Constraint satisfaction problems

In which we see how treating states as more than just little black boxes leads to the invention of a range of powerful new search methods and a deeper understanding of problem structure and complexity.

Piotr Fulmański

Wydział Matematyki i Informatyki,
Uniwersytet Łódzki, Polska

April 8, 2010

Spis treści

1. Introduction to lecture
2. Idea – examples in R
3. Definition
4. Examples
5. Constraint propagation: inference in CSPs
6. Backtracking search for CSP
7. Local search for CSP
8. From constrained to unconstrained

From factored representation to constraint

- In lecture *Solving problems by searching* and *Beyond classical search* we explored the idea that problems can be solved by searching in a space of states.
- These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states.
- From the point of view of the search algorithm, however, each state is atomic – its internal structure is hidden.
- Now we use a factored representation.
- For each state we define a set of variables, each of which has a value.
- Next we set constraints on those variables.

From factored representation to constraint

- In lecture *Solving problems by searching* and *Beyond classical search* we explored the idea that problems can be solved by searching in a space of states.
- These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states.
- From the point of view of the search algorithm, however, each state is atomic – its internal structure is hidden.
- Now we use a factored representation.
- For each state we define a set of variables, each of which has a value.
- Next we set constraints on those variables.

From factored representation to constraint

- In lecture *Solving problems by searching* and *Beyond classical search* we explored the idea that problems can be solved by searching in a space of states.
- These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states.
- From the point of view of the search algorithm, however, each state is atomic – its internal structure is hidden.
- Now we use a factored representation.
- For each state we define a set of variables, each of which has a value.
- Next we set constraints on those variables.

From factored representation to constraint

- In lecture *Solving problems by searching* and *Beyond classical search* we explored the idea that problems can be solved by searching in a space of states.
- These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states.
- From the point of view of the search algorithm, however, each state is atomic – its internal structure is hidden.
- Now we use a factored representation.
- For each state we define a set of variables, each of which has a value.
- Next we set constraints on those variables.

From factored representation to constraint

- In lecture *Solving problems by searching* and *Beyond classical search* we explored the idea that problems can be solved by searching in a space of states.
- These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states.
- From the point of view of the search algorithm, however, each state is atomic – its internal structure is hidden.
- Now we use a factored representation.
- For each state we define a set of variables, each of which has a value.
- Next we set constraints on those variables.

From factored representation to constraint

- In lecture *Solving problems by searching* and *Beyond classical search* we explored the idea that problems can be solved by searching in a space of states.
- These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states.
- From the point of view of the search algorithm, however, each state is atomic – its internal structure is hidden.
- Now we use a factored representation.
- For each state we define a set of variables, each of which has a value.
- Next we set constraints on those variables.

From factored representation to constraint

- In lecture *Solving problems by searching* and *Beyond classical search* we explored the idea that problems can be solved by searching in a space of states.
- These states can be evaluated by domain-specific heuristics and tested to see whether they are goal states.
- From the point of view of the search algorithm, however, each state is atomic – its internal structure is hidden.
- Now we use a factored representation.
- For each state we define a set of variables, each of which has a value.
- Next we set constraints on those variables.

From factored representation to constraint

- A problem is solved when each variable has a value that satisfies all the constraints on the variable.
- A problem described this way is called a **constraint satisfaction problem** (CSP).
- Rather than problem-specific, CSP search algorithms use general-purpose heuristics.
- Taking advantage of the structure of states enable the solution of complex problems.
- The main idea is to eliminate large portions of the search space all at once by identifying variable-value combinations that violate the constraints.

From factored representation to constraint

- A problem is solved when each variable has a value that satisfies all the constraints on the variable.
- A problem described this way is called a **constraint satisfaction problem** (CSP).
- Rather than problem-specific, CSP search algorithms use general-purpose heuristics.
- Taking advantage of the structure of states enable the solution of complex problems.
- The main idea is to eliminate large portions of the search space all at once by identifying variable-value combinations that violate the constraints.

