Spaces and Search

Marco Kuhlmann & Guido Tack
Lecture 6

The story so far

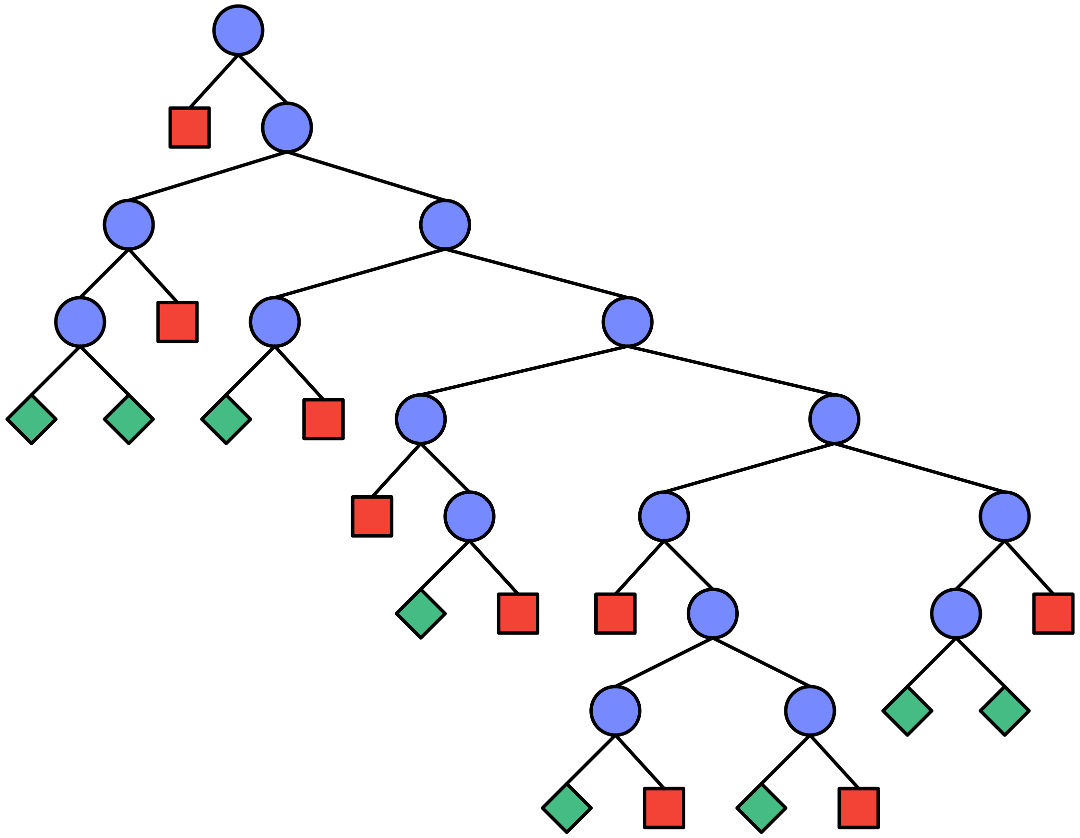
- modelling constraint satisfaction problems using Gecode/J
- formal model for solving constraint satisfaction problems
- implementation of propagators

Today

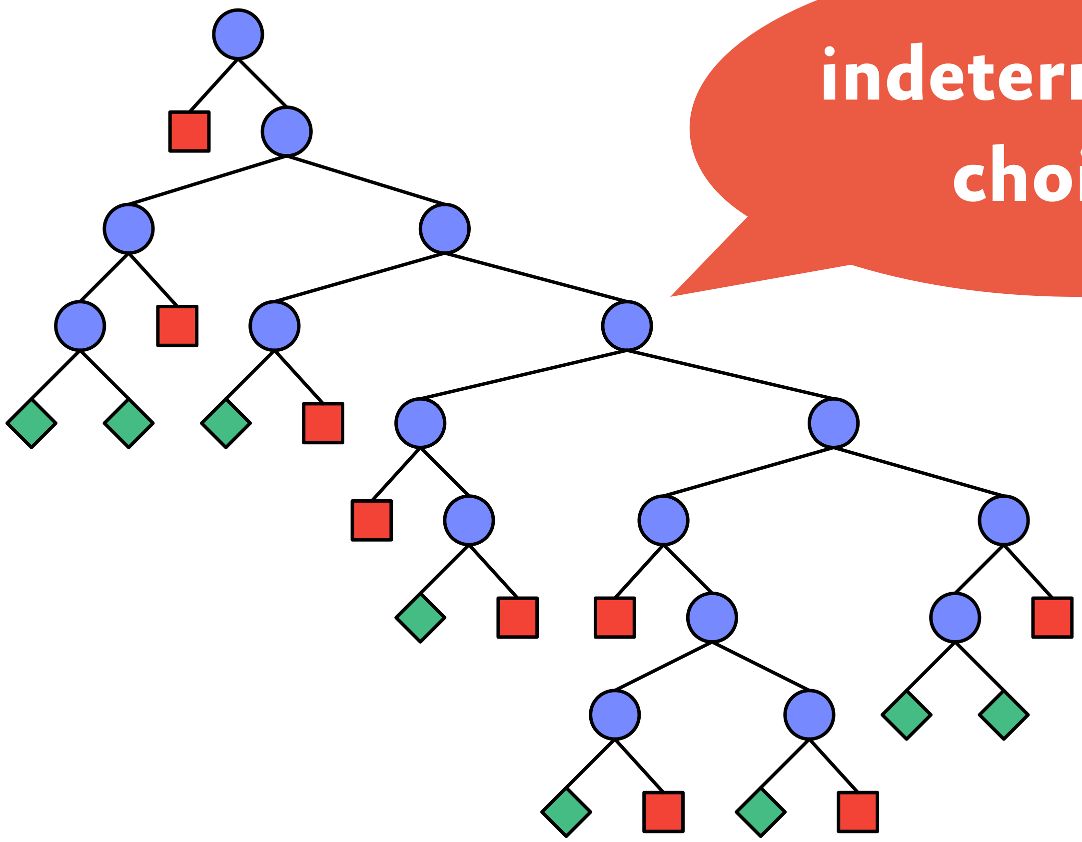
- an architecture for search
- writing simple search engines
- limited discrepancy search
- branch & bound search
- recomputation

Search trees

Search tree



Search tree



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, branch & bound, ...)
  - determines the computation order

Two questions

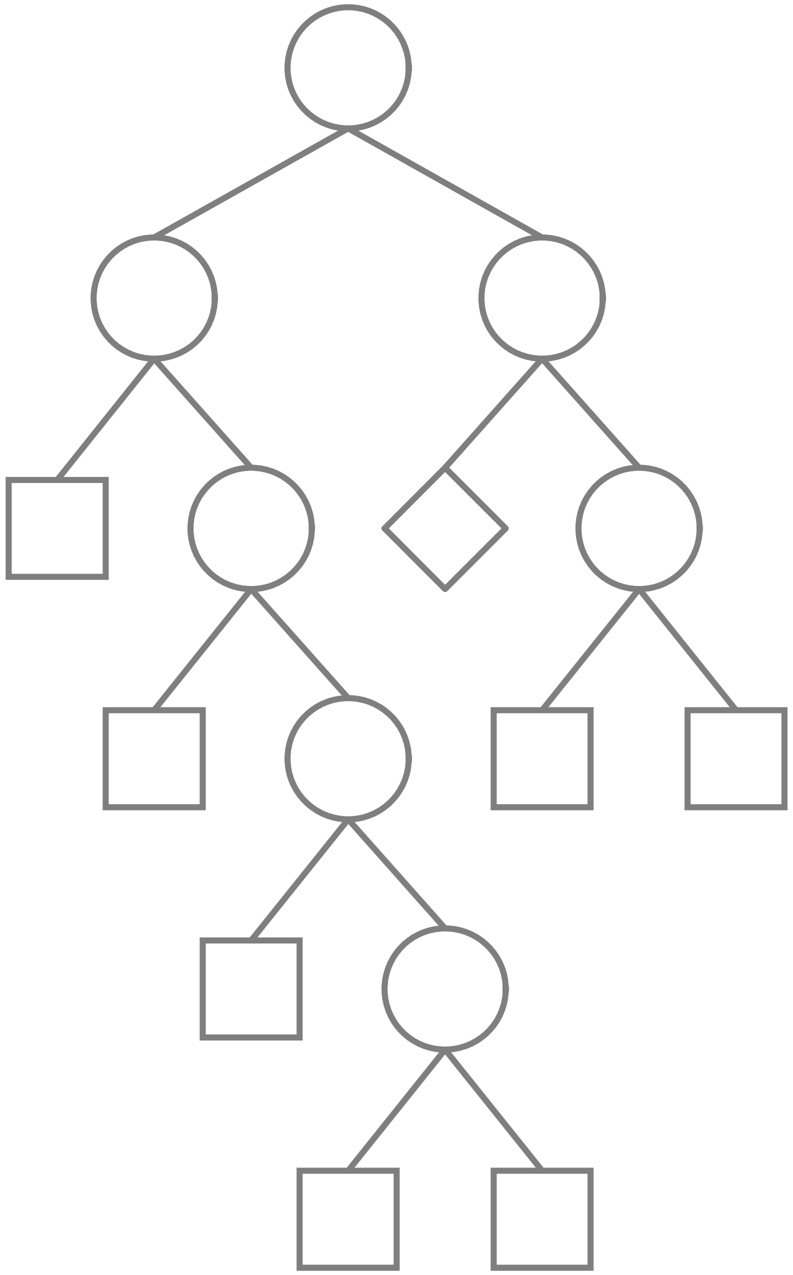
simplification:
binary branching

- **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, branch & bound, ...)
  - determines the computation order

