Principles of Constraint Programming

Krzysztof R. Apt

Chapter 8
Search

Objectives

- Introduce **search trees**,
- Discuss various types of **labeling trees**, in particular trees for
  - forward checking,
  - partial look ahead, and
  - maintaining arc consistency (MAC).
- Discuss various **search algorithms** for labeling trees.
- Discuss **search algorithms** for Constrained Optimization Problems:
- Introduce various **heuristics** for search algorithms.

Useful Slogan

Search Algorithm =
Search Tree + Traversal Algorithm.

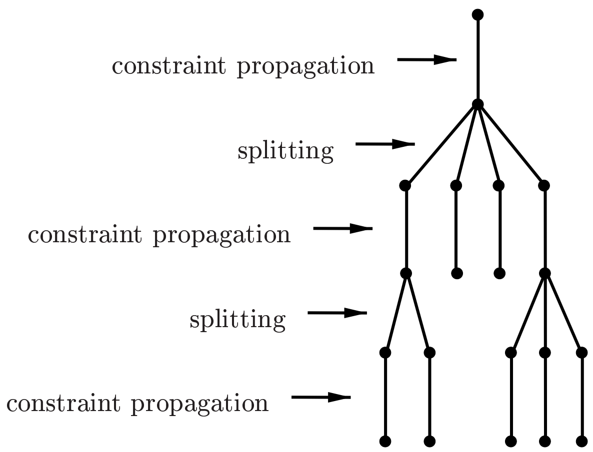
Search Trees

Consider a CSP $\mathcal{P}$ with a sequence of variables X.

Search tree for $\mathcal{P}$:
a finite tree such that

- its nodes are CSP's,
- its root is $\mathcal{P}$,
- the nodes at an even level have exactly one direct descendant,
- if $\mathcal{P}_1, \ldots, \mathcal{P}_m$ are direct descendants of $\mathcal{P}_0$, then the union of $\mathcal{P}_1, \ldots, \mathcal{P}_m$ is equivalent w.r.t. X to $\mathcal{P}_0$.

Intuition



Labeling Trees

Specific search trees for finite CSP's.

- splitting consists of **labeling** of a domain of a variable.
- constraint propagation consists of a domain reduction method.

Complete Labeling Trees

Constraint propagation **absent**.

Given:
– a CSP $\mathcal{P}$ with non-empty domains,
– $x_1, \ldots, x_n$ the sequence of its variables linearly ordered by $\prec$.

Complete labeling tree associated with $\mathcal{P}$ and $\prec$: a tree such that

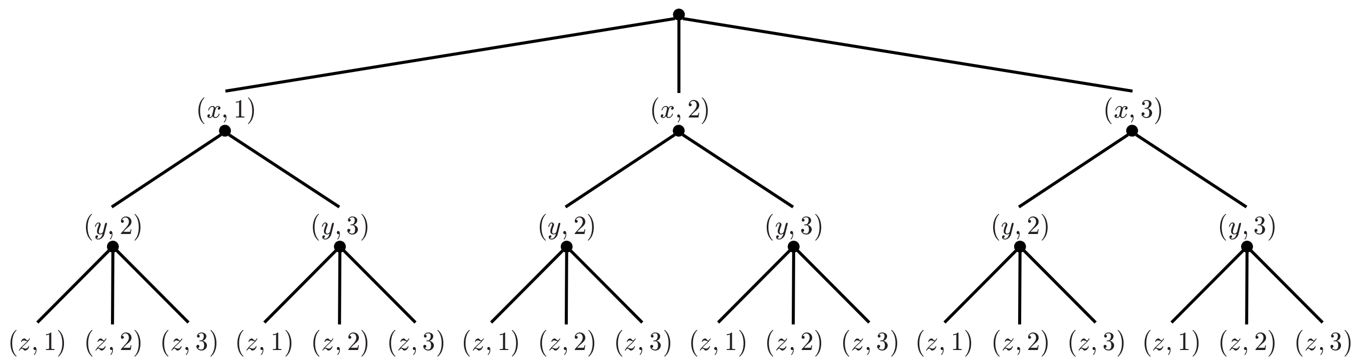
- the direct descendants of the root are of the form (x_1, d),
- the direct descendants of a node (x_j, d), where $j \in [1..n-1]$, are of the form (x_{j+1}, e),
- its branches determine all the instantiations with the domain $\{x_1, \ldots, x_n\}$.