From factored representation to constraint

- A problem is solved when each variable has a value that satisfies all the constraints on the variable.
- A problem described this way is called a **constraint satisfaction problem** (CSP).
- Rather than problem-specific, CSP search algorithms use general-purpose heuristics.
- Taking advantage of the structure of states enable the solution of complex problems.
- The main idea is to eliminate large portions of the search space all at once by identifying variable-value combinations that violate the constraints.

From factored representation to constraint

- A problem is solved when each variable has a value that satisfies all the constraints on the variable.
- A problem described this way is called a **constraint satisfaction problem** (CSP).
- Rather than problem-specific, CSP search algorithms use general-purpose heuristics.
- Taking advantage of the structure of states enable the solution of complex problems.
- The main idea is to eliminate large portions of the search space all at once by identifying variable-value combinations that violate the constraints.

From factored representation to constraint

- A problem is solved when each variable has a value that satisfies all the constraints on the variable.
- A problem described this way is called a **constraint satisfaction problem** (CSP).
- Rather than problem-specific, CSP search algorithms use general-purpose heuristics.
- Taking advantage of the structure of states enable the solution of complex problems.
- The main idea is to eliminate large portions of the search space all at once by identifying variable-value combinations that violate the constraints.

From factored representation to constraint

- A problem is solved when each variable has a value that satisfies all the constraints on the variable.
- A problem described this way is called a **constraint satisfaction problem** (CSP).
- Rather than problem-specific, CSP search algorithms use general-purpose heuristics.
- Taking advantage of the structure of states enable the solution of complex problems.
- The main idea is to eliminate large portions of the search space all at once by identifying variable-value combinations that violate the constraints.

Example 1

Find minimum for function $f(x)$, $x \in R$

$$f(x) = x^2 + 2x + 4$$

with constraint

$$h(x) = x^2 - 4 = 0$$

Example 1

Find minimum for function $f(x)$, $x \in R$

$$f(x) = (x+3)(x+1)(x-1)(x-3)$$

with constraint

$$\begin{aligned} g_1(x) : -x^3 - 2x^2 &\leq 0 \\ g_2(x) : 2x^3 - 4x^2 &\leq 0 \end{aligned}$$

Definition

A constraint satisfaction problem consist of three components

- X – a set of variables, $\{x_1, \ldots, x_n\}$
- D – a set of domains, $\{D_1, \ldots, D_n\}$
- C – a set of constraints that specify allowable combinations of values.

Each domain D_i consist of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i. Each constraint C_i consist of a pair $\{V, R\}$, where V is a tuple of variables that participate in the constraint and R is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. A discrete domain can be infinite. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values.

Definition

A constraint satisfaction problem consist of three components

- X – a set of variables, $\{x_1, \ldots, x_n\}$
- D – a set of domains, $\{D_1, \ldots, D_n\}$
- C – a set of constraints that specify allowable combinations of values.

Each domain D_i consist of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i. Each constraint C_i consist of a pair $\{V, R\}$, where V is a tuple of variables that participate in the constraint and R is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. A discrete domain can be infinite. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values.

Definition

A constraint satisfaction problem consist of three components

- X – a set of variables, $\{x_1, \ldots, x_n\}$
- D – a set of domains, $\{D_1, \ldots, D_n\}$
- C – a set of constraints that specify allowable combinations of values.

Each domain D_i consist of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i. Each constraint C_i consist of a pair $\{V, R\}$, where V is a tuple of variables that participate in the constraint and R is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. A discrete domain can be infinite. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values.

Definition

A constraint satisfaction problem consist of three components

- X – a set of variables, $\{x_1, \ldots, x_n\}$
- D – a set of domains, $\{D_1, \ldots, D_n\}$
- C – a set of constraints that specify allowable combinations of values.

