Efficient Sequential Algorithms, Comp309

University of Liverpool

2010–2011

Module Organiser, Igor Potapov

Part 5: Approximation Algorithms and Complexity

References:

G. Ausiello, P. Crescenzi, G. Gambosi, V. Kann, A. Marchetti-Spaccamela, M. Protasi, *Complexity and Approximation* Springer 2003.

R. Motwani, Lecture Notes on Approximation Algorithms - Volume 1, Stanford University 1992

Coping with hard computational problems

A large number of the optimisation problems, including those that we need to solve in practice, are NP-hard.

These problems are unlikely to have an efficient algorithm.

Nevertheless, we still need to solve the problems.

If $\mathrm{P} \neq \mathrm{NP}$, we cannot find algorithms which will find optimal solutions to all instances in polynomial time.

There are three possibilities for relaxing the requirements.

Coping with hard computational problems

A large number of the optimisation problems, including those that we need to solve in practice, are NP-hard.

These problems are unlikely to have an efficient algorithm.

Nevertheless, we still need to solve the problems.

If $\mathrm{P} \neq \mathrm{NP}$, we cannot find algorithms which will find optimal solutions to all instances in polynomial time.

There are three possibilities for relaxing the requirements.

Coping with hard computational problems

A large number of the optimisation problems, including those that we need to solve in practice, are NP-hard.

These problems are unlikely to have an efficient algorithm.

Nevertheless, we still need to solve the problems.

If $\mathrm{P} \neq \mathrm{NP}$, we cannot find algorithms which will find optimal solutions to all instances in polynomial time.

There are three possibilities for relaxing the requirements.

Coping with hard computational problems

A large number of the optimisation problems, including those that we need to solve in practice, are NP-hard.

These problems are unlikely to have an efficient algorithm.

Nevertheless, we still need to solve the problems.

If $\mathrm{P} \neq \mathrm{NP}$, we cannot find algorithms which will find optimal solutions to all instances in polynomial time.

There are three possibilities for relaxing the requirements.

Heuristics

Do not require polynomial time.

Sometimes (but not very often!) we can use techniques such as branch-and-bound and dynamic programming to come up with algorithms which are not much worse than polynomial.

Heuristics

Do not require polynomial time.

Sometimes (but not very often!) we can use techniques such as branch-and-bound and dynamic programming to come up with algorithms which are not much worse than polynomial.

Probabilistic analysis

Do not require success on all instances.

Sometimes (but no so often!) we have information about the probability distribution from which inputs are chosen.
Sometimes we can find a polynomial-time algorithm that finds an optimal solution with high probability, when an input is chosen from the distribution.

Approximation algorithms

Do not require an optimal solution.

Sometimes we can design a polynomial-time algorithm that is guaranteed to produce a solution that is not much worse than the best solution.

An optimisation problem P is defined by four components $(\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ where:

(1) $\mathcal{I}$ is the set of the instances of P.

(2) For each $x \in \mathcal{I}$, $\mathcal{S}(x)$ is the set of feasible solutions associated with x.

(3) For each $x \in \mathcal{I}$ and each $y \in \mathcal{S}(x)$, $\mathbf{v}(x, y)$ is a positive integer, which is the value of solution y for instance x.

(4) $\mathrm{goal} \in \{\max, \min\}$ is the optimisation criterion and tells if the problem P is a maximisation or a minimisation problem.

An optimisation problem P is defined by four components $(\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ where:

(1) $\mathcal{I}$ is the set of the instances of P.

(2) For each $x \in \mathcal{I}$, $\mathcal{S}(x)$ is the set of feasible solutions associated with x.

(3) For each $x \in \mathcal{I}$ and each $y \in \mathcal{S}(x)$, $\mathbf{v}(x, y)$ is a positive integer, which is the value of solution y for instance x.

(4) $\mathrm{goal} \in \{\max, \min\}$ is the optimisation criterion and tells if the problem P is a maximisation or a minimisation problem.

An optimisation problem P is defined by four components $(\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ where:

(1) $\mathcal{I}$ is the set of the instances of P.

(2) For each $x \in \mathcal{I}$, $\mathcal{S}(x)$ is the set of feasible solutions associated with x.

(3) For each $x \in \mathcal{I}$ and each $y \in \mathcal{S}(x)$, $\mathbf{v}(x, y)$ is a positive integer, which is the value of solution y for instance x.

(4) $\mathrm{goal} \in \{\max, \min\}$ is the optimisation criterion and tells if the problem P is a maximisation or a minimisation problem.

An optimisation problem P is defined by four components $(\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ where:

(1) $\mathcal{I}$ is the set of the instances of P.

(2) For each $x \in \mathcal{I}$, $\mathcal{S}(x)$ is the set of feasible solutions associated with x.

(3) For each $x \in \mathcal{I}$ and each $y \in \mathcal{S}(x)$, $\mathbf{v}(x, y)$ is a positive integer, which is the value of solution y for instance x.

(4) $\mathrm{goal} \in \{\max, \min\}$ is the optimisation criterion and tells if the problem P is a maximisation or a minimisation problem.

An optimisation problem P is defined by four components $(\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ where:

(1) $\mathcal{I}$ is the set of the instances of P.

(2) For each $x \in \mathcal{I}$, $\mathcal{S}(x)$ is the set of feasible solutions associated with x.

(3) For each $x \in \mathcal{I}$ and each $y \in \mathcal{S}(x)$, $\mathbf{v}(x, y)$ is a positive integer, which is the value of solution y for instance x.

(4) $\mathrm{goal} \in \{\max, \min\}$ is the optimisation criterion and tells if the problem P is a maximisation or a minimisation problem.

An optimisation problem P is defined by four components $(\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ where:

(1) $\mathcal{I}$ is the set of the instances of P.

(2) For each $x \in \mathcal{I}$, $\mathcal{S}(x)$ is the set of feasible solutions associated with x.

(3) For each $x \in \mathcal{I}$ and each $y \in \mathcal{S}(x)$, $\mathbf{v}(x, y)$ is a positive integer, which is the value of solution y for instance x.

(4) $\mathrm{goal} \in \{\max, \min\}$ is the optimisation criterion and tells if the problem P is a maximisation or a minimisation problem.

For an instance $x \in \mathcal{I}$, we use the notation $\mathbf{v}^*(x)$ to denote the value of the optimal solution in $\mathcal{S}(x)$.

If $\mathrm{goal} = \min$ then $\mathbf{v}^*(x) = \min\{\mathbf{v}(x, y) \mid y \in \mathcal{S}(x)\}$.

If $\mathrm{goal} = \max$ then $\mathbf{v}^*(x) = \max\{\mathbf{v}(x, y) \mid y \in \mathcal{S}(x)\}$.

Example: Minimum Vertex Cover

$\mathcal{I}$ is the set of undirected graphs.

For every $G \in \mathcal{I}$, $\mathcal{S}(G)$ is the set of vertex covers of G.

(Recall that a vertex cover is a set $U \subseteq V(G)$ such that every edge of G has at least one endpoint in U.)

The value $\mathbf{v}(G, U)$ is the size of U.

Finally, this is a minimisation problem, so $\mathrm{goal} = \min$.

Example: Minimum Vertex Cover

$\mathcal{I}$ is the set of undirected graphs.

For every $G \in \mathcal{I}$, $\mathcal{S}(G)$ is the set of vertex covers of G.

(Recall that a vertex cover is a set $U \subseteq V(G)$ such that every edge of G has at least one endpoint in U.)

The value $\mathbf{v}(G, U)$ is the size of U.

Finally, this is a minimisation problem, so $\mathrm{goal} = \min$.

Example: Minimum Vertex Cover

$\mathcal{I}$ is the set of undirected graphs.

For every $G \in \mathcal{I}$, $\mathcal{S}(G)$ is the set of vertex covers of G.

(Recall that a vertex cover is a set $U \subseteq V(G)$ such that every edge of G has at least one endpoint in U.)

The value $\mathbf{v}(G, U)$ is the size of U.

Finally, this is a minimisation problem, so $\mathrm{goal} = \min$.

Example: Minimum Vertex Cover

$\mathcal{I}$ is the set of undirected graphs.

For every $G \in \mathcal{I}$, $\mathcal{S}(G)$ is the set of vertex covers of G.

(Recall that a vertex cover is a set $U \subseteq V(G)$ such that every edge of G has at least one endpoint in U.)

The value $\mathbf{v}(G, U)$ is the size of U.

Finally, this is a minimisation problem, so $\mathrm{goal} = \min$.

Example: Minimum Vertex Cover

$\mathcal{I}$ is the set of undirected graphs.

For every $G \in \mathcal{I}$, $\mathcal{S}(G)$ is the set of vertex covers of G.

(Recall that a vertex cover is a set $U \subseteq V(G)$ such that every edge of G has at least one endpoint in U.)

The value $\mathbf{v}(G, U)$ is the size of U.

Finally, this is a minimisation problem, so $\mathrm{goal} = \min$.

Small generalisation

We have said that for an instance x and a solution y the value $\mathbf{v}(x, y)$ should be an integer. Sometimes it is useful to generalise the framework a little bit and allow $\mathbf{v}(x, y)$ to be a rational.

This generalisation does not matter because we could transform a problem with rational values into one with integer values.

Example: Minimum Bin Packing

Here is an example where values are integers, but instances involve rationals. We are given a collection of items, each with an associated size, which is a number between 0 and 1. We are required to pack the items into size-1 bins so as to minimise the number of bins used. Thus, we have the following minimisation problem.

An instance in $\mathcal{I}$ is a multiset $I = \{s_1, s_2, \ldots, s_n\}$ such that $\forall i, s_i \in (0, 1]$.

A solution in $\mathcal{S}(I)$ is a disjoint partition $P = \{B_1, B_2, \ldots, B_k\}$ of I so that for all parts B_j, $\sum_{s_i \in B_j} s_i \leq 1$.

The value $\mathbf{v}(I, P)$ is k.

For example, the instance $\{0.1, 0.8, 0.3, 0.5\}$ has a solution with $B_1 = \{0.1, 0.8\}$ and $B_2 = \{0.3, 0.5\}$.

Example: Minimum Bin Packing

Here is an example where values are integers, but instances involve rationals. We are given a collection of items, each with an associated size, which is a number between 0 and 1. We are required to pack the items into size-1 bins so as to minimise the number of bins used. Thus, we have the following minimisation problem.

An instance in $\mathcal{I}$ is a multiset $I = \{s_1, s_2, \ldots, s_n\}$ such that $\forall i, s_i \in (0, 1]$.

A solution in $\mathcal{S}(I)$ is a disjoint partition $P = \{B_1, B_2, \ldots, B_k\}$ of I so that for all parts B_j, $\sum_{s_i \in B_j} s_i \leq 1$.

The value $\mathbf{v}(I, P)$ is k.

For example, the instance $\{0.1, 0.8, 0.3, 0.5\}$ has a solution with $B_1 = \{0.1, 0.8\}$ and $B_2 = \{0.3, 0.5\}$.

Example: Minimum Bin Packing

Here is an example where values are integers, but instances involve rationals. We are given a collection of items, each with an associated size, which is a number between 0 and 1. We are required to pack the items into size-1 bins so as to minimise the number of bins used. Thus, we have the following minimisation problem.

An instance in $\mathcal{I}$ is a multiset $I = \{s_1, s_2, \ldots, s_n\}$ such that $\forall i, s_i \in (0, 1]$.

A solution in $\mathcal{S}(I)$ is a disjoint partition $P = \{B_1, B_2, \ldots, B_k\}$ of I so that for all parts B_j, $\sum_{s_i \in B_j} s_i \leq 1$.

The value $\mathbf{v}(I, P)$ is k.

For example, the instance $\{0.1, 0.8, 0.3, 0.5\}$ has a solution with $B_1 = \{0.1, 0.8\}$ and $B_2 = \{0.3, 0.5\}$.

Example: Minimum Bin Packing

Here is an example where values are integers, but instances involve rationals. We are given a collection of items, each with an associated size, which is a number between 0 and 1. We are required to pack the items into size-1 bins so as to minimise the number of bins used. Thus, we have the following minimisation problem.

An instance in $\mathcal{I}$ is a multiset $I = \{s_1, s_2, \ldots, s_n\}$ such that $\forall i, s_i \in (0, 1]$.

A solution in $\mathcal{S}(I)$ is a disjoint partition $P = \{B_1, B_2, \ldots, B_k\}$ of I so that for all parts B_j, $\sum_{s_i \in B_j} s_i \leq 1$.

The value $\mathbf{v}(I, P)$ is k.

For example, the instance $\{0.1, 0.8, 0.3, 0.5\}$ has a solution with $B_1 = \{0.1, 0.8\}$ and $B_2 = \{0.3, 0.5\}$.

Example: Minimum Bin Packing

Here is an example where values are integers, but instances involve rationals. We are given a collection of items, each with an associated size, which is a number between 0 and 1. We are required to pack the items into size-1 bins so as to minimise the number of bins used. Thus, we have the following minimisation problem.

An instance in $\mathcal{I}$ is a multiset $I = \{s_1, s_2, \ldots, s_n\}$ such that $\forall i, s_i \in (0, 1]$.

A solution in $\mathcal{S}(I)$ is a disjoint partition $P = \{B_1, B_2, \ldots, B_k\}$ of I so that for all parts B_j, $\sum_{s_i \in B_j} s_i \leq 1$.

The value $\mathbf{v}(I, P)$ is k.

For example, the instance $\{0.1, 0.8, 0.3, 0.5\}$ has a solution with $B_1 = \{0.1, 0.8\}$ and $B_2 = \{0.3, 0.5\}$.

Decision Problems

An optimisation problem P with components $(\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ can be associated with a corresponding decision problem P_D.

If $\mathrm{goal} = \max$, the corresponding decision problem is as follows: Given an instance $x \in \mathcal{I}$ and a positive integer K, decide whether $\mathbf{v}^*(x) \geq K$.

If $\mathrm{goal} = \min$, the corresponding decision problem is as follows: Given an instance $x \in \mathcal{I}$ and a positive integer K, decide whether $\mathbf{v}^*(x) \leq K$.

The class of NPO optimisation problems

The following complexity class is analogous to the class NP of decision problems.

An optimisation problem $P = (\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ is in NPO if the following holds.

(1) The set $\mathcal{I}$ of instances is recognizable in polynomial time.

