Feedback Arc Set

Problem Definition, Hardness Proof, Solution Approach

Faiyaz Rashid **(1405069)**, Rubayet Tusher **(1505004)**
Ishtiak Kabir **(1605013)**, Towsif Abdullah **(1605020)**
Shahjalal Forhad **(1605052)**

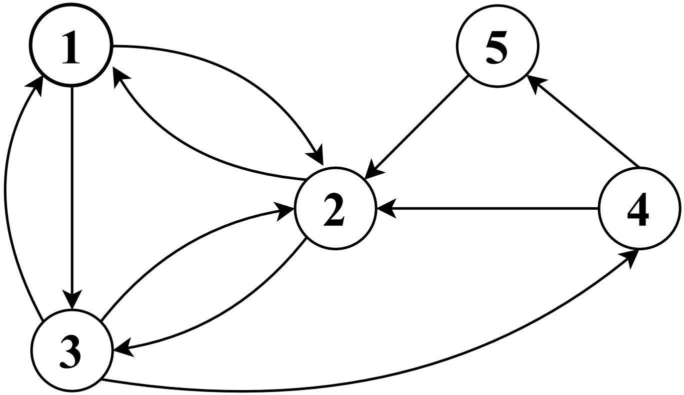
Department of Computer Science & Engineering
Bangladesh University of Engineering & Technology

12 January, 2022

Outline

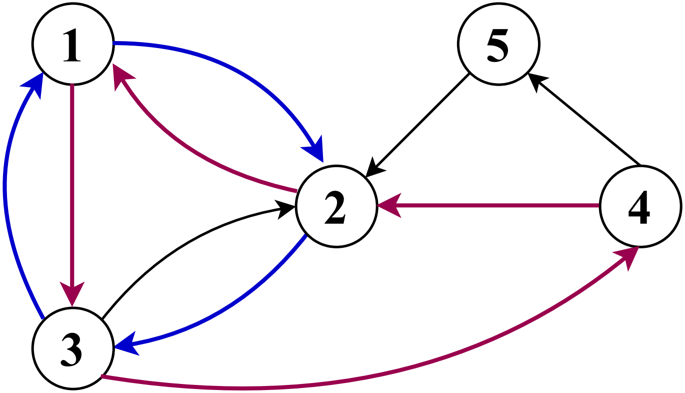
Introduction

Consider a ***directed*** **graph**,



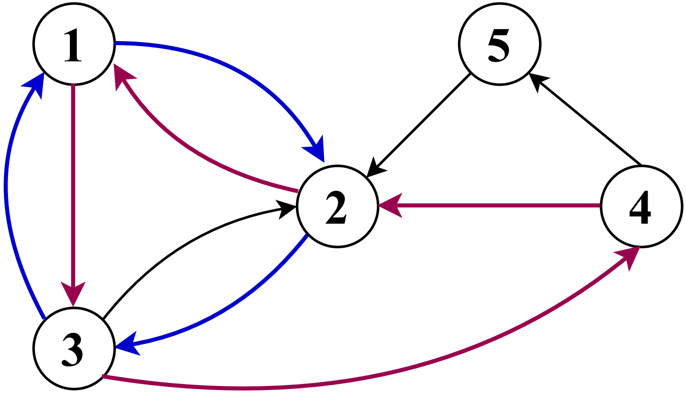
Introduction

Consider a ***directed* graph**,
possibly containing one or more **cycles**.



Introduction

Consider a ***directed*** **graph**,
possibly containing one or more **cycles**.



We want to remove those cycles through **arc deletions**.

Outline

Problem Formulation → Naive

A feedback arc set of a directed graph is a set of arcs whose removal leaves the graph acyclic. More formally,

Problem Formulation → Naive

A feedback arc set of a directed graph is a set of arcs whose removal leaves the graph acyclic. More formally,

Definition (Feedback Arc Set) → Naive Version

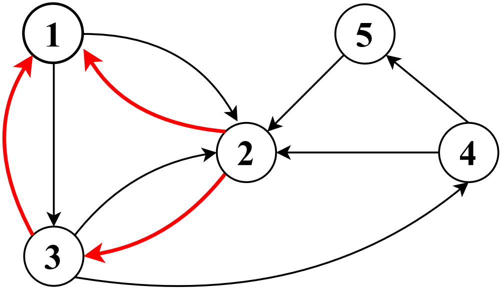
Given a directed graph $G = (V, A)$, a feedback arc set $F \subseteq A$ is a set of arcs such that $G - F$ is acyclic.

Problem Formulation → Naive

A feedback arc set of a directed graph is a set of arcs whose removal leaves the graph acyclic. More formally,

Definition (Feedback Arc Set) → Naive Version

Given a directed graph $G = (V, A)$, a feedback arc set $F \subseteq A$ is a set of arcs such that $G - F$ is acyclic.

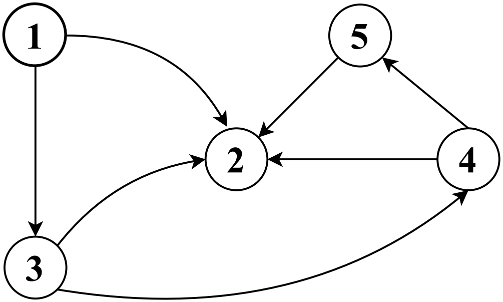


Problem Formulation → Naive

A feedback arc set of a directed graph is a set of arcs whose removal leaves the graph acyclic. More formally,

Definition (Feedback Arc Set) → Naive Version

Given a directed graph $G = (V, A)$, a feedback arc set $F \subseteq A$ is a set of arcs such that $G - F$ is acyclic.



Problem Formulation $\rightarrow$ Optimized

Definition (Feedback Arc Set) $\rightarrow$ Optimization Version

Given a directed graph $G = (V, A)$, find a minimum cardinality feedback arc set of G.

Definition (Feedback Arc Set) $\rightarrow$ Decision Version

Given a directed graph $G = (V, A)$ and an integer k, does G have a feedback arc set of size at most k?

Problem Formulation → Optimized

Definition (Feedback Arc Set) → Optimization Version

Given a directed graph $G = (V, A)$, find a minimum cardinality feedback arc set of G.

Definition (Feedback Arc Set) → Decision Version

Given a directed graph $G = (V, A)$ and an integer k, does G have a feedback arc set of size at most k?

Outline

Hardness Status of FAS

- The decision version of the feedback arc set problem turns out to be *NP*-complete. (Bad news!)
- One of the first 21 problems to be proved *NP*-complete (Karp, 1972)[7]

Proof of *NP*-completeness

To prove that feedback arc set is *NP*-complete, firstly, we need to prove that, FAS is in *NP*.

Given a set of arcs (of size *at most* k), we can check if it is a feedback arc set by first deleting the arcs and then running depth-first search to check if there are any cycles left in the graph.

Clearly, this takes polynomial time. So, the feedback arc set problem is clearly in *NP*.

Proof of *NP*-completeness

To prove that feedback arc set is *NP*-complete, firstly, we need to prove that, FAS is in *NP*.

Given a set of arcs (of size *at most* k), we can check if it is a feedback arc set by first deleting the arcs and then running depth-first search to check if there are any cycles left in the graph.

Clearly, this takes polynomial time. So, the feedback arc set problem is clearly in *NP*.

Proof of *NP*-completeness

To prove that feedback arc set is *NP*-complete, firstly, we need to prove that, FAS is in *NP*.

Given a set of arcs (of size *at most* k), we can check if it is a feedback arc set by first deleting the arcs and then running depth-first search to check if there are any cycles left in the graph.

Clearly, this takes polynomial time. So, the feedback arc set problem is clearly in *NP*.

Proof of *NP*-completeness

Now, we need to prove that, FAS is *NP*-hard.

To do so, we will show that, FAS is ***at least*** as hard as the Vertex Cover problem which is already known to be *NP*-hard.

Claim

VERTEX COVER $\leq_p$ FEEDBACK ARC SET

Proof of *NP*-completeness

Now, we need to prove that, FAS is *NP*-hard.

To do so, we will show that, FAS is ***at least*** as hard as the Vertex Cover problem which is already known to be *NP*-hard.

Claim

VERTEX COVER $\leq_p$ FEEDBACK ARC SET

Proof of *NP*-completeness

Now, we need to prove that, FAS is *NP*-hard.

To do so, we will show that, FAS is ***at least*** as hard as the Vertex Cover problem which is already known to be *NP*-hard.

Claim

VERTEX COVER $\leq_p$ FEEDBACK ARC SET

Why Vertex Cover?

FAS is actually a ***cover*** problem. It makes sense to reduce another ***cover*** problem to it.

In the vertex cover problem, we want to cover all the ***edges*** with ***vertices***.

In the feedback arc set problem, we want to cover all the ***cycles*** with ***arcs***.

So, in the reduction process, there should be some sort of correspondence between,

The things we want to cover: i.e. ***edges*** and ***cycles***

The things that we want to cover with: i.e. ***vertices*** and ***arcs***

Why Vertex Cover?

FAS is actually a ***cover*** problem. It makes sense to reduce another ***cover*** problem to it.

In the vertex cover problem, we want to cover all the ***edges*** with ***vertices***.

In the feedback arc set problem, we want to cover all the ***cycles*** with ***arcs***.

So, in the reduction process, there should be some sort of correspondence between,

The things we want to cover: i.e. ***edges*** and ***cycles***

The things that we want to cover with: i.e. ***vertices*** and ***arcs***

Why Vertex Cover?

FAS is actually a ***cover*** problem. It makes sense to reduce another ***cover*** problem to it.

In the vertex cover problem, we want to cover all the ***edges*** with ***vertices***.

In the feedback arc set problem, we want to cover all the ***cycles*** with ***arcs***.

So, in the reduction process, there should be some sort of correspondence between,

The things we want to cover: i.e. ***edges*** and ***cycles***

The things that we want to cover with: i.e. ***vertices*** and ***arcs***

Why Vertex Cover?

FAS is actually a ***cover*** problem. It makes sense to reduce another ***cover*** problem to it.

In the vertex cover problem, we want to cover all the ***edges*** with ***vertices***.

In the feedback arc set problem, we want to cover all the ***cycles*** with ***arcs***.

So, in the reduction process, there should be some sort of correspondence between,

The things we want to cover: i.e. ***edges*** and ***cycles***

The things that we want to cover with: i.e. ***vertices*** and ***arcs***

Reduction from Vertex Cover

- An instance of the vertex cover problem is an undirected graph $G = (V, E)$ and an integer k.
- An instance of the feedback arc set problem is a directed graph $G' = (V', A')$ and an integer k'.
- Given any vertex cover instance (G, k), we have to construct from it a feedback arc set instance in polynomial time. How can we do it?

Reduction from Vertex Cover

Step 1 (Creation of *Vertex Arcs*)

Split every vertex of G in two i.e. for each vertex $v \in V$, add two vertices v_0 and v_1 in G'. Then add the arc (v_0, v_1).

We call arc $(v_0, v_1) \rightarrow$ the arc "***corresponding to vertex*** v".

Reduction from Vertex Cover

Step 1 (Creation of ***Vertex Arcs***)

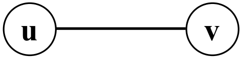
Split every vertex of G in two i.e. for each vertex $v \in V$, add two vertices v_0 and v_1 in G'. Then add the arc (v_0, v_1).

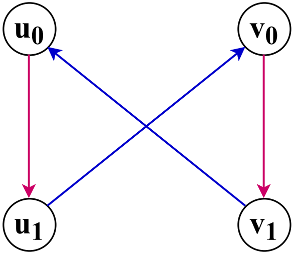
We call arc $(v_0, v_1) \rightarrow$ the arc "***corresponding to vertex*** v".

Reduction from Vertex Cover

Step 2 (Creation of ***Edge Cycles***)

For each edge $e = \{u, v\}$ in E, add the arcs (u_1, v_0) and (v_1, u_0) in A'.



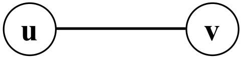


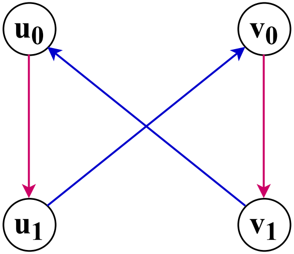
Note how every edge in G corresponds to a directed cycle of length 4 in G'.

Reduction from Vertex Cover

Step 2 (Creation of ***Edge Cycles***)

For each edge $e = \{u, v\}$ in E, add the arcs (u_1, v_0) and (v_1, u_0) in A'.





Note how every edge in G corresponds to a directed cycle of length 4 in G'.

Reduction from Vertex Cover

Step 3 (Set k')

Let $k' = k$.

Clearly, this whole construction process is ***polynomial***.

Claim

G contains a vertex cover of size **at most** k *if and only if* G' contains a feedback arc set of size **at most** k.

Reduction from Vertex Cover

Step 3 (Set k')

Let $k' = k$.

Clearly, this whole construction process is ***polynomial***.

Claim

G contains a vertex cover of size **at most** k *if and only if* G' contains a feedback arc set of size **at most** k.

Reduction from Vertex Cover

Step 3 (Set k')

Let $k' = k$.

Clearly, this whole construction process is ***polynomial***.

Claim

G contains a vertex cover of size **at most** k *if and only if* G' contains a feedback arc set of size **at most** k.

Reduction from Vertex Cover → *Necessity* proof

- The proof is in two parts.
- First, we need to prove that, if G has a vertex cover of size at most k, then G' has a feedback arc set of size *at most* k.
- Let, S be a vertex cover of G with $|S| \leq k$.
- Then $F = \{(v_0, v_1) : v \in S\}$ is a feedback arc set of G'.
- If not, then $G' - F$ contains at least one cycle.
- That cycle uses some arc of the form (u_1, v_0).
- Any cycle that uses the arc (u_1, v_0) has to use both of the arcs (u_0, u_1) and (v_0, v_1).
- So, for such a cycle to exist, $\{(u_0, u_1), (v_0, v_1)\} \cap F = \emptyset$. Therefore, $\{u, v\} \cap S = \emptyset$, a contradiction!

Reduction from Vertex Cover → *Necessity* proof

- The proof is in two parts.
- First, we need to prove that, if G has a vertex cover of size at most k, then G' has a feedback arc set of size *at most* k.
- Let, S be a vertex cover of G with $|S| \leq k$.
- Then $F = \{(v_0, v_1) : v \in S\}$ is a feedback arc set of G'.
- If not, then $G' - F$ contains at least one cycle.
- That cycle uses some arc of the form (u_1, v_0).
- Any cycle that uses the arc (u_1, v_0) has to use both of the arcs (u_0, u_1) and (v_0, v_1).
- So, for such a cycle to exist, $\{(u_0, u_1), (v_0, v_1)\} \cap F = \emptyset$. Therefore, $\{u, v\} \cap S = \emptyset$, a contradiction!

Reduction from Vertex Cover → *Necessity* proof

- The proof is in two parts.
- First, we need to prove that, if G has a vertex cover of size at most k, then G' has a feedback arc set of size *at most* k.
- Let, S be a vertex cover of G with $|S| \leq k$.
- Then $F = \{(v_0, v_1) : v \in S\}$ is a feedback arc set of G'.
- If not, then $G' - F$ contains at least one cycle.
- That cycle uses some arc of the form (u_1, v_0).
- Any cycle that uses the arc (u_1, v_0) has to use both of the arcs (u_0, u_1) and (v_0, v_1).
- So, for such a cycle to exist, $\{(u_0, u_1), (v_0, v_1)\} \cap F = \emptyset$. Therefore, $\{u, v\} \cap S = \emptyset$, a contradiction!

