Figaro
Yet Another Constraint Programming Library

Martin Henz, Ka Boon Ng
National University of Singapore
Tobias Müller
Universität des Saarlandes

Finite Domain Constraint Programming

A technique for solving combinatorial search and optimization problems.

```
  S E N D
+ M O R E
---------
= M O N E Y
```

Modeling

```
  S E N D
+ M O R E
---------
= M O N E Y
```

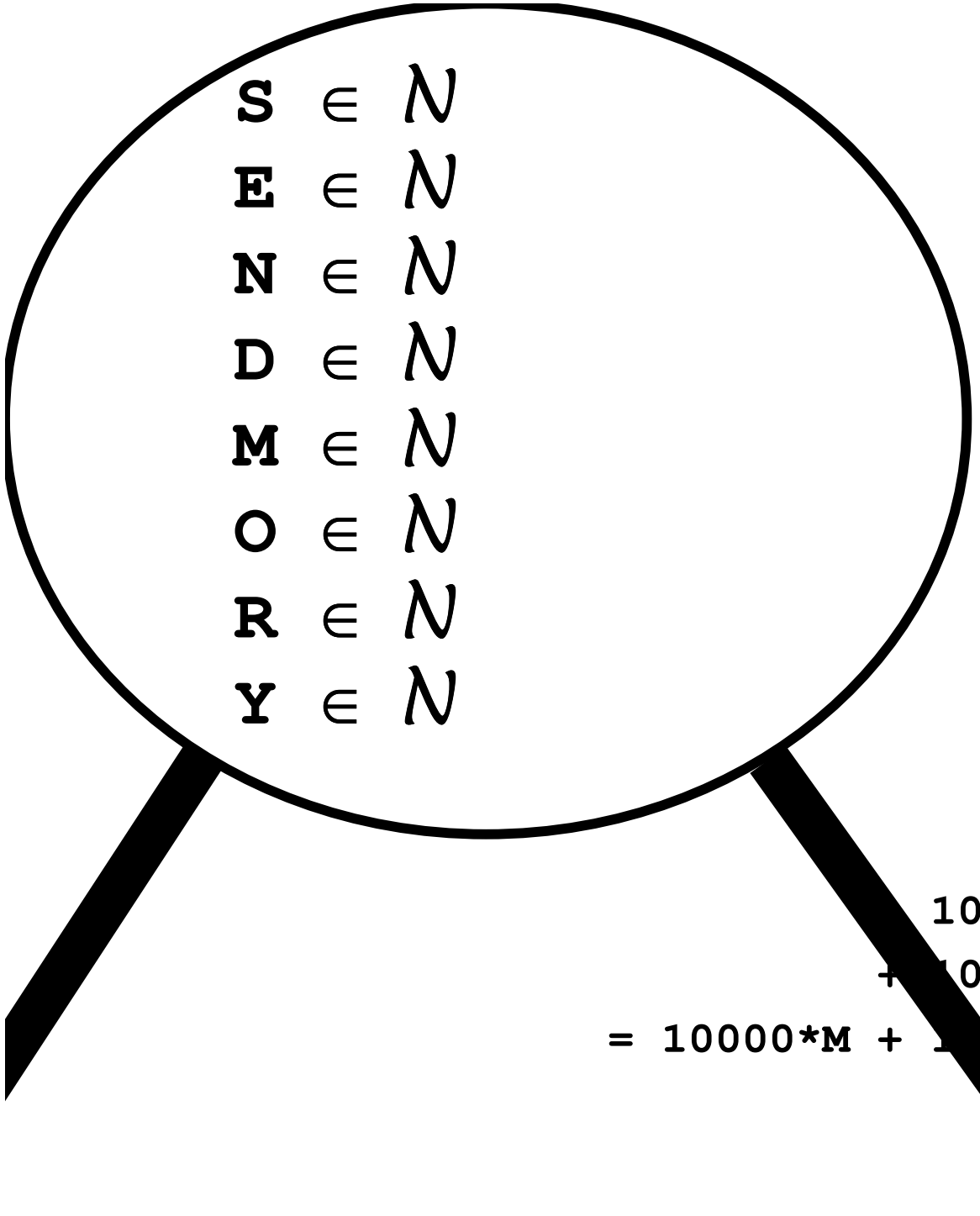
$0 \le S, \ldots, Y \le 9$

$S \neq 0$

$M \neq 0$

S…Y all different

```
      1000*S + 100*E + 10*N + D
+ 1000*M + 100*O + 10*R + E
= 10000*M + 1000*O + 100*N + 10*E + Y
```




```
  S E N D
+ M O R E
---------
= M O N E Y
```

$0 \le S, \ldots, Y \le 9$

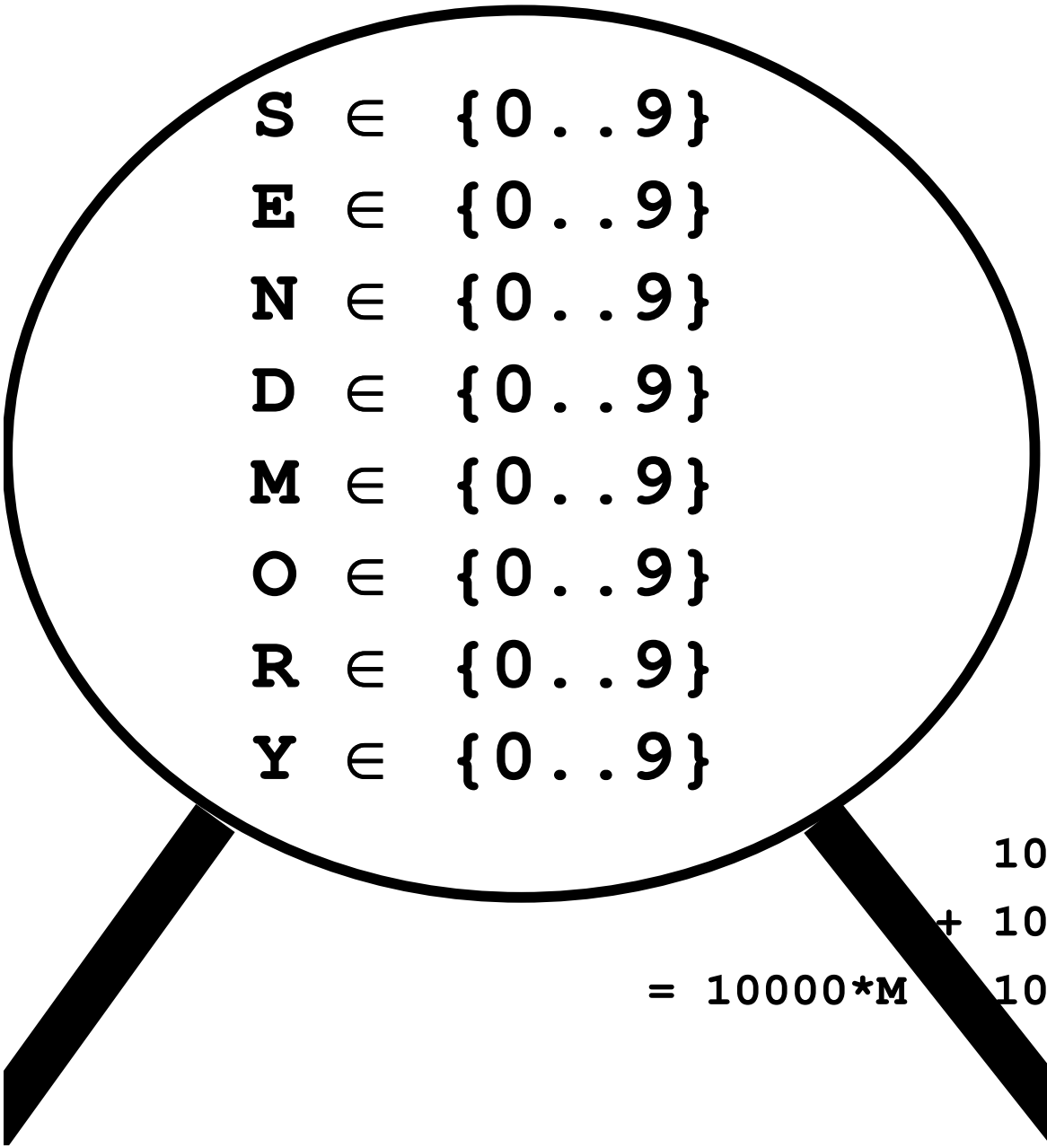
$S \neq 0$

$M \neq 0$

S…Y all different

```
  1000*S + 100*E + 10*N + D
+ 1000*M + 100*O + 10*R + E
= 10000*M + 1000*O + 100*N + 10*E + Y
```

Propagate




```
  S E N D
+ M O R E
---------
= M O N E Y
```

$0 \le S, \ldots, Y \le 9$

$S \neq 0$

$M \neq 0$

S…Y all different

```
  1000*S + 100*E + 10*N + D
+ 1000*M + 100*O + 10*R + E
= 10000*M + 1000*O + 100*N + 10*E + Y
```

Propagate

```
  S E N D
+ M O R E
---------
= M O N E Y
```

$S \in \{1..9\}$
$E \in \{0..9\}$
$N \in \{0..9\}$
$D \in \{0..9\}$
$M \in \{1..9\}$
$O \in \{0..9\}$
$R \in \{0..9\}$
$Y \in \{0..9\}$

