

Lecture 7

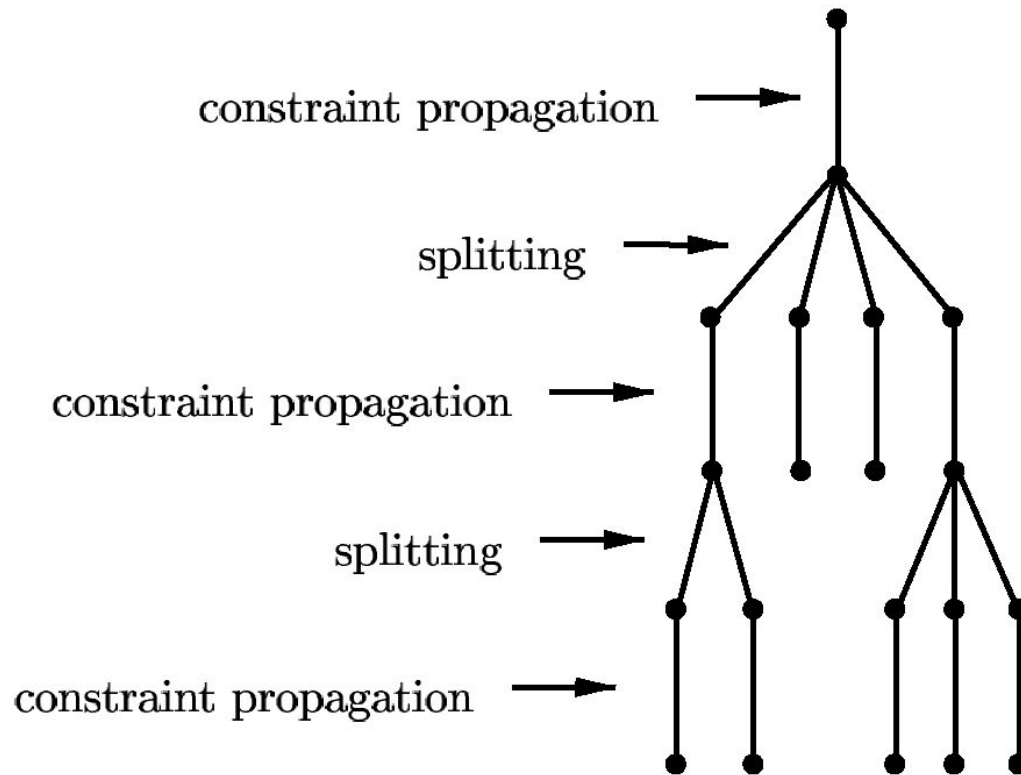
Search

Outline

- Introduce search trees
- Discuss various types of labeling trees, in particular trees for
 - forward checking
 - partial look ahead
 - 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 $\mathcal{P}_1, \dots, \mathcal{P}_m$ are direct descendants of \mathcal{P}_0 , then the union of $\mathcal{P}_1, \dots, \mathcal{P}_m$ is equivalent w.r.t. X to \mathcal{P}_0

Labeling Trees

Specific search trees for finite CSP's

- Splitting consists of labeling of the 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, \dots, x_n the sequence of its variables linearly ordered by $<$

Complete labeling tree associated with \mathcal{P} and $<$:

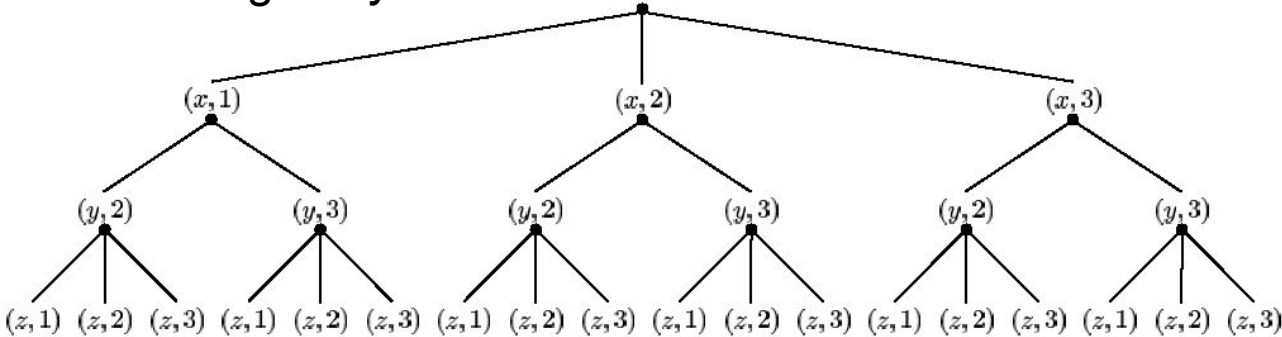
- the direct descendants of the root are of the form (x_1, d)
- the direct descendants of a node (x_j, d) , where $j \in [1..n - 1]$, are of the form (x_{j+1}, e)
- its branches determine all the instantiations with the domain $\{x_1, \dots, x_n\}$

