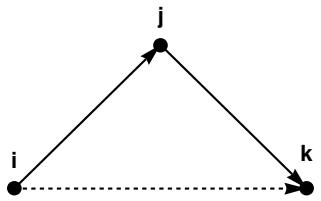


**Fig. 3.** Diagram for Lemma 2

### Proof of Lemma 3

**Lemma 3.** *The matrix $T$ output by the **Algorithm 1** is a maximal transitive relation contained in $A$.*

*Proof.* $T$ is transitive by Lemma 2. Also by Remark 1(1) the output matrix is contained in $A$. So the only thing remaining to prove is that the output matrix $T$ is maximal.

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 1, 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 1(5) and by Lemma 1 we have $r < i$.

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

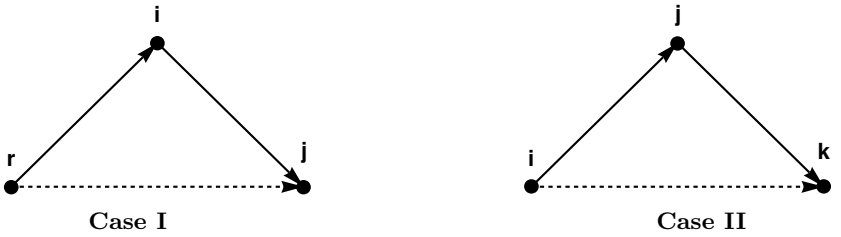


**Fig. 4.** Diagrams of the two cases for Lemma 3

**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 1(1)). But by Lemma 1 $t_{ri} = 1$ as the arc $(r,i)$ is being visited. So $T \cup \{(i,j)\}$ is not transitive (cf. Figure 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 1(1)). But by Lemma 1 $t_{jk} = 1$ as the arc $(j,k)$ is being visited. So $T \cup \{(i,j)\}$ is not transitive (cf. Figure 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 $A$.