Each domain D_i consist of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i. Each constraint C_i consist of a pair $\{V, R\}$, where V is a tuple of variables that participate in the constraint and R is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. A discrete domain can be infinite. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values.

Definition

A constraint satisfaction problem consist of three components

- X – a set of variables, $\{x_1, \ldots, x_n\}$
- D – a set of domains, $\{D_1, \ldots, D_n\}$
- C – a set of constraints that specify allowable combinations of values.

Each domain D_i consist of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i. Each constraint C_i consist of a pair $\{V, R\}$, where V is a tuple of variables that participate in the constraint and R is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. A discrete domain can be infinite. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values.

Definition

A constraint satisfaction problem consist of three components

- X – a set of variables, $\{x_1, \ldots, x_n\}$
- D – a set of domains, $\{D_1, \ldots, D_n\}$
- C – a set of constraints that specify allowable combinations of values.

Each domain D_i consist of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i. Each constraint C_i consist of a pair $\{V, R\}$, where V is a tuple of variables that participate in the constraint and R is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. A discrete domain can be infinite. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values.

Definition

A constraint satisfaction problem consist of three components

- X – a set of variables, $\{x_1, \ldots, x_n\}$
- D – a set of domains, $\{D_1, \ldots, D_n\}$
- C – a set of constraints that specify allowable combinations of values.

Each domain D_i consist of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i. Each constraint C_i consist of a pair $\{V, R\}$, where V is a tuple of variables that participate in the constraint and R is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. A discrete domain can be infinite. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values.

Definition

A constraint satisfaction problem consist of three components

- X – a set of variables, $\{x_1, \ldots, x_n\}$
- D – a set of domains, $\{D_1, \ldots, D_n\}$
- C – a set of constraints that specify allowable combinations of values.

Each domain D_i consist of a set of allowable values, $\{v_1, \ldots, v_k\}$ for variable X_i. Each constraint C_i consist of a pair $\{V, R\}$, where V is a tuple of variables that participate in the constraint and R is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. A discrete domain can be infinite. With infinite domains, it is no longer possible to describe constraints by enumerating all allowed combinations of values.

Map coloring

Job-shop scheduling

- The whole job is composed of tasks.
- We can model each task as a variable.
- The value of each variable is the time that the task starts, expressed as an integer number of minutes.
- Constraints can assert that one task must occure before another.
- Constraints can also specify that a task takes a certain amount of time to complete.

Job-shop scheduling

- The whole job is composed of tasks.
- We can model each task as a variable.
- The value of each variable is the time that the task starts, expressed as an integer number of minutes.
- Constraints can assert that one task must occure before another.
- Constraints can also specify that a task takes a certain amount of time to complete.

Job-shop scheduling

- The whole job is composed of tasks.
- We can model each task as a variable.
- The value of each variable is the time that the task starts, expressed as an integer number of minutes.
- Constraints can assert that one task must occure before another.
- Constraints can also specify that a task takes a certain amount of time to complete.

Job-shop scheduling

- The whole job is composed of tasks.
- We can model each task as a variable.
- The value of each variable is the time that the task starts, expressed as an integer number of minutes.
- Constraints can assert that one task must occure before another.
- Constraints can also specify that a task takes a certain amount of time to complete.

Job-shop scheduling

- The whole job is composed of tasks.
- We can model each task as a variable.
- The value of each variable is the time that the task starts, expressed as an integer number of minutes.
- Constraints can assert that one task must occure before another.
- Constraints can also specify that a task takes a certain amount of time to complete.

Job-shop scheduling

- The whole job is composed of tasks.
- We can model each task as a variable.
- The value of each variable is the time that the task starts, expressed as an integer number of minutes.
- Constraints can assert that one task must occure before another.
- Constraints can also specify that a task takes a certain amount of time to complete.