(2) There is a polynomial q so that, for every instance $x \in \mathcal{I}$ and any feasible solution $y \in \mathcal{S}(x)$, we have $|y| \leq q(|x|)$. Also, it is decidable in polynomial time whether $y \in \mathcal{S}(x)$.

(3) $\mathbf{v}(x, y)$ is computable in polynomial time.

The class of NPO optimisation problems

The following complexity class is analogous to the class NP of decision problems.

An optimisation problem $P = (\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ is in NPO if the following holds.

(1) The set $\mathcal{I}$ of instances is recognizable in polynomial time.

(2) There is a polynomial q so that, for every instance $x \in \mathcal{I}$ and any feasible solution $y \in \mathcal{S}(x)$, we have $|y| \leq q(|x|)$. Also, it is decidable in polynomial time whether $y \in \mathcal{S}(x)$.

(3) $\mathbf{v}(x, y)$ is computable in polynomial time.

The class of NPO optimisation problems

The following complexity class is analogous to the class NP of decision problems.

An optimisation problem $P = (\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ is in NPO if the following holds.

(1) The set $\mathcal{I}$ of instances is recognizable in polynomial time.

(2) There is a polynomial q so that, for every instance $x \in \mathcal{I}$ and any feasible solution $y \in \mathcal{S}(x)$, we have $|y| \leq q(|x|)$. Also, it is decidable in polynomial time whether $y \in \mathcal{S}(x)$.

(3) $\mathbf{v}(x, y)$ is computable in polynomial time.

The class of NPO optimisation problems

The following complexity class is analogous to the class NP of decision problems.

An optimisation problem $P = (\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ is in NPO if the following holds.

(1) The set $\mathcal{I}$ of instances is recognizable in polynomial time.

(2) There is a polynomial q so that, for every instance $x \in \mathcal{I}$ and any feasible solution $y \in \mathcal{S}(x)$, we have $|y| \leq q(|x|)$. Also, it is decidable in polynomial time whether $y \in \mathcal{S}(x)$.

(3) $\mathbf{v}(x, y)$ is computable in polynomial time.

Example: Minimum Vertex Cover

Minimum Vertex Cover is in NPO, since

(1) The set of instances (undirected graphs) is recognizable in polynomial time.

(2) If $G = (V, E)$ is an instance with n vertices then a feasible solution is a vertex cover, which is a subset U of V. Checking whether U is a vertex cover can be done in polynomial time.

(3) $\mathbf{v}(G, U)$ is just the size of U, which can easily be computed in polynomial time.

The reason for our interest in the class NPO is this.

If P is in NPO then the corresponding decision problem P_D is in NP.

Proof: Suppose that P is a minimisation problem. Given an instance $x \in \mathcal{I}$ and an integer K, we can solve P_D by performing the following nondeterministic algorithm.

Guess a string y with $|y| \leq q(|x|)$. Check whether $y \in \mathcal{S}(x)$. Compute $\mathbf{v}(x, y)$. If $\mathbf{v}(x, y) \leq K$, output "yes". Otherwise, output "No".

(Output "no" if $y \notin \mathcal{S}(x)$. Also output "no" if $\mathbf{v}(x, y) > K$.)

An optimisation problem $P = (\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ is in PO if it is in NPO and there is a polynomial-time algorithm $\mathcal{A}$ that, for any instance $x \in \mathcal{I}$, computes a feasible solution $y \in \mathcal{S}(x)$ with $\mathbf{v}(x, y) = \mathbf{v}^*(x)$.

It is not known whether $PO = NPO$. This question is equivalent to P=NP, so instead of dwelling on this, we will turn to approximation algorithms for problems in NPO.

An optimisation problem $P = (\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ is in PO if it is in NPO and there is a polynomial-time algorithm $\mathcal{A}$ that, for any instance $x \in \mathcal{I}$, computes a feasible solution $y \in \mathcal{S}(x)$ with $\mathbf{v}(x, y) = \mathbf{v}^*(x)$.

It is not known whether $PO = NPO$. This question is equivalent to P=NP, so instead of dwelling on this, we will turn to approximation algorithms for problems in NPO.

Approximation algorithms

Given an optimisation problem $P = (\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$, an algorithm $\mathcal{A}$ is an approximation algorithm for P if, for any given instance $x \in \mathcal{I}$, it returns a feasible solution $\mathcal{A}(x) \in \mathcal{S}(x)$.

We will be interested in polynomial-time approximation algorithms.

A performance guarantee for an approximation algorithm tells us how far the value of the approximate solution is from the value of an optimal one.

There are several kinds of performance guarantees.

Approximation algorithms

Given an optimisation problem $P = (\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$, an algorithm $\mathcal{A}$ is an approximation algorithm for P if, for any given instance $x \in \mathcal{I}$, it returns a feasible solution $\mathcal{A}(x) \in \mathcal{S}(x)$.

We will be interested in polynomial-time approximation algorithms.

A performance guarantee for an approximation algorithm tells us how far the value of the approximate solution is from the value of an optimal one.

There are several kinds of performance guarantees.

Approximation algorithms

Given an optimisation problem $P = (\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$, an algorithm $\mathcal{A}$ is an approximation algorithm for P if, for any given instance $x \in \mathcal{I}$, it returns a feasible solution $\mathcal{A}(x) \in \mathcal{S}(x)$.

We will be interested in polynomial-time approximation algorithms.

A performance guarantee for an approximation algorithm tells us how far the value of the approximate solution is from the value of an optimal one.

There are several kinds of performance guarantees.

Absolute Performance Guarantees

Given an optimisation problem P, for any instance x and any feasible solution y of x, the absolute error of y with respect to x is defined as

$$D(x, y) = |\mathbf{v}^*(x) - \mathbf{v}(x, y)|.$$

Given an optimisation problem P and an approximation algorithm $\mathcal{A}$ for P, we say that $\mathcal{A}$ is an absolute approximation algorithm if there exists a constant k such that, for every instance x of P, $D(x, \mathcal{A}(x)) \leq k$.

Absolute Performance Guarantees

Given an optimisation problem P, for any instance x and any feasible solution y of x, the absolute error of y with respect to x is defined as

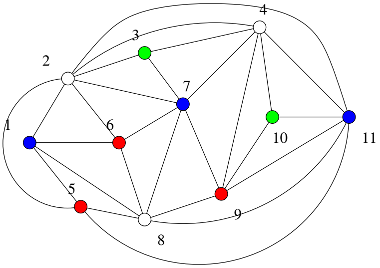
$$D(x, y) = |\mathbf{v}^*(x) - \mathbf{v}(x, y)|.$$

Given an optimisation problem P and an approximation algorithm $\mathcal{A}$ for P, we say that $\mathcal{A}$ is an absolute approximation algorithm if there exists a constant k such that, for every instance x of P, $D(x, \mathcal{A}(x)) \leq k$.

Example 1. Colouring Planar Graphs

A proper vertex colouring of a graph G is a function from $V(G)$ to the set of colours such that no two adjacent vertices have the same colour.

The chromatic number $\chi(G)$ is the minimum number of colours needed to colour G.



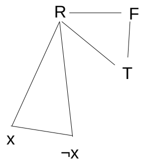
Determining whether a graph is k-colourable is NP-complete

To show that it is NP-hard to determine whether a graph is 3-colourable, we will give a polynomial-time reduction from 3-CNF.

Suppose that F is a boolean Formula expressed as an AND of clauses, each of which is the OR of exactly three distinct literals.

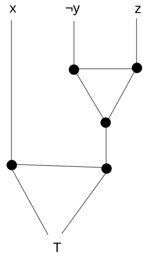
We will show a polynomial-time construction of a graph G which has is 3-colourable iff F is satisfiable.

Add vertices R, T, and F connected like this. For each variable x, add vertices x and $\neg x$, connected to R like this.



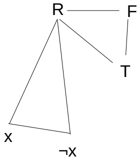
In any 3-colouring, let r, t and f be the colours of vertices R, T and F, respectively. Note that x and $\neg x$ get colours t and f (in some order).

For each clause $x \vee \neg y \vee z$ add a subgraph with five new (unnamed) vertices connected like this.



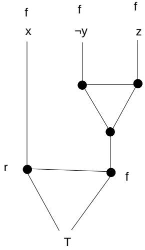
Note that this cannot be 3-coloured if all 3 literals in the clause are f. Otherwise, it can. (See next few slides.)

Add vertices R, T, and F connected like this. For each variable x, add vertices x and $\neg x$, connected to R like this.



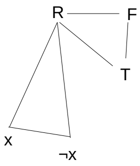
In any 3-colouring, let r, t and f be the colours of vertices R, T and F, respectively. Note that x and $\neg x$ get colours t and f (in some order).

For each clause $x \vee \neg y \vee z$ add a subgraph with five new (unnamed) vertices connected like this.

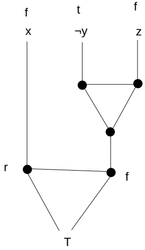


Note that this cannot be 3-coloured if all 3 literals in the clause are f. Otherwise, it can. (See next few slides.)

Add vertices R, T, and F connected like this. For each variable x, add vertices x and $\neg x$, connected to R like this.



For each clause $x \vee \neg y \vee z$ add a subgraph with five new (unnamed) vertices connected like this.

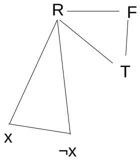


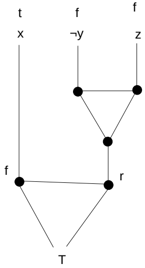
In any 3-colouring, let r, t and f be the colours of vertices R, T and F, respectively. Note that x and $\neg x$ get colours t and f (in some order).

Note that this cannot be 3-coloured if all 3 literals in the clause are f. Otherwise, it can. (See next few slides.)

Add vertices R, T, and F connected like this. For each variable x, add vertices x and $\neg x$, connected to R like this.

For each clause $x \vee \neg y \vee z$ add a subgraph with five new (unnamed) vertices connected like this.

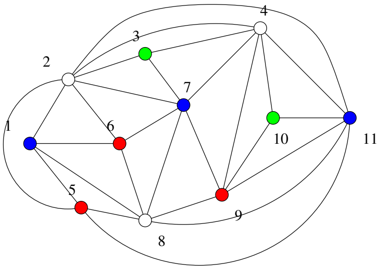




In any 3-colouring, let r, t and f be the colours of vertices R, T and F, respectively. Note that x and $\neg x$ get colours t and f (in some order).

Note that this cannot be 3-coloured if all 3 literals in the clause are f. Otherwise, it can. (See next few slides.)

We will not prove it here, but it is NP-hard to determine whether a graph is 3-colourable even when the graph is planar.



A planar graph is a graph that can be drawn on the plane with every vertex distinct, and no edges crossing.

The (planar) Vertex Colouring optimisation problem

$\mathcal{I} =$ planar graphs

For every planar graph G,

$$\mathcal{S}(G) = \{\sigma \mid \sigma \text{ is a proper colouring of } G\}$$

$\mathbf{v}(G, \sigma) =$ number of colours used in σ

$\mathrm{goal} = \min$

An approximation algorithm

Fact: Every planar graph has a vertex of degree at most 5

The proof of this fact is no more difficult than some other things we have proved in this module, but it is a bit of a digression. We will simply use the fact, without proving it.

We will give an approximation algorithm $\mathcal{A}$ that takes an input G and returns a colouring σ using at most 5 colours Thus, the absolute error is

$$D(G, \mathcal{A}(G)) = \mathbf{v}(G, \mathcal{A}(G)) - \mathbf{v}^*(G) \leq 5 - 1 = 4.$$

An approximation algorithm

Fact: Every planar graph has a vertex of degree at most 5

The proof of this fact is no more difficult than some other things we have proved in this module, but it is a bit of a digression. We will simply use the fact, without proving it.

We will give an approximation algorithm $\mathcal{A}$ that takes an input G and returns a colouring σ using at most 5 colours Thus, the absolute error is

$$D(G, \mathcal{A}(G)) = \mathbf{v}(G, \mathcal{A}(G)) - \mathbf{v}^*(G) \leq 5 - 1 = 4.$$

The algorithm

If the graph has only one vertex, give it a colour.

Otherwise, If the graph is disconnected, then recursively colour each component.

Otherwise, let v be a vertex of degree at most 5. Recursively colour $G - v$, and let σ be the resulting colouring. If the neighbours of v are coloured with at most 4 colours then choose some other colour for v.

Otherwise, let $v_1, \ldots, v_5$ be the neighbours of v and let $c_1, \ldots, c_5$ be their respective colours in σ. We will modify σ to obtain a colouring of $G \ldots$

The algorithm

If the graph has only one vertex, give it a colour.

Otherwise, If the graph is disconnected, then recursively colour each component.

Otherwise, let v be a vertex of degree at most 5. Recursively colour $G - v$, and let σ be the resulting colouring. If the neighbours of v are coloured with at most 4 colours then choose some other colour for v.

Otherwise, let $v_1, \ldots, v_5$ be the neighbours of v and let $c_1, \ldots, c_5$ be their respective colours in σ. We will modify σ to obtain a colouring of $G \ldots$

The algorithm

If the graph has only one vertex, give it a colour.

Otherwise, If the graph is disconnected, then recursively colour each component.

Otherwise, let v be a vertex of degree at most 5. Recursively colour $G - v$, and let σ be the resulting colouring. If the neighbours of v are coloured with at most 4 colours then choose some other colour for v.

Otherwise, let $v_1, \ldots, v_5$ be the neighbours of v and let $c_1, \ldots, c_5$ be their respective colours in σ. We will modify σ to obtain a colouring of $G \ldots$

The algorithm

If the graph has only one vertex, give it a colour.

Otherwise, If the graph is disconnected, then recursively colour each component.

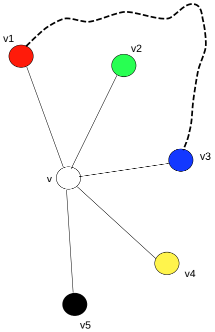
Otherwise, let v be a vertex of degree at most 5. Recursively colour $G - v$, and let σ be the resulting colouring. If the neighbours of v are coloured with at most 4 colours then choose some other colour for v.

Otherwise, let $v_1, \ldots, v_5$ be the neighbours of v and let $c_1, \ldots, c_5$ be their respective colours in σ. We will modify σ to obtain a colouring of $G \ldots$

Let G_{13} be the subgraph of $G \setminus v$ induced by the vertices coloured c_1 and c_3.

