Constraint Programming
– Optimizing –

Christophe Lecoutre
lecoutre@cril.fr

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

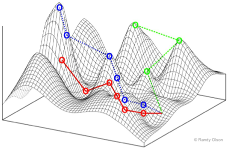
January 2021

Outline

1. Complete Approaches for Optimization
2. Complete vs Incomplete Approaches
3. Large Neighborhood Search

Outline

Constraint Optimization



For Constraint Optimization Problems (COPs), solvers must find a complete instantiation of the variables such that:

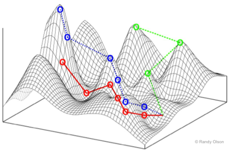
- all constraints are satisfied
- the objective function is optimized

Important: It is not possible to stop at the first found solution

Two related approaches:

- Branch and Bound
- Iterative Optimization

Constraint Optimization



For Constraint Optimization Problems (COPs), solvers must find a complete instantiation of the variables such that:

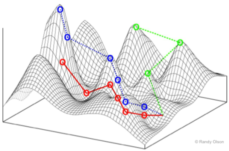
- all constraints are satisfied
- the objective function is optimized

Important: It is not possible to stop at the first found solution

Two related approaches:

- Branch and Bound
- Iterative Optimization

Constraint Optimization



For Constraint Optimization Problems (COPs), solvers must find a complete instantiation of the variables such that:

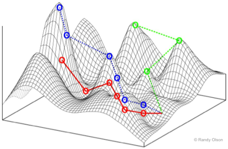
- all constraints are satisfied
- the objective function is optimized

Important: It is not possible to stop at the first found solution

Two related approaches:

- Branch and Bound
- Iterative Optimization

Constraint Optimization



For Constraint Optimization Problems (COPs), solvers must find a complete instantiation of the variables such that:

- all constraints are satisfied
- the objective function is optimized

Important: It is not possible to stop at the first found solution

Two related approaches:

- Branch and Bound
- Iterative Optimization

Branch and Bound

For an objective function f represented by an arithmetic expression, at each new solution S, add a constraint:

- $f < f(S)$, for minimization
- $f > f(S)$, for maximization

Stop when no more solutions

Remark.
A proof of optimality can be obtained from the last found solution.

Branch and Bound

For an objective function f represented by an arithmetic expression, at each new solution S, add a constraint:

- $f < f(S)$, for minimization
- $f > f(S)$, for maximization

Stop when no more solutions

Remark.
A proof of optimality can be obtained from the last found solution.

Branch and Bound

For an objective function f represented by an arithmetic expression, at each new solution S, add a constraint:

- $f < f(S)$, for minimization
- $f > f(S)$, for maximization

Stop when no more solutions

Remark.
A proof of optimality can be obtained from the last found solution.

Iterative Optimization

For minimization:

1. compute a lower bound *lb* of the objective function *f*
2. add a constraint $f = lb$ to the problem instance, which then becomes a CSP instance
3. solve the CSP instance
   - if a solution is found, it is optimal
   - if no solution is found, increase *lb* and restart

At the end of the process, an optimal solution is obtained.

Remark.
We proceed similarly for maximization.

Iterative Optimization

For minimization:

1. compute a lower bound *lb* of the objective function *f*
2. add a constraint $f = lb$ to the problem instance, which then becomes a CSP instance
3. solve the CSP instance
   - if a solution is found, it is optimal
   - if no solution is found, increase *lb* and restart

At the end of the process, an optimal solution is obtained.

Remark.
We proceed similarly for maximization.

Iterative Optimization

For minimization:

1. compute a lower bound lb of the objective function f
2. add a constraint $f = lb$ to the problem instance, which then becomes a CSP instance
3. solve the CSP instance
   - if a solution is found, it is optimal
   - if no solution is found, increase lb and restart

At the end of the process, an optimal solution is obtained.

Remark.
We proceed similarly for maximization.

Iterative Optimization

For minimization:

1. compute a lower bound *lb* of the objective function f
2. add a constraint $f = lb$ to the problem instance, which then becomes a CSP instance
3. solve the CSP instance
   - if a solution is found, it is optimal
   - if no solution is found, increase *lb* and restart

At the end of the process, an optimal solution is obtained.

Remark.
We proceed similarly for maximization.

Iterative Optimization

For minimization:

1. compute a lower bound lb of the objective function f
2. add a constraint $f = lb$ to the problem instance, which then becomes a CSP instance
3. solve the CSP instance
   - if a solution is found, it is optimal
   - if no solution is found, increase lb and restart

At the end of the process, an optimal solution is obtained.

Remark.
We proceed similarly for maximization.

Iterative Optimization

For minimization:

1. compute a lower bound *lb* of the objective function f
2. add a constraint $f = lb$ to the problem instance, which then becomes a CSP instance
3. solve the CSP instance
   - if a solution is found, it is optimal
   - if no solution is found, increase *lb* and restart

At the end of the process, an optimal solution is obtained.

Remark.
We proceed similarly for maximization.

