Parameterized Algorithms

Introductory Techniques and Complexity

Ellie Anastasiadi

May 12, 2018

Outline

Outline

Outline

Facing Untractability via Parameters

When he hit the wall of NP-completeness we try other methods

- Probabilistic Algorithms (losing certainty)
- Approximations (sacrificing the exact solution)

Through Parameterized algorithms we avoid the above by searching solutions for only part or the universe of the instances

Parameters

We correlate a problem with a parameter and design algorithms on the notion that our instances have the given parameter bounded and in general terms small

Facing Untractability via Parameters

When he hit the wall of NP-completeness we try other methods

- Probabilistic Algorithms (losing certainty)
- Approximations (sacrificing the exact solution)

Through Parameterized algorithms we avoid the above by searching solutions for only part or the universe of the instances

Parameters

We correlate a problem with a parameter and design algorithms on the notion that our instances have the given parameter bounded and in general terms small

Facing Untractability via Parameters

When he hit the wall of NP-completeness we try other methods

- Probabilistic Algorithms (losing certainty)
- Approximations (sacrificing the exact solution)

Through Parameterized algorithms we avoid the above by searching solutions for only part or the universe of the instances

Parameters

We correlate a problem with a parameter and design algorithms on the notion that our instances have the given parameter bounded and in general terms small

Facing Untractability via Parameters

When he hit the wall of NP-completeness we try other methods

- Probabilistic Algorithms (losing certainty)
- Approximations (sacrificing the exact solution)

Through Parameterized algorithms we avoid the above by searching solutions for only part or the universe of the instances

Parameters

We correlate a problem with a parameter and design algorithms on the notion that our instances have the given parameter bounded and in general terms small

Outline

1 Introduction
- Basic Idea
- Formal Definitions
- Parameters and Problems

Our Tools

Parameterized Problem

A *parameterization* of Σ^* is a *recursive* function $k : \Sigma^* \rightarrow \mathbb{N}$.
A *parameterized problem* is a tuple (L, k), where $L \subseteq \Sigma^*$ and k is a parameterization of Σ^*.

The Class FPT

The class of parameterized problems that can be solved in time

$$O(f(k) * n^c)$$

, where f(k) is computable.

As always the classification of problems in classes refers to the best known algorithm or reduction for a parameterized problem

Our Tools

Parameterized Problem

A *parameterization* of Σ^* is a *recursive* function $k : \Sigma^* \rightarrow \mathbb{N}$.
A *parameterized problem* is a tuple (L, k), where $L \subseteq \Sigma^*$ and k is a parameterization of Σ^*.

The Class FPT

The class of parameterized problems that can be solved in time

$$O(f(k) * n^c)$$

, where f(k) is computable.

As always the classification of problems in classes refers to the best known algorithm or reduction for a parameterized problem

Our Tools

Parameterized Problem

A *parameterization* of Σ^* is a *recursive* function $k : \Sigma^* \rightarrow \mathbb{N}$.
A *parameterized problem* is a tuple (L, k), where $L \subseteq \Sigma^*$ and k is a parameterization of Σ^*.

The Class FPT

The class of parameterized problems that can be solved in time

$$O(f(k) * n^c)$$

, where f(k) is computable.

As always the classification of problems in classes refers to the best known algorithm or reduction for a parameterized problem

Outline

Picking your weapon

One can therefore design an algorithm that runs in the above time for any parameter he chooses. For instance we could try parameterizing a problem by the parameter n-1 where n is th size. The above definition does not give any information towards the *nature* of the parameter. The example given here would characterise any NP-hard problem as FPT. This is obviously not what we meant when requesting parameterized tractability for NP-hard problems . There are two important thing to keep in mind when choosing a parameter.

1. The parameter will be considered constant and small - We have to choose it in a way that is realistic .
2. The instances that have the parameter satisfying the above should be as many as possible

Picking your weapon

One can therefore design an algorithm that runs in the above time for any parameter he chooses. For instance we could try parameterizing a problem by the parameter n-1 where n is th size. The above definition does not give any information towards the *nature* of the parameter. The example given here would characterise any NP-hard problem as FPT. This is obviously not what we meant when requesting parameterized tractability for NP-hard problems . There are two important thing to keep in mind when choosing a parameter.

1. The parameter will be considered constant and small - We have to choose it in a way that is realistic .
2. The instances that have the parameter satisfying the above should be as many as possible

Picking your weapon

One can therefore design an algorithm that runs in the above time for any parameter he chooses. For instance we could try parameterizing a problem by the parameter n-1 where n is th size. The above definition does not give any information towards the *nature* of the parameter. The example given here would characterise any NP-hard problem as FPT. This is obviously not what we meant when requesting parameterized tractability for NP-hard problems . There are two important thing to keep in mind when choosing a parameter.