Reduction from Vertex Cover → *Necessity* proof

- The proof is in two parts.
- First, we need to prove that, if G has a vertex cover of size at most k, then G' has a feedback arc set of size *at most* k.
- Let, S be a vertex cover of G with $|S| \leq k$.
- Then $F = \{(v_0, v_1) : v \in S\}$ is a feedback arc set of G'.
- If not, then $G' - F$ contains at least one cycle.
- That cycle uses some arc of the form (u_1, v_0).
- Any cycle that uses the arc (u_1, v_0) has to use both of the arcs (u_0, u_1) and (v_0, v_1).
- So, for such a cycle to exist, $\{(u_0, u_1), (v_0, v_1)\} \cap F = \emptyset$. Therefore, $\{u, v\} \cap S = \emptyset$, a contradiction!

Reduction from Vertex Cover → *Necessity* proof

- The proof is in two parts.
- First, we need to prove that, if G has a vertex cover of size at most k, then G' has a feedback arc set of size *at most* k.
- Let, S be a vertex cover of G with $|S| \leq k$.
- Then $F = \{(v_0, v_1) : v \in S\}$ is a feedback arc set of G'.
- If not, then $G' - F$ contains at least one cycle.
- That cycle uses some arc of the form (u_1, v_0).
- Any cycle that uses the arc (u_1, v_0) has to use both of the arcs (u_0, u_1) and (v_0, v_1).
- So, for such a cycle to exist, $\{(u_0, u_1), (v_0, v_1)\} \cap F = \emptyset$. Therefore, $\{u, v\} \cap S = \emptyset$, a contradiction!

Reduction from Vertex Cover → *Necessity* proof

- The proof is in two parts.
- First, we need to prove that, if G has a vertex cover of size at most k, then G' has a feedback arc set of size *at most* k.
- Let, S be a vertex cover of G with $|S| \leq k$.
- Then $F = \{(v_0, v_1) : v \in S\}$ is a feedback arc set of G'.
- If not, then $G' - F$ contains at least one cycle.
- That cycle uses some arc of the form (u_1, v_0).
- Any cycle that uses the arc (u_1, v_0) has to use both of the arcs (u_0, u_1) and (v_0, v_1).
- So, for such a cycle to exist, $\{(u_0, u_1), (v_0, v_1)\} \cap F = \emptyset$. Therefore, $\{u, v\} \cap S = \emptyset$, a contradiction!

Reduction from Vertex Cover → *Necessity* proof

- The proof is in two parts.
- First, we need to prove that, if G has a vertex cover of size at most k, then G' has a feedback arc set of size *at most* k.
- Let, S be a vertex cover of G with $|S| \leq k$.
- Then $F = \{(v_0, v_1) : v \in S\}$ is a feedback arc set of G'.
- If not, then $G' - F$ contains at least one cycle.
- That cycle uses some arc of the form (u_1, v_0).
- Any cycle that uses the arc (u_1, v_0) has to use both of the arcs (u_0, u_1) and (v_0, v_1).
- So, for such a cycle to exist, $\{(u_0, u_1), (v_0, v_1)\} \cap F = \emptyset$. Therefore, $\{u, v\} \cap S = \emptyset$, a contradiction!

Reduction from Vertex Cover $\rightarrow$ *Necessity* proof

- The proof is in two parts.
- First, we need to prove that, if G has a vertex cover of size at most k, then G' has a feedback arc set of size *at most* k.
- Let, S be a vertex cover of G with $|S| \leq k$.
- Then $F = \{(v_0, v_1) : v \in S\}$ is a feedback arc set of G'.
- If not, then $G' - F$ contains at least one cycle.
- That cycle uses some arc of the form (u_1, v_0).
- Any cycle that uses the arc (u_1, v_0) has to use both of the arcs (u_0, u_1) and (v_0, v_1).
- So, for such a cycle to exist, $\{(u_0, u_1), (v_0, v_1)\} \cap F = \emptyset$. Therefore, $\{u, v\} \cap S = \emptyset$, a contradiction!

Reduction from Vertex Cover → *Sufficiency* proof

- Now, we need to prove in the opposite direction.
- If G' contains a feedback arc set of size at most k, then G contains a vertex cover of size *at most* k.
- Let, F be a feedback arc set of G' with $|F| \leq k$.
- Without loss of generality, F only contains arcs of the form (v_0, v_1).
- This is because, any cycle that an arc of the form (u_1, v_0) participates in, also contains the arc (v_0, v_1). So, deleting the arc (v_0, v_1) instead of the arc (u_1, v_0) removes at least as many cycles (if not more)
- The vertices corresponding to the arcs in F constitute a vertex cover in G.
- If not, then some edge $\{u, v\}$ of G will be uncovered.
- As a result, F will not contain any arc from the length-4 cycle u_0, u_1, v_0, v_1, u_0, which is again, a contradiction!

Reduction from Vertex Cover → *Sufficiency* proof

- Now, we need to prove in the opposite direction.
- If G' contains a feedback arc set of size at most k, then G contains a vertex cover of size *at most* k.
- Let, F be a feedback arc set of G' with $|F| \leq k$.
- Without loss of generality, F only contains arcs of the form (v_0, v_1).
- This is because, any cycle that an arc of the form (u_1, v_0) participates in, also contains the arc (v_0, v_1). So, deleting the arc (v_0, v_1) instead of the arc (u_1, v_0) removes at least as many cycles (if not more)
- The vertices corresponding to the arcs in F constitute a vertex cover in G.
- If not, then some edge $\{u, v\}$ of G will be uncovered.
- As a result, F will not contain any arc from the length-4 cycle u_0, u_1, v_0, v_1, u_0, which is again, a contradiction!

Reduction from Vertex Cover → *Sufficiency* proof

- Now, we need to prove in the opposite direction.
- If G' contains a feedback arc set of size at most k, then G contains a vertex cover of size *at most* k.
- Let, F be a feedback arc set of G' with $|F| \leq k$.
- Without loss of generality, F only contains arcs of the form (v_0, v_1).
- This is because, any cycle that an arc of the form (u_1, v_0) participates in, also contains the arc (v_0, v_1). So, deleting the arc (v_0, v_1) instead of the arc (u_1, v_0) removes at least as many cycles (if not more)
- The vertices corresponding to the arcs in F constitute a vertex cover in G.
- If not, then some edge $\{u, v\}$ of G will be uncovered.
- As a result, F will not contain any arc from the length-4 cycle u_0, u_1, v_0, v_1, u_0, which is again, a contradiction!

Reduction from Vertex Cover → *Sufficiency* proof

- Now, we need to prove in the opposite direction.
- If G' contains a feedback arc set of size at most k, then G contains a vertex cover of size *at most* k.
- Let, F be a feedback arc set of G' with $|F| \leq k$.
- Without loss of generality, F only contains arcs of the form (v_0, v_1).
- This is because, any cycle that an arc of the form (u_1, v_0) participates in, also contains the arc (v_0, v_1). So, deleting the arc (v_0, v_1) instead of the arc (u_1, v_0) removes at least as many cycles (if not more)
- The vertices corresponding to the arcs in F constitute a vertex cover in G.
- If not, then some edge $\{u, v\}$ of G will be uncovered.
- As a result, F will not contain any arc from the length-4 cycle u_0, u_1, v_0, v_1, u_0, which is again, a contradiction!

Reduction from Vertex Cover → *Sufficiency* proof

- Now, we need to prove in the opposite direction.
- If G' contains a feedback arc set of size at most k, then G contains a vertex cover of size *at most* k.
- Let, F be a feedback arc set of G' with $|F| \leq k$.
- Without loss of generality, F only contains arcs of the form (v_0, v_1).
- This is because, any cycle that an arc of the form (u_1, v_0) participates in, also contains the arc (v_0, v_1). So, deleting the arc (v_0, v_1) instead of the arc (u_1, v_0) removes at least as many cycles (if not more)
- The vertices corresponding to the arcs in F constitute a vertex cover in G.
- If not, then some edge $\{u, v\}$ of G will be uncovered.
- As a result, F will not contain any arc from the length-4 cycle u_0, u_1, v_0, v_1, u_0, which is again, a contradiction!

Reduction from Vertex Cover → *Sufficiency* proof

- Now, we need to prove in the opposite direction.
- If G' contains a feedback arc set of size at most k, then G contains a vertex cover of size *at most* k.
- Let, F be a feedback arc set of G' with $|F| \leq k$.
- Without loss of generality, F only contains arcs of the form (v_0, v_1).
- This is because, any cycle that an arc of the form (u_1, v_0) participates in, also contains the arc (v_0, v_1). So, deleting the arc (v_0, v_1) instead of the arc (u_1, v_0) removes at least as many cycles (if not more)
- The vertices corresponding to the arcs in F constitute a vertex cover in G.
- If not, then some edge $\{u, v\}$ of G will be uncovered.
- As a result, F will not contain any arc from the length-4 cycle u_0, u_1, v_0, v_1, u_0, which is again, a contradiction!

Reduction from Vertex Cover → *Sufficiency* proof

- Now, we need to prove in the opposite direction.
- If G' contains a feedback arc set of size at most k, then G contains a vertex cover of size *at most* k.
- Let, F be a feedback arc set of G' with $|F| \leq k$.
- Without loss of generality, F only contains arcs of the form (v_0, v_1).
- This is because, any cycle that an arc of the form (u_1, v_0) participates in, also contains the arc (v_0, v_1). So, deleting the arc (v_0, v_1) instead of the arc (u_1, v_0) removes at least as many cycles (if not more)
- The vertices corresponding to the arcs in F constitute a vertex cover in G.
- If not, then some edge $\{u, v\}$ of G will be uncovered.
- As a result, F will not contain any arc from the length-4 cycle u_0, u_1, v_0, v_1, u_0, which is again, a contradiction!

Reduction from Vertex Cover → *Sufficiency* proof

- Now, we need to prove in the opposite direction.
- If G' contains a feedback arc set of size at most k, then G contains a vertex cover of size *at most* k.
- Let, F be a feedback arc set of G' with $|F| \leq k$.
- Without loss of generality, F only contains arcs of the form (v_0, v_1).
- This is because, any cycle that an arc of the form (u_1, v_0) participates in, also contains the arc (v_0, v_1). So, deleting the arc (v_0, v_1) instead of the arc (u_1, v_0) removes at least as many cycles (if not more)
- The vertices corresponding to the arcs in F constitute a vertex cover in G.
- If not, then some edge $\{u, v\}$ of G will be uncovered.
- As a result, F will not contain any arc from the length-4 cycle u_0, u_1, v_0, v_1, u_0, which is again, a contradiction!

Feedback Arc Set is *NP*-complete!

Theorem

The feedback arc set problem is *NP*-complete.

Outline

How to approach for FAS solution

As now we know, FEEDBACK ARC SET is NP-complete, unless $P{=}NP$, there do *not* exist polynomial time algorithms that can solve FEEDBACK ARC SET *exactly*.

Any *exact* algorithm for FEEDBACK ARC SET must contend with the fact that it might take exponential time on some input instances.

But, there are some Good news also! By using *clever* algorithmic techniques, we can sometimes have *exact* algorithms that are significantly better than the *naive* brute-force algorithms. And, then there are also Parameterized & Approximation algorithms.

So, what algorithms are we going to explore?

How to approach for FAS solution

As now we know, FEEDBACK ARC SET is NP-complete, unless $P=NP$, there do *not* exist polynomial time algorithms that can solve FEEDBACK ARC SET *exactly*.

Any *exact* algorithm for FEEDBACK ARC SET must contend with the fact that it might take exponential time on some input instances.

But, there are some Good news also! By using *clever* algorithmic techniques, we can sometimes have *exact* algorithms that are significantly better than the *naive* brute-force algorithms. And, then there are also Parameterized & Approximation algorithms.

So, what algorithms are we going to explore?

How to approach for FAS solution

As now we know, FEEDBACK ARC SET is NP-complete, unless $P{=}NP$, there do *not* exist polynomial time algorithms that can solve FEEDBACK ARC SET *exactly*.

Any *exact* algorithm for FEEDBACK ARC SET must contend with the fact that it might take exponential time on some input instances.

But, there are some Good news also! By using *clever* algorithmic techniques, we can sometimes have *exact* algorithms that are significantly better than the *naive* brute-force algorithms. And, then there are also Parameterized & Approximation algorithms.

So, what algorithms are we going to explore?

How to approach for FAS solution

As now we know, FEEDBACK ARC SET is NP-complete, unless $P{=}NP$, there do *not* exist polynomial time algorithms that can solve FEEDBACK ARC SET *exactly*.

Any *exact* algorithm for FEEDBACK ARC SET must contend with the fact that it might take exponential time on some input instances.

But, there are some Good news also! By using *clever* algorithmic techniques, we can sometimes have *exact* algorithms that are significantly better than the *naive* brute-force algorithms. And, then there are also Parameterized & Approximation algorithms.

So, what algorithms are we going to explore?

Algorithms to Explore

1. Exact Algorithms
   - Brute-Force
     - Naive approach $\rightarrow \mathcal{O}^*(2^{n^2})$
     - Slightly better (insightful) approach $\rightarrow \mathcal{O}^*(n!)$
   - Dynamic Programming $\rightarrow \mathcal{O}^*(2^n)$ with space $\mathcal{O}^*(2^n)$
   - Divide & Conquer $\rightarrow \mathcal{O}^*(4^n)$ with space $\mathcal{O}^*(1)$
2. Parameterized Algorithms
3. Approximation Algorithms
4. **Polynomial Time** Algorithm for *Restricted* instances!

Outline

Exact Algorithms → Brute-Force

- Brute-Force algorithms usually have very bad running times and are only feasible on the smallest of input instances.
- Then, why should we even care about brute-force algorithms in the first place?
- Because, they can often be a *launchpad* for more sophisticated exact algorithms.
- By understanding what makes brute-force algorithms inefficient, we can sometimes avoid unnecessary computation and end up with algorithms that have better guaranteed running times. Soon, we are going to see an example of this.

Exact Algorithms → Brute-Force

- Brute-Force algorithms usually have very bad running times and are only feasible on the smallest of input instances.
- Then, why should we even care about brute-force algorithms in the first place?
- Because, they can often be a *launchpad* for more sophisticated exact algorithms.
- By understanding what makes brute-force algorithms inefficient, we can sometimes avoid unnecessary computation and end up with algorithms that have better guaranteed running times. Soon, we are going to see an example of this.

Exact Algorithms → Brute-Force

- Brute-Force algorithms usually have very bad running times and are only feasible on the smallest of input instances.
- Then, why should we even care about brute-force algorithms in the first place?
- Because, they can often be a *launchpad* for more sophisticated exact algorithms.
- By understanding what makes brute-force algorithms inefficient, we can sometimes avoid unnecessary computation and end up with algorithms that have better guaranteed running times. Soon, we are going to see an example of this.

Exact Algorithms → Brute-Force

- Brute-Force algorithms usually have very bad running times and are only feasible on the smallest of input instances.
- Then, why should we even care about brute-force algorithms in the first place?
- Because, they can often be a *launchpad* for more sophisticated exact algorithms.
- By understanding what makes brute-force algorithms inefficient, we can sometimes avoid unnecessary computation and end up with algorithms that have better guaranteed running times. Soon, we are going to see an example of this.

Brute-Force → Naive Approach

- Let us first try to solve Feedback Arc Set in the most naive way possible. What is that?
- We can look at every possible subset of the arcs and check if it is a Feedback Arc Set or not.
- And, this gives us the following algorithm.

Brute-Force → Naive Approach

- Let us first try to solve FEEDBACK ARC SET in the most naive way possible. What is that?
- We can look at every possible subset of the arcs and check if it is a FEEDBACK ARC SET or not.
- And, this gives us the following algorithm.

Brute-Force → Naive Approach

- Let us first try to solve FEEDBACK ARC SET in the most naive way possible. What is that?
- We can look at every possible subset of the arcs and check if it is a FEEDBACK ARC SET or not.
- And, this gives us the following algorithm.