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$ ✗
CSP Search, with the constraint $f = 6$ ✗
CSP Search, with the constraint $f = 7$ ✗
CSP Search, with the constraint $f = 8$ ✗
CSP Search, with the constraint $f = 9$ ✗
CSP Search, with the constraint $f = 10$ ✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$ ✗
CSP Search, with the constraint $f = 6$ ✗
CSP Search, with the constraint $f = 7$ ✗
CSP Search, with the constraint $f = 8$ ✗
CSP Search, with the constraint $f = 9$ ✗
CSP Search, with the constraint $f = 10$ ✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$ ✗
CSP Search, with the constraint $f = 6$ ✗
CSP Search, with the constraint $f = 7$ ✗
CSP Search, with the constraint $f = 8$ ✗
CSP Search, with the constraint $f = 9$ ✗
CSP Search, with the constraint $f = 10$ ✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$ ✗
CSP Search, with the constraint $f = 6$ ✗
CSP Search, with the constraint $f = 7$ ✗
CSP Search, with the constraint $f = 8$ ✗
CSP Search, with the constraint $f = 9$ ✗
CSP Search, with the constraint $f = 10$ ✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$ ✗
CSP Search, with the constraint $f = 6$ ✗
CSP Search, with the constraint $f = 7$ ✗
CSP Search, with the constraint $f = 8$ ✗
CSP Search, with the constraint $f = 9$ ✗
CSP Search, with the constraint $f = 10$ ✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$ ✗
CSP Search, with the constraint $f = 6$ ✗
CSP Search, with the constraint $f = 7$ ✗
CSP Search, with the constraint $f = 8$ ✗
CSP Search, with the constraint $f = 9$ ✗
CSP Search, with the constraint $f = 10$ ✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$ ✗
CSP Search, with the constraint $f = 6$ ✗
CSP Search, with the constraint $f = 7$ ✗
CSP Search, with the constraint $f = 8$ ✗
CSP Search, with the constraint $f = 9$ ✗
CSP Search, with the constraint $f = 10$ ✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$ ✗
CSP Search, with the constraint $f = 6$ ✗
CSP Search, with the constraint $f = 7$ ✗
CSP Search, with the constraint $f = 8$ ✗
CSP Search, with the constraint $f = 9$ ✗
CSP Search, with the constraint $f = 10$ ✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$ ✗
CSP Search, with the constraint $f = 6$ ✗
CSP Search, with the constraint $f = 7$ ✗
CSP Search, with the constraint $f = 8$ ✗
CSP Search, with the constraint $f = 9$ ✗
CSP Search, with the constraint $f = 10$ ✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$ ✗
CSP Search, with the constraint $f = 6$ ✗
CSP Search, with the constraint $f = 7$ ✗
CSP Search, with the constraint $f = 8$ ✗
CSP Search, with the constraint $f = 9$ ✗
CSP Search, with the constraint $f = 10$ ✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$ ✗
CSP Search, with the constraint $f = 6$ ✗
CSP Search, with the constraint $f = 7$ ✗
CSP Search, with the constraint $f = 8$ ✗
CSP Search, with the constraint $f = 9$ ✗
CSP Search, with the constraint $f = 10$ ✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$	✗
CSP Search, with the constraint $f = 6$	✗
CSP Search, with the constraint $f = 7$	✗
CSP Search, with the constraint $f = 8$	✗
CSP Search, with the constraint $f = 9$	✗
CSP Search, with the constraint $f = 10$	✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$	✗
CSP Search, with the constraint $f = 6$	✗
CSP Search, with the constraint $f = 7$	✗
CSP Search, with the constraint $f = 8$	✗
CSP Search, with the constraint $f = 9$	✗
CSP Search, with the constraint $f = 10$	✓

Illustration of Iterative Optimization

Suppose that the optimal value of the objective function is 10 and that we initially compute a lower-bound $lb = 5$.

CSP Search, with the constraint $f = 5$ ✗
CSP Search, with the constraint $f = 6$ ✗
CSP Search, with the constraint $f = 7$ ✗
CSP Search, with the constraint $f = 8$ ✗
CSP Search, with the constraint $f = 9$ ✗
CSP Search, with the constraint $f = 10$ ✓

Optimization Types

An objective function can be represented by:

- a variable, as for example in:

```
minimize x
```

- a general expression, typically based on arithmetic operators, as for example in:

```
minimize x*y + z
```

- a specialized expression indicating what we must compute:
  - a sum
  - a minimum
  - a maximum
  - a number of distinct values
  - a tuple, compared lexicographically to others

Optimization Types

An objective function can be represented by:

- a variable, as for example in:

  ```
  minimize x
  ```

- a general expression, typically based on arithmetic operators, as for example in:

  ```
  minimize x*y + z
  ```

- a specialized expression indicating what we must compute:
  - a sum
  - a minimum
  - a maximum
  - a number of distinct values
  - a tuple, compared lexicographically to others

Optimization Types

An objective function can be represented by:

- a variable, as for example in:

```
minimize x
```

- a general expression, typically based on arithmetic operators, as for example in:

```
minimize x*y + z
```

- a specialized expression indicating what we must compute:
  - a sum
  - a minimum
  - a maximum
  - a number of distinct values
  - a tuple, compared lexicographically to others

Optimization Types

An objective function can be represented by:

- a variable, as for example in:

```
minimize x
```

- a general expression, typically based on arithmetic operators, as for example in:

```
minimize x*y + z
```

- a specialized expression indicating what we must compute:
  - a sum
  - a minimum
  - a maximum
  - a number of distinct values
  - a tuple, compared lexicographically to others

Specialized Optimization Expressions

They involve:

- a sequence of variables X
- possibly, a sequence of coefficients C
- an operator that can be sum, product, ...