Examples

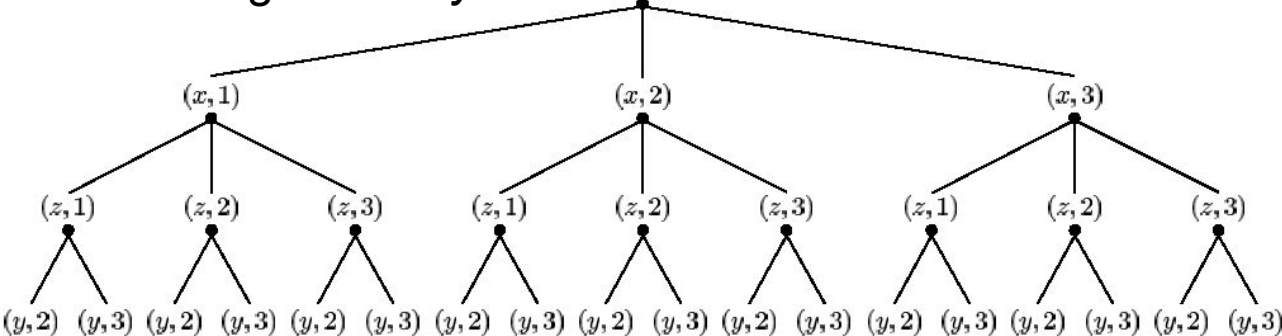
Consider

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

1. with the ordering $x < y < z$



2. with the ordering $x < z < y$



Sizes of Complete Labeling Trees

Given:

- a CSP with non-empty domains
- x_1, \dots, x_n the sequence of its variables linearly ordered by $<$
- D_1, \dots, D_n the corresponding variable domains
- The number of nodes in the complete labeling tree associated with $<$ is

$$1 + \sum_{i=1}^n \left(\prod_{j=1}^i |D_j| \right)$$

$|A|$: the cardinality of set A

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

Examples

Tree in 1. (cf. Slide 7):

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

Tree in 2. (cf. Slide 7):

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, \dots, x_n if its domain is $\{x_1, \dots, x_j\}$ for some $j \in [1..n]$.

Given:

- a CSP \mathcal{P} with non-empty domains
- x_1, \dots, x_n the sequence of its variables linearly ordered by $<$

Reduced labeling tree associated with \mathcal{P} and $<$:

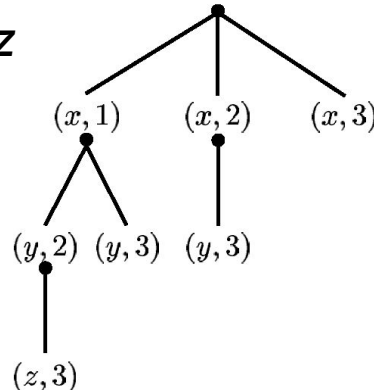
- the direct descendants of the root are of the form (x_1, d)
- the direct descendants of a node (x_j, d) , where $j \in [1..n - 1]$, are of the form (x_{j+1}, e)
- its branches determine all consistent instantiations along the ordering x_1, \dots, x_n

