Spaces and Search

Constraint Programming, Lecture 6

Marco Kuhlmann, Guido Tack

The story so far

- modelling constraint satisfaction problems using GeCoDE and Alice
- formal model for solving constraint satisfaction problems
- implementing the propagation rule (propagation loop, propagator properties)

Propagation

$$\frac{\langle X \cup \{x : D\} \,;\, \mathfrak{C} \cup \{C\}\rangle \qquad d \in D \qquad \mathsf{sat}(C) \cap \{\, \alpha \in \mathsf{ass}(X) \mid \alpha x = d \,\} = \emptyset}{\langle X \cup \{x : D - \{d\}\} \,;\, \mathfrak{C} \cup \{C\}\rangle}$$

$$\frac{\langle X \cup \{x : \emptyset\} \,;\, \mathfrak{C}\rangle}{\textbf{fail}}$$

Today

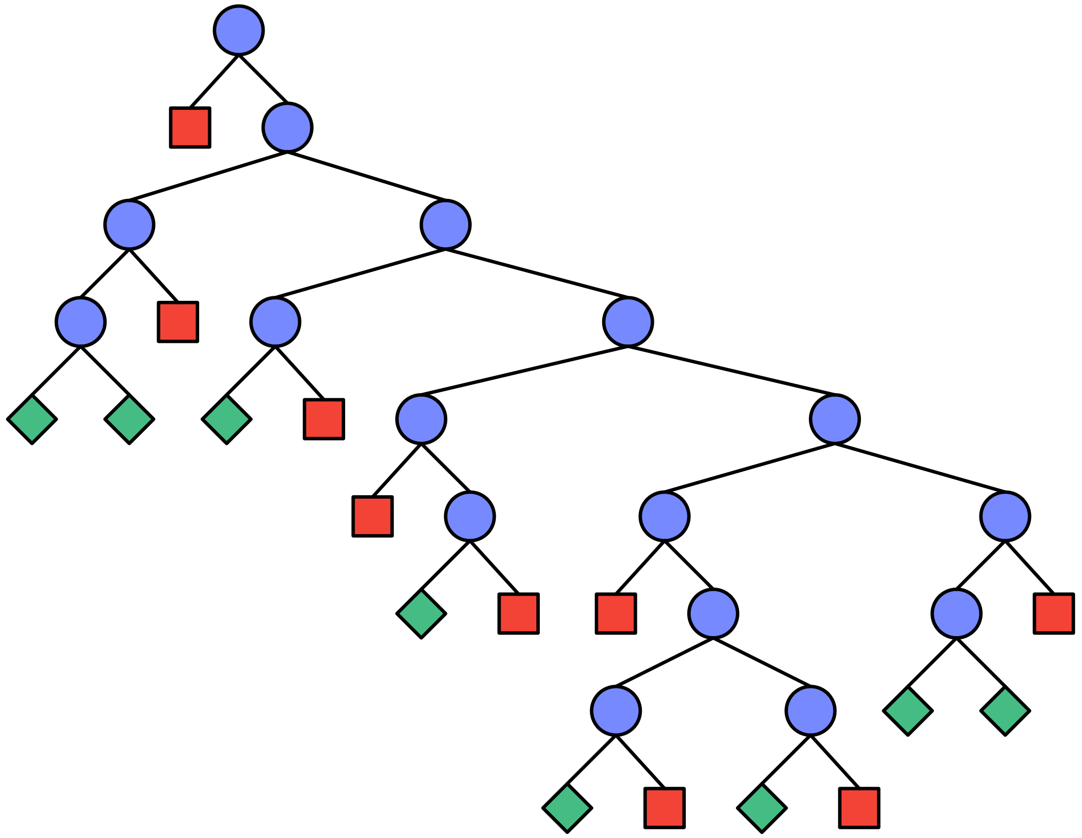
- an architecture for search
- writing simple search engines
- limited discrepancy search
- branch & bound search

Search trees

The Branching Rule

$$\frac{\langle X \cup \{x : D\}\ ;\ \mathfrak{C}\rangle \qquad |D| > 1 \qquad D = D_1 \uplus \cdots \uplus D_k}{\langle X \cup \{x : D_1\}\ ;\ \mathfrak{C}\rangle \mid \cdots \mid \langle X \cup \{x : D_k\}\ ;\ \mathfrak{C}\rangle}$$