1. The parameter will be considered constant and small - We have to choose it in a way that is realistic .
2. The instances that have the parameter satisfying the above should be as many as possible

Examples

- In Optimisation problems one of the most common parameters is the size of the solution (called natural parameterization)
- On Constraint problems (such as SAT) we often parameterize by the number of constraints (in SAT that would mean the number of Clauses)
- For properties of graphs we often use parameters such as max degree , colour number and other easy or hard to track properties (such as tree-width which we will study later)

Examples

- In Optimisation problems one of the most common parameters is the size of the solution (called natural parameterization)
- On Constraint problems (such as SAT) we often parameterize by the number of constraints (in SAT that would mean the number of Clauses)
- For properties of graphs we often use parameters such as max degree , colour number and other easy or hard to track properties (such as tree-width which we will study later)

Examples

- In Optimisation problems one of the most common parameters is the size of the solution (called natural parameterization)
- On Constraint problems (such as SAT) we often parameterize by the number of constraints (in SAT that would mean the number of Clauses)
- For properties of graphs we often use parameters such as max degree , colour number and other easy or hard to track properties (such as tree-width which we will study later)

Outline

Some guys walk into a bar

In a town the Doorman of a bar must choose who he lets in so that there will be no feuds between them. We represent the people by vertices and the feuds by edges between them.

Vertex Cover

Given a graph G=(V,E) find the min opt number of vertices to delete so there will be no edges left in the graph.

Official Parameterized Version

Input: A Graph G=(V,E) Parameter: A Positive Integer k
Output: Does G have a VC of size k?

Since this is an optimisation problem as we already mentioned we usually use as a parameter the size of the solution.

Some guys walk into a bar

In a town the Doorman of a bar must choose who he lets in so that there will be no feuds between them. We represent the people by vertices and the feuds by edges between them.

Vertex Cover

Given a graph G=(V,E) find the min opt number of vertices to delete so there will be no edges left in the graph.

Official Parameterized Version

Input: A Graph G=(V,E) Parameter: A Positive Integer k
Output: Does G have a VC of size k?

Since this is an optimisation problem as we already mentioned we usually use as a parameter the size of the solution.

Some guys walk into a bar

In a town the Doorman of a bar must choose who he lets in so that there will be no feuds between them. We represent the people by vertices and the feuds by edges between them.

Vertex Cover

Given a graph G=(V,E) find the min opt number of vertices to delete so there will be no edges left in the graph.

Official Parameterized Version

Input: A Graph G=(V,E) Parameter: A Positive Integer k
Output: Does G have a VC of size k?

Since this is an optimisation problem as we already mentioned we usually use as a parameter the size of the solution.

Outline

Solving Vertex Cover

Main Idea

The running time or an algorithms usually explodes when there is branching affected by the size. Make the branching bound by the parameter and we will have the requested time bound. .

Lets try an algorithm.

1. Begin with the root node labelled by zero. (Represents the an empty VC containing none of V)
2. For one edge (uv) of the graph corresponding to this levels each node branch tho children one containing u and one containing v and label accordingly.
3. update the graph by each time deleting the node's label vertices.
4. repeat k times

Solving Vertex Cover

Main Idea

The running time or an algorithms usually explodes when there is branching affected by the size. Make the branching bound by the parameter and we will have the requested time bound. .

Lets try an algorithm.

1. Begin with the root node labelled by zero. (Represents the an empty VC containing none of V)
2. For one edge (uv) of the graph corresponding to this levels each node branch tho children one containing u and one containing v and label accordingly.
3. update the graph by each time deleting the node's label vertices.
4. repeat k times

Solving Vertex Cover

Main Idea

The running time or an algorithms usually explodes when there is branching affected by the size. Make the branching bound by the parameter and we will have the requested time bound. .

Lets try an algorithm.

1. Begin with the root node labelled by zero. (Represents the an empty VC containing none of V)
2. For one edge (uv) of the graph corresponding to this levels each node branch tho children one containing u and one containing v and label accordingly.
3. update the graph by each time deleting the node's label vertices.
4. repeat k times

Solving Vertex Cover

Main Idea

The running time or an algorithms usually explodes when there is branching affected by the size. Make the branching bound by the parameter and we will have the requested time bound. .

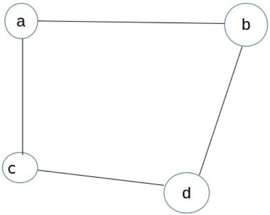
Lets try an algorithm.