The semantics is:

- $\texttt{minimize}(X, C, sum)$: minimize $\sum_{i=1}^{|X|} c_i \times x_i$
- $\texttt{minimize}(X, C, minimum)$: minimize $\min_{i=1}^{|X|} c_i \times x_i$
- $\texttt{minimize}(X, C, maximum)$: minimize $\max_{i=1}^{|X|} c_i \times x_i$
- $\texttt{minimize}(X, C, nValues)$: minimize $|\{c_i \times x_i : 1 \leq i \leq |X|\}|$
- $\texttt{minimize}(X, C, lex)$: minimize$_{lex}$ $\langle c_1 \times x_1, c_2 \times x_2, \ldots, c_k \times x_k \rangle$

Remark.
Of course, coefficients can be ignored when they are all equal to 1.

Specialized Optimization Expressions

They involve:

- a sequence of variables X
- possibly, a sequence of coefficients C
- an operator that can be sum, product, ...

The semantics is:

- $\texttt{minimize}(X, C, sum)$: minimize $\sum_{i=1}^{|X|} c_i \times x_i$
- $\texttt{minimize}(X, C, minimum)$: minimize $\min_{i=1}^{|X|} c_i \times x_i$
- $\texttt{minimize}(X, C, maximum)$: minimize $\max_{i=1}^{|X|} c_i \times x_i$
- $\texttt{minimize}(X, C, nValues)$: minimize $|\{c_i \times x_i : 1 \leq i \leq |X|\}|$
- $\texttt{minimize}(X, C, lex)$: minimize$_{lex}$ $\langle c_1 \times x_1, c_2 \times x_2, \ldots, c_k \times x_k \rangle$

Remark.

Of course, coefficients can be ignored when they are all equal to 1.

Specialized Optimization Expressions

They involve:

- a sequence of variables X
- possibly, a sequence of coefficients C
- an operator that can be sum, product, ...

The semantics is:

- $\texttt{minimize}(X, C, sum)$: minimize $\sum_{i=1}^{|X|} c_i \times x_i$
- $\texttt{minimize}(X, C, minimum)$: minimize $\min_{i=1}^{|X|} c_i \times x_i$
- $\texttt{minimize}(X, C, maximum)$: minimize $\max_{i=1}^{|X|} c_i \times x_i$
- $\texttt{minimize}(X, C, nValues)$: minimize $|\{c_i \times x_i : 1 \leq i \leq |X|\}|$
- $\texttt{minimize}(X, C, lex)$: minimize$_{lex}$ $\langle c_1 \times x_1, c_2 \times x_2, \ldots, c_k \times x_k \rangle$

Remark.
Of course, coefficients can be ignored when they are all equal to 1.

Optimization Illustration in PyCSP^3

File Rlfap.py

```
from pycsp3 import *

domains, variables, constraints, _, _ = data
n = len(variables)

# f[i] is the frequency of the ith radio link
f = VarArray(size=n, dom=lambda i: domains[variables[i].domain])

satisfy(
  # managing pre-assigned frequencies
  [f[i] == v for i, (_, v, mob) in enumerate(variables) if v],

  # hard constraints on radio-links
  [expr(op, abs(f[i] - f[j]), k) for (i, j, op, k, _) in constraints]
)

if variant("span"):
  minimize(
    # minimizing the largest frequency
    Maximum(f)
  )
elif variant("card"):
  minimize(
    # minimizing the number of used frequencies
    NValues(f)
  )
```

How to implement Branch and Bound?

If the objective is a variable x, then post a constraint $x < k$.

If the objective is given by a specialized expression, post one of the following constraints:

- `sum`
- `minimum`
- `maximum`
- `nValues`
- `lex`

integrating a condition $(\odot, k)$, which is $(<, k)$.

Remark.

- k is initially an upper bound of the optimum (possibly, $+\infty$)
- k is modified every time a new solution is found

How to implement Branch and Bound?

If the objective is a variable x, then post a constraint $x < k$.

If the objective is given by a specialized expression, post one of the following constraints:

- `sum`
- `minimum`
- `maximum`
- `nValues`
- `lex`

integrating a condition $(\odot, k)$, which is $(<, k)$.

Remark.

- k is initially an upper bound of the optimum (possibly, $+\infty$)
- k is modified every time a new solution is found

How to implement Branch and Bound?

If the objective is a variable x, then post a constraint $x < k$.

If the objective is given by a specialized expression, post one of the following constraints:

- `sum`
- `minimum`
- `maximum`
- `nValues`
- `lex`

integrating a condition $(\odot, k)$, which is $(<, k)$.

Remark.

- k is initially an upper bound of the optimum (possibly, $+\infty$)
- k is modified every time a new solution is found

Optimization Strategies

Minimization being assumed, Branch and Bound and Iterative optimization, correspond to two different stategies for guiding optimization search:

- decreasingly with Branch and Bound, as k is continually reduced
- increasingly with Iterative Optimization, as k is continually augmented

Why not using a dichotomic process? At any moment, we must know

- the best objective value b that has been obtained so far
- the interval of bounds $I = lb..ub$ where to search.

Then, as long as I is not empty, we run search in $lb..(ub - lb)/2$:

- if a solution of cost b' is found, b is updated (with value b') and I becomes $lb..b' - 1$
- if no solution is found, I becomes $(ul - lb)/2 + 1..ub$

Optimization Strategies

Minimization being assumed, Branch and Bound and Iterative optimization, correspond to two different stategies for guiding optimization search:

- decreasingly with Branch and Bound, as k is continually reduced
- increasingly with Iterative Optimization, as k is continually augmented

Why not using a dichotomic process? At any moment, we must know

- the best objective value b that has been obtained so far
- the interval of bounds $I = lb..ub$ where to search.

