Constraint Programming
– Searching : Part 1 –

Christophe Lecoutre
lecoutre@cril.fr

CRIL-CNRS UMR 8188
Universite d'Artois
Lens, France

January 2021

Outline

1. Backtracking Search
2. Search Ordering Heuristics
3. Guiding Search toward Conflicts

Search Space

For a given CN P such that:

- n is the number of variables
- d is the greatest domain size
- e is the number of constraints
- r is the greatest constraint arity

What is the complexity of a Generate and Test approach?

Answer: $O(d^n er)$, assuming that a constraint check is $O(r)$

Search Space

For a given CN P such that:

- n is the number of variables
- d is the greatest domain size
- e is the number of constraints
- r is the greatest constraint arity

What is the complexity of a Generate and Test approach?

Answer: $O(d^n e r)$, assuming that a constraint check is $O(r)$

Search Space

For a given CN P such that:

- n is the number of variables
- d is the greatest domain size
- e is the number of constraints
- r is the greatest constraint arity

What is the complexity of a Generate and Test approach?

Answer: $O(d^n er)$, assuming that a constraint check is $O(r)$

Exponential Growth

Suppose that:

- the complexity is only $O(2^n)$
- 10^9 complete instantiations can be processed any new second

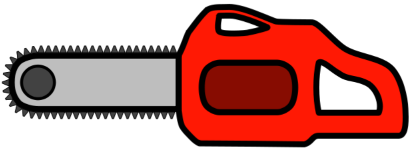
n	2^n	Processing Time
10	around 10^3	around 1 nanosecond
20	around 10^6	around 1 millisecond
30	around 10^9	around 1 second
40	around 10^{12}	around 16 minutes
50	around 10^{15}	around 11 days
60	around 10^{18}	around 32 years
70	around 10^{21}	around 317 centuries

Search Tree

Most of the time, the search space can be perceived as a search tree.

Pruning the Search Tree

Constraint Inference (Filtering/Propagation) can help us!



Pruning the Search Tree

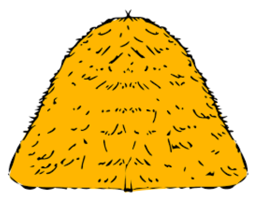
Finding a solution may become realistic in a reduced search tree.

Constraint Inference Only?

Solving a CN by only employing constraint propagation is very rare. To find a solution, one has then to explore the search tree with:

- either a complete method: full exploration of the search space
- or an incomplete method: partial exploration of the search space

In any case, it may look like searching a needle in a haystack!



Complete Exploration

Classical approach

- depth-first traversal
- backtracking mecanism
- interleaving of
  - decisions
  - propagations

Remark.
Other strategies exist:

- breadth-first traversal
- limited discrepancy search (LDS)
- large neighborhood search (LNS)
- ...

Complete Exploration

Classical approach

- depth-first traversal
- backtracking mecanism
- interleaving of
  - decisions
  - propagations

Remark.

Other strategies exist:

- breadth-first traversal
- limited discrepancy search (LDS)
- large neighborhood search (LNS)
- ...

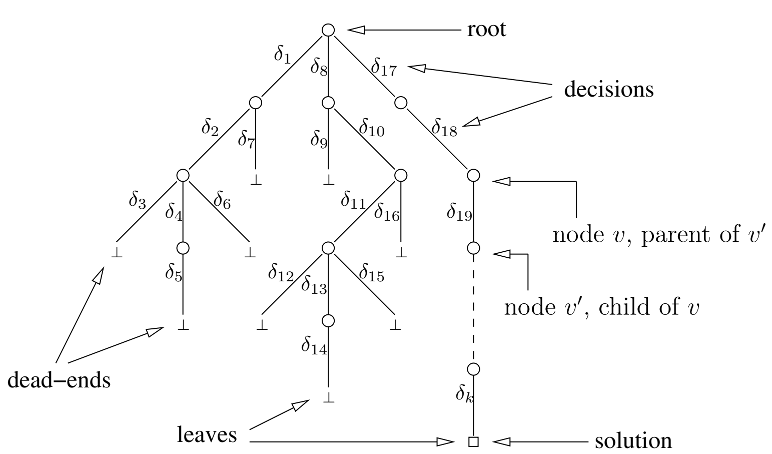
Outline

1. Backtracking Search

2. Search Ordering Heuristics

3. Guiding Search toward Conflicts

Search Tree



Nonbinary ϕ-search

Algorithm 1: nonbinary-ϕ-search(P: CN): Boolean

```
P ← φ(P)
if P = ⊥ then
    return false
if ∀x ∈ vars(P), |dom(x)| = 1 then
    return true
select a variable x of P such that |dom(x)| > 1
foreach value a ∈ dom(x) do
    if nonbinary-φ-search(P|x=a) then
        return true
return false
```

Remark.

ϕ denotes the process (level) of filtering under the form of a consistency to enforce (AC, BC, ...).

Nonbinary ϕ-search

Algorithm 2: nonbinary-ϕ-search(P: CN): Boolean

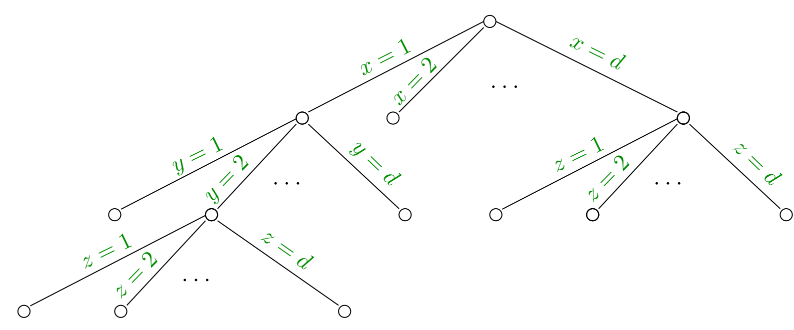
$P \leftarrow \phi(P)$
if $P = \perp$ **then**
    return *false*
if $\forall x \in vars(P), |dom(x)| = 1$ **then**
    return *true*
select a variable x of P such that $|dom(x)| > 1$
foreach *value* $a \in dom(x)$ **do**
    if *nonbinary-ϕ-search*($P|_{x=a}$) **then**
        return *true*
return *false*

Remark.

ϕ denotes the process (level) of filtering under the form of a consistency to enforce (AC, BC, ...).

Illustration

For simplicity, we assume that the domains of all variables is $\{1, 2, \ldots, d\}$.

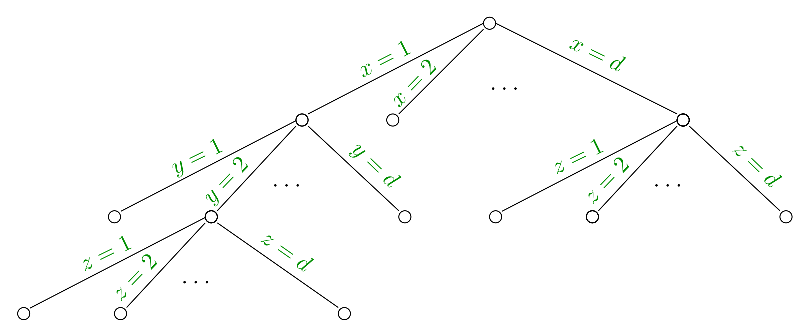


Remark.

Note that all taken decisions are positive, i.e., variable assignments.

Illustration

For simplicity, we assume that the domains of all variables is $\{1, 2, \ldots, d\}$.



Remark.

Note that all taken decisions are positive, i.e., variable assignments.

Binary ϕ-search

Algorithm 3: binary-ϕ-search(P: CN): Boolean

$P \leftarrow \phi(P)$
if $P = \bot$ **then**
  return *false*
if $\forall x \in vars(P), |dom(x)| = 1$ **then**
  return *true*
select a value (x, a) of P such that $|dom(x)| > 1$
return $binary\text{-}\phi\text{-}search(P|_{x=a}) \vee binary\text{-}\phi\text{-}search(P|_{x \neq a})$

Question: Is there a condition on ϕ ?

Answer: ϕ must at least check covered constraints (also true for nonbinary ϕ-search)

Binary ϕ-search

Algorithm 4: binary-ϕ-search(P: CN): Boolean

$P \leftarrow \phi(P)$
if $P = \bot$ **then**
  return *false*
if $\forall x \in vars(P), |dom(x)| = 1$ **then**
  return *true*
select a value (x, a) of P such that $|dom(x)| > 1$
return $binary\text{-}\phi\text{-}search(P|_{x=a}) \vee binary\text{-}\phi\text{-}search(P|_{x \neq a})$

Question: Is there a condition on ϕ ?

Answer: ϕ must at least check covered constraints (also true for nonbinary ϕ-search)

Binary ϕ-search

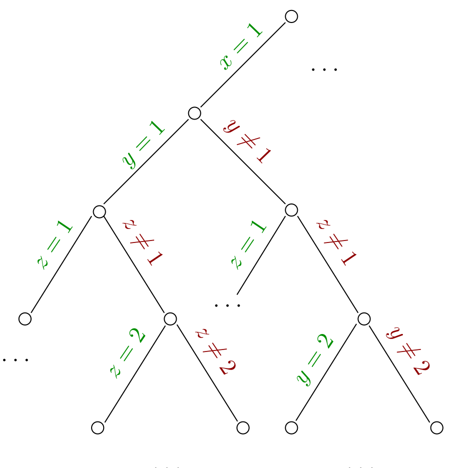
Algorithm 5: binary-ϕ-search(P: CN): Boolean

$P \leftarrow \phi(P)$
if $P = \bot$ **then**
  return *false*
if $\forall x \in vars(P), |dom(x)| = 1$ **then**
  return *true*
select a value (x, a) of P such that $|dom(x)| > 1$
return $binary\text{-}\phi\text{-}search(P|_{x=a}) \vee binary\text{-}\phi\text{-}search(P|_{x\neq a})$

Question: Is there a condition on ϕ ?

Answer: ϕ must at least check covered constraints (also true for nonbinary ϕ-search)

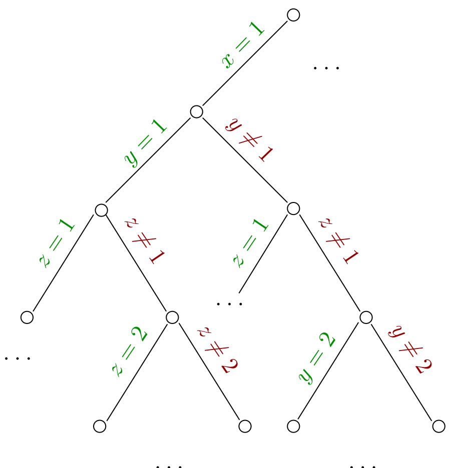
Illustration



Remark.