NaiveFeedbackArcSet(G)

Algorithm 1 NaiveFeedbackArcSet(G)

Input: A directed graph $G = (V, A)$
Output: A smallest possible set $F \subseteq A$ such that $G - F$ is acyclic

```
1: $m \leftarrow \infty$;
2: $F^* \leftarrow \emptyset$;
3: for all $F \subseteq$A do
4:     $G' \leftarrow G - F$;
5:     if $G'$ is acyclic and $|F| < m$ then
6:         $m \leftarrow |F|$;
7:         $F^* \leftarrow F$;
8:     end if
9: end for
   return $F^*$
```

Time complexity of NAIVEFEEDBACKARCSET(G)

- How bad is this algorithm?
- This algorithm always has to look at every possible subset of the arcs.
- Therefore, this algorithm always has a running time of $\mathcal{O}^*(2^m)$, where m is the number of arcs in the graph. For dense graphs, $m = \Theta(n^2)$ and so, the running time is $\mathcal{O}^*(2^{n^2})$.
- Terrible! Only feasible on the tiniest of input graphs.

Time complexity of NaiveFeedbackArcSet(G)

- How bad is this algorithm?
- This algorithm always has to look at every possible subset of the arcs.
- Therefore, this algorithm always has a running time of $\mathcal{O}^*(2^m)$, where m is the number of arcs in the graph. For dense graphs, $m = \Theta(n^2)$ and so, the running time is $\mathcal{O}^*(2^{n^2})$.
- Terrible! Only feasible on the tiniest of input graphs.

Time complexity of NAIVEFEEDBACKARCSET(G)

- How bad is this algorithm?
- This algorithm always has to look at every possible subset of the arcs.
- Therefore, this algorithm always has a running time of $\mathcal{O}^*(2^m)$, where m is the number of arcs in the graph. For dense graphs, $m = \Theta(n^2)$ and so, the running time is $\mathcal{O}^*(2^{n^2})$.
- Terrible! Only feasible on the tiniest of input graphs.

Time complexity of NAIVEFEEDBACKARCSET(G)

- How bad is this algorithm?
- This algorithm always has to look at every possible subset of the arcs.
- Therefore, this algorithm always has a running time of $\mathcal{O}^*(2^m)$, where m is the number of arcs in the graph. For dense graphs, $m = \Theta(n^2)$ and so, the running time is $\mathcal{O}^*(2^{n^2})$.
- Terrible! Only feasible on the tiniest of input graphs.

A Better Brute-Force Algorithm

We can come up with a slightly cleverer algorithm by using the fact that, every directed acyclic graph has a topological ordering.

Definition (Topological Ordering)

A topological ordering is a permutation of the vertices in which for every arc (u, v), u comes before v in the permutation.

A Better Brute-Force Algorithm

We can come up with a slightly cleverer algorithm by using the fact that, every directed acyclic graph has a topological ordering.

Definition (Topological Ordering)

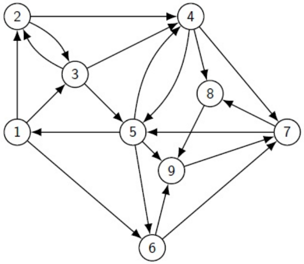
A topological ordering is a permutation of the vertices in which for every arc (u, v), u comes before v in the permutation.

A Better Brute-Force Algorithm

The idea:

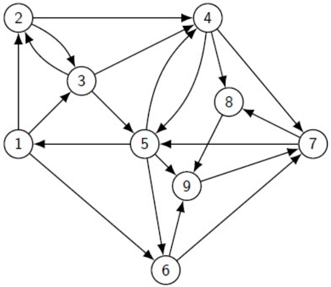
Consider any arbitrary permutation of the vertices.

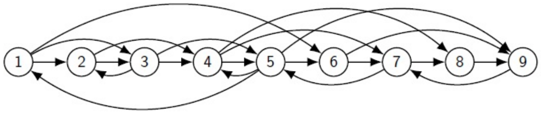
Then the set of "backward" arcs constitute a feedback arc set.



A Better Brute-Force Algorithm

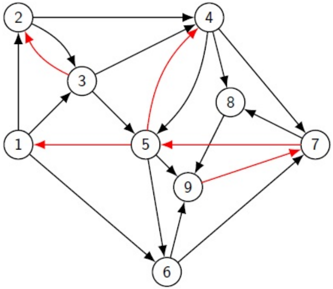
The idea:
Consider any arbitrary permutation of the vertices.
Then the set of "backward" arcs constitute a feedback arc set.

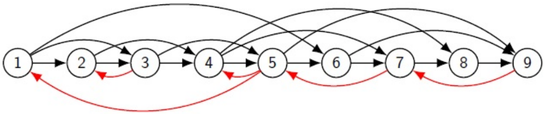




A Better Brute-Force Algorithm

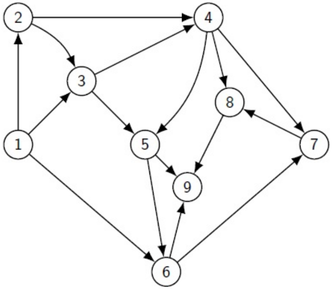
The idea:
Consider any arbitrary permutation of the vertices.
Then the set of "backward" arcs constitute a feedback arc set.

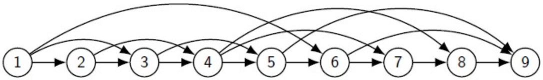




A Better Brute-Force Algorithm

The idea:
Consider any arbitrary permutation of the vertices.
Then the set of "backward" arcs constitute a feedback arc set.





A Better Brute-Force Algorithm

More formally, we have the following theorem.

Theorem

Let $G = (V, A)$ be a directed graph with $V = \{v_1, v_2, \ldots, n\}$ and π be a permutation of the numbers $1, 2, \ldots, n$.
Let, $F = \{(v_{\pi(i)}, v_{\pi(j)}) \in A : \pi(i) > \pi(j)\}$. Then $G - F$ is acyclic.

A Better Brute-Force Algorithm

- This gives us the idea for another algorithm for FEEDBACK ARC SET.
- For every possible permutation of the vertices, count the number of "**backward**" arc that results in.
- Pick a permutation that results in the least number of "**backward**" arcs.

A Better Brute-Force Algorithm

- This gives us the idea for another algorithm for FEEDBACK ARC SET.
- For every possible permutation of the vertices, count the number of "**backward**" arc that results in.
- Pick a permutation that results in the least number of "**backward**" arcs.

A Better Brute-Force Algorithm

- This gives us the idea for another algorithm for FEEDBACK ARC SET.
- For every possible permutation of the vertices, count the number of "**backward**" arc that results in.
- Pick a permutation that results in the least number of "**backward**" arcs.

PermutationFeedbackArcSet(G)

Algorithm 2 PermutationFeedbackArcSet(G)

Input: A directed graph $G = (V, A)$
Output: A smallest possible set $F \subseteq A$ such that $G - F$ is acyclic

```
1: $m \leftarrow \infty$;
2: $F^* \leftarrow \emptyset$;
3: for all permutation $\pi$ of the numbers $1, 2, \dots, |V|$ do
4:     $F \leftarrow \{v_{\pi(i)}, v_{\pi(j)}) \in A : \pi(i) > \pi(j)\}$;
5:     if $|F| < m$ then
6:         $m \leftarrow |F|$;
7:         $F^* \leftarrow F$;
8:     end if
9: end for
   return $F^*$
```

Analysis of PermutationFeedbackArcSet(G)

- The running time of PermutationFeedbackArcSet(G) is $\mathcal{O}^*(n!)$ (since it looks at every possible permutation of the vertices).
- Better than before but still not good enough.
- But, it does offer us with a very important insight.

Analysis of PermutationFeedbackArcSet(G)

- The running time of PermutationFeedbackArcSet(G) is $\mathcal{O}^*(n!)$ (since it looks at every possible permutation of the vertices).
- Better than before but still not good enough.
- But, it does offer us with a very important insight.

Analysis of PERMUTATIONFEEDBACKARCSET(G)

- The running time of PERMUTATIONFEEDBACKARCSET(G) is $\mathcal{O}^*(n!)$ (since it looks at every possible permutation of the vertices).
- Better than before but still not good enough.
- But, it does offer us with a very important insight.

The Insight

The Insight

Feedback Arc Set can be thought of as finding a permutation of the vertices with the minimum **"cost"**.

- Therefore, we might be able to exploit techniques used in solving *other* optimal permutation or sequencing problems (The **TSP** for example).
- In particular, we are going to consider the dynamic programming approach which has been very successful in solving such problems.

The Insight

The Insight

Feedback Arc Set can be thought of as finding a permutation of the vertices with the minimum **"cost"**.

- Therefore, we might be able to exploit techniques used in solving *other* optimal permutation or sequencing problems (The **TSP** for example).
- In particular, we are going to consider the dynamic programming approach which has been very successful in solving such problems.

The Insight

The Insight

FEEDBACK ARC SET can be thought of as finding a permutation of the vertices with the minimum **"cost"**.

- Therefore, we might be able to exploit techniques used in solving *other* optimal permutation or sequencing problems (The **TSP** for example).
- In particular, we are going to consider the dynamic programming approach which has been very successful in solving such problems.

Beating Brute Force → Dynamic Programming

- Now we are going to use the insight from the previous slides and give a dynamic programming algorithm for the FEEDBACK ARC SET problem.
- We are essentially going to mimic the idea used in classical Held-Karp algorithm (1962) for solving the TRAVELING SALESPERSON PROBLEM [6].

Beating Brute Force → Dynamic Programming

- Now we are going to use the insight from the previous slides and give a dynamic programming algorithm for the FEEDBACK ARC SET problem.
- We are essentially going to mimic the idea used in classical Held-Karp algorithm (1962) for solving the TRAVELING SALESPERSON PROBLEM [6].

Setting Up the Dynamic Program

- Let us first formally write down what we want.
- We have a directed graph $G = (V, A)$ with $V = \{v_1, v_2, \ldots, v_n\}$.
- What we want is a permutation of the vertices that minimizes the number of **"backward"** arcs.
- In other words, we want a permutation of π of the numbers $1, 2, \ldots, n$ that minimizes the cardinality of the following set.

$$F = \{(v_{\pi(i)}, v_{\pi(j)}) \in A : \pi(i) > \pi(j)\}$$

Setting Up the Dynamic Program

- Let us first formally write down what we want.
- We have a directed graph $G = (V, A)$ with $V = \{v_1, v_2, \ldots, v_n\}$.
- What we want is a permutation of the vertices that minimizes the number of **"backward"** arcs.
- In other words, we want a permutation of π of the numbers $1, 2, \ldots, n$ that minimizes the cardinality of the following set.

$$F = \{(v_{\pi(i)}, v_{\pi(j)}) \in A : \pi(i) > \pi(j)\}$$

Setting Up the Dynamic Program

- Let us first formally write down what we want.
- We have a directed graph $G = (V, A)$ with $V = \{v_1, v_2, \ldots, v_n\}$.
- What we want is a permutation of the vertices that minimizes the number of **"backward"** arcs.
- In other words, we want a permutation of π of the numbers $1, 2, \ldots, n$ that minimizes the cardinality of the following set.

$$F = \{(v_{\pi(i)}, v_{\pi(j)}) \in A : \pi(i) > \pi(j)\}$$

Setting Up the Dynamic Program

- Let us first formally write down what we want.
- We have a directed graph $G = (V, A)$ with $V = \{v_1, v_2, \dots, v_n\}$.
- What we want is a permutation of the vertices that minimizes the number of **"backward"** arcs.
- In other words, we want a permutation of π of the numbers $1, 2, \dots, n$ that minimizes the cardinality of the following set.

$$F = \{(v_{\pi(i)}, v_{\pi(j)}) \in A : \pi(i) > \pi(j)\}$$

Setting Up the Dynamic Program

- Let us first formally write down what we want.
- We have a directed graph $G = (V, A)$ with $V = \{v_1, v_2, \ldots, v_n\}$.
- What we want is a permutation of the vertices that minimizes the number of **"backward"** arcs.
- In other words, we want a permutation of π of the numbers $1, 2, \ldots, n$ that minimizes the cardinality of the following set.

$$F = \{(v_{\pi(i)}, v_{\pi(j)}) \in A : \pi(i) > \pi(j)\}$$

Setting Up the Dynamic Program

- To design a dynamic programming algorithm, we must first define the sub-problems.
- In our formulation, we will have one sub-problem per each subset of the vertices.

The Sub-problems

For every non-empty $S \subseteq V$, let $OPT[S]$ be the size of a minimum feedback arc set of the graph induced by the vertices of S.

Setting Up the Dynamic Program

- To design a dynamic programming algorithm, we must first define the sub-problems.
- In our formulation, we will have one sub-problem per each subset of the vertices.

The Sub-problems

For every non-empty $S \subseteq V$, let $OPT[S]$ be the size of a minimum feedback arc set of the graph induced by the vertices of S.

Setting Up the Dynamic Program

- To design a dynamic programming algorithm, we must first define the sub-problems.
- In our formulation, we will have one sub-problem per each subset of the vertices.

The Sub-problems

For every non-empty $S \subseteq V$, let $OPT[S]$ be the size of a minimum feedback arc set of the graph induced by the vertices of S.

Setting Up the Dynamic Program

- We now know that the value of $OPT[S]$ corresponds to a permutation of the vertices in S that results in the least number of backward arcs.
- By conditioning on the *last* vertex appears in such a permutation, we can express the value of $OPT[S]$ in the following way.

The Recurrence

$$OPT[S] = \min_{v \in S}\{OPT[S - \{v\}] + c(v, S - \{v\})\}$$

where $c(v, S - \{v\})$ is the number arcs going from v to $S - \{v\}$.

Setting Up the Dynamic Program

- We now know that the value of $OPT[S]$ corresponds to a permutation of the vertices in S that results in the least number of backward arcs.
- By conditioning on the *last* vertex appears in such a permutation, we can express the value of $OPT[S]$ in the following way.

The Recurrence

$$OPT[S] = \min_{v \in S}\{OPT[S - \{v\}] + c(v, S - \{v\})\}$$

where $c(v, S - \{v\})$ is the number arcs going from v to $S - \{v\}$.

Setting Up the Dynamic Program

- We now know that the value of $OPT[S]$ corresponds to a permutation of the vertices in S that results in the least number of backward arcs.
- By conditioning on the *last* vertex appears in such a permutation, we can express the value of $OPT[S]$ in the following way.

The Recurrence

$$OPT[S] = \min_{v \in S}\{OPT[S - \{v\}] + c(v, S - \{v\})\}$$

where $c(v, S - \{v\})$ is the number arcs going from v to $S - \{v\}$.

Setting Up the Dynamic Program

The Recurrence

$$OPT[S] = \min_{v \in S}\{OPT[S - \{v\}] + c(v, S - \{v\})\}$$

where $c(v, S - \{v\})$ is the number arcs going from v to $S - \{v\}$.