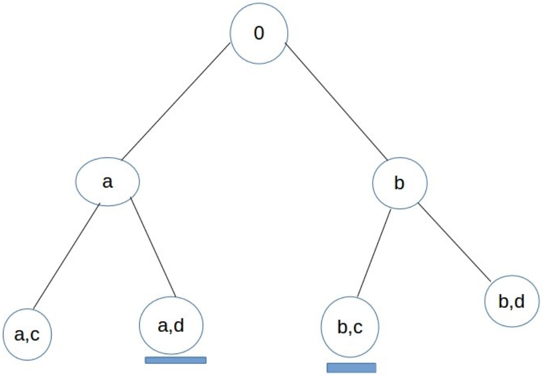
1. Begin with the root node labelled by zero. (Represents the an empty VC containing none of V)
2. For one edge (uv) of the graph corresponding to this levels each node branch tho children one containing u and one containing v and label accordingly.
3. update the graph by each time deleting the node's label vertices.
4. repeat k times

Example

We will check a Graph for a VC of size 2

G:





Cont.

What me manage this way :

- Resulting tree is of depth k
- Each level i of resulting tree T has nodes with exactly i vertices in the label
- If there is a leaf that by deleting its label's vertices from G there is no edge left in G then this set of vertices is a VC of size k.

Why is the above procedure correct? yes! For each edge we have to delete at least one end at some point. Since we explore both options if there is a VC of size k this algorithm will find it .

Outline

2 Elementary techniques

- Bounded search trees
- Kernelization

Find the reason the problem is hard

Lets consider again the example of the bar.

1. The Doorman only wants to forbid k people from entering. If someone has more that k feuds he has to go (otherwise we would have to send away his k neighbours)
2. If someone has no feuds then he enters without checking
3. If someone has only one feud remove his neighbour from the set

Are those enough to result to a FPT algorithm? YES!
With only one iteration (O(n) time) we are left with vertices of degree 1,2,..,k-1
How many simple graphs are there with these degrees?
We need a VC of size k which means we have at most k^2 edges (otherwize there is no VC of size k)

Find the reason the problem is hard

Lets consider again the example of the bar.

1. The Doorman only wants to forbid k people from entering. If someone has more that k feuds he has to go (otherwise we would have to send away his k neighbours)
2. If someone has no feuds then he enters without checking
3. If someone has only one feud remove his neighbour from the set

Are those enough to result to a FPT algorithm? YES!
With only one iteration (O(n) time) we are left with vertices of degree 1,2,..,k-1
How many simple graphs are there with these degrees?
We need a VC of size k which means we have at most k^2 edges (otherwize there is no VC of size k)

Find the reason the problem is hard

Lets consider again the example of the bar.

1. The Doorman only wants to forbid k people from entering. If someone has more that k feuds he has to go (otherwise we would have to send away his k neighbours)
2. If someone has no feuds then he enters without checking
3. If someone has only one feud remove his neighbour from the set

Are those enough to result to a FPT algorithm? YES!
With only one iteration (O(n) time) we are left with vertices of degree 1,2,..,k-1
How many simple graphs are there with these degrees?
We need a VC of size k which means we have at most k^2 edges (otherwize there is no VC of size k)

Find the reason the problem is hard

Lets consider again the example of the bar.

1. The Doorman only wants to forbid k people from entering. If someone has more that k feuds he has to go (otherwise we would have to send away his k neighbours)
2. If someone has no feuds then he enters without checking
3. If someone has only one feud remove his neighbour from the set

Are those enough to result to a FPT algorithm? YES!
With only one iteration (O(n) time) we are left with vertices of degree 1,2,..,k-1
How many simple graphs are there with these degrees?
We need a VC of size k which means we have at most k^2 edges (otherwize there is no VC of size k)

Find the reason the problem is hard

Lets consider again the example of the bar.

1. The Doorman only wants to forbid k people from entering. If someone has more that k feuds he has to go (otherwise we would have to send away his k neighbours)
2. If someone has no feuds then he enters without checking
3. If someone has only one feud remove his neighbour from the set

Are those enough to result to a FPT algorithm? YES!
With only one iteration (O(n) time) we are left with vertices of degree 1,2,..,k-1
How many simple graphs are there with these degrees?
We need a VC of size k which means we have at most k^2 edges (otherwize there is no VC of size k)

Brute Force the Small kernel

Our instance now is of a size expresed only with respect to k

So what do we do?

1. We could use brute force. In the FPT framework we are already "fast"
2. Run a Bounded Depth First Search on the k^2 plausible sets to find a VC set. (Previous Technique)

Of course when using the reduction rule (1) we have to decrease our k accordingly . This doesn't change the complexity result.

Note for Reference

There is a parameterized algorithm that solves VC in $O(1.2738^k + kn)$ time. Extremely usefully in computational biology or small towns with only one bar.

Brute Force the Small kernel

Our instance now is of a size expresed only with respect to k

So what do we do?

1. We could use brute force. In the FPT framework we are already "fast"
2. Run a Bounded Depth First Search on the k^2 plausible sets to find a VC set. (Previous Technique)

Of course when using the reduction rule (1) we have to decrease our k accordingly . This doesn't change the complexity result.