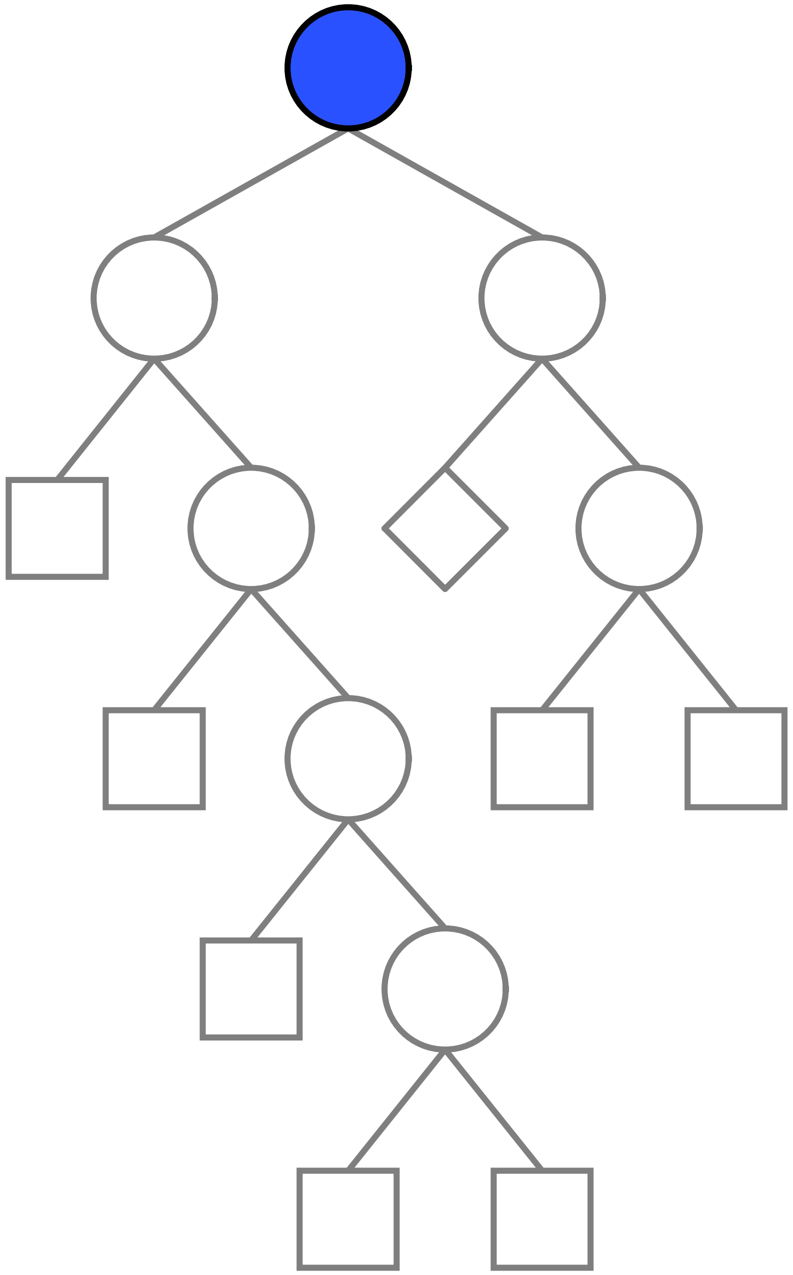
Backtracking

- **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**

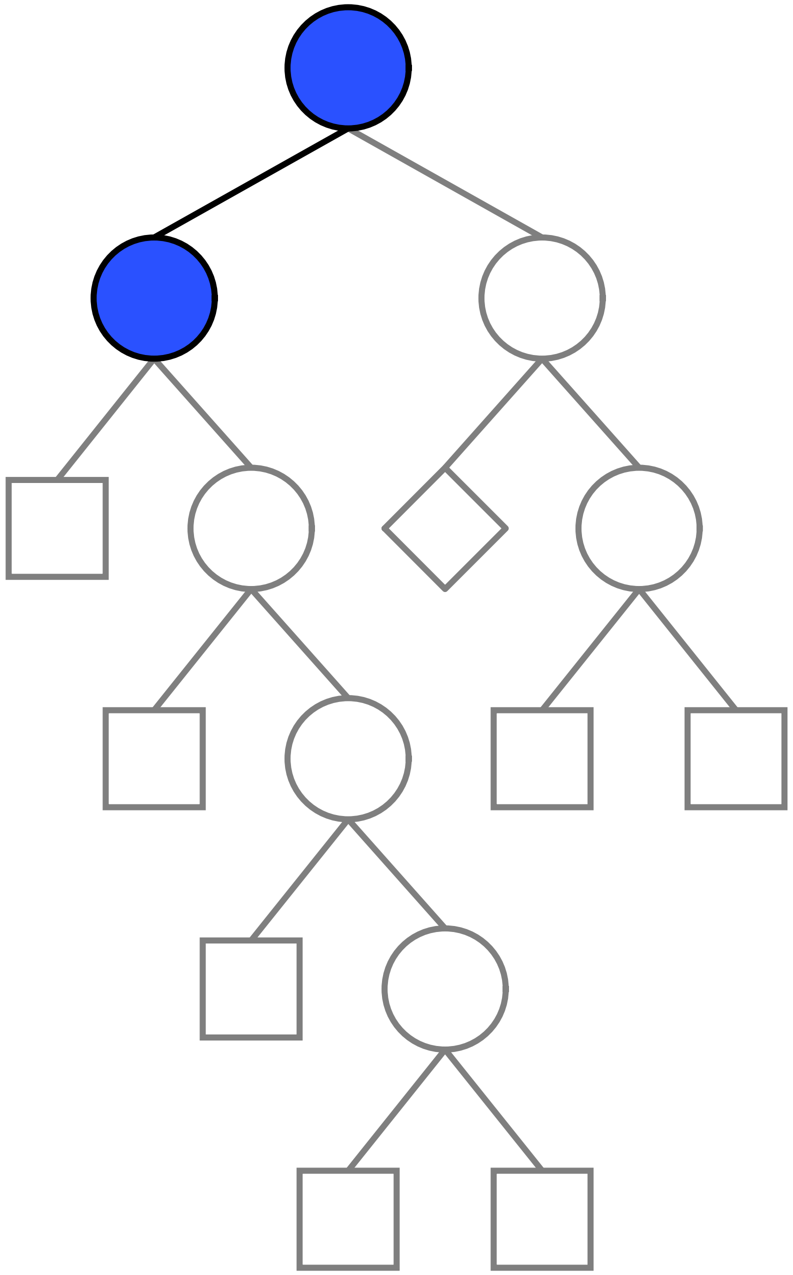
Backtracking



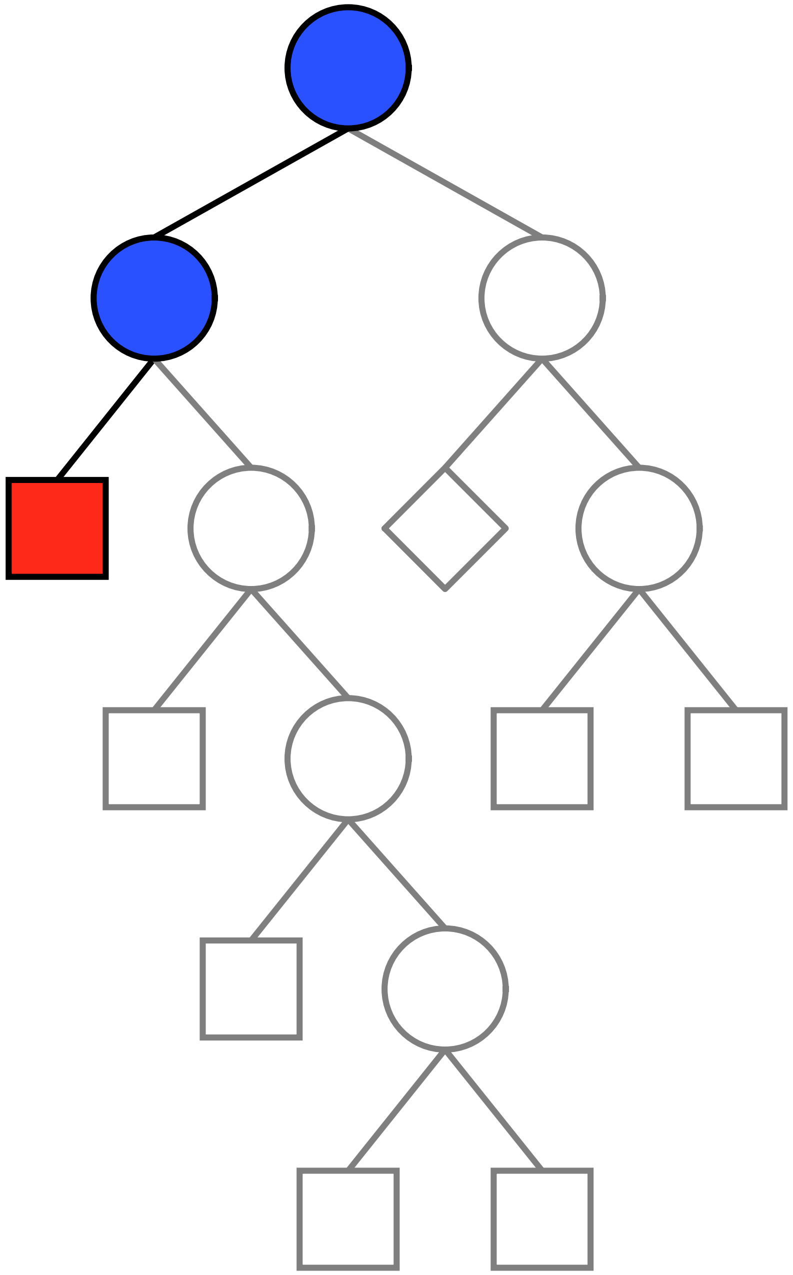
Backtracking



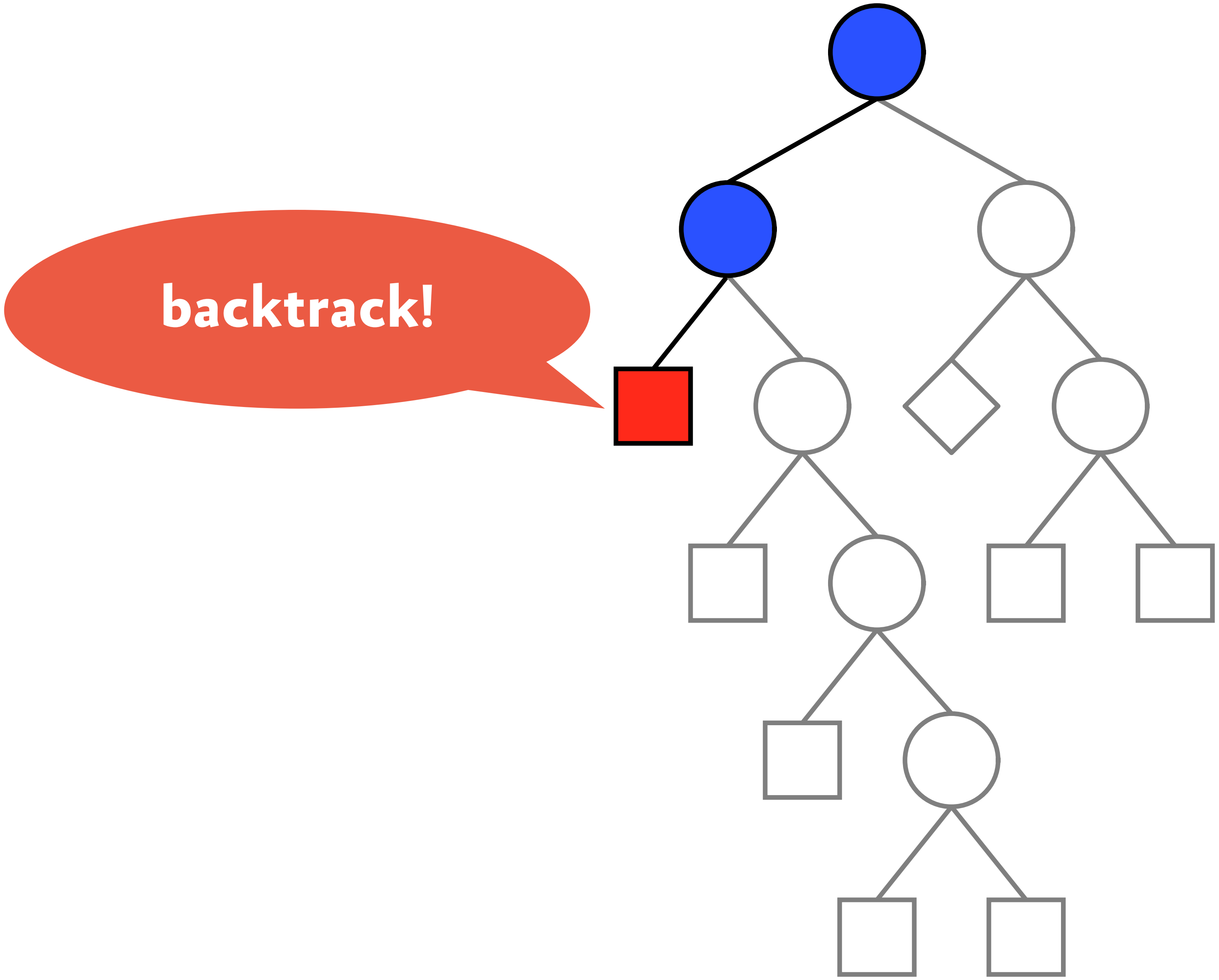
Backtracking