$0 \le S, \ldots, Y \le 9$

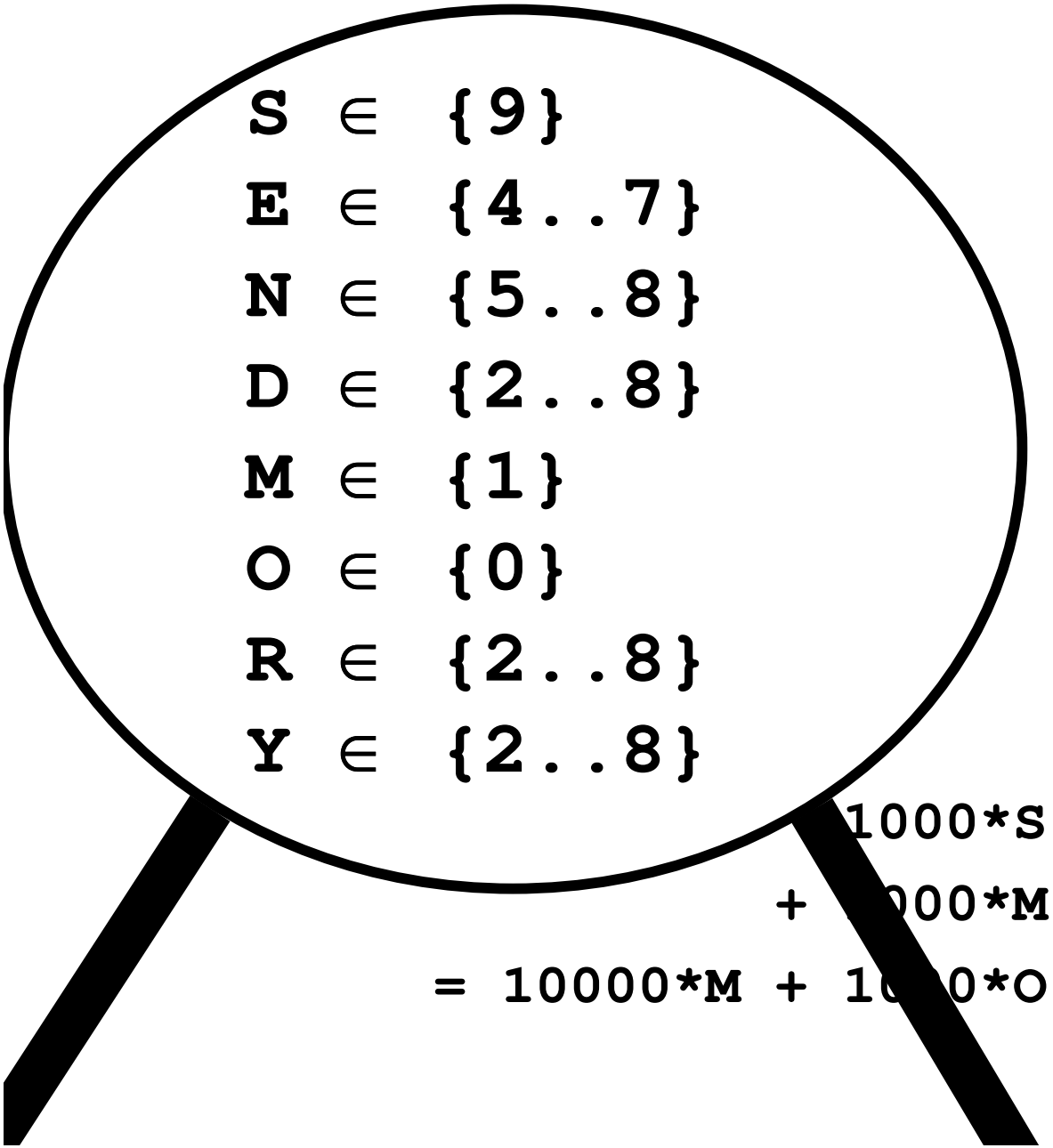
$S \neq 0$

$M \neq 0$

S…Y all different

$1000*S + 100*E + 10*N + D$
$+ 1000*M + 100*O + 10*R + E$
$= 10000*M + 100*O + 100*N + 10*E + Y$

Propagate




```
  S E N D
+ M O R E
---------
= M O N E Y
```

$0 \le S, \ldots, Y \le 9$

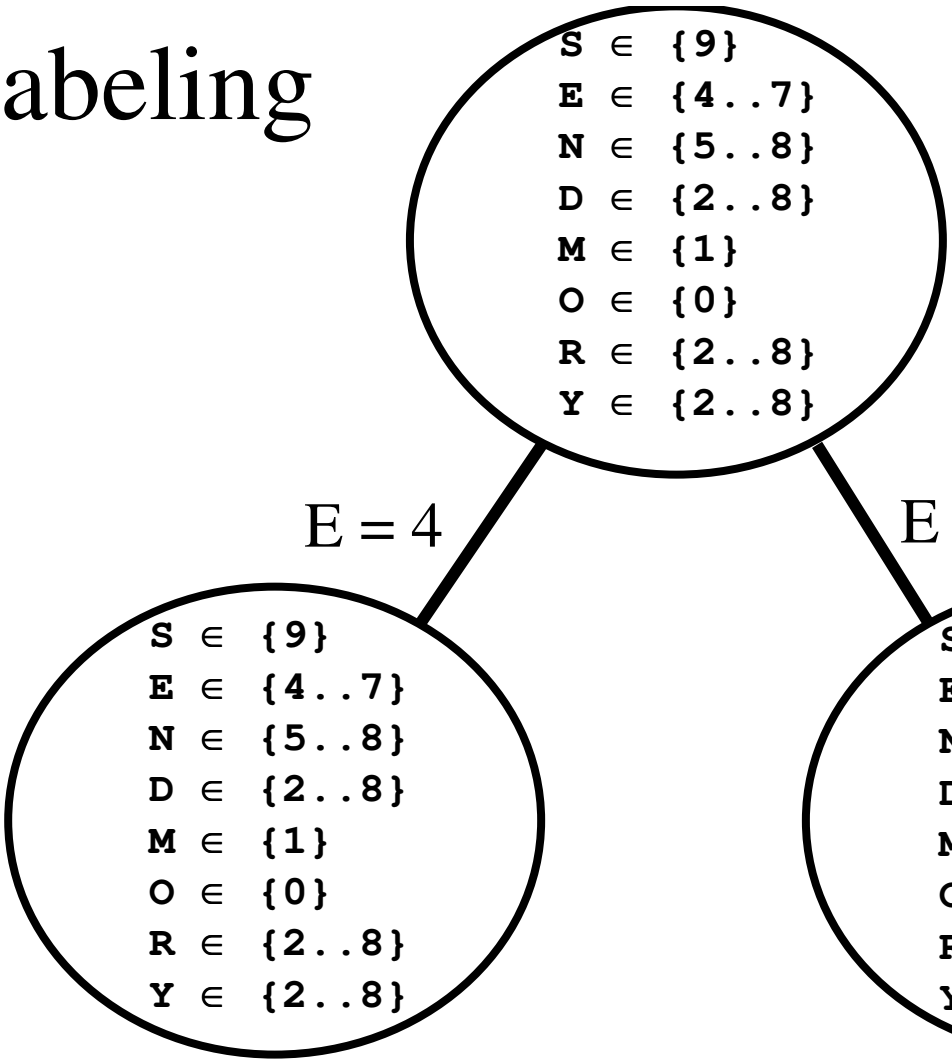
$S \neq 0$

$M \neq 0$

S…Y all different

$$1000*S + 100*E + 10*N + D$$
$$+ \ 1000*M + 100*O + 10*R + E$$
$$= 10000*M + 1000*O + 100*N + 10*E + Y$$

Labeling




```
  S E N D
+ M O R E
---------
= M O N E Y
```

$0 \le S, \ldots, Y \le 9$

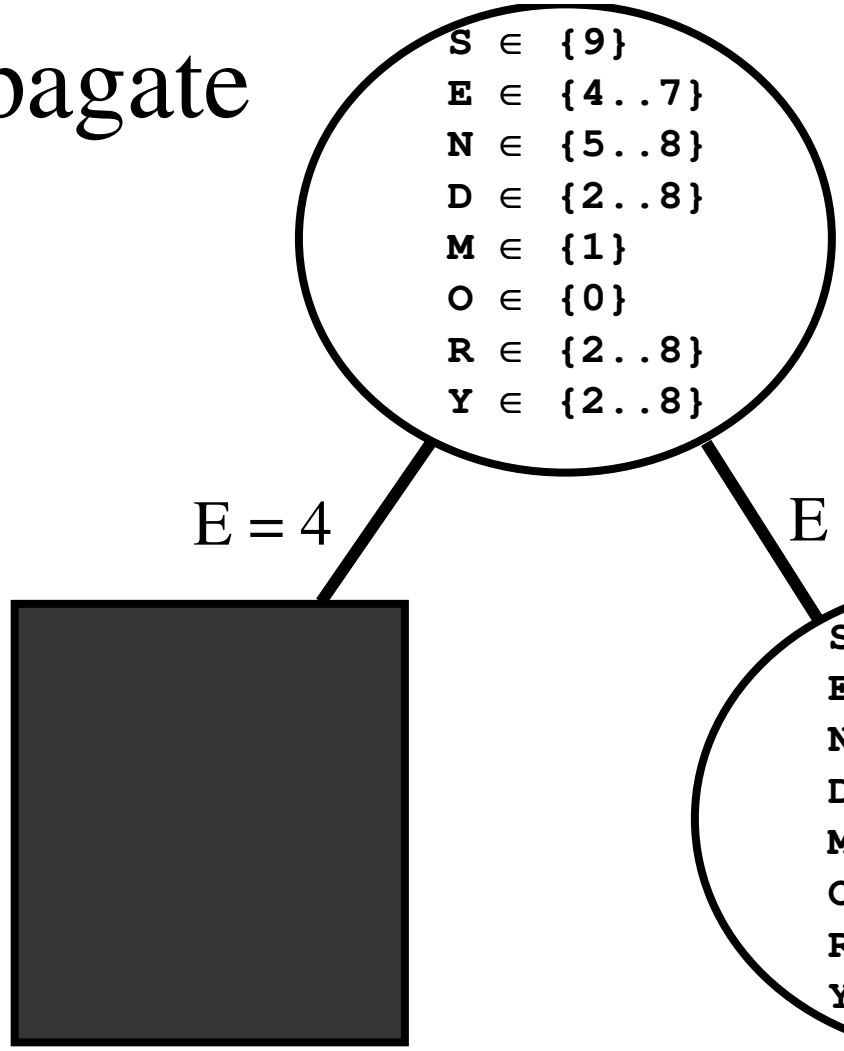
$S \neq 0$

$M \neq 0$

S…Y all different

$$1000*S + 100*E + 10*N + D$$