Examples

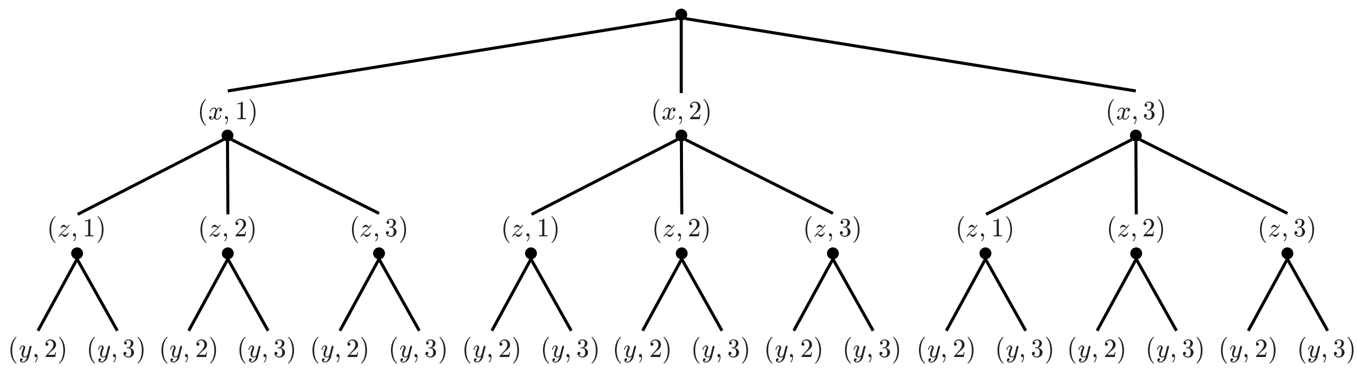
Consider

$\langle x < y, y < z \,;\, x \in \{1,2,3\}, y \in \{2,3\}, z \in \{1,2,3\}\rangle.$

1. with the ordering $x \prec y \prec z$.



2. with the ordering $x \prec z \prec y$.



Sizes of Complete Labeling Trees

Given:
– a CSP with non-empty domains,
– $x_1, \ldots, x_n$ the sequence of its variables linearly ordered by $\prec$.
– $D_1, \ldots, D_n$ the corresponding variable domains. Then

- The number of nodes in the complete labeling tree associated with $\prec$ is

  $$1 + \Sigma_{i=1}^{n}(\Pi_{j=1}^{i}|D_j|),$$

  $|A|$: the cardinality of the set A.

- The complete labeling tree has the least number of nodes if the variables are ordered by their domain sizes in the **increasing** order.

Examples

1.: Tree in **1.**

The cardinalities of the domains: 3, 2, 3.

The tree has $1 + 3 + 3 \cdot 2 + 3 \cdot 2 \cdot 3$, i.e., 28 nodes.

2.: Tree in **2.**

The cardinalities of the domains: 3, 3, 2.

The tree has $1 + 3 + 3 \cdot 3 + 3 \cdot 3 \cdot 2$, i.e., 31 nodes.

Both trees have the same number of leaves: 18.

Reduced Labeling Trees

An instantiation I is **along the ordering** $x_1, \ldots, x_n$ if its domain is $\{x_1, \ldots, x_j\}$ for some $j \in [1..n]$.

Given:
– a CSP $\mathcal{P}$ with non-empty domains,
– $x_1, \ldots, x_n$ the sequence of its variables linearly ordered by $\prec$.