If v_1 and v_3 belong to different components of G_{13} then interchange the colours of the vertices in the component containing v_1. Vertex v can now be coloured c_1.

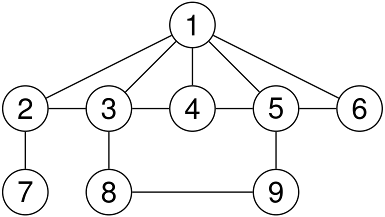
Otherwise, we have



Now just use colours c_2 and c_4 instead.
That is, since v_1 and v_3 belong to the same component, there exists a path P between v_1 and v_3 such that $P + v$ forms a cycle. This cycle cuts v_2 off from v_4. We can then complete the colouring using G_{24} and assigning c_2 to v.

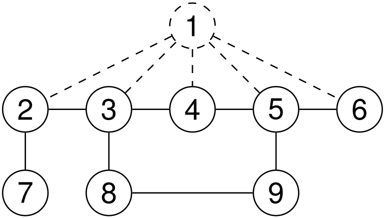
Note: It is important that we numbered the vertices $v_1, \ldots, v_1, \ldots, v_5$ in cyclical order (on the plane) so that v_2 gets cut off by the v_1-v_3 path.

example



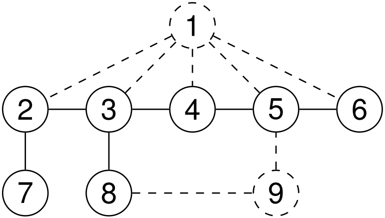
Vertex 1 has degree at most 5. Take it out and recursively colour $G - \{1\} \ldots$

Vertices to be added back and coloured: 1



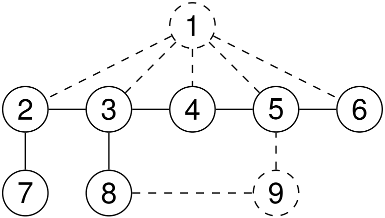
Vertex 9 has degree at most 5. Take it out and ecursively colour $G - \{9, 1\} \ldots$

Vertices to be added back and coloured: 9, 1



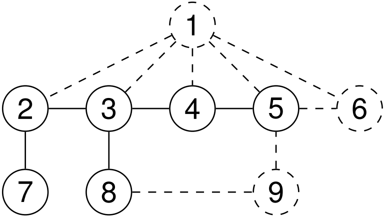
Vertex 6 has degree at most 5. Take it out and recursively colour $G - \{6, 9, 1\}$.

Vertices to be added back and coloured: $9, 1$



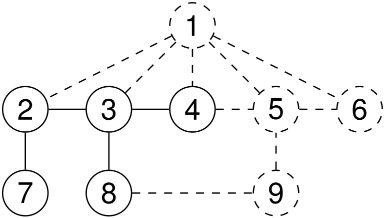
Vertex 6 has degree at most 5. Take it out and recursively colour $G - \{6, 9, 1\}$.

Vertices to be added back and coloured: $6, 9, 1$



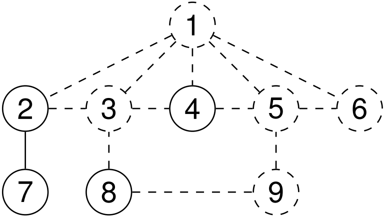
Vertex 5 has degree at most 5. Take it out and recursively colour $G - \{5, 6, 9, 1\}$.

Vertices to be added back and coloured: $5, 6, 9, 1$



Vertex 3 has degree at most 5. Take it out and recursively colour $G - \{3, 5, 6, 9, 1\}$.

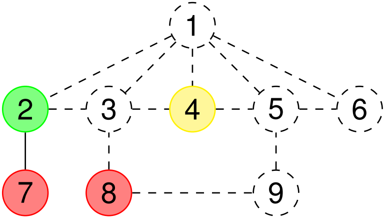
Vertices to be added back and coloured: $3, 5, 6, 9, 1$



At this point the graph splits into three components, which get coloured separately. Vertex 8 is isolated, so gets any colour, say red. Vertex 4 is isolated, so gets any colour, say yellow.

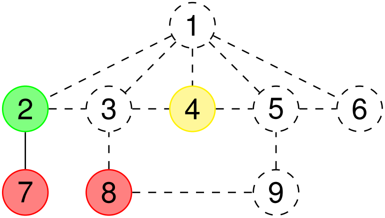
In the recursive colouring of the component $(2, 7)$, vertex 2 is removed, then vertex 7 is isolated and gets any colour, say red. Then vertex 2 is put back and coloured some other colour, say green.

Vertices to be added back and coloured: $3, 5, 6, 9, 1$



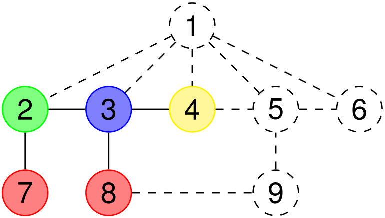
Now vertex 3 is put back and gets some colour other than red, green, or yellow, say blue.

Vertices to be added back and coloured: 3, 5, 6, 9, 1



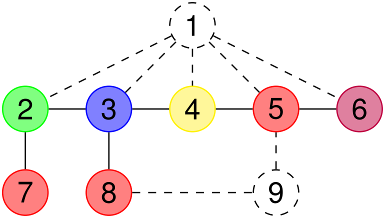
Now vertex 3 is put back and gets some colour other than red, green, or yellow, say blue.

Vertices to be added back and coloured: $5, 6, 9, 1$



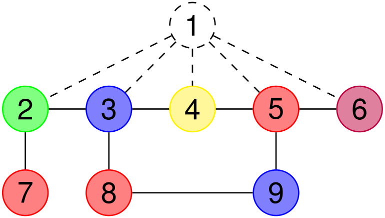
Now vertex 5 is put back and coloured some colour other than yellow, say red, and then vertex 6 is put back and coloured some colour other than red, say purple.

Vertices to be added back and coloured: 9, 1



Then vertex 9 is put back and is coloured some colour other than red, say blue.

Vertices to be added back and coloured: 1



Now, we would like to colour vertex 1, but all five colours are used at its neighbours. So we find two neighbours, say vertex 2 and vertex 4 which are not connected by a green-yellow path. We can swap green and yellow in the component of vertex 4, colouring vertex 4 green, and then we'll be able to colour vertex 1 yellow.

Improving the absolute error

The approximation algorithm $\mathcal{A}$ that we have just described takes as input a planar graph G and returns a proper colouring σ of G using at most 5 colours. Thus, the absolute error is

$$D(G, \mathcal{A}(G)) = \mathbf{v}(G, \mathcal{A}(G)) - \mathbf{v}^*(G) \leq 5 - 1 = 4.$$

How can the absolute error be improved?

Check first whether G can be coloured with 1 or 2 colours. If so, return an optimal colouring in polynomial time. If not, return a colouring using at most 5 colours. Then $D(G, \mathcal{A}(G)) \leq 5 - 3 = 2$.

Improving the absolute error

The approximation algorithm $\mathcal{A}$ that we have just described takes as input a planar graph G and returns a proper colouring σ of G using at most 5 colours. Thus, the absolute error is

$$D(G, \mathcal{A}(G)) = \mathbf{v}(G, \mathcal{A}(G)) - \mathbf{v}^*(G) \leq 5 - 1 = 4.$$

How can the absolute error be improved?

Check first whether G can be coloured with 1 or 2 colours. If so, return an optimal colouring in polynomial time. If not, return a colouring using at most 5 colours. Then $D(G, \mathcal{A}(G)) \leq 5 - 3 = 2$.

Improving the absolute error

The approximation algorithm $\mathcal{A}$ that we have just described takes as input a planar graph G and returns a proper colouring σ of G using at most 5 colours. Thus, the absolute error is

$$D(G, \mathcal{A}(G)) = \mathbf{v}(G, \mathcal{A}(G)) - \mathbf{v}^*(G) \leq 5 - 1 = 4.$$

How can the absolute error be improved?

Check first whether G can be coloured with 1 or 2 colours. If so, return an optimal colouring in polynomial time. If not, return a colouring using at most 5 colours. Then
$D(G, \mathcal{A}(G)) \leq 5 - 3 = 2$.

How do you check whether a graph is 2-colourable?

Depth-first search.

DFS_Colour(V,E)
    Forall $u \in V$
        $\text{colour}[u] \leftarrow \text{blank}$
    For all $u \in V$
        If $\text{colour}[u] = \text{blank}$
            Make_Blue(u)
    Return Yes

How do you check whether a graph is 2-colourable?

Depth-first search.

```
DFS_Colour(V,E)
   Forall u ∈ V
      colour[u] ← blank
   For all u ∈ V
      If colour[u] = blank
         Make_Blue(u)
   Return Yes
```

How do you check whether a graph is 2-colourable?

Depth-first search.

```
DFS_Colour(V,E)
   Forall u ∈ V
      colour[u] ← blank
   For all u ∈ V
      If colour[u] = blank
         Make_Blue(u)
   Return Yes
```

```
Make_Blue(u)
    colour[u] ← blue
    For all v ~ u
        If colour[v] = blue
            Return No
        If colour[v] = blank
            Make_Green(v)
```

```
Make_Green(u)
    colour[u] ← green
    For all v ~ u
        If colour[v] = green
            Return No
        If colour[v] = blank
            Make_Blue(v)
```

The algorithm for finding a 5-colouring in a planar graph is based on an argument by Kempe, which was an early attempt to prove the 4-colour theorem, which says that every planar graph can be coloured with just four colours. Kempe's argument had an error, but his ideas were useful and Appel and Haken eventually proved the theorem.

Example 2: edge colouring

We are given a graph and we want to colour its edges with the smallest possible number of colours such that no two adjacent edges have the same colour.

$\mathcal{I}$ = graphs

For every graph G,
$S(G) = \{\sigma \mid \sigma$ is a proper edge-colouring of $G\}$

$\mathbf{v}(G, \sigma)$ = number of colours used in σ

goal = min

Example 2: edge colouring

We are given a graph and we want to colour its edges with the smallest possible number of colours such that no two adjacent edges have the same colour.

$\mathcal{I}$ = graphs

For every graph G,
$S(G) = \{\sigma \mid \sigma$ is a proper edge-colouring of $G\}$

$\mathbf{v}(G, \sigma)$ = number of colours used in σ

goal = min

Example 2: edge colouring

We are given a graph and we want to colour its edges with the smallest possible number of colours such that no two adjacent edges have the same colour.

$\mathcal{I} =$ graphs

For every graph G,
$S(G) = \{\sigma \mid \sigma \text{ is a proper edge-colouring of } G\}$

$\mathbf{v}(G, \sigma) =$ number of colours used in σ

$\mathrm{goal} =$ min

Example 2: edge colouring

We are given a graph and we want to colour its edges with the smallest possible number of colours such that no two adjacent edges have the same colour.

$\mathcal{I}$ = graphs

For every graph G,
$S(G) = \{\sigma \mid \sigma$ is a proper edge-colouring of $G\}$

$\mathbf{v}(G, \sigma)$ = number of colours used in σ

$\mathrm{goal} = \min$

Example 2: edge colouring

We are given a graph and we want to colour its edges with the smallest possible number of colours such that no two adjacent edges have the same colour.

$\mathcal{I} =$ graphs

For every graph G,
$S(G) = \{\sigma \mid \sigma$ is a proper edge-colouring of $G\}$

$\mathbf{v}(G, \sigma) =$ number of colours used in σ

$\mathrm{goal} =$ min

Suppose that G has degree Δ. Then $\nu^*(G) \geq \Delta$.

Vizing has shown that there is a polynomial-time algorithm that takes as input a graph G and returns an edge colouring with at most $\Delta + 1$ colours, where Δ is the maximum degree of G.

Thus, we have an approximation algorithm $\mathcal{A}$ with absolute error

$$D(G, \mathcal{A}(G)) = \nu(G, \mathcal{A}(G)) - \nu^*(G) \leq \Delta + 1 - \Delta = 1.$$

Hoyler has shown that deciding whether a graph is edge-colourable with Δ colours is NP-complete.

Suppose that G has degree Δ. Then $\nu^*(G) \geq \Delta$.

Vizing has shown that there is a polynomial-time algorithm that takes as input a graph G and returns an edge colouring with at most $\Delta + 1$ colours, where Δ is the maximum degree of G.

Thus, we have an approximation algorithm $\mathcal{A}$ with absolute error

$$D(G, \mathcal{A}(G)) = \nu(G, \mathcal{A}(G)) - \nu^*(G) \leq \Delta + 1 - \Delta = 1.$$

Hoyler has shown that deciding whether a graph is edge-colourable with Δ colours is NP-complete.

Suppose that G has degree Δ. Then $\nu^*(G) \geq \Delta$.

Vizing has shown that there is a polynomial-time algorithm that takes as input a graph G and returns an edge colouring with at most $\Delta + 1$ colours, where Δ is the maximum degree of G.

Thus, we have an approximation algorithm $\mathcal{A}$ with absolute error

$$D(G, \mathcal{A}(G)) = \nu(G, \mathcal{A}(G)) - \nu^*(G) \leq \Delta + 1 - \Delta = 1.$$

Hoyler has shown that deciding whether a graph is edge-colourable with Δ colours is NP-complete.

Suppose that G has degree Δ. Then $\nu^*(G) \geq \Delta$.

Vizing has shown that there is a polynomial-time algorithm that takes as input a graph G and returns an edge colouring with at most $\Delta + 1$ colours, where Δ is the maximum degree of G.

Thus, we have an approximation algorithm $\mathcal{A}$ with absolute error

$$D(G, \mathcal{A}(G)) = \nu(G, \mathcal{A}(G)) - \nu^*(G) \leq \Delta + 1 - \Delta = 1.$$

Hoyler has shown that deciding whether a graph is edge-colourable with Δ colours is NP-complete.

Unfortunately, there are few problems for which there are absolute approximation algorithms.

For example, consider the Clique optimisation problem.

$\mathcal{I} =$ graphs

For every graph G, $S(G) = \{U \subseteq V(G) \mid$ every pair of vertices in U is connected by an edge$\}$

$\nu(G, U) = |U|$

$\mathrm{goal} = \max$

We have shown in the "NP-completeness" section that this optimisation problem is NP-hard.