Note for Reference

There is a parameterized algorithm that solves VC in $O(1.2738^k + kn)$ time. Extremely usefully in computational biology or small towns with only one bar.

Brute Force the Small kernel

Our instance now is of a size expresed only with respect to k

So what do we do?

1. We could use brute force. In the FPT framework we are already "fast"
2. Run a Bounded Depth First Search on the k^2 plausible sets to find a VC set. (Previous Technique)

Of course when using the reduction rule (1) we have to decrease our k accordingly . This doesn't change the complexity result.

Note for Reference

There is a parameterized algorithm that solves VC in $O(1.2738^k + kn)$ time. Extremely usefully in computational biology or small towns with only one bar.

Brute Force the Small kernel

Our instance now is of a size expresed only with respect to k

So what do we do?

1. We could use brute force. In the FPT framework we are already "fast"
2. Run a Bounded Depth First Search on the k^2 plausible sets to find a VC set. (Previous Technique)

Of course when using the reduction rule (1) we have to decrease our k accordingly . This doesn't change the complexity result.

Note for Reference

There is a parameterized algorithm that solves VC in $O(1.2738^k + kn)$ time. Extremely usefully in computational biology or small towns with only one bar.

Outline

Heavy tools

Until now our tools have basically been all about utilising extra knowledge and structure in order to construct better algorithms.

But we have other more interesting Programming techniques to build effective algorithms.

What happens when say we combine the fixed parameter approach with the notion of dynamic programming?

Parameterized Smaller Instance

To Use Dynamic programming we have to be able to express the solution of the current instance as an optimal solution of smaller ones. We will see how this is done when the smaller instances are defined by a bound parameter.

Heavy tools

Until now our tools have basically been all about utilising extra knowledge and structure in order to construct better algorithms.

But we have other more interesting Programming techniques to build effective algorithms.

What happens when say we combine the fixed parameter approach with the notion of dynamic programming?

Parameterized Smaller Instance

To Use Dynamic programming we have to be able to express the solution of the current instance as an optimal solution of smaller ones. We will see how this is done when the smaller instances are defined by a bound parameter.

Heavy tools

Until now our tools have basically been all about utilising extra knowledge and structure in order to construct better algorithms.

But we have other more interesting Programming techniques to build effective algorithms.

What happens when say we combine the fixed parameter approach with the notion of dynamic programming?

Parameterized Smaller Instance

To Use Dynamic programming we have to be able to express the solution of the current instance as an optimal solution of smaller ones. We will see how this is done when the smaller instances are defined by a bound parameter.

Heavy tools

Until now our tools have basically been all about utilising extra knowledge and structure in order to construct better algorithms.
But we have other more interesting Programming techniques to build effective algorithms.
What happens when say we combine the fixed parameter approach with the notion of dynamic programming?

Parameterized Smaller Instance

To Use Dynamic programming we have to be able to express the solution of the current instance as an optimal solution of smaller ones. We will see how this is done when the smaller instances are defined by a bound parameter.

Outline

3 Teqniques based on graph structure
- Treewidth and tree decomposition
- Dynamic programming

Other Metrics

There are many ways to parameterize a problem.

- you can parameterize by rank of a Matrix
- by the ecentricity of a vertex
- by the dencity of a graph

BUT! : As we mentioned you have to be sure that by assuming the parameter bound and small you are not ignoring important or common instances of a problem.

What have we came up with? Treewidth! .

Treewidth is a graph metric we use to define in a way how far a graph is from a tree.

And why do we do that? Because most graph properties are extremely easy to be checked on Trees

Other Metrics

There are many ways to parameterize a problem.

- you can parameterize by rank of a Matrix
- by the ecentricity of a vertex
- by the dencity of a graph

BUT! : As we mentioned you have to be sure that by assuming the parameter bound and small you are not ignoring important or common instances of a problem.

What have we came up with? Treewidth! .

Treewidth is a graph metric we use to define in a way how far a graph is from a tree.

And why do we do that? Because most graph properties are extremely easy to be checked on Trees

Other Metrics

There are many ways to parameterize a problem.

- you can parameterize by rank of a Matrix
- by the ecentricity of a vertex
- by the dencity of a graph

BUT! : As we mentioned you have to be sure that by assuming the parameter bound and small you are not ignoring important or common instances of a problem.

What have we came up with? Treewidth! .

Treewidth is a graph metric we use to define in a way how far a graph is from a tree.

And why do we do that? Because most graph properties are extremely easy to be checked on Trees

Other Metrics

There are many ways to parameterize a problem.

- you can parameterize by rank of a Matrix
- by the ecentricity of a vertex
- by the dencity of a graph

BUT! : As we mentioned you have to be sure that by assuming the parameter bound and small you are not ignoring important or common instances of a problem.