Reduced labeling tree associated with $\mathcal{P}$ and $\prec$: a finite tree such that

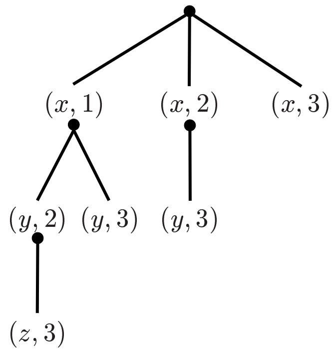
- the direct descendants of the root are of the form (x_1, d),
- the direct descendants of a node (x_j, d), where $j \in [1..n-1]$, are of the form (x_{j+1}, e),
- its branches determine all consistent instantiations along the ordering $x_1, \ldots, x_n$.

Examples

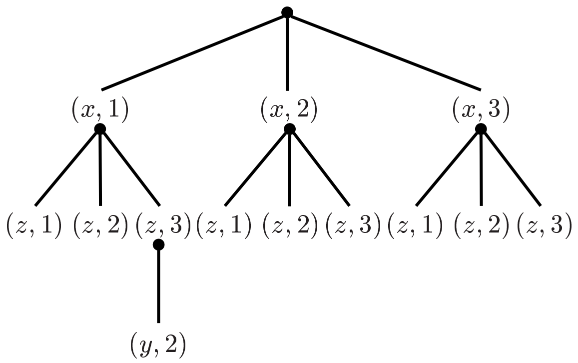
Consider

$\langle x < y, y < z\,;\, x \in \{1,2,3\}, y \in \{2,3\}, z \in \{1,2,3\}\rangle.$

1. with the ordering $x \prec y \prec z$.



2. with the ordering $x \prec z \prec y$.



Reduced labeling trees can have different number of nodes and different number of leaves.

Labeling Trees with Constraint Propagation

Given:

$$\mathcal{P} := \langle \mathcal{C} \; ; \; x_1 \in D_1, \ldots, x_n \in D_n \rangle.$$

- Assume fixed form of **constraint propagation** *prop(i)* in the form of a domain reduction, where $i \in [0..n-1]$.
- i determines the sequence $x_{\mathbf{i+1}}, \ldots, x_n$ of the variables to the domains of which *prop(i)* is applied.
- Given current variable domains $E_1, \ldots, E_n$, constraint propagation *prop(i)* transforms only $E_{\mathbf{i+1}}, \ldots, E_n$.
- *prop(i)* depends on the original constraints $\mathcal{C}$ of $\mathcal{P}$ and on the domains $E_1, \ldots, E_i$.

prop Labeling Trees

***prop* labeling tree associated with $\mathcal{P}$:**
a tree such that

- its nodes are sequences of the domain expressions $x_1 \in E_1, \ldots, x_n \in E_n$,
- its root is $x_1 \in D_1, x_2 \in D_2, \ldots, x_n \in D_n$,
- each node at an **even** level $2i$ with $i \in [0..n]$ is of the form

  $$x_1 \in \{d_1\}, \ldots, x_i \in \{d_i\}, x_{i+1} \in E_{i+1}, \ldots, x_n \in E_n.$$

  If $i = n$, this node is a leaf. Otherwise, it has exactly one direct descendant, obtained using *prop(i)*:

  $$x_1 \in \{d_1\}, \ldots, x_i \in \{d_i\}, x_{i+1} \in E'_{i+1}, \ldots, x_n \in E'_n$$

  where $E'_j \subseteq E_j$ for $j \in [i+1..n]$
- each node at an **odd** level $2i+1$ with $i \in [0..n-1]$ is of the form

  $$x_1 \in \{d_1\}, \ldots, x_i \in \{d_i\}, x_{i+1} \in E_{i+1}, \ldots, x_n \in E_n.$$

  If $E_j = \emptyset$ for some $j \in [i+1..n]$, this node is a leaf. Otherwise it has direct descendants of the form

  $$x_1 \in \{d_1\}, \ldots, x_i \in \{d_i\}, x_{i+1} \in \{d\},$$
  $$x_{i+2} \in E_{i+2}, \ldots, x_n \in E_n,$$

  for all $d \in E_{i+1}$ such that the instantiation $\{(x_1, d_1), \ldots, (x_i, d_i), (x_{i+1}, d)\}$ is consistent.

