SAT: Modelling and Implementations

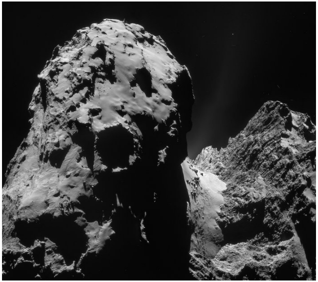
Mohamed Siala
https://siala.github.io

INSA-Toulouse & LAAS-CNRS

January 18, 2022

Context: Solving (Very) Hard Combinatorial Problems

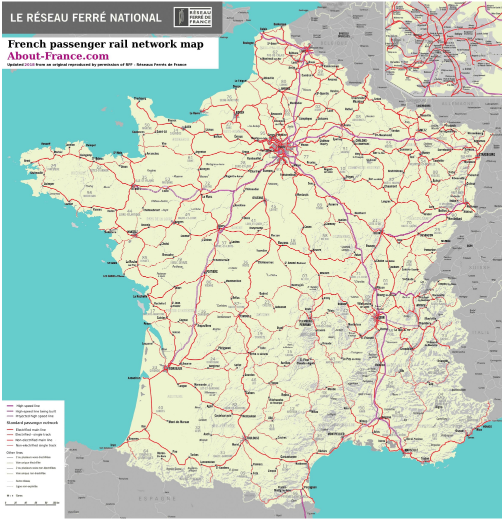
Context: Solving (Very) Hard Combinatorial Problems



https://homepages.laas.fr/ehebrard/rosetta.html

Context: Solving (Very) Hard Combinatorial Problems

Context: Solving (Very) Hard Combinatorial Problems



Context: Solving (Very) Hard Combinatorial Problems

Context: Solving (Very) Hard Combinatorial Problems

Context: Solving (Very) Hard Combinatorial Problems

Context: Solving (Very) Hard Combinatorial Problems

Why this Lecture?

- I noticed that most graduate students are doing software development.

Why this Lecture?

- I noticed that most graduate students are doing software development.
- We are missing job opportunities in optimisation!

Why this Lecture?

- I noticed that most graduate students are doing software development.
- We are missing job opportunities in optimisation!
- Resources: many.. a good start would be the online course on discrete optimisation
  `https://www.coursera.org/learn/discrete-optimization`

Solving Methodologies

Solving Methodologies

1. Adhoc methods

Solving Methodologies

1. Adhoc methods

Solving Methodologies

1. Adhoc methods
    1. Specific exact algorithm

Solving Methodologies

1. Adhoc methods
   1. Specific exact algorithm
   2. Heuristic method

Solving Methodologies

1. Adhoc methods
   1. Specific exact algorithm
   2. Heuristic method
   3. Meta-heuristic (genetic algorithms, ant colony, ..)
2. Declarative Approached

Solving Methodologies

1. Adhoc methods
   1. Specific exact algorithm
   2. Heuristic method
   3. Meta-heuristic (genetic algorithms, ant colony, ..)
2. Declarative Approached
   1. (Mixed) Integer Programming,

Solving Methodologies

1. Adhoc methods
   1. Specific exact algorithm
   2. Heuristic method
   3. Meta-heuristic (genetic algorithms, ant colony, ..)
2. Declarative Approached
   1. (Mixed) Integer Programming,
   2. Constraint Programming

Solving Methodologies

1. Adhoc methods
   1. Specific exact algorithm
   2. Heuristic method
   3. Meta-heuristic (genetic algorithms, ant colony, ..)
2. Declarative Approached
   1. (Mixed) Integer Programming,
   2. Constraint Programming
   3. Boolean Satisfiability (SAT)
   4. ...

Why Declarative Approaches?

Solving Methodologies

1. Adhoc methods
   1. Specific exact algorithm
   2. Heuristic method
   3. Meta-heuristic (genetic algorithms, ant colony, ..)
2. Declarative Approached
   1. (Mixed) Integer Programming,
   2. Constraint Programming
   3. Boolean Satisfiability (SAT)
   4. ...

Why Declarative Approaches?

- They are problem independent! The user models the problem in a specific language and the solver do the job!

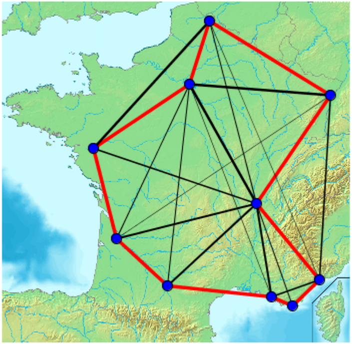
Solving Methodologies

1. Adhoc methods
   1. Specific exact algorithm
   2. Heuristic method
   3. Meta-heuristic (genetic algorithms, ant colony, ..)
2. Declarative Approached
   1. (Mixed) Integer Programming,
   2. Constraint Programming
   3. Boolean Satisfiability (SAT)
   4. ...

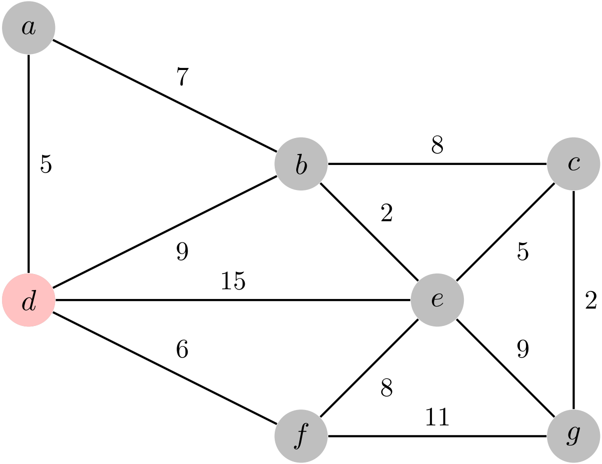
Why Declarative Approaches?

- They are problem independent! The user models the problem in a specific language and the solver do the job!
- Very active community