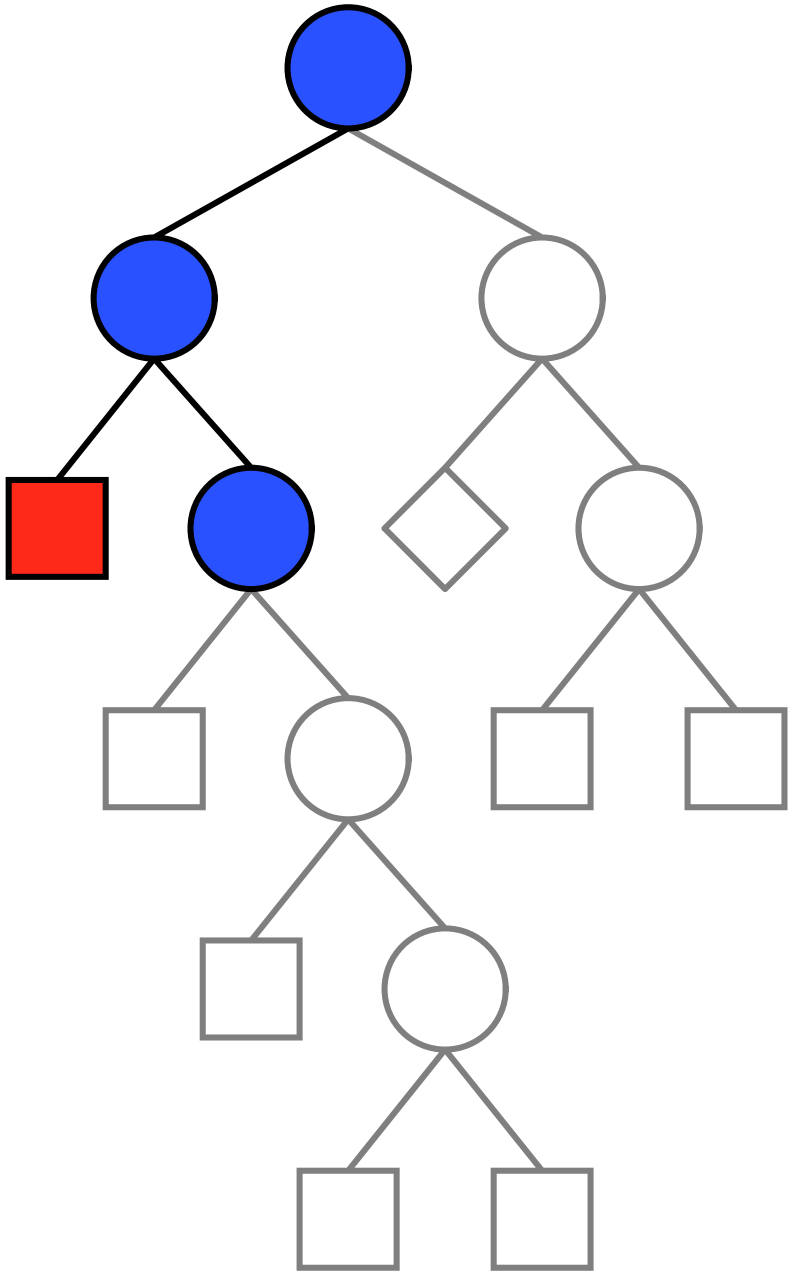
Backtracking



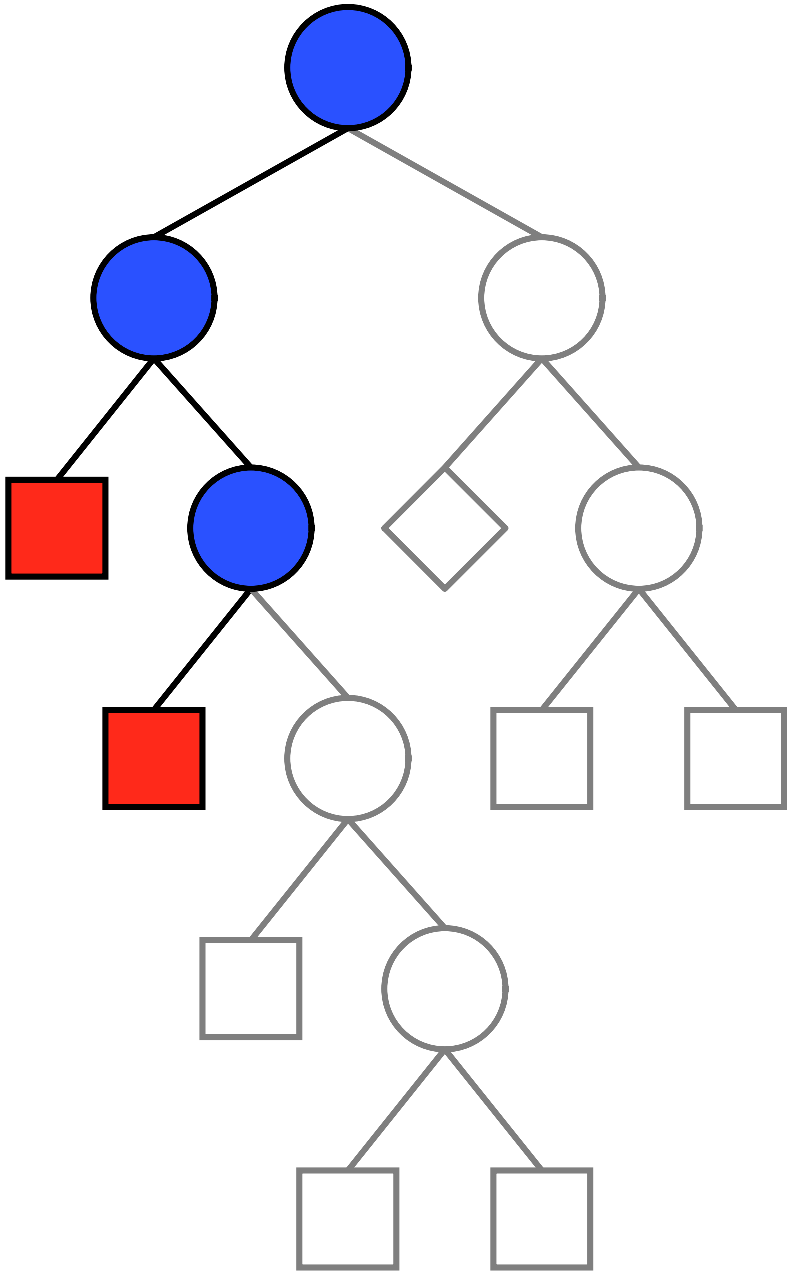
Backtracking



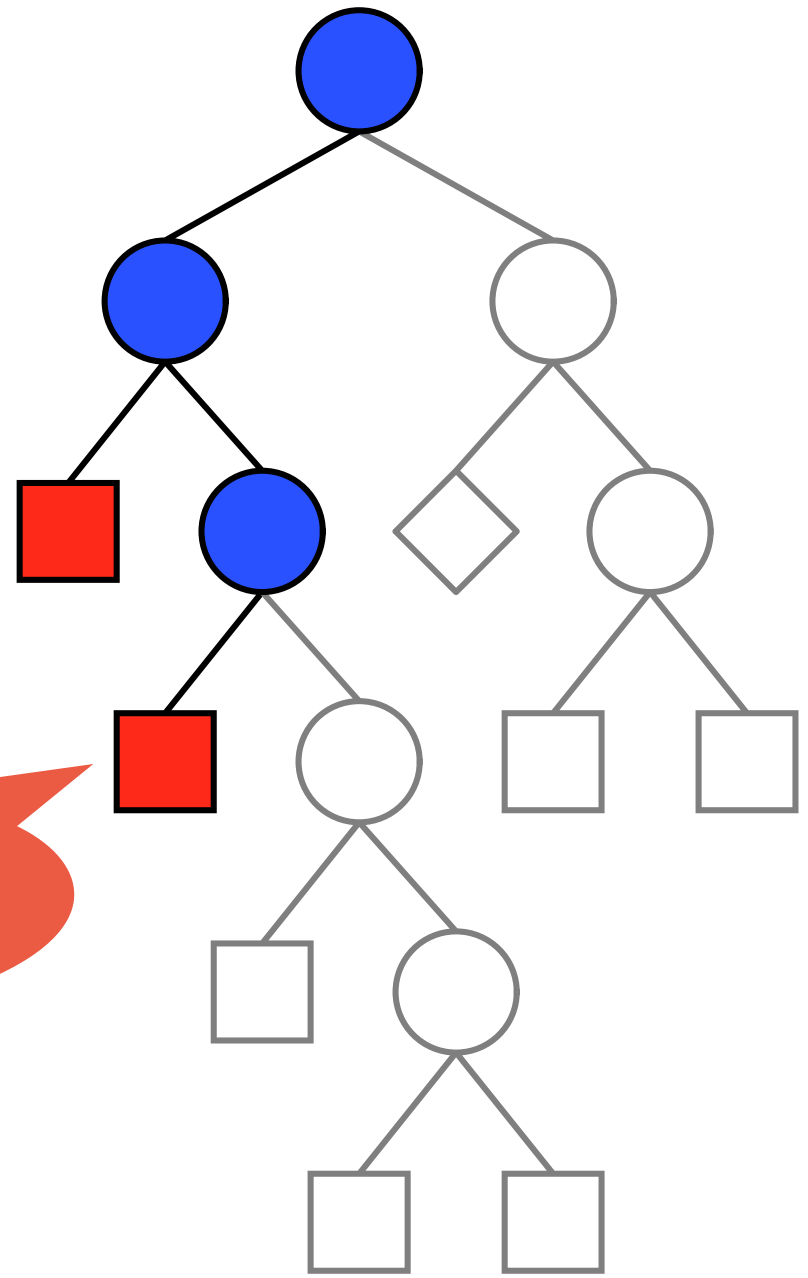
Backtracking



Backtracking

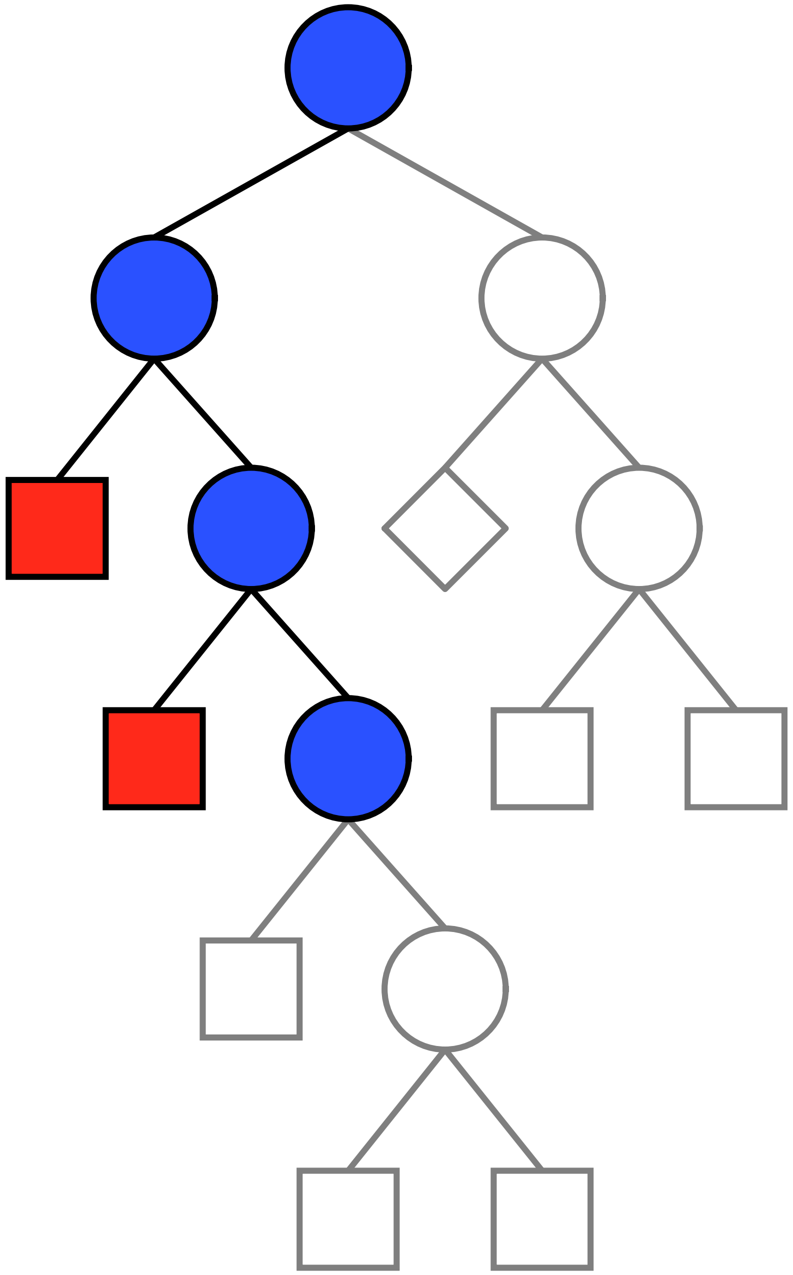


Backtracking

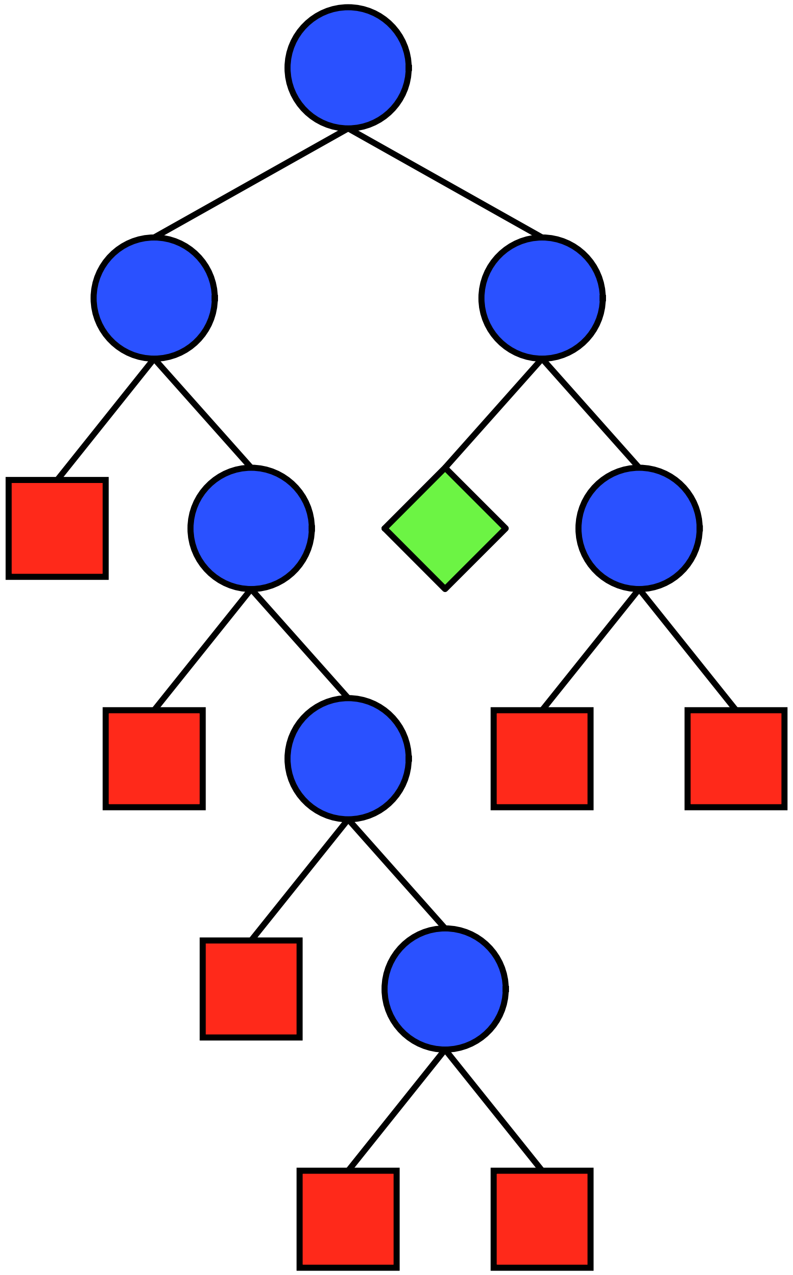


backtrack!

Backtracking



Backtracking



Backtracking strategies

- **copying:**
  backup the state of the system
  before making a choice
- **trailing:**
  remember an undo action for the choice
- **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,
  may provide additional functionality
  (one or all solutions, user interaction, ...)

