Exact Algorithms for the Feedback Arc Set Problem

Rayhan Rashed (1505006) Nishat Anjum Bristy (1505007)
Tawhidul Hasan (1505008) Sadia Afroz (1505030)
Mursalin Habib (1505049)

Bangladesh University of Engineering and Technology

September 28, 2020

Table of Contents

A Recap

- Today we will be talking about exact algorithms for the FEEDBACK ARC SET problem.
- Let us first recap what feedback arc sets are.

A Recap

- Today we will be talking about exact algorithms for the Feedback Arc Set problem.
- Let us first recap what feedback arc sets are.

What is a Feedback Arc Set?

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

Definition (Feedback Arc Set)

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.

What is a Feedback Arc Set?

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

Definition (Feedback Arc Set)

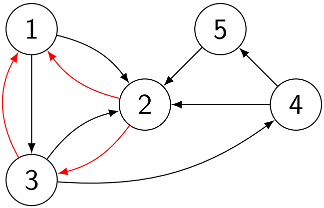
Given a directed graph $G = (V, A)$, a feedback arc set $F \subseteq A$ is a set of arcs such that $G - F$ is acyclic.

What is a Feedback Arc Set?

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

Definition (Feedback Arc Set)

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.

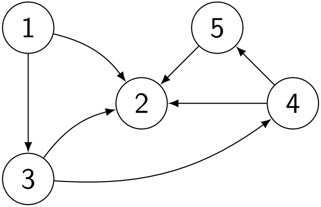


What is a Feedback Arc Set?

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

Definition (Feedback Arc Set)

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.



Recap: FEEDBACK ARC SET is Hard!

- From our previous presentation, we know that FEEDBACK ARC SET is *NP*-hard.
- So, unless $P = NP$, there do not exist polynomial time algorithms that 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.

Recap: Feedback Arc Set is Hard!

- From our previous presentation, we know that Feedback Arc Set is *NP*-hard.
- So, unless $P = NP$, there do not exist polynomial time algorithms that 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.

Recap: FEEDBACK ARC SET is Hard!

- From our previous presentation, we know that FEEDBACK ARC SET is *NP*-hard.
- So, unless $P = NP$, there do not exist polynomial time algorithms that 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.

Recap: FEEDBACK ARC SET is Hard!

- However, not all hope is lost!
- By using clever algorithmic techniques, we can sometimes have exact algorithms that are significantly better than the naive brute-force algorithms one might come up with at first.
- Understanding how to design such algorithms is the goal of this presentation.

Recap: Feedback Arc Set is Hard!

- However, not all hope is lost!
- By using clever algorithmic techniques, we can sometimes have exact algorithms that are significantly better than the naive brute-force algorithms one might come up with at first.
- Understanding how to design such algorithms is the goal of this presentation.

Recap: Feedback Arc Set is Hard!

- However, not all hope is lost!
- By using clever algorithmic techniques, we can sometimes have exact algorithms that are significantly better than the naive brute-force algorithms one might come up with at first.
- Understanding how to design such algorithms is the goal of this presentation.

Table of Contents

Organization of the Presentation

Let us talk a little bit about how this presentation will be organized.

- First, a small discussion on brute-force search algorithms (bad running times but are a starting point for designing better algorithms).
- Next, modify one of these brute-force algorithms with dynamic programming (significantly better running time but exponential space).
- After that, fix the space complexity issue using divide-and-conquer (polynomial space but worse running time than dynamic programming).
- Finally, some comments on the parameterized complexity of FEEDBACK ARC SET and its polynomial variants.

Organization of the Presentation

Let us talk a little bit about how this presentation will be organized.

- First, a small discussion on brute-force search algorithms (bad running times but are a starting point for designing better algorithms).
- Next, modify one of these brute-force algorithms with dynamic programming (significantly better running time but exponential space).
- After that, fix the space complexity issue using divide-and-conquer (polynomial space but worse running time than dynamic programming).
- Finally, some comments on the parameterized complexity of Feedback Arc Set and its polynomial variants.

Organization of the Presentation

Let us talk a little bit about how this presentation will be organized.

- First, a small discussion on brute-force search algorithms (bad running times but are a starting point for designing better algorithms).
- Next, modify one of these brute-force algorithms with dynamic programming (significantly better running time but exponential space).
- After that, fix the space complexity issue using divide-and-conquer (polynomial space but worse running time than dynamic programming).
- Finally, some comments on the parameterized complexity of FEEDBACK ARC SET and its polynomial variants.