Note that first decisions are positive, i.e., variable assignments, and second decisions are negative, i.e., value refutations.

Illustration



Remark.

Note that first decisions are positive, i.e., variable assignments, and second decisions are negative, i.e., value refutations.

About Node Expansion

At each node η of the search tree, there is a subdivision of the current CN P^η into a set of reduced CNs whose union is equivalent to P^η. This guarantees completeness provided that all CNs are explored.

Branching is the fact of subdividing nodes of the search tree.

Remark.
Other forms of branching, different from:

- non-binary branching
- binary branching

introduced earlier can be considered, as for example,

- (binary) domain splitting

where two branches labelled with $x < k$ and $x \geq k$ are generated.

About Node Expansion

At each node η of the search tree, there is a subdivision of the current CN P^η into a set of reduced CNs whose union is equivalent to P^η. This guarantees completeness provided that all CNs are explored.

Branching is the fact of subdividing nodes of the search tree.

Remark.

Other forms of branching, different from:

- non-binary branching
- binary branching

introduced earlier can be considered, as for example,

- (binary) domain splitting

where two branches labelled with $x < k$ and $x \geq k$ are generated.

About Node Expansion

At each node η of the search tree, there is a subdivision of the current CN P^η into a set of reduced CNs whose union is equivalent to P^η. This guarantees completeness provided that all CNs are explored.

Branching is the fact of subdividing nodes of the search tree.

Remark.

Other forms of branching, different from:

- non-binary branching
- binary branching

introduced earlier can be considered, as for example,

- (binary) domain splitting

where two branches labelled with $x < k$ and $x \geq k$ are generated.

Look-ahead and Look-back Schemes

Depending on the chosen level of filtering corresponding to ϕ, we obtain different look-ahead algorithms. Classical algorithms are:

- BT
- FC (Forward Checking)
- MAC (Maintaining Arc Consistency)

Remark.
There exist look-back schemes that allo us to perform intelligent backtracking:

- CBJ (Conflict-directed backjumping)
- DBT (Dynamic Backtracking)

Look-ahead and Look-back Schemes

Depending on the chosen level of filtering corresponding to ϕ, we obtain different look-ahead algorithms. Classical algorithms are:

- BT
- FC (Forward Checking)
- MAC (Maintaining Arc Consistency)

Remark.

There exist look-back schemes that allo us to perform intelligent backtracking:

- CBJ (Conflict-directed backjumping)
- DBT (Dynamic Backtracking)

BT

BT is a nonbinary ϕ-search where at each node ϕ simply checks that all constraints covered by the current instantiation are satisfied.

Definition

Let I be an instantiation and c be a constraint. c is covered by I iff $scp(c) \subseteq vars(I)$.

Example.

Let $I = \{x = 1, y = 3, z = 2\}$ be an instantiation. we have:

- the constraint $x + z = y$ is covered and satisfied by I
- the constraint $w = x$ is not covered by I
- the constraint $x > y$ is covered but not satisfied by I

BT

BT is a nonbinary ϕ-search where at each node ϕ simply checks that all constraints covered by the current instantiation are satisfied.

Definition

Let I be an instantiation and c be a constraint. c is covered by I iff $scp(c) \subseteq vars(I)$.

Example.

Let $I = \{x = 1, y = 3, z = 2\}$ be an instantiation. we have:

- the constraint $x + z = y$ is covered and satisfied by I
- the constraint $w = x$ is not covered by I
- the constraint $x > y$ is covered but not satisfied by I

Illustration of BT

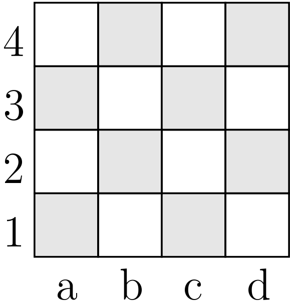


Illustration of BT

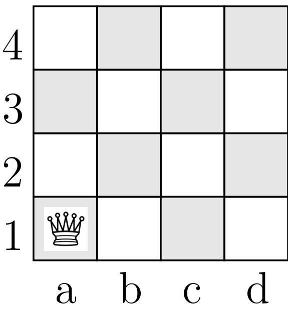


Illustration of BT

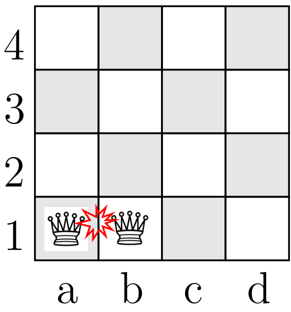


Illustration of BT

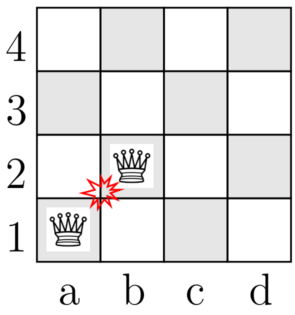


Illustration of BT

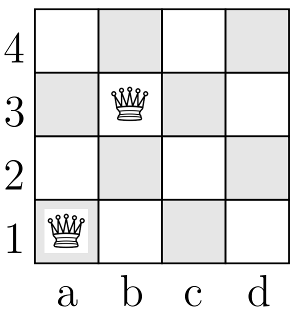


Illustration of BT

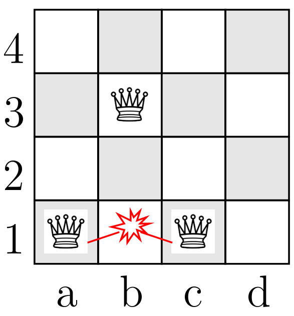


Illustration of BT

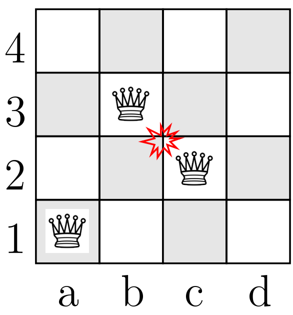


Illustration of BT

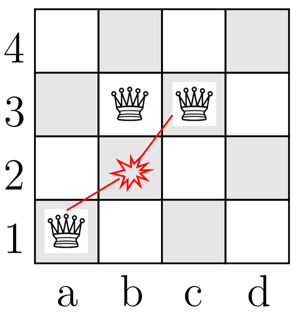


Illustration of BT

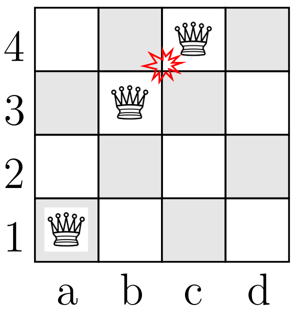


Illustration of BT

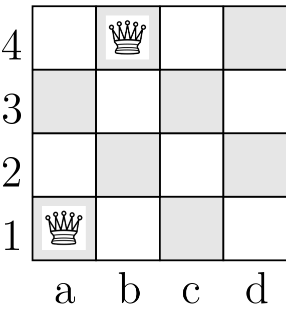


Illustration of BT

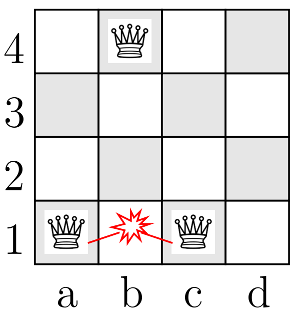


Illustration of BT

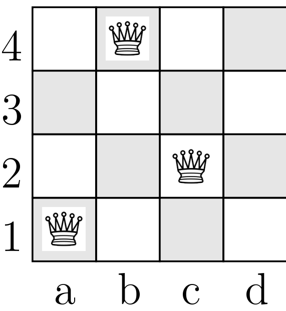


Illustration of BT

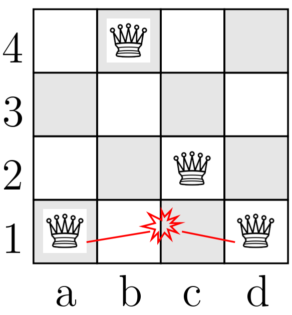


Illustration of BT

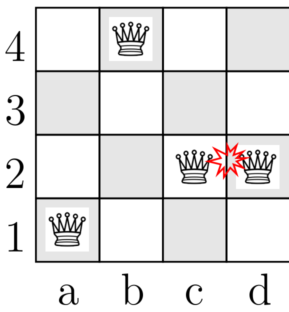


Illustration of BT

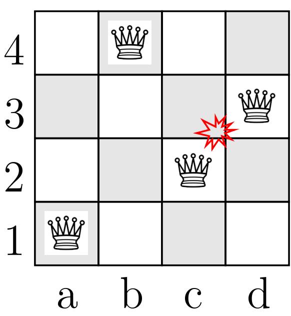


Illustration of BT

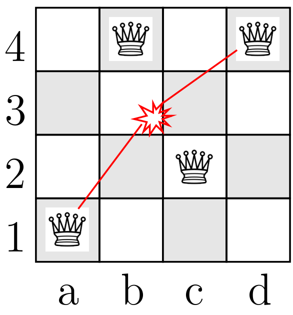


Illustration of BT

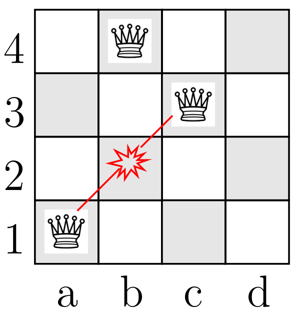


Illustration of BT

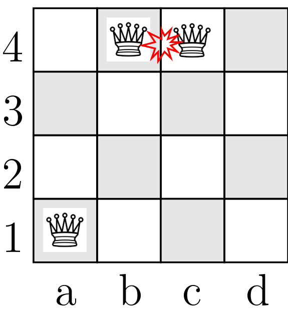


Illustration of BT

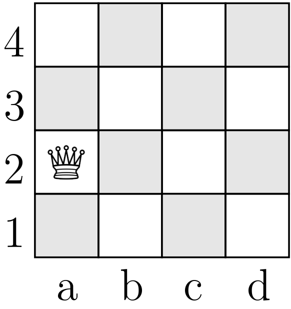


Illustration of BT

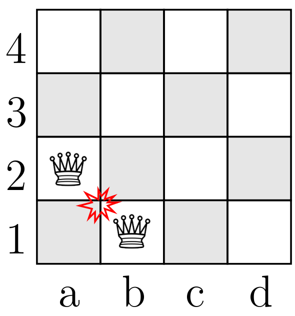
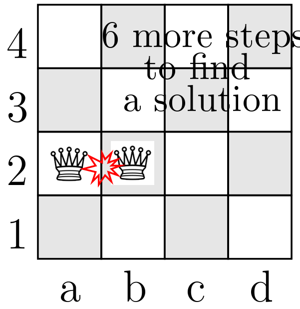


Illustration of BT



FC

FC (Forward Checking) is a nonbinary ϕ-search where at each node ϕ enforces arc consistency only on constraints that are almost covered by the current instantiation.