What have we came up with? Treewidth! .

Treewidth is a graph metric we use to define in a way how far a graph is from a tree.

And why do we do that? Because most graph properties are extremely easy to be checked on Trees

Other Metrics

There are many ways to parameterize a problem.

- you can parameterize by rank of a Matrix
- by the ecentricity of a vertex
- by the dencity of a graph

BUT! : As we mentioned you have to be sure that by assuming the parameter bound and small you are not ignoring important or common instances of a problem.

What have we came up with? Treewidth! .

Treewidth is a graph metric we use to define in a way how far a graph is from a tree.

And why do we do that? Because most graph properties are extremely easy to be checked on Trees

Treewidth

Definitions

1. A Tree decomposition of a graph G=(V,E) is a tree T together with a collection of subsets T_x (called bags) of V labelled with the vertices x of T such that $\cup T_x = V$ and the following hold
   - For every edge uv of G there is a some x such that u,v $\in T_x$
   - If y is a vertex of on the unique path in T from x to z then $T_x \cap T_z \subseteq T_y$
2. The width of a tree decomposition is the maximum value of $|T_x|$ -1 over all the vertices of the tree T of the decomposition.
3. The treewidth of Graph G is the minimum treewidth of all thee decompositions of G.

Treewidth

Definitions

1. A Tree decomposition of a graph G=(V,E) is a tree T together with a collection of subsets T_x (called bags) of V labelled with the vertices x of T such that $\cup T_x = V$ and the following hold
   - For every edge uv of G there is a some x such that u,v $\in T_x$
   - If y is a vertex of on the unique path in T from x to z then $T_x \cap T_z \subseteq T_y$
2. The width of a tree decomposition is the maximum value of $|T_x|$ -1 over all the vertices of the tree T of the decomposition.
3. The treewidth of Graph G is the minimum treewidth of all thee decompositions of G.

Treewidth

Definitions

1. A Tree decomposition of a graph G=(V,E) is a tree T together with a collection of subsets T_x (called bags) of V labelled with the vertices x of T such that $\cup T_x = V$ and the following hold
   - For every edge uv of G there is a some x such that u,v $\in T_x$
   - If y is a vertex of on the unique path in T from x to z then $T_x \cap T_z \subseteq T_y$
2. The width of a tree decomposition is the maximum value of $|T_x|$ -1 over all the vertices of the tree T of the decomposition.
3. The treewidth of Graph G is the minimum treewidth of all thee decompositions of G.

Treewidth

Definitions

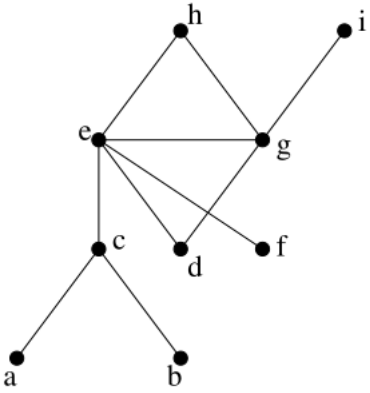
1. A Tree decomposition of a graph G=(V,E) is a tree T together with a collection of subsets T_x (called bags) of V labelled with the vertices x of T such that $\cup T_x = V$ and the following hold
   - For every edge uv of G there is a some x such that u,v $\in T_x$
   - If y is a vertex of on the unique path in T from x to z then $T_x \cap T_z \subseteq T_y$
2. The width of a tree decomposition is the maximum value of $|T_x|$ -1 over all the vertices of the tree T of the decomposition.
3. The treewidth of Graph G is the minimum treewidth of all thee decompositions of G.

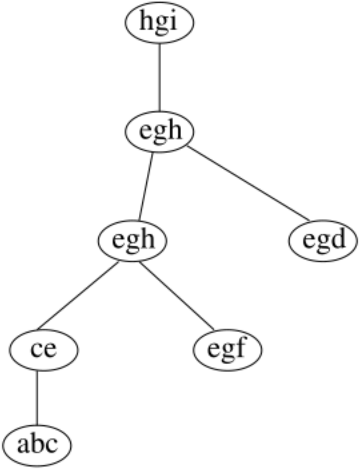
Treewidth

Definitions

1. A Tree decomposition of a graph G=(V,E) is a tree T together with a collection of subsets T_x (called bags) of V labelled with the vertices x of T such that $\cup T_x = V$ and the following hold
   - For every edge uv of G there is a some x such that u,v $\in T_x$
   - If y is a vertex of on the unique path in T from x to z then $T_x \cap T_z \subseteq T_y$
2. The width of a tree decomposition is the maximum value of $|T_x|$ -1 over all the vertices of the tree T of the decomposition.
3. The treewidth of Graph G is the minimum treewidth of all thee decompositions of G.