Organization of the Presentation

Let us talk a little bit about how this presentation will be organized.

- First, a small discussion on brute-force search algorithms (bad running times but are a starting point for designing better algorithms).
- Next, modify one of these brute-force algorithms with dynamic programming (significantly better running time but exponential space).
- After that, fix the space complexity issue using divide-and-conquer (polynomial space but worse running time than dynamic programming).
- Finally, some comments on the parameterized complexity of Feedback Arc Set and its polynomial variants.

Organization of the Presentation

Let us talk a little bit about how this presentation will be organized.

- First, a small discussion on brute-force search algorithms (bad running times but are a starting point for designing better algorithms).
- Next, modify one of these brute-force algorithms with dynamic programming (significantly better running time but exponential space).
- After that, fix the space complexity issue using divide-and-conquer (polynomial space but worse running time than dynamic programming).
- Finally, some comments on the parameterized complexity of Feedback Arc Set and its polynomial variants.

Table of Contents

Brute-Force Algorithms for FEEDBACK ARC SET

- Why do we care about brute-force algorithms?
- Although brute-force algorithms usually have very bad running times and are only feasible on the smallest of input instances, 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.
- We will see an example of this later.

Brute-Force Algorithms for FEEDBACK ARC SET

- Why do we care about brute-force algorithms?
- Although brute-force algorithms usually have very bad running times and are only feasible on the smallest of input instances, 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.
- We will see an example of this later.

Brute-Force Algorithms for FEEDBACK ARC SET

- Why do we care about brute-force algorithms?
- Although brute-force algorithms usually have very bad running times and are only feasible on the smallest of input instances, 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.
- We will see an example of this later.

Brute-Force Algorithms for Feedback Arc Set

- Why do we care about brute-force algorithms?
- Although brute-force algorithms usually have very bad running times and are only feasible on the smallest of input instances, 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.
- We will see an example of this later.

Brute-Force Algorithms for Feedback Arc Set

- Let us first try to solve Feedback Arc Set in the most naive way possible.
- We can look at every subset of the arcs and check if it is a feedback arc set.
- This gives us the following algorithm.

Brute-Force Algorithms for Feedback Arc Set

- Let us first try to solve Feedback Arc Set in the most naive way possible.
- We can look at every subset of the arcs and check if it is a feedback arc set.
- This gives us the following algorithm.

Brute-Force Algorithms for Feedback Arc Set

- Let us first try to solve Feedback Arc Set in the most naive way possible.
- We can look at every subset of the arcs and check if it is a feedback arc set.
- 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

$m \leftarrow \infty$;
$F^* \leftarrow \varnothing$;
foreach $F \subseteq A$ **do**
  $G' \leftarrow G - F$;
  if G' *is acyclic* **and** $|F| < m$ **then**
    $m \leftarrow |F|$;
    $F^* \leftarrow F$;
  end
end
return F^*

Running Time of NAIVEFEEDBACKARCSET(G)

- How bad is this algorithm?
- This algorithm always has to look at every possible subset of the arcs.
- Therefore, this algorithm 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.

Running Time of NAIVEFEEDBACKARCSET(G)

- How bad is this algorithm?
- This algorithm always has to look at every possible subset of the arcs.
- Therefore, this algorithm 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.

Running Time of NaiveFeedbackArcSet(G)

- How bad is this algorithm?
- This algorithm always has to look at every possible subset of the arcs.
- Therefore, this algorithm 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.

Running Time of NAIVEFEEDBACKARCSET(G)

- How bad is this algorithm?
- This algorithm always has to look at every possible subset of the arcs.
- Therefore, this algorithm 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 that 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

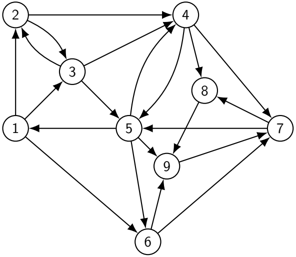
- We can come up with a slightly cleverer algorithm by using that fact that every directed acyclic graph has a topological ordering.

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.