Search tree



The Branching Rule

$$\frac{\langle X \cup \{x : D\} \,;\, \mathfrak{C}\rangle \qquad |D| > 1 \qquad D = D_1 \uplus \cdots \uplus D_k}{\langle X \cup \{x : D_1\} \,;\, \mathfrak{C}\rangle \mid \cdots \mid \langle X \cup \{x : D_k\} \,;\, \mathfrak{C}\rangle}$$

indeterministic choice

Two questions

- *How to branch?*
  - branching strategy (naive, first-fail, …)
  - determines the *shape* of the search tree
- *How to make the choice operation deterministic?*
  - search strategy (depth-first, b & b, …)
  - determines the *order* in which
    the nodes of the search tree are visited

Backtracking

- no way to predict whether a choice is good
- consequence: choices need to be undone
  - choice may not have lead to any solution
  - choice may not have yielded all solutions
- backtracking = undoing choices

Backtracking strategies

- *copying:*
  backup the state of the system
  before making a choice

- *trailing:*
  remember an … the choice

  next lecture

- *recomputation:*
  recompute the state of the system
  before the choice was made

Terminology

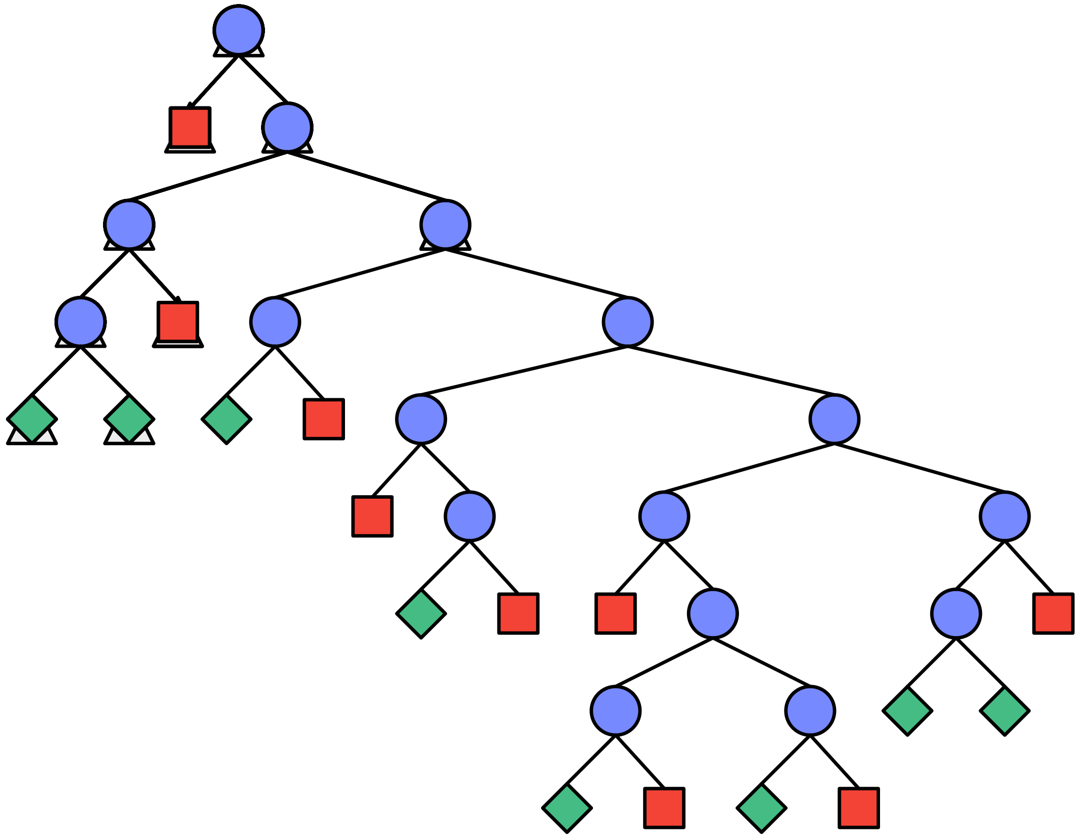
- *search strategy:*
  how to explore the search tree
- *search engine:*
  implements a search strategy, but
  may provide additional functionality:
  one or all solutions, user interaction, …