$$+ 1000*M + 100*O + 10*R + E$$
$$= 10000*M + 1000*O + 100*N + 10*E + Y$$

Propagate




```
  S E N D
+ M O R E
---------
= M O N E Y
```

$0 \le S, \ldots, Y \le 9$

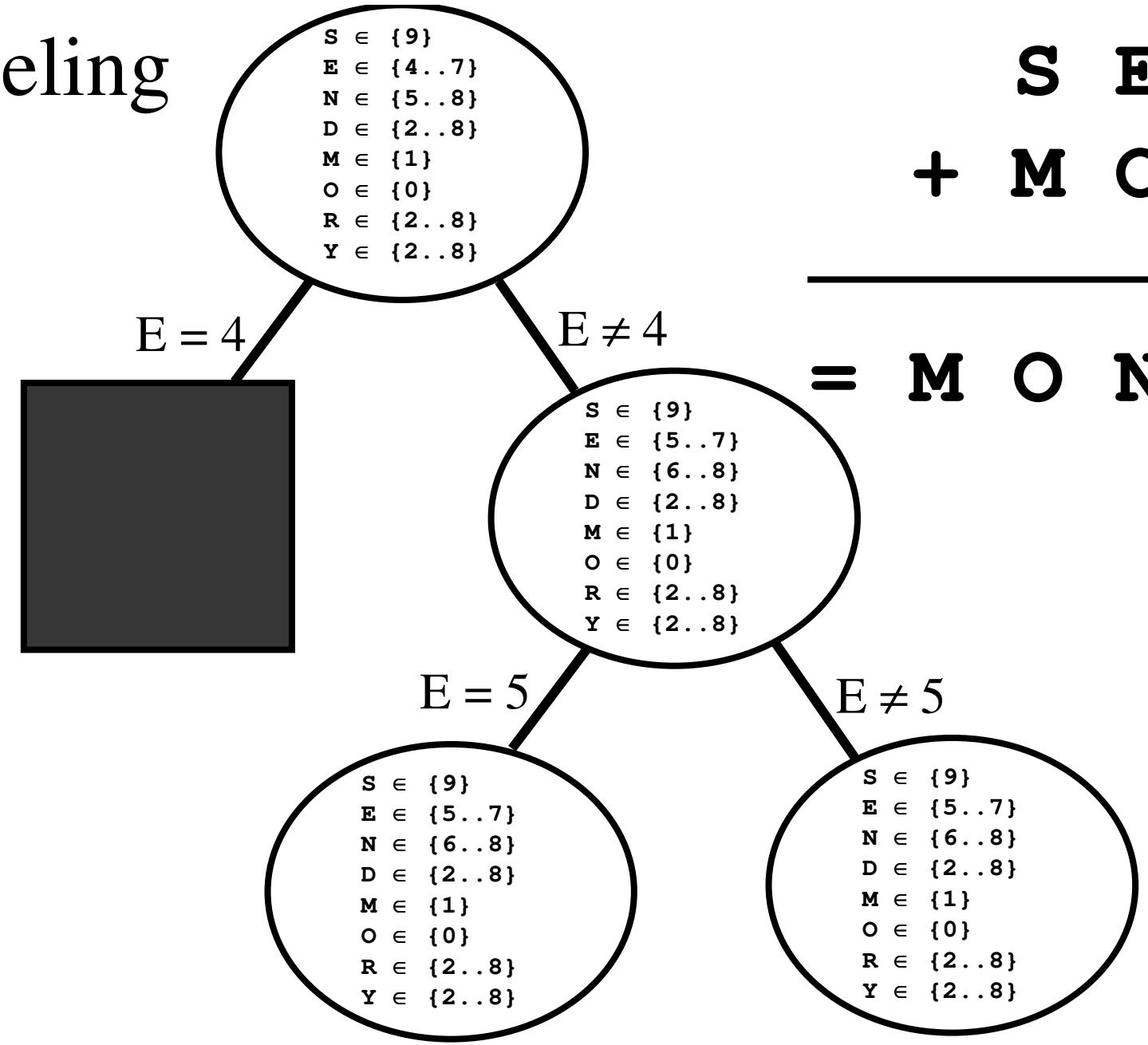
$S \neq 0$

$M \neq 0$

S…Y all different

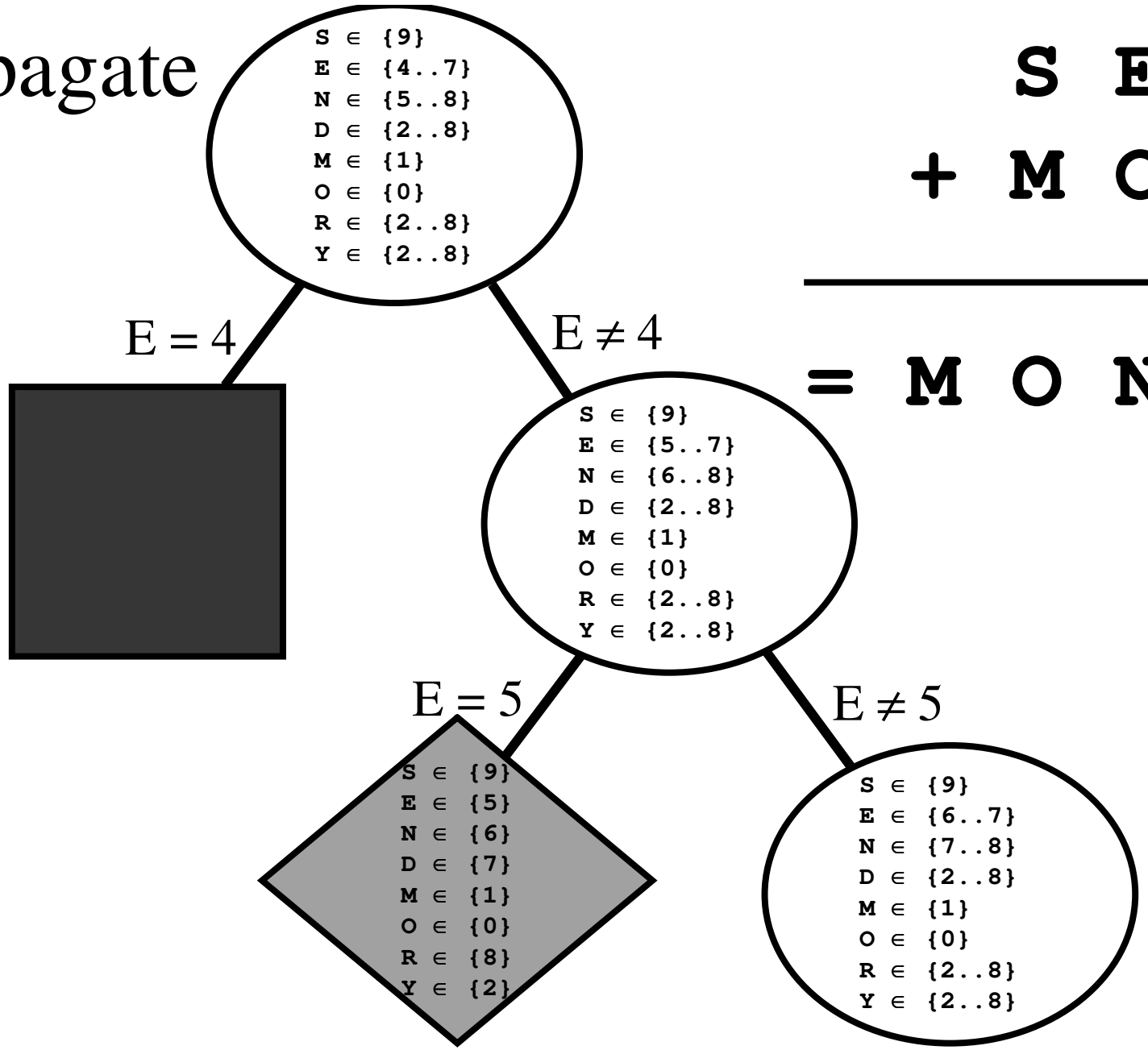
```
1000*S + 100*E + 10*N + D
+ 1000*M + 100*O + 10*R + E
= 10000*M + 1000*O + 100*N + 10*E + Y
```

Labeling



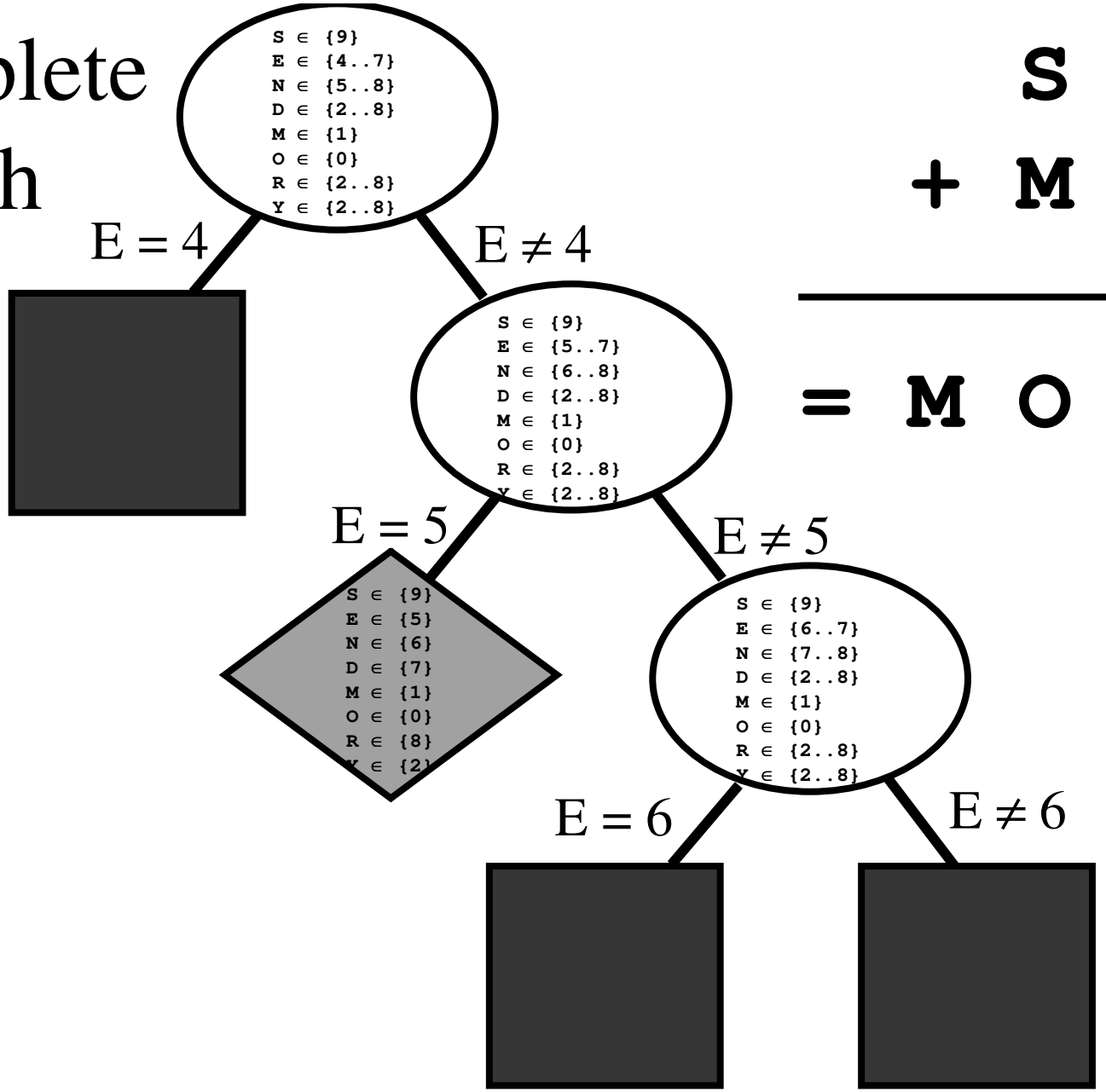

```
  S E N D
+ M O R E
---------
= M O N E Y
```

Propagate




```
  S E N D
+ M O R E
---------
= M O N E Y
```

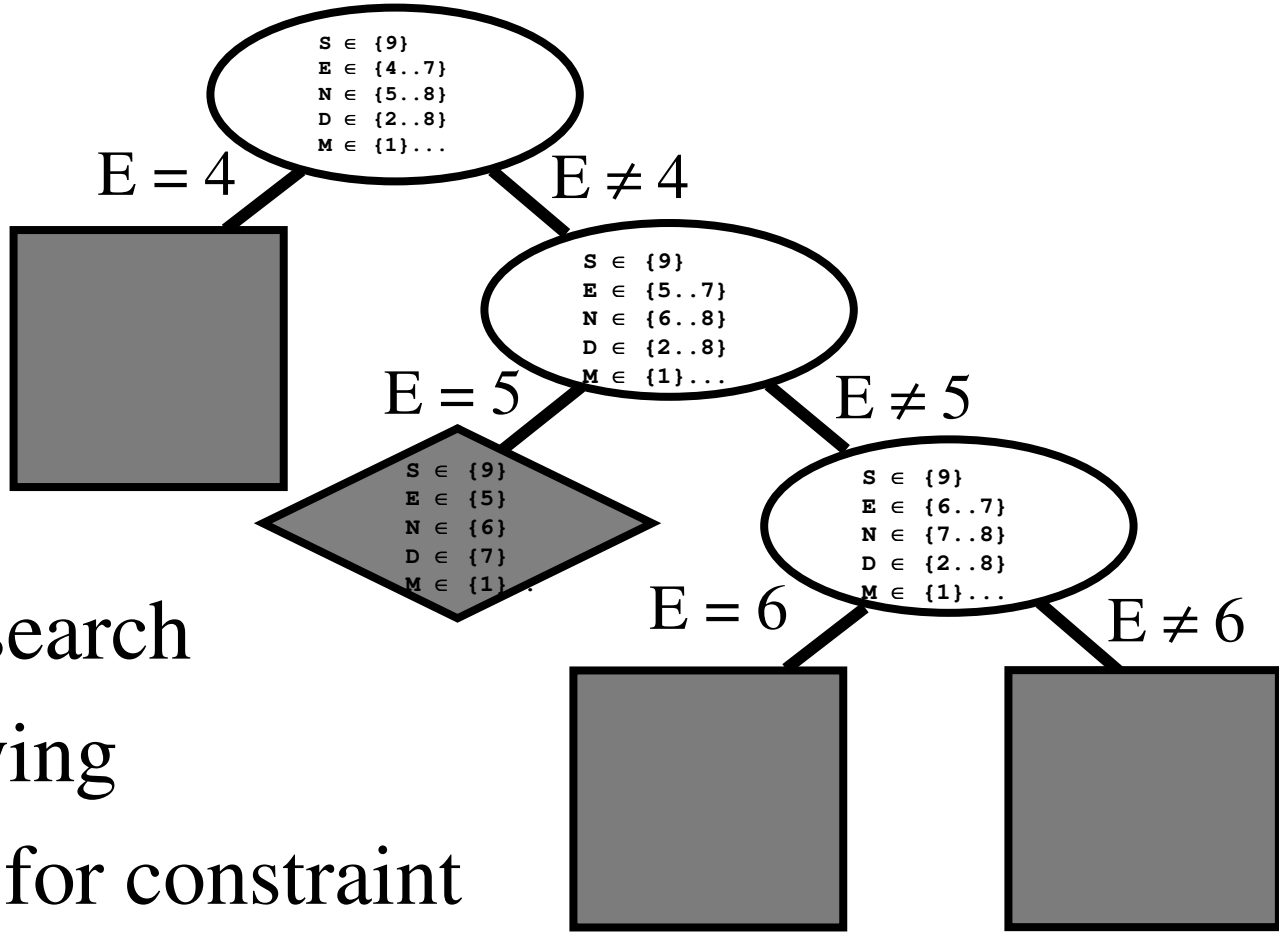
Complete Search Tree




```
  S E N D
+ M O R E
---------
= M O N E Y
```

Is That All?