Definition

Let I be an instantiation and c be a constraint. c is almost covered by I iff $|scp(c) \setminus vars(I)| = 1$.

Example.

Let $I = \{x = 1, y = 3, z = 2\}$ be an instantiation. we have:

- the constraint $x + z = y + w$ is almost covered by I
- the constraint $w = t$ is neither covered nor almost covered by I

Remark.

FC enforces a partial form of Arc Consistency.

FC

FC (Forward Checking) is a nonbinary ϕ-search where at each node ϕ enforces arc consistency only on constraints that are almost covered by the current instantiation.

Definition

Let I be an instantiation and c be a constraint. c is almost covered by I iff $|scp(c) \setminus vars(I)| = 1$.

Example.

Let $I = \{x = 1, y = 3, z = 2\}$ be an instantiation. we have:

- the constraint $x + z = y + w$ is almost covered by I
- the constraint $w = t$ is neither covered nor almost covered by I

Remark.

FC enforces a partial form of Arc Consistency.

FC

FC (Forward Checking) is a nonbinary ϕ-search where at each node ϕ enforces arc consistency only on constraints that are almost covered by the current instantiation.

Definition

Let I be an instantiation and c be a constraint. c is almost covered by I iff $|scp(c) \setminus vars(I)| = 1$.

Example.

Let $I = \{x = 1, y = 3, z = 2\}$ be an instantiation. we have:

- the constraint $x + z = y + w$ is almost covered by I
- the constraint $w = t$ is neither covered nor almost covered by I

Remark.

FC enforces a partial form of Arc Consistency.

Illustration of FC

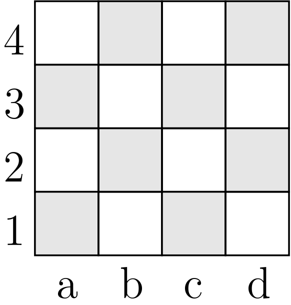


Illustration of FC

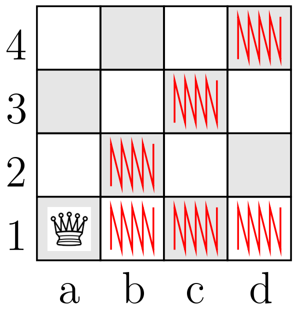


Illustration of FC

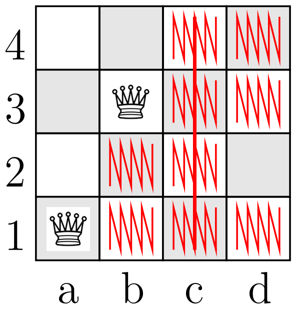


Illustration of FC

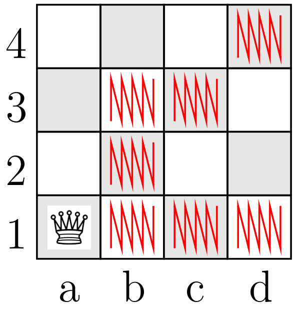


Illustration of FC

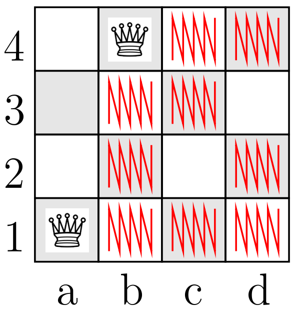


Illustration of FC

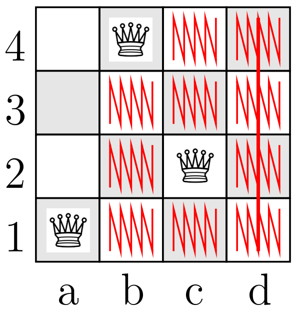


Illustration of FC

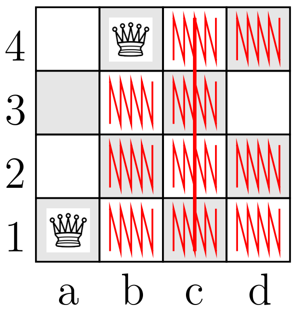


Illustration of FC

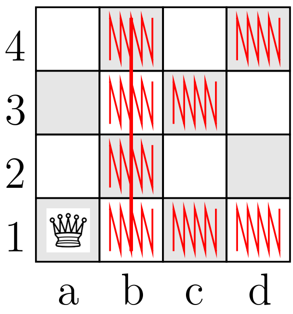


Illustration of FC

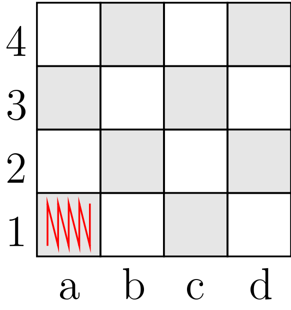


Illustration of FC

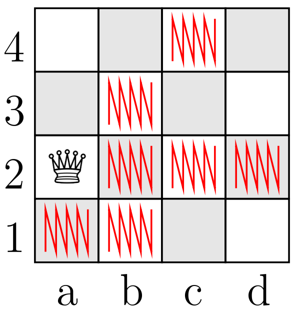


Illustration of FC

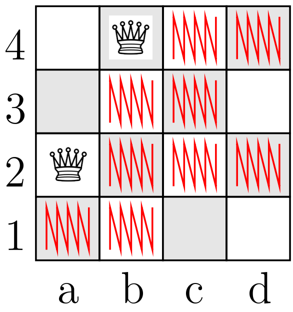


Illustration of FC

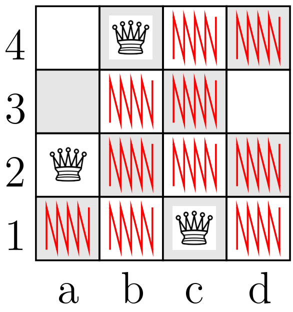
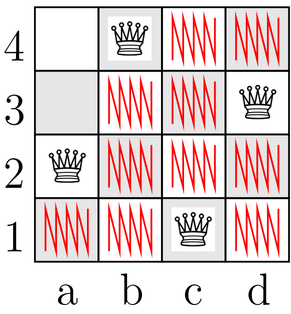


Illustration of FC



MAC

MAC is a binary search enforcing (maintaining) Arc Consistency at each node of the search tree.

In the following algorithm,

- we introduce I that represents a stack with the successive positive decisions that are taken along the current branch
- we write $AC(P, S)$ for enforcing arc consistency, with Q initialized with S. Note that:
  - S is $vars(P)$ initially, to guarantee AC at the root of the search tree
  - S is the variable x involved in the last taken decision during search
- we record information about value removals at each level $|I|$ of the search tree.

Remark.

We consider here that the level in the search tree corresponds to the number of taken positive decisions (consequently ignoring negative decisions in this regard).

MAC=Binary-AC-search

Algorithm 6: MAC(P: CN)

```
consistent ← AC(P, vars(P))                                   // AC initially enforced
if ¬consistent then
    return
I ← ∅                                          // I represents the current instantiation
finished ← false
while ¬finished do
    select a value (x, a) of P such that x ∉ vars(I)
    I.push(x, a)
    dom(x).reduceTo(a, |I|)                     // x is assigned the value a at level |I|
    consistent ← AC(P, {x})                    // AC maintained after positive decisions
    if consistent ∧ |I| = n then
        print(I)                               // A solution has been found and is printed
        consistent ← false                     // Inserted to keep searching for solutions
    while ¬consistent ∧ I ≠ ∅ do
        (x, a) ← I.pop()
        foreach variable y ∈ vars(P) \ vars(I) do
            dom(y).restoreAt(|I|)
        dom(x).remove(a, |I|)                   // a is removed from dom(x) at level |I|
        if dom(x) = ∅ then
            consistent ← false
        else
            consistent ← AC(P, {x})             // AC maintained after negative decisions
    if ¬consistent then
        finished ← true
```

Illustration of MAC

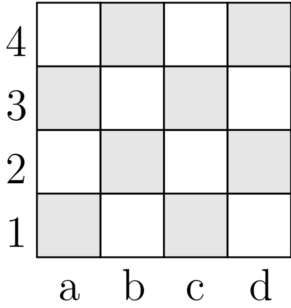


Illustration of MAC

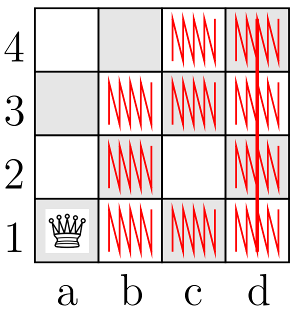


Illustration of MAC

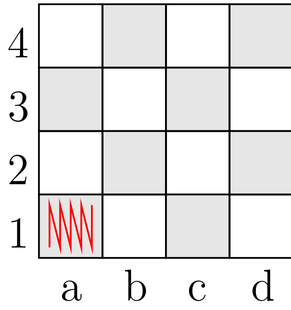


Illustration of MAC

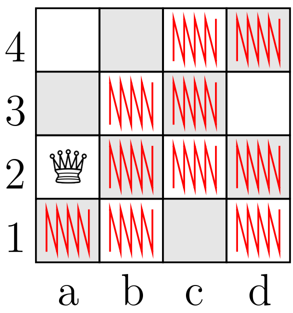


Illustration of MAC

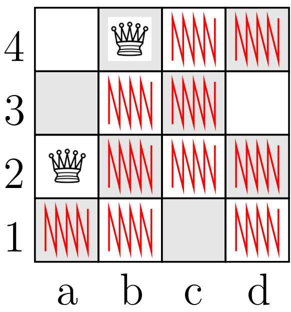


Illustration of MAC

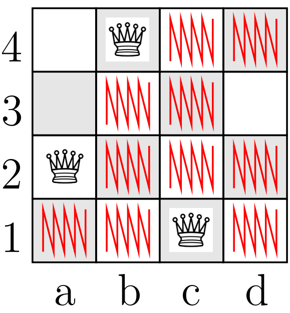
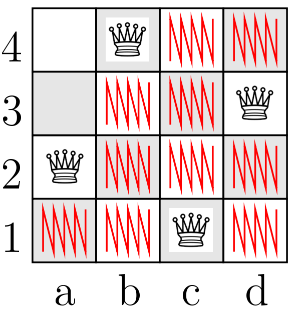