Example





Outline

3 Teqniques based on graph structure
- Treewidth and tree decomposition
- Dynamic programming

Independent Set

Given a Graph G find the maximum Set L such that if $u, v \in L$ then uv $\notin E$

This Problem is NP-hard

BUT!: The kinds or real-world problems that require us to check this property have bounded treewidth! So:

Independent Set

Given a Graph G find the maximum Set L such that if $u, v \in L$ then uv $\notin E$

This Problem is NP-hard

BUT!: The kinds or real-world problems that require us to check this property have bounded treewidth! So:

Dynamic Programming on Bounded TW

The Algorithm

1. Given a Graph and a tree Decomposition of tw k. (we will use the one given in the previous example)
2. For each node of T we construct a vector with 2^k positions as follows

Ø	a	b	c	ab	ac	bc	abc
0	1	1	1	2	–	–	–

We store in each position of the vector the size of the larger Independent set this far. That is the size of the set corresponding to the vectors bit plus the size of the previously larger Independent set for the vectors already filled.

Cont.

Of course we are careful if the current bit of the vector has common vertices with the previous max independent set . But we only have to make this check for adjacent nodes of T .
Continuing by adding up independent sets for empty leaf nodes we get the Max independent set.
We Can pause here and try it for the above tree decomposition.
The proof of this algorithm can be found in the literature given later.

Outline

Optimisation

So what have we learned?

- We have found more efficient ways to solve decision problems using parameters.
- The problems we tried this on are NP-optimisation ones.

What is the correlation? Do the algorithms we described remain efficient?
We will try to formulate this
What is the FPT situation of the above problem?

Optimisation

So what have we learned?

- We have found more efficient ways to solve decision problems using parameters.
- The problems we tried this on are NP-optimisation ones.

What is the correlation? Do the algorithms we described remain efficient?

We will try to formulate this

What is the FPT situation of the above problem?

Optimisation

So what have we learned?

- We have found more efficient ways to solve decision problems using parameters.
- The problems we tried this on are NP-optimisation ones.

What is the correlation? Do the algorithms we described remain efficient?

We will try to formulate this

What is the FPT situation of the above problem?

Optimisation

So what have we learned?

- We have found more efficient ways to solve decision problems using parameters.
- The problems we tried this on are NP-optimisation ones.

What is the correlation? Do the algorithms we described remain efficient?

We will try to formulate this

What is the FPT situation of the above problem?

Optimisation

So what have we learned?

- We have found more efficient ways to solve decision problems using parameters.
- The problems we tried this on are NP-optimisation ones.

What is the correlation? Do the algorithms we described remain efficient?
We will try to formulate this
What is the FPT situation of the above problem?

Optimisation

So what have we learned?

- We have found more efficient ways to solve decision problems using parameters.
- The problems we tried this on are NP-optimisation ones.

What is the correlation? Do the algorithms we described remain efficient?
We will try to formulate this

DECISION PROBLEM ASSOCIATED WITH AN NP OPTIMIZATION PROBLEM $Q = (I_Q, S_Q, f_Q, opt_Q)$.

Input: $x \in I_Q$.
Parameter: A positive integer k.
Question: Does $R(opt_Q(x), k)$ hold?

What is the FPT situation of the above problem?

Cont

Thanks to Parameterized Complexity Theory we have the following

Theorem. Cai and Chen

Iff you can check if the decision version of an NP optimisation problem in FPT time then you can find the optimal in FPT time.

Can you think of a proof?

This is not the actual formulation of the theorem

Cont

Thanks to Parameterized Complexity Theory we have the following

Theorem. Cai and Chen

Iff you can check if the decision version of an NP optimisation problem in FPT time then you can find the optimal in FPT time.

Can you think of a proof?

This is not the actual formulation of the theorem

Cont

Thanks to Parameterized Complexity Theory we have the following

Theorem. Cai and Chen

Iff you can check if the decision version of an NP optimisation problem in FPT time then you can find the optimal in FPT time.

Can you think of a proof?

This is not the actual formulation of the theorem

FPT and Appoximation

1. Pick your favourite optimisation Problem (Preferably one from the lessons you had on approximation algorithms)
2. Make it into a Decision one and then parameterize it by the size of the solution
3. Is it FTP?

Theorem : Bazgan , Cai and Chen

If a NP-optimisation Problem has a fully Polynomial time approximation scheme then it is FPT

And we are going to prove this

FPT and Appoximation

1. Pick your favourite optimisation Problem (Preferably one from the lessons you had on approximation algorithms)
2. Make it into a Decision one and then parameterize it by the size of the solution
3. Is it FTP?

Theorem : Bazgan , Cai and Chen

If a NP-optimisation Problem has a fully Polynomial time approximation scheme then it is FPT

And we are going to prove this

FPT and Appoximation

1. Pick your favourite optimisation Problem (Preferably one from the lessons you had on approximation algorithms)
2. Make it into a Decision one and then parameterize it by the size of the solution
3. Is it FTP?