- Complex symbolic constraints
- Programmable search
- Trailing vs copying
- Design patterns for constraint programming
- Integration of other search techniques



FD(CP) Languages and Libraries

- Languages
  - CLP(FD)
  - Eclipse, CHIP

  } Prolog based
  - CLAIRE — new language
  - Oz — new language
- Libraries
  - PECOS — LISP
  - Ilog Solver — C++
  - Figaro — C++, under development

Design Issues

- Languages
  - CLP(FD)
  - Eclipse, CHIP
  - CLAIRE
  - Oz
- Libraries
  - PECOS
  - Ilog Solver
- Syntax
- Integration with application programming
- Programming propagation algorithms
- Programming search strategies
- Programming search algorithms

Representing the Constraint Store

```
class store {
public:
   store();
   var newvar(int lo,
              int hi);
   var getlo(var v);
   var gethi(var v);
   ...
}
```

Representing the Constraint Store

```
class store {
public:
   store();
   var newvar(int lo,
              int hi);
   var getlo(var v);
   var gethi(var v);
   ...
}
```

```
int main(...) {
   store*st=new store();
...}
```

Representing the Constraint Store

```
class store {
public:
   store();
   var newvar(int lo,
              int hi);
   var getlo(var v);
   var gethi(var v);
   ...
}
```

```
int main(...) {
   store*st=new store();
   s = st -> newvar(0,9);
   ...
   y = st -> newvar(0,9);


...}
```

Representing the Constraint Store

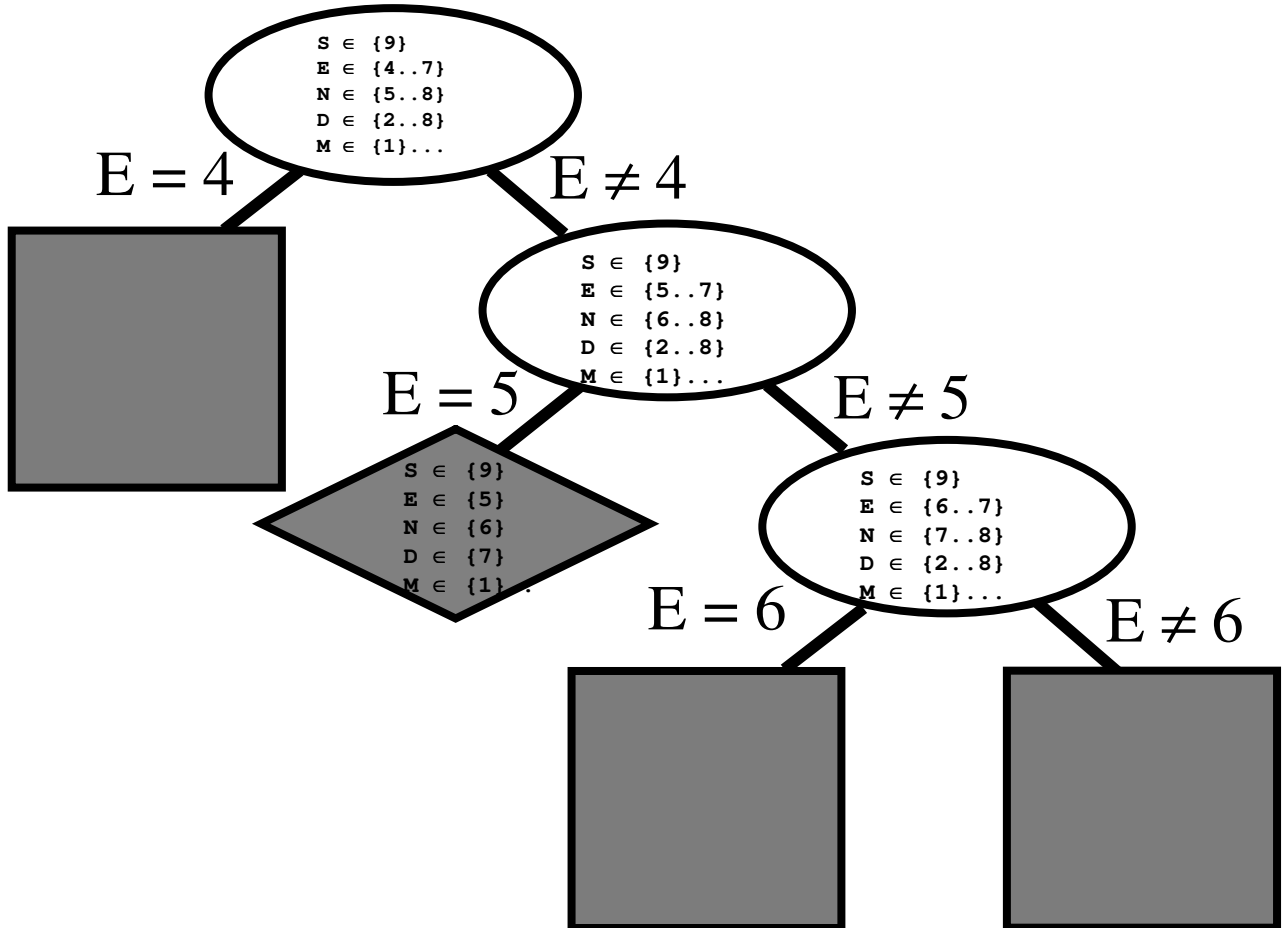
```
class store {
public:
   store();
   var newvar(int lo,
              int hi);
   var getlo(var v);
   var gethi(var v);
   ...
}
```

```
int main(...) {
    store*st=new store();
    s = st -> newvar(0,9);
    ...
    y = st -> newvar(0,9);

    all_different(st,
                  s,..,y);
...}
```

Representing Search Trees

```
class node {
   public: virtual node* child(store *,int)=0;
}
```

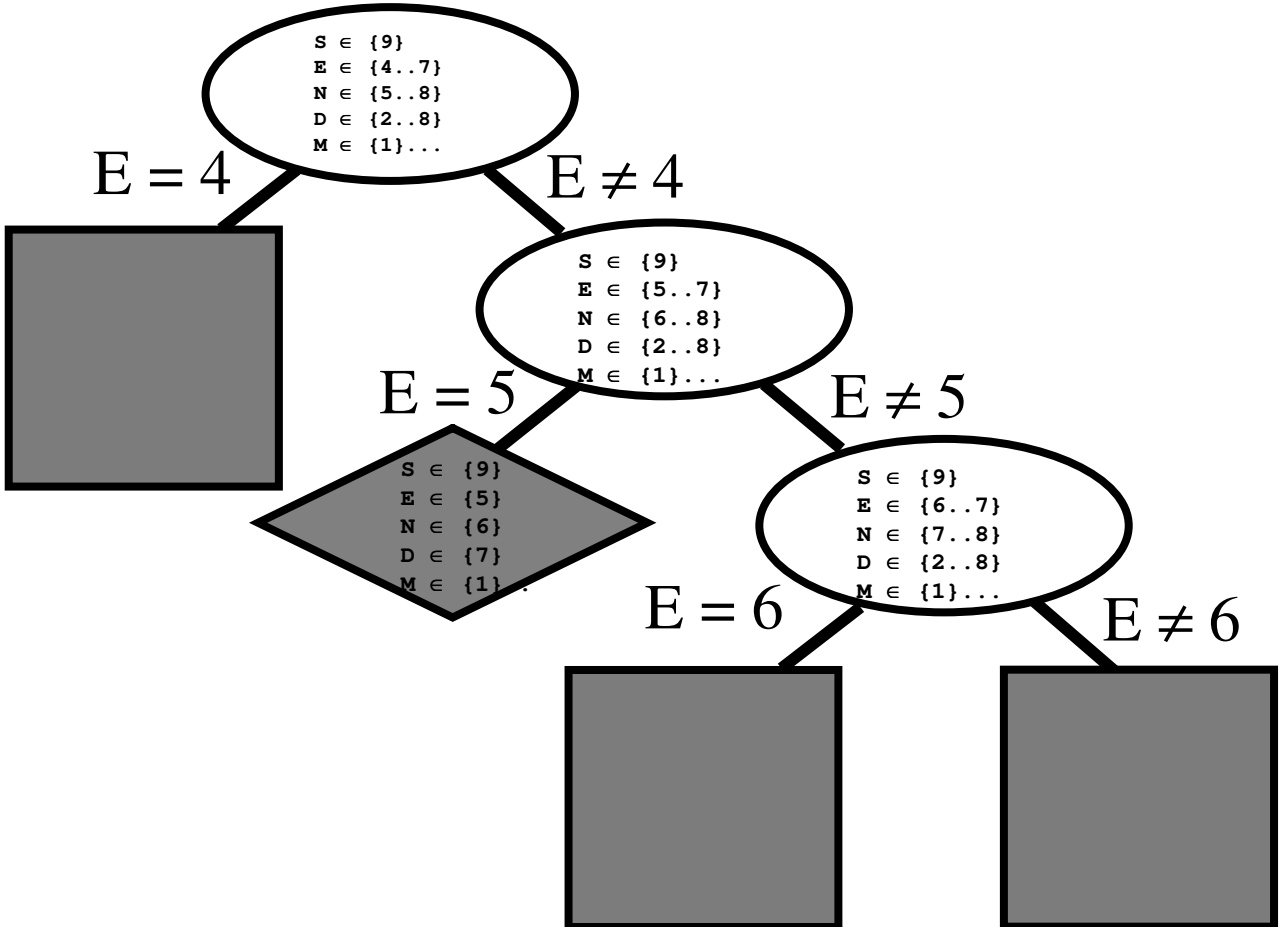


Representing Search Trees

```
class naive : public node {
  private: int idx;vector<var> vars;node*subtree;
  public :
   naive(vector<var> vs,int i,node*t) : ... {}
   node* child(store*st, int i) {
    if (i==0) {
      st->tell(vars[idx],st->getlo(vars[idx]));
      return (idx+1==var.size() ? subtree :
              new naive(vars,idx+1,subtree));
    } else {
      st->tell(vars[idx],st->getlo(vars[idx])+1,
                         st->gethi(vars[idx]));
      return new naive(vars,idx,subtree);
    }}};
```

Backtracking

```
mark store::mark();
void store::backtrack(mark m);
```



Depth-first Search (backtracking)