Unfortunately, there are few problems for which there are absolute approximation algorithms.

For example, consider the Clique optimisation problem.

$\mathcal{I}$ = graphs

For every graph G, $S(G) = \{U \subseteq V(G) \mid$ every pair of vertices in U is connected by an edge$\}$

$\nu(G, U) = |U|$

$\mathrm{goal} =$ max

We have shown in the “NP-completeness” section that this optimisation problem is NP-hard.

Unfortunately, there are few problems for which there are absolute approximation algorithms.

For example, consider the Clique optimisation problem.

$\mathcal{I} =$ graphs

For every graph G, $S(G) = \{U \subseteq V(G) \mid$ every pair of vertices in U is connected by an edge$\}$

$\nu(G, U) = |U|$

$\text{goal} = \max$

We have shown in the "NP-completeness" section that this optimisation problem is NP-hard.

Unfortunately, there are few problems for which there are absolute approximation algorithms.

For example, consider the Clique optimisation problem.

$\mathcal{I} =$ graphs

For every graph G, $S(G) = \{U \subseteq V(G) \mid$ every pair of vertices in U is connected by an edge$\}$

$\nu(G, U) = |U|$

$\text{goal} = \max$

We have shown in the "NP-completeness" section that this optimisation problem is NP-hard.

Unfortunately, there are few problems for which there are absolute approximation algorithms.

For example, consider the Clique optimisation problem.

$\mathcal{I} =$ graphs

For every graph G, $S(G) = \{U \subseteq V(G) \mid$ every pair of vertices in U is connected by an edge$\}$

$\nu(G, U) = |U|$

$\mathrm{goal} = \max$

We have shown in the "NP-completeness" section that this optimisation problem is NP-hard.

Unfortunately, there are few problems for which there are absolute approximation algorithms.

For example, consider the Clique optimisation problem.

$\mathcal{I} =$ graphs

For every graph G, $S(G) = \{U \subseteq V(G) \mid$ every pair of vertices in U is connected by an edge$\}$

$\nu(G, U) = |U|$

$\text{goal} =$ max

We have shown in the "NP-completeness" section that this optimisation problem is NP-hard.

Unfortunately, there are few problems for which there are absolute approximation algorithms.

For example, consider the Clique optimisation problem.

$\mathcal{I} =$ graphs

For every graph G, $S(G) = \{U \subseteq V(G) \mid$
every pair of vertices in U is connected by an edge$\}$

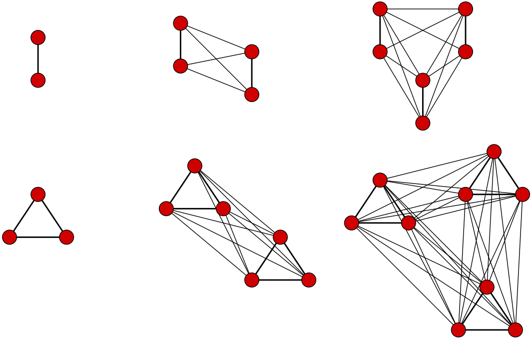
$\nu(G, U) = |U|$

$\mathrm{goal} =$ max

We have shown in the “NP-completeness” section that this optimisation problem is NP-hard.

Here is a tool that will use to show that the Clique optimisation problem has no absolute approximation algorithm (unless P=NP).

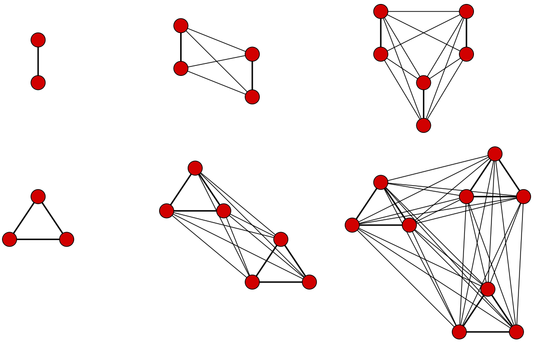
Define the *m-power of a graph G*, (written G^m) by taking m copies of G and connecting any two vertices that lie in different copies.



Claim. $\mathbf{v}^*(G^m) = m \cdot \mathbf{v}^*(G)$.

Here is a tool that will use to show that the Clique optimisation problem has no absolute approximation algorithm (unless P=NP).

Define the *m-power of a graph G*, (written G^m) by taking m copies of G and connecting any two vertices that lie in different copies.



Claim. $\mathbf{v}^*(G^m) = m \cdot \mathbf{v}^*(G)$.

Theorem: If there is an absolute approximation algorithm for the Clique problem then P=NP

Suppose $\mathcal{A}$ is an approximation algorithm with error k. Let $\mathcal{B}$ be an approximation that does the following. Given a graph G, run $\mathcal{A}$ on input G^{k+1} and let C be the resulting clique. Using C, find a clique of size at least $1/(k+1)$ times as large as C in G.

Note that $\mathbf{v}(G, \mathcal{B}(G)) \geq \mathbf{v}(G^{k+1}, \mathcal{A}(G^{k+1}))/(k+1)$.

Theorem: If there is an absolute approximation algorithm for the Clique problem then P=NP

Suppose $\mathcal{A}$ is an approximation algorithm with error k. Let $\mathcal{B}$ be an approximation that does the following. Given a graph G, run $\mathcal{A}$ on input G^{k+1} and let C be the resulting clique. Using C, find a clique of size at least $1/(k+1)$ times as large as C in G.

Note that $\mathbf{v}(G, \mathcal{B}(G)) \geq \mathbf{v}(G^{k+1}, \mathcal{A}(G^{k+1}))/(k+1)$.

$$
\begin{aligned}
\mathbf{v}^*(G) - \mathbf{v}(G, \mathcal{B}(G)) &\leq \mathbf{v}^*(G) - \frac{\mathbf{v}(G^{k+1}, \mathcal{A}(G^{k+1}))}{k+1} \\
&= \frac{(k+1)\mathbf{v}^*(G) - \mathbf{v}(G^{k+1}, \mathcal{A}(G^{k+1}))}{k+1} \\
&= \frac{\mathbf{v}^*(G^{k+1}) - \mathbf{v}(G^{k+1}, \mathcal{A}(G^{k+1}))}{k+1} \\
&\leq \frac{k}{k+1} < 1 = 0,
\end{aligned}
$$

since these quantities are integers.

So we cannot have an absolute approximation algorithm for the clique optimisation problem unless P=NP (in which case we can solve it exactly). Similar results exist for most other problems.

Relative Performance Guarantees

Example: Multiprocessor scheduling

The input consists of n jobs, $J_1, \ldots, J_n$.

Job J_i has a corresponding runtime p_i (a rational number).

The jobs are to be distributed between m identical machines

The finish-time is the maximum, over machines M, of the total runtime of jobs assigned to M.

The goal is to minimise the finish-time.

Relative Performance Guarantees

Example: Multiprocessor scheduling

The input consists of n jobs, $J_1, \ldots, J_n$.

Job J_i has a corresponding runtime p_i (a rational number).

The jobs are to be distributed between m identical machines

The finish-time is the maximum, over machines M, of the total runtime of jobs assigned to M.

The goal is to minimise the finish-time.

Relative Performance Guarantees

Example: Multiprocessor scheduling

The input consists of n jobs, $J_1, \ldots, J_n$.

Job J_i has a corresponding runtime p_i (a rational number).

The jobs are to be distributed between m identical machines

The finish-time is the maximum, over machines M, of the total runtime of jobs assigned to M.

The goal is to minimise the finish-time.

Relative Performance Guarantees

Example: Multiprocessor scheduling

The input consists of n jobs, $J_1, \ldots, J_n$.

Job J_i has a corresponding runtime p_i (a rational number).

The jobs are to be distributed between m identical machines

The finish-time is the maximum, over machines M, of the total runtime of jobs assigned to M.

The goal is to minimise the finish-time.

Relative Performance Guarantees

Example: Multiprocessor scheduling

The input consists of n jobs, $J_1, \ldots, J_n$.

Job J_i has a corresponding runtime p_i (a rational number).

The jobs are to be distributed between m identical machines

The finish-time is the maximum, over machines M, of the total runtime of jobs assigned to M.

The goal is to minimise the finish-time.

An instance $I \in \mathcal{I}$ is a set of jobs $\{p_1, \ldots, p_n\}$.

A feasible solution in $S(I)$ is a partition B of $\{p_1, \ldots, p_n\}$ into m subsets $B_1, \ldots, B_m$.

The value of the partition $\mathbf{v}(I, B)$ is $\max_{j=1}^{m} \sum_{p_i \in B_j} p_i$.

$\mathrm{goal} =$ min.

This optimisation problem is known to be NP-hard even for the special case $m = 2$.

A greedy approximation algorithm

The following approximation algorithm is due to Graham, and is called the “List scheduling algorithm” (we will refer to it as LS).

Consider the n jobs one-by-one.

When a job is considered, pick the currently least-loaded machine, and assign the job to that machine.

We will show that the "relative performance" of algorithm LS is good in the sense that, for any instance x,

$$\frac{\mathbf{v}(x, \mathrm{LS}(x))}{\mathbf{v}^*(x)} \leq 2 - \frac{1}{m}.$$

Let M be the machine that ends up with the highest load. Let L be this load, namely $L = \mathbf{v}(x, \mathrm{LS}(x))$.

Let J_j be the last job assigned to machine M.

Every machine has load at least $L - p_j$. (Otherwise, J_j would have been given to a different machine.)

So $\sum_{i=1}^{n} p_i \geq m(L - p_j) + p_j$.

Also, $\mathbf{v}^*(x) \geq \frac{1}{m} \sum_{i=1}^{n} p_i$, since some machine gets at least the average load.

So $\mathbf{v}^*(x) \geq L - p_j + \frac{p_j}{m} = \mathbf{v}(x, \mathrm{LS}(x)) - (1 - \frac{1}{m})p_j$.

Let M be the machine that ends up with the highest load. Let L be this load, namely $L = \mathbf{v}(x, \mathrm{LS}(x))$.

Let J_j be the last job assigned to machine M.

Every machine has load at least $L - p_j$. (Otherwise, J_j would have been given to a different machine.)

So $\sum_{i=1}^{n} p_i \geq m(L - p_j) + p_j$.

Also, $\mathbf{v}^*(x) \geq \frac{1}{m} \sum_{i=1}^{n} p_i$, since some machine gets at least the average load.

So $\mathbf{v}^*(x) \geq L - p_j + \frac{p_j}{m} = \mathbf{v}(x, \mathrm{LS}(x)) - (1 - \frac{1}{m})p_j$.

Let M be the machine that ends up with the highest load. Let L be this load, namely $L = \mathbf{v}(x, \mathrm{LS}(x))$.

Let J_j be the last job assigned to machine M.

Every machine has load at least $L - p_j$. (Otherwise, J_j would have been given to a different machine.)

So $\sum_{i=1}^{n} p_i \geq m(L - p_j) + p_j$.

Also, $\mathbf{v}^*(x) \geq \frac{1}{m} \sum_{i=1}^{n} p_i$, since some machine gets at least the average load.

So $\mathbf{v}^*(x) \geq L - p_j + \frac{p_j}{m} = \mathbf{v}(x, \mathrm{LS}(x)) - (1 - \frac{1}{m})p_j$.

Let M be the machine that ends up with the highest load. Let L be this load, namely $L = \mathbf{v}(x, \mathrm{LS}(x))$.

Let J_j be the last job assigned to machine M.

Every machine has load at least $L - p_j$. (Otherwise, J_j would have been given to a different machine.)

So $\sum_{i=1}^{n} p_i \geq m(L - p_j) + p_j$.

Also, $\mathbf{v}^*(x) \geq \frac{1}{m} \sum_{i=1}^{n} p_i$, since some machine gets at least the average load.

So $\mathbf{v}^*(x) \geq L - p_j + \frac{p_j}{m} = \mathbf{v}(x, \mathrm{LS}(x)) - (1 - \frac{1}{m})p_j$.

Let M be the machine that ends up with the highest load. Let L be this load, namely $L = \mathbf{v}(x, \mathrm{LS}(x))$.

Let J_j be the last job assigned to machine M.

Every machine has load at least $L - p_j$. (Otherwise, J_j would have been given to a different machine.)

So $\sum_{i=1}^{n} p_i \geq m(L - p_j) + p_j$.

Also, $\mathbf{v}^*(x) \geq \frac{1}{m} \sum_{i=1}^{n} p_i$, since some machine gets at least the average load.

So $\mathbf{v}^*(x) \geq L - p_j + \frac{p_j}{m} = \mathbf{v}(x, \mathrm{LS}(x)) - (1 - \frac{1}{m})p_j$.

Let M be the machine that ends up with the highest load. Let L be this load, namely $L = \mathbf{v}(x, \mathrm{LS}(x))$.

Let J_j be the last job assigned to machine M.

Every machine has load at least $L - p_j$. (Otherwise, J_j would have been given to a different machine.)

So $\sum_{i=1}^{n} p_i \geq m(L - p_j) + p_j$.

Also, $\mathbf{v}^*(x) \geq \frac{1}{m} \sum_{i=1}^{n} p_i$, since some machine gets at least the average load.

So $\mathbf{v}^*(x) \geq L - p_j + \frac{p_j}{m} = \mathbf{v}(x, \mathrm{LS}(x)) - (1 - \frac{1}{m})p_j$.

Finally, we can rewrite

$$\mathbf{v}^*(x) \geq \mathbf{v}(x, \mathrm{LS}(x)) - \left(1 - \frac{1}{m}\right) p_j$$

as

$$\frac{\mathbf{v}(x, \mathrm{LS}(x))}{\mathbf{v}^*(x)} \leq 1 + \left(1 - \frac{1}{m}\right) \frac{p_j}{\mathbf{v}^*(x)},$$

and the right-hand-side is at most $1 + (1 - \frac{1}{m})$ since some processor has to take job J_j so $\mathbf{v}^*(x) \geq p_j$.

Finally, we can rewrite

$$\mathbf{v}^*(x) \geq \mathbf{v}(x, \mathrm{LS}(x)) - \left(1 - \frac{1}{m}\right) p_j$$

as

$$\frac{\mathbf{v}(x, \mathrm{LS}(x))}{\mathbf{v}^*(x)} \leq 1 + \left(1 - \frac{1}{m}\right) \frac{p_j}{\mathbf{v}^*(x)},$$

and the right-hand-side is at most $1 + (1 - \frac{1}{m})$ since some processor has to take job J_j so $\mathbf{v}^*(x) \geq p_j$.