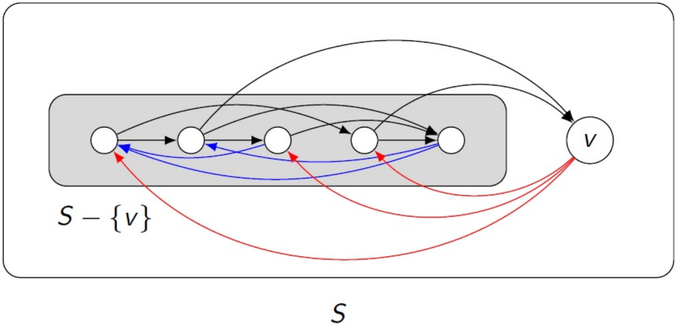


Figure 1: Number of blue arcs $= OPT[S - \{v\}]$, Number of red arcs $= c(v, S - \{v\})$

Setting Up the Dynamic Program

- The size of a minimum feedback arc set of the graph is therefore $OPT[V]$.
- A recurrence like the one shown in the previous slide can be transformed into a dynamic programming algorithm by solving sub-problems in order of their sizes.
- The following algorithm can be attributed to Lawler (1964) [9]

Setting Up the Dynamic Program

- The size of a minimum feedback arc set of the graph is therefore $OPT[V]$.
- A recurrence like the one shown in the previous slide can be transformed into a dynamic programming algorithm by solving sub-problems in order of their sizes.
- The following algorithm can be attributed to Lawler (1964) [9]

Setting Up the Dynamic Program

- The size of a minimum feedback arc set of the graph is therefore $OPT[V]$.
- A recurrence like the one shown in the previous slide can be transformed into a dynamic programming algorithm by solving sub-problems in order of their sizes.
- The following algorithm can be attributed to Lawler (1964) [9]

DP-FeedbackArcSet(G)

Algorithm 3 DP-FeedbackArcSet(G)

Input: A directed graph $G = (V, A)$
Output: The size of a smallest possible set $F \subseteq A$ such that $G - F$ is acyclic

```
1: for all $v \in V$ do
2:     $OPT[\{v\}] \leftarrow 0$;
3: end for
4: for $i \leftarrow 2$ to $n$ do
5:     for all $S \subseteq V$ with $|S| = i$ do
6:         $OPT[S] \leftarrow \min_{v \in S}\{OPT[S - \{v\}] + c(v, S - \{v\})\}$;
7:     end for
8: end for
   return $OPT[V]$
```

Time Complexity of DP-FeedbackArcSet(G)

- The recipe to analyzing the running time of any dynamic programming algorithm is simple.
- We first count the number of sub-problems.
- Then we tally up the work done per sub-problem.

Time Complexity of DP-FeedbackArcSet(G)

- The recipe to analyzing the running time of any dynamic programming algorithm is simple.
- We first count the number of sub-problems.
- Then we tally up the work done per sub-problem.

Time Complexity of DP-FEEDBACKARCSET(G)

- The recipe to analyzing the running time of any dynamic programming algorithm is simple.
- We first count the number of sub-problems.
- Then we tally up the work done per sub-problem.

Time Complexity of DP-FeedbackArcSet(G)

- The number of sub-problems of size k is $\binom{n}{k}$.
- Given the adjacency matrix of the graph, a sub-problem of size k can be solved in $O(k^2)$ time.
- Therefore, the running time of our algorithm is:

$$\begin{aligned} & O\left(\sum_{k=0}^{n} k^2 \binom{n}{k}\right) \\ = {} & O\left(\sum_{k=0}^{n}\left(n\binom{n-1}{k-1} + n(n-1)\binom{n-2}{k-2}\right)\right) \\ = {} & O(n^2 2^n) = \mathcal{O}^*(2^n). \end{aligned}$$

Time Complexity of DP-FeedbackArcSet(G)

- The number of sub-problems of size k is $\binom{n}{k}$.
- Given the adjacency matrix of the graph, a sub-problem of size k can be solved in $O(k^2)$ time.
- Therefore, the running time of our algorithm is:

$$\begin{aligned} &O\left(\sum_{k=0}^{n} k^2 \binom{n}{k}\right) \\ &= O\left(\sum_{k=0}^{n} \left(n\binom{n-1}{k-1} + n(n-1)\binom{n-2}{k-2}\right)\right) \\ &= O(n^2 2^n) = \mathcal{O}^*(2^n). \end{aligned}$$

Time Complexity of DP-FeedbackArcSet(G)

- The number of sub-problems of size k is $\binom{n}{k}$.
- Given the adjacency matrix of the graph, a sub-problem of size k can be solved in $O(k^2)$ time.
- Therefore, the running time of our algorithm is:

$$\begin{aligned} & O\left(\sum_{k=0}^{n} k^2 \binom{n}{k}\right) \\ = & O\left(\sum_{k=0}^{n} \left(n\binom{n-1}{k-1} + n(n-1)\binom{n-2}{k-2}\right)\right) \\ = & O(n^2 2^n) = \mathcal{O}^*(2^n). \end{aligned}$$

Time Complexity of DP-FeedbackArcSet(G)

Theorem

DP-FeedbackArcSet(G) runs in $\mathcal{O}^*(2^n)$ time.

Analyzing the Space Complexity

- This is a significant improvement!
- This algorithm has a downside, however.
- The OPT table has an entry for each subset of V.
- Therefore, the space complexity of DP-FeedbackArcSet(G) is $\Omega(2^n)$.

Analyzing the Space Complexity

- This is a significant improvement!
- This algorithm has a downside, however.
- The OPT table has an entry for each subset of V.
- Therefore, the space complexity of DP-FeedbackArcSet(G) is $\Omega(2^n)$.

Analyzing the Space Complexity

- This is a significant improvement!
- This algorithm has a downside, however.
- The OPT table has an entry for each subset of V.
- Therefore, the space complexity of DP-FEEDBACKARCSET(G) is $\Omega(2^n)$.

Analyzing the Space Complexity

- This is a significant improvement!
- This algorithm has a downside, however.
- The OPT table has an entry for each subset of V.
- Therefore, the space complexity of DP-FeedbackArcSet(G) is $\Omega(2^n)$.

Analyzing the Space Complexity

- Ideally, we want our algorithms to use only polynomial space.
- So, next we will see an algorithm that uses only polynomial space.
- This reduction in space complexity is not free, however.
- We are going to have to contend with a larger running time in exchange for it.

Analyzing the Space Complexity

- Ideally, we want our algorithms to use only polynomial space.
- So, next we will see an algorithm that uses only polynomial space.
- This reduction in space complexity is not free, however.
- We are going to have to contend with a larger running time in exchange for it.

Analyzing the Space Complexity

- Ideally, we want our algorithms to use only polynomial space.
- So, next we will see an algorithm that uses only polynomial space.
- This reduction in space complexity is not free, however.
- We are going to have to contend with a larger running time in exchange for it.

Analyzing the Space Complexity

- Ideally, we want our algorithms to use only polynomial space.
- So, next we will see an algorithm that uses only polynomial space.
- This reduction in space complexity is not free, however.
- We are going to have to contend with a larger running time in exchange for it.

Trading Time for Space: Divide & Conquer!

- Now we will attempt to solve FEEDBACK ARC SET using only polynomial space while still having an *acceptable* running time.
- The idea is to use another very versatile algorithm design paradigm: **Divide & Conquer**.

Trading Time for Space: Divide & Conquer!

- Now we will attempt to solve FEEDBACK ARC SET using only polynomial space while still having an *acceptable* running time.
- The idea is to use another very versatile algorithm design paradigm: **Divide & Conquer**.

Setting Up Divide & Conquer Program

- For a set $S \subseteq V$, let $OPT(S)$ be the number of backward arcs in an optimal permutation of the vertices in S.
- To use the $OPT(S)$ values in a divide-conquer style algorithm, we have to set up a recurrence relation.
- The idea is to condition on the first half of the vertices in an optimal permutation of S.

Setting Up Divide & Conquer Program

- For a set $S \subseteq V$, let $OPT(S)$ be the number of backward arcs in an optimal permutation of the vertices in S.
- To use the $OPT(S)$ values in a divide-conquer style algorithm, we have to set up a recurrence relation.
- The idea is to condition on the first half of the vertices in an optimal permutation of S.

Setting Up Divide & Conquer Program

- For a set $S \subseteq V$, let $OPT(S)$ be the number of backward arcs in an optimal permutation of the vertices in S.
- To use the $OPT(S)$ values in a divide-conquer style algorithm, we have to set up a recurrence relation.
- The idea is to condition on the first half of the vertices in an optimal permutation of S.

The Recurrence!

The Recurrence

$$OPT(S) = \min_{\substack{S' \subseteq S \\ |S'| = \left\lceil \frac{|S|}{2} \right\rceil}} \{OPT(S') + OPT(S - S') + c(S - S', S')\}$$

where $c(S - S', S')$ is the number of arcs going from $S - S'$ to S'

The Recurrence!

The Recurrence

$$OPT(S) = \min_{\substack{S' \subseteq S \\ |S'| = \left\lceil \frac{|S|}{2} \right\rceil}} \{OPT(S') + OPT(S - S') + c(S - S', S')\}$$

where $c(S - S', S')$ is the number of arcs going from $S - S'$ to S'

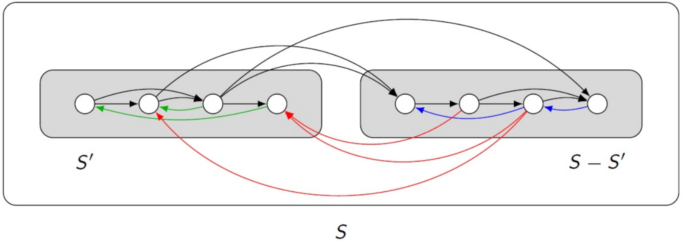


Figure 2: Number of green arcs $= OPT(S')$, Number of blue arcs $= OPT(S - S')$, Number of red arcs $= c(S - S', S')$

Divide & Conquer for Feedback Arc Set

- The size of a minimum feedback arc set is $OPT(V)$.
- We can now use the recurrence from the previous slide to design a recursive algorithm for the Feedback Arc Set problem.

D&C-FeedbackArcSet(G)

Algorithm 4 D&C-FeedbackArcSet(G)

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

Output: The size of a smallest possible set $F \subseteq A$ such that $G - F$ is acyclic

```
1: function OPT($S$)
2:     if $|S| = 1$ then
3:         return 0
4:     end if
5:     return $\min_{S' \subseteq S, |S'| = \lceil \frac{|S|}{2} \rceil} \{OPT(S') + OPT(S - S') + c(S - S', S')\}$;
6: end function
   return $OPT(V)$
```

Analyzing the Space Complexity

- This algorithm requires only polynomial space.
- This is because on each recursion level, we use only polynomial space and the depth of the recursion tree is $\log(n)$.

Analyzing the Time Complexity

- The running time analysis is slightly trickier.

The Recurrence: Recap

$$OPT(S) = \min_{\substack{S' \subseteq S \\ |S'| = \left\lceil \frac{|S|}{2} \right\rceil}} \{OPT(S') + OPT(S - S') + c(S - S', S')\}$$

- For a fixed subset S of size k, the number of subsets S' of S that we can try is bounded above by 2^k.
- After fixing such an S', we must then compute $c(S - S', S')$. Given the adjacency matrix of the graph, this takes $O(k^2)$ time.
- If $T(n)$ is the running time on a graph with n vertices, then:

The Running Time

$$T(n) \leq 2^n \left(T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn^2\right)$$

Analyzing the Time Complexity

- The running time analysis is slightly trickier.

The Recurrence: Recap

$$OPT(S) = \min_{\substack{S' \subseteq S \\ |S'| = \left\lceil \frac{|S|}{2} \right\rceil}} \{OPT(S') + OPT(S - S') + c(S - S', S')\}$$

- For a fixed subset S of size k, the number of subsets S' of S that we can try is bounded above by 2^k.
- After fixing such an S', we must then compute $c(S - S', S')$. Given the adjacency matrix of the graph, this takes $O(k^2)$ time.
- If $T(n)$ is the running time on a graph with n vertices, then:

The Running Time

$$T(n) \leq 2^n \left(T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn^2\right)$$

Analyzing the Time Complexity

- The running time analysis is slightly trickier.

The Recurrence: Recap

$$OPT(S) = \min_{\substack{S' \subseteq S \\ |S'| = \left\lceil \frac{|S|}{2} \right\rceil}} \{OPT(S') + OPT(S - S') + c(S - S', S')\}$$

- For a fixed subset S of size k, the number of subsets S' of S that we can try is bounded above by 2^k.
- After fixing such an S', we must then compute $c(S - S', S')$. Given the adjacency matrix of the graph, this takes $O(k^2)$ time.
- If $T(n)$ is the running time on a graph with n vertices, then:

The Running Time

$$T(n) \leq 2^n \left(T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn^2\right)$$

Analyzing the Time Complexity

- The running time analysis is slightly trickier.

The Recurrence: Recap

$$OPT(S) = \min_{\substack{S' \subseteq S \\ |S'| = \left\lceil \frac{|S|}{2} \right\rceil}} \{OPT(S') + OPT(S - S') + c(S - S', S')\}$$

- For a fixed subset S of size k, the number of subsets S' of S that we can try is bounded above by 2^k.
- After fixing such an S', we must then compute $c(S - S', S')$. Given the adjacency matrix of the graph, this takes $O(k^2)$ time.
- If $T(n)$ is the running time on a graph with n vertices, then:

The Running Time

$$T(n) \leq 2^n \left(T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn^2\right)$$

Analyzing the Time Complexity

- The running time analysis is slightly trickier.

The Recurrence: Recap

$$OPT(S) = \min_{\substack{S' \subseteq S \\ |S'| = \left\lceil \frac{|S|}{2} \right\rceil}} \{OPT(S') + OPT(S - S') + c(S - S', S')\}$$

- For a fixed subset S of size k, the number of subsets S' of S that we can try is bounded above by 2^k.
- After fixing such an S', we must then compute $c(S - S', S')$. Given the adjacency matrix of the graph, this takes $O(k^2)$ time.
- If $T(n)$ is the running time on a graph with n vertices, then:

The Running Time

$$T(n) \leq 2^n \left(T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn^2\right)$$

Analyzing the Time Complexity

- The running time analysis is slightly trickier.

The Recurrence: Recap

$$OPT(S) = \min_{\substack{S' \subseteq S \\ |S'| = \left\lceil \frac{|S|}{2} \right\rceil}} \{OPT(S') + OPT(S - S') + c(S - S', S')\}$$

- For a fixed subset S of size k, the number of subsets S' of S that we can try is bounded above by 2^k.
- After fixing such an S', we must then compute $c(S - S', S')$. Given the adjacency matrix of the graph, this takes $O(k^2)$ time.
- If $T(n)$ is the running time on a graph with n vertices, then:

The Running Time

$$T(n) \leq 2^n \left(T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn^2\right)$$

Analyzing the Time Complexity

The Running Time

$$T(n) \leq 2^n \left(T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn^2\right)$$

Analyzing the Time Complexity

The Running Time

$$\begin{aligned} T(n) &\leq 2^n \left(T\left(\left\lceil \frac{n}{2} \right\rceil \right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor \right) + cn^2 \right) \\ &\approx 2^n \cdot 2T\left(\frac{n}{2}\right) + 2^n cn^2 \end{aligned}$$

Approximating *ceiling* and *floor* to *exact* value.

Analyzing the Time Complexity

The Running Time

$$\begin{aligned}
T(n) &\leq 2^n \left(T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn^2\right) \\
&\approx 2^n \cdot 2T\left(\frac{n}{2}\right) + 2^n cn^2 \\
&\approx 2^{n+\frac{n}{2}+\cdots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)+1} T(1) + 2^n cn^2 + 2^{n+\frac{n}{2}} \cdot 2c\left(\frac{n}{2}\right)^2 + \\
&\quad \cdots + 2^{n+\frac{n}{2}+\cdots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)} c\left(\frac{n}{2^{\log(n)}}\right)^2
\end{aligned}$$

Expanding and using the fact that $\log(n)$ substitutions are needed to reach $T(1)$.