Then, as long as I is not empty, we run search in $lb..(ub - lb)/2$:

- if a solution of cost b' is found, b is updated (with value b') and I becomes $lb..b' - 1$
- if no solution is found, I becomes $(ul - lb)/2 + 1..ub$

Optimization Strategies

Minimization being assumed, Branch and Bound and Iterative optimization, correspond to two different stategies for guiding optimization search:

- decreasingly with Branch and Bound, as k is continually reduced
- increasingly with Iterative Optimization, as k is continually augmented

Why not using a dichotomic process? At any moment, we must know

- the best objective value b that has been obtained so far
- the interval of bounds $I = lb..ub$ where to search.

Then, as long as I is not empty, we run search in $lb..(ub - lb)/2$:

- if a solution of cost b' is found, b is updated (with value b') and I becomes $lb..b' - 1$
- if no solution is found, I becomes $(ul - lb)/2 + 1..ub$

Illustration of a Dichotomic Search

Initially, we have $b = \perp$ (no solution found) and $I = 0..100$.

Search in 0..50 ✗ $\Rightarrow I = 51..100$
Search in 51..75 ✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60 ✗ $\Rightarrow I = 61..71$
Search in 61..65 ✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62 ✗ $\Rightarrow I = 63..64$
Search in 63..63 ✗ $\Rightarrow I = 64..64$
Search in 64..64 ✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \perp$ (no solution found) and $I = 0..100$.

Search in 0..50 ✗ $\Rightarrow I = 51..100$
Search in 51..75 ✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60 ✗ $\Rightarrow I = 61..71$
Search in 61..65 ✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62 ✗ $\Rightarrow I = 63..64$
Search in 63..63 ✗ $\Rightarrow I = 64..64$
Search in 64..64 ✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \bot$ (no solution found) and $I = 0..100$.

Search in 0..50 ✗ $\Rightarrow I = 51..100$
Search in 51..75 ✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60 ✗ $\Rightarrow I = 61..71$
Search in 61..65 ✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62 ✗ $\Rightarrow I = 63..64$
Search in 63..63 ✗ $\Rightarrow I = 64..64$
Search in 64..64 ✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \bot$ (no solution found) and $I = 0..100$.

Search in 0..50	✗ $\Rightarrow I = 51..100$
Search in 51..75	✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60	✗ $\Rightarrow I = 61..71$
Search in 61..65	✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62	✗ $\Rightarrow I = 63..64$
Search in 63..63	✗ $\Rightarrow I = 64..64$
Search in 64..64	✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \bot$ (no solution found) and $I = 0..100$.

Search in 0..50 ✗ $\Rightarrow I = 51..100$
Search in 51..75 ✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60 ✗ $\Rightarrow I = 61..71$
Search in 61..65 ✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62 ✗ $\Rightarrow I = 63..64$
Search in 63..63 ✗ $\Rightarrow I = 64..64$
Search in 64..64 ✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \perp$ (no solution found) and $I = 0..100$.

Search in 0..50 ✗ $\Rightarrow I = 51..100$
Search in 51..75 ✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60 ✗ $\Rightarrow I = 61..71$
Search in 61..65 ✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62 ✗ $\Rightarrow I = 63..64$
Search in 63..63 ✗ $\Rightarrow I = 64..64$
Search in 64..64 ✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \perp$ (no solution found) and $I = 0..100$.

Search in 0..50	✗ $\Rightarrow I = 51..100$
Search in 51..75	✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60	✗ $\Rightarrow I = 61..71$
Search in 61..65	✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62	✗ $\Rightarrow I = 63..64$
Search in 63..63	✗ $\Rightarrow I = 64..64$
Search in 64..64	✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \bot$ (no solution found) and $I = 0..100$.

Search in 0..50	✗ $\Rightarrow I = 51..100$
Search in 51..75	✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60	✗ $\Rightarrow I = 61..71$
Search in 61..65	✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62	✗ $\Rightarrow I = 63..64$
Search in 63..63	✗ $\Rightarrow I = 64..64$
Search in 64..64	✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \bot$ (no solution found) and $I = 0..100$.

Search in 0..50 ✗ $\Rightarrow I = 51..100$
Search in 51..75 ✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60 ✗ $\Rightarrow I = 61..71$
Search in 61..65 ✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62 ✗ $\Rightarrow I = 63..64$
Search in 63..63 ✗ $\Rightarrow I = 64..64$
Search in 64..64 ✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \perp$ (no solution found) and $I = 0..100$.

Search in 0..50	✗ $\Rightarrow I = 51..100$
Search in 51..75	✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60	✗ $\Rightarrow I = 61..71$
Search in 61..65	✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62	✗ $\Rightarrow I = 63..64$
Search in 63..63	✗ $\Rightarrow I = 64..64$
Search in 64..64	✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \perp$ (no solution found) and $I = 0..100$.

Search in 0..50	✗ $\Rightarrow I = 51..100$
Search in 51..75	✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60	✗ $\Rightarrow I = 61..71$
Search in 61..65	✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62	✗ $\Rightarrow I = 63..64$
Search in 63..63	✗ $\Rightarrow I = 64..64$
Search in 64..64	✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \bot$ (no solution found) and $I = 0..100$.