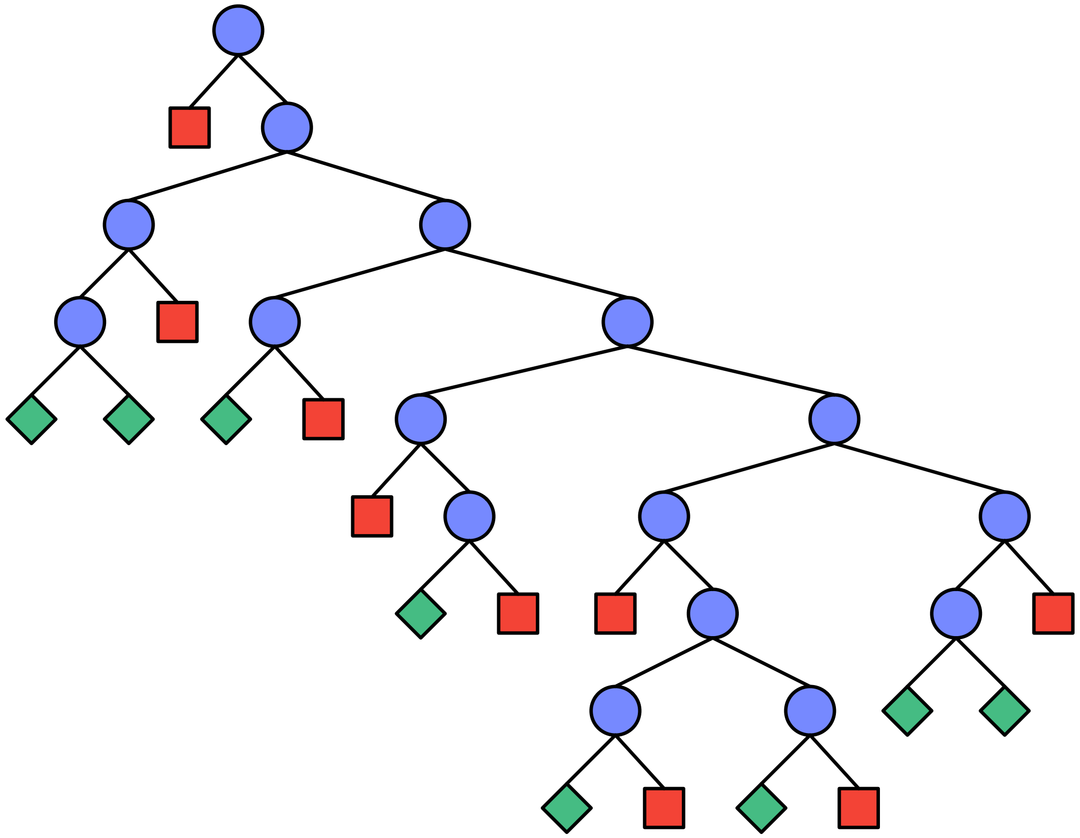
An architecture for search

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

Depth-First Exploration



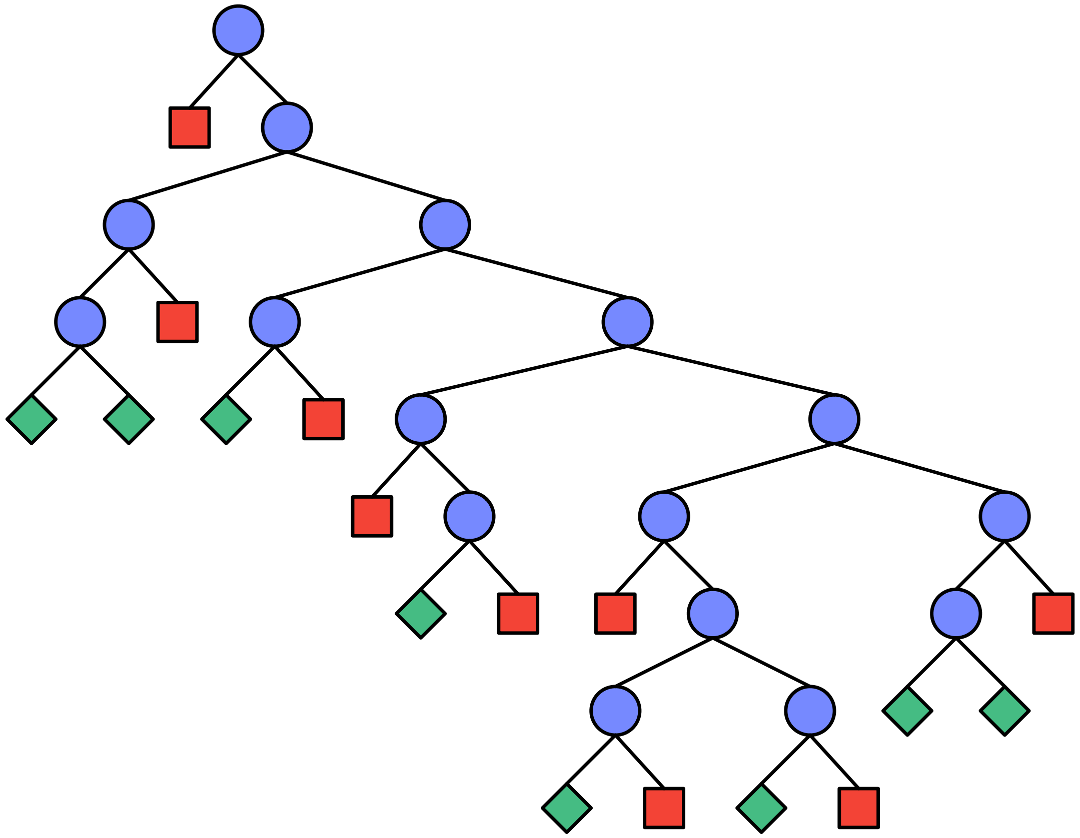
Operations on spaces

- **SpaceStatus status()**
  determines the status of a space (failed, solved, branch)
- **Space cloneSpace()**
  returns a backup clone of a space
- **void commit(long alternative)**
  commit a space to one of its alternatives

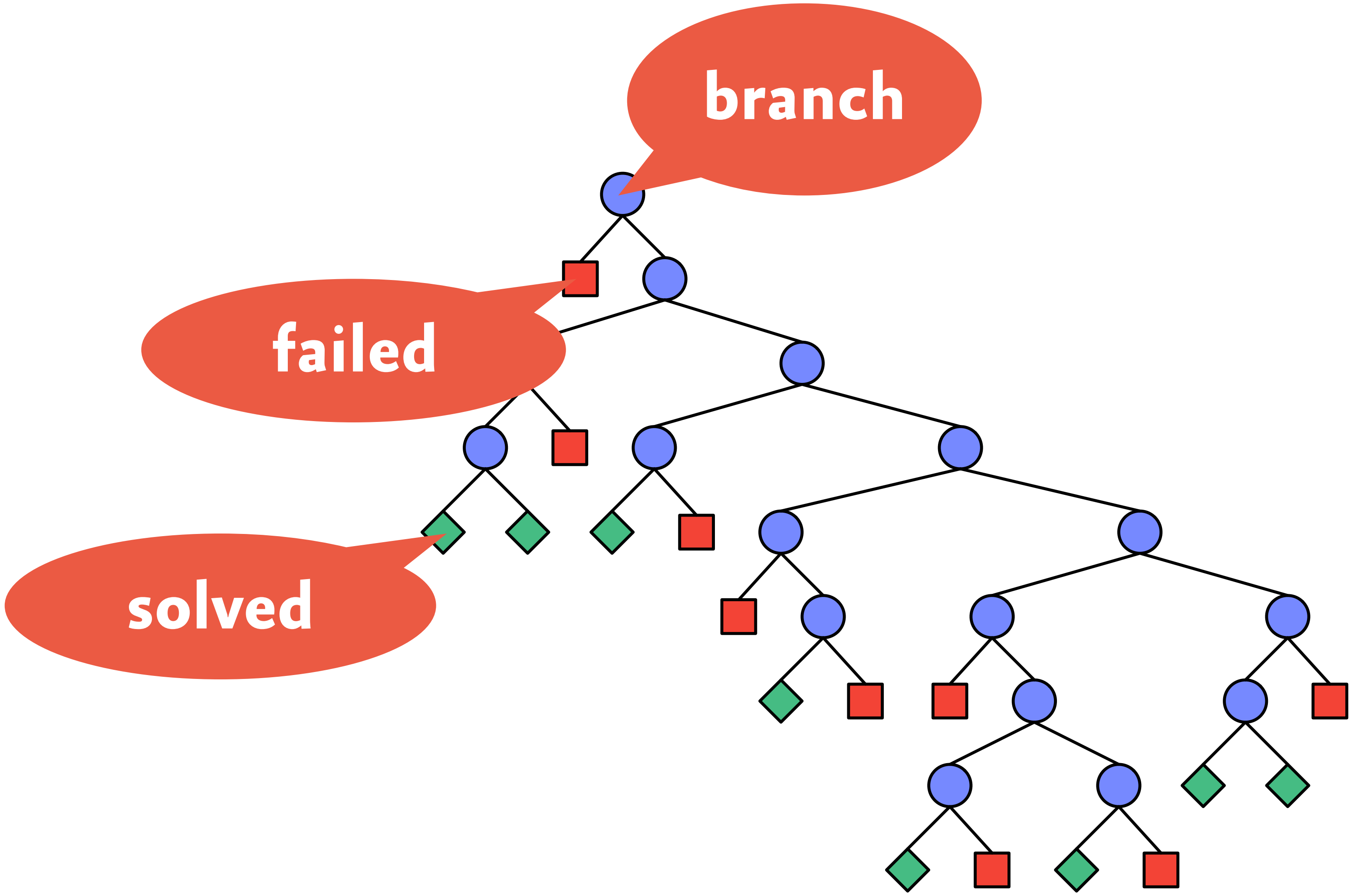
Status messages

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

Status messages



Status messages



Implementing DFS

```
public static Space dfs(Space space) {
  switch (space.status()) {
    case SS_FAILED: return null;
    case SS_SOLVED: return space;
    case SS_BRANCH:
      Space c = space.cloneSpace();
      space.commit(0);
      Space s = dfs(space);
      if (s != null) {
        return s;
      } else {
        c.commit(1);
        return dfs(c);
      }}}
```

Explicit agenda (1)

```
private Stack<Space> agenda;

public DepthFirstSearch(Space s) {
  this.agenda = new Stack<Space>();
  agenda.push(s);
}
```

Explicit agenda (2)

```
public Space next() {
  if (agenda.empty()) return null;
  Space s = agenda.pop();
  switch (s.status()) {
    case SS_FAILED: return next();
    case SS_SOLVED: return s;
    case SS_BRANCH:
      Space c = s.cloneSpace();
      c.commit(1); agenda.push(c);
      s.commit(0); agenda.push(s);
      return next();
  }}
```

Generic search

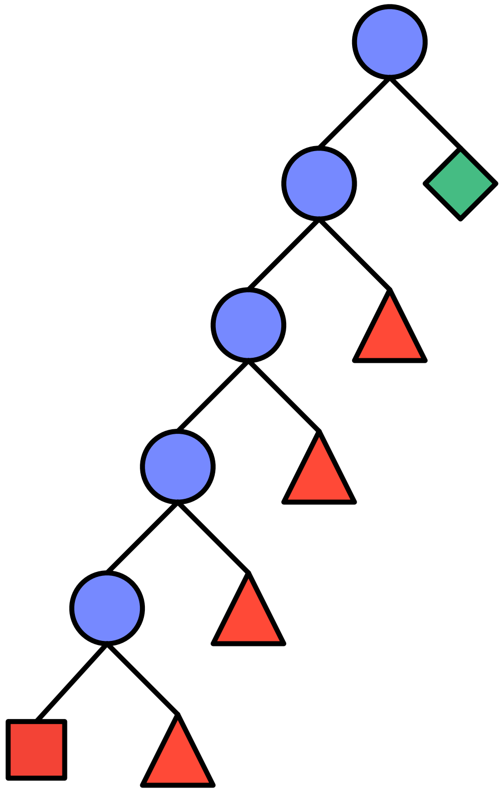
- **depth-first search:**
  agenda is a stack
- **breadth-first depth:**
  agenda is a queue
- **best-first search:**
  agenda is a priority queue