Analyzing the Time Complexity

The Running Time

$$\begin{aligned}
T(n) &\leq 2^n \left(T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn^2\right) \\
&\approx 2^n \cdot 2T\left(\frac{n}{2}\right) + 2^n cn^2 \\
&\approx 2^{n+\frac{n}{2}+\cdots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)+1} T(1) + 2^n cn^2 + 2^{n+\frac{n}{2}} \cdot 2c\left(\frac{n}{2}\right)^2 + \\
&\quad \cdots + 2^{n+\frac{n}{2}+\cdots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)} c \left(\frac{n}{2^{\log(n)}}\right)^2 \\
&< 2^{n+\frac{n}{2}+\cdots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)} \left(2T(1) + cn^2(\log(n)+1)\right)
\end{aligned}$$

There are $\log(n) + 1$ terms containing cn^2 and $2^{n+\frac{n}{2}+\cdots+\frac{n}{2^{\log(n)}}} \cdot cn^2$ is greater than any of those.
(Note: We are doing a bit loose calculation, as a tighter analysis will not result in a better $\mathcal{O}^*$ complexity.)

Analyzing the Time Complexity

The Running Time

$$\begin{aligned}
T(n) &\leq 2^n \left(T\left(\left\lceil\frac{n}{2}\right\rceil\right) + T\left(\left\lfloor\frac{n}{2}\right\rfloor\right) + cn^2\right) \\
&\approx 2^n \cdot 2T\left(\frac{n}{2}\right) + 2^n cn^2 \\
&\approx 2^{n+\frac{n}{2}+\cdots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)+1} T(1) + 2^n cn^2 + 2^{n+\frac{n}{2}} \cdot 2c\left(\frac{n}{2}\right)^2 + \\
&\quad \cdots + 2^{n+\frac{n}{2}+\cdots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)} c\left(\frac{n}{2^{\log(n)}}\right)^2 \\
&< 2^{n+\frac{n}{2}+\cdots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)} \left(2T(1) + cn^2(\log(n)+1)\right) \\
&= 2^{n\left(1+\frac{1}{2}+\frac{1}{4}+\cdots\right)} n \left(2T(1) + cn^2(\log(n)+1)\right)
\end{aligned}$$

Replacing the finite sum with an infinite sum.

Analyzing the Time Complexity

The Running Time

$$
\begin{aligned}
T(n) &\le 2^n \left(T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn^2\right) \\
&\approx 2^n \cdot 2T\left(\frac{n}{2}\right) + 2^n cn^2 \\
&\approx 2^{n+\frac{n}{2}+\cdots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)+1} T(1) + 2^n cn^2 + 2^{n+\frac{n}{2}} \cdot 2c\left(\frac{n}{2}\right)^2 + \\
&\quad \cdots + 2^{n+\frac{n}{2}+\cdots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)} c\left(\frac{n}{2^{\log(n)}}\right)^2 \\
&< 2^{n+\frac{n}{2}+\cdots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)} \left(2T(1) + cn^2(\log(n)+1)\right) \\
&= 2^{n\left(1+\frac{1}{2}+\frac{1}{4}+\cdots\right)} n \left(2T(1) + cn^2(\log(n)+1)\right) \\
&= O(4^n n^3 \log(n))
\end{aligned}
$$

Analyzing the Time Complexity

The Running Time

$$\begin{aligned}
T(n) &\leq 2^n \left(T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn^2\right) \\
&\approx 2^n \cdot 2T\left(\frac{n}{2}\right) + 2^n cn^2 \\
&\approx 2^{n+\frac{n}{2}+\dots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)+1} T(1) + 2^n cn^2 + 2^{n+\frac{n}{2}} \cdot 2c\left(\frac{n}{2}\right)^2 + \\
&\quad \dots + 2^{n+\frac{n}{2}+\dots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)} c \left(\frac{n}{2^{\log(n)}}\right)^2 \\
&< 2^{n+\frac{n}{2}+\dots+\frac{n}{2^{\log(n)}}} \cdot 2^{\log(n)} \left(2T(1) + cn^2(\log(n)+1)\right) \\
&= 2^{n\left(1+\frac{1}{2}+\frac{1}{4}+\cdots\right)} n \left(2T(1) + cn^2(\log(n)+1)\right) \\
&= O(4^n n^3 \log(n)) \\
&= \mathcal{O}^*(4^n)
\end{aligned}$$

Analyzing the Running Time

Theorem

D&C-FeedbackArcSet(G) runs in $\mathcal{O}^*(4^n)$ time.

Product of Time and Space

- The time and space complexities of our divide-and-conquer algorithm are respectively $\mathcal{O}^*(4^n)$ and $\mathcal{O}^*(1)$.
- The time and space complexities of our dynamic programming algorithm are respectively $\mathcal{O}^*(2^n)$ and $\mathcal{O}^*(2^n)$.
- In both cases, $TIME \times SPACE = \mathcal{O}^*(4^n)$.
- The dynamic programming algorithm saves a lot of time in exchange for space.
- The divide and conquer algorithm goes to the other extreme and uses only a polynomial amount of space but does worse in the time complexity department.

Product of Time and Space

- The time and space complexities of our divide-and-conquer algorithm are respectively $\mathcal{O}^*(4^n)$ and $\mathcal{O}^*(1)$.
- The time and space complexities of our dynamic programming algorithm are respectively $\mathcal{O}^*(2^n)$ and $\mathcal{O}^*(2^n)$.
- In both cases, $TIME \times SPACE = \mathcal{O}^*(4^n)$.
- The dynamic programming algorithm saves a lot of time in exchange for space.
- The divide and conquer algorithm goes to the other extreme and uses only a polynomial amount of space but does worse in the time complexity department.

Product of Time and Space

- The time and space complexities of our divide-and-conquer algorithm are respectively $\mathcal{O}^*(4^n)$ and $\mathcal{O}^*(1)$.
- The time and space complexities of our dynamic programming algorithm are respectively $\mathcal{O}^*(2^n)$ and $\mathcal{O}^*(2^n)$.
- In both cases, $TIME \times SPACE = \mathcal{O}^*(4^n)$.
- The dynamic programming algorithm saves a lot of time in exchange for space.
- The divide and conquer algorithm goes to the other extreme and uses only a polynomial amount of space but does worse in the time complexity department.

Product of Time and Space

- The time and space complexities of our divide-and-conquer algorithm are respectively $\mathcal{O}^*(4^n)$ and $\mathcal{O}^*(1)$.
- The time and space complexities of our dynamic programming algorithm are respectively $\mathcal{O}^*(2^n)$ and $\mathcal{O}^*(2^n)$.
- In both cases, $TIME \times SPACE = \mathcal{O}^*(4^n)$.
- The dynamic programming algorithm saves a lot of time in exchange for space.
- The divide and conquer algorithm goes to the other extreme and uses only a polynomial amount of space but does worse in the time complexity department.

Product of Time and Space

- The time and space complexities of our divide-and-conquer algorithm are respectively $\mathcal{O}^*(4^n)$ and $\mathcal{O}^*(1)$.
- The time and space complexities of our dynamic programming algorithm are respectively $\mathcal{O}^*(2^n)$ and $\mathcal{O}^*(2^n)$.
- In both cases, $TIME \times SPACE = \mathcal{O}^*(4^n)$.
- The dynamic programming algorithm saves a lot of time in exchange for space.
- The divide and conquer algorithm goes to the other extreme and uses only a polynomial amount of space but does worse in the time complexity department.

Product of Time and Space

- It is possible to try a hybrid of both dynamic programming and divide-and-conquer and get a balance of both space and time.
- The idea is to start with divide-and-conquer first, stop as soon as the sub-problem sizes drop below a certain amount and use dynamic programming after that.
- However, $TIME \times SPACE$ is still $\mathcal{O}^*(4^n)$ in this hybrid approach.
- But, using an idea by Koivisto and Parviainen (2010) [8], it is possible to get $TIME \times SPACE = \mathcal{O}^*(3.93^n)$.

Product of Time and Space

- It is possible to try a hybrid of both dynamic programming and divide-and-conquer and get a balance of both space and time.
- The idea is to start with divide-and-conquer first, stop as soon as the sub-problem sizes drop below a certain amount and use dynamic programming after that.
- However, $TIME \times SPACE$ is still $\mathcal{O}^*(4^n)$ in this hybrid approach.
- But, using an idea by Koivisto and Parviainen (2010) [8], it is possible to get $TIME \times SPACE = \mathcal{O}^*(3.93^n)$.

Product of Time and Space

- It is possible to try a hybrid of both dynamic programming and divide-and-conquer and get a balance of both space and time.
- The idea is to start with divide-and-conquer first, stop as soon as the sub-problem sizes drop below a certain amount and use dynamic programming after that.
- However, $TIME \times SPACE$ is still $\mathcal{O}^*(4^n)$ in this hybrid approach.
- But, using an idea by Koivisto and Parviainen (2010) [8], it is possible to get $TIME \times SPACE = \mathcal{O}^*(3.93^n)$.

Product of Time and Space

- It is possible to try a hybrid of both dynamic programming and divide-and-conquer and get a balance of both space and time.
- The idea is to start with divide-and-conquer first, stop as soon as the sub-problem sizes drop below a certain amount and use dynamic programming after that.
- However, $TIME \times SPACE$ is still $\mathcal{O}^*(4^n)$ in this hybrid approach.
- But, using an idea by Koivisto and Parviainen (2010) [8], it is possible to get $TIME \times SPACE = \mathcal{O}^*(3.93^n)$.

Outline

Parameterized Complexity

- The decision version of FEEDBACK ARC SET (when parameterized by the size of the feedback arc set desired) is **Fixed Parameter Tractable** (FPT).
- The running time of the best known such algorithm is $O\left((k+1)! \cdot 4^k \cdot k^3 \cdot n(n+m)\right)$ [3] (2008) (where k is the size of the feedback arc set being asked for).
- This algorithm uses a technique known as **Iterative Compression**.
- Due to time constraints, we are hoping to add more details about this technique in the Final Report.

Parameterized Complexity

- The decision version of FEEDBACK ARC SET (when parameterized by the size of the feedback arc set desired) is **Fixed Parameter Tractable** (FPT).
- The running time of the best known such algorithm is $O\left((k+1)! \cdot 4^k \cdot k^3 \cdot n(n+m)\right)$ [3] (2008) (where k is the size of the feedback arc set being asked for).
- This algorithm uses a technique known as **Iterative Compression**.
- Due to time constraints, we are hoping to add more details about this technique in the Final Report.

Parameterized Complexity

- The decision version of FEEDBACK ARC SET (when parameterized by the size of the feedback arc set desired) is **Fixed Parameter Tractable** (FPT).
- The running time of the best known such algorithm is $O\left((k+1)! \cdot 4^k \cdot k^3 \cdot n(n+m)\right)$ [3] (2008) (where k is the size of the feedback arc set being asked for).
- This algorithm uses a technique known as **Iterative Compression**.
- Due to time constraints, we are hoping to add more details about this technique in the Final Report.

Parameterized Complexity

- The decision version of FEEDBACK ARC SET (when parameterized by the size of the feedback arc set desired) is **Fixed Parameter Tractable** (FPT).
- The running time of the best known such algorithm is $O\left((k+1)! \cdot 4^k \cdot k^3 \cdot n(n+m)\right)$ [3] (2008) (where k is the size of the feedback arc set being asked for).
- This algorithm uses a technique known as **Iterative Compression**.
- Due to time constraints, we are hoping to add more details about this technique in the Final Report.

Outline

Approximation Algorithms

- From this point, we will insist on having *only* polynomial-time algorithms but either at the cost of correctness or by restricting input instances.
- In other words, our algorithms will not always give us optimal solutions but they will run very fast.
- However, our algorithms *will have* some sort of guarantee about the quality of the solutions they produce.
- Let's enter the world of *Approximation Algorithms*!.

Approximation Algorithms

- From this point, we will insist on having *only* polynomial-time algorithms but either at the cost of correctness or by restricting input instances.
- In other words, our algorithms will not always give us optimal solutions but they will run very fast.
- However, our algorithms *will have* some sort of guarantee about the quality of the solutions they produce.
- Let's enter the world of *Approximation Algorithms*!.

Approximation Algorithms

- From this point, we will insist on having *only* polynomial-time algorithms but either at the cost of correctness or by restricting input instances.
- In other words, our algorithms will not always give us optimal solutions but they will run very fast.
- However, our algorithms *will have* some sort of guarantee about the quality of the solutions they produce.
- Let's enter the world of *Approximation Algorithms*!.

Approximation Algorithms

- From this point, we will insist on having *only* polynomial-time algorithms but either at the cost of correctness or by restricting input instances.
- In other words, our algorithms will not always give us optimal solutions but they will run very fast.
- However, our algorithms *will have* some sort of guarantee about the quality of the solutions they produce.
- Let's enter the world of *Approximation Algorithms*!.

Different Approximation Classes

- Let us first talk about different approximation classes & more importantly, why it is hard to find good approximate solutions to certain problems.
- In approximation algorithm design, the holy-grail is something known as *polynomial time approximation scheme* (PTAS).
- A PTAS for a problem is a family of polynomial-time algorithms with approximation ratios arbitrarily close to 1.

Definition (Polynomial-Time Approximation Scheme)

A problem is said to have a *polynomial-time approximation scheme* or PTAS if for any $\epsilon > 0$, there exists a $(1 + \epsilon)$-approximation algorithm (which runs in time polynomial in the input size) for that problem.

Different Approximation Classes

- Let us first talk about different approximation classes & more importantly, why it is hard to find good approximate solutions to certain problems.
- In approximation algorithm design, the holy-grail is something known as *polynomial time approximation scheme* (PTAS).
- A PTAS for a problem is a family of polynomial-time algorithms with approximation ratios arbitrarily close to 1.

Definition (Polynomial-Time Approximation Scheme)

A problem is said to have a *polynomial-time approximation scheme* or PTAS if for any $\epsilon > 0$, there exists a $(1 + \epsilon)$-approximation algorithm (which runs in time polynomial in the input size) for that problem.

Different Approximation Classes

- Let us first talk about different approximation classes & more importantly, why it is hard to find good approximate solutions to certain problems.
- In approximation algorithm design, the holy-grail is something known as *polynomial time approximation scheme* (PTAS).
- A PTAS for a problem is a family of polynomial-time algorithms with approximation ratios arbitrarily close to 1.

Definition (Polynomial-Time Approximation Scheme)

A problem is said to have a *polynomial-time approximation scheme* or PTAS if for any $\epsilon > 0$, there exists a $(1 + \epsilon)$-approximation algorithm (which runs in time polynomial in the input size) for that problem.

Different Approximation Classes

- Let us first talk about different approximation classes & more importantly, why it is hard to find good approximate solutions to certain problems.
- In approximation algorithm design, the holy-grail is something known as *polynomial time approximation scheme* (PTAS).
- A PTAS for a problem is a family of polynomial-time algorithms with approximation ratios arbitrarily close to 1.

Definition (Polynomial-Time Approximation Scheme)

A problem is said to have a *polynomial-time approximation scheme* or PTAS if for any $\epsilon > 0$, there exists a $(1 + \epsilon)$-approximation algorithm (which runs in time polynomial in the input size) for that problem.

Different Approximation Classes

- A particular special case is when the running time of the algorithm is a polynomial in both the input size and $\frac{1}{\epsilon}$.
- When that happens, the problem is said to have a *fully polynomial time approximation scheme* (FPTAS).

Definition (Fully Polynomial-Time Approximation Scheme)

A problem is said to have a *fully polynomial-time approximation scheme* or FPTAS if it has a PTAS with running time that is polynomial in both the input size and $\frac{1}{\epsilon}$.

Different Approximation Classes

- A particular special case is when the running time of the algorithm is a polynomial in both the input size and $\frac{1}{\epsilon}$.
- When that happens, the problem is said to have a *fully polynomial time approximation scheme* (FPTAS).

Definition (Fully Polynomial-Time Approximation Scheme)

A problem is said to have a *fully polynomial-time approximation scheme* or FPTAS if it has a PTAS with running time that is polynomial in both the input size and $\frac{1}{\epsilon}$.