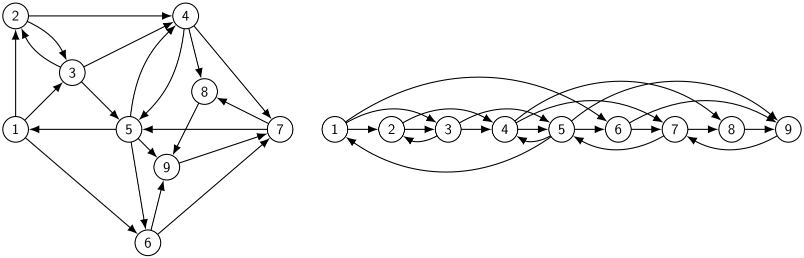
A Better Brute-Force Algorithm

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



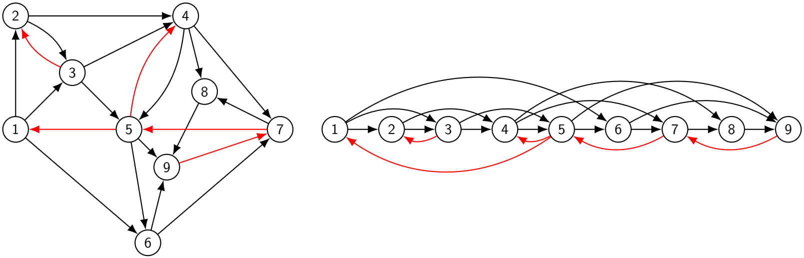
A Better Brute-Force Algorithm

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



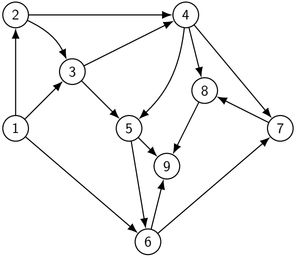
A Better Brute-Force Algorithm

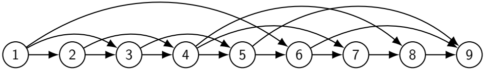
- **The idea:** Consider any arbitrary permutation of the vertices. Then the set of "backward" arcs constitute a feedback arc set.



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 $V = \{v_1, v_2, \cdots, v_n\}$ and π be a permutation of the numbers $1, 2, \cdots, n$. 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

$m \leftarrow \infty$;
$F^* \leftarrow \varnothing$;
foreach *permutation* π *of the numbers* $1, 2, \cdots, |V|$ **do**
  $F \leftarrow \{(v_{\pi(i)}, v_{\pi(j)}) \in A : \pi(i) > \pi(j)\}$;
  if $|F| < m$ **then**
    $m \leftarrow |F|$;
    $F^* \leftarrow F$;
  end
end
return F^*

Analyzing 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 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 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 consider the dynamic programming approach which has been very successful in solving such problems.

Table of Contents

A DP Algorithm for Feedback Arc Set

- As promised, 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 the classic Held-Karp algorithm (1962) for solving the Traveling Salesperson Problem [1].

A DP Algorithm for FEEDBACK ARC SET

- As promised, 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 the classic Held-Karp algorithm (1962) for solving the TRAVELING SALESPERSON PROBLEM [1].

Setting Up the Dynamic Program

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

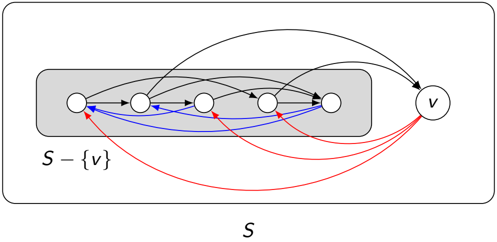


Figure 1: Number of blue arcs $= OPT[S - \{v\}]$, number of red arcs$=c(v, S - \{v\})$.

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) [2].

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) [2].

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) [2].

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

foreach $v \in V$ **do**
  $OPT[\{v\}] \leftarrow 0$;
end
for $i \leftarrow 2$ **to** n **do**
  foreach $S \subseteq V$ *with* $|S| = i$ **do**
    $OPT[S] \leftarrow \min_{v \in S}\{OPT[S - \{v\}] + c(v, S - \{v\})\}$;
  end
end
return $OPT[V]$

Analyzing the Running Time

- 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.

Analyzing the Running Time

- 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.

Analyzing the Running Time

- 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.