Search in 0..50	✗ $\Rightarrow I = 51..100$
Search in 51..75	✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60	✗ $\Rightarrow I = 61..71$
Search in 61..65	✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62	✗ $\Rightarrow I = 63..64$
Search in 63..63	✗ $\Rightarrow I = 64..64$
Search in 64..64	✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \perp$ (no solution found) and $I = 0..100$.

Search in 0..50	✗ $\Rightarrow I = 51..100$
Search in 51..75	✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60	✗ $\Rightarrow I = 61..71$
Search in 61..65	✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62	✗ $\Rightarrow I = 63..64$
Search in 63..63	✗ $\Rightarrow I = 64..64$
Search in 64..64	✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \perp$ (no solution found) and $I = 0..100$.

Search in 0..50 ✗ $\Rightarrow I = 51..100$
Search in 51..75 ✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60 ✗ $\Rightarrow I = 61..71$
Search in 61..65 ✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62 ✗ $\Rightarrow I = 63..64$
Search in 63..63 ✗ $\Rightarrow I = 64..64$
Search in 64..64 ✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \bot$ (no solution found) and $I = 0..100$.

Search in 0..50	✗ $\Rightarrow I = 51..100$
Search in 51..75	✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60	✗ $\Rightarrow I = 61..71$
Search in 61..65	✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62	✗ $\Rightarrow I = 63..64$
Search in 63..63	✗ $\Rightarrow I = 64..64$
Search in 64..64	✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \bot$ (no solution found) and $I = 0..100$.

Search in 0..50	✗ $\Rightarrow I = 51..100$
Search in 51..75	✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60	✗ $\Rightarrow I = 61..71$
Search in 61..65	✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62	✗ $\Rightarrow I = 63..64$
Search in 63..63	✗ $\Rightarrow I = 64..64$
Search in 64..64	✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Illustration of a Dichotomic Search

Initially, we have $b = \perp$ (no solution found) and $I = 0..100$.

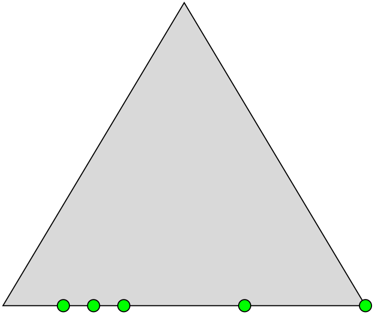
Search in 0..50	✗ $\Rightarrow I = 51..100$
Search in 51..75	✓ $b = 72 \Rightarrow I = 51..71$
Search in 51..60	✗ $\Rightarrow I = 61..71$
Search in 61..65	✓ $b = 65 \Rightarrow I = 61..64$
Search in 61..62	✗ $\Rightarrow I = 63..64$
Search in 63..63	✗ $\Rightarrow I = 64..64$
Search in 64..64	✗ $\Rightarrow I = \emptyset$

Optimum proved at 65

Outline

Exploration of the Search Tree

The search tree may look like:



with a few solutions represented by green circles here.

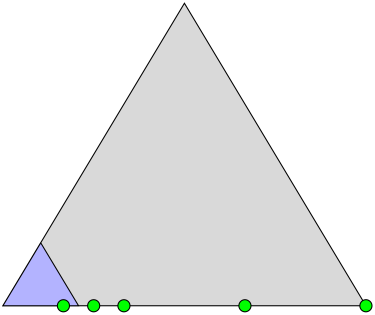
Diversification?

When the problem is too hard to be solved to optimality:

- the search is stopped after a time/backtrack limit
- and the best found solution may not be optimal

Importantly,

- branch and bound usually does not show good diversification
- can even fail to find a single solution



Iterative Optimization

Iterative Optimization is not adapted at all at solving hard problems.

- the search is only conducted for proving optimality
- consequently, if the search space is too large, iterative optimization does not find any solution

Remark.
Dichotomic variants may suffer from the same behaviour.

Iterative Optimization

Iterative Optimization is not adapted at all at solving hard problems.

- the search is only conducted for proving optimality
- consequently, if the search space is too large, iterative optimization does not find any solution

Remark.

Dichotomic variants may suffer from the same behaviour.

What to do when a problem is too hard?

On hard problems, should we use complete or incomplete methods?

Complete methods suffer from extensive solving time:

- completeness is a great asset
- but sometimes it is too costly

Incomplete methods can always be controlled:

- they usually find good solutions quickly
- but solutions may not be optimal
- or not known to be (no proof of optimality)

What to do when a problem is too hard?

On hard problems, should we use complete or incomplete methods?

Complete methods suffer from extensive solving time:

- completeness is a great asset
- but sometimes it is too costly

Incomplete methods can always be controlled:

- they usually find good solutions quickly
- but solutions may not be optimal
- or not known to be (no proof of optimality)

What to do when a problem is too hard?

On hard problems, should we use complete or incomplete methods?