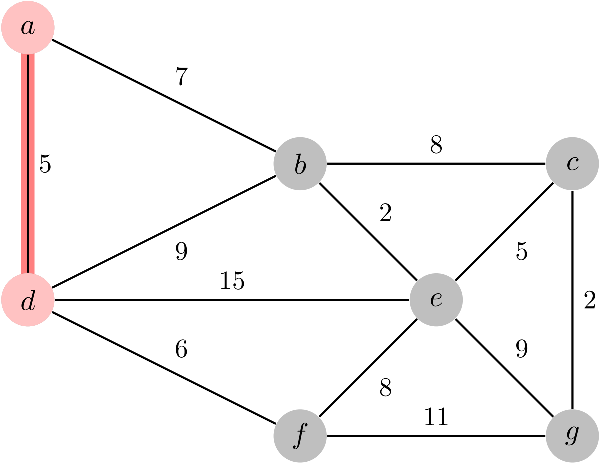
Travelling Salesman Problem



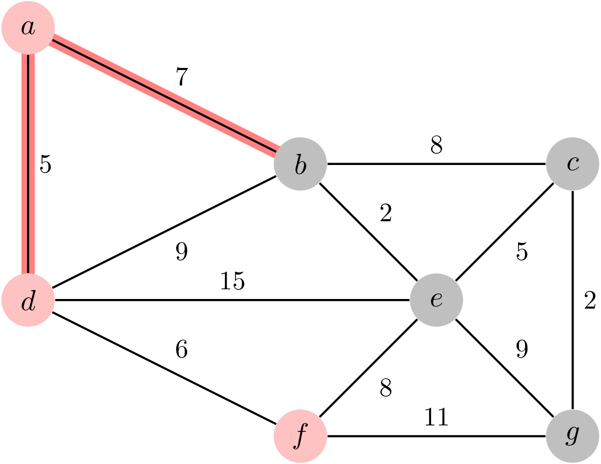
Exemple



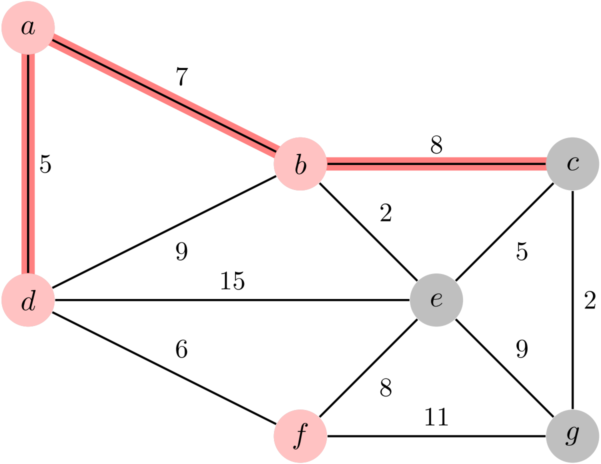
Exemple



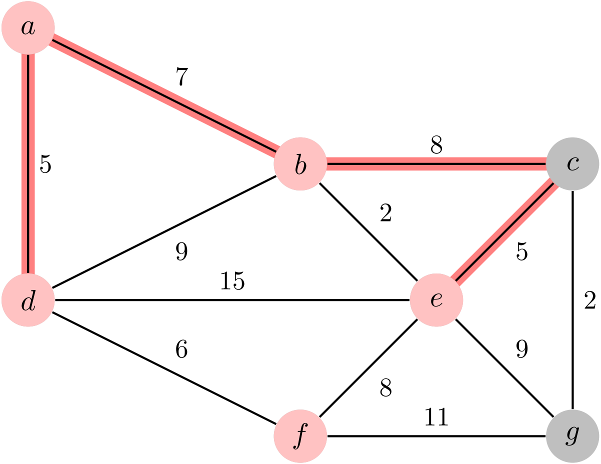
Exemple



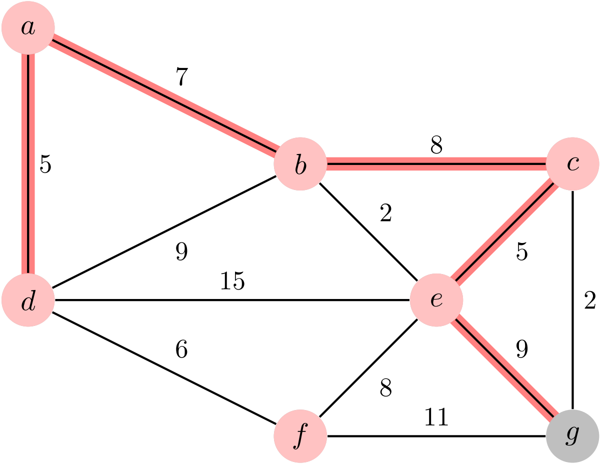
Exemple



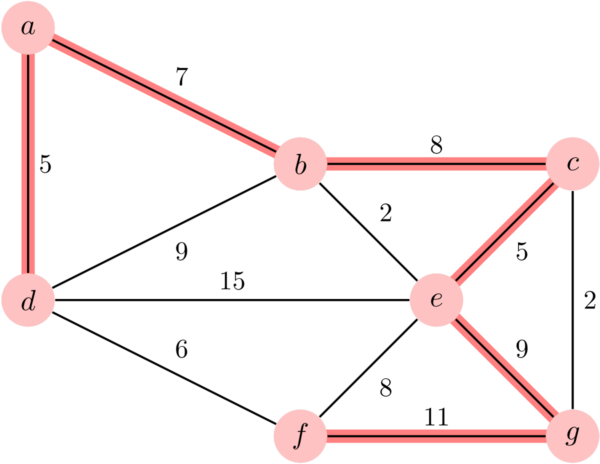
Exemple



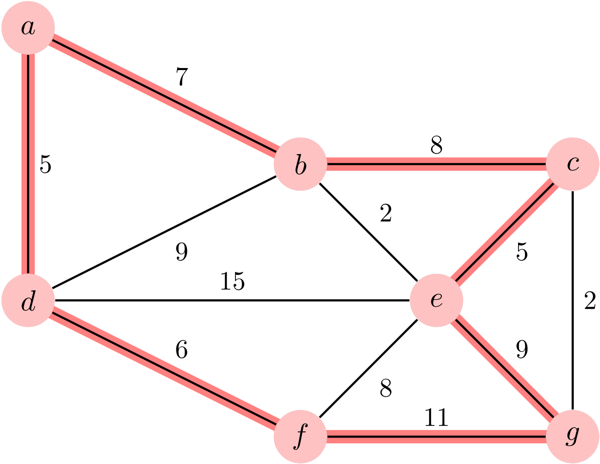
Exemple



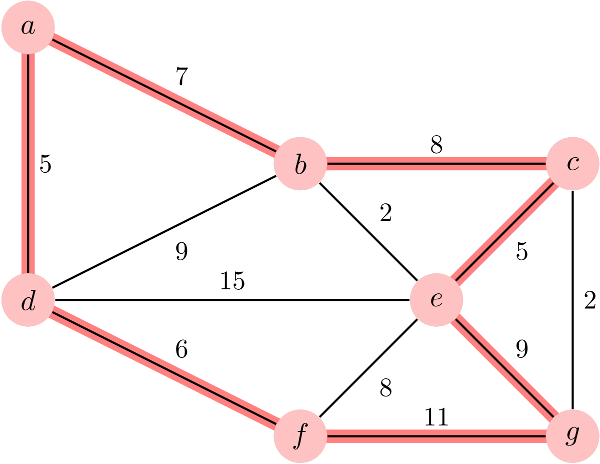
Exemple



Exemple

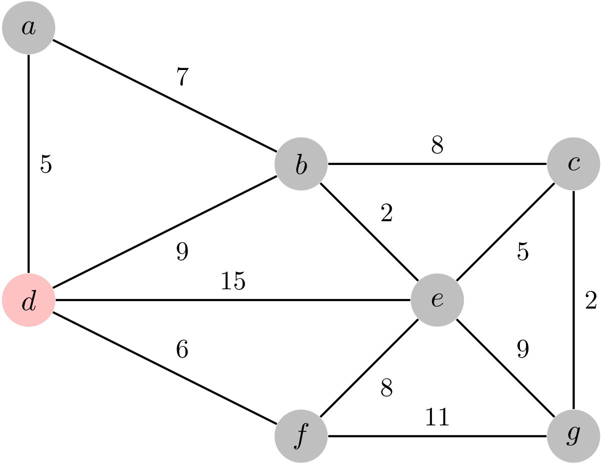


Exemple

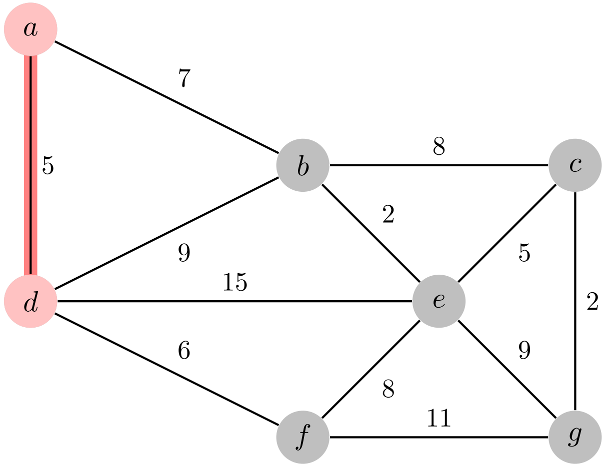


$-- > Cost : 5 + 7 + 8 + 5 + 9 + 11 + 6 = 53Km$

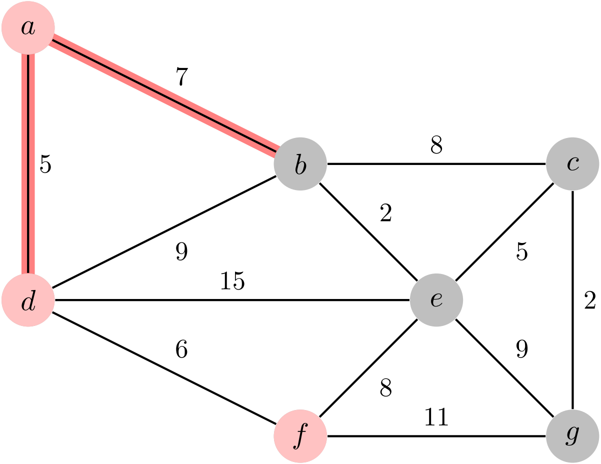
Example



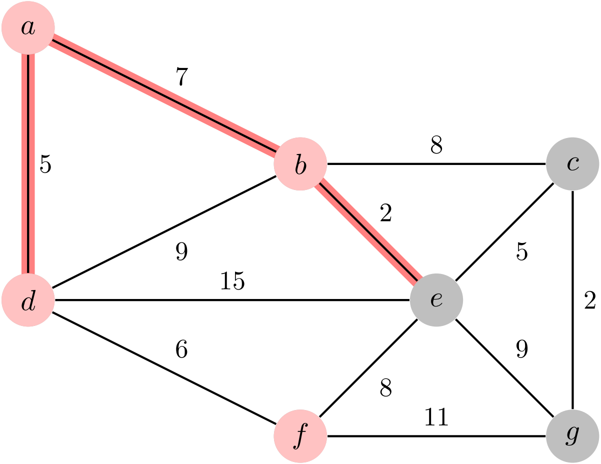
Example



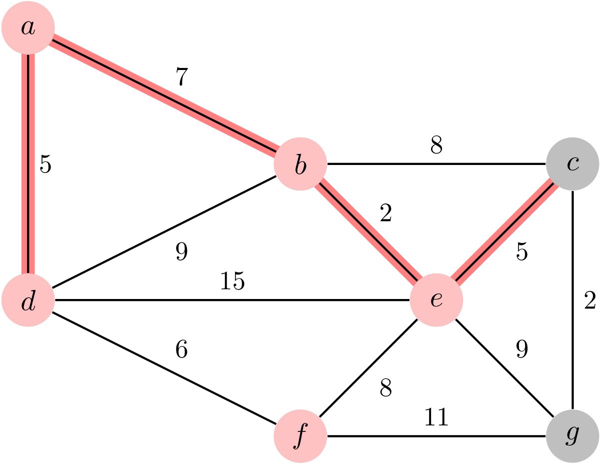
Example



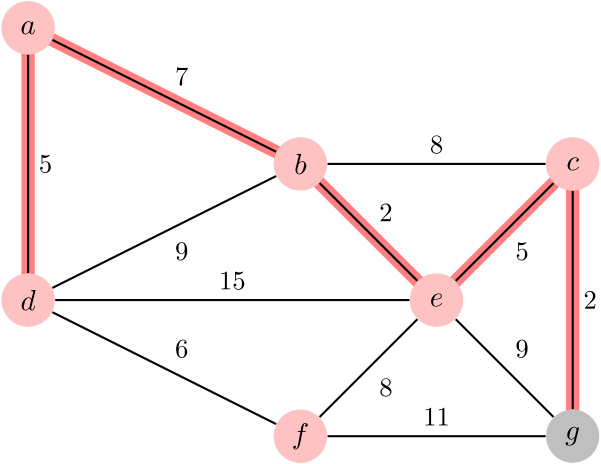
Example



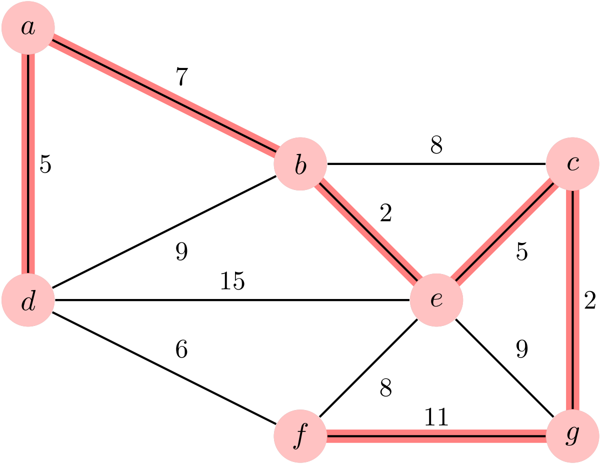
Example



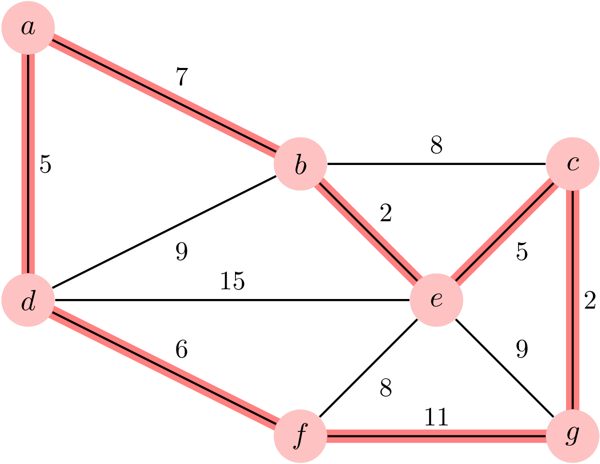
Example



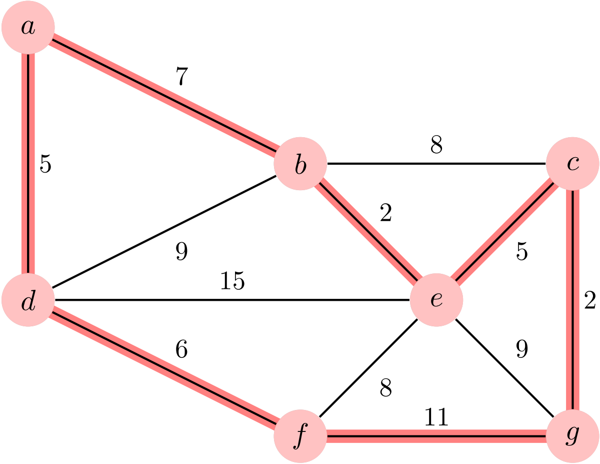
Example



Example



Example



$--> Cost : 5 + 7 + 2 + 5 + 2 + 11 + 6 = 38Km$

What if we check all possibilities?

What if we check all possibilities?

- 2 Cities $\rightarrow$1

What if we check all possibilities?

- 2 Cities $\rightarrow$1
- 5 Cities $\rightarrow$24

What if we check all possibilities?

- 2 Cities $\rightarrow$1
- 5 Cities $\rightarrow$24
- 8 Cities $\rightarrow$4032

What if we check all possibilities?

- 2 Cities $\rightarrow$1
- 5 Cities $\rightarrow$24
- 8 Cities $\rightarrow$4032
- 40 Cities

What if we check all possibilities?

- 2 Cities $\rightarrow$1
- 5 Cities $\rightarrow$24
- 8 Cities $\rightarrow$4032
- 40 Cities $\rightarrow 2.10^{46}$ (with a modern machine: 3.10^{27} years!)

What if we check all possibilities?

- 2 Cities $\rightarrow$1
- 5 Cities $\rightarrow$24
- 8 Cities $\rightarrow$4032
- 40 Cities $\rightarrow 2.10^{46}$ (with a modern machine: 3.10^{27} years!)
- 95 Cities, if we use a Plack (the shortest possible time interval that can be measured) processor and fill the universe with a processor per mm^3, we need $3\times$ the age of the universe

What if we check all possibilities?

- 2 Cities $\rightarrow$1
- 5 Cities $\rightarrow$24
- 8 Cities $\rightarrow$4032
- 40 Cities $\rightarrow 2.10^{46}$ (with a modern machine: 3.10^{27} years!)
- 95 Cities, if we use a Plack (the shortest possible time interval that can be measured) processor and fill the universe with a processor per mm^3, we need $3\times$ the age of the universe

The problem is inherently hard. However, the Concorde algorithm can solve instances up to 86 000 cities!