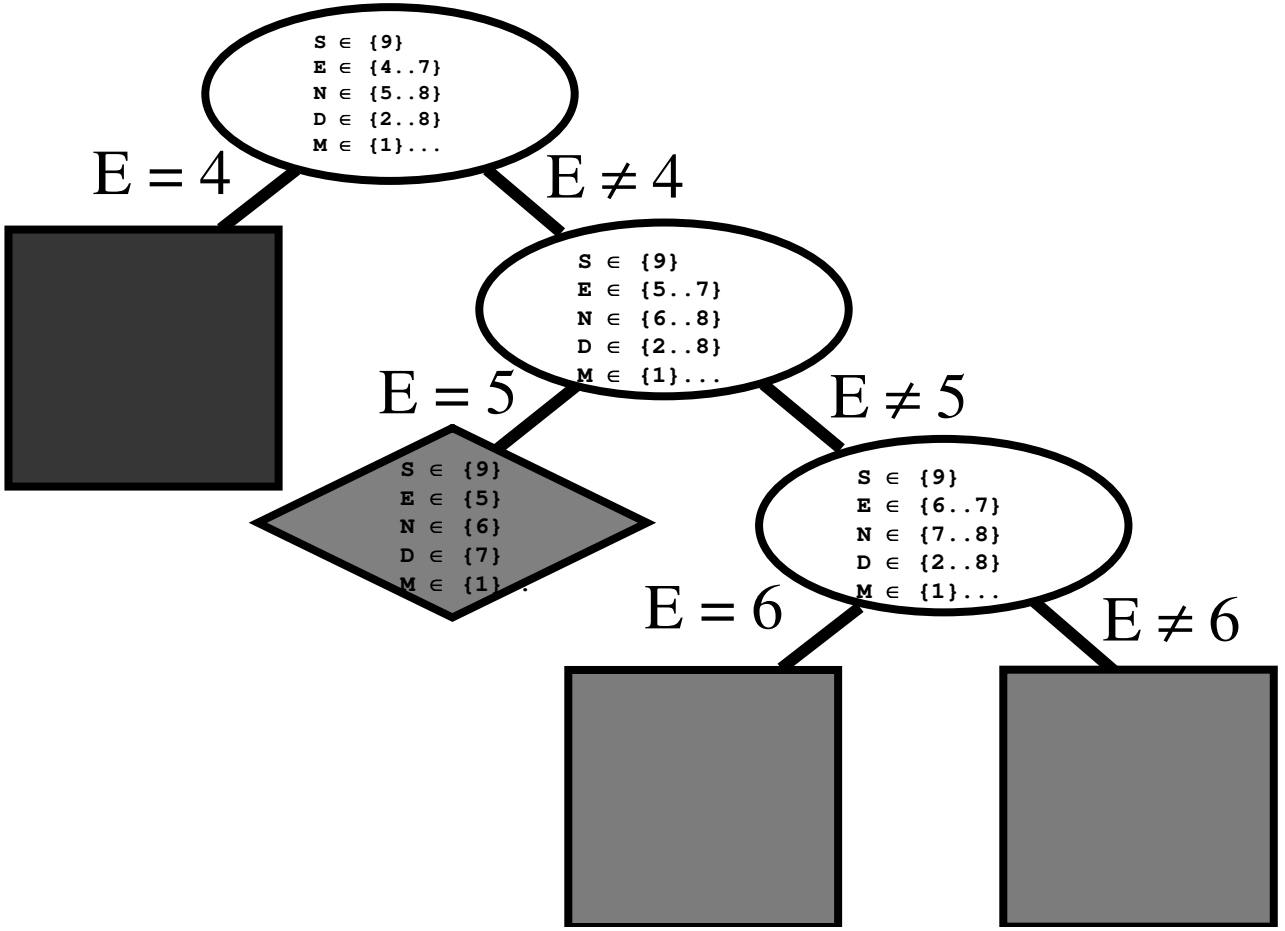
```
node * solve_one(store * s,node * t) {
   if (t == NULL) return t;
   mark m = s->mark();
   try {return solve_one(s,t->child(s,0));}
   catch (Failure) {
      s->backtrack(m);
      return solve_one(s,t->child(s,1));
   }
}
```

Stepping Back

- Nothing new so far; Oz has programmable search algorithms through spaces (Schulte, ICLP'97); Ilog Solver has **`IlcManager`** objects.
- But:
  - "lean" stores
  - nodes as an abstraction for programming search strategies
- more on the basic data structures in the proceedings
- moving on: Trailing vs. Copying (Schulte, ICLP'99)

Trailing vs Copying

- Trail and backtrack
- Copy and abandon



Copying Stores

- Simply copy the store with all variables and constraints:

```
store::store(const store*);
```

- But: nodes refer to variables;
  their addresses change!

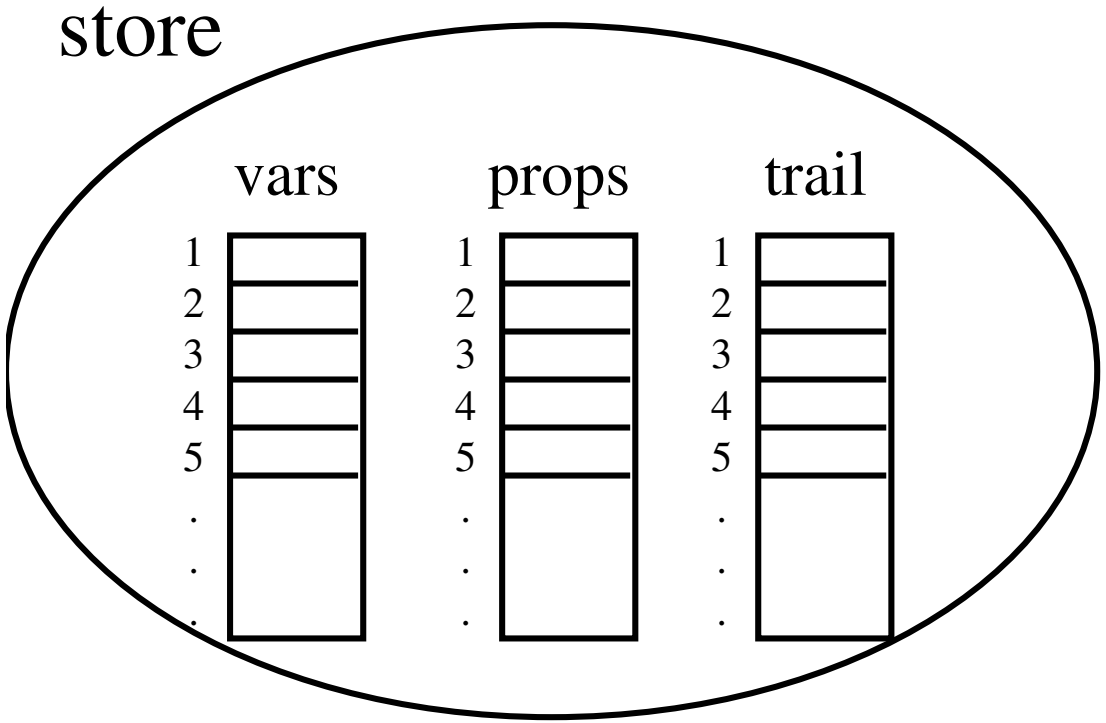
Labeling Refers to Variables

```
class naive : public node {
  private: int idx;vector<var> vars;node*subtree;
  public :
   naive(vector<var> vs,int i,node*t) : ... {}
   node* child(store*st, int i) {
    if (i==0) {
      st->tell(vars[idx],st->getlo(vars[idx]));
      return (idx+1==vars.size() ? subtree :
              new naive(vars,idx+1,subtree));
    } else {
      st->tell(vars[idx],st->getlo(vars[idx])+1,
                         st->gethi(vars[idx]));
      return new naive(vars,idx,subtree);
    }}};
```

Search Refers to Marks

```
node * solve_one(store * s,node * t) {
    if (t == NULL) return t;
    mark m = s->mark();
    try {return solve_one(s,t->child(s,0));}
    catch (Failure) {
        s->backtrack(m);
        return solve_one(s,t->child(s,1));
    }
}
```

Indirect Addressing



only use indices in arrays to address variables, propagators and marks

Now nodes don't need to care, whether they work on original or copy.

Depth-first Search (copying)

```
node * solve_one(store * s,node * t) {
    if (t == NULL) return t;
    store s2 = new store(s);
    try {return solve_one(s,t->child(s,0));}
    catch (Failure) {
      return solve_one(s2,t->child(s,1));
    }
}
```

Stepping Back

- Indirect addressing allows copy and trailing-based search
- Possibly interesting combinations of the two to be explored
- Moving on: Memory policy just one dimension in the “design space” for search algorithms.

Dimensions in the Design Space for Search Algorithms

- Exploration: depth-first search, limited-discrepancy search,...
- Memory policy: copying, trailing,...
- Optimization: branch-and-bound, restart optimization,...
- Interaction: first, last, all solution search, tracing,...
- Visualization: “Oz Explorer” like search tree visualization

STK: Software Engineering of Inference Engines

- STK originally developed for Oz
- Allows to separate design dimensions into modules
- Software architecture ensures reusability of components
- STK ideas will be at the heart of Figaro

Example: Exploration and Interaction

- Exploration is defined by a class that extends the abstract class **`exploration`**.

```
class exploration {
public: virtual node * one_step()=0; }
```

Example: Exploration and Interaction

- Exploration is defined by a class that extends the abstract class **`exploration`**.

```
class exploration {
public: virtual node * one_step()=0; }
```

- Interaction calls **`one_step`** according to the desired mode of interaction.

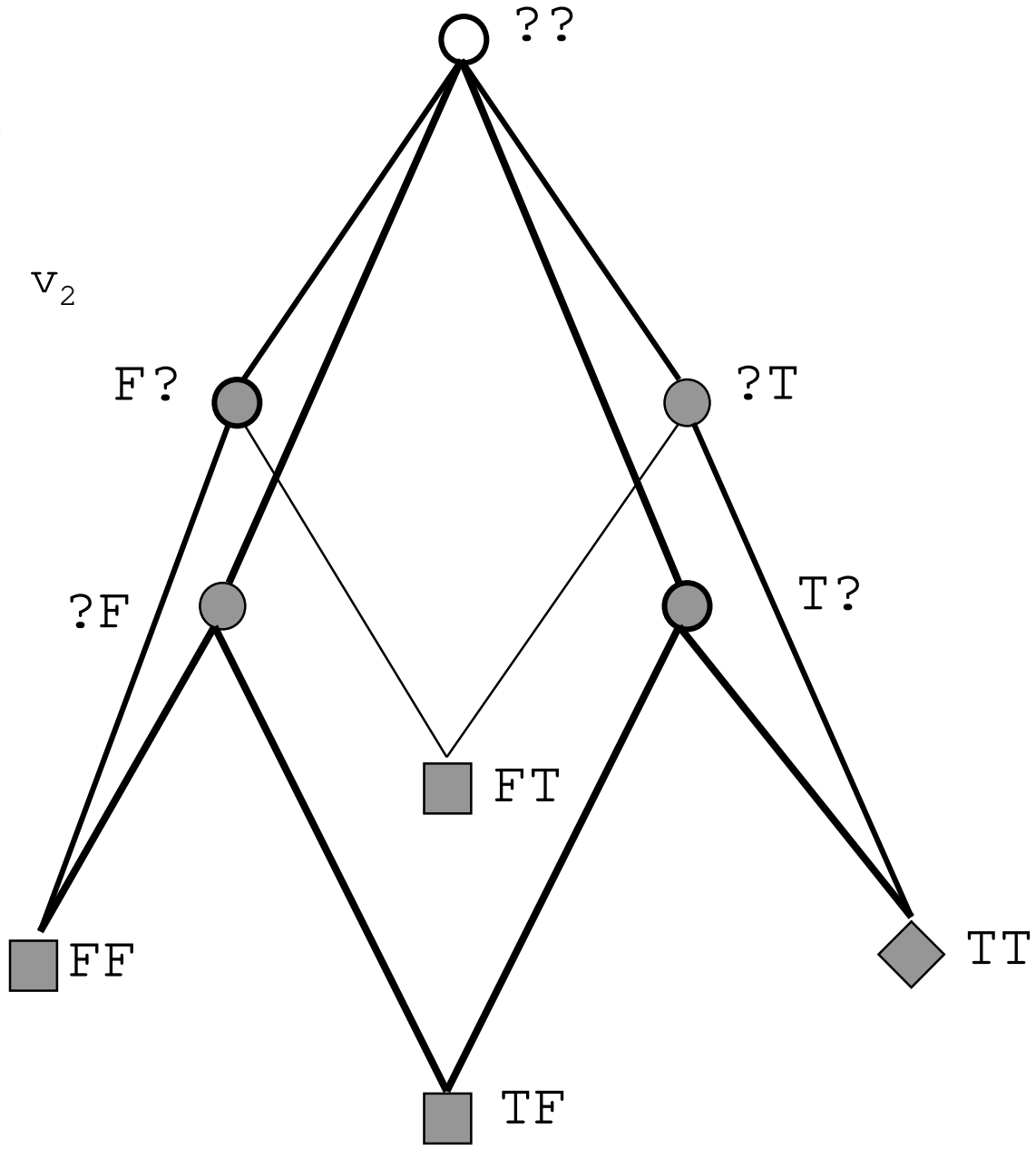
```
node * first(exploration * e) {
    node * n;
    while (n = e->one_step())
        if (n==success_node) break;
    return n; }
```

Stepping Back

- Design patterns for tree search
- More in paper: A Search Toolkit for Constraint Programming (PADL'00)
- Moving on: Constraint programming is just one technique: local search?