Complete methods suffer from extensive solving time:

- completeness is a great asset
- but sometimes it is too costly

Incomplete methods can always be controlled:

- they usually find good solutions quickly
- but solutions may not be optimal
- or not known to be (no proof of optimality)

Constraint-based Local Search

Constraint Programming:

- modeling based on constraints
- constructive approach
- complete search
- solving model: branch and propagate

⇒ finds optimal solutions (but can take too much time)

Constraint-based Local Search:

- modeling based on constraints
- perturbative approach
- incomplete search
- solving model: neighborhoods

⇒ finds good solutions (quickly)

Local Search

Local Search (LS) proceeds as follows:

- LS handles one complete instantiation:
  - called current “solution”
  - which is is not optimal or not known to be
- LS iteratively improves the solution:
  - by defining the neighborhood of the current solution
  - by selecting one of the neighbors
  - by accepting it (or not) as the new current solution
- LS searches new solutions close to the current one, hence the name local search

Constraint-based Local Search

Constraint-based Local Search (CBLS) uses the same principle as LS, but focuses on constraints:

- some hard constraints cannot be violated (in the current solution)
- some soft constraints can be violated

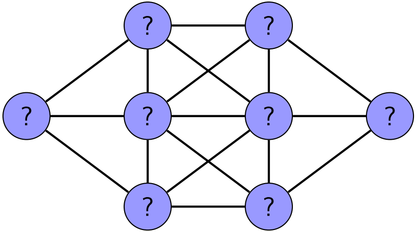
As LS, CBLS tries to iteratively improve the current solution. But it benefits from:

- information: the way soft constraints are violated
- reduction: propagation on hard constraints and objective function

Example

Problem: assign to each node of the following graph a number from 1 to 8 such that:

- each number appears only once
- no two adjacent nodes have consecutive numbers



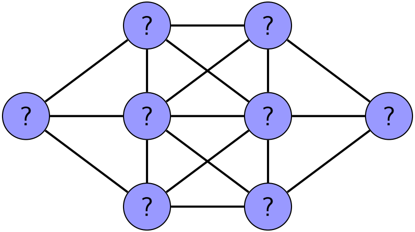
CBLS model:

- hard constraint: each number appears only once
- soft constraint(s): no two adjacent nodes have consecutive numbers
- objective: to minimize violations of soft-constraint(s)

Example

Problem: assign to each node of the following graph a number from 1 to 8 such that:

- each number appears only once
- no two adjacent nodes have consecutive numbers

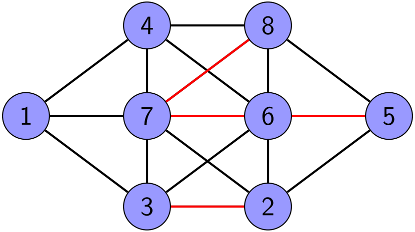


CBLS model:

- hard constraint: each number appears only once
- soft constraint(s): no two adjacent nodes have consecutive numbers
- objective: to minimize violations of soft-constraint(s)

Example

Assume that the initial solution is:



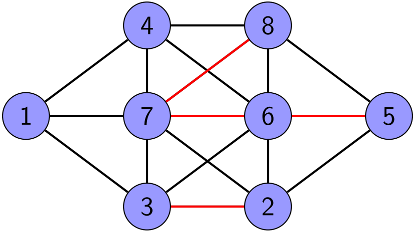
Note the violation cost: 4, as the number of violated binary constraints

What about the possibles moves?

- neighborhood: defined by swapping the values of two nodes,
- which guarantees that the hard constraint `allDifferent` remains satisfied.

Example

Assume that the initial solution is:



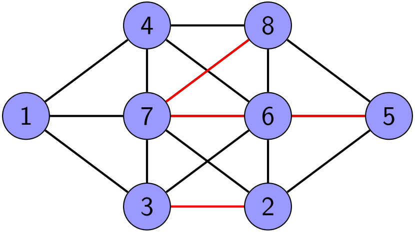
Note the violation cost: 4, as the number of violated binary constraints

What about the possibles moves?

- neighborhood: defined by swapping the values of two nodes,
- which guarantees that the hard constraint `allDifferent` remains satisfied.

Example

Assume that the initial solution is:



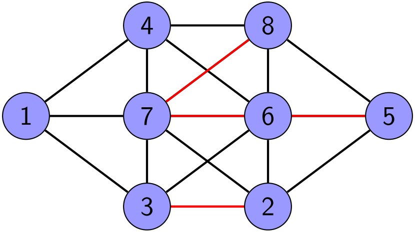
Note the violation cost: 4, as the number of violated binary constraints

What about the possibles moves?

- neighborhood: defined by swapping the values of two nodes,
- which guarantees that the hard constraint `allDifferent` remains satisfied.

Example

Assume that the initial solution is:



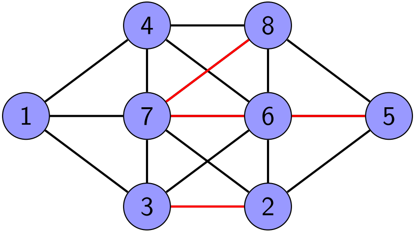
Note the violation cost: 4, as the number of violated binary constraints

What about the possibles moves?

- neighborhood: defined by swapping the values of two nodes,
- which guarantees that the hard constraint `allDifferent` remains satisfied.

Example

Assume that the initial solution is:

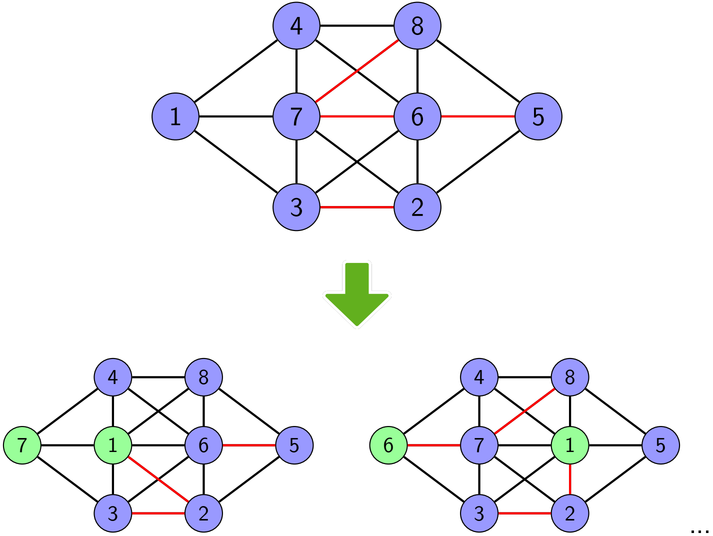


Note the violation cost: 4, as the number of violated binary constraints

What about the possibles moves?