Variables definition

$$X =$$

Domain definition

$$D =$$

Precedence constraints

$$C =$$

A cryptarithmetics problem

`TWO + TWO = FOUR`

Variables definition

$$X =$$

Domain definition

$$D =$$

A cryptarithmetics problem

Precedence constraints

$$C =$$

Constraint propagation: inference in CSPs

Constraint propagation: inference in CSPs

In regular state-space search, an algorithm can do only one thing: search.
In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of inference called constraint propagation: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.
Constraint propagation can be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.

Constraint propagation: inference in CSPs

Constraint propagation: inference in CSPs

In regular state-space search, an algorithm can do only one thing: search.
In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of inference called constraint propagation: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.
Constraint propagation can be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.

Constraint propagation: inference in CSPs

Constraint propagation: inference in CSPs

In regular state-space search, an algorithm can do only one thing: search. In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of inference called constraint propagation: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.
Constraint propagation can be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.

Constraint propagation: inference in CSPs

Constraint propagation: inference in CSPs

In regular state-space search, an algorithm can do only one thing: search. In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of inference called constraint propagation: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.

Constraint propagation can be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.

Constraint propagation: inference in CSPs

Constraint propagation: inference in CSPs

In regular state-space search, an algorithm can do only one thing: search.
In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of inference called constraint propagation: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.
Constraint propagation can be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.

Constraint propagation: inference in CSPs

In regular state-space search, an algorithm can do only one thing: search. In CSPs there is a choice: an algorithm can search (choose a new variable assignment from several possibilities) or do a specific type of inference called constraint propagation: using the constraints to reduce the number of legal values for a variable, which in turn can reduce the legal values for another variable, and so on.

Constraint propagation can be intertwined with search, or it may be done as a preprocessing step, before search starts. Sometimes this preprocessing can solve the whole problem, so no search is required at all.

Local consistency

The key idea is local consistency. If we treat each variable as a node in a graph and each binary constraint as an arc, then the process of enforcing local consistency in each part of the graph causes inconsistent values to be eliminated throughout the graph. There are different types of local consistency, which we now cover in turn.

Node consistency

A single variable is node consistent if all the values in the variable's domain satisfy the variable's unary constraints.

Example

Consider the constraint: $X_1, X_2 \in N$ and $X_1 \leq 10$ and $X_2 \leq 10$

Node consistency

A single variable is node consistent if all the values in the variable's domain satisfy the variable's unary constraints.

Example

Consider the constraint: $X_1, X_2 \in N$ and $X_1 \leq 10$ and $X_2 \leq 10$

Arc consistency

A variable is arc consistent if every value in the domain satisfies the variable's binary constraints.

Arc consistency

More formally, X_i is arc consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j)

Example

Consider the additional (to previous) constraint $X_2 = X_1^2$. We can write this constraint explicitly as

$$\{(X_1, X_2), \{(0,0), (1,1), (2,4), (3,9)\}\}$$

To make X_1 arc consistent with respect to X_2, we reduce X_1's domain to $\{0,1,2,3\}$. To make X_2 arc consistent with respect to X_1, we reduce X_2's domain to $\{0,1,4,9\}$.

Arc consistency

A variable is arc consistent if every value in the domain satisfies the variable's binary constraints.

Arc consistency

More formally, X_i is arc consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j)

Example

Consider the additional (to previous) constraint $X_2 = X_1^2$. We can write this constraint explicitly as

$$\{(X_1, X_2), \{(0,0), (1,1), (2,4), (3,9)\}\}$$

To make X_1 arc consistent with respect to X_2, we reduce X_1's domain to $\{0,1,2,3\}$. To make X_2 arc consistent with respect to X_1, we reduce X_2's domain to $\{0,1,4,9\}$.

Arc consistency

A variable is arc consistent if every value in the domain satisfies the variable's binary constraints.