Intuition

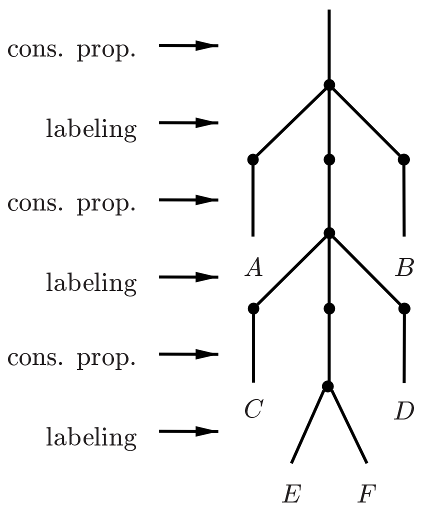
Given: node $x_1 \in E_1, \ldots, x_n \in E_n$ at level $2i-1$ or $2i$,

- if $i \in [2..n-1]$, we call $x_1, \ldots x_{i-1}$ its **past variables**,
- if $i \in [1..n]$, we call x_i its **current variable**, and
- if $i \in [0..n-1]$, we call $x_{i+1}, \ldots x_n$ its **future variables**.

prop(i) affects only the domains of the future variables.

Example of a *prop* labeling tree

Consider a CSP with three variables, x_1, x_2, x_3.

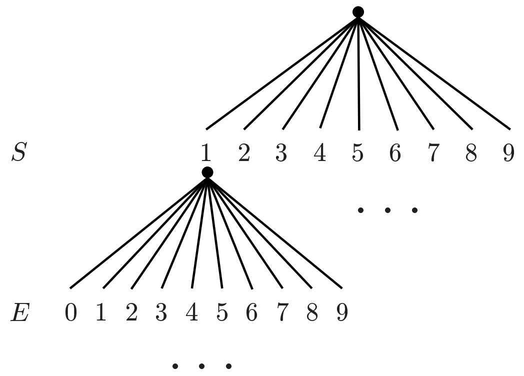


levels	past variables	current variable	future variables
0			
1		x_1	x_2, x_3
2		x_1	x_2, x_3
3	x_1	x_2	x_3
4	x_1	x_2	x_3
5	x_1, x_2	x_3	
6	x_1, x_2	x_3	

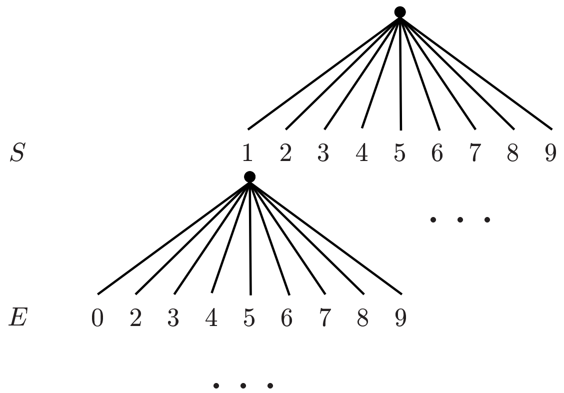
A, B, C and D are **failed** nodes.
E and F are **success** nodes.