Different Approximation Classes

- A particular special case is when the running time of the algorithm is a polynomial in both the input size and $\frac{1}{\epsilon}$.
- When that happens, the problem is said to have a *fully polynomial time approximation scheme* (FPTAS).

Definition (Fully Polynomial-Time Approximation Scheme)

A problem is said to have a *fully polynomial-time approximation scheme* or FPTAS if it has a PTAS with running time that is polynomial in both the input size and $\frac{1}{\epsilon}$.

No FPTAS!

- The idea of an FPTAS seems too good to be true! Very few problems have an FPTAS.
- So, it is not surprising at all that FEEDBACK ARC SET does *not* have one either.

Theorem

FEEDBACK ARC SET does not have an FPTAS unless $P{=}NP$.

No FPTAS!

- The idea of an FPTAS seems too good to be true! Very few problems have an FPTAS.
- So, it is not surprising at all that Feedback Arc Set does *not* have one either.

Theorem

Feedback Arc Set does not have an FPTAS unless $P{=}NP$.

No FPTAS!

- The idea of an FPTAS seems too good to be true! Very few problems have an FPTAS.
- So, it is not surprising at all that Feedback Arc Set does *not* have one either.

Theorem

Feedback Arc Set does not have an FPTAS unless $P{=}NP$.

No FPTAS!

Proof

By contradiction. Assume that FEEDBACK ARC SET does have an FPTAS.

Set $\epsilon = \frac{1}{n^2}$. Note that the definition of an FPTAS ensures that our algorithm runs in polynomial-time even after setting ϵ so low.

No FPTAS!

Proof (continued)

Now we have

$$\frac{ALG_{\epsilon}(G)}{OPT(G)} \leq 1 + \epsilon$$

No FPTAS!

Proof (continued)

Now we have

$$\frac{ALG_\epsilon(G)}{OPT(G)} \leq 1 + \frac{1}{n^2}$$

No FPTAS!

Proof (continued)

Now we have

$$\frac{ALG_\epsilon(G)}{OPT(G)} \leq \frac{n^2+1}{n^2}$$

No FPTAS!

Proof (continued)

Now we have

$$\frac{ALG_\epsilon(G)}{OPT(G)} \leq \frac{n^2+1}{n^2}$$

$$\therefore OPT(G) \geq \frac{n^2}{n^2+1} ALG_\epsilon(G)$$

No FPTAS!

Proof (continued)

Now we have

$$\begin{aligned}\frac{ALG_\epsilon(G)}{OPT(G)} &\leq \frac{n^2+1}{n^2} \\ \therefore OPT(G) &\geq \frac{n^2}{n^2+1} ALG_\epsilon(G) \\ &= ALG_\epsilon(G) - \frac{ALG_\epsilon(G)}{n^2+1}\end{aligned}$$

No FPTAS!

Proof (continued)

Now we have

$$\begin{aligned}
\frac{ALG_\epsilon(G)}{OPT(G)} &\leq \frac{n^2+1}{n^2} \\
\therefore OPT(G) &\geq \frac{n^2}{n^2+1} ALG_\epsilon(G) \\
&= ALG_\epsilon(G) - \frac{ALG_\epsilon(G)}{n^2+1} \\
&> ALG_\epsilon(G) - 1
\end{aligned}$$

No FPTAS!

Proof (continued)

Now we have

$$\begin{aligned}
\frac{ALG_\epsilon(G)}{OPT(G)} &\leq \frac{n^2+1}{n^2} \\
\therefore OPT(G) &\geq \frac{n^2}{n^2+1} ALG_\epsilon(G) \\
&= ALG_\epsilon(G) - \frac{ALG_\epsilon(G)}{n^2+1} \\
&> ALG_\epsilon(G) - 1 \\
\therefore ALG_\epsilon(G) &< OPT(G) + 1
\end{aligned}$$

No FPTAS!

Proof (continued)

The final conclusion from all this algebra is

$$ALG_\epsilon(G) < OPT(G) + 1$$

Since in our problem, feasible solutions always have integer objective function values, it follows that our algorithm does at least as well as the optimum. A contradiction unless $P{=}NP$!

No PTAS as well!

- So an FPTAS for Feedback Arc Set is out of the picture.
- But what about a plain-old vanilla PTAS?
- Again, there is bad news in the form of the following theorem:

Theorem

Unless $P = NP$, Feedback Arc Set does not have a PTAS.

No PTAS as well!

- So an FPTAS for FEEDBACK ARC SET is out of the picture.
- But what about a plain-old vanilla PTAS?
- Again, there is bad news in the form of the following theorem:

Theorem

Unless $P = NP$, FEEDBACK ARC SET does not have a PTAS.

No PTAS as well!

- So an FPTAS for Feedback Arc Set is out of the picture.
- But what about a plain-old vanilla PTAS?
- Again, there is bad news in the form of the following theorem:

Theorem

Unless $P = NP$, Feedback Arc Set does not have a PTAS.

No PTAS as well!

- So an FPTAS for FEEDBACK ARC SET is out of the picture.
- But what about a plain-old vanilla PTAS?
- Again, there is bad news in the form of the following theorem:

Theorem

Unless $P = NP$, FEEDBACK ARC SET does not have a PTAS.

Inapproximability and Open Question

- But, that is not even the worst news!
- It is still not known, for general directed graphs, whether FEEDBACK ARC SET has a constant-factor approximation algorithm or not.
- The best known approximation algorithm has an approximation ratio of $O(\log(n)\log(\log(n)))$ (Sudan *et al.*, 1998) [4]

Open Question

Does FEEDBACK ARC SET have a constant factor approximation algorithm or equibalently, is FEEDBACK ARC SET in *APX* ?

Inapproximability and Open Question

- But, that is not even the worst news!
- It is still not known, for general directed graphs, whether FEEDBACK ARC SET has a constant-factor approximation algorithm or not.
- The best known approximation algorithm has an approximation ratio of $O(\log(n)\log(\log(n)))$ (Sudan *et al.*, 1998) [4]

Open Question

Does FEEDBACK ARC SET have a constant factor approximation algorithm or equibalently, is FEEDBACK ARC SET in *APX* ?

Inapproximability and Open Question

- But, that is not even the worst news!
- It is still not known, for general directed graphs, whether FEEDBACK ARC SET has a constant-factor approximation algorithm or not.
- The best known approximation algorithm has an approximation ratio of $O(\log(n)\log(\log(n)))$ (Sudan *et al.*, 1998) [4]

Open Question

Does FEEDBACK ARC SET have a constant factor approximation algorithm or equibalently, is FEEDBACK ARC SET in APX ?

Inapproximability and Open Question

- But, that is not even the worst news!
- It is still not known, for general directed graphs, whether FEEDBACK ARC SET has a constant-factor approximation algorithm or not.
- The best known approximation algorithm has an approximation ratio of $O(\log(n)\log(\log(n)))$ (Sudan *et al.*, 1998) [4]

Open Question

Does FEEDBACK ARC SET have a constant factor approximation algorithm or equibalently, is FEEDBACK ARC SET in APX ?

Feedback Arc Set on Tournaments

- We now know that, FEEDBACK ARC SET does not have a PTAS unless *P*=*NP*
- However, it does have a PTAS if we restrict our inputs to only tournaments (Mathieu and Schudy, 2009) [10].
- Recall that, a tournament is a directed graph with the property of, given any two vertices u and v in a tournament, exactly one of the arcs (u, v) or (v, u) is in the tournament.

Feedback Arc Set on Tournaments

- We now know that, FEEDBACK ARC SET does not have a PTAS unless P=NP
- However, it does have a PTAS if we restrict our inputs to only tournaments (Mathieu and Schudy, 2009) [10].
- Recall that, a tournament is a directed graph with the property of, given any two vertices u and v in a tournament, exactly one of the arcs (u, v) or (v, u) is in the tournament.

Feedback Arc Set on Tournaments

- We now know that, FEEDBACK ARC SET does not have a PTAS unless $P{=}NP$
- However, it does have a PTAS if we restrict our inputs to only tournaments (Mathieu and Schudy, 2009) [10].
- Recall that, a tournament is a directed graph with the property of, given any two vertices u and v in a tournament, exactly one of the arcs (u, v) or (v, u) is in the tournament.

Feedback Arc Set on Tournaments

- The algorithm given by Mathieu and Schudy is extremely complicated & not suitable for this brief presentation.
- So, instead we will settle for an algorithm that achieves a constant factor approximation ration in expectation, based on an idea by Alion, Charikar and Newman (2008) [1].
- Due to its striking similarities to the randomized quicksort algorithm, the authors refer to it as KwikSort.

Feedback Arc Set on Tournaments

- The algorithm given by Mathieu and Schudy is extremely complicated & not suitable for this brief presentation.
- So, instead we will settle for an algorithm that achieves a constant factor approximation ration in expectation, based on an idea by Alion, Charikar and Newman (2008) [1].
- Due to its striking similarities to the randomized quicksort algorithm, the authors refer to it as KwikSort.

Feedback Arc Set on Tournaments

- The algorithm given by Mathieu and Schudy is extremely complicated & not suitable for this brief presentation.
- So, instead we will settle for an algorithm that achieves a constant factor approximation ration in expectation, based on an idea by Alion, Charikar and Newman (2008) [1].
- Due to its striking similarities to the randomized quicksort algorithm, the authors refer to it as KwikSort.

KwikSort(G)

Algorithm 5 KwikSort(G)

Input: A tournament $G = (V, A)$.
Output: A permutation of the vertices in V.

```
1: if $V = \emptyset$ then
2:     return the empty permutation
3: end if

4: $v \leftarrow$ a vertex chosen uniformly at random from $V$ ▷ pivot selection
5: $V_L \leftarrow \{v_L : v_L \in V\}$ and $(v_L, v) \in A$      ▷ the in-neighbors
6: $V_R \leftarrow \{v_R : v_R \in V\}$ and $(v_R, v) \in A$      ▷ the out-neighbors

7: $G_L = (V_L, A_L) \leftarrow$ the tournament induced by the vertices in $V_L$
8: $G_R = (V_R, A_R) \leftarrow$ the tournament induced by the vertices in $V_R$
   return KwikSort($G_L$), $v$, KwikSort($G_R$)
```

KwikSort(G)

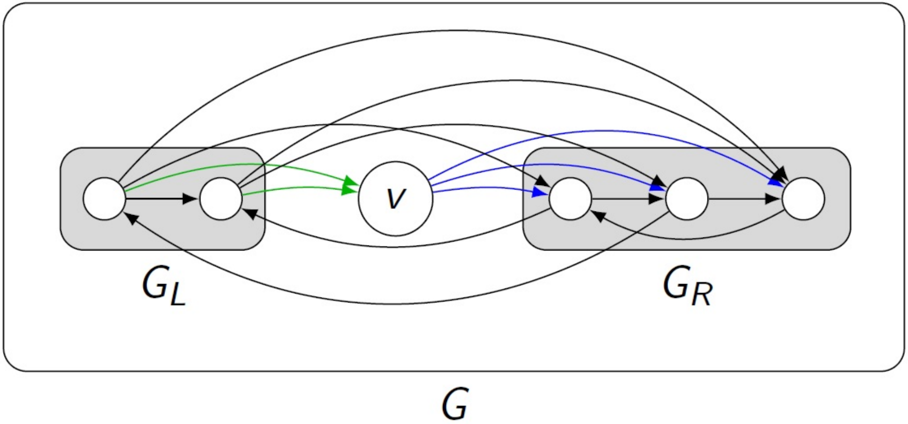


Figure 3: KwikSort visualized. The vertex v is chosen as pivot. The tails of all green arcs constitute the graph G_L. The heads of all the blue arcs constitute the graph G_R.

KwikSort(G)

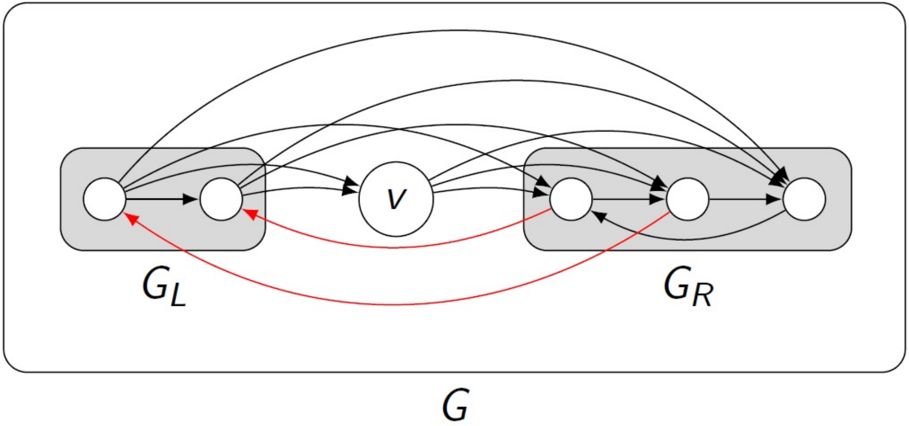


Figure 3: KwikSort visualized. The vertex v is chosen as pivot. The number of red arcs is the cost of choosing v as the pivot.

KwikSort(G) is 3-Approximation

- As simple as KwikSort looks, it has some nice desirable properties.
- It is actually used as a preprocessing step in the PTAS for Feedback Arc Set on tournaments.
- Even on its own, it can produce a feedback arc set with expected size at most three times of the optimal.

Theorem

Let $ALG(\mathrm{G})$ be the number of "backward" arcs generated by running KwikSort on the graph G. If $OPT(\mathrm{G})$ is the size of a minimum feedback arc set of G, then $E[ALG(\mathrm{G}) \leq 3OPT(\mathrm{G})$.

This theorem can be proven using *Linear Programming Duality.*

KwikSort(G) is 3-Approximation

- As simple as KwikSort looks, it has some nice desirable properties.
- It is actually used as a preprocessing step in the PTAS for Feedback Arc Set on tournaments.
- Even on its own, it can produce a feedback arc set with expected size at most three times of the optimal.

Theorem

Let $ALG(\mathrm{G})$ be the number of "backward" arcs generated by running KwikSort on the graph G. If $OPT(\mathrm{G})$ is the size of a minimum feedback arc set of G, then $E[ALG(\mathrm{G}) \leq 3OPT(\mathrm{G})$.

This theorem can be proven using *Linear Programming Duality*.

KwikSort(G) is 3-Approximation

- As simple as KwikSort looks, it has some nice desirable properties.
- It is actually used as a preprocessing step in the PTAS for Feedback Arc Set on tournaments.
- Even on its own, it can produce a feedback arc set with expected size at most three times of the optimal.

Theorem

Let ALG(G) be the number of "backward" arcs generated by running KwikSort on the graph G. If OPT(G) is the size of a minimum feedback arc set of G, then $E[ALG$(G) $\leq 3OPT$(G).

This theorem can be proven using *Linear Programming Duality.*

KwikSort(G) is 3-Approximation

- As simple as KwikSort looks, it has some nice desirable properties.
- It is actually used as a preprocessing step in the PTAS for Feedback Arc Set on tournaments.
- Even on its own, it can produce a feedback arc set with expected size at most three times of the optimal.

Theorem

Let $ALG(\mathrm{G})$ be the number of "backward" arcs generated by running KwikSort on the graph G. If $OPT(\mathrm{G})$ is the size of a minimum feedback arc set of G, then $E[ALG(\mathrm{G}) \leq 3OPT(\mathrm{G})$.

This theorem can be proven using *Linear Programming Duality.*

KwikSort(G) is 3-Approximation

- As simple as KwikSort looks, it has some nice desirable properties.
- It is actually used as a preprocessing step in the PTAS for Feedback Arc Set on tournaments.
- Even on its own, it can produce a feedback arc set with expected size at most three times of the optimal.