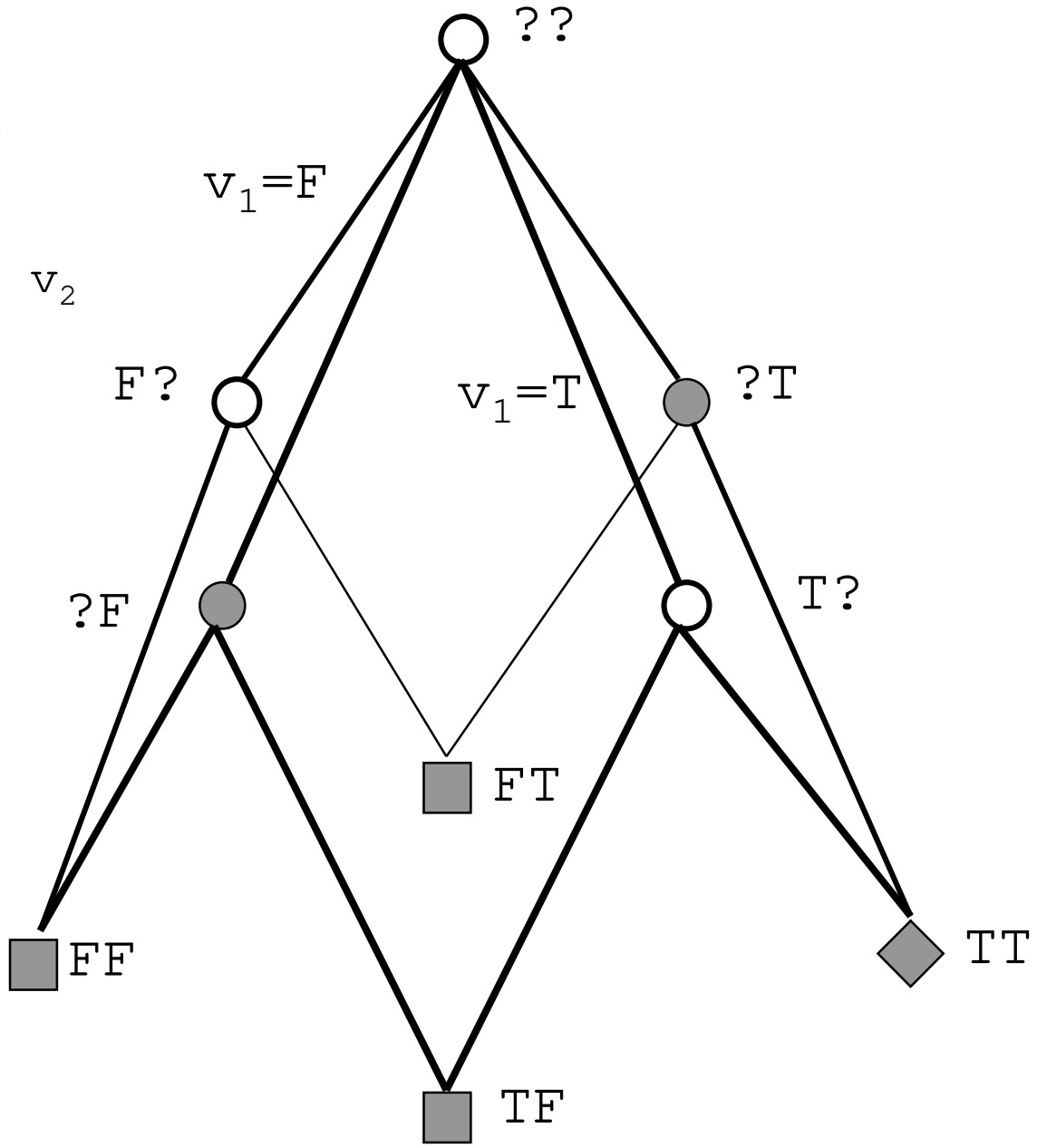
Global Search

$\overline{v_1} \vee v_2, v_1 \vee \overline{v_2}, v_1 \vee v_2$



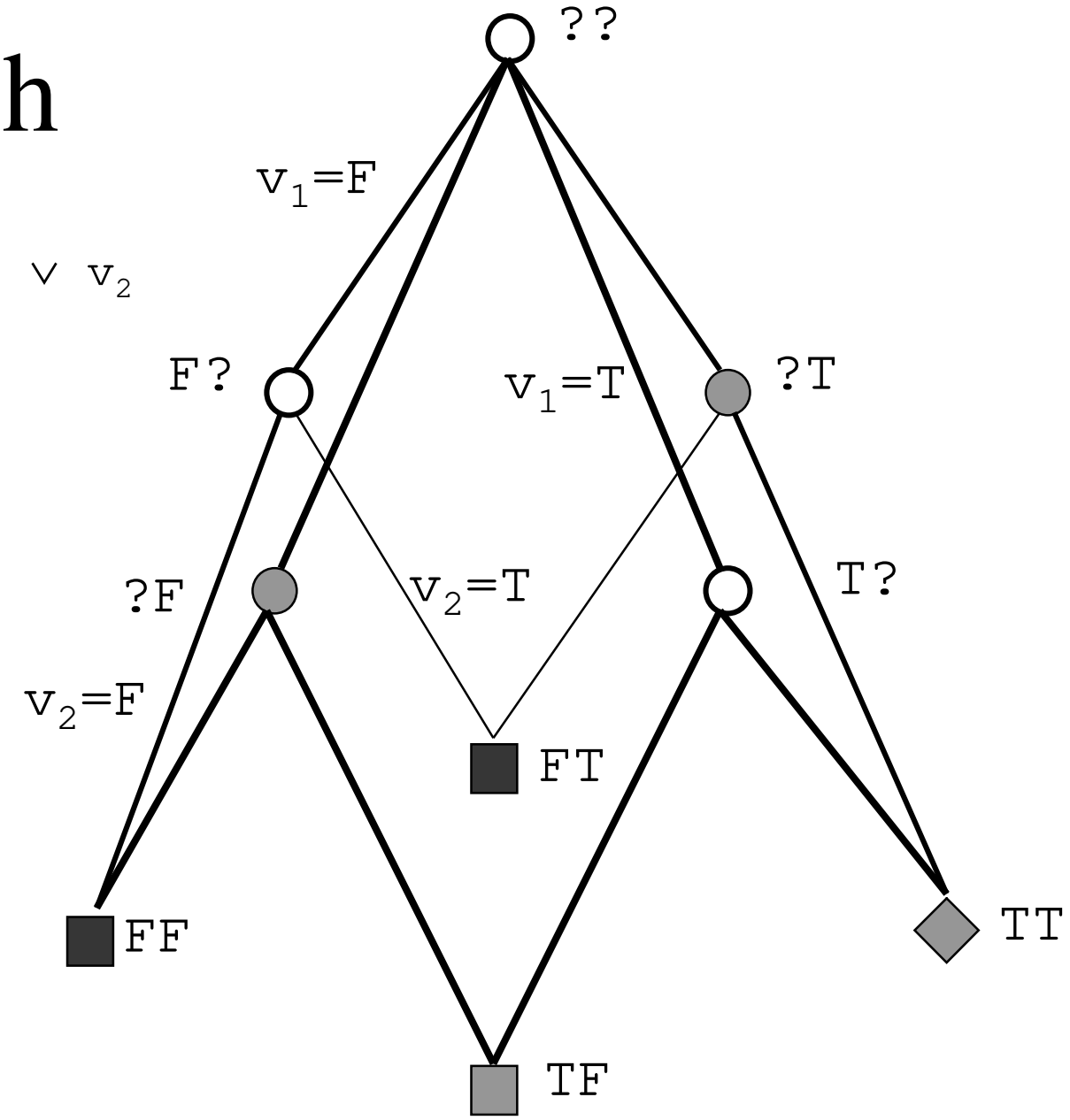
Global Search

$\overline{v_1} \vee v_2, v_1 \vee \overline{v_2}, v_1 \vee v_2$



Global Search

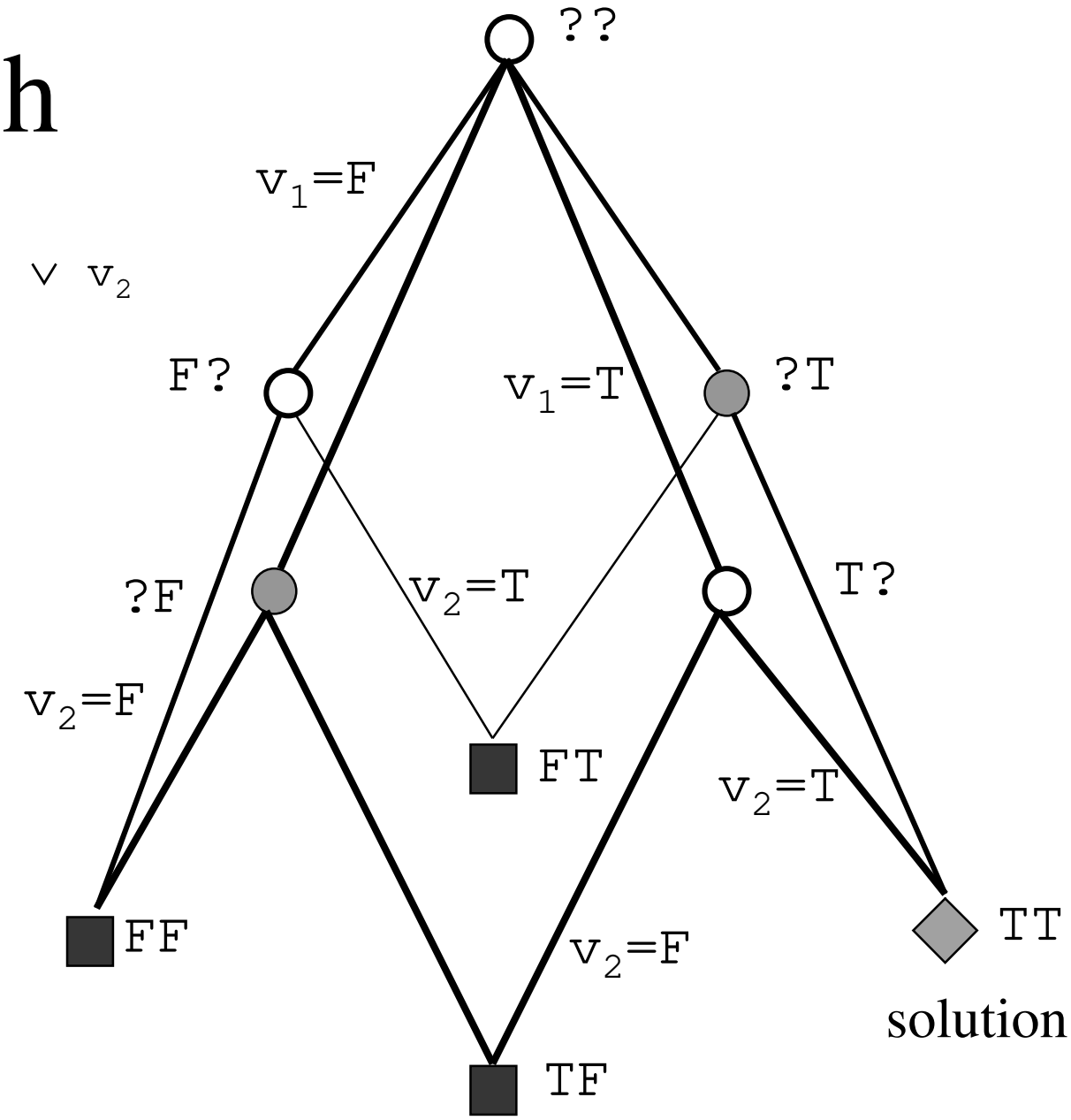
$\overline{v_1} \vee v_2, v_1 \vee \overline{v_2}, v_1 \vee v_2$