Example: *SEND + MORE = MONEY*

Complete Labeling Tree:



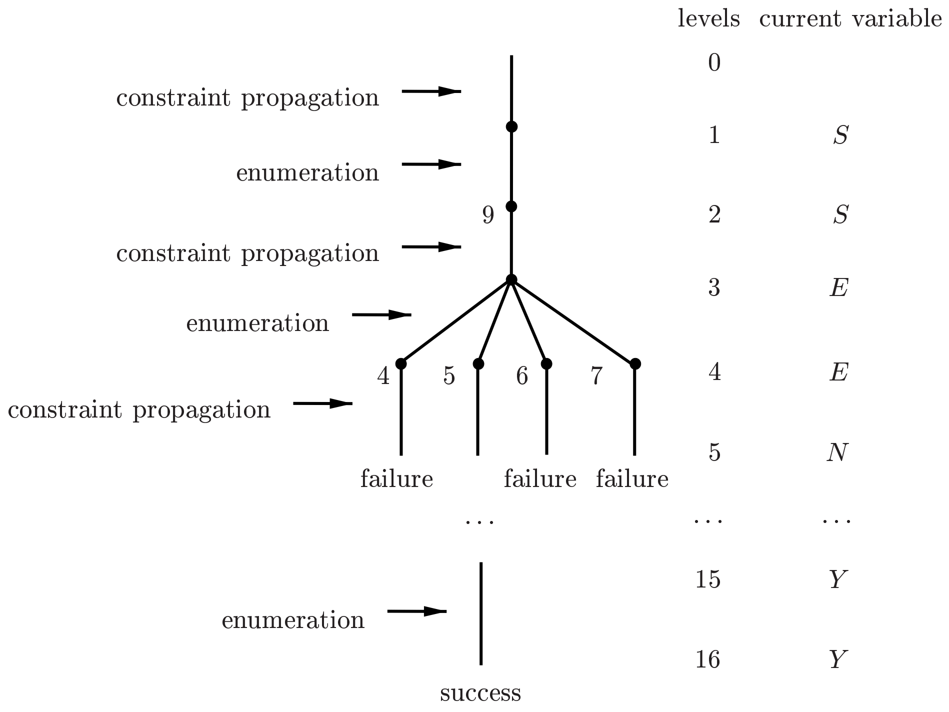
Reduced Labeling Tree:



SEND + MORE = MONEY, ctd

Use as ***prop*** the domain reduction rules for linear constraints over integer intervals from Chapter 6.

prop Labeling Tree:



Sizes of the Generated Trees

For $SEND + MORE = MONEY$:

- Complete labeling tree.

  Total number of leaves: $9^2 \cdot 10^6 = 81000000$.

- Reduced labeling tree.

  Total number of leaves:

  $10 \cdot 9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4 - 2 \cdot (9 \cdot 8 \cdot 7 \cdot 6 \cdot 5 \cdot 4) = 483840$.

  Gain: 99.4% with respect to the complete labeling tree.

- *prop* labeling tree.

  Total number of leaves: 4.

Instances of *prop* Labeling Trees

- **forward checking**,
- **partial look ahead**,
- **maintaining arc consistency (MAC)** (aka full look ahead).

Forward Checking Search Tree

Recall from the definition of *prop* labeling trees:

- each node at an **even** level $2i$ with $i \in [0..n]$ is of the form

  $x_1 \in \{d_1\}, \ldots, x_i \in \{d_i\}, x_{i+1} \in E_{i+1}, \ldots, x_n \in E_n.$

  If $i = n$, this node is a leaf. Otherwise, it has exactly one direct descendant, obtained using *prop(i)*:

  $x_1 \in \{d_1\}, \ldots, x_i \in \{d_i\}, x_{i+1} \in E'_{i+1}, \ldots, x_n \in E'_n$

  where $E'_j \subseteq E_j$ for $j \in [i+1..n]$.