Limited Discrepancy Search

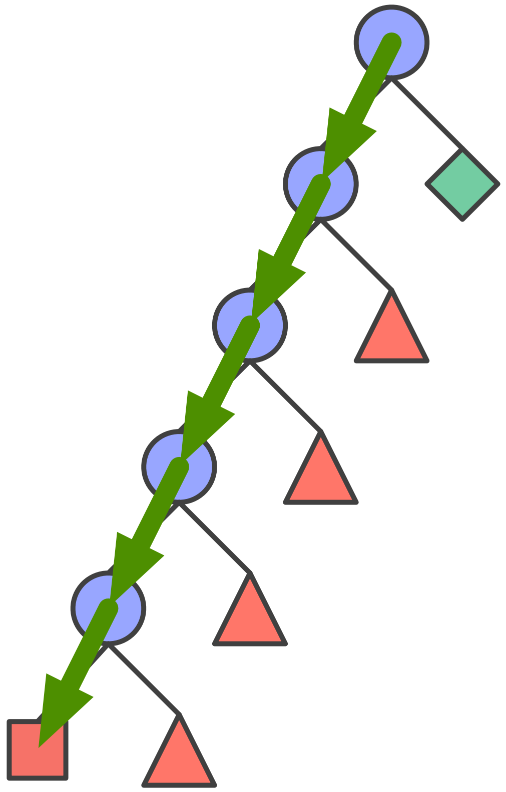
Motivation

- 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 (discrepancies) of the heuristic.

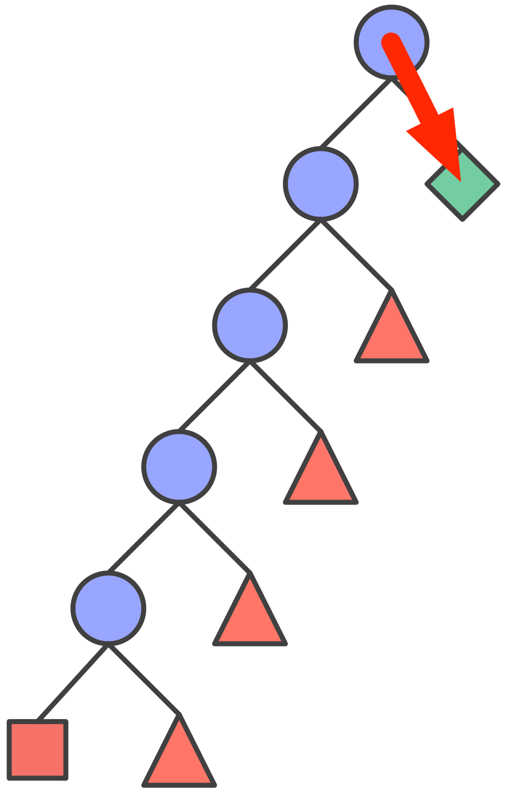
Example



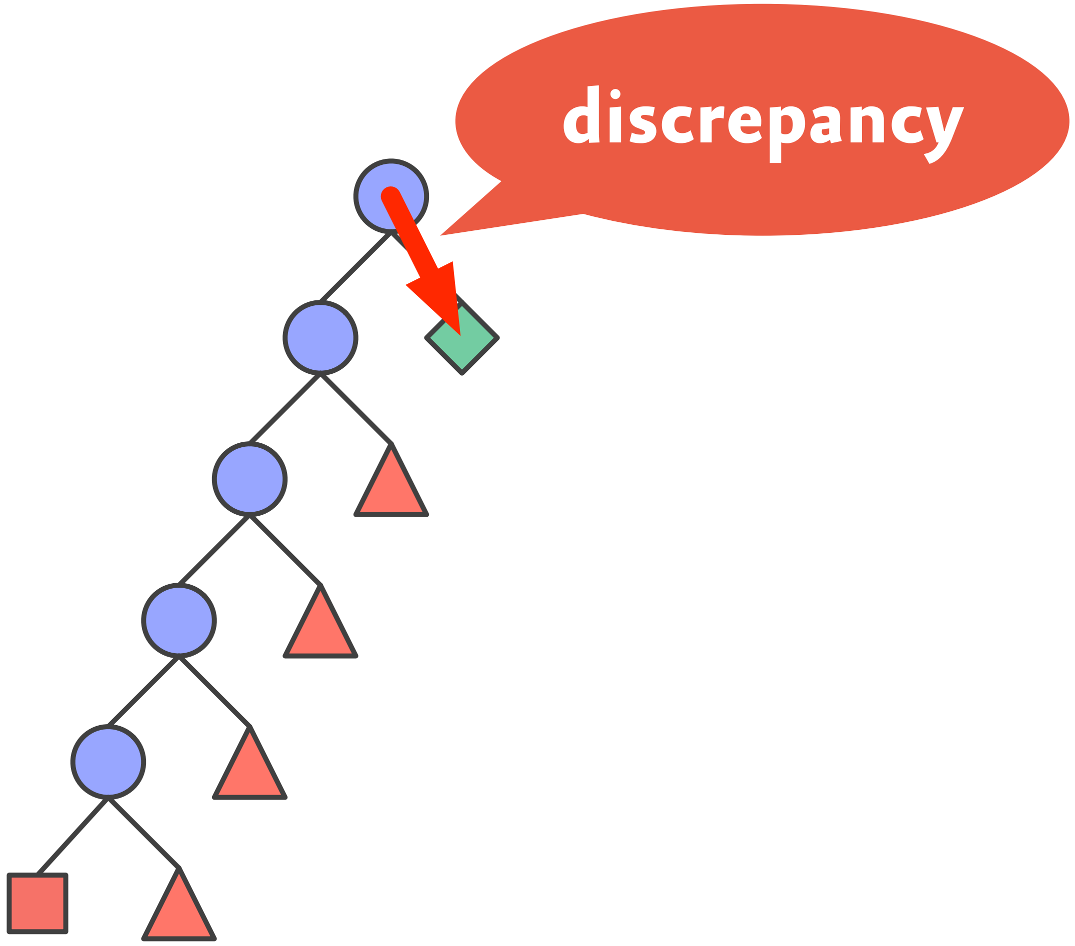
Example



Example

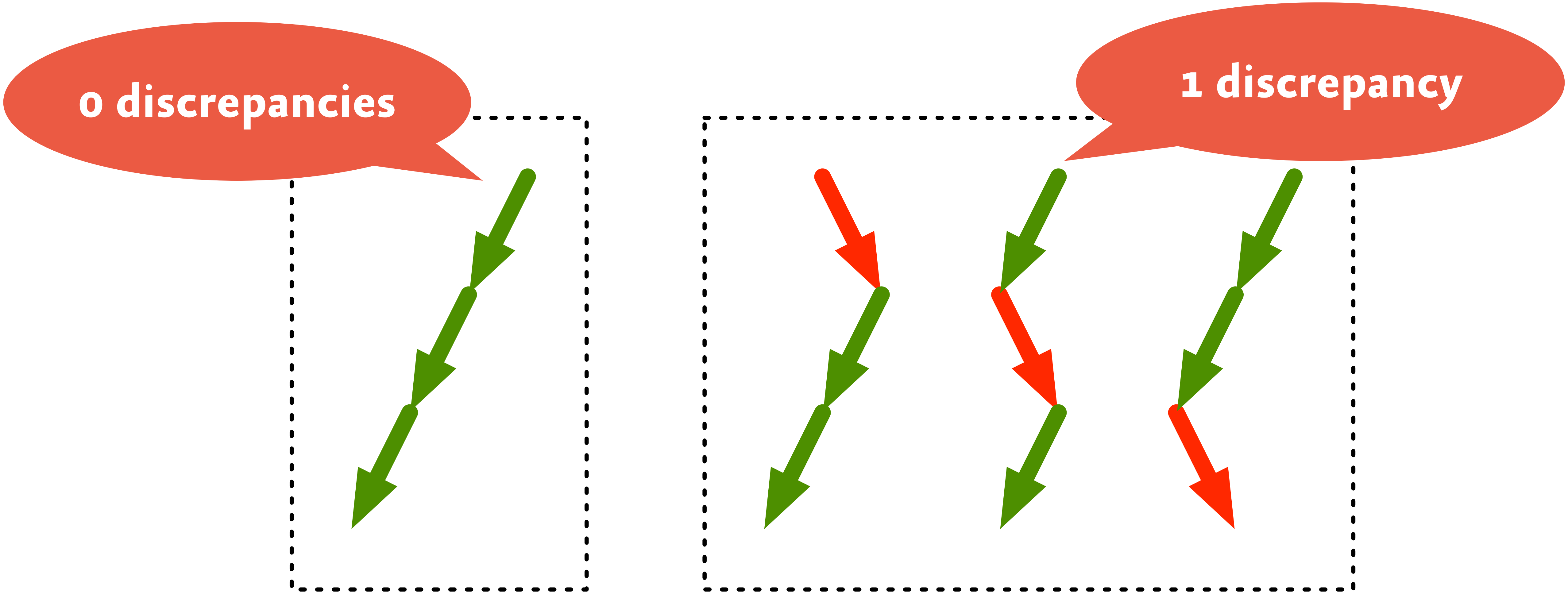


Example



discrepancy

Probing



Limited Discrepancy Search

```
public static Space lds(Space space, int d) {
  switch (space.status()) {
    case SS_FAILED: return null;
    case SS_SOLVED: return space;
    case SS_BRANCH:
      Space c = space.cloneSpace();
      space.commit(0);
      Space s = lds(space, d);
      if (s != null || d < 1) {
        return s;
      } else {
        c.commit(1);
        return dfs(c, d-1);
      }}}
```

LDS as best-solution search

- For some problems, it holds that
  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

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

Remember: Send Most Money

```java
/**
 * Ensure that subsequent solutions
 * are better than best.
 */

public void constrain(Space best) {
  rel(this, money, IRT_GR, best.money.val());
}
```