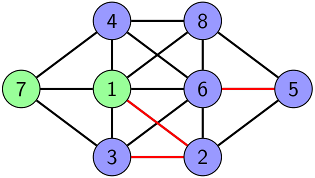
- neighborhood: defined by swapping the values of two nodes,
- which guarantees that the hard constraint `allDifferent` remains satisfied.

Neighborhood

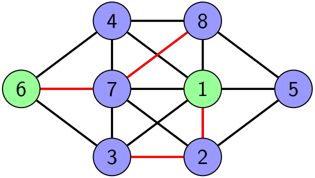


Neighborhood

For each neighbor, we can compute its violation cost:



⇒ violation=3



⇒ violation=4

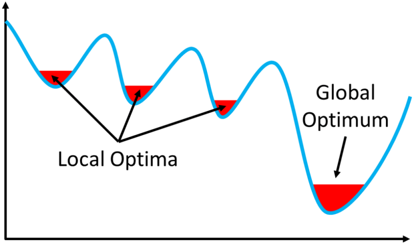
...

Local Search and Local Optima

In optimization: global optima are searched for.

Local search may lead to local optima.

- this is due to the local nature of LS
- sophisticated techniques can be used to escape local optima:
    - tabu search
    - simulated annealing
    - ...



Hybrization of complete and incomplete methods

Assets of CP

- CP is complete
- CP can intensify efficiently (propagate information)

Assets of LS

- LS is efficient at finding good solutions quickly
- LS can diversify efficiently

Why not having both?

Hybrization of complete and incomplete methods

Assets of CP

- CP is complete
- CP can intensify efficiently (propagate information)

Assets of LS

- LS is efficient at finding good solutions quickly
- LS can diversify efficiently

Why not having both?

Hybrization of complete and incomplete methods

Assets of CP

- CP is complete
- CP can intensify efficiently (propagate information)

Assets of LS

- LS is efficient at finding good solutions quickly
- LS can diversify efficiently

Why not having both?

Hybrization of CP and LS

Strength of CP

Speed of LS

Types of Hybrization

Sequential (one time) cooperation:

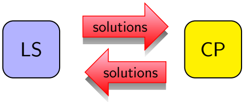
- CP then LS

  - CP computes an initial solution, respecting hard-constraints
  - LS starts with this solution

- LS then CP

  - LS computes an initial solution
  - CP starts with this solution (or its bound) to prove optimality

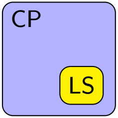
Parallel cooperation (CP and LS in parallel)



Types of Hybrization

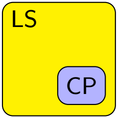
Master-Slave cooperation:

- Master CP, slave LS (Integrated LS within a CP search)



  - LS improves solutions found by CP:
    - LS run after each solution found to return improved bounds
    - CP remembers the best solution found

- Master LS, slave CP (Integrated CP inside a LS search)



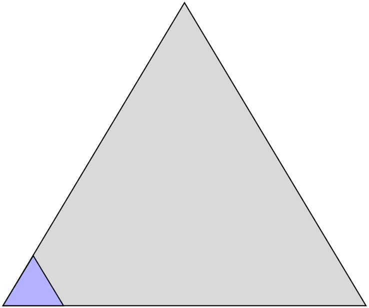
  - CP used to define and explore neighborhoods
  - Importance of Neighborhood size
    - Large neighborhoods: small path to optimum but costly to explore
    - Small neighborhood: long path to optimum but cheap to explore

Outline

Poor Diversification

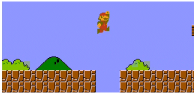
Weaknesses of CP for hard COPs:

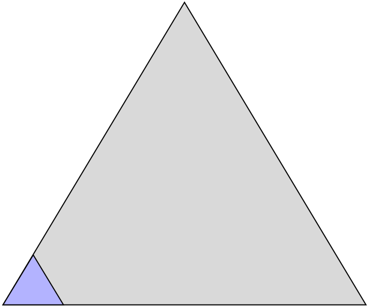
- very poor diversification
- you will never have a chance to reach the right part of the tree.



Principle of LNS

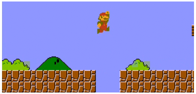
If stuck too long, jump in the search space.

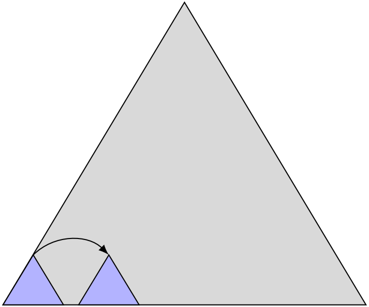




Principle of LNS

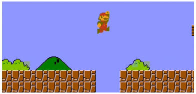
If stuck too long, jump in the search space.