A step back: Problems, Instances, and Algorithms

A step back: Problems, Instances, and Algorithms

- A problem is a question that associates an input of an output

A step back: Problems, Instances, and Algorithms

- A problem is a question that associates an input of an output
- Many instances (instantiation of the input) for the same problem

A step back: Problems, Instances, and Algorithms

- A problem is a question that associates an input of an output
- Many instances (instantiation of the input) for the same problem
- Many algorithms (methodologies) to solve the same problem

A step back: Problems, Instances, and Algorithms

- A problem is a question that associates an input of an output
- Many instances (instantiation of the input) for the same problem
- Many algorithms (methodologies) to solve the same problem

Example: The Sorting Integers problem

Example: The Sorting Integers problem

- Problem: sort a given sequence of n integers.

Example: The Sorting Integers problem

- Problem: sort a given sequence of n integers.
- Instance: a sequence of n integers

Example: The Sorting Integers problem

- Problem: sort a given sequence of n integers.
- Instance: a sequence of n integers
- A simple algorithm:

Example: The Sorting Integers problem

- Problem: sort a given sequence of n integers.
- Instance: a sequence of n integers
- A simple algorithm:
  - Scan the list to look for the smallest element

Example: The Sorting Integers problem

- Problem: sort a given sequence of n integers.
- Instance: a sequence of n integers
- A simple algorithm:
  - Scan the list to look for the smallest element
  - Swap it with the first position

Example: The Sorting Integers problem

- Problem: sort a given sequence of n integers.
- Instance: a sequence of n integers
- A simple algorithm:
  - Scan the list to look for the smallest element
  - Swap it with the first position
  - Repeat for the list of remaining elements

Example: The Sorting Integers problem

- Problem: sort a given sequence of n integers.
- Instance: a sequence of n integers
- A simple algorithm:
  - Scan the list to look for the smallest element
  - Swap it with the first position
  - Repeat for the list of remaining elements
- Example with the instance : 9, 3, 8, 7, 2

Example: The Sorting Integers problem

- Problem: sort a given sequence of n integers.
- Instance: a sequence of n integers
- A simple algorithm:
  - Scan the list to look for the smallest element
  - Swap it with the first position
  - Repeat for the list of remaining elements
- Example with the instance : 9, 3, 8, 7, 2
  - 2, 9, 3, 8, 7

Example: The Sorting Integers problem

- Problem: sort a given sequence of n integers.
- Instance: a sequence of n integers
- A simple algorithm:
  - Scan the list to look for the smallest element
  - Swap it with the first position
  - Repeat for the list of remaining elements
- Example with the instance : 9, 3, 8, 7, 2
  - 2, 9, 3, 8, 7
  - 2, 3, 9, 8, 7

Example: The Sorting Integers problem

- Problem: sort a given sequence of n integers.
- Instance: a sequence of n integers
- A simple algorithm:
  - Scan the list to look for the smallest element
  - Swap it with the first position
  - Repeat for the list of remaining elements
- Example with the instance : 9, 3, 8, 7, 2
  - 2, 9, 3, 8, 7
  - 2, 3, 9, 8, 7
  - 2, 3, 7, 9, 8

Example: The Sorting Integers problem

- Problem: sort a given sequence of n integers.
- Instance: a sequence of n integers
- A simple algorithm:
  - Scan the list to look for the smallest element
  - Swap it with the first position
  - Repeat for the list of remaining elements
- Example with the instance : 9, 3, 8, 7, 2
  - 2, 9, 3, 8, 7
  - 2, 3, 9, 8, 7
  - 2, 3, 7, 9, 8
  - 2, 3, 7, 8, 9

Example: The Sorting Integers problem

- Problem: sort a given sequence of n integers.
- Instance: a sequence of n integers
- A simple algorithm:
  - Scan the list to look for the smallest element
  - Swap it with the first position
  - Repeat for the list of remaining elements
- Example with the instance : 9, 3, 8, 7, 2
  - 2, 9, 3, 8, 7
  - 2, 3, 9, 8, 7
  - 2, 3, 7, 9, 8
  - 2, 3, 7, 8, 9
  - 2, 3, 7, 8, 9

Complexity

Complexity

- Complexity: a measure to analyze/classify algorithms based on the amount of resource required (Time and Memory)

Complexity

- Complexity: a measure to analyze/classify algorithms based on the amount of resource required (Time and Memory)
- Time Complexity: number of operations as a function of the size of the input

Complexity

- Complexity: a measure to analyze/classify algorithms based on the amount of resource required (Time and Memory)
- Time Complexity: number of operations as a function of the size of the input
- Space Complexity: memory occupied by the algorithm as a function of the size of the input

Complexity

- Complexity: a measure to analyze/classify algorithms based on the amount of resource required (Time and Memory)
- Time Complexity: number of operations as a function of the size of the input
- Space Complexity: memory occupied by the algorithm as a function of the size of the input
- The evaluation is made usually by reasoning about the worst case.

Complexity

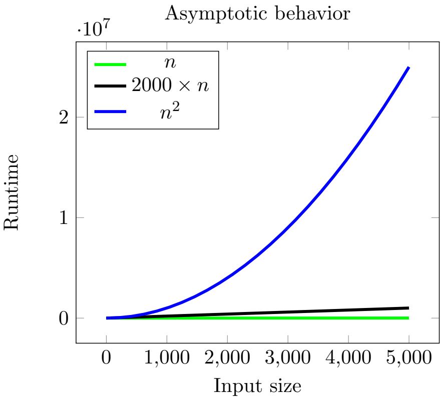
- Complexity: a measure to analyze/classify algorithms based on the amount of resource required (Time and Memory)
- Time Complexity: number of operations as a function of the size of the input
- Space Complexity: memory occupied by the algorithm as a function of the size of the input
- The evaluation is made usually by reasoning about the worst case.
- The analysis is given with regard with the asymptotic behaviour

Complexity

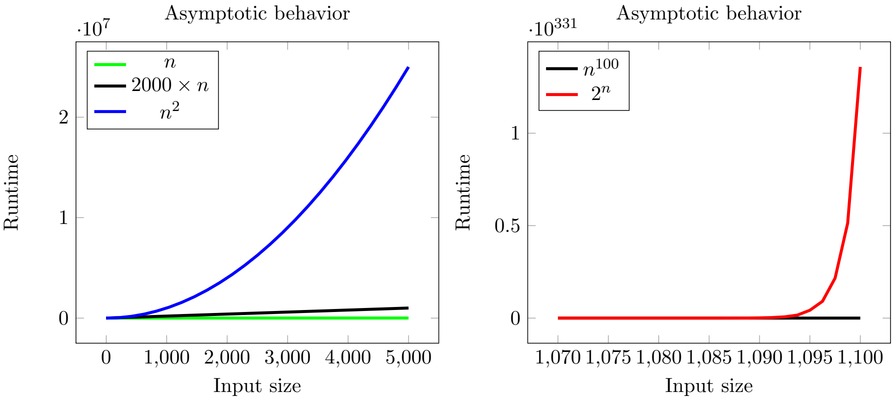
- Complexity: a measure to analyze/classify algorithms based on the amount of resource required (Time and Memory)
- Time Complexity: number of operations as a function of the size of the input
- Space Complexity: memory occupied by the algorithm as a function of the size of the input
- The evaluation is made usually by reasoning about the worst case.
- The analysis is given with regard with the asymptotic behaviour

Asymptotic behaviour

Asymptotic behaviour



Asymptotic behaviour



- If f is a polynomial and g is exponential then $f \in O(g)$.
  For instance $n^{10000} \in O(2^n)$

- If f is a polynomial and g is exponential then $f \in O(g)$.
  For instance $n^{10000} \in O(2^n)$
- Convention:
  - Easy/Tractable Problem: We know a polynomial time algorithm to solve the problem
  - Hard/Intractable: No known polynomial algorithm

- If f is a polynomial and g is exponential then $f \in O(g)$.
  For instance $n^{10000} \in O(2^n)$
- Convention:
  - Easy/Tractable Problem: We know a polynomial time algorithm to solve the problem
  - Hard/Intractable: No known polynomial algorithm
- Example: Th sorting problem is easy because we have an algorithm that runs in the worst case in $O(n^2)$ (and actually the same for memory consumption)

- If f is a polynomial and g is exponential then $f \in O(g)$.
  For instance $n^{10000} \in O(2^n)$
- Convention:
  - Easy/Tractable Problem: We know a polynomial time algorithm to solve the problem
  - Hard/Intractable: No known polynomial algorithm
- Example: Th sorting problem is easy because we have an algorithm that runs in the worst case in $O(n^2)$ (and actually the same for memory consumption)
- What if we don't know if a problem has a polynomial time algorithm?

Classes of problems

Classes of problems

- **P** is the class of problems that are **solvable** in polynomial time (easy problems)

Classes of problems

- **P** is the class of problems that are **solvable** in polynomial time (easy problems)
- **NP** is the class of problems that are **verifiable** in polynomial time algorithm

Classes of problems

- **P** is the class of problems that are **solvable** in polynomial time (easy problems)
- **NP** is the class of problems that are **verifiable** in polynomial time algorithm
- We know that $P \in NP$ (if you can solve then you can verify)

Classes of problems

- **P** is the class of problems that are **solvable** in polynomial time (easy problems)
- **NP** is the class of problems that are **verifiable** in polynomial time algorithm
- We know that $P \in NP$ (if you can solve then you can verify)
- For many Problems in NP, we don't know if a polynomial time algorithm exists.

Classes of problems

- **P** is the class of problems that are **solvable** in polynomial time (easy problems)
- **NP** is the class of problems that are **verifiable** in polynomial time algorithm
- We know that $P \in NP$ (if you can solve then you can verify)
- For many Problems in NP, we don't know if a polynomial time algorithm exists.
- **1 Million $** question: Is P=NP?

The Boolean Satisfiability Problem (SAT)

The Boolean Satisfiability Problem (SAT)

Definitions

- Atoms (Boolean variables): x_1, x_2, …

The Boolean Satisfiability Problem (SAT)

Definitions

- Atoms (Boolean variables): x_1, x_2, ...
- Literal: $x_1, \neg x_1$

The Boolean Satisfiability Problem (SAT)

Definitions

- Atoms (Boolean variables): x_1, x_2, ...
- Literal: $x_1, \neg x_1$
- Clauses: a clause is a disjunction of literals

The Boolean Satisfiability Problem (SAT)

Definitions

- Atoms (Boolean variables): x_1, x_2, ...
- Literal: $x_1, \neg x_1$
- Clauses: a clause is a disjunction of literals
- Example of clause: $(\neg x_1 \vee \neg x_4 \vee x_7)$

The Boolean Satisfiability Problem (SAT)

Definitions

- Atoms (Boolean variables): x_1, x_2, ...
- Literal: $x_1, \neg x_1$
- Clauses: a clause is a disjunction of literals
- Example of clause: $(\neg x_1 \vee \neg x_4 \vee x_7)$
- Propositional formula Φ given in a `Conjunctive Normal Form` (`CNF`) $\Phi : c_1 \wedge .. \wedge c_n$

The Boolean Satisfiability Problem (SAT)

Definitions

- Atoms (Boolean variables): x_1, x_2, ...
- Literal: $x_1, \neg x_1$
- Clauses: a clause is a disjunction of literals
- Example of clause: $(\neg x_1 \vee \neg x_4 \vee x_7)$
- Propositional formula Φ given in a `Conjunctive Normal Form` (`CNF`) $\Phi : c_1 \wedge .. \wedge c_n$

The Boolean Satisfiability Problem (SAT)

Definitions

- Atoms (Boolean variables): x_1, x_2, ...
- Literal: $x_1, \neg x_1$
- Clauses: a clause is a disjunction of literals
- Example of clause: $(\neg x_1 \vee \neg x_4 \vee x_7)$
- Propositional formula Φ given in a `Conjunctive Normal Form` (`CNF`) $\Phi : c_1 \wedge .. \wedge c_n$

Given a set of Boolean variables $x_1, \ldots x_n$ and a CNF formulae Φ over $x_1, \ldots x_n$, the Boolean Satisfiability problem (SAT) is to find an assignment of the variables that satisfies all the clauses.

Why SAT?

Why SAT?