An architecture for search

Design decisions

- Prolog
  - first system to do computation by search
  - one single opaque search strategy
- Mozart (Oz) and GeCoDE
  - more than one search strategy
  - architecture for writing new search engines

Depth-First Exploration



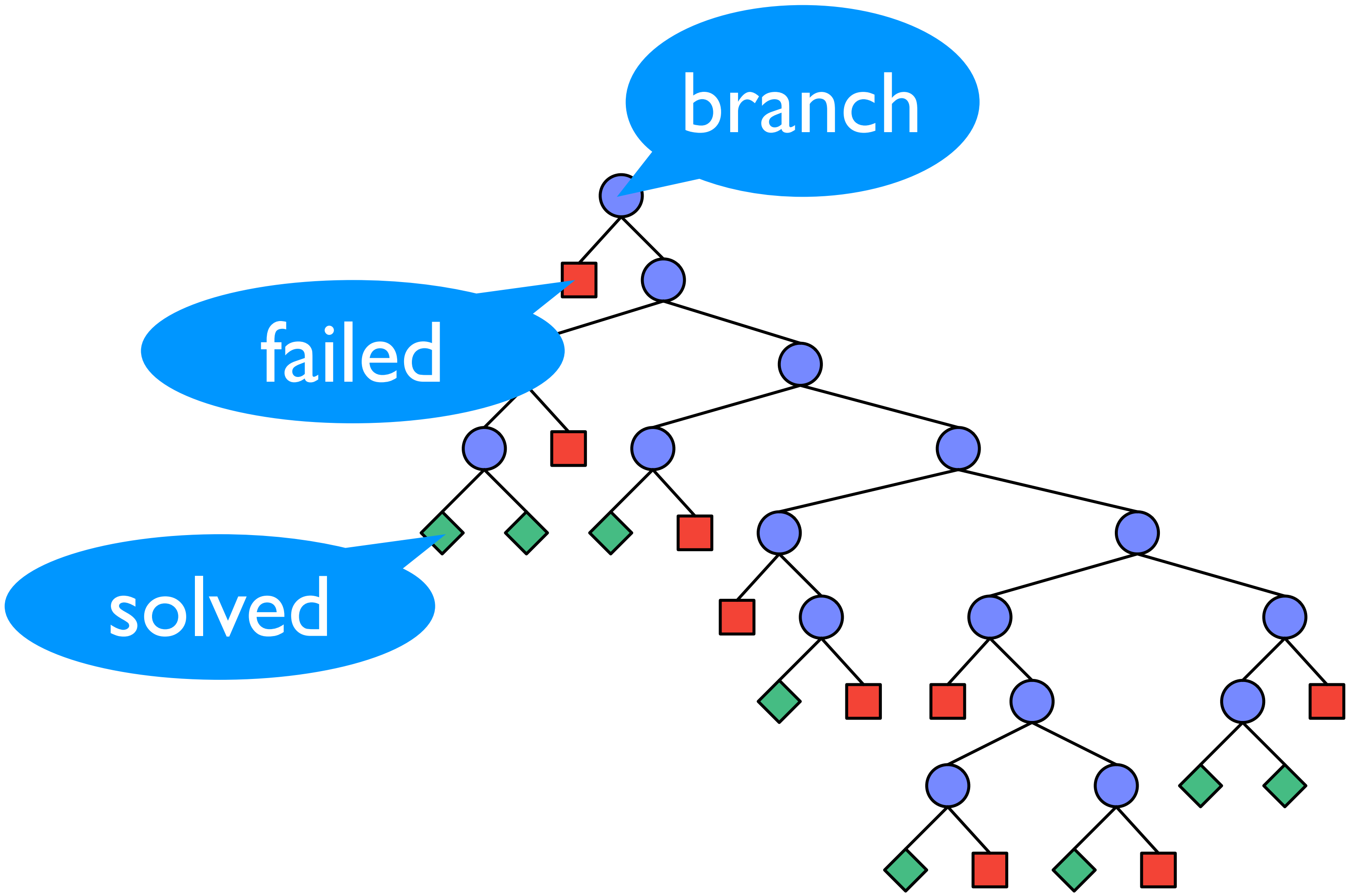
Operations on spaces

- **`status : space -> status`**
  determines the status of a space
- **`clone : space -> space`**
  returns a backup clone of a space
- **`commit : space * int -> unit`**
  commit a space to one of its alternatives