Theorem : Bazgan , Cai and Chen

If a NP-optimisation Problem has a fully Polynomial time approximation scheme then it is FPT

And we are going to prove this

FPT and Appoximation

1. Pick your favourite optimisation Problem (Preferably one from the lessons you had on approximation algorithms)
2. Make it into a Decision one and then parameterize it by the size of the solution
3. Is it FTP?

Theorem : Bazgan , Cai and Chen

If a NP-optimisation Problem has a fully Polynomial time approximation scheme then it is FPT

And we are going to prove this

FPT and Appoximation

1. Pick your favourite optimisation Problem (Preferably one from the lessons you had on approximation algorithms)
2. Make it into a Decision one and then parameterize it by the size of the solution
3. Is it FTP?

Theorem : Bazgan , Cai and Chen

If a NP-optimisation Problem has a fully Polynomial time approximation scheme then it is FPT

And we are going to prove this

Proof

1. W.L.G Say the problem is a maximisation one
2. Since it has a fully PTAS there is an algorithm A that runs in time $O(p((1/e) * |x|))$ and approximates it by an error of e .
3. we only need to prove that the decision version is FPT
4. For an instance $< x, k >$ run A for $< x, 1/2k >$

   What is the running time of this? Find e with respect to k?

5.
   - if $k < f(x)$ then $x < opt(x)$ since this is a maximisation problem
   - if $k > f(x)$ then $x - 1 \geq f(x)$, but $e < k \Rightarrow k > opt(x)$

Proof

1. W.L.G Say the problem is a maximisation one
2. Since it has a fully PTAS there is an algorithm A that runs in time $O(p((1/e) * |x|))$ and approximates it by an error of e .
3. we only need to prove that the decision version is FPT
4. For an instance $< x, k >$ run A for $< x, 1/2k >$

   What is the running time of this? Find e with respect to k?

5.
   - if $k < f(x)$ then $x < opt(x)$ since this is a maximisation problem
   - if $k > f(x)$ then $x - 1 \geq f(x)$, but $e < k \Rightarrow k > opt(x)$

Proof

1. W.L.G Say the problem is a maximisation one
2. Since it has a fully PTAS there is an algorithm A that runs in time $O(p((1/e) * |x|))$ and approximates it by an error of e .
3. we only need to prove that the decision version is FPT
4. For an instance $< x, k >$ run A for $< x, 1/2k >$

   What is the running time of this? Find e with respect to k?

5.
   - if $k < f(x)$ then $x < opt(x)$ since this is a maximisation problem
   - if $k > f(x)$ then $x - 1 \geq f(x)$, but $e < k \Rightarrow k > opt(x)$

Proof

1. W.L.G Say the problem is a maximisation one
2. Since it has a fully PTAS there is an algorithm A that runs in time $O(p((1/e) * |x|))$ and approximates it by an error of e .
3. we only need to prove that the decision version is FPT
4. For an instance $< x, k >$ run A for $< x, 1/2k >$

   What is the running time of this? Find e with respect to k?

5. - if $k < f(x)$ then $x < opt(x)$ since this is a maximisation problem
   - if $k > f(x)$ then $x - 1 \geq f(x)$, but $e < k \Rightarrow k > opt(x)$

Proof

1. W.L.G Say the problem is a maximisation one
2. Since it has a fully PTAS there is an algorithm A that runs in time $O(p((1/e) * |x|))$ and approximates it by an error of e .
3. we only need to prove that the decision version is FPT
4. For an instance $< x, k >$ run A for $< x, 1/2k >$

   What is the running time of this? Find e with respect to k?

5. - if $k < f(x)$ then $x < opt(x)$ since this is a maximisation problem
   - if $k > f(x)$ then $x - 1 \geq f(x)$, but $e < k \Rightarrow k > opt(x)$

Proof

1. W.L.G Say the problem is a maximisation one
2. Since it has a fully PTAS there is an algorithm A that runs in time $O(p((1/e) * |x|))$ and approximates it by an error of e .
3. we only need to prove that the decision version is FPT
4. For an instance $< x, k >$ run A for $< x, 1/2k >$

   What is the running time of this? Find e with respect to k?

5. - if $k < f(x)$ then $x < opt(x)$ since this is a maximisation problem
   - if $k > f(x)$ then $x - 1 \geq f(x)$, but $e < k \Rightarrow k > opt(x)$

Proof Cont.

- Therefore $k < F(x)$ iff $k < opt(x)$
- A runs in time $O(p((2k) * |x|))$
- Therefore The Problem is FPT

Unfortunately a convex does not exist.
Given a FPT algorithm we cannot guarantee the existence of a fully time approximation Scheme .
However this allows us to establish that many problems are FPT without effort

For instance

- Bounded Knapsack
- Planar Independent Set
- Linear Extension Count

Are all FPT

Proof Cont.