Global Search

$\overline{v_1} \vee v_2, v_1 \vee \overline{v_2}, v_1 \vee v_2$

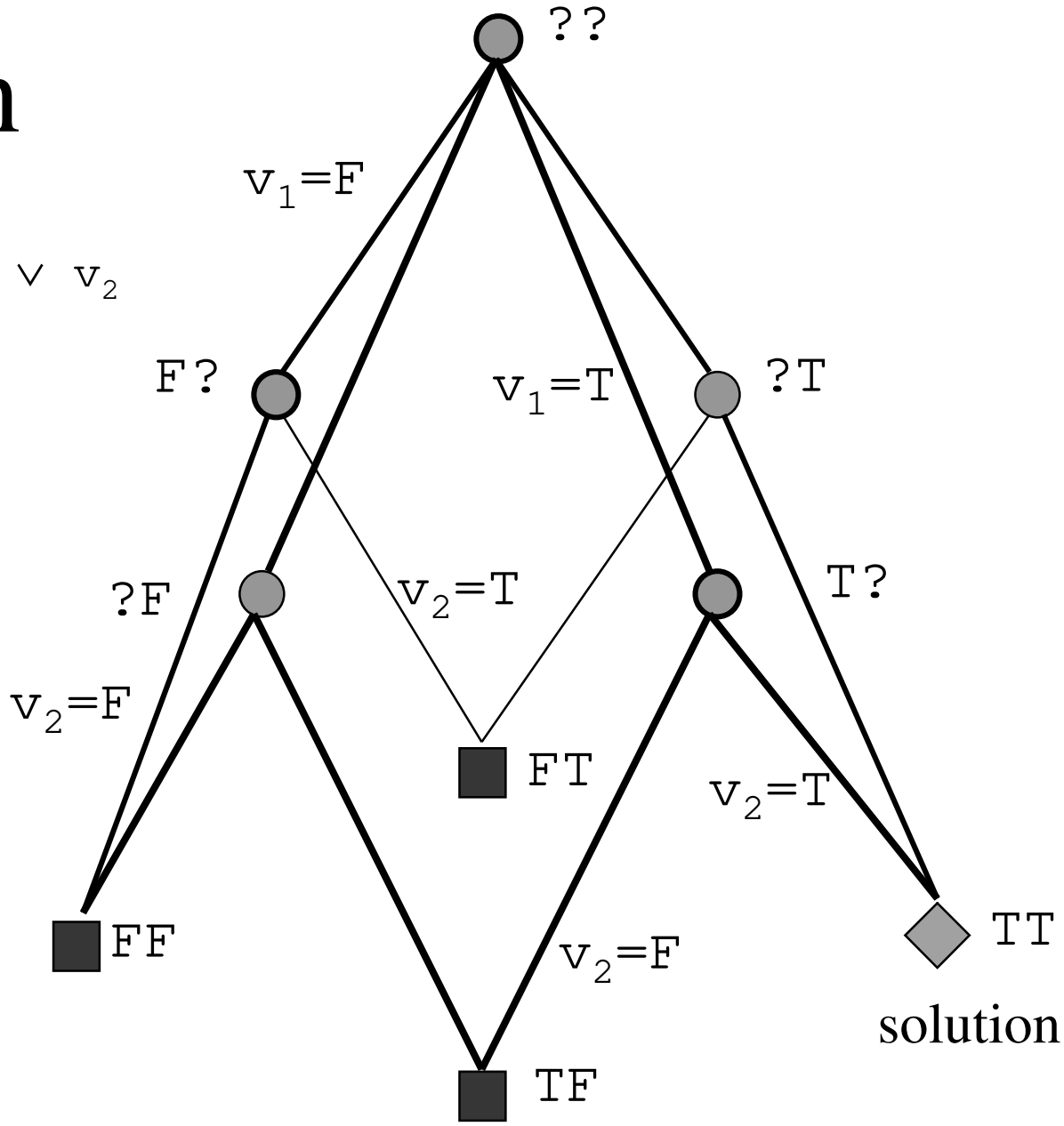
Search Tree



Local Search

$\overline{v_1} \vee v_2, v_1 \vee \overline{v_2}, v_1 \vee v_2$

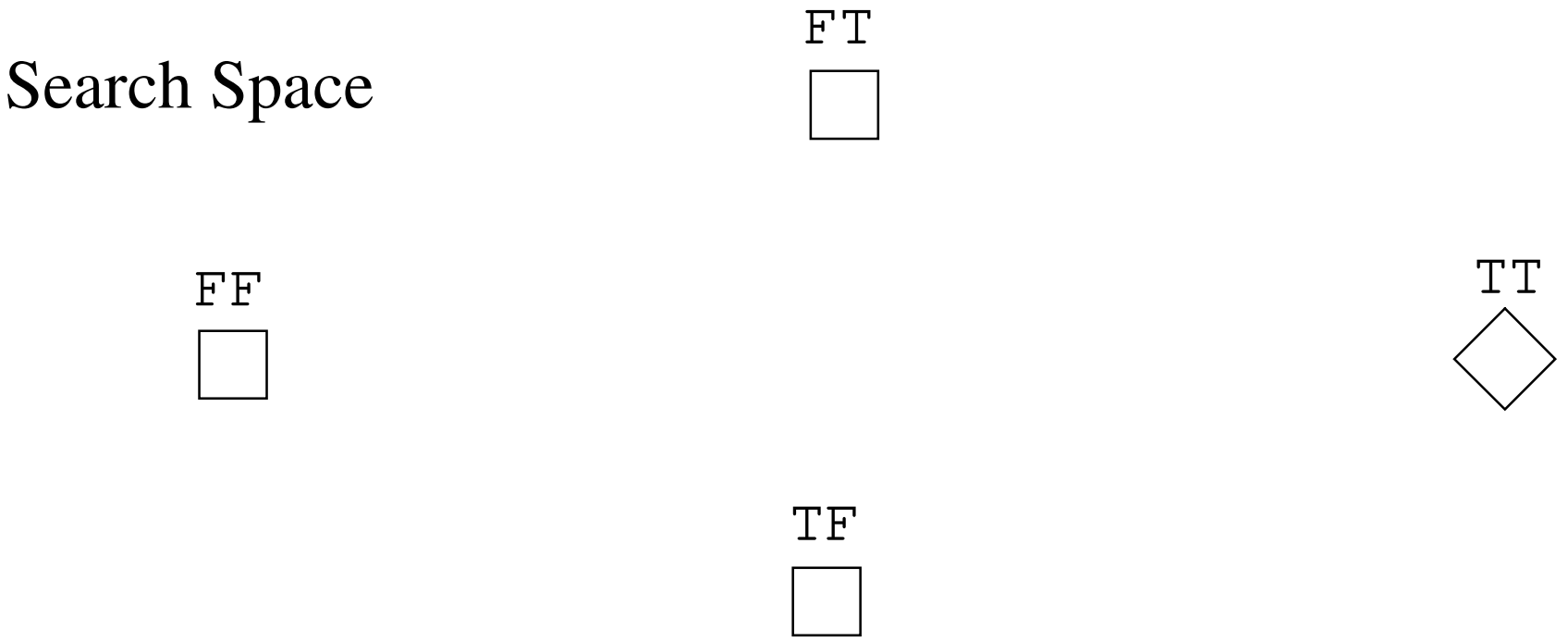
consider only full assignments of variables to values



Local Search

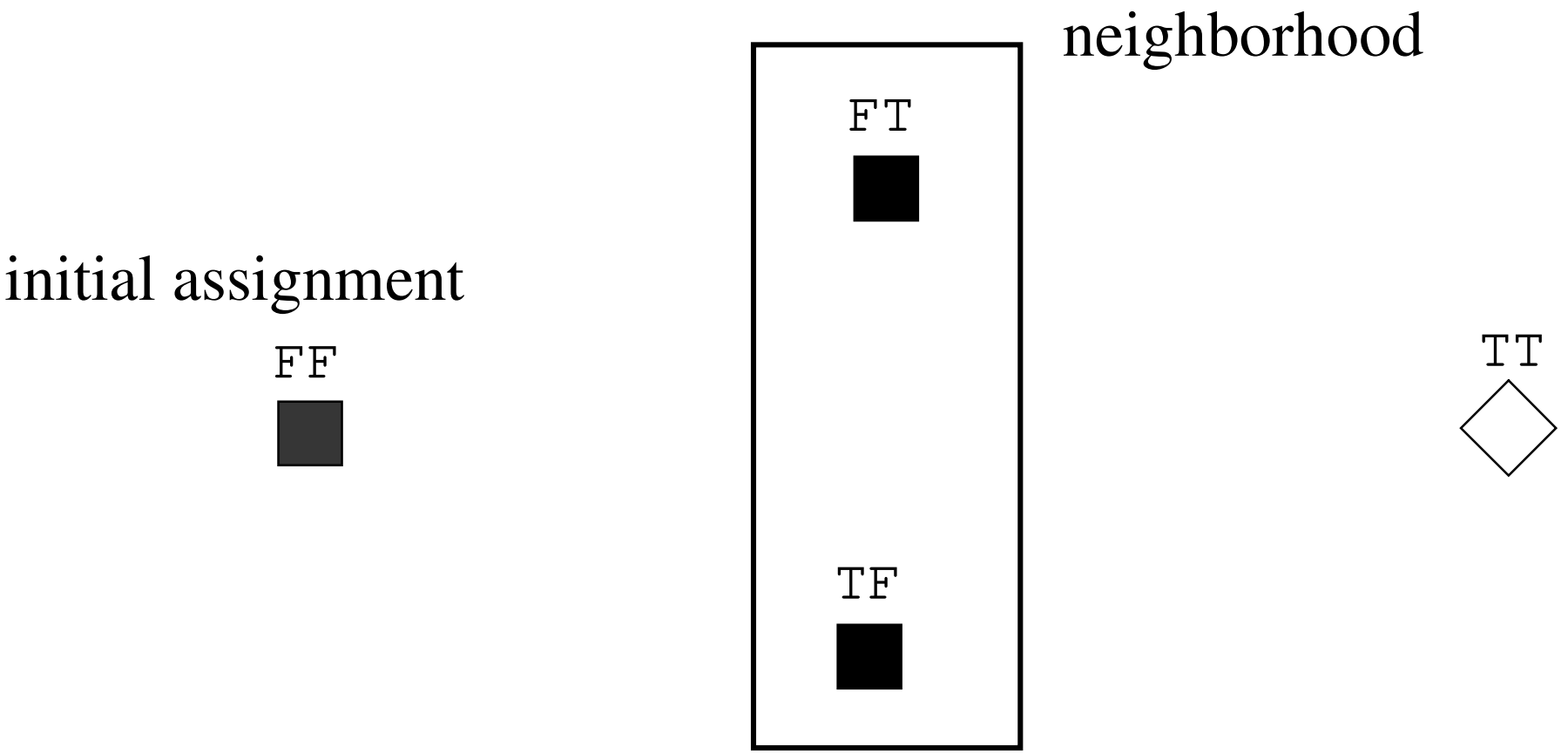
$\overline{v_1} \vee v_2, v_1 \vee \overline{v_2}, v_1 \vee v_2$