- SAT is the first problem that is shown to be in the class NP-Complete (the hardest problems in NP)

Why SAT?

- SAT is the first problem that is shown to be in the class NP-Complete (the hardest problems in NP)
- Many theoretical properties

Why SAT?

- SAT is the first problem that is shown to be in the class NP-Complete (the hardest problems in NP)
- Many theoretical properties
- Huge practical improvements in the past 2 decades

Why SAT?

- SAT is the first problem that is shown to be in the class NP-Complete (the hardest problems in NP)
- Many theoretical properties
- Huge practical improvements in the past 2 decades
- Is considered today as a powerful technology to solve computational problems

Why SAT?

- SAT is the first problem that is shown to be in the class NP-Complete (the hardest problems in NP)
- Many theoretical properties
- Huge practical improvements in the past 2 decades
- Is considered today as a powerful technology to solve computational problems

In this lecture, we focus on the practical side

Why SAT?

- SAT is the first problem that is shown to be in the class NP-Complete (the hardest problems in NP)
- Many theoretical properties
- Huge practical improvements in the past 2 decades
- Is considered today as a powerful technology to solve computational problems

In this lecture, we focus on the practical side

- How to use it to solve problems (Modelling)

Why SAT?

- SAT is the first problem that is shown to be in the class NP-Complete (the hardest problems in NP)
- Many theoretical properties
- Huge practical improvements in the past 2 decades
- Is considered today as a powerful technology to solve computational problems

In this lecture, we focus on the practical side

- How to use it to solve problems (Modelling)
- Discover some efficient implementations

Example

Example

$x \vee \neg y \vee z$

$\neg x \vee \neg z$

$y \vee w$

$\neg w \vee \neg x$

Example

$x \vee \neg y \vee z$

$\neg x \vee \neg z$

$y \vee w$

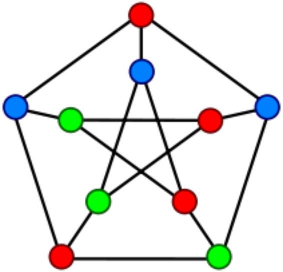
$\neg w \vee \neg x$

A possible solution:

$$x \leftarrow 1; y \leftarrow 1; z \leftarrow 0; w \leftarrow 0$$

The example of Graph Colouring

- Graph Coloring is a well know combinatorial problem that has many applications (in particular in scheduling problems)
- Let $G = (V, E)$ be an undirected graph where V is a set of n vertices and E is a set of m edges. Is it possible to colour the graph with k colours such that no two adjacent nodes share the same colour?



Modelling in SAT: The Example of Graph Coloring

Modelling in SAT: The Example of Graph Coloring

- Propose a SAT model for this problem
  (hint $x \rightarrow y$ is equivalent to $\neg y \rightarrow \neg x$ and both are translated into the clause $\neg x \vee y$).

Modelling in SAT: The Example of Graph Coloring

- Propose a SAT model for this problem
  (hint $x \rightarrow y$ is equivalent to $\neg y \rightarrow \neg x$ and both are translated into the clause $\neg x \vee y$).

The Example of Graph Coloring: A Possible Model

The Example of Graph Coloring: A Possible Model

Let x_i^k be the Boolean variable that is True iff node i is coloured with the colour k.

The Example of Graph Coloring: A Possible Model

Let x_i^k be the Boolean variable that is True iff node i is coloured with the colour k.

- Each node has to be colored with at least one color:

$$\forall i \in [1, n], x_i^1 \vee x_i^2 \dots x_i^k$$

The Example of Graph Coloring: A Possible Model

Let x_i^k be the Boolean variable that is True iff node i is coloured with the colour k.

- Each node has to be colored with at least one color:

$$\forall i \in [1, n], x_i^1 \vee x_i^2 \dots x_i^k$$

- If a node is coloured with a colour a, the other colours are forbidden:

$$\forall i \in [1, n], \forall a \neq b \in [1, k], : \neg x_i^a \vee \neg x_i^b$$

The Example of Graph Coloring: A Possible Model

Let x_i^k be the Boolean variable that is True iff node i is coloured with the colour k.

- Each node has to be colored with at least one color:

$$\forall i \in [1, n], x_i^1 \vee x_i^2 \dots x_i^k$$

- If a node is coloured with a colour a, the other colours are forbidden:

$$\forall i \in [1, n], \forall a \neq b \in [1, k], : \neg x_i^a \vee \neg x_i^b$$

  (This is a translation of $x_i^a \to \neg x_i^b$)

The Example of Graph Coloring: A Possible Model

Let x_i^k be the Boolean variable that is True iff node i is coloured with the colour k.

- Each node has to be colored with at least one color:

$$\forall i \in [1, n], x_i^1 \vee x_i^2 \ldots x_i^k$$

- If a node is coloured with a colour a, the other colours are forbidden:

$$\forall i \in [1, n], \forall a \neq b \in [1, k], : \neg x_i^a \vee \neg x_i^b$$

(This is a translation of $x_i^a \rightarrow \neg x_i^b$)

- Forbid two nodes that share an edge to be coloured with the same colour

$$\forall \{i, j\} \in E, \forall a \in [1, k] : \neg x_i^a \rightarrow \neg x_j^a$$

(This is a translation of $x_i^a \rightarrow \neg x_j^a$)

The Example of Graph Coloring: The Model Size

What is the (space) size of the model?

- $n \times k$ Boolean variables
- Constraints form 1: n clauses with k literals each
- Constraints form 2: $n \times k^2$ binary clauses
- Constraints form 3: $m \times k$ binary clauses

The Example of Graph Coloring: The Minimization Version

- Propose a method that uses SAT for the minimisation version of the problem. That is, given $G = (V, E)$, we seek to find the minimum value of k to satisfy the colouring requirements.

A Straightforward Approach

A Straightforward Approach

- Find a valid upper bound UB and a lower bound LB for k

A Straightforward Approach

- Find a valid upper bound UB and a lower bound LB for k
- Run iteratively the decision version until converging to the optimal value

A Straightforward Approach

- Find a valid upper bound UB and a lower bound LB for k
- Run iteratively the decision version until converging to the optimal value
- Let's call $SAT(V, E, K)$ the SAT model of the decision version of the problem (i.e., can we find a valid colouring of $G(V, E)$ with k colours). Use $SAT(V, E, K)$ as an oracle within an iterative search. For instance:

A Straightforward Approach

- Find a valid upper bound UB and a lower bound LB for k
- Run iteratively the decision version until converging to the optimal value
- Let's call $SAT(V, E, K)$ the SAT model of the decision version of the problem (i.e., can we find a valid colouring of $G(V, E)$ with k colours). Use $SAT(V, E, K)$ as an oracle within an iterative search. For instance:
  - **Decreasing linear Search:** Run iteratively $SAT(V, E, UB - 1), SAT(V, E, UB - 2), \ldots$ until the problem is unsatisfiable. The last satisfiable value of k is the optimal value

A Straightforward Approach

- Find a valid upper bound UB and a lower bound LB for k
- Run iteratively the decision version until converging to the optimal value
- Let's call $SAT(V, E, K)$ the SAT model of the decision version of the problem (i.e., can we find a valid colouring of $G(V, E)$ with k colours). Use $SAT(V, E, K)$ as an oracle within an iterative search. For instance:
  - **Decreasing linear Search:** Run iteratively $SAT(V, E, UB - 1), SAT(V, E, UB - 2), \ldots$ until the problem is unsatisfiable. The last satisfiable value of k is the optimal value
  - **Binary search:** Run iteratively $SAT(V, E, z)$ as long as $UB > LB$ where $z = \lceil (UB - LB)/2 \rceil$. If the result is satisfiable, then and $UB \leftarrow z$ otherwise $LB \leftarrow z$

Upper/Lower Bound?

Upper/Lower Bound?

- Upper bound: For instance, we can run the following iterative greedy algorithm:

Upper/Lower Bound?

- Upper bound: For instance, we can run the following iterative greedy algorithm:
  - Each vertex v is considered non-coloured and has a portfolio S_v of available colours. The set is initially $\{1, 2, \ldots n\}$ for each vertex

Upper/Lower Bound?

- Upper bound: For instance, we can run the following iterative greedy algorithm:
  - Each vertex v is considered non-coloured and has a portfolio S_v of available colours. The set is initially $\{1, 2, \ldots n\}$ for each vertex
  - At leach iteration, look for a non-coloured vertex v that has the greatest number of non coloured neighbours. Colour it with the smallest colour in S_v and remove its colour from all its neighbours.

Upper/Lower Bound?

- Upper bound: For instance, we can run the following iterative greedy algorithm:
  - Each vertex v is considered non-coloured and has a portfolio S_v of available colours. The set is initially $\{1, 2, \ldots n\}$ for each vertex
  - At leach iteration, look for a non-coloured vertex v that has the greatest number of non coloured neighbours. Colour it with the smallest colour in S_v and remove its colour from all its neighbours.
  - The resulting colouring is valid and the the upper bound is the number of different colours used.

Upper/Lower Bound?

- Upper bound: For instance, we can run the following iterative greedy algorithm:
  - Each vertex v is considered non-coloured and has a portfolio S_v of available colours. The set is initially $\{1, 2, \ldots n\}$ for each vertex
  - At leach iteration, look for a non-coloured vertex v that has the greatest number of non coloured neighbours. Colour it with the smallest colour in S_v and remove its colour from all its neighbours.
  - The resulting colouring is valid and the the upper bound is the number of different colours used.
  - The run time complexity is $O(n^2 \times m)$

Upper/Lower Bound?

- Upper bound: For instance, we can run the following iterative greedy algorithm:
  - Each vertex v is considered non-coloured and has a portfolio S_v of available colours. The set is initially $\{1, 2, \ldots n\}$ for each vertex
  - At leach iteration, look for a non-coloured vertex v that has the greatest number of non coloured neighbours. Colour it with the smallest colour in S_v and remove its colour from all its neighbours.
  - The resulting colouring is valid and the the upper bound is the number of different colours used.
  - The run time complexity is $O(n^2 \times m)$
- Lower bound: Well, we can simply consider 2 as long as there is an edge. A more advanced one is to look for a clique in the graph.

Upper/Lower Bound?

- Upper bound: For instance, we can run the following iterative greedy algorithm:
  - Each vertex v is considered non-coloured and has a portfolio S_v of available colours. The set is initially $\{1, 2, \ldots n\}$ for each vertex
  - At leach iteration, look for a non-coloured vertex v that has the greatest number of non coloured neighbours. Colour it with the smallest colour in S_v and remove its colour from all its neighbours.
  - The resulting colouring is valid and the the upper bound is the number of different colours used.
  - The run time complexity is $O(n^2 \times m)$
- Lower bound: Well, we can simply consider 2 as long as there is an edge. A more advanced one is to look for a clique in the graph.

Upper/Lower Bound?

- Upper bound: For instance, we can run the following iterative greedy algorithm:
  - Each vertex v is considered non-coloured and has a portfolio S_v of available colours. The set is initially $\{1, 2, \dots n\}$ for each vertex
  - At leach iteration, look for a non-coloured vertex v that has the greatest number of non coloured neighbours. Colour it with the smallest colour in S_v and remove its colour from all its neighbours.
  - The resulting colouring is valid and the the upper bound is the number of different colours used.
  - The run time complexity is $O(n^2 \times m)$
- Lower bound: Well, we can simply consider 2 as long as there is an edge. A more advanced one is to look for a clique in the graph.
- An alternative approach is to look for valid theoretical bounds in the literature.

Modelling Cardinality Constraints

- The general form of cardinality constraints is the following:

$$a \leq \sum_{1}^{n} x_i \leq b$$

  where a and b are positive integers and $x_1 \ldots x_n$ are Boolean variables
- Cardinality constraints are everywhere!
- Many ways to encode such constraints. See for instance `https://www.carstensinz.de/papers/CP-2005.pdf`

Quadratic encoding for $\sum_1^n x_i = 1$

- At least one constraint:

$$x_1 \vee x_2 \ldots x_n$$

- at most one constraints:

$$\forall i, j : \neg x_i \vee \neg x_j$$

This generates one clause of size n and (n^2) binary clauses without introducing additional variables.