Status messages

- *failed* –
  the variable domains are inconsistent
- *solved* –
  the variable domains form an assignment
- *branch* –
  the variable domains require branching

Status messages



Implementing DFS

```
fun dfs (s) =
  case status s of
    FAILED => nil
  | SOLVED => [s]
  | BRANCH =>
      let val c = clone s in
        commit (s, 1);
        commit (c, 2);
        dfs s @ dfs c
      end
```

all solutions

One-solution search

```
exception Solved of space

fun dfs (s) =
  case status s of
    FAILED => ()
  | SOLVED => raise Solved(s)
  | BRANCH =>
      let val c = clone s in
        commit (s, 1); dfs s;
        commit (c, 2); dfs c
      end
```

Explicit agenda

```
fun dfs nil   = ()
  | dfs s::ss =
    case status s of
      FAILED => dfs ss
    | SOLVED => raise Solved(s)
    | BRANCH =>
        let val c = clone s in
          commit (s, 1);
          commit (c, 2);
          dfs s::c::ss
        end
```

```
fun gs a =
  if empty a then () else
    let val s = get a in
      case status s of
        FAILED => gs a
      | SOLVED => raise Solved(s)
      | BRANCH =>
          let val c = clone s in
            commit (s, 1);
            commit (c, 2);
            gs (put [s, c] a)
          end
    end
```

Generic search

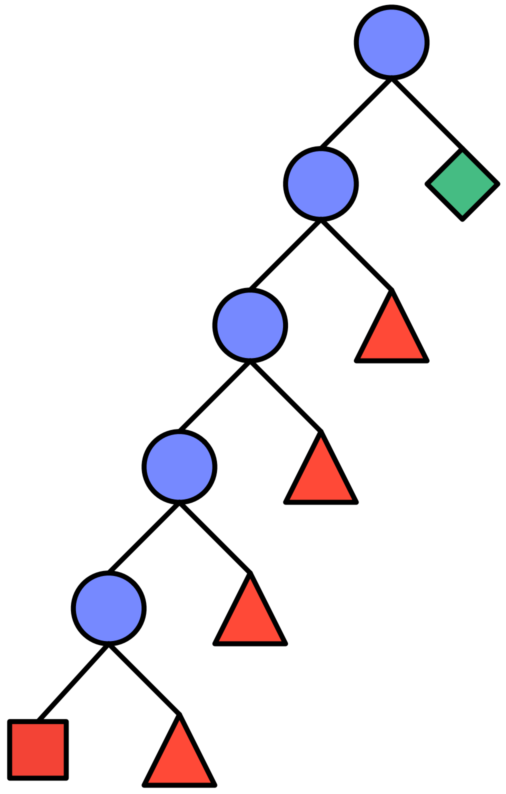
- *depth-first search:*
  agenda is a list
- *breadth-first depth:*
  agenda is a queue
- *best-first search:*
  agenda is a priority queue

Limited Discrepancy Search (LDS)

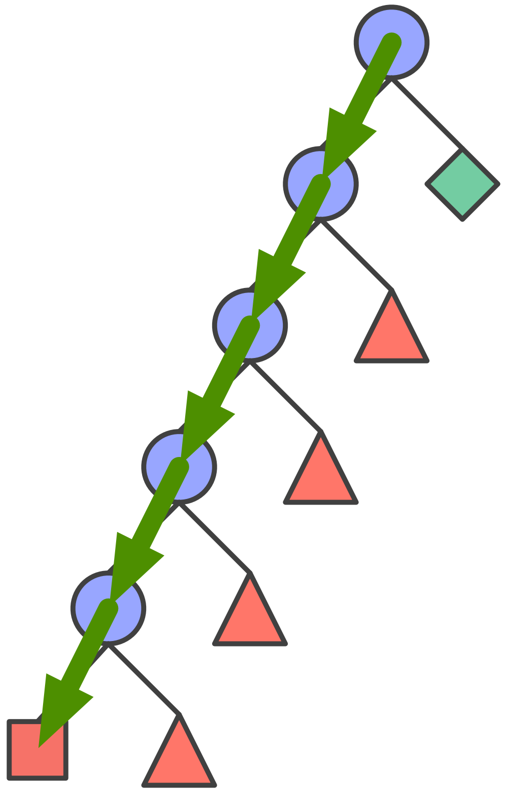
Motivation

- Branching strategies are often designed to put good alternatives first.
- But sometimes violating this heuristic pays off.
- *Limited discrepancy search* is a search strategy that allows a limited number of violations of the heuristic, *discrepancies*.