Examples

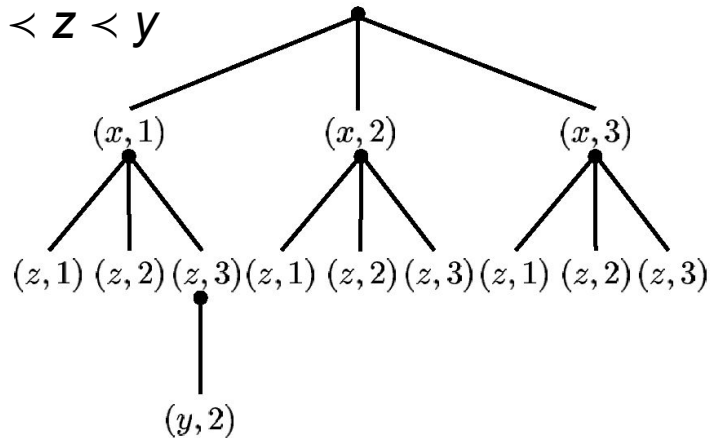
Consider

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

1. with the ordering $x < y < z$



2. with the ordering $x < z < y$



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

Labeling Trees with Constraint Propagation

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

- 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_{i+1}, \dots, x_n of the variables to whose domains $prop(i)$ is applied
- Given current variable domains E_1, \dots, E_n , constraint propagation $prop(i)$ transforms only E_{i+1}, \dots, E_n
- $prop(i)$ depends on the original constraints C of \mathcal{P} and on the domains E_1, \dots, E_i

prop Labeling Trees

prop labeling tree associated with \mathcal{P} :

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

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

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

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

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

prop Labeling Trees, ctd

- each node at an **odd** level $2i + 1$ with $i \in [0..n - 1]$ is of the form

$$x_1 \in \{d_1\}, \dots, x_j \in \{d_j\}, x_{i+1} \in E_{i+1}, \dots, x_n \in E_n$$

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

$$x_1 \in \{d_1\}, \dots, x_j \in \{d_j\}, x_{i+1} \in \{d\}, x_{i+2} \in E_{i+2}, \dots, x_n \in E_n$$

for all $d \in E_{i+1}$ such that the instantiation $\{(x_1, d_1), \dots, (x_j, d_j), (x_{i+1}, d)\}$ is consistent

Intuition

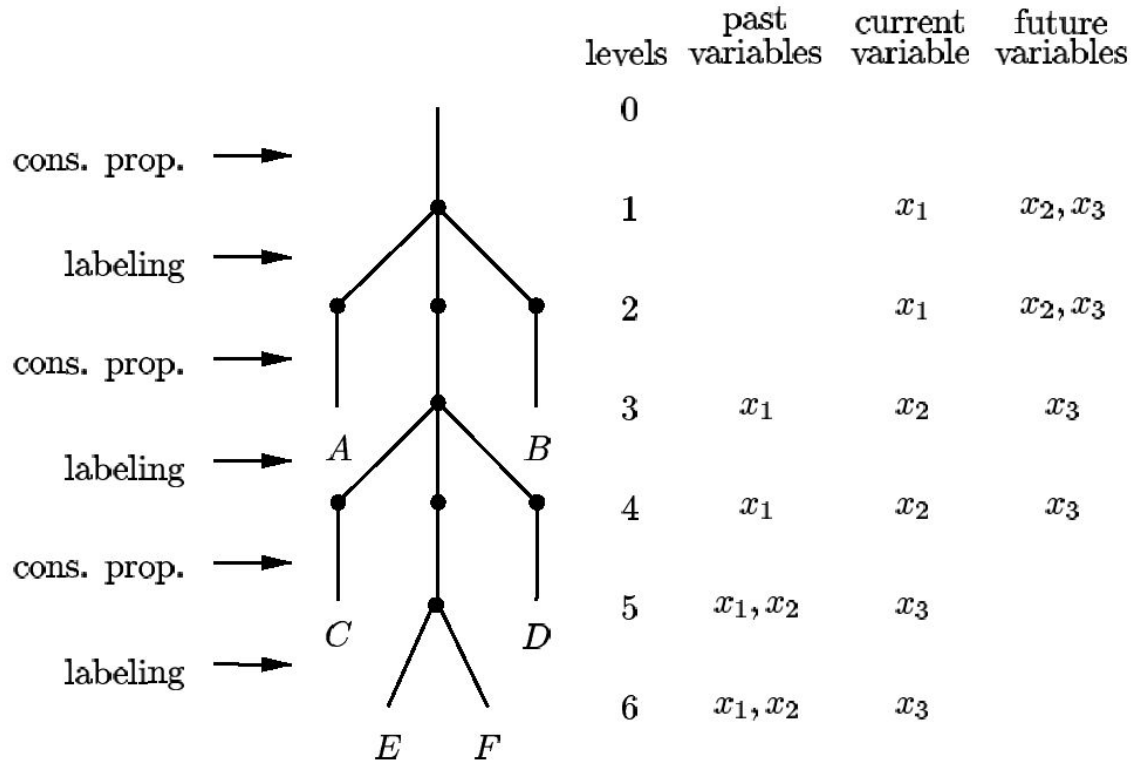
Given: node $x_1 \in E_1, \dots, x_n \in E_n$ at level $2i - 1$ or $2i$