Arc consistency

More formally, X_i is arc consistent with respect to another variable X_j if for every value in the current domain D_i there is some value in the domain D_j that satisfies the binary constraint on the arc (X_i, X_j)

Example

Consider the additional (to previous) constraint $X_2 = X_1^2$. We can write this constraint explicitly as

$$\{(X_1, X_2), \{(0,0), (1,1), (2,4), (3,9)\}\}$$

To make X_1 arc consistent with respect to X_2, we reduce X_1's domain to $\{0,1,2,3\}$. To make X_2 arc consistent with respect to X_1, we reduce X_2's domain to $\{0,1,4,9\}$.

The arc consistency algorithm (AC3)

The arc consistency algorithm (AC3)

After applying, either every arc is arc consistent, or some variable has an empty domain, indicating that the CSP cannot be solved.

The arc consistency algorithm (AC3)

```
input: a binary CSP with components (X,D,C)
local variables: queue, a queue of arcs, initially all the arcs in CSP

function AC3(csp) return false if an inconsistency is found and true otherwise
{
  while (queue is not empty)
  {
    (X_{i},X_{j}) := removeFirst(queue)
    if (revise(CSP,X_{i},X_{j})) then
    {
      if (size of D_{i} = 0) then
        return false
      for each X_{k} in X_{i}.neighbours - {X_{j}}
        queue.add( (X_{k},X_{i}) )
    }
  }
}

function revise(csp,X_{i},X_{j}) return true iff we revise the domain of X_{i}
{
  revise := false
  for each x in D_{i}
  {
    if (no value y in D_{j} allows (x,y) to satisfy the constraint
        between X_{i} and X_{j})
    {
      delete x from D_{i}
      revise := true
    }
  }
}
```

Path consistency

A two variable set is $\{X_i, X_j\}$ is path consistent with respect to a third variable X_m if, for every assignment $\{X_i = a, X_j = b\}$ consistent with the constraints on $\{X_i, X_j\}$, there is an assignment to X_m that satisfies the constraints on $\{X_i, X_m\}$ and $\{X_m, X_j\}$. This is called path consistency because one can think of it as looking at a path from X_i to X_j with X_m in the middle.

Sudoku example

Forward checking

AC3 algorithm can infer reductions in the domain of variables **before** we begin search. But inference can be even more powerful in the course of search: every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reduction on the neighboring variables.

Forward checking

One of the simplest form of inference is called forward checking. Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X. Because forward chcecking only does arc consistency inferences, there is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

Forward checking

AC3 algorithm can infer reductions in the domain of variables **before** we begin search. But inference can be even more powerful in the course of search: every time we make a choice of a value for a variable, we have a brand-new opportunity to infer new domain reduction on the neighboring variables.

Forward checking

One of the simplest form of inference is called forward checking. Whenever a variable X is assigned, the forward-checking process establishes arc consistency for it: for each unassigned variable Y that is connected to X by a constraint, delete from Y's domain any value that is inconsistent with the value chosen for X. Because forward chcecking only does arc consistency inferences, there is no reason to do forward checking if we have already done arc consistency as a preprocessing step.

Forward checking

Example

Backtracking search for CSP

Backtracking search for CSP

The term backtracking search is used for a depth-first search that chooses values for one variable at a time and backtracks when a variable has no legal values left to assign. Algorithm repeatedly chooses an unassigned variable, and then tries all values in the domain variable in turn, trying to find solution. If an inconsistency is detected then failure is returned, causing the previous call to try another value.

Backtracking search (BS)

```
function BS(csp) return a solution, or failure
  return backtrack({},csp)

function backtrack(assignment,csp) return solution, or failure
{
  if (assignment is complete) then
    return assignment
  var := selectUnassignedVariable(csp)
  for each value in orderDomainValues(var,assignment,csp)
  {
    if (value is consistent with assignment)
    {
      add {var = value} to assignment
      inferences := inference(csp,var,value)
      if inferences != failure
      {
        add inferences to assignment
        result := backtrack(assignment,csp)
        if (result != failure)
          return result
      }
    }
    remove {var = value} and inferences from assignment
  }
  return failure
}
```

Backtracking search for CSP

By varing the function selectUnassignedVariable and orderDomainValues, we can implement the general-purpose heuristics. The function `inference` can optionally be used to impose node, arc, path consistency, as desired.