Linear encoding for $\sum_1^n x_i = 1$

New variables are added as follows: for $i \in [1, n], y_i$ is a new variable that is true iff $\sum_{l=1}^{l=i} x_l = 1$.

$$x_1 \vee x_2 \dots x_n$$

$$y_n^1$$

$$\forall i \in [1, n-1] : y_i \rightarrow y_{i+1}$$

$$\forall i \in [1, n-1] : y_i \rightarrow \neg x_{i+1}$$

$$\forall i \in [1, n] : x_i \rightarrow y_i$$

Size: n new variables, 1 $n-$ary clause and $3 \times n$ binary clauses,

Linear encoding for $\sum_1^n x_i \geq k$

New variables: $\forall z \in [0, k], \forall i \in [1, n], y_i^z \iff \sum_{l=1}^{l=i} x_l \geq z$

Linear encoding for $\sum_1^n x_i \geq k$

New variables: $\forall z \in [0, k], \forall i \in [1, n], y_i^z \iff \sum_{l=1}^{l=i} x_l \geq z$

$$\forall i \in [0, n] : y_i^0 \leftarrow 1$$

Linear encoding for $\sum_1^n x_i \geq k$

New variables: $\forall z \in [0,k], \forall i \in [1,n], y_i^z \iff \sum_{l=1}^{l=i} x_l \geq z$

$$\forall i \in [0,n] : y_i^0 \leftarrow 1$$

$$y_1^1 \leftarrow x_1 \; and \; \forall z \in [2,k], y_1^z \leftarrow 0$$
$$y_n^k \leftarrow 1$$

Linear encoding for $\sum_1^n x_i \geq k$

New variables: $\forall z \in [0, k], \forall i \in [1, n], y_i^z \iff \sum_{l=1}^{l=i} x_l \geq z$

$$\forall i \in [0, n] : y_i^0 \leftarrow 1$$

$$y_1^1 \leftarrow x_1 \; and \; \forall z \in [2, k], y_1^z \leftarrow 0$$
$$y_n^k \leftarrow 1$$

$$\forall i \in [1, n], \forall z \in [1, k-1] : y_i^{z+1} \rightarrow y_i^z$$

Linear encoding for $\sum_1^n x_i \geq k$

New variables: $\forall z \in [0,k], \forall i \in [1,n], y_i^z \iff \sum_{l=1}^{l=i} x_l \geq z$

$$\forall i \in [0,n] : y_i^0 \leftarrow 1$$

$$y_1^1 \leftarrow x_1 \ and \ \forall z \in [2,k], y_1^z \leftarrow 0$$
$$y_n^k \leftarrow 1$$

$$\forall i \in [1,n], \forall z \in [1,k-1] : y_i^{z+1} \rightarrow y_i^z$$

$$\forall i \in [1,n-1], \forall z \in [1,k] : y_i^z \rightarrow y_{i+1}^z$$

Linear encoding for $\sum_1^n x_i \geq k$

New variables: $\forall z \in [0,k], \forall i \in [1,n], y_i^z \iff \sum_{l=1}^{l=i} x_l \geq z$

$$\forall i \in [0,n] : y_i^0 \leftarrow 1$$

$$y_1^1 \leftarrow x_1 \ and\ \forall z \in [2,k], y_1^z \leftarrow 0$$
$$y_n^k \leftarrow 1$$

$$\forall i \in [1,n], \forall z \in [1,k-1] : y_i^{z+1} \rightarrow y_i^z$$

$$\forall i \in [1,n-1], \forall z \in [1,k] : y_i^z \rightarrow y_{i+1}^z$$

$$\neg y_{i-1}^z \rightarrow \neg y_i^{z+1}$$

Linear encoding for $\sum_1^n x_i \geq k$

New variables: $\forall z \in [0, k], \forall i \in [1, n], y_i^z \iff \sum_{l=1}^{l=i} x_l \geq z$

$$\forall i \in [0, n] : y_i^0 \leftarrow 1$$

$$y_1^1 \leftarrow x_1 \; and \; \forall z \in [2, k], y_1^z \leftarrow 0$$
$$y_n^k \leftarrow 1$$

$$\forall i \in [1, n], \forall z \in [1, k-1] : y_i^{z+1} \rightarrow y_i^z$$

$$\forall i \in [1, n-1], \forall z \in [1, k] : y_i^z \rightarrow y_{i+1}^z$$

$$\neg y_{i-1}^z \rightarrow \neg y_i^{z+1}$$

$$y_{i-1}^z \wedge x_i \rightarrow y_i^{z+1}$$

Linear encoding for $\sum_1^n x_i \geq k$

Size of the encoding:

- $\Theta(n \times k)$ variables
- $\Theta(n + k)$ unary clauses
- $\Theta(n \times k)$ binary clauses
- $\Theta(n \times k)$ ternary clauses

Linear encoding for $\sum_1^n x_i = k$?

Linear encoding for $\sum_1^n x_i = k$?

Linear encoding for $\sum_1^n x_i = k$?

- Encode $\sum_1^n x_i \geq k + 1$

Linear encoding for $\sum_1^n x_i = k$?

- Encode $\sum_1^n x_i \geq k+1$
- Force y_n^{k+1} to be false and y_n^k to be true

Size of the encoding: Same as $\sum_1^n x_i \geq k$ (asymptotically)

Linear encoding for $\sum_1^n x_i \leq k$?

Linear encoding for $\sum_1^n x_i \leq k$?

- Encode $\sum_1^n x_i \geq k + 1$

Linear encoding for $\sum_1^n x_i \leq k$?

- Encode $\sum_1^n x_i \geq k+1$
- Force y_n^{k+1} to be false

Size of the encoding: Same as $\sum_1^n x_i \geq k$ (asymptotically)

Linear encoding for $a \leq \sum_1^n x_i \leq b$?

Linear encoding for $a \leq \sum_1^n x_i \leq b$?

- Encode $\sum_1^n x_i \leq b$

Linear encoding for $a \leq \sum_1^n x_i \leq b$?

- Encode $\sum_1^n x_i \leq b$
- Force y_n^a to be true

Size of the encoding: Same as $\sum_1^n x_i \geq b$ (asymptotically)

Extensions: MaxSAT

Extensions: MaxSAT

- MaxSAT is an optimisation extension of SAT where some clauses are "hard" (must be satisfied) and others are "soft" (can be violated).

Extensions: MaxSAT

- MaxSAT is an optimisation extension of SAT where some clauses are "hard" (must be satisfied) and others are "soft" (can be violated).
- The task is to find an assignment of the variables that satisfy the hard clauses and maximises the number of "soft" clauses

Extensions: MaxSAT

- MaxSAT is an optimisation extension of SAT where some clauses are "hard" (must be satisfied) and others are "soft" (can be violated).
- The task is to find an assignment of the variables that satisfy the hard clauses and maximises the number of "soft" clauses
- MaxSAT:

Extensions: MaxSAT

- MaxSAT is an optimisation extension of SAT where some clauses are "hard" (must be satisfied) and others are "soft" (can be violated).
- The task is to find an assignment of the variables that satisfy the hard clauses and maximises the number of "soft" clauses
- MaxSAT:
    - Variables: Booleans, Clauses: hard and soft clauses

Extensions: MaxSAT

- MaxSAT is an optimisation extension of SAT where some clauses are "hard" (must be satisfied) and others are "soft" (can be violated).
- The task is to find an assignment of the variables that satisfy the hard clauses and maximises the number of "soft" clauses
- MaxSAT:
  - Variables: Booleans, Clauses: hard and soft clauses

Extensions: MaxSAT

- MaxSAT is an optimisation extension of SAT where some clauses are "hard" (must be satisfied) and others are "soft" (can be violated).
- The task is to find an assignment of the variables that satisfy the hard clauses and maximises the number of "soft" clauses
- MaxSAT:
  - Variables: Booleans, Clauses: hard and soft clauses
  - Maximisation problem: Is there an assignment of the variables that satisfy all the hard clauses, and maximises the number of satisfied soft clauses?

Extensions: MaxSAT

- MaxSAT is an optimisation extension of SAT where some clauses are "hard" (must be satisfied) and others are "soft" (can be violated).
- The task is to find an assignment of the variables that satisfy the hard clauses and maximises the number of "soft" clauses
- MaxSAT:
  - Variables: Booleans, Clauses: hard and soft clauses
  - Maximisation problem: Is there an assignment of the variables that satisfy all the hard clauses, and maximises the number of satisfied soft clauses?
- Weighted MaxSAT: Extension of MaxSAT where every soft clause is associated with a weight

Extensions: MaxSAT

- MaxSAT is an optimisation extension of SAT where some clauses are "hard" (must be satisfied) and others are "soft" (can be violated).
- The task is to find an assignment of the variables that satisfy the hard clauses and maximises the number of "soft" clauses
- MaxSAT:
  - Variables: Booleans, Clauses: hard and soft clauses
  - Maximisation problem: Is there an assignment of the variables that satisfy all the hard clauses, and maximises the number of satisfied soft clauses?
- Weighted MaxSAT: Extension of MaxSAT where every soft clause is associated with a weight
- Objective: satisfy hard clauses and maximizes the weighted sum of satisfied soft clauses.

Extensions: MaxSAT

- MaxSAT is an optimisation extension of SAT where some clauses are "hard" (must be satisfied) and others are "soft" (can be violated).
- The task is to find an assignment of the variables that satisfy the hard clauses and maximises the number of "soft" clauses
- MaxSAT:
  - Variables: Booleans, Clauses: hard and soft clauses
  - Maximisation problem: Is there an assignment of the variables that satisfy all the hard clauses, and maximises the number of satisfied soft clauses?
- Weighted MaxSAT: Extension of MaxSAT where every soft clause is associated with a weight
- Objective: satisfy hard clauses and maximizes the weighted sum of satisfied soft clauses.
- Check the MaxSAT competition

Example of applications for MaxSAT

Let $G = (V, E)$ be an undirected graph where V is the set of vertices and E is the set of edges. In the (decision version of the) graph colouring problem, we are given k colours to colour the graph such that no two adjacent nodes share the same colour.

- Propose a MaxSAT model for the minimisation version of the problem. That is, given $G = (V, E)$, we seek to find the minimum value of k to satisfy the colouring requirements.

Example of applications for MaxSAT

Let $G = (V, E)$ be an undirected graph where V is the set of vertices and E is the set of edges. In the (decision version of the) graph colouring problem, we are given k colours to colour the graph such that no two adjacent nodes share the same colour.

- Propose a MaxSAT model for the minimisation version of the problem. That is, given $G = (V, E)$, we seek to find the minimum value of k to satisfy the colouring requirements.

The Example of Graph Coloring: A Possible MaxSAT Model

- We shall extend the previous model:
- Consider the previous model $SAT(V, E, k)$ with k an upper bound.
- All the previous clauses are hard.
- Add the following hard clauses:

$$\forall i \in [1, n], \forall a \in [1, k] : \neg u_a \rightarrow \neg x_i^a$$

- Eventually we can add symmetry constraints: $u_a \rightarrow u_{a-1}$
- Then add the soft clauses:

$$\forall a \in [1, k] : \neg u_a$$

- A MaxSAT Optimal solution satisfies all the hard coloring clauses (valid colouring) and maximizes the number of non used colours.