Illustration of MAC



Experiment with Ace

Try:

```
java -jar ACE-21-01.jar Queens-0010.xml -r_c=max -varh=Dom -s=all
```

where:

- `-s=all` means that we look for all solutions
- `-r_c=max` means that the restart cutoff is the maximal value (so there is no restart)
- `-varh=Dom` means that we use the variable ordering heuristic *min-dom*

Comparing BT/FC/MAC :

- BT: add `-p=BT`
- FC: add `-p=FC`
- MAC: default value (this is equivalent to add `-p=GAC`)

Experiment with Ace

Try:

```
java -jar ACE-21-01.jar Queens-0010.xml -r_c=max -varh=Dom -s=all
```

where:

- `-s=all` means that we look for all solutions
- `-r_c=max` means that the restart cutoff is the maximal value (so there is no restart)
- `-varh=Dom` means that we use the variable ordering heuristic *min-dom*

Comparing BT/FC/MAC :

- BT: add `-p=BT`
- FC: add `-p=FC`
- MAC: default value (this is equivalent to add `-p=GAC`)

Backtracking Information

BT, FC and MAC are backtracking algorithms. And when backtracking, we have to:

- restore domains
- possibly restore constraints (for example, the structure *last* used with AC2001 or the dynamic tables used in STR)

Two classical solutions exist:

- Copying
  - All structures that must be backtracked are copied at each level.
  - The copy is performed before taking the next decision.
  - We have to be careful about the memory!
  - On backtrack: use the copy recorded at the right level.
- Trailing:
  - Only modifications to structures are recorded.
  - On backtrack: undo recorded modifications.

Backtracking Information

BT, FC and MAC are backtracking algorithms. And when backtracking, we have to:

- restore domains
- possibly restore constraints (for example, the structure *last* used with AC2001 or the dynamic tables used in STR)

Two classical solutions exist:

- Copying
  - All structures that must be backtracked are copied at each level.
  - The copy is performed before taking the next decision.
  - We have to be careful about the memory!
  - On backtrack: use the copy recorded at the right level.
- Trailing:
  - Only modifications to structures are recorded.
  - On backtrack: undo recorded modifications.

Intelligent Backtracking: CBJ

BT, FC and MAC basically performs chronological backtracking, whereas CBJ performs intelligent backtracking.

For simplicity, we consider binary constraints only, and we consider that ϕ simply checks covered constraints (as in BT).

The method works as follows:

1. Whenever a new assignment $y = b$ is performed, and happens to be incompatible with a previously assigned value (x, a), we record a nogood $\neg(x = a \wedge y = b)$, which can also be written as

   $$x = a \rightarrow y \neq b$$

   with $x = a$ being the explanation of $y \neq b$. We note:

   $$expl(y \neq b) = \{x = a\}.$$

2. Whenever a domain wipeout occurs (for a variable y), we can deduce a new nogood:

   $$\wedge_{b \in dom^{init}(y)} expl(y \neq b)$$

3. Checking inferred nogoods permit to backtrack more than chronological backtracking.

Intelligent Backtracking: CBJ

BT, FC and MAC basically performs chronological backtracking, whereas CBJ performs intelligent backtracking.

For simplicity, we consider binary constraints only, and we consider that ϕ simply checks covered constraints (as in BT).

The method works as follows:

1. Whenever a new assignment $y = b$ is performed, and happens to be incompatible with a previously assigned value (x, a), we record a nogood $\neg(x = a \land y = b)$, which can also be written as

   $$x = a \rightarrow y \neq b$$

   with $x = a$ being the explanation of $y \neq b$. We note:

   $$expl(y \neq b) = \{x = a\}.$$

2. Whenever a domain wipeout occurs (for a variable y), we can deduce a new nogood:

   $$\land_{b \in dom^{init}(y)} expl(y \neq b)$$

3. Checking inferred nogoods permit to backtrack more than chronological backtracking.

Intelligent Backtracking: CBJ

BT, FC and MAC basically performs chronological backtracking, whereas CBJ performs intelligent backtracking.

For simplicity, we consider binary constraints only, and we consider that ϕ simply checks covered constraints (as in BT).

The method works as follows:

1. Whenever a new assignment $y = b$ is performed, and happens to be incompatible with a previously assigned value (x, a), we record a nogood $\neg(x = a \wedge y = b)$, which can also be written as

   $$x = a \rightarrow y \neq b$$

   with $x = a$ being the explanation of $y \neq b$. We note:

   $$expl(y \neq b) = \{x = a\}.$$

2. Whenever a domain wipeout occurs (for a variable y), we can deduce a new nogood:

   $$\wedge_{b \in dom^{init}(y)} expl(y \neq b)$$

3. Checking inferred nogoods permit to backtrack more than chronological backtracking.

Illustration of CBJ

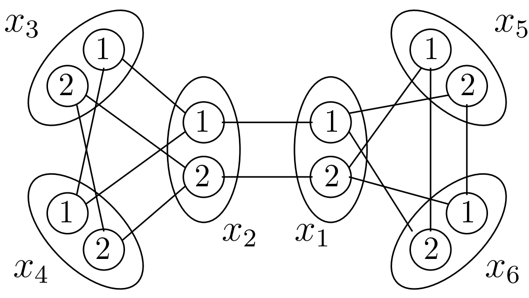


Illustration of CBJ

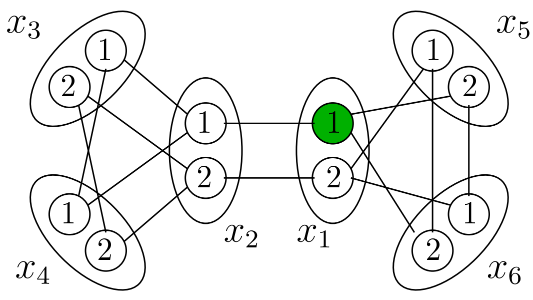


Illustration of CBJ

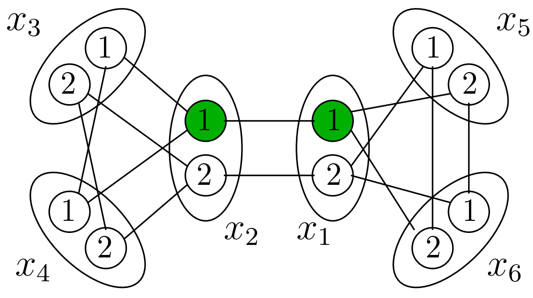


Illustration of CBJ

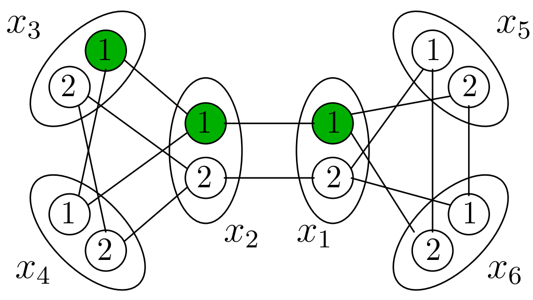


Illustration of CBJ

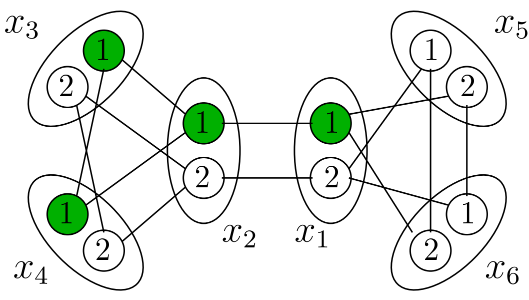


Illustration of CBJ

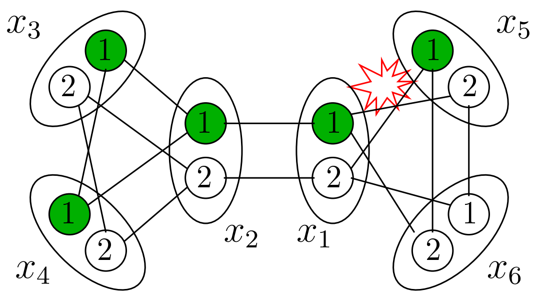


Illustration of CBJ

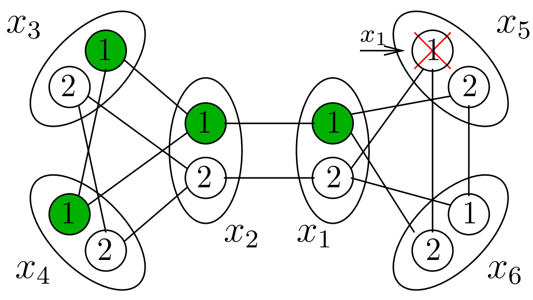


Illustration of CBJ

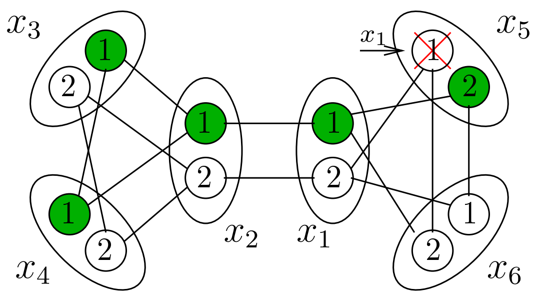


Illustration of CBJ

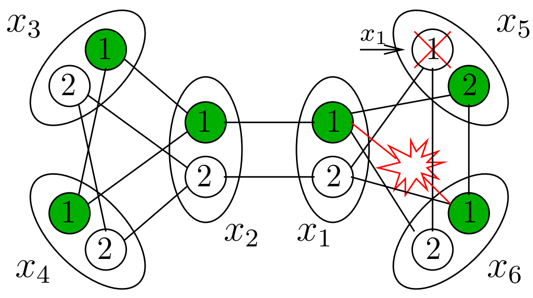


Illustration of CBJ

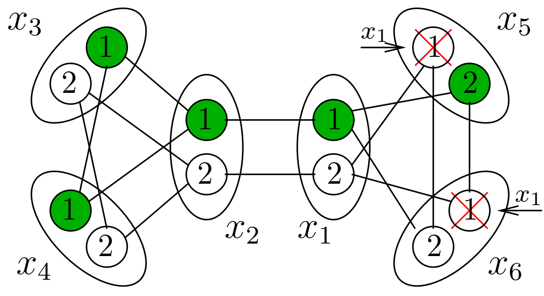


Illustration of CBJ

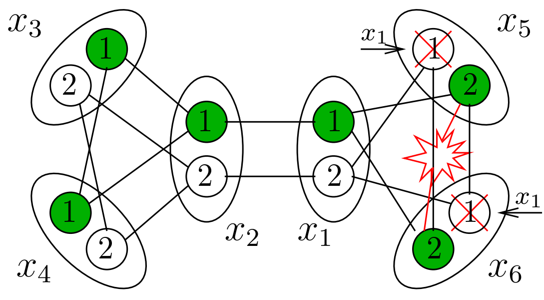


Illustration of CBJ

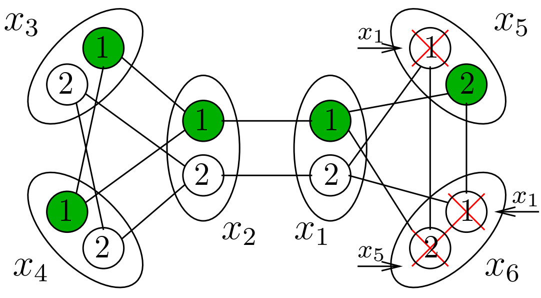


Illustration of CBJ

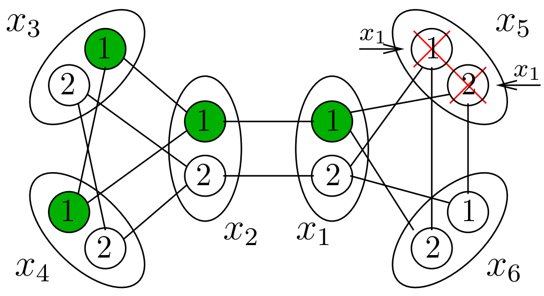
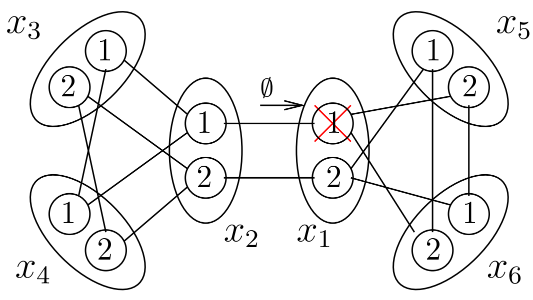


Illustration of CBJ



Outline

1. Backtracking Search
2. Search Ordering Heuristics
3. Guiding Search toward Conflicts

Search-guiding Heuristics

Important:

- The order in which variables are assigned by a backtracking search algorithm has been recognized as a key issue for a long time.
- Using different search ordering heuristics to solve a CSP instance can lead to drastically different results in terms of efficiency.
- Simply introducing some form of randomization to a given search ordering heuristic may exhibit a large variability in performance.

Goal of such heuristics: to minimize the size of the search trees

Typically, when conducting a backtrack search, we sollicit:

- a variable ordering heuristic to select the next variable x to be assigned
- a value ordering heuristic to select the value a to assign to x

Search-guiding Heuristics

Important:

- The order in which variables are assigned by a backtracking search algorithm has been recognized as a key issue for a long time.
- Using different search ordering heuristics to solve a CSP instance can lead to drastically different results in terms of efficiency.
- Simply introducing some form of randomization to a given search ordering heuristic may exhibit a large variability in performance.

Goal of such heuristics: to minimize the size of the search trees

Typically, when conducting a backtrack search, we sollicit:

- a variable ordering heuristic to select the next variable x to be assigned
- a value ordering heuristic to select the value a to assign to x

Search-guiding Heuristics

Important:

- The order in which variables are assigned by a backtracking search algorithm has been recognized as a key issue for a long time.
- Using different search ordering heuristics to solve a CSP instance can lead to drastically different results in terms of efficiency.
- Simply introducing some form of randomization to a given search ordering heuristic may exhibit a large variability in performance.

Goal of such heuristics: to minimize the size of the search trees

Typically, when conducting a backtrack search, we sollicit:

- a variable ordering heuristic to select the next variable x to be assigned
- a value ordering heuristic to select the value a to assign to x

Search-guiding Heuristics

General rules to adopt for efficieny:

1. It is better to assign first the variables that belong to the hard parts of the problem. Fail-first principle:

   "To succed, try first where you are most likely to fail"

2. To find quickly a solution, it is better to assign first the value that most likely belongs to a solution (Succeed-first or Promise principle).
3. The initial variable/value choices are particularly important.

Remark.

Depending on the context, the second rule may not be so important.

Search-guiding Heuristics

General rules to adopt for efficieny:

1. It is better to assign first the variables that belong to the hard parts of the problem. Fail-first principle:

   "To succed, try first where you are most likely to fail"

2. To find quickly a solution, it is better to assign first the value that most likely belongs to a solution (Succeed-first or Promise principle).
3. The initial variable/value choices are particularly important.

Remark.
Depending on the context, the second rule may not be so important.

Variable Ordering Heuristics

Basically, a variable ordering heuristic associates a score (real value) with every variable. Then, we can choose between:

- *min*: selecting the variable with the lowest score
- *max*: selecting the variable with the highest score

Example.

- *min-dom*, simply denoted by *dom* most of the time, is the heuristic that selects the variable with the smallest current domain.
- *max-deg*, simply denoted by *deg* most of the time, is the heuristic that selects the variable with the highest degree.

In case of several variables with best equal scores, we need a tie-breaker. For example, *brelaz* is *dom+deg*.

Variable Ordering Heuristics

Basically, a variable ordering heuristic associates a score (real value) with every variable. Then, we can choose between:

- *min*: selecting the variable with the lowest score
- *max*: selecting the variable with the highest score

Example.

- *min-dom*, simply denoted by *dom* most of the time, is the heuristic that selects the variable with the smallest current domain.
- *max-deg*, simply denoted by *deg* most of the time, is the heuristic that selects the variable with the highest degree.

In case of several variables with best equal scores, we need a tie-breaker. For example, *brelaz* is *dom+deg*.

Variable Ordering Heuristics

Basically, a variable ordering heuristic associates a score (real value) with every variable. Then, we can choose between:

- *min*: selecting the variable with the lowest score
- *max*: selecting the variable with the highest score

Example.

- *min-dom*, simply denoted by *dom* most of the time, is the heuristic that selects the variable with the smallest current domain.
- *max-deg*, simply denoted by *deg* most of the time, is the heuristic that selects the variable with the highest degree.

In case of several variables with best equal scores, we need a tie-breaker. For example, *brelaz* is *dom+deg*.

Categories of Variable Ordering Heuristics

Static variable ordering heuristics precomputes ordering before search.

- *lexico*
- *deg* and *ddeg* (Ullmann, 1976; Dechter & Meiri, 1989)

Dynamic variable ordering heuristics performs a computation at each node using the current state.

- *dom* (Haralick & Elliott, 1980)
- *dom/ddeg* (Bessiere & Régin, 1996)
- *brelaz* (Brelaz, 1979; Smith, 1999)

Adaptive variable ordering heuristics performs a computation at each node using the current state and the history (of explored nodes).

- *wdeg*, *dom/wdeg* (Boussemart *et al.*, 2004)
- *impact* (Refalo, 2004)
- *activity* (Michel & Hentenryck, 2012)

Categories of Variable Ordering Heuristics

Static variable ordering heuristics precomputes ordering before search.

- *lexico*
- *deg* and *ddeg* (Ullmann, 1976; Dechter & Meiri, 1989)

Dynamic variable ordering heuristics performs a computation at each node using the current state.

- *dom* (Haralick & Elliott, 1980)
- *dom/ddeg* (Bessiere & Régin, 1996)
- *brelaz* (Brelaz, 1979; Smith, 1999)

Adaptive variable ordering heuristics performs a computation at each node using the current state and the history (of explored nodes).

- *wdeg*, *dom/wdeg* (Boussemart *et al.*, 2004)
- *impact* (Refalo, 2004)
- *activity* (Michel & Hentenryck, 2012)

Categories of Variable Ordering Heuristics

Static variable ordering heuristics precomputes ordering before search.

- *lexico*
- *deg* and *ddeg* (Ullmann, 1976; Dechter & Meiri, 1989)

Dynamic variable ordering heuristics performs a computation at each node using the current state.

- *dom* (Haralick & Elliott, 1980)
- *dom/ddeg* (Bessiere & Régin, 1996)
- *brelaz* (Brelaz, 1979; Smith, 1999)

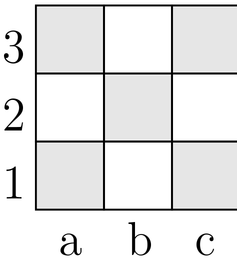
Adaptive variable ordering heuristics performs a computation at each node using the current state and the history (of explored nodes).