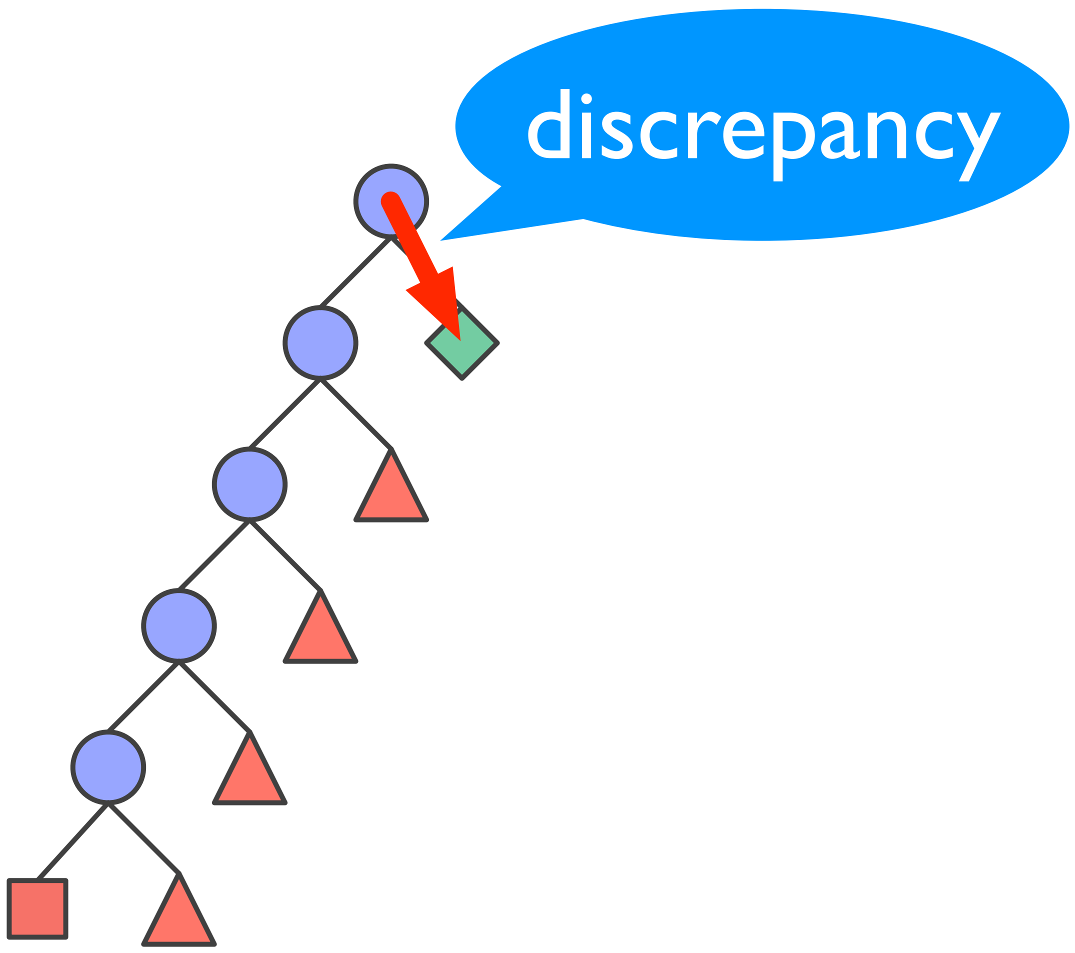
Example



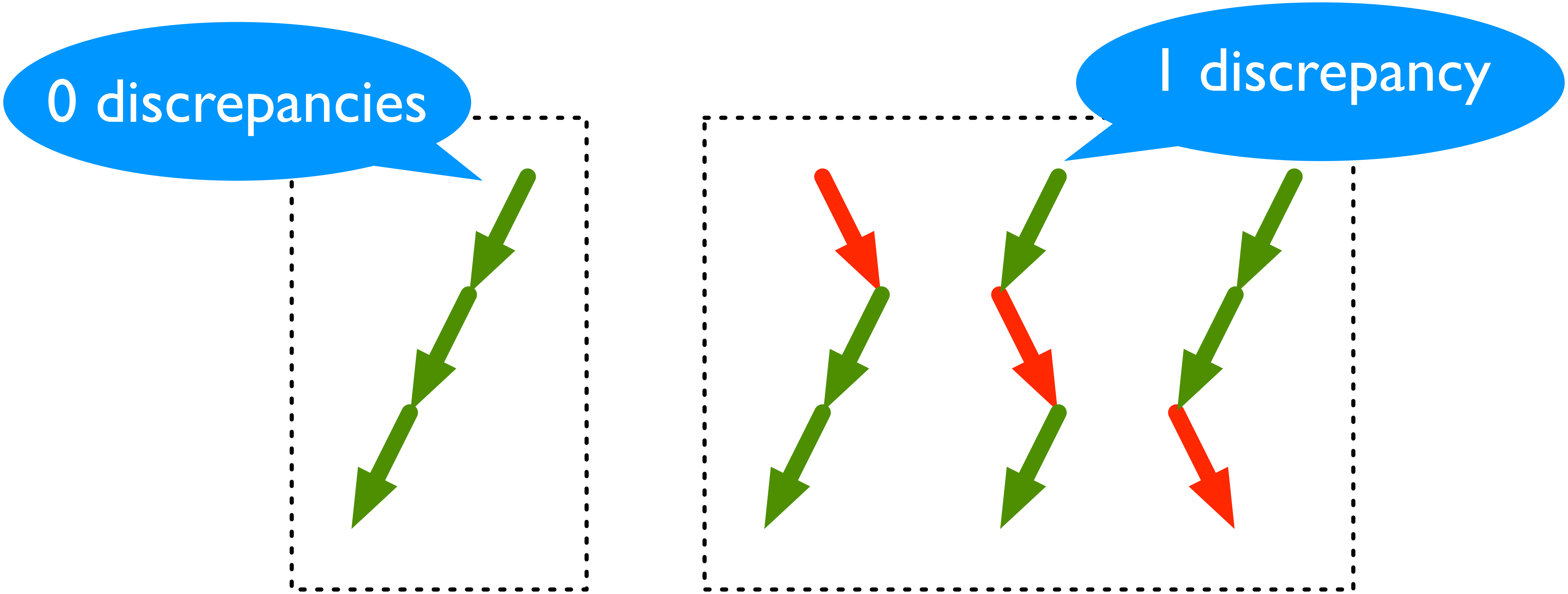
Example



Example



Probing



```
fun probe (s, d) =
  case status s of
    FAILED => ()
  | SOLVED => raise Solved(s)
  | BRANCH =>
      if d > 0 then
        let val c = Space.clone s in
          commit (s, 2); probe (s, d-1);
          commit (c, 1); probe (c, d)
        end
      else
        commit (s, 1); probe (s, 0)
```

LDS as best-solution search

- the less discrepancies, the better the solution
- LDS finds solutions with
  fewer discrepancies first: best solution search
- example: allocating students to tutorials

Branch & Bound Search

Motivation

- optimisation problems are ubiquitous
- not feasible to explore the
  complete tree and look for optimal solution
- idea: use previously found solutions
  to prune the search tree

Remember: SMM+

needs to be a solution

```
fun constrain (s, r) =
  let
    val rmoney = Reflect.value (r, money)
  in
    post (s, FD(money) `> `rmoney, FD.BND)
  end
```

```
fun babs (s, best) =
  case status s of
    FAILED => best
  | SOLVED => s
  | BRANCH =>
      let
        val c = clone s
      in
        commit (s, 1);
        commit (c, 2);
        let
          val better = babs (s, best)
        in
          constrain (c, better);
          babs (c, better)
        end
      end
```

Summary

- separate propagation and branching from search
- components of the architecture interact
- spaces provide an architecture for writing search engines
- simple primitives, complex search engines