We have shown that for any instance x,

$$\frac{\mathbf{v}(x, \mathrm{LS}(x))}{\mathbf{v}^*(x)} \leq 2 - \frac{1}{m}.$$

This bound cannot be improved — there is an instance x on which the algorithm really does this badly.

Here is one such instance x. Let $n = m(m-1) + 1$. The first $n-1$ jobs each have runtime 1 and the last job has $p_n = m$.

$\mathbf{v}^*(x) = m$ since $m-1$ of the machines share the $n-1$ runtime 1 jobs and $(n-1)/(m-1) = m$.

But LS gives each of the machines $(n-1)/m = m-1$ of the first $n-1$ jobs. The last job has to go somewhere, so $\mathbf{v}(x, \mathrm{LS}(x)) = 2m - 1$.

We have shown that for any instance x,

$$\frac{\mathbf{v}(x, \mathrm{LS}(x))}{\mathbf{v}^*(x)} \leq 2 - \frac{1}{m}.$$

This bound cannot be improved — there is an instance x on which the algorithm really does this badly.

Here is one such instance x. Let $n = m(m-1) + 1$. The first $n - 1$ jobs each have runtime 1 and the last job has $p_n = m$.

$\mathbf{v}^*(x) = m$ since $m - 1$ of the machines share the $n - 1$ runtime 1 jobs and $(n-1)/(m-1) = m$.

But LS gives each of the machines $(n-1)/m = m - 1$ of the first $n - 1$ jobs. The last job has to go somewhere, so $\mathbf{v}(x, \mathrm{LS}(x)) = 2m - 1$.

We have shown that for any instance x,

$$\frac{\mathbf{v}(x, \mathrm{LS}(x))}{\mathbf{v}^*(x)} \leq 2 - \frac{1}{m}.$$

This bound cannot be improved — there is an instance x on which the algorithm really does this badly.

Here is one such instance x. Let $n = m(m-1) + 1$. The first $n - 1$ jobs each have runtime 1 and the last job has $p_n = m$.

$\mathbf{v}^*(x) = m$ since $m - 1$ of the machines share the $n - 1$ runtime 1 jobs and $(n-1)/(m-1) = m$.

But LS gives each of the machines $(n-1)/m = m - 1$ of the first $n - 1$ jobs. The last job has to go somewhere, so $\mathbf{v}(x, \mathrm{LS}(x)) = 2m - 1$.

We have shown that for any instance x,

$$\frac{\mathbf{v}(x, \mathrm{LS}(x))}{\mathbf{v}^*(x)} \leq 2 - \frac{1}{m}.$$

This bound cannot be improved — there is an instance x on which the algorithm really does this badly.

Here is one such instance x. Let $n = m(m-1) + 1$. The first $n - 1$ jobs each have runtime 1 and the last job has $p_n = m$.

$\mathbf{v}^*(x) = m$ since $m - 1$ of the machines share the $n - 1$ runtime 1 jobs and $(n-1)/(m-1) = m$.

But LS gives each of the machines $(n-1)/m = m - 1$ of the first $n - 1$ jobs. The last job has to go somewhere, so $\mathbf{v}(x, \mathrm{LS}(x)) = 2m - 1$.

We have shown that for any instance x,

$$\frac{\mathbf{v}(x, \mathrm{LS}(x))}{\mathbf{v}^*(x)} \leq 2 - \frac{1}{m}.$$

Thus, we have measured the quality of algorithm LS in terms of the ratio between the value of its solution and the value of the optimal solution.

This is what is meant by a relative performance measure.

Performance ratio

Given an optimisation problem P, an instance x of P and a feasible solution y of x, the performance ratio of y with respect to x is defined as

$$R(x, y) = \max\left(\frac{\mathbf{v}(x, y)}{\mathbf{v}^*(x)}, \frac{\mathbf{v}^*(x)}{\mathbf{v}(x, y)}\right)$$

The performance ratio $R(x, y)$ is at least 1, and is equal to 1 iff y is optimal.

r-approximation algorithm

Given an optimisation problem P and an approximation algorithm $\mathcal{A}$ for P, we say that $\mathcal{A}$ is an r-approximation algorithm for P if, given any input instance x of P,

$$R(x, \mathcal{A}(x)) \leq r.$$

If P has an r-approximation algorithm then we say that it can be approximated with ratio r.

For example, we have seen that the list-scheduling algorithm is a $(2 - 1/m)$-approximation algorithm for the m-machine scheduling problem.

It is sometimes useful to generalise this definition.

r-approximation algorithm

Given an optimisation problem P and an approximation algorithm $\mathcal{A}$ for P, we say that $\mathcal{A}$ is an r-approximation algorithm for P if, given any input instance x of P,

$$R(x, \mathcal{A}(x)) \leq r.$$

If P has an r-approximation algorithm then we say that it can be approximated with ratio r.

For example, we have seen that the list-scheduling algorithm is a $(2 - 1/m)$-approximation algorithm for the m-machine scheduling problem.

It is sometimes useful to generalise this definition.

r-approximation algorithm

Given an optimisation problem P and an approximation algorithm $\mathcal{A}$ for P, we say that $\mathcal{A}$ is an r-approximation algorithm for P if, given any input instance x of P,

$$R(x, \mathcal{A}(x)) \leq r.$$

If P has an r-approximation algorithm then we say that it can be approximated with ratio r.

For example, we have seen that the list-scheduling algorithm is a $(2 - 1/m)$-approximation algorithm for the m-machine scheduling problem.

It is sometimes useful to generalise this definition.

r-approximation algorithm

Given an optimisation problem P and an approximation algorithm $\mathcal{A}$ for P, we say that $\mathcal{A}$ is an r-approximation algorithm for P if, given any input instance x of P,

$$R(x, \mathcal{A}(x)) \leq r.$$

If P has an r-approximation algorithm then we say that it can be approximated with ratio r.

For example, we have seen that the list-scheduling algorithm is a $(2 - 1/m)$-approximation algorithm for the m-machine scheduling problem.

It is sometimes useful to generalise this definition.

$r(n)$-approximation algorithm

Let $r : \mathbb{N} \to \mathbb{Q}$ be a function.

Given an optimisation problem P and an approximation algorithm $\mathcal{A}$ for P, we say that $\mathcal{A}$ is an $r(n)$-approximation algorithm for P if, given any input instance x of P,

$$R(x, \mathcal{A}(x)) \leq r(|x|).$$

Improvement

The LS algorithm can be improved by first sorting the jobs so that $p_1 \geq \cdots \geq p_n$. We will call the resulting algorithm LPT for "largest-processing time first".

LPT is a $(\frac{4}{3} - \frac{1}{3m})$-approximation algorithm.

We will not prove this, but we prove a slightly weaker result — we will prove that LPT is a $(\frac{3}{2} - \frac{1}{2m})$-approximation algorithm (so it is a $3/2$-approximation algorithm).

Proof

Using the exact same argument as before, we find that

$\mathbf{v}(x, \mathrm{LPT}(x)) \leq \mathbf{v}^*(x) + (1 - \frac{1}{m})p_j$, where job J_j is the last job to be assigned to machine M, which is a machine that ends up with the largest load, namely load $\mathbf{v}(x, \mathrm{LPT}(x))$.

But we can assume $j > m$ (otherwise the algorithm is optimal because J_j gets its own machine and $\mathbf{v}(x, \mathrm{LPT}(x)) = p_j$).

So $p_1 \geq p_2 \geq \cdots \geq p_{m+1} \geq p_j$.

But there must be two jobs from the first $m+1$ that share a machine (since we only have m machines) so $\mathbf{v}^*(x) \geq 2p_j$.

Then $\mathbf{v}(x, \mathrm{LPT}(x)) \leq \mathbf{v}^*(x) + (1 - \frac{1}{m})\frac{\mathbf{v}^*(x)}{2} = \mathbf{v}^*(x)(\frac{3}{2} - \frac{1}{2m})$.

Proof

Using the exact same argument as before, we find that

$\mathbf{v}(x, \mathrm{LPT}(x)) \leq \mathbf{v}^*(x) + (1 - \frac{1}{m})p_j$, where job J_j is the last job to be assigned to machine M, which is a machine that ends up with the largest load, namely load $\mathbf{v}(x, \mathrm{LPT}(x))$.

But we can assume $j > m$ (otherwise the algorithm is optimal because J_j gets its own machine and $\mathbf{v}(x, \mathrm{LPT}(x)) = p_j$).

So $p_1 \geq p_2 \geq \cdots \geq p_{m+1} \geq p_j$.

But there must be two jobs from the first $m + 1$ that share a machine (since we only have m machines) so $\mathbf{v}^*(x) \geq 2p_j$.

Then $\mathbf{v}(x, \mathrm{LPT}(x)) \leq \mathbf{v}^*(x) + (1 - \frac{1}{m})\frac{\mathbf{v}^*(x)}{2} = \mathbf{v}^*(x)(\frac{3}{2} - \frac{1}{2m})$.

Proof

Using the exact same argument as before, we find that

$\mathbf{v}(x, \mathrm{LPT}(x)) \leq \mathbf{v}^*(x) + (1 - \frac{1}{m})p_j$, where job J_j is the last job to be assigned to machine M, which is a machine that ends up with the largest load, namely load $\mathbf{v}(x, \mathrm{LPT}(x))$.

But we can assume $j > m$ (otherwise the algorithm is optimal because J_j gets its own machine and $\mathbf{v}(x, \mathrm{LPT}(x)) = p_j$).

So $p_1 \geq p_2 \geq \cdots \geq p_{m+1} \geq p_j$.

But there must be two jobs from the first $m + 1$ that share a machine (since we only have m machines) so $\mathbf{v}^*(x) \geq 2p_j$.

Then $\mathbf{v}(x, \mathrm{LPT}(x)) \leq \mathbf{v}^*(x) + (1 - \frac{1}{m})\frac{\mathbf{v}^*(x)}{2} = \mathbf{v}^*(x)(\frac{3}{2} - \frac{1}{2m})$.

Proof

Using the exact same argument as before, we find that

$\mathbf{v}(x, \mathrm{LPT}(x)) \leq \mathbf{v}^*(x) + (1 - \frac{1}{m})p_j$, where job J_j is the last job to be assigned to machine M, which is a machine that ends up with the largest load, namely load $\mathbf{v}(x, \mathrm{LPT}(x))$.

But we can assume $j > m$ (otherwise the algorithm is optimal because J_j gets its own machine and $\mathbf{v}(x, \mathrm{LPT}(x)) = p_j$).

So $p_1 \geq p_2 \geq \cdots \geq p_{m+1} \geq p_j$.

But there must be two jobs from the first $m + 1$ that share a machine (since we only have m machines) so $\mathbf{v}^*(x) \geq 2p_j$.

Then $\mathbf{v}(x, \mathrm{LPT}(x)) \leq \mathbf{v}^*(x) + (1 - \frac{1}{m})\frac{\mathbf{v}^*(x)}{2} = \mathbf{v}^*(x)(\frac{3}{2} - \frac{1}{2m})$.

Proof

Using the exact same argument as before, we find that

$\mathbf{v}(x, \mathrm{LPT}(x)) \leq \mathbf{v}^*(x) + (1 - \frac{1}{m})p_j$, where job J_j is the last job to be assigned to machine M, which is a machine that ends up with the largest load, namely load $\mathbf{v}(x, \mathrm{LPT}(x))$.

But we can assume $j > m$ (otherwise the algorithm is optimal because J_j gets its own machine and $\mathbf{v}(x, \mathrm{LPT}(x)) = p_j$).

So $p_1 \geq p_2 \geq \cdots \geq p_{m+1} \geq p_j$.

But there must be two jobs from the first $m + 1$ that share a machine (since we only have m machines) so $\mathbf{v}^*(x) \geq 2p_j$.

Then $\mathbf{v}(x, \mathrm{LPT}(x)) \leq \mathbf{v}^*(x) + (1 - \frac{1}{m})\frac{\mathbf{v}^*(x)}{2} = \mathbf{v}^*(x)(\frac{3}{2} - \frac{1}{2m})$.

Example

processing times: $1, 2, 1, 3, 3, 2, 6$

LS would proceed as follows

machines	1	2	3	4
	1	2	1	3
	1,3	2	1	3
	1,3	2	1,2	3
	1,3	2,6	1,2	3

Value is 8.

sorted processing times: $6, 3, 3, 2, 2, 1, 1$

LPT would proceed as follows

machines	1	2	3	4
	6	3	3	2
	6	3	3	2,2
	6	3,1	3	2,2
	6	3,1	3,1	2,2

Value is 6 (which is optimal since some machine has to take the job with processing time 6)

Example: approximation algorithms for vertex covers

Recall the vertex cover optimisation problem:

$\mathcal{I}$ is the set of undirected graphs.

For every $G \in \mathcal{I}$, $\mathcal{S}(G)$ is the set of vertex covers of G.

(Recall that a vertex cover is a set $U \subseteq V(G)$ such that every edge of G has at least one endpoint in U.)

The value $\mathbf{v}(G, U)$ is the size of U.

Finally, this is a minimisation problem, so $\mathrm{goal} = \min$.

We showed in the NP-completeness section that it is NP-complete to decide whether a graph G has a vertex cover of size k, so this optimisation problem is NP-hard.

Example: approximation algorithms for vertex covers

Recall the vertex cover optimisation problem:

$\mathcal{I}$ is the set of undirected graphs.

For every $G \in \mathcal{I}$, $\mathcal{S}(G)$ is the set of vertex covers of G.

(Recall that a vertex cover is a set $U \subseteq V(G)$ such that every edge of G has at least one endpoint in U.)

The value $\mathbf{v}(G, U)$ is the size of U.

Finally, this is a minimisation problem, so $\mathrm{goal} = \min$.

We showed in the NP-completeness section that it is NP-complete to decide whether a graph G has a vertex cover of size k, so this optimisation problem is NP-hard.

Example: approximation algorithms for vertex covers

Recall the vertex cover optimisation problem:

$\mathcal{I}$ is the set of undirected graphs.