- *wdeg*, *dom/wdeg* (Boussemart *et al.*, 2004)
- *impact* (Refalo, 2004)
- *activity* (Michel & Hentenryck, 2012)

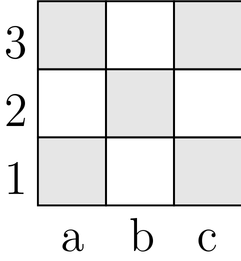
Illustration

Compare the (binary) search trees built by FC on the instance `3-queens` while using:

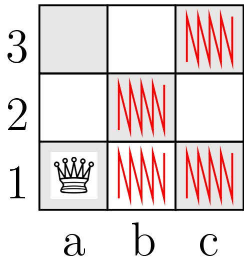
- the heuristic *min-dom*, with *lexico* as tie-breaker
- the anti-heuristic *max-dom*, with *lexico* as tie-breaker



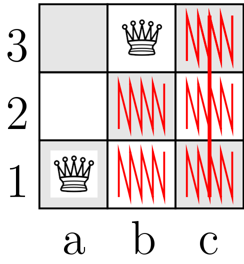
FC-*min-dom* on 3-queens



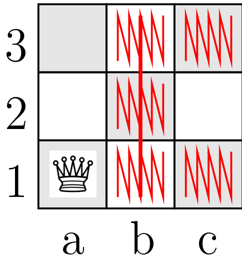
FC-*min-dom* on 3-queens



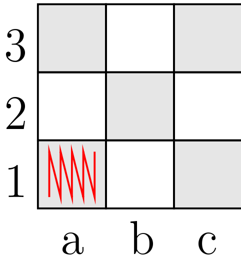
FC-*min-dom* on 3-queens



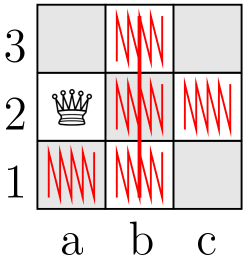
FC-*min-dom* on 3-queens



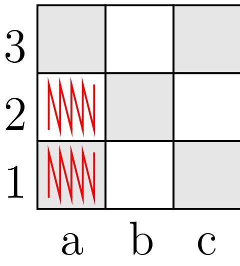
FC-*min-dom* on 3–queens



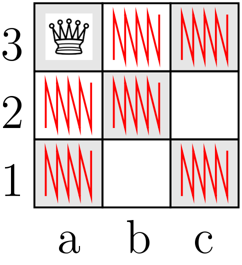
FC-*min-dom* on 3-queens



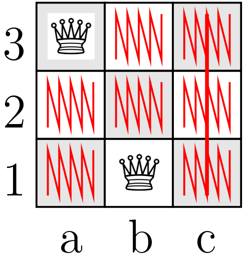
FC-*min-dom* on 3–queens



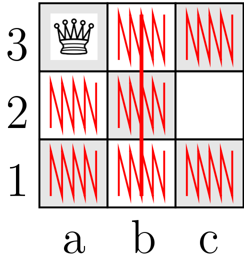
FC-*min-dom* on 3-queens



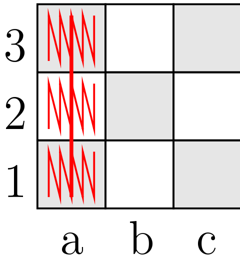
FC-*min-dom* on 3–queens



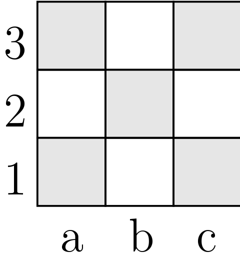
FC-*min-dom* on 3–queens



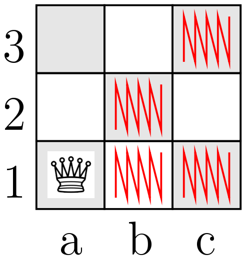
FC-*min-dom* on 3-queens



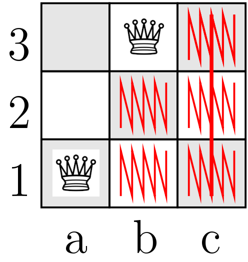
FC-*max-dom* on 3–queens



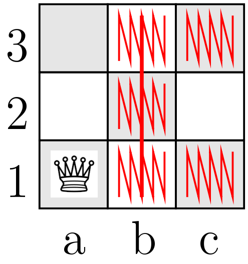
FC-*max-dom* on 3–queens



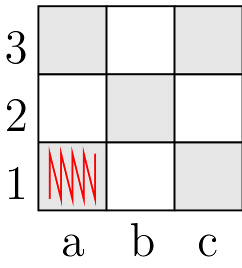
FC-*max-dom* on 3–queens



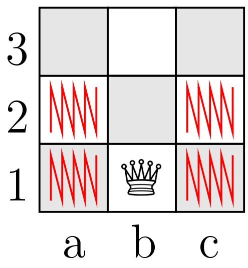
FC-*max-dom* on 3–queens



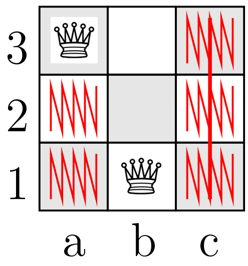
FC-*max-dom* on 3–queens



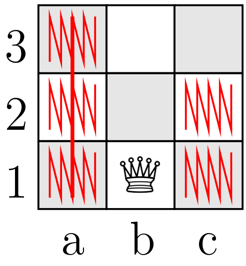
FC-*max-dom* on 3−queens



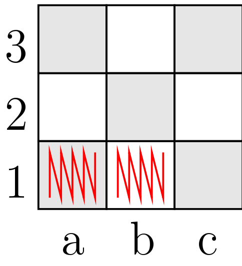
FC-*max-dom* on 3–queens



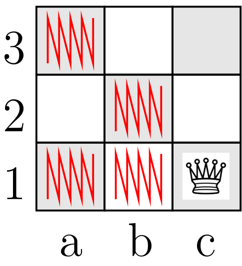
FC-*max-dom* on 3-queens



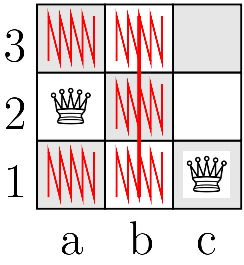
FC-*max-dom* on 3–queens



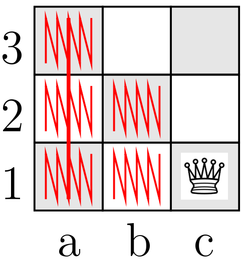
FC-*max-dom* on 3–queens



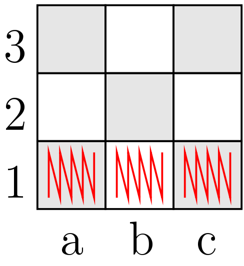
FC-*max-dom* on 3–queens



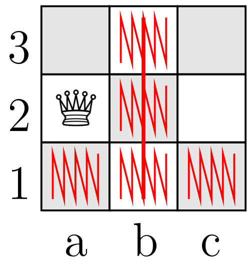
FC-*max-dom* on 3–queens



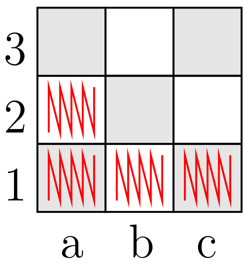
FC-*max-dom* on 3–queens



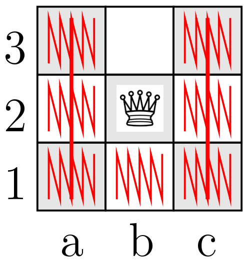
FC-*max-dom* on 3–queens



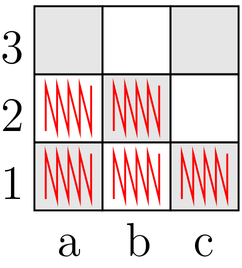
FC-*max-dom* on 3–queens



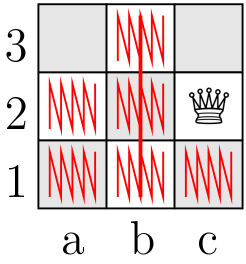
FC-*max-dom* on 3–queens



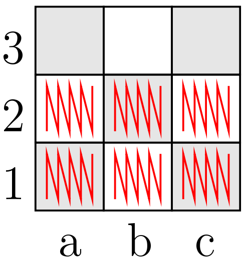
FC-*max-dom* on 3–queens



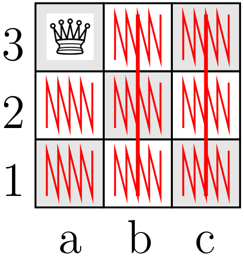
FC-*max-dom* on 3–queens



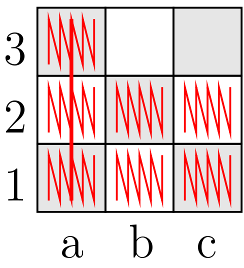
FC-*max-dom* on 3–queens



FC-*max-dom* on 3–queens



FC-*max-dom* on 3–queens



Value Ordering Heuristics

Classical heuristics:

- *lexico*
- *random*
- *min-conflicts*
- *max-conflicts*

For a value (x, a), the conflict count of (x, a) on a CN P is an integer, denoted by $cc(x, a)$, computed as follows:

$$\sum_{c \in ctrs(P): x \in scp(c)} |\{\tau \in \Pi_{y \in scp(c)} dom(y) \setminus rel(c) \mid \tau[x] = a\}|$$

Value Ordering Heuristics

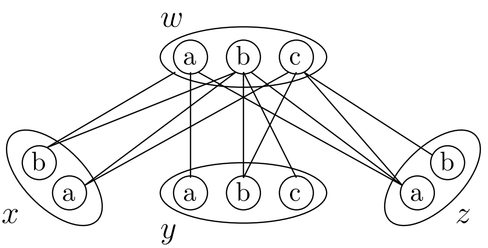
Classical heuristics:

- *lexico*
- *random*
- *min-conflicts*
- *max-conflicts*

For a value (x, a), the conflict count of (x, a) on a CN P is an integer, denoted by $cc(x, a)$, computed as follows:

$$\sum_{c \in ctrs(P) : x \in scp(c)} |\{\tau \in \Pi_{y \in scp(c)} dom(y) \setminus rel(c) \mid \tau[x] = a\}|$$

Illustration



The variable *w* involved in three binary constraints. We have:

- $cc(w, a) = 1 + 2 + 1 = 4$
- $cc(w, b) = 0 + 2 + 1 = 3$
- $cc(w, c) = 1 + 1 + 0 = 2$

The order given by *min-conflicts* for *w* is then *c*, *b* and *a*.

Outline

1. Backtracking Search
2. Search Ordering Heuristics
3. Guiding Search toward Conflicts

Adaptive Heuristics

A search-guiding heuristic is said to be adaptive when the selection it performs at a given node of the search tree depends on the current state of the problem instance as well as on the past states encountered so far.

In other words, some information concerning the sub-tree already explored is taken into account by an adaptive heuristic to perform its selection. It implements then a kind of learning, as for example:

- *wdeg*: constraint weighting
- *impacts* and *activity*: memorization of search space reductions
- *lc* (last conflicts): memorization of the last failed assignments // used in conjunction with a variable heuristic

Our focus is on wdeg and lc, which aim at guiding search towards conflicts.

Adaptive Heuristics

A search-guiding heuristic is said to be adaptive when the selection it performs at a given node of the search tree depends on the current state of the problem instance as well as on the past states encountered so far.

In other words, some information concerning the sub-tree already explored is taken into account by an adaptive heuristic to perform its selection. It implements then a kind of learning, as for example:

- *wdeg*: constraint weighting
- *impacts* and *activity*: memorization of search space reductions
- *lc* (last conflicts): memorization of the last failed assignments // used in conjunction with a variable heuristic

Our focus is on wdeg and lc, which aim at guiding search towards conflicts.