Remember: Send Most Money

```
/**
 * Ensure that subsequent solutions
 * are better than best.
 */

public void constrain(Space best) {
  rel(this, money, IRT_GR, best.money.val());
}
```

needs to be a solution

Branch & Bound Search

```java
public static Space bbs(Space space, Space best) {
  switch (space.status()) {
    case SS_FAILED: return best;
    case SS_SOLVED: return space;
    case SS_BRANCH:
      Space c = space.cloneSpace();
      space.commit(0);
      Space better = bbs(space, best);
      c.commit(1);
      if (better != null) c.constrain(better);
      return bbs(c, better);
  }}
```

Recomputation

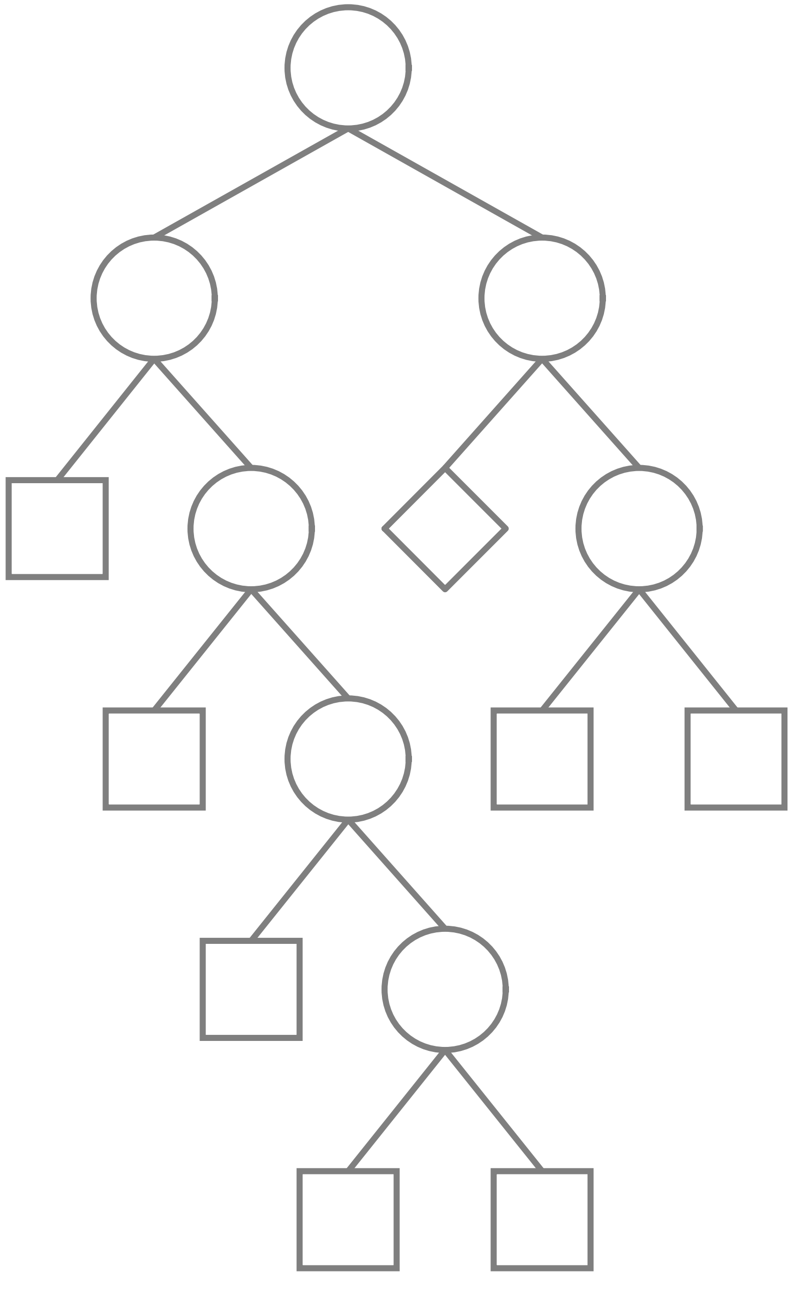
Backtracking strategies

- **copying:**
  backup the state of the system
  before making a choice
- **trailing:**
  remember an undo action for the choice
- **recomputation:**
  recompute the state of the system
  before the choice was made

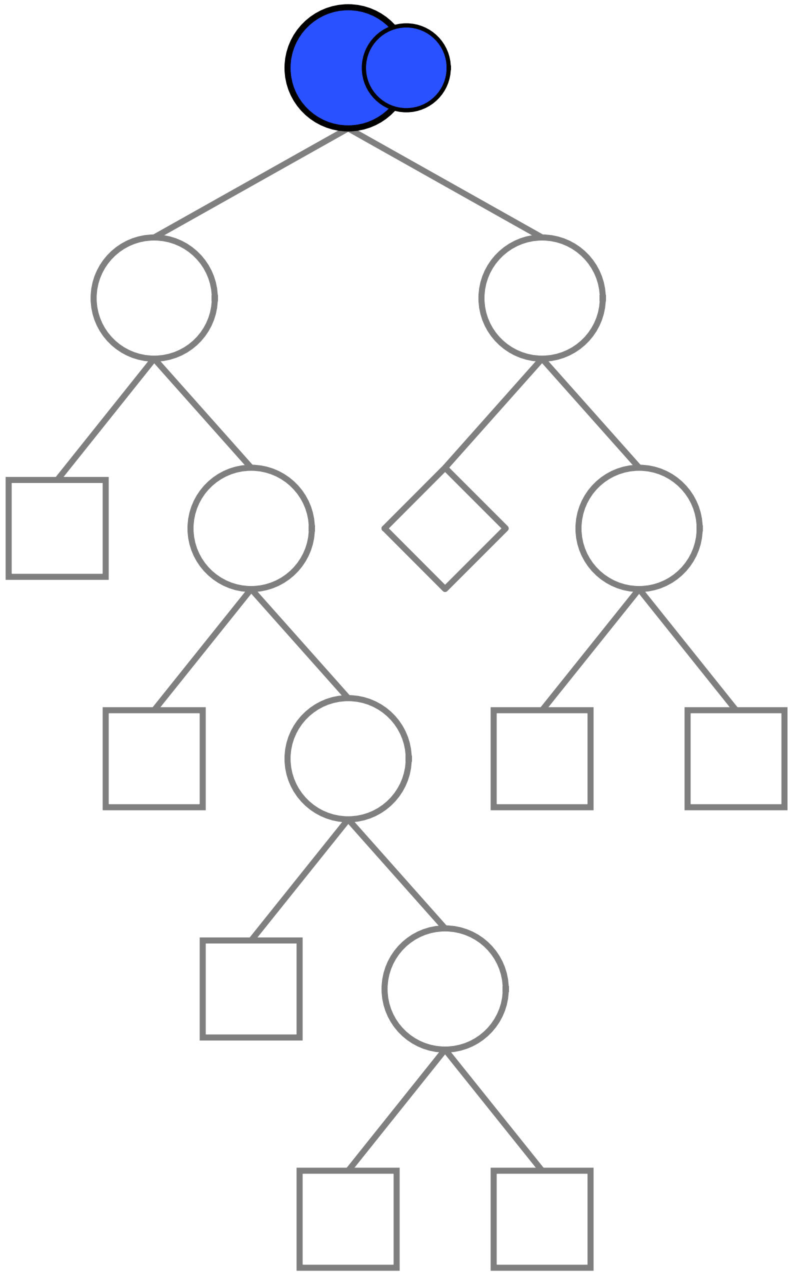
Backtracking strategies

- **copying:**
  backup the state of the system
  before making a choice
- **trailing:**
  remember an undo action for the choice
- **recomputation:**
  recompute the state of the system
  before the choice was made