Analyzing the Running Time

- 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:
- $O\left(\sum_{k=0}^{n} k^2\binom{n}{k}\right) = O\left(\sum_{k=0}^{n}\left(n\binom{n-1}{k-1} + n(n-1)\binom{n-2}{k-2}\right)\right) = O(n^2 2^n) = O^*(2^n)$.

Analyzing the Running Time

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

Analyzing the Running Time

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

Analyzing the Running Time

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

Analyzing the Running Time

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.

Table of Contents

Trading Time for Space: Divide and 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 and conquer**.

Trading Time for Space: Divide and 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 and conquer**.

Trading Time for Space: Divide and Conquer!

- 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-and-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.

Trading Time for Space: Divide and Conquer!

- 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-and-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.

Trading Time for Space: Divide and Conquer!

- 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-and-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'.

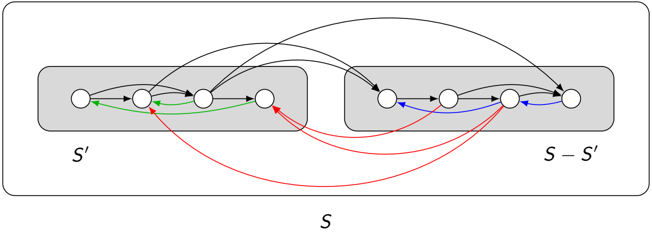


Figure 2: $OPT(S')$ =number of green arcs, $OPT(S - S')$ =number of blue arcs, $c(S - S', S')$ =number of red arcs.

Divide-and-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.

Divide-and-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&CFEEDBACKARCSET(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

Function $OPT(S)$
  if $|S| = 1$ **then**
    return 0
  end
  return $\min_{\substack{S' \subseteq S \\ |S'| = \lceil \frac{|S|}{2} \rceil}} \{OPT(S') + OPT(S - S') + c(S - S', S')\};$
end
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 lg n.

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 $\lg n$.

Analyzing the Time Complexity

- The running time analysis is slightly trickier.

The Recurrence: Recap

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

- For a fixed subset S of size k, the number of subsets S' of S that we have 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 have 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

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

- For a fixed subset S of size k, the number of subsets S' of S that we have 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 have 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

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

- For a fixed subset S of size k, the number of subsets S' of S that we have 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 *floor* and *ceiling* to *exact* value.

Analyzing the Time Complexity

The Running Time

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

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

Analyzing the Time Complexity

The Running Time

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

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

Analyzing the Time Complexity

The Running Time

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

Replacing the finite sum with an infinite sum.

Analyzing the Time Complexity

The Running Time

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

Analyzing the Time Complexity

The Running Time

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

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 $\mathcal{O}^*(4^n)$ and $\mathcal{O}^*(1)$.
- The time and space complexities of our dynamic programming algorithm are $\mathcal{O}^*(2^n)$ and $\mathcal{O}^*(2^n)$.
- In both cases, $\mathit{TIME} \times \mathit{SPACE} = \mathcal{O}^*(4^n)$.
- 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 $\mathcal{O}^*(4^n)$ and $\mathcal{O}^*(1)$.
- The time and space complexities of our dynamic programming algorithm are $\mathcal{O}^*(2^n)$ and $\mathcal{O}^*(2^n)$.
- In both cases, $\mathit{TIME} \times \mathit{SPACE} = \mathcal{O}^*(4^n)$.
- 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 $\mathcal{O}^*(4^n)$ and $\mathcal{O}^*(1)$.
- The time and space complexities of our dynamic programming algorithm are $\mathcal{O}^*(2^n)$ and $\mathcal{O}^*(2^n)$.
- In both cases, $\mathit{TIME} \times \mathit{SPACE} = \mathcal{O}^*(4^n)$.
- 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 $\mathcal{O}^*(4^n)$ and $\mathcal{O}^*(1)$.
- The time and space complexities of our dynamic programming algorithm are $\mathcal{O}^*(2^n)$ and $\mathcal{O}^*(2^n)$.
- In both cases, $\mathit{TIME} \times \mathit{SPACE} = \mathcal{O}^*(4^n)$.
- 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 $\mathcal{O}^*(4^n)$ and $\mathcal{O}^*(1)$.
- The time and space complexities of our dynamic programming algorithm are $\mathcal{O}^*(2^n)$ 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.
- $TIME \times SPACE$ is still $\mathcal{O}^*(4^n)$ in this hybrid approach.
- However, using an idea by Koivisto and Parviainen (2010) [3], 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.
- $TIME \times SPACE$ is still $\mathcal{O}^*(4^n)$ in this hybrid approach.
- However, using an idea by Koivisto and Parviainen (2010) [3], 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.
- $\mathit{TIME} \times \mathit{SPACE}$ is still $\mathcal{O}^*(4^n)$ in this hybrid approach.
- However, using an idea by Koivisto and Parviainen (2010) [3], it is possible to get $\mathit{TIME} \times \mathit{SPACE} = \mathcal{O}^*(3.93^n)$.