Extensions: Quantified Boolean Formula (QBF)

- A QBF has the form Q.F , where F is a CNF-SAT formulae, and Q is a sequence of quantified variables ($\forall x$ or $\exists x$).
- Example $\forall x, \exists y, \exists z,\ (x \vee \neg y) \wedge (\neg y \vee z)$
- QBF Solver Competition: `https://www.qbflib.org/solvers_list.php`
- QBF is less used in practice

Other Extensions

- Satisfiability Modulo Theories
- Answer Set Programming
- More generally: Automated reasoning community
- Check the SAT/SMT summer schools
  `http://satassociation.org/sat-smt-school.html`

Modern SAT Solvers: Conflict Driven Clause Learning (CDCL)

Modern SAT Solvers: Conflict Driven Clause Learning (CDCL)

- [Silva and Sakallah, 1999, Moskewicz et al., 2001]

Modern SAT Solvers: Conflict Driven Clause Learning (CDCL)

- [Silva and Sakallah, 1999, Moskewicz et al., 2001]
- DPLL [Davis et al., 1962] $\oplus$ Resolution [Robinson, 1965]

Modern SAT Solvers: Conflict Driven Clause Learning (CDCL)

- [Silva and Sakallah, 1999, Moskewicz et al., 2001]
- DPLL [Davis et al., 1962] $\oplus$ Resolution [Robinson, 1965]
- DPLL: Backtracking + Unit Propagation

Modern SAT Solvers: Conflict Driven Clause Learning (CDCL)

- [Silva and Sakallah, 1999, Moskewicz et al., 2001]
- DPLL [Davis et al., 1962] $\oplus$ Resolution [Robinson, 1965]
- DPLL: Backtracking + Unit Propagation
- Resolution: Learning based on the rule
  $(l \vee c_1) \wedge (\neg l \vee c_2) \Rightarrow (c_1 \vee c_2)$

Modern SAT Solvers: Conflict Driven Clause Learning (CDCL)

- [Silva and Sakallah, 1999, Moskewicz et al., 2001]
- DPLL [Davis et al., 1962] $\oplus$ Resolution [Robinson, 1965]
- DPLL: Backtracking + Unit Propagation
- Resolution: Learning based on the rule
  $(l \vee c_1) \wedge (\neg l \vee c_2) \Rightarrow (c_1 \vee c_2)$
- **Can be seen as a CP Solver (Search, propagation) augmented by clause learning**

Modern SAT Solvers: Conflict Driven Clause Learning (CDCL)

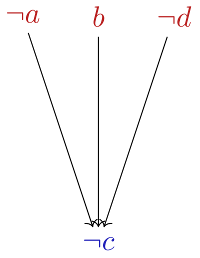
- [Silva and Sakallah, 1999, Moskewicz et al., 2001]
- DPLL [Davis et al., 1962] $\oplus$ Resolution [Robinson, 1965]
- DPLL: Backtracking + Unit Propagation
- Resolution: Learning based on the rule
  $(l \vee c_1) \wedge (\neg l \vee c_2) \Rightarrow (c_1 \vee c_2)$
- **Can be seen as a CP Solver (Search, propagation) augmented by clause learning**
- But also :
  - Activity-based branching
  - Lazy data structures (2-Watched Literals)
  - Clause Database Reduction
  - Simplifications
  - Restarts
  - ...

Exercise: Propose a filtering algorithm for clauses. The algorithm takes as input a clause and has access (read and write) for the variables domains.

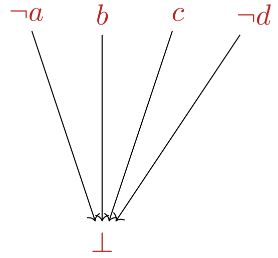
Unit Propagation

Given a clause C of arity n. If $n-1$ literals are false then set the last one to be true.

Example: $(a \vee \neg b \vee \neg c \vee d)$



$\neg a \wedge b \wedge \neg d \Rightarrow \neg c$



$\neg a \wedge b \wedge c \wedge \neg d \Rightarrow \perp$

Two Watched Literals

- Unit propagation is implemented with an “intelligent” data structure called Two-watched literals
- Observe first that propagation happens only in two cases:
  - The clause becomes unit (i.e., all variables except one is instantiated): Propagate the only uninstantiated literal to satisfy the clause
  - All literals are instantiated and none of them satisfy the clause

Two Watched Literals

- Unit propagation is implemented with an "intelligent" data structure called Two-watched literals
- Observe first that propagation happens only in two cases:
  - The clause becomes unit (i.e., all variables except one is instantiated): Propagate the only uninstantiated literal to satisfy the clause
  - All literals are instantiated and none of them satisfy the clause
- Therefore for each clause C, as long as there are two literals non instantiated in C, nothing happens!

Two Watched Literals

- Unit propagation is implemented with an “intelligent” data structure called Two-watched literals
- Observe first that propagation happens only in two cases:
  - The clause becomes unit (i.e., all variables except one is instantiated): Propagate the only uninstantiated literal to satisfy the clause
  - All literals are instantiated and none of them satisfy the clause
- Therefore for each clause C, as long as there are two literals non instantiated in C, nothing happens!
- The idea of the Two-watched literals is to keep 2 literals for every clause that are not instantiated. Those literals will “watch the clause” and guarantee that no propagation is needed.

Two Watched Literals

- Unit propagation is implemented with an "intelligent" data structure called Two-watched literals
- Observe first that propagation happens only in two cases:
  - The clause becomes unit (i.e., all variables except one is instantiated): Propagate the only uninstantiated literal to satisfy the clause
  - All literals are instantiated and none of them satisfy the clause
- Therefore for each clause C, as long as there are two literals non instantiated in C, nothing happens!
- The idea of the Two-watched literals is to keep 2 literals for every clause that are not instantiated. Those literals will "watch the clause" and guarantee that no propagation is needed.
- If a literal watching a clause C becomes $false$, look for replacement. If no replacement found, then perform propagation

Implication Graph

	f		

$\neg a \vee \neg f \vee g$

$\neg a \vee \neg b \vee \neg h$

$a \vee c$

$a \vee \neg i \vee \neg l$

$a \vee \neg k \vee \neg j$

$b \vee d$

$b \vee g \vee \neg n$

$b \vee \neg f \vee n \vee k$

$\neg c \vee k$

$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$

$c \vee l$

$d \vee \neg k \vee l$

$d \vee \neg g \vee l$

$\neg g \vee n \vee o$

$h \vee \neg o \vee \neg j \vee n$

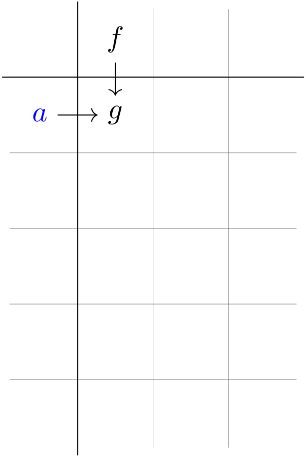
$\neg i \vee j$

$\neg d \vee \neg l \vee \neg m$

$\neg e \vee m \vee \neg n$

$\neg f \vee h \vee i$

Implication Graph



$\neg a \vee \neg f \vee g$

$\neg a \vee \neg b \vee \neg h$

$a \vee c$

$a \vee \neg i \vee \neg l$

$a \vee \neg k \vee \neg j$

$b \vee d$

$b \vee g \vee \neg n$

$b \vee \neg f \vee n \vee k$

$\neg c \vee k$

$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$

$c \vee l$

$d \vee \neg k \vee l$

$d \vee \neg g \vee l$

$\neg g \vee n \vee o$

$h \vee \neg o \vee \neg j \vee n$

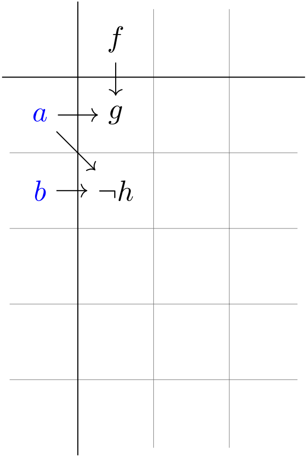
$\neg i \vee j$

$\neg d \vee \neg l \vee \neg m$

$\neg e \vee m \vee \neg n$

$\neg f \vee h \vee i$

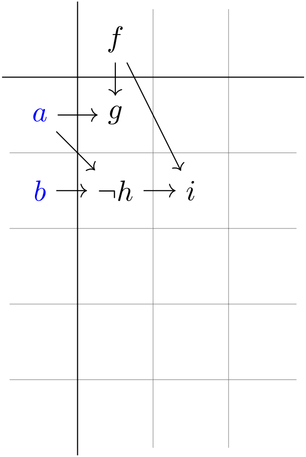
Implication Graph



$\neg a \vee \neg f \vee g$
$\neg a \vee \neg b \vee \neg h$
$a \vee c$
$a \vee \neg i \vee \neg l$
$a \vee \neg k \vee \neg j$
$b \vee d$
$b \vee g \vee \neg n$
$b \vee \neg f \vee n \vee k$
$\neg c \vee k$
$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$
$c \vee l$
$d \vee \neg k \vee l$
$d \vee \neg g \vee l$
$\neg g \vee n \vee o$
$h \vee \neg o \vee \neg j \vee n$
$\neg i \vee j$
$\neg d \vee \neg l \vee \neg m$
$\neg e \vee m \vee \neg n$
$\neg f \vee h \vee i$

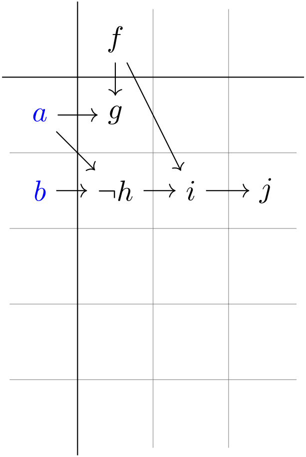
Implication Graph



$\neg a \vee \neg f \vee g$
$\neg a \vee \neg b \vee \neg h$
$a \vee c$
$a \vee \neg i \vee \neg l$
$a \vee \neg k \vee \neg j$
$b \vee d$
$b \vee g \vee \neg n$
$b \vee \neg f \vee n \vee k$
$\neg c \vee k$
$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$
$c \vee l$
$d \vee \neg k \vee l$
$d \vee \neg g \vee l$
$\neg g \vee n \vee o$
$h \vee \neg o \vee \neg j \vee n$
$\neg i \vee j$
$\neg d \vee \neg l \vee \neg m$
$\neg e \vee m \vee \neg n$
$\neg f \vee h \vee i$