Example

Knight problem

Local search for CSP

In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables – the min-conflicts heuristics.

Example

Min-conflicts for an 8-queens problem.

Min-conflicts (MC)

```
conflicts - function counts the number of constraints violated by a particular value,
            given the rest of the current assignment

function MC(csp,maxSteps) return a solution, or failure
{
  current := an initial complete assignment for csp
  for i := 1 to maxSteps
  {
    if (current is a solution for csp)
      return current
    var := a randomly chosen conflicted variable from csp.variables
    value := the value v for var that minimizes conflicts(var,v,current,csp)
    var := value in current
  }
  return failure
}
```

Penalty methods

One way to solve the inequality-constrained minimization problem

$$\begin{cases} \text{Minimize } f(x) \text{ subject to} \\ g_1(x) \leq 0, \ldots, g_m(x) \leq 0 \end{cases}$$

is to approximate this problem with an unconstrained minimization problem

$$\text{Minimize } F(x)$$

where the objective function $F(x)$ for the unconstrained problem is constructed from the objective function $f(x)$ and the constraints $h_i(x) \leq 0$, $i = 1, \ldots, m$ for the given constrained problem in such a way that

- $F(x)$ includes a penalty term which increases the value of $F(x)$ whenever a constraint $h_i(x) \leq 0$ (one or more) is violated. Larger violations results in larger increases.
- The unconstrained minimizer $x_{F_{\mathbf{min}}}$ of $F(x)$ is „near'' a constrained minimizer for the given constrained problem.

Penalty function

Using this approach, we hope that, as the size of the penalty term in $F(x)$ increases, the minimizer x_F^* of $F(x)$ will approach a point x^* that is feasible and a minimizer for the given constrained problem.

For a given constraint $g(x) \leq 0$, note that the function $g^+(x)$ defined by

$$g^+(x) = \begin{cases} 0 & \text{if} \quad g(x) \leq 0 \\ g(x) & \text{if} \quad g(x) > 0 \end{cases}$$

is zero for all x that satisfy the constraint and that it has a positive value whenever this constraint is violated

Penalty function

Using this approach, we hope that, as the size of the penalty term in $F(x)$ increases, the minimizer x_F^* of $F(x)$ will approach a point x^* that is feasible and a minimizer for the given constrained problem.
For a given constraint $g(x) \leq 0$, note that the function $g^+(x)$ defined by

$$g^+(x) = \begin{cases} 0 & \text{if} \quad g(x) \leq 0 \\ g(x) & \text{if} \quad g(x) > 0 \end{cases}$$

is zero for all x that satisfy the constraint and that it has a positive value whenever this constraint is violated

Penalty function

Using this approach, we hope that, as the size of the penalty term in $F(x)$ increases, the minimizer x_F^* of $F(x)$ will approach a point x^* that is feasible and a minimizer for the given constrained problem.
For a given constraint $g(x) \leq 0$, note that the function $g^+(x)$ defined by

$$g^+(x) = \begin{cases} 0 & \text{if} \quad g(x) \leq 0 \\ g(x) & \text{if} \quad g(x) > 0 \end{cases}$$

is zero for all x that satisfy the constraint and that it has a positive value whenever this constraint is violated

Penalty function

Using this approach, we hope that, as the size of the penalty term in $F(x)$ increases, the minimizer x_F^* of $F(x)$ will approach a point x^* that is feasible and a minimizer for the given constrained problem.
For a given constraint $g(x) \leq 0$, note that the function $g^+(x)$ defined by

$$g^+(x) = \begin{cases} 0 & \text{if} \quad g(x) \leq 0 \\ g(x) & \text{if} \quad g(x) > 0 \end{cases}$$

is zero for all x that satisfy the constraint and that it has a positive value whenever this constraint is violated

Penalty function

Moreover, large violations in the constraint $g(x) \leq 0$ result in large values for $g^+(x)$. Thus, $g^+(x)$ has the penalty features we want relative to the single constraint $g(x) \leq 0$.