For every $G \in \mathcal{I}$, $\mathcal{S}(G)$ is the set of vertex covers of G.

(Recall that a vertex cover is a set $U \subseteq V(G)$ such that every edge of G has at least one endpoint in U.)

The value $\mathbf{v}(G, U)$ is the size of U.

Finally, this is a minimisation problem, so $\mathrm{goal} = \min$.

We showed in the NP-completeness section that it is NP-complete to decide whether a graph G has a vertex cover of size k, so this optimisation problem is NP-hard.

Example: approximation algorithms for vertex covers

Recall the vertex cover optimisation problem:

$\mathcal{I}$ is the set of undirected graphs.

For every $G \in \mathcal{I}$, $\mathcal{S}(G)$ is the set of vertex covers of G.

(Recall that a vertex cover is a set $U \subseteq V(G)$ such that every edge of G has at least one endpoint in U.)

The value $\mathbf{v}(G, U)$ is the size of U.

Finally, this is a minimisation problem, so $\mathrm{goal} = \min$.

We showed in the NP-completeness section that it is NP-complete to decide whether a graph G has a vertex cover of size k, so this optimisation problem is NP-hard.

Example: approximation algorithms for vertex covers

Recall the vertex cover optimisation problem:

$\mathcal{I}$ is the set of undirected graphs.

For every $G \in \mathcal{I}$, $\mathcal{S}(G)$ is the set of vertex covers of G.

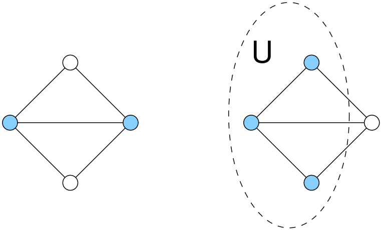
(Recall that a vertex cover is a set $U \subseteq V(G)$ such that every edge of G has at least one endpoint in U.)

The value $\mathbf{v}(G, U)$ is the size of U.

Finally, this is a minimisation problem, so $\mathrm{goal} = \min$.

We showed in the NP-completeness section that it is NP-complete to decide whether a graph G has a vertex cover of size k, so this optimisation problem is NP-hard.

Example



An optimal vertex cover is shown on the left. It has size 2. The vertex cover U has size 3, so $R(G, U) = 3/2$.

We will look at several approximation algorithms for vertex cover.

Simplest Greedy

A natural heuristic for VC is a greedy algorithm which repeatedly picks an edge that has not yet been covered, and places one of its end-points in the current covering set.

GREEDY1 (G)
  $C \leftarrow \emptyset$
  while $E \neq \emptyset$
    Pick any edge $e \in E$ and any endpoint v of e
    $C \leftarrow C \cup \{v\}$
    $E \leftarrow E \setminus \{e' \in E : v \sim e'\}$
  return C

It is easy to see that this algorithm outputs a vertex cover. We will show that GREEDY1 does not achieve a bounded performance ratio.

We will start by constructing a useful graph to use as the input.

First, how big is $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{r}$?

This is the "r-th Harmonic number" It is $\ln(r) + O(1)$. (You can find a proof in CLR) So it is $\Theta(\log(r))$.

Now, how big is $\frac{r}{1} + \frac{r}{2} + \frac{r}{3} + \cdots + \frac{r}{r}$? $\Theta(r \log r)$.

Finally, how big is $\lfloor \frac{r}{1} \rfloor + \lfloor \frac{r}{2} \rfloor + \lfloor \frac{r}{3} \rfloor + \cdots + \lfloor \frac{r}{r} \rfloor$? Also $\Theta(r \log r)$.

Our graph will be a bipartite graph B with vertex sets L and R. $|L| = r$ and $R = R_1 \cup R_2 \cup \cdots \cup R_r$ where $|R_i| = \lfloor \frac{r}{i} \rfloor$.

Each vertex in R_i will connect to exactly i vertices in L. Each vertex in L will connect to at most one vertex in R_i.

We will start by constructing a useful graph to use as the input.

First, how big is $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{r}$?

This is the "r-th Harmonic number" It is $\ln(r) + O(1)$. (You can find a proof in CLR) So it is $\Theta(\log(r))$.

Now, how big is $\frac{r}{1} + \frac{r}{2} + \frac{r}{3} + \cdots + \frac{r}{r}$? $\Theta(r \log r)$.

Finally, how big is $\lfloor \frac{r}{1} \rfloor + \lfloor \frac{r}{2} \rfloor + \lfloor \frac{r}{3} \rfloor + \cdots + \lfloor \frac{r}{r} \rfloor$? Also $\Theta(r \log r)$.

Our graph will be a bipartite graph B with vertex sets L and R. $|L| = r$ and $R = R_1 \cup R_2 \cup \cdots \cup R_r$ where $|R_i| = \lfloor \frac{r}{i} \rfloor$.

Each vertex in R_i will connect to exactly i vertices in L. Each vertex in L will connect to at most one vertex in R_i.

We will start by constructing a useful graph to use as the input.

First, how big is $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{r}$?

This is the "r-th Harmonic number" It is $\ln(r) + O(1)$. (You can find a proof in CLR) So it is $\Theta(\log(r))$.

Now, how big is $\frac{r}{1} + \frac{r}{2} + \frac{r}{3} + \cdots + \frac{r}{r}$? $\Theta(r \log r)$.

Finally, how big is $\lfloor \frac{r}{1} \rfloor + \lfloor \frac{r}{2} \rfloor + \lfloor \frac{r}{3} \rfloor + \cdots + \lfloor \frac{r}{r} \rfloor$? Also $\Theta(r \log r)$.

Our graph will be a bipartite graph B with vertex sets L and R. $|L| = r$ and $R = R_1 \cup R_2 \cup \cdots \cup R_r$ where $|R_i| = \lfloor \frac{r}{i} \rfloor$.

Each vertex in R_i will connect to exactly i vertices in L. Each vertex in L will connect to at most one vertex in R_i.

We will start by constructing a useful graph to use as the input.

First, how big is $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{r}$?

This is the "r-th Harmonic number" It is $\ln(r) + O(1)$. (You can find a proof in CLR) So it is $\Theta(\log(r))$.

Now, how big is $\frac{r}{1} + \frac{r}{2} + \frac{r}{3} + \cdots + \frac{r}{r}$? $\Theta(r \log r)$.

Finally, how big is $\lfloor \frac{r}{1} \rfloor + \lfloor \frac{r}{2} \rfloor + \lfloor \frac{r}{3} \rfloor + \cdots + \lfloor \frac{r}{r} \rfloor$? Also $\Theta(r \log r)$.

Our graph will be a bipartite graph B with vertex sets L and R. $|L| = r$ and $R = R_1 \cup R_2 \cup \cdots \cup R_r$ where $|R_i| = \lfloor \frac{r}{i} \rfloor$.

Each vertex in R_i will connect to exactly i vertices in L. Each vertex in L will connect to at most one vertex in R_i.

We will start by constructing a useful graph to use as the input.

First, how big is $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{r}$?

This is the "r-th Harmonic number" It is $\ln(r) + O(1)$. (You can find a proof in CLR) So it is $\Theta(\log(r))$.

Now, how big is $\frac{r}{1} + \frac{r}{2} + \frac{r}{3} + \cdots + \frac{r}{r}$? $\Theta(r \log r)$.

Finally, how big is $\lfloor \frac{r}{1} \rfloor + \lfloor \frac{r}{2} \rfloor + \lfloor \frac{r}{3} \rfloor + \cdots + \lfloor \frac{r}{r} \rfloor$? Also $\Theta(r \log r)$.

Our graph will be a bipartite graph B with vertex sets L and R. $|L| = r$ and $R = R_1 \cup R_2 \cup \cdots \cup R_r$ where $|R_i| = \lfloor \frac{r}{i} \rfloor$.

Each vertex in R_i will connect to exactly i vertices in L. Each vertex in L will connect to at most one vertex in R_i.

We will start by constructing a useful graph to use as the input.

First, how big is $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{r}$?

This is the "r-th Harmonic number" It is $\ln(r) + O(1)$. (You can find a proof in CLR) So it is $\Theta(\log(r))$.

Now, how big is $\frac{r}{1} + \frac{r}{2} + \frac{r}{3} + \cdots + \frac{r}{r}$? $\Theta(r \log r)$.

Finally, how big is $\lfloor \frac{r}{1} \rfloor + \lfloor \frac{r}{2} \rfloor + \lfloor \frac{r}{3} \rfloor + \cdots + \lfloor \frac{r}{r} \rfloor$? Also $\Theta(r \log r)$.

Our graph will be a bipartite graph B with vertex sets L and R. $|L| = r$ and $R = R_1 \cup R_2 \cup \cdots \cup R_r$ where $|R_i| = \lfloor \frac{r}{i} \rfloor$.

Each vertex in R_i will connect to exactly i vertices in L. Each vertex in L will connect to at most one vertex in R_i.

We will start by constructing a useful graph to use as the input.

First, how big is $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{r}$?

This is the "r-th Harmonic number" It is $\ln(r) + O(1)$. (You can find a proof in CLR) So it is $\Theta(\log(r))$.

Now, how big is $\frac{r}{1} + \frac{r}{2} + \frac{r}{3} + \cdots + \frac{r}{r}$? $\Theta(r \log r)$.

Finally, how big is $\lfloor \frac{r}{1} \rfloor + \lfloor \frac{r}{2} \rfloor + \lfloor \frac{r}{3} \rfloor + \cdots + \lfloor \frac{r}{r} \rfloor$? Also $\Theta(r \log r)$.

Our graph will be a bipartite graph B with vertex sets L and R. $|L| = r$ and $R = R_1 \cup R_2 \cup \cdots \cup R_r$ where $|R_i| = \lfloor \frac{r}{i} \rfloor$.

Each vertex in R_i will connect to exactly i vertices in L. Each vertex in L will connect to at most one vertex in R_i.

We will start by constructing a useful graph to use as the input.

First, how big is $\frac{1}{1} + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{r}$?

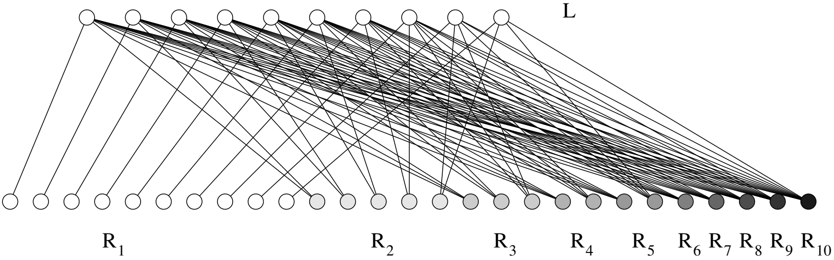
This is the "r-th Harmonic number" It is $\ln(r) + O(1)$. (You can find a proof in CLR) So it is $\Theta(\log(r))$.

Now, how big is $\frac{r}{1} + \frac{r}{2} + \frac{r}{3} + \cdots + \frac{r}{r}$? $\Theta(r \log r)$.

Finally, how big is $\lfloor \frac{r}{1} \rfloor + \lfloor \frac{r}{2} \rfloor + \lfloor \frac{r}{3} \rfloor + \cdots + \lfloor \frac{r}{r} \rfloor$? Also $\Theta(r \log r)$.

Our graph will be a bipartite graph B with vertex sets L and R. $|L| = r$ and $R = R_1 \cup R_2 \cup \cdots \cup R_r$ where $|R_i| = \lfloor \frac{r}{i} \rfloor$.

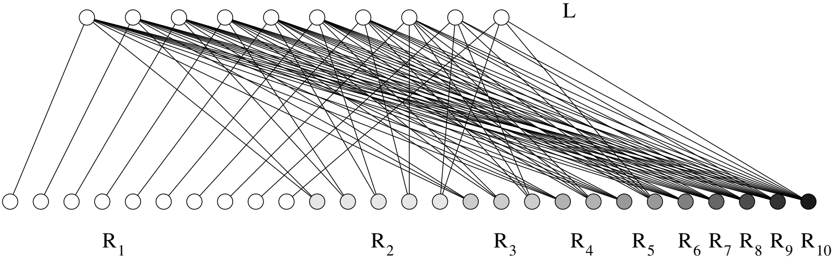
Each vertex in R_i will connect to exactly i vertices in L. Each vertex in L will connect to at most one vertex in R_i.



Now $\mathbf{v}(B, R) = \Theta(r \log(r))$ and $\mathbf{v}^*(B) \leq \mathbf{v}(B, L) = r$, so the performance ratio of R with respect to the instance B is

$$\frac{\mathbf{v}(B, R)}{\mathbf{v}^*(B)} = \Theta(\log(r)).$$

Note that the algorithm could choose R as its output.



Now $\mathbf{v}(B, R) = \Theta(r \log(r))$ and $\mathbf{v}^*(B) \leq \mathbf{v}(B, L) = r$, so the performance ratio of R with respect to the instance B is

$$\frac{\mathbf{v}(B, R)}{\mathbf{v}^*(B)} = \Theta(\log(r)).$$

Note that the algorithm could choose R as its output.

Better greedy algorithm

How do we achieve a better ratio than this?

Let us try the obvious strategy of modifying the Algorithm GREEDY1 to be less arbitrary in its choice of vertices to be included in the cover. A natural modification is to repeatedly choose vertices which are incident to the largest number of *currently* uncovered edges.

```
GREEDY2 (G)
    C ← ∅
    while E ≠ ∅
        Pick a vertex v ∈ V of
            maximum degree in the current graph
        C ← C ∪ {v}
        E ← E \ {e′ ∈ E : v ∼ e′}
    return C
```

Algorithm analysis

On input B, GREEDY2 could also output R as a vertex cover!

Vertices in L have degree at most r. The algorithm could choose vertices from R_r at the very first stage.

After this, vertices in L have degree at most $r - 1$ so the algorithm could choose vertices from R_{r-1}.

In general, it would choose the highest degree vertices from R at each stage.

Could this be an improvement?

```
GREEDY1 (G)
  C ← ∅
  while E ≠ ∅
    Pick any edge e ∈ E and any endpoint v of e
    C ← C ∪ {v}
    E ← E \ {e' ∈ E : v ∼ e'}
  return C
```

```
GREEDYBOTH (G)
  C ← ∅
  while E ≠ ∅
    Pick any edge e = (u, v) ∈ E
    C ← C ∪ {u, v}
    E ← E \ {e' ∈ E : e ∼ e'}
  return C
```

GreedyBoth is a 2-approximation algorithm.