Adaptive Heuristics

A search-guiding heuristic is said to be adaptive when the selection it performs at a given node of the search tree depends on the current state of the problem instance as well as on the past states encountered so far.

In other words, some information concerning the sub-tree already explored is taken into account by an adaptive heuristic to perform its selection. It implements then a kind of learning, as for example:

- *wdeg*: constraint weighting
- *impacts* and *activity*: memorization of search space reductions
- *lc* (last conflicts): memorization of the last failed assignments // used in conjunction with a variable heuristic

Our focus is on wdeg and lc, which aim at guiding search towards conflicts.

Adaptive Heuristics

A search-guiding heuristic is said to be adaptive when the selection it performs at a given node of the search tree depends on the current state of the problem instance as well as on the past states encountered so far.

In other words, some information concerning the sub-tree already explored is taken into account by an adaptive heuristic to perform its selection. It implements then a kind of learning, as for example:

- *wdeg*: constraint weighting
- *impacts* and *activity*: memorization of search space reductions
- *lc* (last conflicts): memorization of the last failed assignments // used in conjunction with a variable heuristic

Our focus is on wdeg and lc, which aim at guiding search towards conflicts.

Adaptive Heuristics

A search-guiding heuristic is said to be adaptive when the selection it performs at a given node of the search tree depends on the current state of the problem instance as well as on the past states encountered so far.

In other words, some information concerning the sub-tree already explored is taken into account by an adaptive heuristic to perform its selection. It implements then a kind of learning, as for example:

- *wdeg*: constraint weighting
- *impacts* and *activity*: memorization of search space reductions
- *lc* (last conflicts): memorization of the last failed assignments // used in conjunction with a variable heuristic

Our focus is on wdeg and lc, which aim at guiding search towards conflicts.

Adaptive Heuristics

A search-guiding heuristic is said to be adaptive when the selection it performs at a given node of the search tree depends on the current state of the problem instance as well as on the past states encountered so far.

In other words, some information concerning the sub-tree already explored is taken into account by an adaptive heuristic to perform its selection. It implements then a kind of learning, as for example:

- *wdeg*: constraint weighting
- *impacts* and *activity*: memorization of search space reductions
- *lc* (last conflicts): memorization of the last failed assignments // used in conjunction with a variable heuristic

Our focus is on wdeg and lc, which aim at guiding search towards conflicts.

Constraint Weighting

The principle is the following:

- a weight is associated with each constraint,
- everytime a conflict occurs while filtering a constraint c, the weight associated with c is incremented.
- the weigth of a variable is the sum of the weigths of all its involving constraints.

The interest is that this heuristic is adaptive, with the expectation to focus on the hard part(s) of the instance.

Constraint Weighting

The principle is the following:

- a weight is associated with each constraint,
- everytime a conflict occurs while filtering a constraint c, the weight associated with c is incremented.
- the weigth of a variable is the sum of the weigths of all its involving constraints.

The interest is that this heuristic is adaptive, with the expectation to focus on the hard part(s) of the instance.

Implementation

Algorithm 7: constraintPropagationOn(P: CN): Boolean

$Q \leftarrow ctrs(P)$
while $Q \neq \emptyset$ **do**
    pick and delete c from Q
    $X_{evt} \leftarrow c.filter()$ `// ` X_{evt} `denotes the set of variables with reduced domains (after filtering by means of` c`)`
    if $\exists x \in X_{evt}$ *such that* $dom(x) = \emptyset$ **then**
        $weight[c] \leftarrow weight[c] + 1$
        return *false* `// global inconsistency detected`
    foreach $c' \in ctrs(P)$ *such that* $c' \neq c$ *and* $X_{evt} \cap scp(c') \neq \emptyset$ **do**
        add c' to Q
return *true*