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

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

Example of a *prop* Labeling Tree

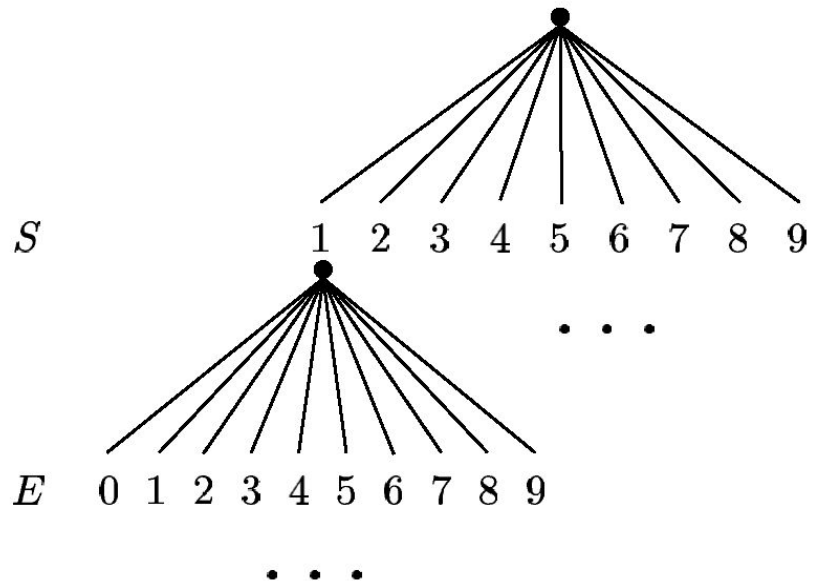
Consider a CSP with three variables, x_1, x_2, x_3



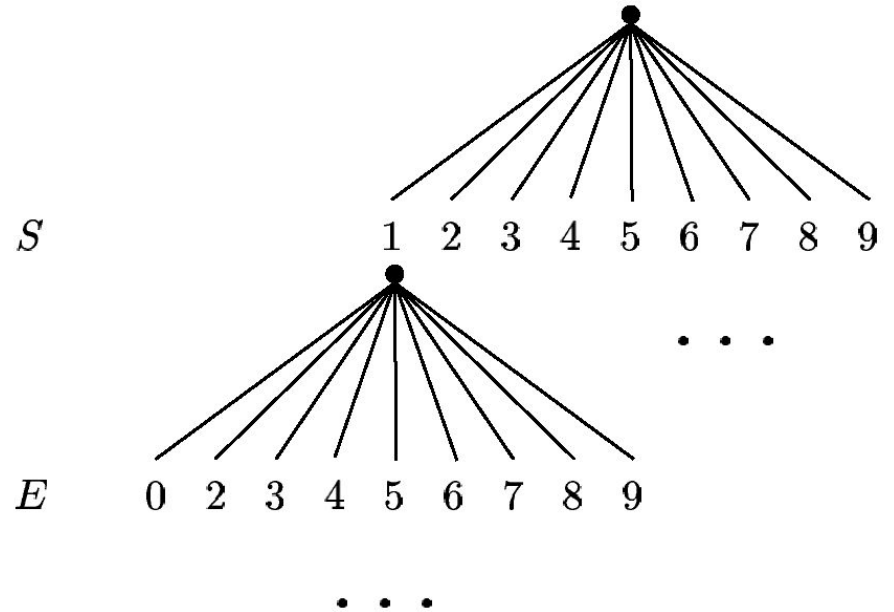
$A, B, C,$ and D are **failed** nodes. E and F are **success** nodes.