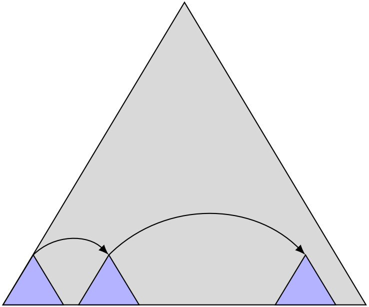




Principle of LNS

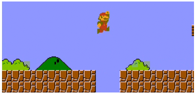
If stuck too long, jump in the search space.

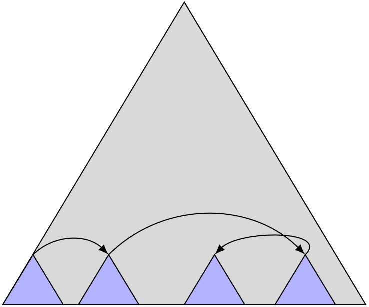




Principle of LNS

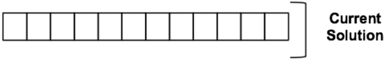
If stuck too long, jump in the search space.





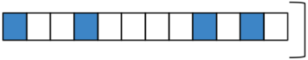
Principle of LNS

LNS has always a current solution



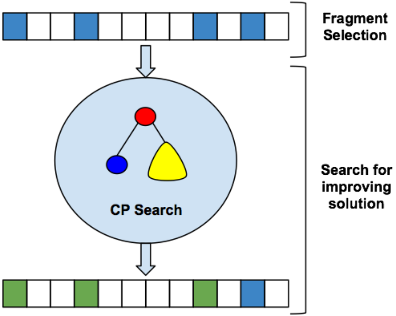
Principle of LNS

When jumping, a subset of variables is selected (the fragment)



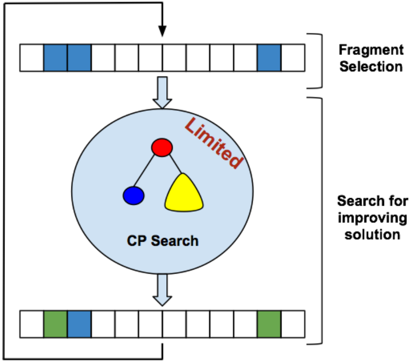
Principle of LNS

Only values in the fragment are allowed to be changed



Principle of LNS

This process is repeated until the time limit is reached



Principle of LNS

At each step of LNS:

- A portion of the variables is selected (called the fragment)
- Those variables are relaxed to their initial domain
- The others are frozen to their value in the current solution
- Improving solutions are searched with a limited CP search

Each time a solution is found: a new bound on the objective is added

Advantages of LNS

Advantages of Large Neighborhood Search:

- good diversification if fragments are well chosen
- intensification done by CP search
- neighborhoods large enough: no metaheuristic needed to avoid local optima
- no need to design complex feasible neighborhoods: CP is in charge of feasibility
- scalability of LS
- efficient exploration of neighborhoods with CP: propagation and heuristics

No Free Lunch!

LNS is a very powerful optimization technique but

LNS is efficient if its parameters are well defined.

No Free Lunch!

LNS is a very powerful optimization technique but

LNS is efficient if its parameters are well defined.

Parameters of LNS

The parameters of LNS are:

- the fragment selection heuristic: which variables do I relax ?
- the fragment size: how many variables do I relax ?
- the neighborhood exploration limit: how long (in term of time or backtracks) do I spend exploring the neighborhood ?

Exploration limit and fragment size:

- strongly linked parameters
- the fragment size determines the neighborhood size
- the exploration limit determines the maximal effort to explore the neighborhood

Parameters of LNS

The parameters of LNS are:

- the fragment selection heuristic: which variables do I relax ?
- the fragment size: how many variables do I relax ?
- the neighborhood exploration limit: how long (in term of time or backtracks) do I spend exploring the neighborhood ?

Exploration limit and fragment size:

- strongly linked parameters
- the fragment size determines the neighborhood size
- the exploration limit determines the maximal effort to explore the neighborhood

Designing a LNS Search

Rule of thumb: a good LNS should never be stopped only by the exploration limit.

If

- too many variables are relaxed or the exploration limit is too small

Then

- neighborhoods are too sparsely explored

And consequently

- LNS might discard promising neighborhood
- LNS might miss improving solutions

Designing a LNS Search

Rule of thumb: a good LNS should never be stopped only by the exploration limit.

If

- too many variables are relaxed or the exploration limit is too small

Then

- neighborhoods are too sparsely explored

And consequently

- LNS might discard promising neighborhood
- LNS might miss improving solutions

Fragment Selection

Fragment selection:

- can be random
- can be specific to a problem
- can be generic (while not random)

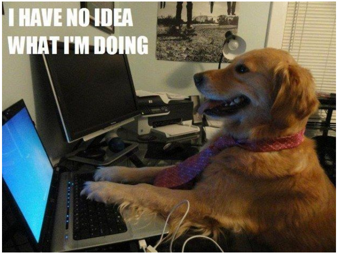
A good fragment should contain:

- important variables, which more likely allows improving the objective
- related variables, which more likely allows variables to be assigned differently

Random Fragment Selection

Random selection is surprisingly good:

- totally generic
- excellent diversification
- intensification from the CP search



Solution Saving

Recently, a very simple method, closely related to LNS, has been shown to be quite effective.

During backtrack search, Solution(-based phase) Saving selects in priority the value in the last found solution.