Implication Graph



$\neg a \vee \neg f \vee g$

$\neg a \vee \neg b \vee \neg h$

$a \vee c$

$a \vee \neg i \vee \neg l$

$a \vee \neg k \vee \neg j$

$b \vee d$

$b \vee g \vee \neg n$

$b \vee \neg f \vee n \vee k$

$\neg c \vee k$

$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$

$c \vee l$

$d \vee \neg k \vee l$

$d \vee \neg g \vee l$

$\neg g \vee n \vee o$

$h \vee \neg o \vee \neg j \vee n$

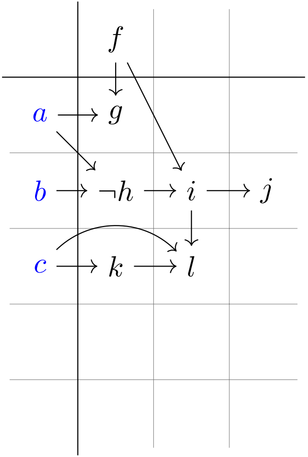
$\neg i \vee j$

$\neg d \vee \neg l \vee \neg m$

$\neg e \vee m \vee \neg n$

$\neg f \vee h \vee i$

Implication Graph



$\neg a \vee \neg f \vee g$

$\neg a \vee \neg b \vee \neg h$

$a \vee c$

$a \vee \neg i \vee \neg l$

$a \vee \neg k \vee \neg j$

$b \vee d$

$b \vee g \vee \neg n$

$b \vee \neg f \vee n \vee k$

$\neg c \vee k$

$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$

$c \vee l$

$d \vee \neg k \vee l$

$d \vee \neg g \vee l$

$\neg g \vee n \vee o$

$h \vee \neg o \vee \neg j \vee n$

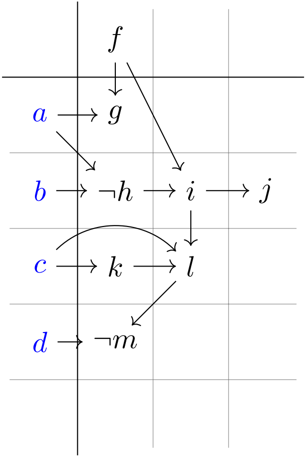
$\neg i \vee j$

$\neg d \vee \neg l \vee \neg m$

$\neg e \vee m \vee \neg n$

$\neg f \vee h \vee i$

Implication Graph



$\neg a \vee \neg f \vee g$

$\neg a \vee \neg b \vee \neg h$

$a \vee c$

$a \vee \neg i \vee \neg l$

$a \vee \neg k \vee \neg j$

$b \vee d$

$b \vee g \vee \neg n$

$b \vee \neg f \vee n \vee k$

$\neg c \vee k$

$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$

$c \vee l$

$d \vee \neg k \vee l$

$d \vee \neg g \vee l$

$\neg g \vee n \vee o$

$h \vee \neg o \vee \neg j \vee n$

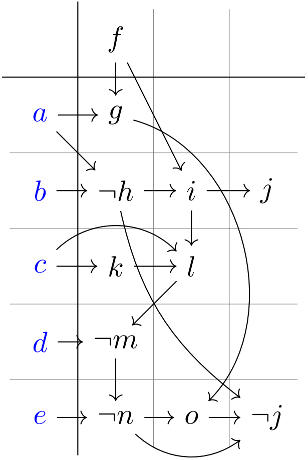
$\neg i \vee j$

$\neg d \vee \neg l \vee \neg m$

$\neg e \vee m \vee \neg n$

$\neg f \vee h \vee i$

Implication Graph



$\neg a \vee \neg f \vee g$

$\neg a \vee \neg b \vee \neg h$

$a \vee c$

$a \vee \neg i \vee \neg l$

$a \vee \neg k \vee \neg j$

$b \vee d$

$b \vee g \vee \neg n$

$b \vee \neg f \vee n \vee k$

$\neg c \vee k$

$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$

$c \vee l$

$d \vee \neg k \vee l$

$d \vee \neg g \vee l$

$\neg g \vee n \vee o$

$h \vee \neg o \vee \neg j \vee n$

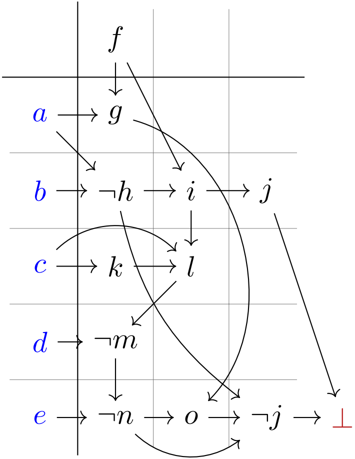
$\neg i \vee j$

$\neg d \vee \neg l \vee \neg m$

$\neg e \vee m \vee \neg n$

$\neg f \vee h \vee i$

Implication Graph



$\neg a \vee \neg f \vee g$

$\neg a \vee \neg b \vee \neg h$

$a \vee c$

$a \vee \neg i \vee \neg l$

$a \vee \neg k \vee \neg j$

$b \vee d$

$b \vee g \vee \neg n$

$b \vee \neg f \vee n \vee k$

$\neg c \vee k$

$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$

$c \vee l$

$d \vee \neg k \vee l$

$d \vee \neg g \vee l$

$\neg g \vee n \vee o$

$\boxed{h \vee \neg o \vee \neg j \vee n}$

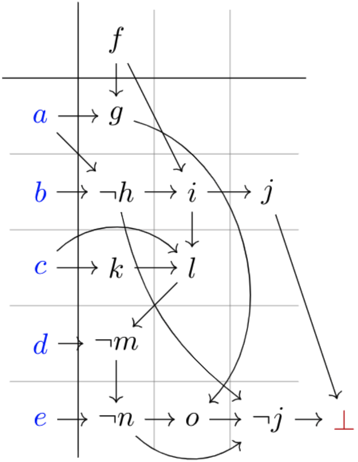
$\neg i \vee j$

$\neg d \vee \neg l \vee \neg m$

$\neg e \vee m \vee \neg n$

$\neg f \vee h \vee i$

Implication Graph



$\neg a \vee \neg f \vee g$

$\neg a \vee \neg b \vee \neg h$

$a \vee c$

$a \vee \neg i \vee \neg l$

$a \vee \neg k \vee \neg j$

$b \vee d$

$b \vee g \vee \neg n$

$b \vee \neg f \vee n \vee k$

$\neg c \vee k$

$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$

$c \vee l$

$d \vee \neg k \vee l$

$d \vee \neg g \vee l$

$\neg g \vee n \vee o$

$\boxed{h \vee \neg o \vee \neg j \vee n}$

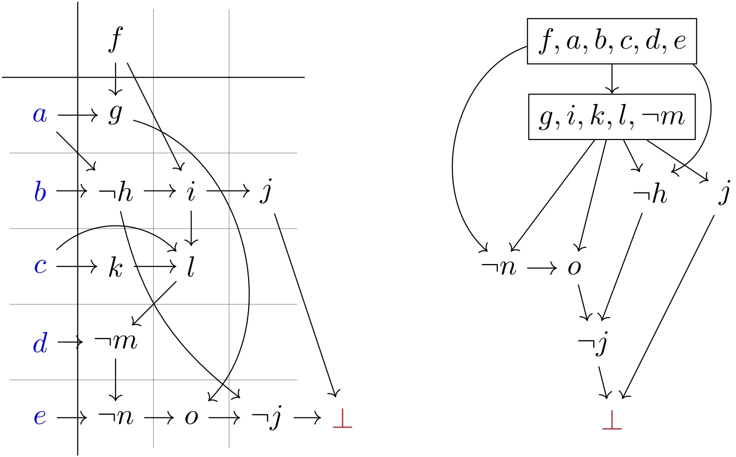
$\neg i \vee j$

$\neg d \vee \neg l \vee \neg m$

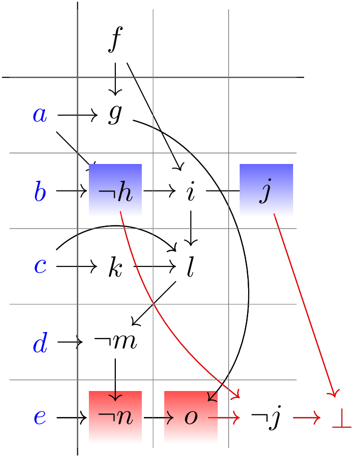
$\neg e \vee m \vee \neg n$

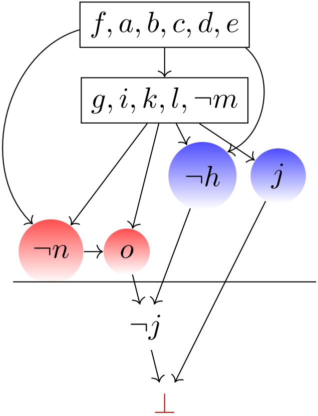
$\neg f \vee h \vee i$

Conflict Analysis

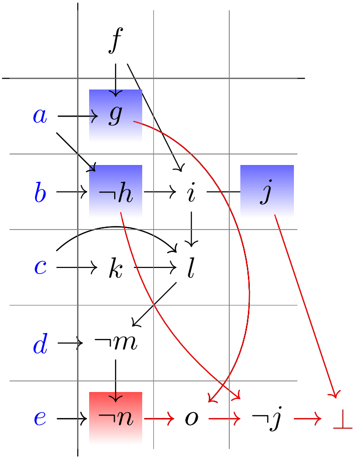


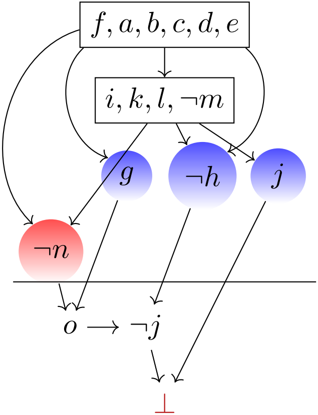
Conflict Analysis



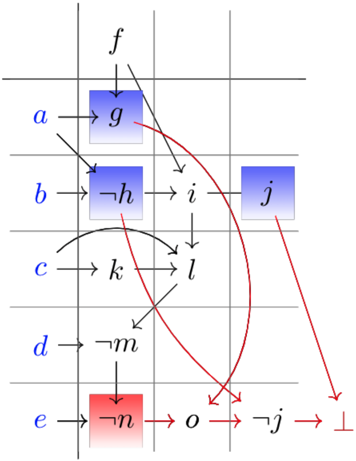


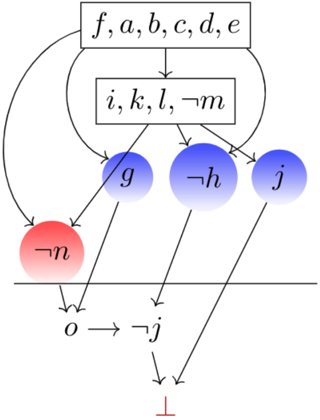
Conflict Analysis



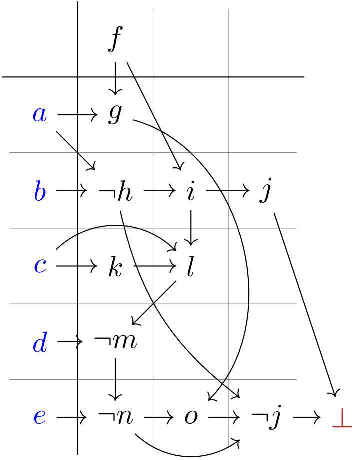


Conflict Analysis





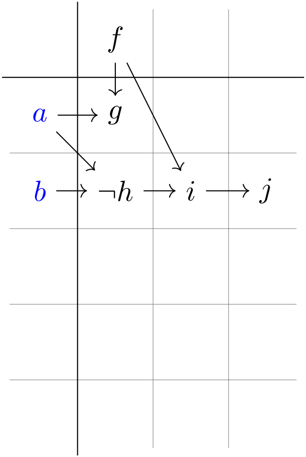
Conflict analysis



$\neg a \vee \neg f \vee g$
$\neg a \vee \neg b \vee \neg h$
$a \vee c$
$a \vee \neg i \vee \neg l$
$a \vee \neg k \vee \neg j$
$b \vee d$
$b \vee g \vee \neg n$
$b \vee \neg f \vee n \vee k$
$\neg c \vee k$
$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$
$c \vee l$
$d \vee \neg k \vee l$
$d \vee \neg g \vee l$
$\neg g \vee n \vee o$
$h \vee \neg o \vee \neg j \vee n$
$\neg i \vee j$
$\neg d \vee \neg l \vee \neg m$
$\neg e \vee m \vee \neg n$
$\neg f \vee h \vee i$