Example: SEND + MORE = MONEY

Complete Labeling Tree:



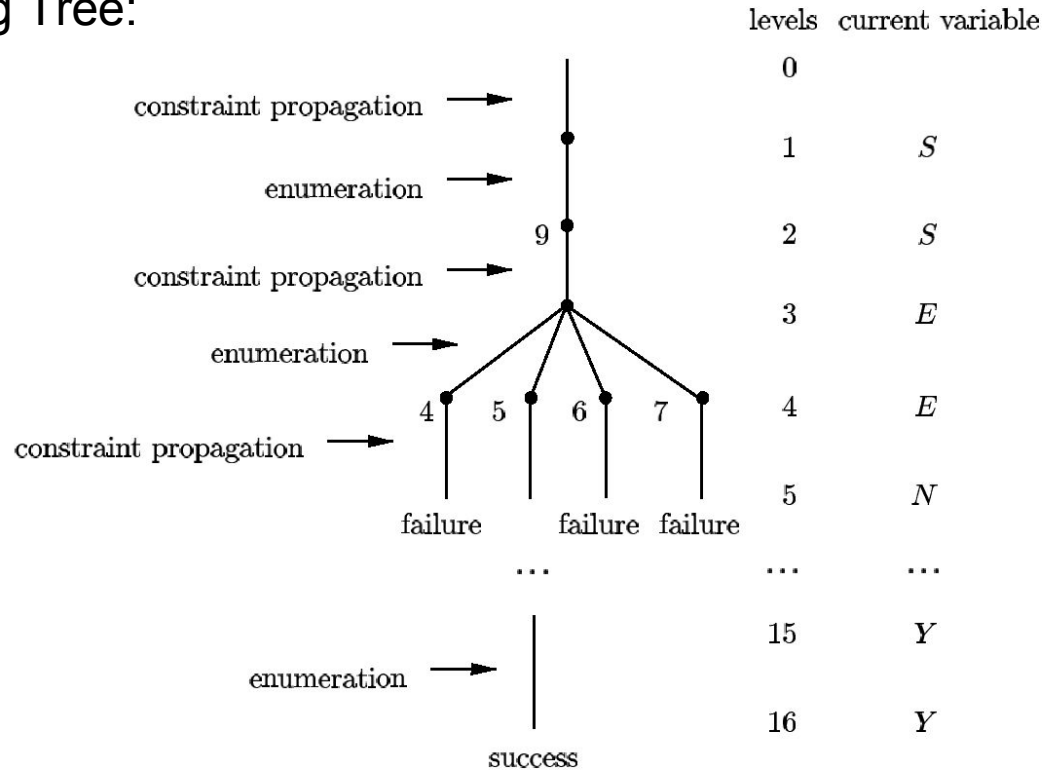
Reduced Labeling Tree:



SEND + MORE = MONEY, ctd

Use as *prop* the domain reduction rules for linear constraints over integer intervals from Chapter 5.

prop Labeling Tree:



Sizes of 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

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

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

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

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

Define

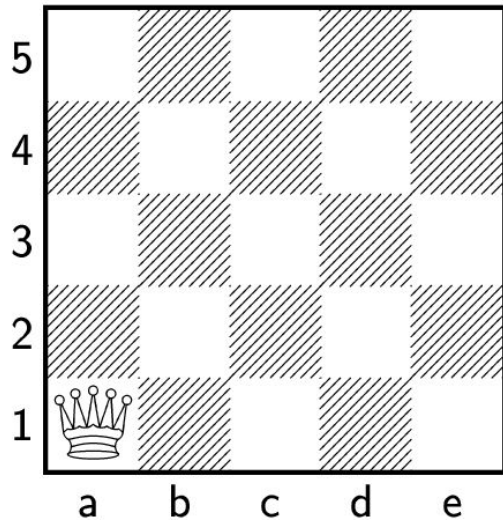
$$E'_j := \{e \in E_j \mid \{(x_1, d_1), \dots, (x_i, d_i), (x_j, e)\} \text{ is consistent}\}$$