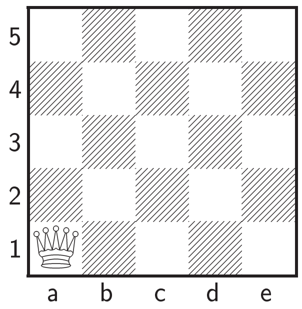
Define

$$E'_j := \{e \in E_j \mid \{(x_1, d_1), \ldots, (x_i, d_i), (x_j, e)\} \text{ is consistent}\}$$

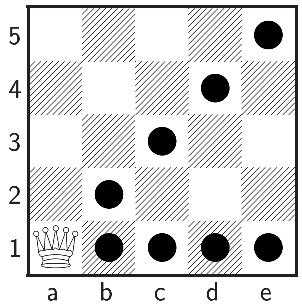
Example: 5 Queens Problem

Take the **standardized** CSP corresponding to 5 Queens Problem.
Interpretation: the variables x_1, x_2, x_3, x_4, x_5 correspond to the columns `a,b,c,d,e`.

First queen placed at `a1`:



Effect of **forward checking**:

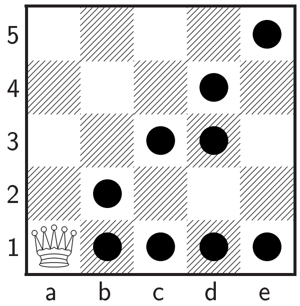


Partial Look Ahead Search Tree

- Impose forward checking.
- Impose **directional arc consistency**, e.g. using the `DARC` algorithm.

Example: 5 Queens Problem.

Effect of **partial look ahead**:

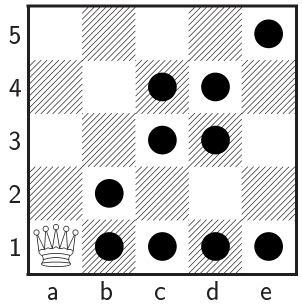


MAC Search Tree

- Impose forward checking.
- Impose **arc consistency**, e.g. using the `ARC` algorithm.

Example: 5 Queens Problem.

Effect of **MAC**:



Search Algorithms for Labeling Trees

- Backtrack-free search
- Backtrack-free search with constraint propagation,
- Backtrack search,
- Backtrack search with constraint propagation:
  - forward checking,
  - partial look ahead, and
  - MAC.

Search algorithms for Constrained Optimization Problems:

- Branch and Bound search,
- Branch and Bound with Constraint Propagation search.

Backtrack-free Search

`cons(inst,j,d)` $\equiv$ "the instantiation $\{(x_1, \texttt{inst}[1]), \ldots, (x_{j-1}, \texttt{inst}[\texttt{j}-1]), (x_j, \texttt{d})\}$ is consistent"

```
MODULE backtrack_free;
TYPE domains =  ARRAY [1..n] OF domain;
     instantiation = ARRAY [1..n] OF elements;
VAR inst: instantiation;
PROCEDURE backtrack_free(j: INTEGER; D: domains;
                         VAR success: BOOLEAN);
BEGIN
  WHILE D[j] <> {} AND NOT success DO
    choose d from D[j];
    D[j] := D[j] - {d};
    IF cons(inst,j,d) THEN
      inst[j] := d;
      success := (j=n);
      IF NOT success THEN
        j := j+1
      END
    END
  END
END backtrack_free;
BEGIN
  success := FALSE;
  backtrack_free(1,D,success)
END backtrack_free;
```

Backtrack-free Search with Constraint Propagation

```
MODULE backtrack_free_prop;
TYPE domains =  ARRAY [1..n] OF domain;
     instantiation = ARRAY [1..n] OF elements;
VAR inst: instantiation;
    failure: BOOLEAN;
PROCEDURE backtrack_free_prop(j: INTEGER; D: domains;
                              VAR success: BOOLEAN);
BEGIN
  WHILE D[j] <> {} AND NOT success DO
    choose d from D[j];
    D[j] := D[j] - {d};
    IF cons(inst,j,d) THEN
      inst[j] := d;
      success := (j=n);
      IF NOT success THEN
        prop(j,D,failure);
        IF NOT failure THEN
          j := j+1
        END
      END
    END
  END
END backtrack_free_prop;
BEGIN
  success := FALSE;
  prop(0,D,failure);
  IF NOT failure THEN backtrack_free_prop(1,D,success) END
END backtrack_free_prop;
```