Conflict analysis



$\neg a \vee \neg f \vee g$

$\neg a \vee \neg b \vee \neg h$

$a \vee c$

$a \vee \neg i \vee \neg l$

$a \vee \neg k \vee \neg j$

$b \vee d$

$b \vee g \vee \neg n$

$b \vee \neg f \vee n \vee k$

$\neg c \vee k$

$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$

$c \vee l$

$d \vee \neg k \vee l$

$d \vee \neg g \vee l$

$\neg g \vee n \vee o$

$h \vee \neg o \vee \neg j \vee n$

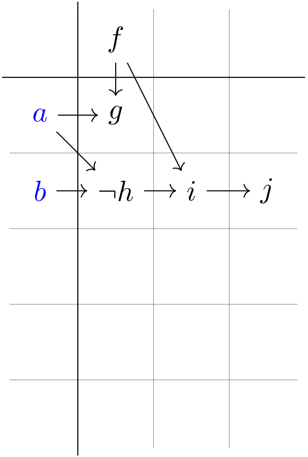
$\neg i \vee j$

$\neg d \vee \neg l \vee \neg m$

$\neg e \vee m \vee \neg n$

$\neg f \vee h \vee i$

Conflict analysis



$\neg a \vee \neg f \vee g$

$\neg a \vee \neg b \vee \neg h$

$a \vee c$

$a \vee \neg i \vee \neg l$

$a \vee \neg k \vee \neg j$

$b \vee d$

$b \vee g \vee \neg n$

$b \vee \neg f \vee n \vee k$

$\neg c \vee k$

$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$

$c \vee l$

$d \vee \neg k \vee l$

$d \vee \neg g \vee l$

$\neg g \vee n \vee o$

$h \vee \neg o \vee \neg j \vee n$

$\neg i \vee j$

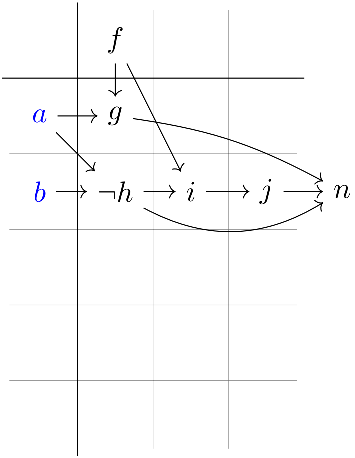
$\neg d \vee \neg l \vee \neg m$

$\neg e \vee m \vee \neg n$

$\neg f \vee h \vee i$

$\boxed{\neg g \vee h \vee \neg j \vee n}$

Conflict analysis



$\neg a \vee \neg f \vee g$
$\neg a \vee \neg b \vee \neg h$
$a \vee c$
$a \vee \neg i \vee \neg l$
$a \vee \neg k \vee \neg j$
$b \vee d$
$b \vee g \vee \neg n$
$b \vee \neg f \vee n \vee k$
$\neg c \vee k$
$\neg c \vee \neg k \vee \neg i \vee l$

$c \vee h \vee n \vee \neg m$
$c \vee l$
$d \vee \neg k \vee l$
$d \vee \neg g \vee l$
$\neg g \vee n \vee o$
$h \vee \neg o \vee \neg j \vee n$
$\neg i \vee j$
$\neg d \vee \neg l \vee \neg m$
$\neg e \vee m \vee \neg n$
$\neg f \vee h \vee i$
$\boxed{\neg g \vee h \vee \neg j \vee n}$

Learning and Backjumping

Learning and Backjumping

- Definition: Explaining a failure: $l_1 \wedge \ldots \wedge l_n \rightarrow \perp$ where $\neg l_1 \vee \ldots \vee \neg l_n$ is the clause triggering failute

Learning and Backjumping

- Definition: Explaining a failure: $l_1 \wedge \ldots \wedge l_n \rightarrow \perp$ where $\neg l_1 \vee \ldots \vee \neg l_n$ is the clause triggering failute
- Definition: Explaining a propagation of l: $l_1 \wedge \ldots \wedge l_n \rightarrow l$ where $\neg l_1 \vee \ldots \vee \neg l_n \vee \neg l$ is the triggering clause

Learning and Backjumping

- Definition: Explaining a failure: $l_1 \wedge \ldots \wedge l_n \rightarrow \perp$ where $\neg l_1 \vee \ldots \vee \neg l_n$ is the clause triggering failute
- Definition: Explaining a propagation of l: $l_1 \wedge \ldots \wedge l_n \rightarrow l$ where $\neg l_1 \vee \ldots \vee \neg l_n \vee \neg l$ is the triggering clause
- At each conflict learn a new clause as following:

Learning and Backjumping

- Definition: Explaining a failure: $l_1 \wedge \ldots \wedge l_n \rightarrow \perp$ where $\neg l_1 \vee \ldots \vee \neg l_n$ is the clause triggering failute
- Definition: Explaining a propagation of l: $l_1 \wedge \ldots \wedge l_n \rightarrow l$ where $\neg l_1 \vee \ldots \vee \neg l_n \vee \neg l$ is the triggering clause
- At each conflict learn a new clause as following:
- Start with the explanation from the clause triggering failure in the form of $l_1 \wedge \ldots \wedge l_n \rightarrow \perp$ and let it be the initial explanation

Learning and Backjumping

- Definition: Explaining a failure: $l_1 \land \ldots \land l_n \rightarrow \perp$ where $\neg l_1 \lor \ldots \lor \neg l_n$ is the clause triggering failute
- Definition: Explaining a propagation of l: $l_1 \land \ldots \land l_n \rightarrow l$ where $\neg l_1 \lor \ldots \lor \neg l_n \lor \neg l$ is the triggering clause
- At each conflict learn a new clause as following:
- Start with the explanation from the clause triggering failure in the form of $l_1 \land \ldots \land l_n \rightarrow \perp$ and let it be the initial explanation
- While there is more than a literal propagated in the last level in the current explanation, replace it with it's explanation from the triggering clause

Learning and Backjumping

- Definition: Explaining a failure: $l_1 \land \ldots \land l_n \rightarrow \perp$ where $\neg l_1 \lor \ldots \lor \neg l_n$ is the clause triggering failute
- Definition: Explaining a propagation of l: $l_1 \land \ldots \land l_n \rightarrow l$ where $\neg l_1 \lor \ldots \lor \neg l_n \lor \neg l$ is the triggering clause
- At each conflict learn a new clause as following:
- Start with the explanation from the clause triggering failure in the form of $l_1 \land \ldots \land l_n \rightarrow \perp$ and let it be the initial explanation
- While there is more than a literal propagated in the last level in the current explanation, replace it with it's explanation from the triggering clause
- When there is only one literal *uip* propagated in the last level in the current explanation, learn the associated new clause C,

Learning and Backjumping

- Definition: Explaining a failure: $l_1 \wedge \ldots \wedge l_n \rightarrow \perp$ where $\neg l_1 \vee \ldots \vee \neg l_n$ is the clause triggering failute
- Definition: Explaining a propagation of l: $l_1 \wedge \ldots \wedge l_n \rightarrow l$ where $\neg l_1 \vee \ldots \vee \neg l_n \vee \neg l$ is the triggering clause
- At each conflict learn a new clause as following:
- Start with the explanation from the clause triggering failure in the form of $l_1 \wedge \ldots \wedge l_n \rightarrow \perp$ and let it be the initial explanation
- While there is more than a literal propagated in the last level in the current explanation, replace it with it's explanation from the triggering clause
- When there is only one literal *uip* propagated in the last level in the current explanation, learn the associated new clause C, backjump (to the last level of propagated literals in C),

Learning and Backjumping

- Definition: Explaining a failure: $l_1 \wedge \ldots \wedge l_n \rightarrow \perp$ where $\neg l_1 \vee \ldots \vee \neg l_n$ is the clause triggering failute
- Definition: Explaining a propagation of l: $l_1 \wedge \ldots \wedge l_n \rightarrow l$ where $\neg l_1 \vee \ldots \vee \neg l_n \vee \neg l$ is the triggering clause
- At each conflict learn a new clause as following:
- Start with the explanation from the clause triggering failure in the form of $l_1 \wedge \ldots \wedge l_n \rightarrow \perp$ and let it be the initial explanation
- While there is more than a literal propagated in the last level in the current explanation, replace it with it's explanation from the triggering clause
- When there is only one literal uip propagated in the last level in the current explanation, learn the associated new clause C, backjump (to the last level of propagated literals in C), propagate $\neg uip$ via the new clause,

Learning and Backjumping

- Definition: Explaining a failure: $l_1 \wedge \ldots \wedge l_n \rightarrow \perp$ where $\neg l_1 \vee \ldots \vee \neg l_n$ is the clause triggering failute
- Definition: Explaining a propagation of l: $l_1 \wedge \ldots \wedge l_n \rightarrow l$ where $\neg l_1 \vee \ldots \vee \neg l_n \vee \neg l$ is the triggering clause
- At each conflict learn a new clause as following:
- Start with the explanation from the clause triggering failure in the form of $l_1 \wedge \ldots \wedge l_n \rightarrow \perp$ and let it be the initial explanation
- While there is more than a literal propagated in the last level in the current explanation, replace it with it's explanation from the triggering clause
- When there is only one literal uip propagated in the last level in the current explanation, learn the associated new clause C, backjump (to the last level of propagated literals in C), propagate $\neg uip$ via the new clause, and continue the exploration

Boosting Search through Randomization and Restarts [Gomes et al., 1998]

Boosting Search through Randomization and Restarts [Gomes et al., 1998]

Heavy-tail phenomena (SAT and CP)

At any time during the experiment there is a non-negligible probability of hitting a problem that requires exponentially more time to solve than any that has been encountered before.

Boosting Search through Randomization and Restarts [Gomes et al., 1998]

Heavy-tail phenomena (SAT and CP)

At any time during the experiment there is a non-negligible probability of hitting a problem that requires exponentially more time to solve than any that has been encountered before.

Hardness = Instance $\oplus$ deterministic algorithm.

Boosting Search through Randomization and Restarts [Gomes et al., 1998]

Heavy-tail phenomena (SAT and CP)

At any time during the experiment there is a non-negligible probability of hitting a problem that requires exponentially more time to solve than any that has been encountered before.

Hardness = Instance $\oplus$ deterministic algorithm.

- Randomization: breaking ties, random decision between k best choices, ...
- Restarts: Geometric/Luby

Other techniques

Other techniques

- Forgetting clauses: The number of the learnt clauses can be exponential, we sometimes need to free some space by forgetting some clauses.

Other techniques

- Forgetting clauses: The number of the learnt clauses can be exponential, we sometimes need to free some space by forgetting some clauses.
- VSIDS (Variable State Independent Decaying Sum): VSIDS is a popular variable ordering heuristic that is based on the notion of activity.

Other techniques

- Forgetting clauses: The number of the learnt clauses can be exponential, we sometimes need to free some space by forgetting some clauses.
- VSIDS (Variable State Independent Decaying Sum): VSIDS is a popular variable ordering heuristic that is based on the notion of activity. The activity of a variable is measured by the number of times it participates in the conflict analysis.

Other techniques

- Forgetting clauses: The number of the learnt clauses can be exponential, we sometimes need to free some space by forgetting some clauses.
- VSIDS (Variable State Independent Decaying Sum): VSIDS is a popular variable ordering heuristic that is based on the notion of activity. The activity of a variable is measured by the number of times it participates in the conflict analysis. Each time a variable x is used during conflict analysis, its activity is incremented.

Other techniques

- Forgetting clauses: The number of the learnt clauses can be exponential, we sometimes need to free some space by forgetting some clauses.
- VSIDS (Variable State Independent Decaying Sum): VSIDS is a popular variable ordering heuristic that is based on the notion of activity. The activity of a variable is measured by the number of times it participates in the conflict analysis. Each time a variable x is used during conflict analysis, its activity is incremented. From time to time, the counters are divided by a constant (to diminish the effect of early conflicts).

SAT Solvers (Few examples)

- MiniSat: http://minisat.se/
- Glucose: `http://www.labri.fr/perso/lsimon/glucose/`
- LingeLing `http://fmv.jku.at/lingeling`
- Any Solver by Armin Biere `http://fmv.jku.at/software/index.html`
- Any winner from past and future SAT competitions: https://www.satcompetition.org/

References I

Davis, M., Logemann, G., and Loveland, D. (1962).
A Machine Program for Theorem-proving.
Communications of the ACM, 5(7):394–397.

Gomes, C. P., Selman, B., and Kautz, H. (1998).
Boosting Combinatorial Search Through Randomization.
In *Proceedings of the 15th National Conference on Artificial Intelligence, AAAI'98, and the 10th Conference on Innovative Applications of Artificial Intelligence, IAAI'98, Madison, Wisconsin*, pages 431–437.

Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001).
Chaff: Engineering an Efficient SAT Solver.
In *Proceedings of the 38th Annual Design Automation Conference, DAC'01, Las Vegas, Nevada, USA*, pages 530–535.

Robinson, J. A. (1965).
A Machine-Oriented Logic Based on the Resolution Principle.
Journal of the ACM, 12(1):23–41.

Silva, J. a. P. M. and Sakallah, K. A. (1999).
Grasp: a search algorithm for propositional satisfiability.
Computers, IEEE Transactions on, 48(5):506–521.