Example: 5 Queens Problem

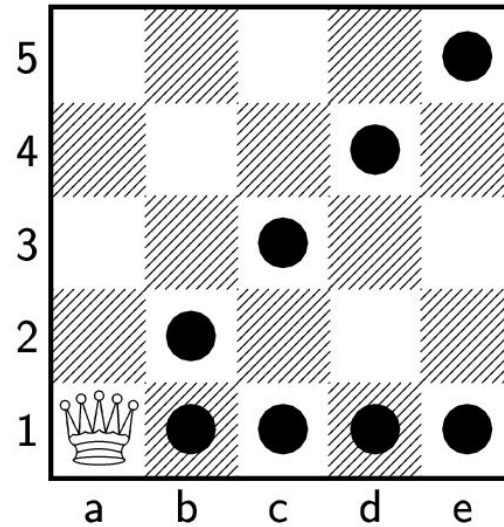
Take the standardized CSP corresponding to 5 Queens Problem.

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:

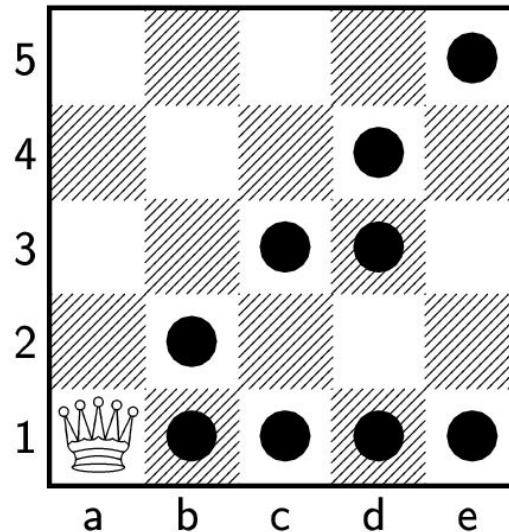


Partial Look Ahead Search Tree

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

Example: 5 Queens Problem

Effect of partial look ahead in the example:

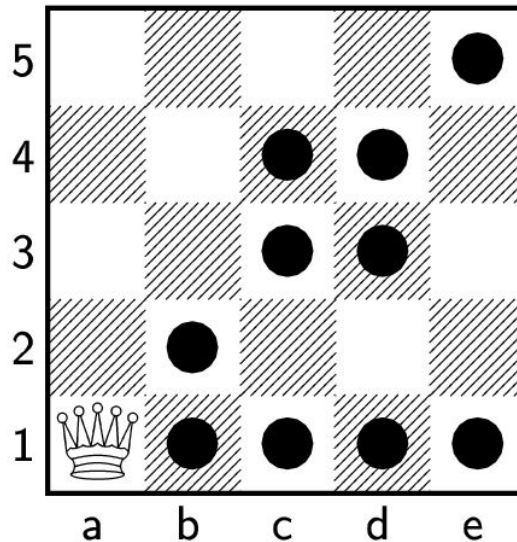


MAC Search Tree

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

Example: 5 Queens Problem

Effect of MAC in the example:



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
 - MAC

Search algorithms for constrained optimization problems:

- Branch and bound search
- Branch and bound with constraint propagation search

Declarations

$\text{cons}(\text{inst}, j, d) \equiv$ “the instantiation
 $\{(x_1, \text{inst}[1]), \dots, (x_{j-1}, \text{inst}[j - 1]), (x_j, d)\}$ is consistent”

type domains = **array** [1.. n] **of** domain;
 instantiation = **array** [1.. n] **of** elements;

var inst: instantiation;
 failure: **boolean**

Backtrack-free Search

```
procedure backtrack_free(j: integer; D: domains; var success: boolean);  
begin  
    while  $D[j] \neq \emptyset$  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-if  
    end-while  
end  
  
begin  
    success := false;  
    backtrack_free(1, D, success)  
end
```