Backtracking

```
MODULE backtrack;
TYPE domains =  ARRAY [1..n] OF domain;
     instantiation = ARRAY [1..n] OF elements;
VAR inst: instantiation;
PROCEDURE backtrack(j: INTEGER; D: domains;
                    VAR success: BOOLEAN);
BEGIN
  WHILE D[j] <> {} AND NOT success DO
    choose d from D[j];
    D[j] := D[j] - {d};
    IF cons(inst,j,d) THEN
      inst[j] := d;
      success := (j=n);
      IF NOT success THEN
        backtrack(j+1,D,success)
      END
    END
  END
END backtrack;
BEGIN
  success := FALSE;
  backtrack(1,D,success)
END backtrack;
```

Backtracking with Constraint Propagation

```
MODULE backtrack_prop;
TYPE domains =  ARRAY [1..n] OF domain;
     instantiation = ARRAY [1..n] OF elements;
VAR inst: instantiation;
    failure: BOOLEAN;
PROCEDURE backtrack_prop(j: INTEGER; D: domains;
                         VAR success: BOOLEAN);
BEGIN
  WHILE D[j] <> {} AND NOT success DO
    choose d from D[j];
    D[j] := D[j] - {d};
    IF cons(inst,j,d) THEN
      inst[j] := d;
      success := (j=n);
      IF NOT success THEN
        prop(j,D,failure);
        IF NOT failure THEN
          backtrack_prop(j+1,D,success)
        END
      END
    END
  END
END backtrack_prop;
BEGIN
  success := FALSE;
  prop(0,D,failure);
  IF NOT failure THEN backtrack_prop(1,D,success) END
END backtrack_prop;
```

Forward Checking

```
PROCEDURE revise(j,k: INTEGER; VAR D: domains);
BEGIN
  D[k] := {d ∈ D[k] | {(x_1,inst[1]),...,(x_j,inst[j]),(x_k,d)}
                        is a consistent instantiation}
END revise;
```

```
PROCEDURE prop(j: INTEGER; VAR D: domains;
               VAR failure: BOOLEAN);
VAR k: INTEGER;
BEGIN
  failure := FALSE;
  k := j+1;
  WHILE k <> n+1 AND NOT failure DO
    revise(j,k,D);
    failure := (D[k] = {});
    k := k+1
  END
END prop;
```

Partial Look Ahead

```
PROCEDURE prop(j: INTEGER; VAR D: domains;
               VAR failure: BOOLEAN);
VAR k: INTEGER;
BEGIN
  failure := FALSE;
  k := j+1;
  WHILE k <> n+1 AND NOT failure DO
    revise(j,k,D);
    failure := (D[k] = {});
    k := k+1
  END;
  IF NOT failure THEN
    darc(j+1,D,failure)
  END
END prop;
```

MAC (Full Look Ahead)

```
PROCEDURE prop(j: INTEGER; VAR D: domains;
               VAR failure: BOOLEAN);
...
  IF NOT failure THEN
    arc(j+1,D,failure)
  END
END prop;
```

Searching for All Solutions

```
MODULE backtrack_all;
TYPE domains =  ARRAY [1..n] OF domain;
     instantiation = ARRAY [1..n] OF elements;
VAR inst: instantiation;
PROCEDURE backtrack_all(j: INTEGER; D: domains);
BEGIN
  WHILE D[j] <> {} DO
    choose d from D[j];
    D[j] := D[j] - {d};
    IF cons(inst,j,d) THEN
      inst[j] := d;
      IF j=n THEN
        PRINT(inst)
      ELSE
        backtrack_all(j+1,D)
      END
    END
  END
END backtrack_all;
BEGIN
  backtrack_all(1,D)
END backtrack_all;
```