Theorem

Let $ALG(\mathrm{G})$ be the number of "backward" arcs generated by running KwikSort on the graph G. If $OPT(\mathrm{G})$ is the size of a minimum feedback arc set of G, then $E[ALG(\mathrm{G}) \leq 3OPT(\mathrm{G})$.

This theorem can be proven using *Linear Programming Duality*.

Outline

Poly-time algorithms for Restricted Instances

- There are not many polynomial time variants of the FEEDBACK ARC SET problem.
- The *undirected* version of FEEDBACK ARC SET, which we can aptly call FEEDBACK EDGE SET, is clearly in P.
- Given an undirected graph, finding a set of edges whose removal leaves the graph acyclic is trivial since one merely needs to compute a spanning *forest* of the graph.
- This can be done using any graph traversal algorithm like **breadth-first** or **depth-first** search.
- It turns out that FEEDBACK ARC SET can be solved in polynomial time if the inputs are restricted to only planar digraphs, Garey (1983) [5].

Poly-time algorithms for Restricted Instances

- There are not many polynomial time variants of the FEEDBACK ARC SET problem.
- The *undirected* version of FEEDBACK ARC SET, which we can aptly call FEEDBACK EDGE SET, is clearly in P.
- Given an undirected graph, finding a set of edges whose removal leaves the graph acyclic is trivial since one merely needs to compute a spanning *forest* of the graph.
- This can be done using any graph traversal algorithm like **breadth-first** or **depth-first** search.
- It turns out that FEEDBACK ARC SET can be solved in polynomial time if the inputs are restricted to only planar digraphs, Garey (1983) [5].

Poly-time algorithms for Restricted Instances

- There are not many polynomial time variants of the FEEDBACK ARC SET problem.
- The *undirected* version of FEEDBACK ARC SET, which we can aptly call FEEDBACK EDGE SET, is clearly in P.
- Given an undirected graph, finding a set of edges whose removal leaves the graph acyclic is trivial since one merely needs to compute a spanning *forest* of the graph.
- This can be done using any graph traversal algorithm like **breadth-first** or **depth-first** search.
- It turns out that FEEDBACK ARC SET can be solved in polynomial time if the inputs are restricted to only planar digraphs, Garey (1983) [5].

Poly-time algorithms for Restricted Instances

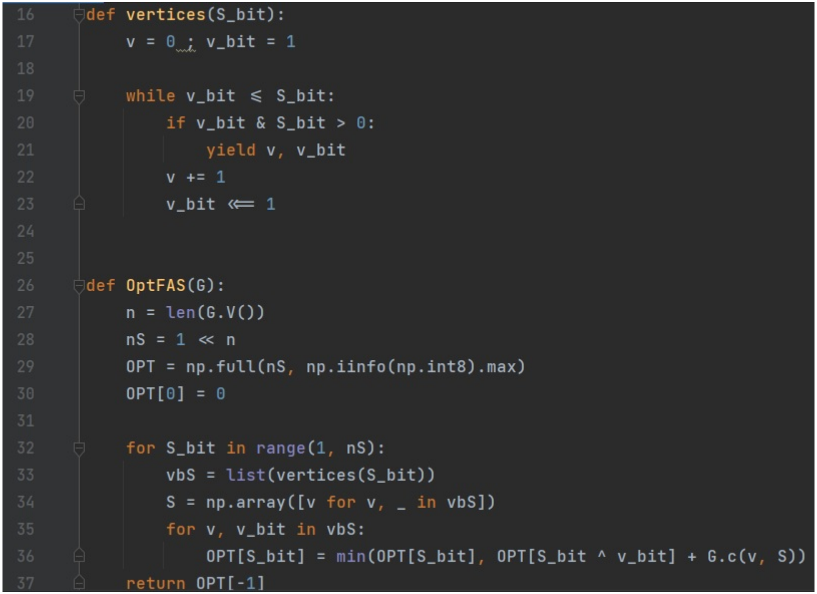
- There are not many polynomial time variants of the FEEDBACK ARC SET problem.
- The *undirected* version of FEEDBACK ARC SET, which we can aptly call FEEDBACK EDGE SET, is clearly in P.
- Given an undirected graph, finding a set of edges whose removal leaves the graph acyclic is trivial since one merely needs to compute a spanning *forest* of the graph.
- This can be done using any graph traversal algorithm like **breadth-first** or **depth-first** search.
- It turns out that FEEDBACK ARC SET can be solved in polynomial time if the inputs are restricted to only planar digraphs, Garey (1983) [5].

Poly-time algorithms for Restricted Instances

- There are not many polynomial time variants of the FEEDBACK ARC SET problem.
- The *undirected* version of FEEDBACK ARC SET, which we can aptly call FEEDBACK EDGE SET, is clearly in P.
- Given an undirected graph, finding a set of edges whose removal leaves the graph acyclic is trivial since one merely needs to compute a spanning *forest* of the graph.
- This can be done using any graph traversal algorithm like **breadth-first** or **depth-first** search.
- It turns out that FEEDBACK ARC SET can be solved in polynomial time if the inputs are restricted to only planar digraphs, Garey (1983) [5].

Outline

Dynamic Programming Implementation



```
def vertices(S_bit):
    v = 0 ; v_bit = 1

    while v_bit <= S_bit:
        if v_bit & S_bit > 0:
            yield v, v_bit
        v += 1
        v_bit <<= 1


def OptFAS(G):
    n = len(G.V())
    nS = 1 << n
    OPT = np.full(nS, np.iinfo(np.int8).max)
    OPT[0] = 0

    for S_bit in range(1, nS):
        vbS = list(vertices(S_bit))
        S = np.array([v for v, _ in vbS])
        for v, v_bit in vbS:
            OPT[S_bit] = min(OPT[S_bit], OPT[S_bit ^ v_bit] + G.c(v, S))
    return OPT[-1]
```

Dynamic Programming for FAS

- "Relatively Fast" exact algorithm and so can be used as a **benchmark** for other approximate algorithms.
- To show that theoretical running time derivations also give accurate estimates in practice.

Dynamic Programming for FAS

- “Relatively Fast” exact algorithm and so can be used as a **benchmark** for other approximate algorithms.
- To show that theoretical running time derivations also give accurate estimates in practice.

Dynamic Programming for FAS

- "Relatively Fast" exact algorithm and so can be used as a **benchmark** for other approximate algorithms.
- To show that theoretical running time derivations also give accurate estimates in practice.

Note: This algorithm has $\Omega(2^n)$ space complexity. A hard limit on the size of the input graphs!

Dynamic Programming for FAS

The following table lists the time taken by the dynamic program on random tournaments of various sizes. Note how the running time *roughly* goes up by a factor of 2 as the input size increases by 1.

# of vertices	time (in seconds)
11	0.00299
12	0.00598
13	0.01296
15	0.0578
17	0.2533
19	1.168
21	4.927
23	29.058
25	112.57

Dynamic Programming for FAS

- The program was run for graphs of size upto 25.
- Larger graphs would exhaust the entire memory.For example, running this algorithm on a 30 vertex graph would use up more than 1 gigabyte of memory.
- The real running time *almost exactly* matches the theoretical estimate!

Dynamic Programming for FAS

- The program was run for graphs of size upto 25.
- Larger graphs would exhaust the entire memory.For example, running this algorithm on a 30 vertex graph would use up more than 1 gigabyte of memory.
- The real running time *almost exactly* matches the theoretical estimate!

Dynamic Programming for FAS

- The program was run for graphs of size upto 25.
- Larger graphs would exhaust the entire memory.For example, running this algorithm on a 30 vertex graph would use up more than 1 gigabyte of memory.
- The real running time *almost exactly* matches the theoretical estimate!

Dynamic Programming for FAS

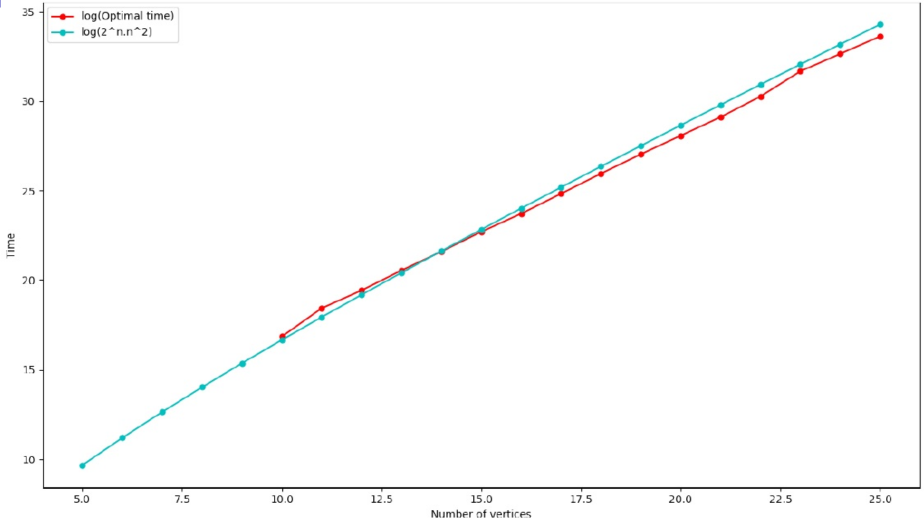


Figure 4: The logarithm of the running time (red) and $\log(2^n n^2)$ (blue). Both these curves have *almost* the same shape. A constant ≈ 26.3045 has been added to red one to make sure these two graphs are superimposed.

Kernelization Algorithm for FAST

- Kernelization is a basic technique in designing FPT algorithms.
- Since we are now in the realm of parameterized complexity, we will only be concerning ourselves with the *decision* version of the problem.

FAST (Decision Version)

Input:	A tournament G and an integer k.
Question:	Does G contain a feedback arc set of size (at most) k?

Kernelization Algorithm for FAST

- Kernelization is a basic technique in designing FPT algorithms.
- Since we are now in the realm of parameterized complexity, we will only be concerning ourselves with the *decision* version of the problem.

FAST (Decision Version)

Input:	A tournament G and an integer k.
Question:	Does G contain a feedback arc set of size (at most) k?

Kernelization Algorithm for FAST

- Kernelization is a basic technique in designing FPT algorithms.
- Since we are now in the realm of parameterized complexity, we will only be concerning ourselves with the *decision* version of the problem.

FAST (Decision Version)

Input:	A tournament G and an integer k.
Question:	Does G contain a feedback arc set of size (at most) k?

Kernelization Review

- Kernelization is like pre-processing: returns in polynomial time an *equivalent* instance which is "much smaller".
- Suffices to solve the smaller instance (with any of our favorite algorithm).

Kernelization

$$(G, k) \xrightarrow{kernelize} (G', k')$$

such that (G, k) is a yes-instance if and only if (G', k') is and the size of G' is "small" (bounded by some $f(k)$).

Kernelization Review

- Kernelization is like pre-processing: returns in polynomial time an *equivalent* instance which is "much smaller".
- Suffices to solve the smaller instance (with any of our favorite algorithm).

Kernelization

$$(G, k) \xrightarrow{kernelize} (G', k')$$

such that (G, k) is a yes-instance if and only if (G', k') is and the size of G' is "small" (bounded by some $f(k)$).

Kernelization Review

- Kernelization is like pre-processing: returns in polynomial time an *equivalent* instance which is “much smaller”.
- Suffices to solve the smaller instance (with any of our favorite algorithm).

Kernelization

$$(G, k) \xrightarrow{kernelize} (G', k')$$

such that (G, k) is a yes-instance if and only if (G', k') is and the size of G' is “small” (bounded by some $f(k)$).

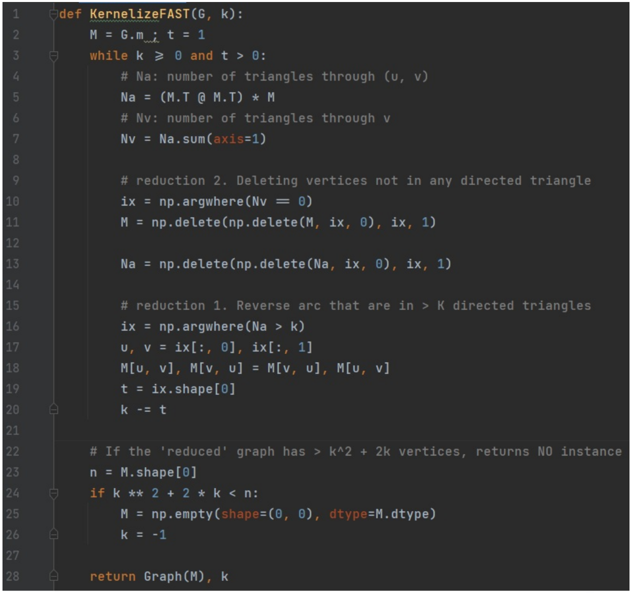
KernelizedFast

KernelizedFast

- If there exists an arc that participates in at least $k+1$ directed triangles, reverse it and decrement k by 1.
- If there exists a vertex that does not participate in any directed triangle, then delete it.

After applying these rules exhaustively, if the graph has $> k^2 + 2k$ vertices (or if $k \leq 0$), then no instance. Otherwise, return this reduced instance.

KernelizedFast



```
def KernelizeFAST(G, k):
    M = G.m ; t = 1
    while k ⩾ 0 and t > 0:
        # Na: number of triangles through (u, v)
        Na = (M.T @ M.T) * M
        # Nv: number of triangles through v
        Nv = Na.sum(axis=1)

        # reduction 2. Deleting vertices not in any directed triangle
        ix = np.argwhere(Nv == 0)
        M = np.delete(np.delete(M, ix, 0), ix, 1)

        Na = np.delete(np.delete(Na, ix, 0), ix, 1)

        # reduction 1. Reverse arc that are in > K directed triangles
        ix = np.argwhere(Na > k)
        u, v = ix[:, 0], ix[:, 1]
        M[u, v], M[v, u] = M[v, u], M[u, v]
        t = ix.shape[0]
        k -= t

    # If the 'reduced' graph has > k^2 + 2k vertices, returns NO instance
    n = M.shape[0]
    if k ** 2 + 2 * k < n:
        M = np.empty(shape=(0, 0), dtype=M.dtype)
        k = -1

    return Graph(M), k
```

KernelizedFAST Simulation

In the following table, we show the effect of KernelizedFAST on a set of tournaments. All input tournaments used in this experiment is up on github.

(n, k) before kernelization	(n, k) after kernelization	comments
$(100, 30)$	$(0, 0)$	yes-instance, kernelization solves it completely.
$(100, 30)$	$(6, -6)$	no-instance, kernelization solves it completely.
$(100, 30)$	$(13, 3)$	kernelization does not solve completely, but does significant size reduction.
$(100, 30)$	$(92, 20)$	no significant reduction **but** look at **k** carefully!

A Branching Parameterized Algorithm for FAST

- The final exact algorithm that we have implemented is a simple branching FPT algorithm for FAST.

A Branching Parameterized Algorithm for FAST

- The final exact algorithm that we have implemented is a simple branching FPT algorithm for FAST.
- Based on the following simple idea:
  - Look for a *directed* triangle in the tournament.
  - If none exists, then **yes** instance.
  - Otherwise, branch on each of its three arcs reversing them one at a time and recursively solve these three instances.

A Branching Parameterized Algorithm for FAST

- The final exact algorithm that we have implemented is a simple branching FPT algorithm for FAST.
- Based on the following simple idea:
    - Look for a *directed* triangle in the tournament.
    - If none exists, then **yes** instance.
    - Otherwise, branch on each of its three arcs reversing them one at a time and recursively solve these three instances.