1. It computes a vertex cover.

2. Let j be the number of edges e that are examined by the algorithm. These edges are not adjacent, so $\mathbf{v}^*(G) \geq j$. On the other hand, $\mathbf{v}(G, \textsc{GreedyBoth}(G)) \leq 2j$

So the performance ratio is at most $2j/j = 2$.

Another way to look at the algorithm

GREEDYBOTH (G)
  $C \leftarrow \emptyset$
  while $E \neq \emptyset$
    Pick any edge $e = (u, v) \in E$
    $C \leftarrow C \cup \{u, v\}$
    $E \leftarrow E \setminus \{e' \in E : e \sim e'\}$
  return C

LOCALRATIO (G)
  For all vertices v, $w(v) \leftarrow 1$
  While there exists an edge (u, v)
    such that $\min(w(u), w(v)) = 1$
      $w(u) \leftarrow w(u) - 1$
      $w(v) \leftarrow w(v) - 1$
  Return $C = \{v \mid w(v) = 0\}$

1. LOCALRATIO returns a vertex cover.

2. Each edge (u, v) that we consider contributes at least one to $\mathbf{v}^*(G)$ and at most two to $\mathbf{v}(G, \text{LocalRatio}(G))$.

Another way to look at the algorithm

GREEDYBOTH (G)
  $C \leftarrow \emptyset$
  while $E \neq \emptyset$
    Pick any edge $e = (u, v) \in E$
    $C \leftarrow C \cup \{u, v\}$
    $E \leftarrow E \setminus \{e' \in E : e \sim e'\}$
  return C

LOCALRATIO (G)
  For all vertices v, $w(v) \leftarrow 1$
  While there exists an edge (u, v)
    such that $\min(w(u), w(v)) = 1$
      $w(u) \leftarrow w(u) - 1$
      $w(v) \leftarrow w(v) - 1$
  Return $C = \{v \mid w(v) = 0\}$

1. LOCALRATIO returns a vertex cover.

2. Each edge (u, v) that we consider contributes at least one to $\mathbf{v}^*(G)$ and at most two to $\mathbf{v}(G, \text{LocalRatio}(G))$.

Another way to look at the algorithm

GREEDYBOTH (G)
  $C \leftarrow \emptyset$
  while $E \neq \emptyset$
    Pick any edge $e = (u, v) \in E$
    $C \leftarrow C \cup \{u, v\}$
    $E \leftarrow E \setminus \{e' \in E : e \sim e'\}$
  return C

LOCALRATIO (G)
  For all vertices v, $w(v) \leftarrow 1$
  While there exists an edge (u, v)
    such that $\min(w(u), w(v)) = 1$
      $w(u) \leftarrow w(u) - 1$
      $w(v) \leftarrow w(v) - 1$
  Return $C = \{v \mid w(v) = 0\}$

1. LOCALRATIO returns a vertex cover.

2. Each edge (u, v) that we consider contributes at least one to $\mathbf{v}^*(G)$ and at most two to $\mathbf{v}(G, \text{LocalRatio}(G))$.

Can you think of a graph G on which the this algorithm really does no better than a factor of 2?

How about n non-intersecting edges.

Can you think of a graph G on which the this algorithm really does no better than a factor of 2?

How about n non-intersecting edges.

It is an important open problem whether there is a $2 - \epsilon$ approximation algorithm for vertex cover for any positive constant ϵ.

A generalisation of the problem

An Instance $G \in \mathcal{I}$ is an (undirected) graph in which each vertex u has a nonnegative weight $w(u)$.

For every $G \in \mathcal{I}$, $\mathcal{S}(G)$ is the set of vertex covers U of G.

The value $\mathbf{v}(G, U)$ is $\sum_{v \in U} w(v)$.

$\mathrm{goal} = \min$.

```
LOCALRATIO (G)
  While there exists an edge (u, v) such that min(w(u), w(v)) > 0
        Let ε = min(w(u), w(v))
        w(u) ← w(u) − ε
        w(v) ← w(v) − ε
  Return C = {v | w(v) = 0}
```

It is obvious that this algorithm returns a vertex cover.

Let (u_i, v_i) be the i'th edge considered, with $\epsilon_i = \min(w(u_i), w(v_i))$. Suppose r edges are considered in all.

Then $\mathbf{v}(G, \text{LOCAL RATIO}(G)) \leq 2\epsilon_1 + 2\epsilon_2 + \cdots + 2\epsilon_r$.

Also, $\mathbf{v}^*(G) \geq \epsilon_1 + \epsilon_2 + \cdots + \epsilon_r$.

So LOCALRATIO is a 2-approximation algorithm.

LOCALRATIO (G)

While there exists an edge (u, v) such that $\min(w(u), w(v)) > 0$

Let $\epsilon = \min(w(u), w(v))$

$w(u) \leftarrow w(u) - \epsilon$

$w(v) \leftarrow w(v) - \epsilon$

Return $C = \{v \mid w(v) = 0\}$

It is obvious that this algorithm returns a vertex cover.

Let (u_i, v_i) be the i'th edge considered, with $\epsilon_i = \min(w(u_i), w(v_i))$. Suppose r edges are considered in all.

Then $\mathbf{v}(G, \text{LOCAL RATIO}(G)) \leq 2\epsilon_1 + 2\epsilon_2 + \cdots + 2\epsilon_r$.

Also, $\mathbf{v}^*(G) \geq \epsilon_1 + \epsilon_2 + \cdots + \epsilon_r$.

So LOCALRATIO is a 2-approximation algorithm.

$\textsc{LocalRatio}$ (G)
While there exists an edge (u, v) such that $\min(w(u), w(v)) > 0$
Let $\epsilon = \min(w(u), w(v))$
$w(u) \leftarrow w(u) - \epsilon$
$w(v) \leftarrow w(v) - \epsilon$
Return $C = \{v \mid w(v) = 0\}$

It is obvious that this algorithm returns a vertex cover.

Let (u_i, v_i) be the i'th edge considered, with $\epsilon_i = \min(w(u_i), w(v_i))$. Suppose r edges are considered in all.

Then $\mathbf{v}(G, \textsc{Local Ratio}(G)) \leq 2\epsilon_1 + 2\epsilon_2 + \cdots + 2\epsilon_r$.

Also, $\mathbf{v}^*(G) \geq \epsilon_1 + \epsilon_2 + \cdots + \epsilon_r$.

So $\textsc{LocalRatio}$ is a 2-approximation algorithm.

$\textsc{LocalRatio}$ (G)
While there exists an edge (u, v) such that $\min(w(u), w(v)) > 0$
Let $\epsilon = \min(w(u), w(v))$
$w(u) \leftarrow w(u) - \epsilon$
$w(v) \leftarrow w(v) - \epsilon$
Return $C = \{v \mid w(v) = 0\}$

It is obvious that this algorithm returns a vertex cover.

Let (u_i, v_i) be the i'th edge considered, with $\epsilon_i = \min(w(u_i), w(v_i))$. Suppose r edges are considered in all.

Then $\mathbf{v}(G, \textsc{Local Ratio}(G)) \leq 2\epsilon_1 + 2\epsilon_2 + \cdots + 2\epsilon_r$.

Also, $\mathbf{v}^*(G) \geq \epsilon_1 + \epsilon_2 + \cdots + \epsilon_r$.

So $\textsc{LocalRatio}$ is a 2-approximation algorithm.

An important example: Maximum Satisfiability

Instance: Set C of disjunctive clauses on a set of variables V

Solution: A truth assignment $f : V \to \{\mathrm{true}, \mathrm{false}\}$.

The value $\mathbf{v}(C, f)$ is the number of clauses in C which are satisfied by f.

Example:

$$C = \{(\neg y_1 \vee \neg x_1 \vee y_1), (\neg y_1 \vee x_1 \vee \neg y_2), (\neg y_1 \vee x_1 \vee y_2)\}.$$

$\mathbf{v}^*(C) = 3$.

An important example: Maximum Satisfiability

Instance: Set C of disjunctive clauses on a set of variables V

Solution: A truth assignment $f : V \to \{\mathrm{true}, \mathrm{false}\}$.

The value $\mathbf{v}(C, f)$ is the number of clauses in C which are satisfied by f.

Example:

$$C = \{(\neg y_1 \vee \neg x_1 \vee y_1), (\neg y_1 \vee x_1 \vee \neg y_2), (\neg y_1 \vee x_1 \vee y_2)\}.$$

$\mathbf{v}^*(C) = 3$.

An important example: Maximum Satisfiability

Instance: Set C of disjunctive clauses on a set of variables V

Solution: A truth assignment $f : V \to \{\mathrm{true}, \mathrm{false}\}$.

The value $\mathbf{v}(C, f)$ is the number of clauses in C which are satisfied by f.

Example:

$$C = \{(\neg y_1 \vee \neg x_1 \vee y_1), (\neg y_1 \vee x_1 \vee \neg y_2), (\neg y_1 \vee x_1 \vee y_2)\}.$$

$\mathbf{v}^*(C) = 3$.

Let $c = |C|$. We showed in the NP-completeness section that it is NP-hard to decide whether an instance C has a solution with value c (a solution that satisfies all clauses), so the optimisation problem is NP-hard.

Maximum satisfiability is 2-approximable.

Before we give a deterministic approximation algorithm for Maximum Satisfiability, let's look at the performance ratio of a simple randomised algorithm.

Algorithm RS (for "Randomised Satisfiability"): For each variable $v \in V$, flip a fair coin. With probability $1/2$, set $f(v) = \mathrm{true}$. Otherwise, set $f(v) = \mathrm{false}$.

Clearly, $\mathbf{v}^*(C) \leq c$.

What about $\mathbf{v}(C, \mathrm{RS}(C))$? This is now a random variable.

This means that $\mathbf{v}(C, \mathrm{RS}(C))$ is a function of the coin-flips that arise when $\mathrm{RS}(C)$ is run.

Maximum satisfiability is 2-approximable.

Before we give a deterministic approximation algorithm for Maximum Satisfiability, let's look at the performance ratio of a simple randomised algorithm.

Algorithm RS (for "Randomised Satisfiability"): For each variable $v \in V$, flip a fair coin. With probability $1/2$, set $f(v) = \mathrm{true}$. Otherwise, set $f(v) = \mathrm{false}$.

Clearly, $\mathbf{v}^*(C) \leq c$.

What about $\mathbf{v}(C, \mathrm{RS}(C))$? This is now a random variable.

This means that $\mathbf{v}(C, \mathrm{RS}(C))$ is a function of the coin-flips that arise when $\mathrm{RS}(C)$ is run.

Maximum satisfiability is 2-approximable.

Before we give a deterministic approximation algorithm for Maximum Satisfiability, let's look at the performance ratio of a simple randomised algorithm.

Algorithm RS (for "Randomised Satisfiability"): For each variable $v \in V$, flip a fair coin. With probability $1/2$, set $f(v) = \mathrm{true}$. Otherwise, set $f(v) = \mathrm{false}$.

Clearly, $\mathbf{v}^*(C) \leq c$.

What about $\mathbf{v}(C, \mathrm{RS}(C))$? This is now a random variable.

This means that $\mathbf{v}(C, \mathrm{RS}(C))$ is a function of the coin-flips that arise when $\mathrm{RS}(C)$ is run.

Maximum satisfiability is 2-approximable.

Before we give a deterministic approximation algorithm for Maximum Satisfiability, let's look at the performance ratio of a simple randomised algorithm.

Algorithm RS (for "Randomised Satisfiability"): For each variable $v \in V$, flip a fair coin. With probability $1/2$, set $f(v) = \text{true}$. Otherwise, set $f(v) = \text{false}$.

Clearly, $\mathbf{v}^*(C) \leq c$.

What about $\mathbf{v}(C, \mathrm{RS}(C))$? This is now a random variable.

This means that $\mathbf{v}(C, \mathrm{RS}(C))$ is a function of the coin-flips that arise when $\mathrm{RS}(C)$ is run.

Maximum satisfiability is 2-approximable.

Before we give a deterministic approximation algorithm for Maximum Satisfiability, let's look at the performance ratio of a simple randomised algorithm.

Algorithm RS (for "Randomised Satisfiability"): For each variable $v \in V$, flip a fair coin. With probability $1/2$, set $f(v) = \mathrm{true}$. Otherwise, set $f(v) = \mathrm{false}$.

Clearly, $\mathbf{v}^*(C) \leq c$.

What about $\mathbf{v}(C, \mathrm{RS}(C))$? This is now a random variable.

This means that $\mathbf{v}(C, \mathrm{RS}(C))$ is a function of the coin-flips that arise when $\mathrm{RS}(C)$ is run.

Let $n = |v|$.

For any sequence of values $\{f_1, \ldots, f_n\}$ from $\{\text{true}, \text{false}\}$,

$\mathbf{v}(C, \mathrm{RS}(C))(f_1, \ldots, f_n)$ is the number of clauses in C that are satisfied when the first coin-flip has value f_1, the second coin-flip has value f_2, and so on.

The expected value of the solution produced by the algorithm is given by

$$E[\mathbf{v}(C, \mathrm{RS}(C))] = \sum_{f_1,\ldots,f_n} \Pr(f_1, \ldots, f_n)\mathbf{v}(C, \mathrm{RS}(C))(f_1, \ldots, f_n),$$

where $\Pr(f_1, \ldots, f_n)$ denotes the probability that the outcome of the coin flips is $f_1, \ldots, f_n$.

Let $n = |v|$.

For any sequence of values $\{f_1, \dots, f_n\}$ from $\{\text{true}, \text{false}\}$,

$\mathbf{v}(C, \text{RS}(C))(f_1, \dots, f_n)$ is the number of clauses in C that are satisfied when the first coin-flip has value f_1, the second coin-flip has value f_2, and so on.

The expected value of the solution produced by the algorithm is given by

$$E[\mathbf{v}(C, \text{RS}(C))] = \sum_{f_1, \dots, f_n} \Pr(f_1, \dots, f_n) \mathbf{v}(C, \text{RS}(C))(f_1, \dots, f_n),$$

where $\Pr(f_1, \dots, f_n)$ denotes the probability that the outcome of the coin flips is $f_1, \dots, f_n$.

Let $n = |v|$.