Search Space



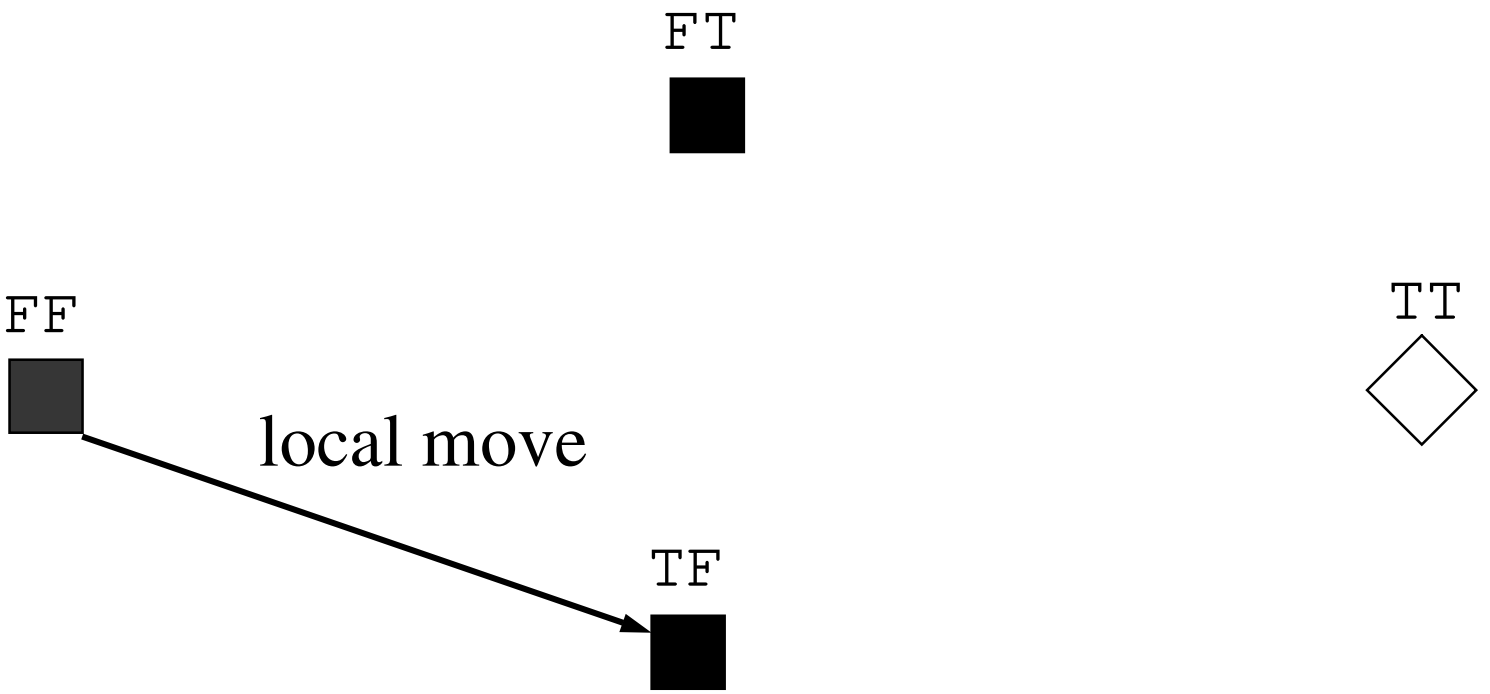
Local Search

$\overline{v_1} \vee v_2, v_1 \vee \overline{v_2}, v_1 \vee v_2$



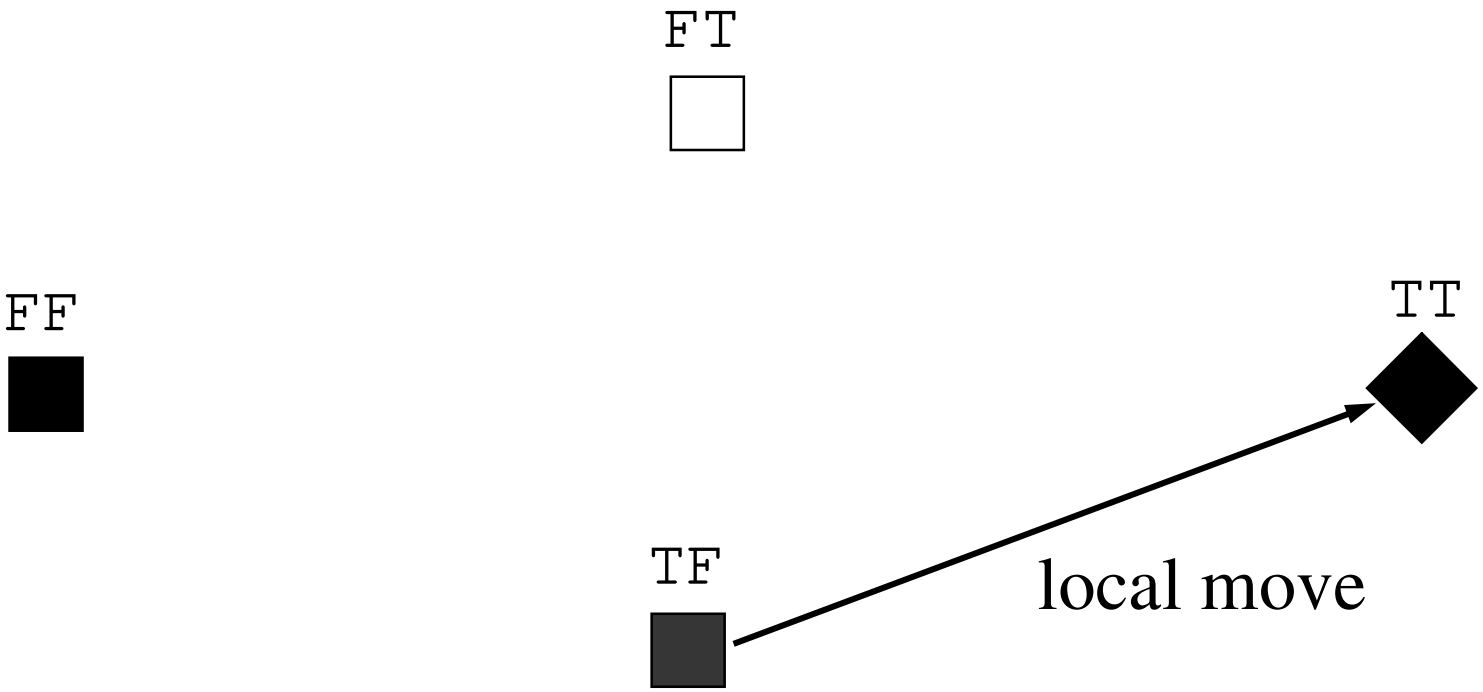
Local Search

$\overline{v_1} \vee v_2, v_1 \vee \overline{v_2}, v_1 \vee v_2$



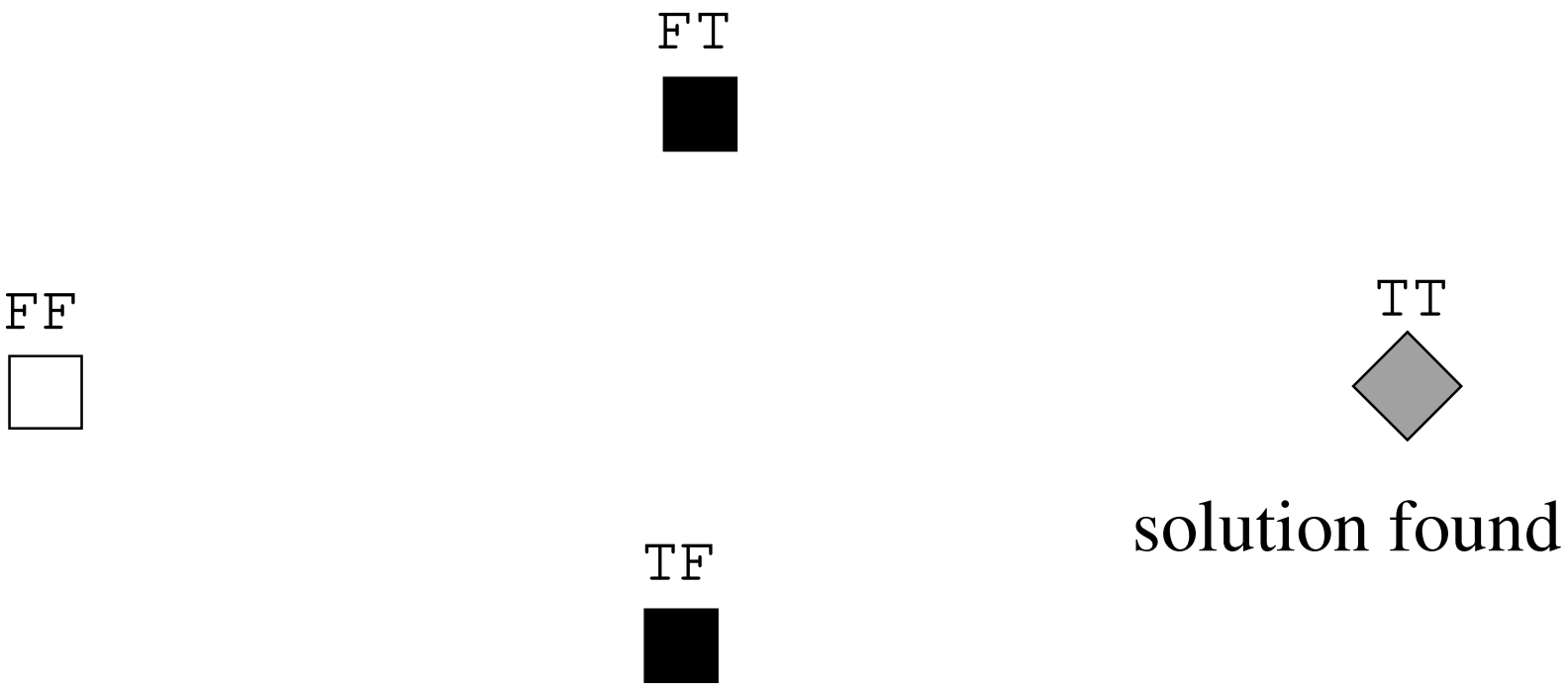
Local Search

$\overline{v_1} \vee v_2, v_1 \vee \overline{v_2}, v_1 \vee v_2$



Local Search

$\overline{v_1} \vee v_2, v_1 \vee \overline{v_2}, v_1 \vee v_2$



Observations

- Data structures needed for constraint programming are the same as in local search.
- Integration of CP and LS has been approached before.
  - Stephen Won, MS thesis supervised by Jimmy Lee (1997)

Stepping Back (for good)

- Viewing constraint programming as a collection of design patterns.
- Flexible memory policy through indirect addressing
- Figaro initiative:
  - library for discrete problem solving and optimization
  - local search and constraint programming (LP,IP?)
  - designed by researchers for researchers
  - open source, current contributors: N U Singapore, partners: CUHK, PSL

Figaro (very last slide)

- Figaro, an offspring of Mozart
- F I nite domain

  GA dget for combinatorial

  p RO blem solving