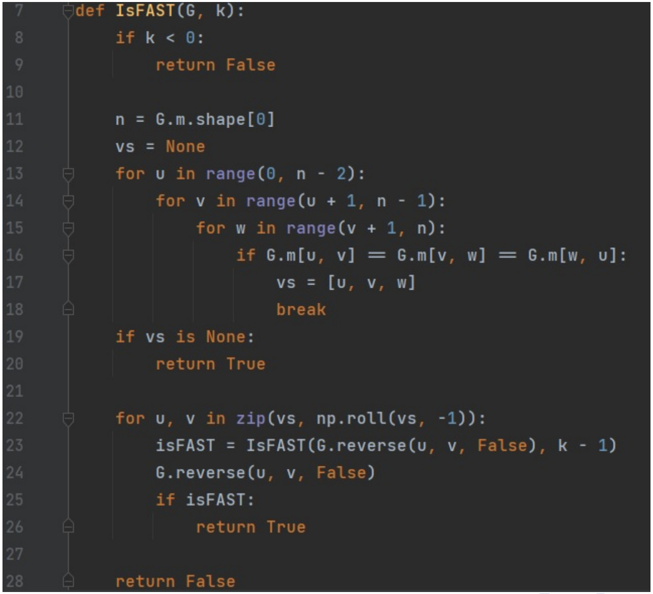
A Branching Parameterized Algorithm for FAST

- The final exact algorithm that we have implemented is a simple branching FPT algorithm for FAST.
- Based on the following simple idea:
  - Look for a *directed* triangle in the tournament.
  - If none exists, then **yes** instance.
  - Otherwise, branch on each of its three arcs reversing them one at a time and recursively solve these three instances.

A Branching Parameterized Algorithm for FAST

- The final exact algorithm that we have implemented is a simple branching FPT algorithm for FAST.
- Based on the following simple idea:
    - Look for a *directed* triangle in the tournament.
    - If none exists, then **yes** instance.
    - Otherwise, branch on each of its three arcs reversing them one at a time and recursively solve these three instances.

A Branching Parameterized Algorithm for FAST



```python
def IsFAST(G, k):
    if k < 0:
        return False

    n = G.m.shape[0]
    vs = None
    for u in range(0, n - 2):
        for v in range(u + 1, n - 1):
            for w in range(v + 1, n):
                if G.m[u, v] == G.m[v, w] == G.m[w, u]:
                    vs = [u, v, w]
                    break
    if vs is None:
        return True

    for u, v in zip(vs, np.roll(vs, -1)):
        isFAST = IsFAST(G.reverse(u, v, False), k - 1)
        G.reverse(u, v, False)
        if isFAST:
            return True

    return False
```

A Branching Parameterized Algorithm for FAST

- Has a running time of $3^k n^{\mathcal{O}(1)}$.
- Feasible on large graphs as long as k is small (since running time is only exponential in k).
- Frontier of tractability can be extended by applying kernelization *before* using the branching algorithm.

A Branching Parameterized Algorithm for FAST

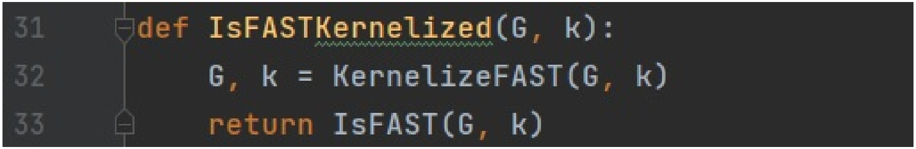
- Has a running time of $3^k n^{\mathcal{O}(1)}$.
- Feasible on large graphs as long as k is small (since running time is only exponential in k).
- Frontier of tractability can be extended by applying kernelization *before* using the branching algorithm.

A Branching Parameterized Algorithm for FAST

- Has a running time of $3^k n^{\mathcal{O}(1)}$.
- Feasible on large graphs as long as k is small (since running time is only exponential in k).
- Frontier of tractability can be extended by applying kernelization *before* using the branching algorithm.

A Branching Parameterized Algorithm for FAST

- Has a running time of $3^k n^{\mathcal{O}(1)}$.
- Feasible on large graphs as long as k is small (since running time is only exponential in k).
- Frontier of tractability can be extended by applying kernelization *before* using the branching algorithm.



```
def IsFASTKernelized(G, k):
    G, k = KernelizeFAST(G, k)
    return IsFAST(G, k)
```

Extending the Frontier of Tractability

- Recall that we used a number of tournaments with 100 vertices and $k = 30$ for our kernelization experiment.
- 3^{30} is a very large number. So, branching alone can not solve these instances. Would take more than 6 years! $(1\mu s/step)$
- But after kernelization, these instances become much more *tractable*.
- Kernelization solves the first two completely in polynomial time.
- The third instance takes a total of around 5-6 milliseconds only!
- Even the last instance (where kernelization did not make significant progress) can be solved in a couple of hours $(1\mu s/step)$.
- From years to hours!

Extending the Frontier of Tractability

- Recall that we used a number of tournaments with 100 vertices and $k = 30$ for our kernelization experiment.
- 3^{30} is a very large number. So, branching alone can not solve these instances. Would take more than 6 years! ($1\mu s/step$)
- But after kernelization, these instances become much more *tractable*.
- Kernelization solves the first two completely in polynomial time.
- The third instance takes a total of around 5-6 milliseconds only!
- Even the last instance (where kernelization did not make significant progress) can be solved in a couple of hours ($1\mu s/step$).
- From years to hours!

Extending the Frontier of Tractability

- Recall that we used a number of tournaments with 100 vertices and $k = 30$ for our kernelization experiment.
- 3^{30} is a very large number. So, branching alone can not solve these instances. Would take more than 6 years! ($1\mu s/step$)
- But after kernelization, these instances become much more *tractable*.
- Kernelization solves the first two completely in polynomial time.
- The third instance takes a total of around 5-6 milliseconds only!
- Even the last instance (where kernelization did not make significant progress) can be solved in a couple of hours ($1\mu s/step$).
- From years to hours!

Extending the Frontier of Tractability

- Recall that we used a number of tournaments with 100 vertices and $k = 30$ for our kernelization experiment.
- 3^{30} is a very large number. So, branching alone can not solve these instances. Would take more than 6 years! ($1\mu s/step$)
- But after kernelization, these instances become much more *tractable*.
- Kernelization solves the first two completely in polynomial time.
- The third instance takes a total of around 5-6 milliseconds only!
- Even the last instance (where kernelization did not make significant progress) can be solved in a couple of hours ($1\mu s/step$).
- From years to hours!

Extending the Frontier of Tractability

- Recall that we used a number of tournaments with 100 vertices and $k = 30$ for our kernelization experiment.
- 3^{30} is a very large number. So, branching alone can not solve these instances. Would take more than 6 years! ($1\mu s/step$)
- But after kernelization, these instances become much more *tractable*.
- Kernelization solves the first two completely in polynomial time.
- The third instance takes a total of around 5-6 milliseconds only!
- Even the last instance (where kernelization did not make significant progress) can be solved in a couple of hours ($1\mu s/step$).
- From years to hours!

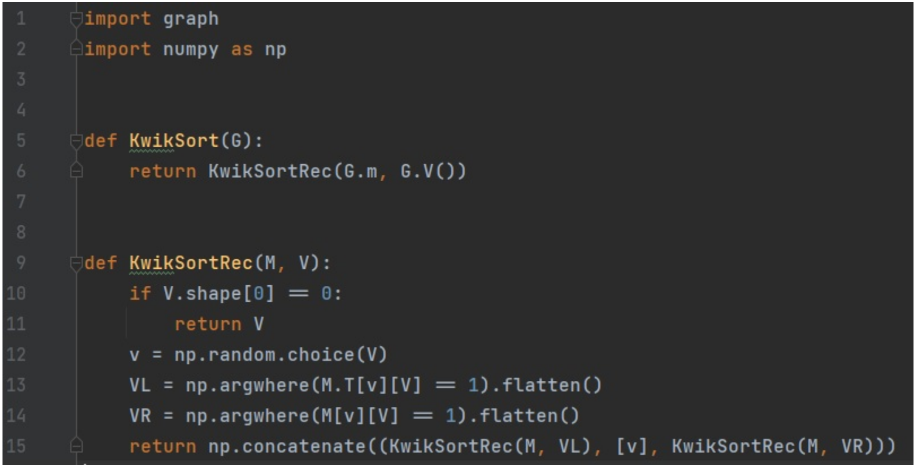
Extending the Frontier of Tractability

- Recall that we used a number of tournaments with 100 vertices and $k = 30$ for our kernelization experiment.
- 3^{30} is a very large number. So, branching alone can not solve these instances. Would take more than 6 years! ($1\mu s/step$)
- But after kernelization, these instances become much more *tractable*.
- Kernelization solves the first two completely in polynomial time.
- The third instance takes a total of around 5-6 milliseconds only!
- Even the last instance (where kernelization did not make significant progress) can be solved in a couple of hours ($1\mu s/step$).
- From years to hours!

Extending the Frontier of Tractability

- Recall that we used a number of tournaments with 100 vertices and $k = 30$ for our kernelization experiment.
- 3^{30} is a very large number. So, branching alone can not solve these instances. Would take more than 6 years! ($1\mu s/step$)
- But after kernelization, these instances become much more *tractable*.
- Kernelization solves the first two completely in polynomial time.
- The third instance takes a total of around 5-6 milliseconds only!
- Even the last instance (where kernelization did not make significant progress) can be solved in a couple of hours ($1\mu s/step$).
- From years to hours!

KwikSort Implementation



```python
import graph
import numpy as np


def KwikSort(G):
    return KwikSortRec(G.m, G.V())


def KwikSortRec(M, V):
    if V.shape[0] == 0:
        return V
    v = np.random.choice(V)
    VL = np.argwhere(M.T[v][V] == 1).flatten()
    VR = np.argwhere(M[v][V] == 1).flatten()
    return np.concatenate((KwikSortRec(M, VL), [v], KwikSortRec(M, VR)))
```

KwikSort Comparison

The following table lists the sizes of feedback arc sets found by KwikSort on a set of random tournaments.

# of Vertices	OPT	KwikSort	A.R. ≤ 3
05	02	02	1.00
07	05	08	1.60
11	14	15	1.07
12	15	21	1.40
15	25	33	1.32
17	31	38	1.23
18	42	49	1.17
20	44	73	1.66
23	72	89	1.24
24	78	98	1.26

KwikSort Comparison

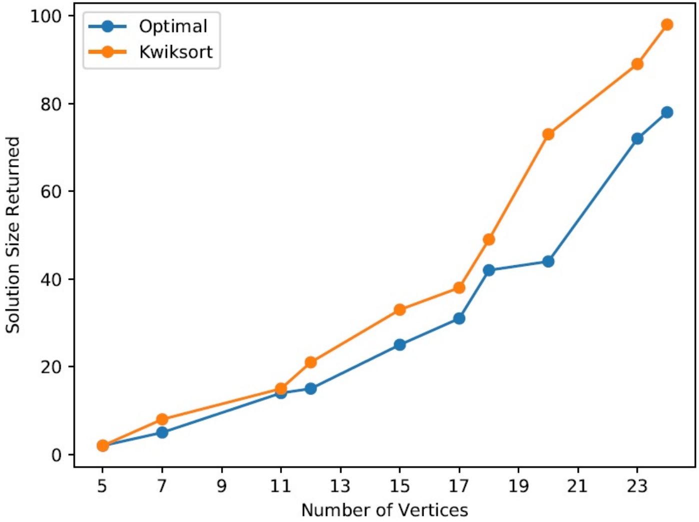


Figure 5: Comparison of KwikSort with Optimal Solution

KwikSort Comparison

- The OPT values were computed using the dynamic programming algorithm shown earlier.
- *This* is the reason why the table stops at $n = 24$
- KwikSort *clearly* does much better than its theoretical guarantee on random tournaments!

KwikSort Comparison

- The OPT values were computed using the dynamic programming algorithm shown earlier.
- *This* is the reason why the table stops at $n = 24$
- KwikSort *clearly* does much better than its theoretical guarantee on random tournaments!

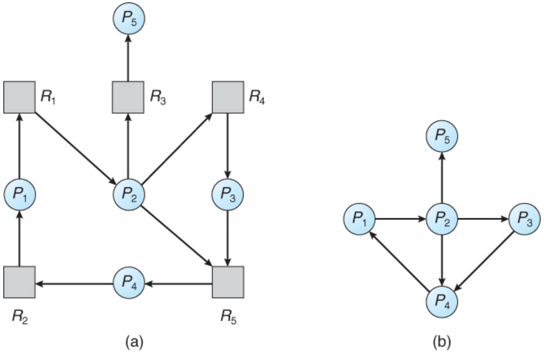
KwikSort Comparison

- The OPT values were computed using the dynamic programming algorithm shown earlier.
- *This* is the reason why the table stops at $n = 24$
- KwikSort *clearly* does much better than its theoretical guarantee on random tournaments!

Outline

Applications

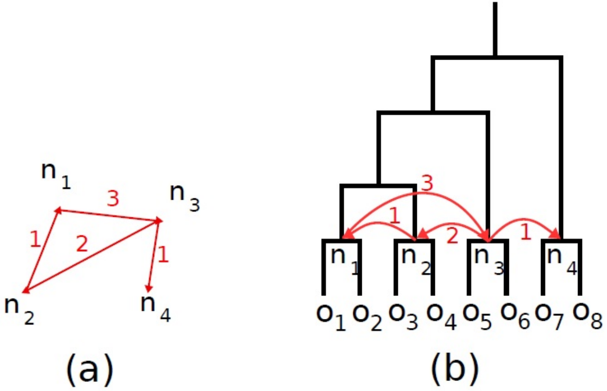
- Can be helpful in **deadlock prevention** in computer systems.



- In **combinational circuit design**, where cycles can potentially lead to race conditions.

Applications

Fast ranking of a phylogenetic tree by Maximum Time Consistency with lateral gene transfers → If we see the branches of the unranked species tree as arcs of a directed graph with infinite weight, and the constraints as weighted arcs in this graph, **then the Maximum Time Consistency problem translates exactly into an instance of the Feedback Arc Set problem** [2].



Citations I

Nir Ailon, Moses Charikar, and Alantha Newman.
Aggregating inconsistent information: Ranking and clustering.
J. ACM, 55(5), nov 2008.

Cédric Chauve, Akbar Rafiey, Adrián A. Davín, Celine Scornavacca, Philippe Veber, Bastien Boussau, Gergely J. Szöllősi, Vincent Daubin, and Eric Tannier.
Maxtic: Fast ranking of a phylogenetic tree by maximum time consistency with lateral gene transfers.
bioRxiv, 2017.

Jianer Chen, Yang Liu, Songjian Lu, Barry O'sullivan, and Igor Razgon.
A fixed-parameter algorithm for the directed feedback vertex set problem.
J. ACM, 55(5), nov 2008.

Citations II

Guy Even, Joseph (Seffi) Naor, Baruch Schieber, and Madhu Sudan.
Approximating minimum feedback sets and multi-cuts in directed graphs.
Algorithmica, 20:151–174, 1998.

Michael R. Garey and David S. Johnson.
Computers and intractability. a guide to the theory of np-completeness.
Journal of Symbolic Logic, 48(2):498–500, 1983.

Michael Held and Richard M. Karp.
A dynamic programming approach to sequencing problems.
Journal of the Society for Industrial and Applied Mathematics, 10(1):196–210, 1962.

Citations III

Richard M. Karp.
Reducibility among Combinatorial Problems, pages 85–103.
Springer US, Boston, MA, 1972.

Mikko Koivisto and Pekka Parviainen.
A space-time tradeoff for permutation problems.
In *In Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2010*, pages 484–492, 2010.

E. Lawler.
A comment on minimum feedback arc sets.
IEEE Transactions on Circuit Theory, 11(2):296–297, 1964.

Claire Mathieu and Warren Schudy.
How to rank with fewer errors – a ptas for feedback arc set in tournaments.

THANK YOU