Finite Constrained Optimization Problems

- $\mathcal{P} := \langle \mathcal{C} \; ; \; x_1 \in D_1, \ldots, x_n \in D_n \rangle$,
- $obj : Sol \to \mathcal{R}$

  from the set Sol of all solutions to $\mathcal{P}$ to $\mathcal{R}$.
- Heuristic function

$$h : \mathcal{P}(D_1) \times \ldots \times \mathcal{P}(D_n) \to \mathcal{R}$$

Monotonicity If $\bar{E}_1 \subseteq \bar{E}_2$, then $h(\bar{E}_1) \leq h(\bar{E}_2)$,

Bound $obj(d_1, \ldots, d_n) \leq h(\{d_1\}, \ldots, \{d_n\})$.

In Programs

```
PROCEDURE obj(inst: instantiation): REAL;

PROCEDURE h(inst: instantiation; j: INTEGER;
            D: domains): REAL;
```

`h(inst,j,D)` returns the value of h on

$$(\{\texttt{inst}[1]\}, \ldots, \{\texttt{inst}[j]\}, \texttt{D}[j+1], \ldots, \texttt{D}[n]).$$

Branch and Bound

```
MODULE branch_and_bound;
TYPE domains =  ARRAY [1..n] OF domain;
     instantiation = ARRAY [1..n] OF elements;
VAR inst: instantiation;
PROCEDURE branch_and_bound(j: INTEGER; D: domains;
    VAR solution: instantiation; VAR bound: REAL);
BEGIN
  WHILE D[j] <> {} DO
    choose d from D[j];
    D[j] := D[j] - {d};
    IF cons(inst,j,d) THEN
      inst[j] := d;
      IF j=n THEN
        IF obj(inst) > bound THEN
          bound := obj(inst); solution := inst
        END
      ELSE
        IF h(inst,j,D) > bound THEN
          branch_and_bound(j+1,D,solution,bound)
        END
      END
    END
  END
END branch_and_bound;
BEGIN
  solution := NIL; bound := -infinity;
  branch_and_bound(1,D,solution,bound)
  END
END branch_and_bound;
```

Branch and Bound with Constraint Propagation

```
MODULE branch_and_bound_prop;
TYPE domains =  ARRAY [1..n] OF domain;
     instantiation = ARRAY [1..n] OF elements;
VAR inst: instantiation;
    failure: BOOLEAN;
PROCEDURE branch_and_bound_prop(j: INTEGER; D: domains;
         VAR solution: instantiation; VAR bound: REAL);
BEGIN
  WHILE D[j] <> {} DO
    choose d from D[j];
    D[j] := D[j] - {d};
    IF cons(inst,j,d) THEN
      inst[j] := d;
      IF j=n THEN
        IF obj(inst) > bound THEN
          bound := obj(inst); solution := inst
        END
      ELSE
        prop(j,D,failure);
        IF NOT failure THEN
          IF h(inst,j,D) > bound THEN
            branch_and_bound_prop(j+1,D,solution,bound)
          END
        END
      END
    END
  END
END branch_and_bound_prop;
```

```
BEGIN
  solution := NIL;
  bound := -infinity;
  prop(0,D,failure);
  IF NOT failure THEN
    branch_and_bound_prop(1,D,solution,bound)
  END
END branch_and_bound_prop;
```

Heuristics for Search Algorithms

Variable Selection

- Select a variable with the smallest domain.
- Select a *most constrained* variable,
- (For numeric domains)

  Select a variable with the smallest difference between its domain bounds.

Value Selection

- select a value for which the heuristic function yields the highest outcome.
- select the smallest value,
- select the largest value,
- select the middle value.

Objectives

- Introduce **search trees**.
- Discuss various types of **labeling trees**, in particular trees for
  - forward checking,
  - partial look ahead, and
  - maintaining arc consistency (MAC).
- Discuss various **search algorithms** for labeling trees.
- Discuss **search algorithms** for Constrained Optimization Problems:
- Introduce various **heuristics** for search algorithms.