Illustration with queensKnights-8-3-mul

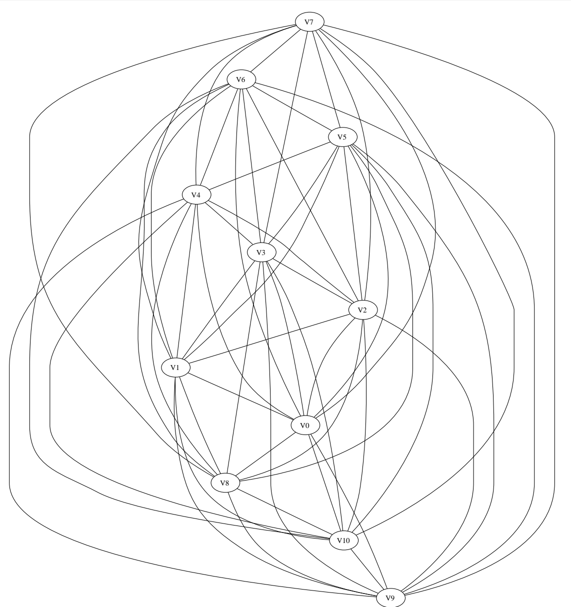


Illustration with queensKnights-8-3-mul

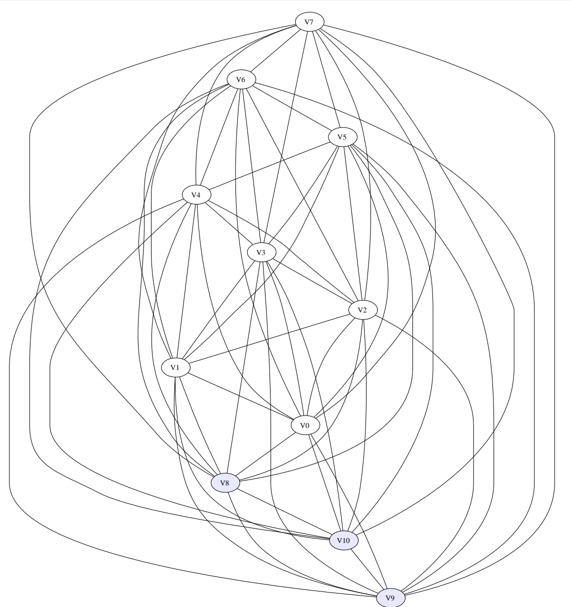


Illustration with queensKnights-8-3-mul

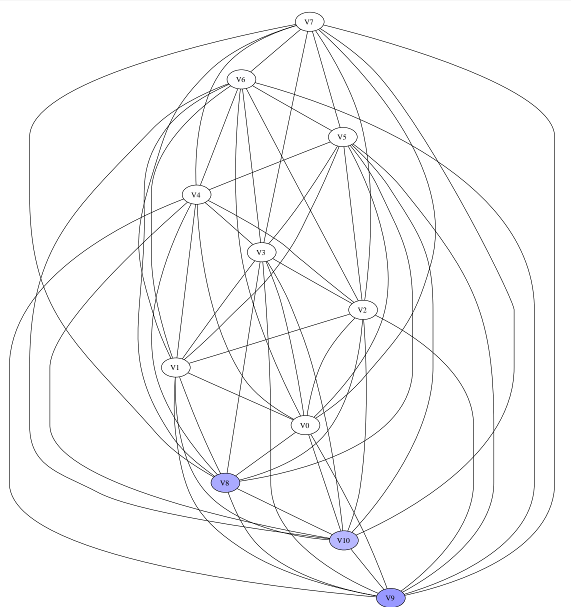


Illustration with queensKnights-8-3-mul

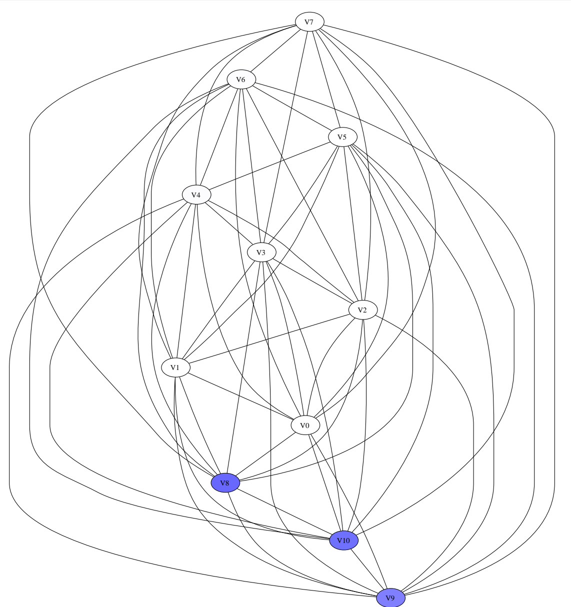


Illustration with scen11-f6

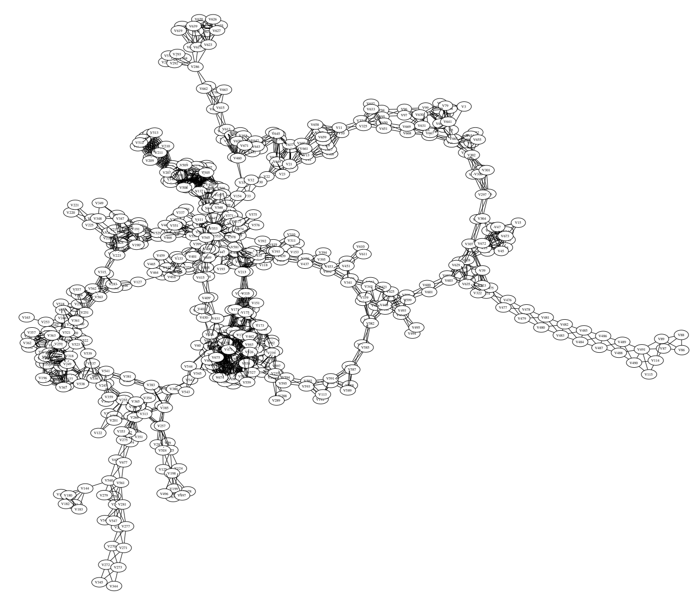


Illustration with scen11-f6

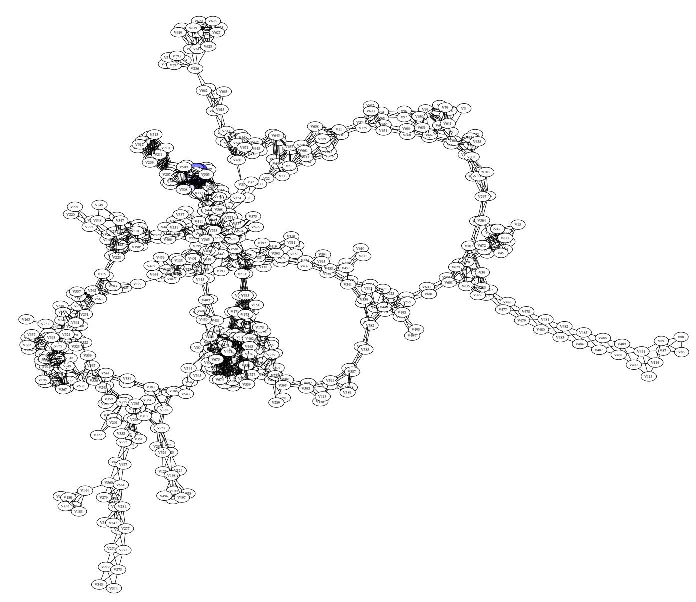
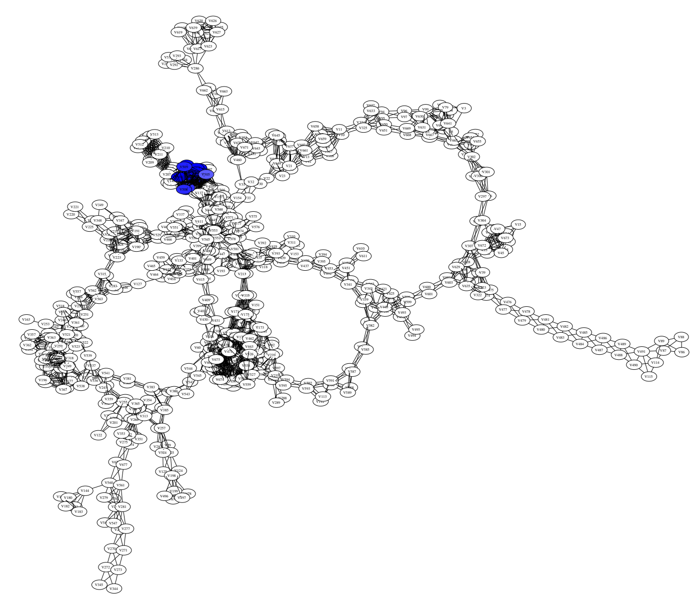
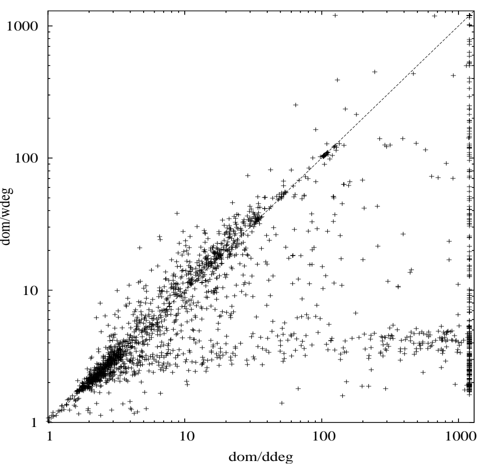


Illustration with scen11-f6

Illustration with scen11-f6



Experimental Results



Pairwise comparison (CPU time) of heuristics when used by MAC to solve the instances from XCSP constraint solver competition.

Last-conflict based Reasoning

The principle is the following: everytime a conflict occurs, the last assigned variable is selected in priority as long as no consistent value is found for it.

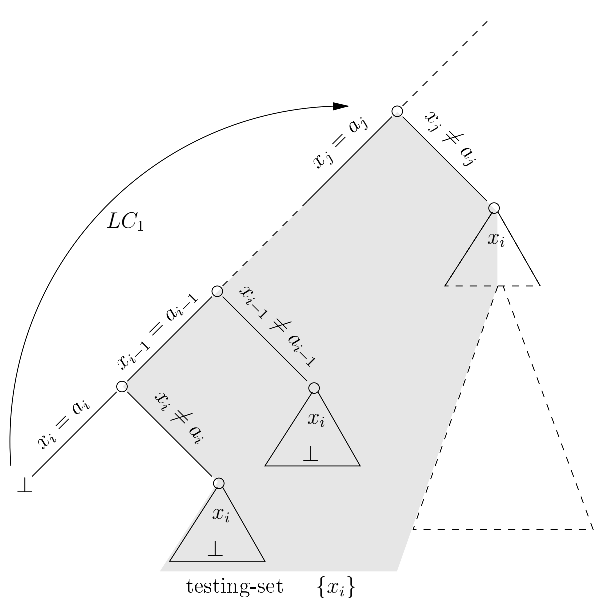
It looks like a lazy identification of nogoods.

Implementation

Algorithm 8: binary-ϕ-search^{LC}(P: ϕ-consistent CN): Boolean

```
if P = ⊥ then
    return false
if ∀x ∈ vars(P), |dom(x)| = 1 then
    return true
if priority ≠ null then
    x ← priority
else
    x ← variableOrderingHeuristic.selectVariable()
a ← valueOrderingHeuristic.selectValueFor(x)
if φ(P|x=a) = ⊥ then
    priority ← x
else
    priority ← null
return binary-φ-search^LC(φ(P|x=a)) ∨ binary-φ-search^LC(φ(P|x≠a))
```

Illustration



Example

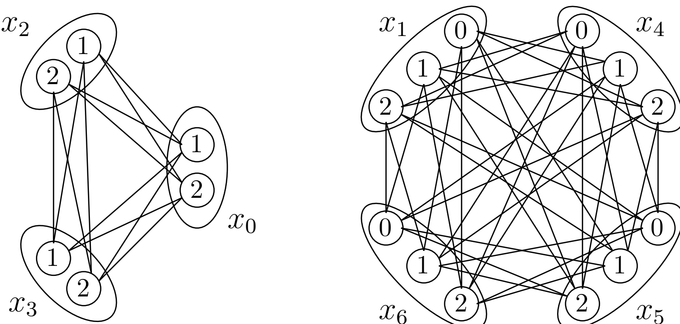


Figure: The compatibility graph of a constraint network involving a clique of constraints of difference and a clique of entailed constraints.

Example

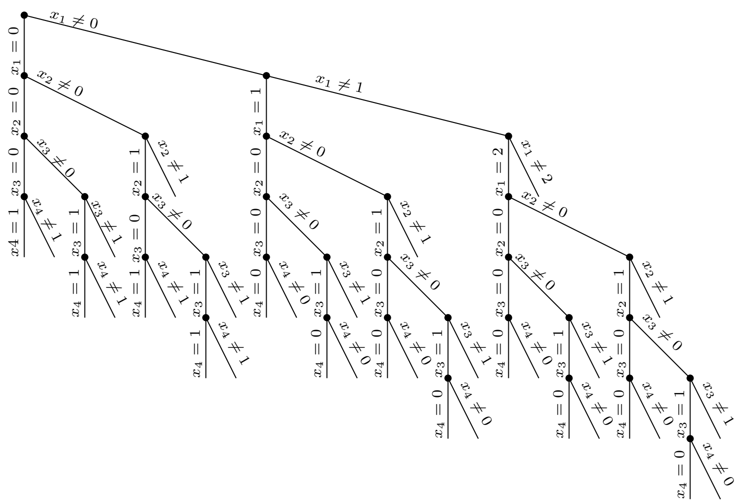


Figure: Search tree built by MAC (68 explored nodes).

Example

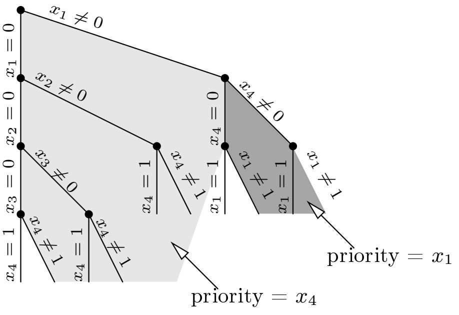
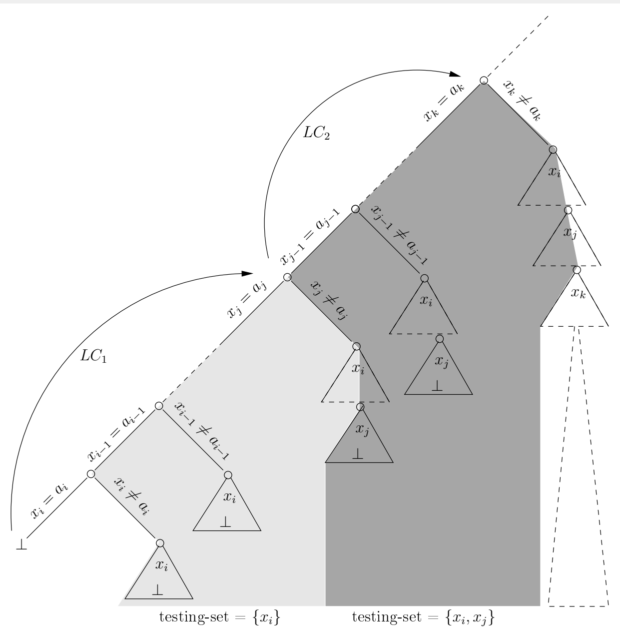


Figure: Search tree built by MAC-LC$_1$ (21 explored nodes).

Generalization



Bessiere, C., & Régin, J. 1996.
MAC and combined heuristics: two reasons to forsake FC (and CBJ?) on hard problems.
Pages 61–75 of: Proceedings of CP'96.

Boussemart, F., Hemery, F., Lecoutre, C., & Sais, L. 2004.
Boosting systematic search by weighting constraints.
Pages 146–150 of: Proceedings of ECAI'04.

Brelaz, D. 1979.
New methods to color the vertices of a graph.
Communications of the ACM, **22**, 251–256.

Dechter, R., & Meiri, I. 1989.
Experimental evaluation of preprocessing techniques in constraint satisfaction problems.
Pages 271–277 of: Proceedings of IJCAI'89.

Haralick, R.M., & Elliott, G.L. 1980.
Increasing tree search efficiency for constraint satisfaction problems.
Artificial Intelligence, **14**, 263–313.

Michel, L., & Hentenryck, P. Van. 2012.
Activity-Based Search for Black-Box Constraint Programming Solvers.
Pages 228–243 of: Proceedings of CPAIOR'12.

Refalo, P. 2004.
Impact-based search strategies for constraint programming.
Pages 557–571 of: Proceedings of CP'04.

Smith, B.M. 1999.
The Brelaz heuristic and optimal static orderings.
Pages 405–418 of: Proceedings of CP'99.

Ullmann, J.R. 1976.
An algorithm for subgraph isomorphism.
Journal of the ACM, **23**(1), 31–42.