Approximating unconstrained program

Approximating unconstrained program

If we now turn to the original constrained minimization problem

$$\begin{cases} \text{Minimize } f(x) \text{ subject to} \\ g_1(x) \leq 0, \ldots, g_m(x) \leq 0 \end{cases}$$

we see from the basic features of the function $g^+(x)$ that one reasonable definition for the objective function for an approximating unconstrained program is

$$Fk(x) = f(x) + k \sum_{i=1}^{m} g_i^+(x),$$

where k is a positive integer.

Approximating unconstrained program

Approximating unconstrained program

The role of the positive integer k is obvious: as k increases, so does the penalty associated with a given choice of x that violate one or more of the constraints $g_i(x) \leq 0$ for $i = 1, 2, \ldots, m$. For this reason, we call k the penalty parameter.

Approximating unconstrained program

Approximating unconstrained program

Our hope is that, for large k, the value of

$$k \sum_{i=1}^{m} g_i^{+}(x_k^*)$$

at a minimizer x_k^* for $F_k(x)$ should be small, x_k^* should be near the feasibility region for constrained minimization problem, and $F_k^{x_k^*}$ should be near a minimum for constrained minimization problem. This leads us to hope that there might be at least a subsequence of $\{x_k^*\}$ that converges to a minimizer x^* for constrained minimization problem.

Approximating unconstrained program

Approximating unconstrained program

Our hope is that, for large k, the value of

$$k \sum_{i=1}^{m} g_i^+(x_k^*)$$

at a minimizer x_k^* for $F_k(x)$ should be small, x_k^* should be near the feasibility region for constrained minimization problem, and $F_k^{x_k^*}$ should be near a minimum for constrained minimization problem. This leads us to hope that there might be at least a subsequence of $\{x_k^*\}$ that converges to a minimizer x^* for constrained minimization problem.

Approximating unconstrained program

Approximating unconstrained program

Our hope is that, for large k, the value of

$$k\sum_{i=1}^{m} g_i^+(x_k^*)$$

at a minimizer x_k^* for $F_k(x)$ should be small, x_k^* should be near the feasibility region for constrained minimization problem, and $F_k^{x_k^*}$ should be near a minimum for constrained minimization problem. This leads us to hope that there might be at least a subsequence of $\{x_k^*\}$ that converges to a minimizer x^* for constrained minimization problem.

Approximating unconstrained program

Approximating unconstrained program

Our hope is that, for large k, the value of

$$k \sum_{i=1}^{m} g_i^+(x_k^*)$$

at a minimizer x_k^* for $F_k(x)$ should be small, x_k^* should be near the feasibility region for constrained minimization problem, and $F_k^{x_k^*}$ should be near a minimum for constrained minimization problem. This leads us to hope that there might be at least a subsequence of $\{x_k^*\}$ that converges to a minimizer x^* for constrained minimization problem.

Approximating unconstrained program

Approximating unconstrained program

Our hope is that, for large k, the value of

$$k\sum_{i=1}^{m} g_i^+(x_k^*)$$

at a minimizer x_k^* for $F_k(x)$ should be small, x_k^* should be near the feasibility region for constrained minimization problem, and $F_k^{x_k^*}$ should be near a minimum for constrained minimization problem. This leads us to hope that there might be at least a subsequence of $\{x_k^*\}$ that converges to a minimizer x^* for constrained minimization problem.

Approximating unconstrained program

Example

Consider the program

$$\begin{cases} \text{Minimize } f(x) = x^2 \text{ subject to} \\ g(x) = 1 - x \leq 0 \quad x \in R \end{cases}$$