For any sequence of values $\{f_1, \ldots, f_n\}$ from $\{\mathrm{true}, \mathrm{false}\}$,

$\mathbf{v}(C, \mathrm{RS}(C))(f_1, \ldots, f_n)$ is the number of clauses in C that are satisfied when the first coin-flip has value f_1, the second coin-flip has value f_2, and so on.

The expected value of the solution produced by the algorithm is given by

$$E[\mathbf{v}(C, \mathrm{RS}(C))] = \sum_{f_1, \ldots, f_n} \Pr(f_1, \ldots, f_n)\mathbf{v}(C, \mathrm{RS}(C))(f_1, \ldots, f_n),$$

where $\Pr(f_1, \ldots, f_n)$ denotes the probability that the outcome of the coin flips is $f_1, \ldots, f_n$.

That may look complicated, but it isn't!

Let C_i be a random variable which is 1 if the ith clause is satisfied, and is 0 otherwise.

C_i is a function of the coin-flips. That means that, given a sequence of values $f_1, \ldots, f_n$, $C_i(f_1, \ldots, f_n)$ is either 0 or 1.

Now, $\Pr(C_i = 1) \geq \frac{1}{2}$ since the probability that the coin-flip sequence satisfies the first literal in c_i is exactly $\frac{1}{2}$. Thus, $E[C_i] \geq \frac{1}{2}$.

That may look complicated, but it isn't!

Let C_i be a random variable which is 1 if the ith clause is satisfied, and is 0 otherwise.

C_i is a function of the coin-flips. That means that, given a sequence of values $f_1, \ldots, f_n$, $C_i(f_1, \ldots, f_n)$ is either 0 or 1.

Now, $\Pr(C_i = 1) \geq \frac{1}{2}$ since the probability that the coin-flip sequence satisfies the first literal in c_i is exactly $\frac{1}{2}$. Thus, $E[C_i] \geq \frac{1}{2}$.

That may look complicated, but it isn't!

Let C_i be a random variable which is 1 if the ith clause is satisfied, and is 0 otherwise.

C_i is a function of the coin-flips. That means that, given a sequence of values $f_1, \ldots, f_n$, $C_i(f_1, \ldots, f_n)$ is either 0 or 1.

Now, $\Pr(C_i = 1) \geq \frac{1}{2}$ since the probability that the coin-flip sequence satisfies the first literal in c_i is exactly $\frac{1}{2}$. Thus, $E[C_i] \geq \frac{1}{2}$.

That may look complicated, but it isn't!

Let C_i be a random variable which is 1 if the ith clause is satisfied, and is 0 otherwise.

C_i is a function of the coin-flips. That means that, given a sequence of values $f_1, \ldots, f_n$, $C_i(f_1, \ldots, f_n)$ is either 0 or 1.

Now, $\Pr(C_i = 1) \geq \frac{1}{2}$ since the probability that the coin-flip sequence satisfies the first literal in c_i is exactly $\frac{1}{2}$. Thus, $E[C_i] \geq \frac{1}{2}$.

Now $\mathbf{v}(C, \mathrm{RS}(C)) = C_1 + \cdots + C_c$ so

$$E[\mathbf{v}(C, \mathrm{RS}(C))] = E[C_1] + \cdots + E[C_c] \geq \frac{c}{2}.$$

Thus, the expected performance ratio is

$$\frac{\mathbf{v}^*(C)}{E[\mathbf{v}(C, \mathrm{RS}(C))]} \leq \frac{c}{c/2} = 2.$$

Now $\mathbf{v}(C, \mathrm{RS}(C)) = C_1 + \cdots + C_c$ so

$$E[\mathbf{v}(C, \mathrm{RS}(C))] = E[C_1] + \cdots + E[C_c] \geq \frac{c}{2}.$$

Thus, the expected performance ratio is

$$\frac{\mathbf{v}^*(C)}{E[\mathbf{v}(C, \mathrm{RS}(C))]} \leq \frac{c}{c/2} = 2.$$

Now $\mathbf{v}(C, \mathrm{RS}(C)) = C_1 + \cdots + C_c$ so

$$E[\mathbf{v}(C, \mathrm{RS}(C))] = E[C_1] + \cdots + E[C_c] \geq \frac{c}{2}.$$

Thus, the expected performance ratio is

$$\frac{\mathbf{v}^*(C)}{E[\mathbf{v}(C, \mathrm{RS}(C))]} \leq \frac{c}{c/2} = 2.$$

A deterministic 2-approximation

Now, let's give a deterministic 2-approximation algorithm for Maximum Satisfiability.

DS (C, V)

For all $v \in V$, $f(v) \leftarrow \text{true}$

While $C \neq \emptyset$

Let ℓ be a literal that occurs in the max number of clauses in C

Let v be the variable so that $\ell = v$ or $\ell = \neg v$

If $\ell = v$

$f(v) \leftarrow \text{true}$

Remove the literal $\neg v$ from every clause in C

Else

$f(v) \leftarrow \text{false}$

remove the literal v from every clause in C

Remove from C any clauses containing ℓ

Remove from C any empty clauses

Algorithm DS runs in polynomial time. It is a greedy algorithm.

DS (C, V)

For all $v \in V$, $f(v) \leftarrow \text{true}$

While $C \neq \emptyset$

Let ℓ be a literal that occurs in the max number of clauses in C

Let v be the variable so that $\ell = v$ or $\ell = \neg v$

If $\ell = v$

$f(v) \leftarrow \text{true}$

Remove the literal $\neg v$ from every clause in C

Else

$f(v) \leftarrow \text{false}$

remove the literal v from every clause in C

Remove from C any clauses containing ℓ

Remove from C any empty clauses

Algorithm DS runs in polynomial time. It is a greedy algorithm.

DS (C, V)

For all $v \in V$, $f(v) \leftarrow \text{true}$

While $C \neq \emptyset$

Let ℓ be a literal that occurs in the max number of clauses in C

Let v be the variable so that $\ell = v$ or $\ell = \neg v$

If $\ell = v$

$f(v) \leftarrow \text{true}$

Remove the literal $\neg v$ from every clause in C

Else

$f(v) \leftarrow \text{false}$

remove the literal v from every clause in C

Remove from C any clauses containing ℓ

Remove from C any empty clauses

Algorithm DS runs in polynomial time. It is a greedy algorithm.

To show that algorithm DS is a 2-approximation, we will show by induction on $n = |V|$ that $\mathbf{v}(\mathrm{DS}(C, V)) \geq c/2$.

Then

the performance ratio is

$$\frac{\mathbf{v}^*(C)}{\mathbf{v}(C, \mathrm{DS}(C))} \leq \frac{c}{c/2} = 2.$$

The base case is $n = 1$.

To show that algorithm DS is a 2-approximation, we will show by induction on $n = |V|$ that $\mathbf{v}(\mathrm{DS}(C, V)) \geq c/2$.

Then

the performance ratio is

$$\frac{\mathbf{v}^*(C)}{\mathbf{v}(C, \mathrm{DS}(C))} \leq \frac{c}{c/2} = 2.$$

The base case is $n = 1$.

To show that algorithm DS is a 2-approximation, we will show by induction on $n = |V|$ that $\mathbf{v}(\mathrm{DS}(C, V)) \geq c/2$.

Then

the performance ratio is

$$\frac{\mathbf{v}^*(C)}{\mathbf{v}(C, \mathrm{DS}(C))} \leq \frac{c}{c/2} = 2.$$

The base case is $n = 1$.

For the inductive step, consider an instance (C, V) with $n = |V| > 1$.

Let c_1 be the number of clauses in which ℓ occurs and c_2 be the number of clauses in which $\neg\ell$ occurs. Since ℓ is chosen, $c_1 \geq c_2$.

The algorithm sets $f(v)$ so as to satisfy the c_1 clauses.

It then considers an instance with $n - 1$ variables and at least $c - c_1 - c_2$ clauses.

By the inductive hypothesis, at least $(c - c_1 - c_2)/2$ of these will be satisfied, so the total number of satisfied clauses is at least

$$c_1 + \frac{c - c_1 - c_2}{2} = \frac{c + c_1 - c_2}{2} \geq \frac{c}{2}.$$

Approximation Complexity Classes

Recall that an optimisation problem $P = (\mathcal{I}, \mathcal{S}, \mathbf{v}, \mathrm{goal})$ is in NPO if the following holds.

(1) The set $\mathcal{I}$ of instances is recognizable in polynomial time.

(2) There is a polynomial q so that, for every instance $x \in \mathcal{I}$ and any feasible solution $y \in \mathcal{S}(x)$, we have $|y| \leq q(|x|)$. Also, it is decidable in polynomial time whether $y \in \mathcal{S}(x)$.

(3) $\mathbf{v}(x, y)$ is computable in polynomial time.

APX is the class of all NPO problems P such that, for some fixed $r \geq 1$, there exists a polynomial-time r-approximation algorithm for P.

We have shown that several problems are in APX:

- Planar Vertex Colouring (performance ratio $5/3$ since at most 5 colour are used and at least 3 are required (otherwise we solve the problem optimally))
- Edge Colouring (performance ratio at $(\Delta + 1)/\Delta \leq 3/2$).
- Multiprocessor Scheduling (performance ratio $3/2$)
- Minimum Vertex Cover (performance ratio 2)
- Maximum Satisfiability (performance ratio 2)

Two other NPO problems that we have considered are

- Minimum Bin Packing
- Maximum Clique.

Minimum Bin Packing is in APX (see section 2.2.2). It can be shown that Maximum Clique is not in APX unless P=NP (see the supplementary notes).

An NPO problem that is unlikely to be in APX

Minimum Travelling Salesperson (MinTSP)

Instance: Set of cities $C = \{c_1, \ldots, c_n\}$. For each pair (c_i, c_j) of cities, a non-negative integer $D(i, j)$, which is the distance between them.

Solution: A tour of the cities. That is, a permutation $\{c_{i_1}, \ldots, c_{i_n}\}$

Value:

$$\left(\sum_{k=1}^{n-1} D(i_k, i_{k+1})\right) + D(i_n, i_1).$$

We will show that if MinTSP is in APX then P=NP.

An NPO problem that is unlikely to be in APX

Minimum Travelling Salesperson (MinTSP)

Instance: Set of cities $C = \{c_1, \ldots, c_n\}$. For each pair (c_i, c_j) of cities, a non-negative integer $D(i,j)$, which is the distance between them.

Solution: A tour of the cities. That is, a permutation $\{c_{i_1}, \ldots, c_{i_n}\}$

Value:

$$\left(\sum_{k=1}^{n-1} D(i_k, i_{k+1})\right) + D(i_n, i_1).$$

We will show that if MinTSP is in APX then P=NP.

You learned in Comp202 that the following problem is NP-complete.

Hamiltonian Circuit

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

Question: Is there a circuit that passes exactly once through every vertex.

Suppose for contradiction that $\mathcal{A}$ is an r-approximation algorithm for MinTSP.

Here is how to use $\mathcal{A}$ to solve Hamiltonian Circuit.

Given $G = (V, E)$, let $n = |V|$. Construct a MinTSP instance (C, D) by setting $C = V$ (so the cities are the vertices of G). If $(v_i, v_j) \in E$ then set $D(i, j) = 1$. Otherwise, set $D(i, j) = 1 + nr$.

You learned in Comp202 that the following problem is NP-complete.

Hamiltonian Circuit

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

Question: Is there a circuit that passes exactly once through every vertex.

Suppose for contradiction that $\mathcal{A}$ is an r-approximation algorithm for MinTSP.

Here is how to use $\mathcal{A}$ to solve Hamiltonian Circuit.

Given $G = (V, E)$, let $n = |V|$. Construct a MinTSP instance (C, D) by setting $C = V$ (so the cities are the vertices of G). If $(v_i, v_j) \in E$ then set $D(i, j) = 1$. Otherwise, set $D(i, j) = 1 + nr$.

You learned in Comp202 that the following problem is NP-complete.

Hamiltonian Circuit

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

Question: Is there a circuit that passes exactly once through every vertex.

Suppose for contradiction that $\mathcal{A}$ is an r-approximation algorithm for MinTSP.

Here is how to use $\mathcal{A}$ to solve Hamiltonian Circuit.

Given $G = (V, E)$, let $n = |V|$. Construct a MinTSP instance (C, D) by setting $C = V$ (so the cities are the vertices of G). If $(v_i, v_j) \in E$ then set $D(i, j) = 1$. Otherwise, set $D(i, j) = 1 + nr$.

You learned in Comp202 that the following problem is NP-complete.

Hamiltonian Circuit

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

Question: Is there a circuit that passes exactly once through every vertex.

Suppose for contradiction that $\mathcal{A}$ is an r-approximation algorithm for MinTSP.

Here is how to use $\mathcal{A}$ to solve Hamiltonian Circuit.

Given $G = (V, E)$, let $n = |V|$. Construct a MinTSP instance (C, D) by setting $C = V$ (so the cities are the vertices of G). If $(v_i, v_j) \in E$ then set $D(i, j) = 1$. Otherwise, set $D(i, j) = 1 + nr$.

Note that a Hamiltonian Circuit of G is a solution to the MinTSP instance (C, D) with value n.

Any other solution to the MinTSP intsance (C, D) uses at least one non-edge of G, so it has value at least $(n-1)+(1+nr) = nr+n$.

If G has a Hamiltonian Circuit, then $\mathbf{v}^*(C, D) = n$. Since $\mathcal{A}$ is an r-approximation algorithm, $\mathbf{v}((C, D), \mathcal{A}(C, D)) \leq nr$.

If G has no Hamiltonian Circuit, then $\mathbf{v}((C, D), \mathcal{A}(C, D)) \geq nr+1$.

Note that a Hamiltonian Circuit of G is a solution to the MinTSP instance (C, D) with value n.

Any other solution to the MinTSP intsance (C, D) uses at least one non-edge of G, so it has value at least $(n-1) + (1 + nr) = nr + n$.

If G has a Hamiltonian Circuit, then $\mathbf{v}^*(C, D) = n$. Since $\mathcal{A}$ is an r-approximation algorithm, $\mathbf{v}((C, D), \mathcal{A}(C, D)) \leq nr$.

If G has no Hamiltonian Circuit, then $\mathbf{v}((C, D), \mathcal{A}(C, D)) \geq nr + 1$.

So MinTSP is not in APX unless P=NP.