- Therefore $k < F(x)$ iff $k < opt(x)$
- A runs in time $O(p((2k) * |x|))$
- Therefore The Problem is FPT

Unfortunately a convex does not exist.
Given a FPT algorithm we cannot guarantee the existence of a fully time approximation Scheme .
However this allows us to establish that many problems are FPT without effort

For instance

- Bounded Knapsack
- Planar Independent Set
- Linear Extension Count

Are all FPT

Proof Cont.

- Therefore $k < F(x)$ iff $k < opt(x)$
- A runs in time $O(p((2k) * |x|))$
- Therefore The Problem is FPT

Unfortunately a convex does not exist.
Given a FPT algorithm we cannot guarantee the existence of a fully time approximation Scheme .
However this allows us to establish that many problems are FPT without effort

For instance

- Bounded Knapsack
- Planar Independent Set
- Linear Extension Count

Are all FPT

Proof Cont.

- Therefore $k < F(x)$ iff $k < opt(x)$
- A runs in time $O(p((2k) * |x|))$
- Therefore The Problem is FPT

Unfortunately a convex does not exist.

Given a FPT algorithm we cannot guarantee the existence of a fully time approximation Scheme .

However this allows us to establish that many problems are FPT without effort

For instance

- Bounded Knapsack
- Planar Independent Set
- Linear Extension Count

Are all FPT

Proof Cont.

- Therefore $k < F(x)$ iff $k < opt(x)$
- A runs in time $O(p((2k) * |x|))$
- Therefore The Problem is FPT

Unfortunately a convex does not exist.
Given a FPT algorithm we cannot guarantee the existence of a fully time approximation Scheme .
However this allows us to establish that many problems are FPT without effort

For instance

- Bounded Knapsack
- Planar Independent Set
- Linear Extension Count

Are all FPT

Proof Cont.

- Therefore $k < F(x)$ iff $k < opt(x)$
- A runs in time $O(p((2k) * |x|))$
- Therefore The Problem is FPT

Unfortunately a convex does not exist.
Given a FPT algorithm we cannot guarantee the existence of a fully time approximation Scheme .
However this allows us to establish that many problems are FPT without effort

For instance

- Bounded Knapsack
- Planar Independent Set
- Linear Extension Count

Are all FPT

Approximation

- Say though that a NP-problem is Fixed parameter Intractable.
- So what do we do?
- Can we approximate? Pause for suspense..

Approximation

- Say though that a NP-problem is Fixed parameter Intractable.
- So what do we do?
- Can we approximate? Pause for suspense..

Approximation

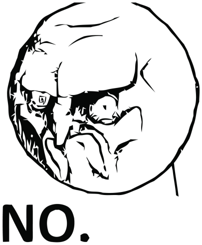
- Say though that a NP-problem is Fixed parameter Intractable.
- So what do we do?
- Can we approximate? Pause for suspense..

Approximation

- Say though that a NP-problem is Fixed parameter Intractable.
- So what do we do?
- Can we approximate? Pause for suspense..

Approximation

- Say though that a NP-problem is Fixed parameter Intractable.
- So what do we do?
- Can we approximate? Pause for suspense..



At least not efficiently

Theorem: Bazgan , Cai and Chen

If a NP-Optimisation Problem is Fixed Parameter Intractable then it has no fully polynomial time approximation Scheme.
To be more specific under the equivalent of The $P! = NP$ for parameterized problems There is not fully PTAS for any "NP" problems.

Only known problem that is "NP" hard but not NP hard is the VC dimension proven to be "NP"-hard by Papadimitriou and Yanakakis

At least not efficiently

Theorem: Bazgan , Cai and Chen

If a NP-Optimisation Problem is Fixed Parameter Intractable then it has no fully polynomial time approximation Scheme. To be more specific under the equivalent of The $P! = NP$ for parameterized problems There is not fully PTAS for any "NP" problems.

Only known problem that is "NP" hard but not NP hard is the VC dimension proven to be "NP"-hard by Papadimitriou and Yanakakis

At least not efficiently

Theorem: Bazgan , Cai and Chen

If a NP-Optimisation Problem is Fixed Parameter Intractable then it has no fully polynomial time approximation Scheme. To be more specific under the equivalent of The $P! = NP$ for parameterized problems There is not fully PTAS for any "NP" problems.

Only known problem that is "NP" hard but not NP hard is the VC dimension proven to be "NP"-hard by Papadimitriou and Yanakakis


That's all Folks!
Any Questions?

References

Fundamentals of Parameterized Complexity
Rodney G. Downey , Michael R. Fellows

Ellie Anastasiadi
anastasiadi.elli0@gmail.com