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.
- $TIME \times SPACE$ is still $\mathcal{O}^*(4^n)$ in this hybrid approach.
- However, using an idea by Koivisto and Parviainen (2010) [3], it is possible to get $TIME \times SPACE = \mathcal{O}^*(3.93^n)$.

Table of Contents

Parameterized Complexity

- The decision version of FEEDBACK ARC SET (when parameterized by the size of the feedback arc set desired) is fixed parameter tractable.
- The running time of the best known such algorithm is $O\left((k+1)! \cdot 4^k \cdot k^3 \cdot n(n+m)\right)$ [4] (2008) (where k is the size of the feedback arc set being asked for).
- This algorithm uses a technique known as "iterative compression".
- Details in the final report submitted at the end of the course.

Parameterized Complexity

- The decision version of FEEDBACK ARC SET (when parameterized by the size of the feedback arc set desired) is fixed parameter tractable.
- The running time of the best known such algorithm is $O\left((k+1)! \cdot 4^k \cdot k^3 \cdot n(n+m)\right)$ [4] (2008) (where k is the size of the feedback arc set being asked for).
- This algorithm uses a technique known as "iterative compression".
- Details in the final report submitted at the end of the course.

Parameterized Complexity

- The decision version of FEEDBACK ARC SET (when parameterized by the size of the feedback arc set desired) is fixed parameter tractable.
- The running time of the best known such algorithm is $O\left((k+1)! \cdot 4^k \cdot k^3 \cdot n(n+m)\right)$ [4] (2008) (where k is the size of the feedback arc set being asked for).
- This algorithm uses a technique known as "iterative compression".
- Details in the final report submitted at the end of the course.

Parameterized Complexity

- The decision version of FEEDBACK ARC SET (when parameterized by the size of the feedback arc set desired) is fixed parameter tractable.
- The running time of the best known such algorithm is $O\left((k+1)! \cdot 4^k \cdot k^3 \cdot n(n+m)\right)$ [4] (2008) (where k is the size of the feedback arc set being asked for).
- This algorithm uses a technique known as "iterative compression".
- Details in the final report submitted at the end of the course.

Table of Contents

Poly-time algorithms for Restricted Instances

- Now we are going to talk about the final topic of this presentation: poly-time algorithms for FEEDBACK ARC SET on restricted inputs.
- Usually, problems on directed graphs are harder than those on undirected graphs since in the latter, more graph theory can be utilized.
- As a result, there are not many polynomial time variants of the FEEDBACK ARC SET problem.

Poly-time algorithms for Restricted Instances

- Now we are going to talk about the final topic of this presentation: poly-time algorithms for FEEDBACK ARC SET on restricted inputs.
- Usually, problems on directed graphs are harder than those on undirected graphs since in the latter, more graph theory can be utilized.
- As a result, there are not many polynomial time variants of the FEEDBACK ARC SET problem.

Poly-time algorithms for Restricted Instances

- Now we are going to talk about the final topic of this presentation: poly-time algorithms for FEEDBACK ARC SET on restricted inputs.
- Usually, problems on directed graphs are harder than those on undirected graphs since in the latter, more graph theory can be utilized.
- As a result, there are not many polynomial time variants of the FEEDBACK ARC SET problem.

Poly-time algorithms for Restricted Instances

- 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.

Poly-time algorithms for Restricted Instances

- 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.

Poly-time algorithms for Restricted Instances

- 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.

Poly-time algorithms for Restricted Instances

- It turns out that FEEDBACK ARC SET can be solved in polynomial time if the inputs are restricted to only planar digraphs [5].
- To understand why this is so, we must learn what dijoins are.