Backtrack-free Search with Constraint Propagation

```
procedure backtrack_free_prop(j: integer; D: domains; var success: boolean);  
begin  
    while  $D[j] \neq \emptyset$  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-if  
        end-if  
    end-while  
end  
  
begin  
    success := false;  
    prop(0, D, failure);  
    if not failure then backtrack_free_prop(1, D, success)  
end
```

Backtracking

```
procedure backtrack(j: integer; D: domains; var success: boolean);  
begin  
    while  $D[j] \neq \emptyset$  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-if  
    end-while  
end  
  
begin  
    success := false;  
    backtrack(1, D, success)  
end
```

Backtracking with Constraint Propagation

```
procedure backtrack_prop(j: integer; D: domains; var success: boolean);  
begin  
    while  $D[j] \neq \emptyset$  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-if  
        end-if  
    end-while  
end  
  
begin  
    success := false;  
    prop(0, D, failure);  
    if not failure then backtrack_prop(1, D, success)  
end
```

Forward Checking

```
procedure revise(j, k: integer; var D: domains);  
begin  
     $D[k] := \{d \in D[k] \mid \{(x_1, \text{inst}[1]), \dots, (x_j, \text{inst}[j]), (x_k, d)\} \text{ is a consistent instantiation}\}$   
end  
  
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] =  $\emptyset$ );  
        k := k + 1  
    end-while  
end
```

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] =  $\phi$ );  
        k := k + 1  
    end-while  
    if not failure then darc(j + 1, D, failure)  
end
```


MAC (Full Look Ahead)

```
procedure prop(j: integer; var D: domains; var failure: boolean);  
...  
    if not failure then arc(j + 1, D, failure)  
end
```

Searching for All Solutions

```
procedure backtrack_all(j: integer; D: domains);  
begin  
    while  $D[j] \neq \emptyset$  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-if  
        end-while  
    end  
  
    begin  
        backtrack_all(1, D)  
    end
```

Finite Constrained Optimization Problems

- $\mathcal{P} := \langle C ; x_1 \in D_1, \dots, x_n \in D_n \rangle$
- $obj : Sol \rightarrow \mathbb{R}$ from the set Sol of all solutions to \mathcal{P} to \mathbb{R}
- Heuristic function $h : \mathcal{P}(D_1) \times \dots \times \mathcal{P}(D_n) \rightarrow \mathbb{R} \cup \{\infty\}$

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

Bound: $obj(d_1, \dots, d_n) \leq h(\{d_1\}, \dots, \{d_n\})$

procedure obj (inst: instantiation): **real**;

procedure h (inst: instantiation; j : **integer**; D : domains): **real**;

h (inst, j , D) returns the value of h on $(\{inst[1]\}, \dots, \{inst[j]\}, D[j + 1], \dots, D[n])$

Branch and Bound

```
procedure branch_and_bound(j: integer; D: domains; var solution: instantiation; var bound: real);  
begin  
    while  $D[j] \neq \emptyset$  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-if  
            else if  $h(\text{inst}, j, D) > \text{bound}$  then  
                branch_and_bound( $j + 1$ , D, solution, bound)  
            end-if  
        end-if  
    end-while  
end  
  
begin  
    solution := nil; bound :=  $-\infty$ ;  
    branch_and_bound(1, D, solution, bound)  
end
```

Branch and Bound with Constraint Propagation

```
procedure branch_and_bound_prop(j: integer; D: domains; var solution: instantiation; var bound: real);  
begin  
  while  $D[j] \neq \emptyset$  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-if  
      else  
        prop(j, D, failure);  
        if not failure and  $h(inst, j, D) > bound$  then  
          branch_and_bound_prop( $j + 1$ , D, solution, bound)  
        end-if  
      end-if  
    end-if  
  end-while  
end
```

Branch and Bound with Constraint Propagation, ctd

```
begin  
  solution := nil;  
  bound :=  $-\infty$ ;  
  prop(0, D, failure);  
  if not failure then  
    branch_and_bound_prop(1, D, solution, bound)  
  end
```

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 the heuristic function that 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
 - maintaining arc consistency (MAC)
- Discuss various search algorithms for labeling trees
- Discuss search algorithms for constrained optimization problems
- Introduce various heuristics for search algorithms