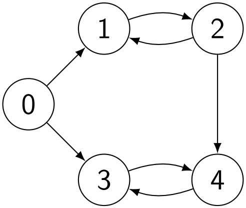
Poly-time algorithms for Restricted Instances

- It turns out that FEEDBACK ARC SET can be solved in polynomial time if the inputs are restricted to only planar digraphs [5].
- To understand why this is so, we must learn what dijoins are.

Poly-time algorithms for Restricted Instances

Definition (Dijoin)

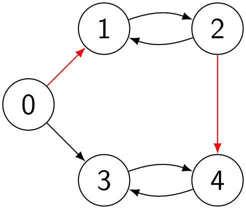
Given a directed graph $G = (V, A)$, a dijoin is a subset F of its arcs such that if we add the reversed version of all the arcs in F to G, G will be strongly-connected.



Poly-time algorithms for Restricted Instances

Definition (Dijoin)

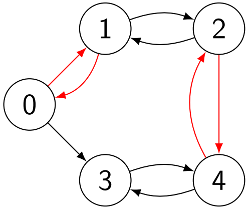
Given a directed graph $G = (V, A)$, a dijoin is a subset F of its arcs such that if we add the reversed version of all the arcs in F to G, G will be strongly-connected.



Poly-time algorithms for Restricted Instances

Definition (Dijoin)

Given a directed graph $G = (V, A)$, a dijoin is a subset F of its arcs such that if we add the reversed version of all the arcs in F to G, G will be strongly-connected.



Poly-time algorithms for Restricted Instances

- The key to solving Feedback Arc Set on Planar Digraphs in polynomial time is the following two results:

Theorem

Finding a minimum dijoin of a directed graph can be done in polynomial time [6, 7, 8] (1981, 1995, 2005).

Theorem

Finding a minimum feedback arc set of a planar digraph is equivalent to finding a minimum dijoin of its dual [9].

- Combining these two results, we get a polynomial time algorithm for Feedback Arc Set on Planar Digraphs.

Poly-time algorithms for Restricted Instances

- The key to solving FEEDBACK ARC SET ON PLANAR DIGRAPHS in polynomial time is the following two results:

Theorem

Finding a minimum dijoin of a directed graph can be done in polynomial time [6, 7, 8] (1981, 1995, 2005).

Theorem

Finding a minimum feedback arc set of a planar digraph is equivalent to finding a minimum dijoin of its dual [9].

- Combining these two results, we get a polynomial time algorithm for FEEDBACK ARC SET ON PLANAR DIGRAPHS.

Poly-time algorithms for Restricted Instances

- The key to solving Feedback Arc Set on Planar Digraphs in polynomial time is the following two results:

Theorem

Finding a minimum dijoin of a directed graph can be done in polynomial time [6, 7, 8] (1981, 1995, 2005).

Theorem

Finding a minimum feedback arc set of a planar digraph is equivalent to finding a minimum dijoin of its dual [9].

- Combining these two results, we get a polynomial time algorithm for Feedback Arc Set on Planar Digraphs.

Poly-time algorithms for Restricted Instances

- The key to solving Feedback Arc Set on Planar Digraphs in polynomial time is the following two results:

Theorem

Finding a minimum dijoin of a directed graph can be done in polynomial time [6, 7, 8] (1981, 1995, 2005).

Theorem

Finding a minimum feedback arc set of a planar digraph is equivalent to finding a minimum dijoin of its dual [9].

- Combining these two results, we get a polynomial time algorithm for Feedback Arc Set on Planar Digraphs.

References I

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.

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

References II

Mikko Koivisto and Pekka Parviainen.
A space–time tradeoff for permutation problems.
In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 484–492. SIAM, 2010.

Jianer Chen, Yang Liu, Songjian Lu, Barry O'sullivan, and Igor Razgon.
A fixed-parameter algorithm for the directed feedback vertex set problem.
In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 177–186, 2008.

References III

- Michael R Garey and David S Johnson.
  Computers and intractability, volume 174.
  freeman San Francisco, 1979.

- András Frank.
  How to make a digraph strongly connected.
  Combinatorica, 1(2):145–153, 1981.

- Harold N Gabow.
  Centroids, representations, and submodular flows.
  Journal of Algorithms, 18(3):586–628, 1995.

References IV

FB Shepherd and A Vetta.
Visualizing, finding and packing dijoins.
In *Graph theory and combinatorial optimization*, pages 219–254. Springer, 2005.

Cláudio Leonardo Lucchesi.
A minimax equality for directed graphs.
1977.