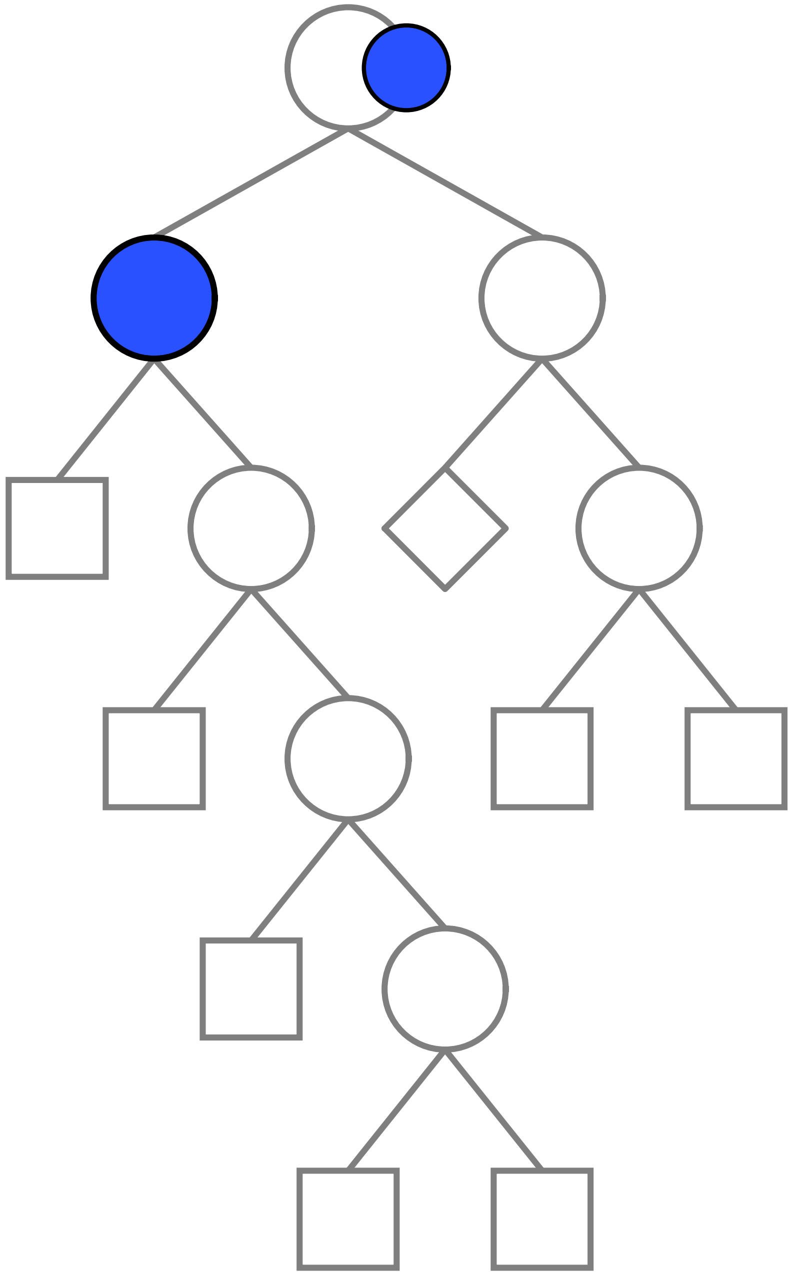
Copying



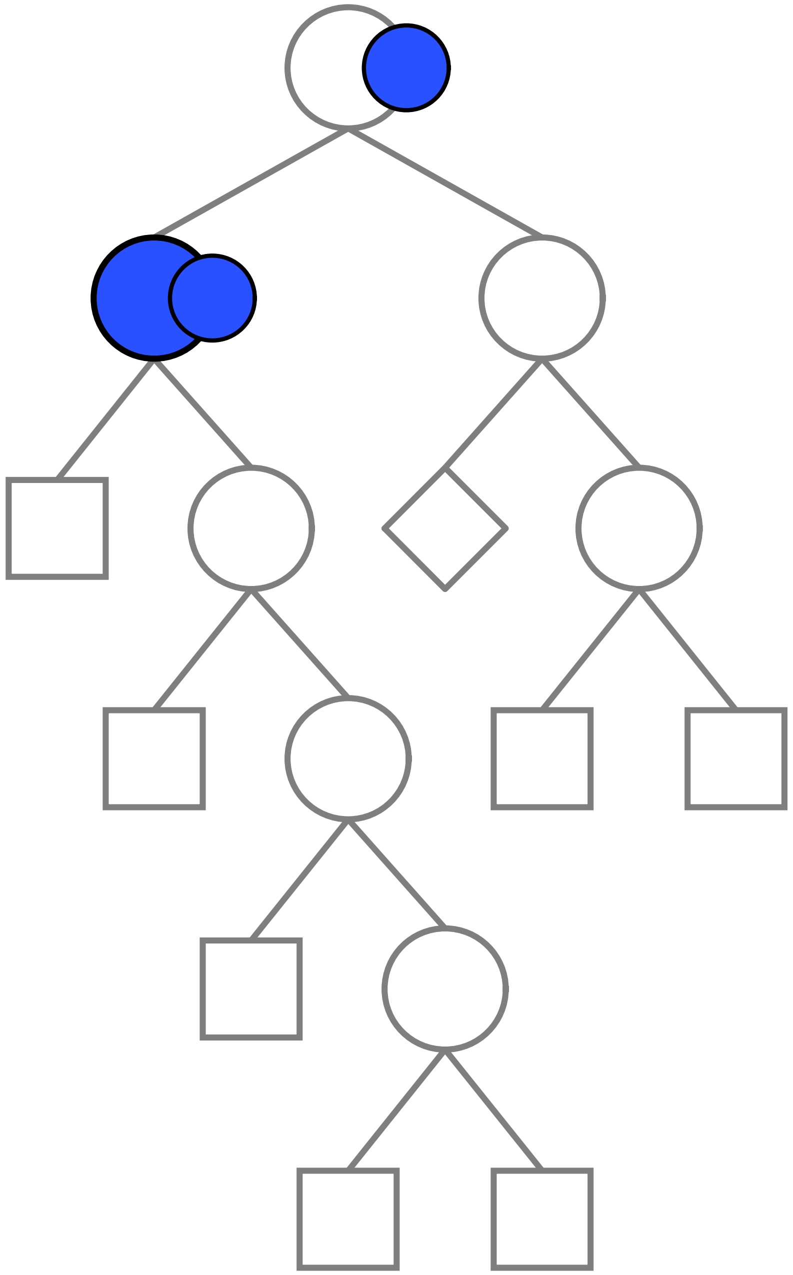
Copying



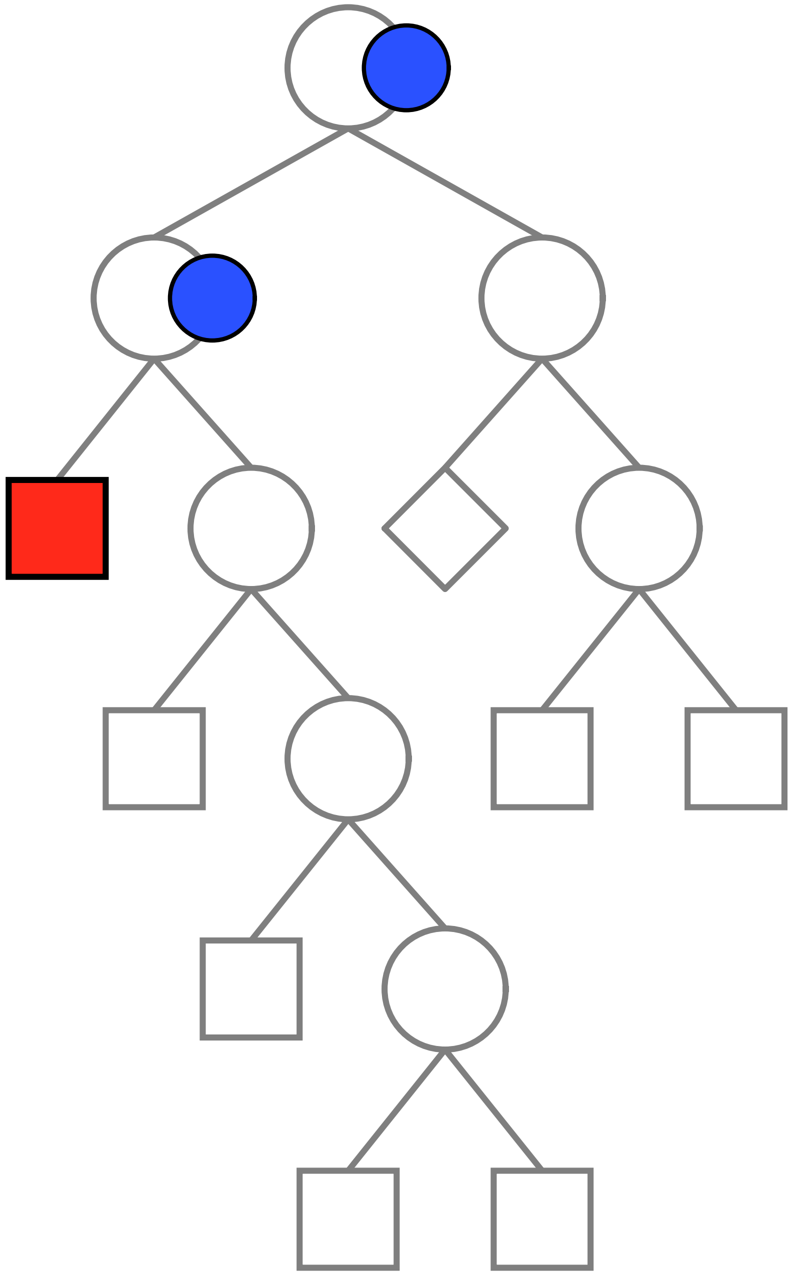
Copying



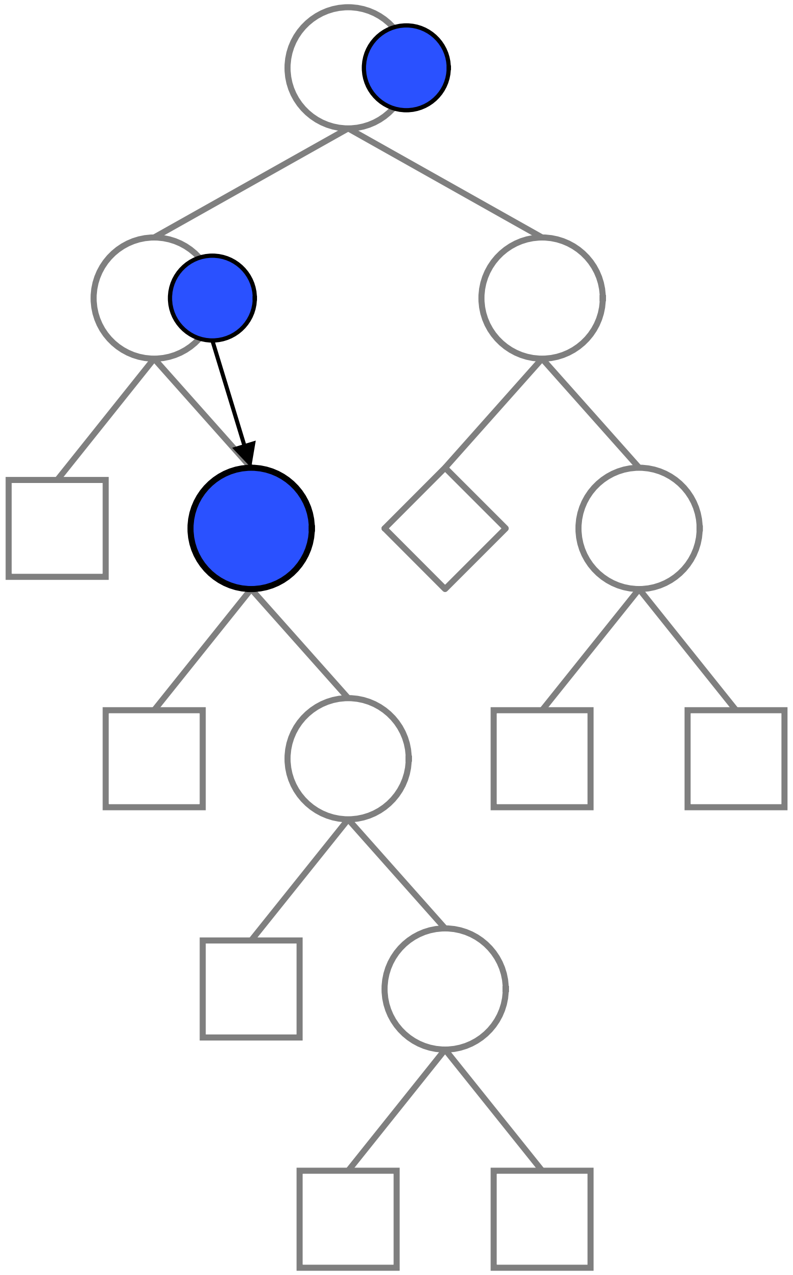
Copying



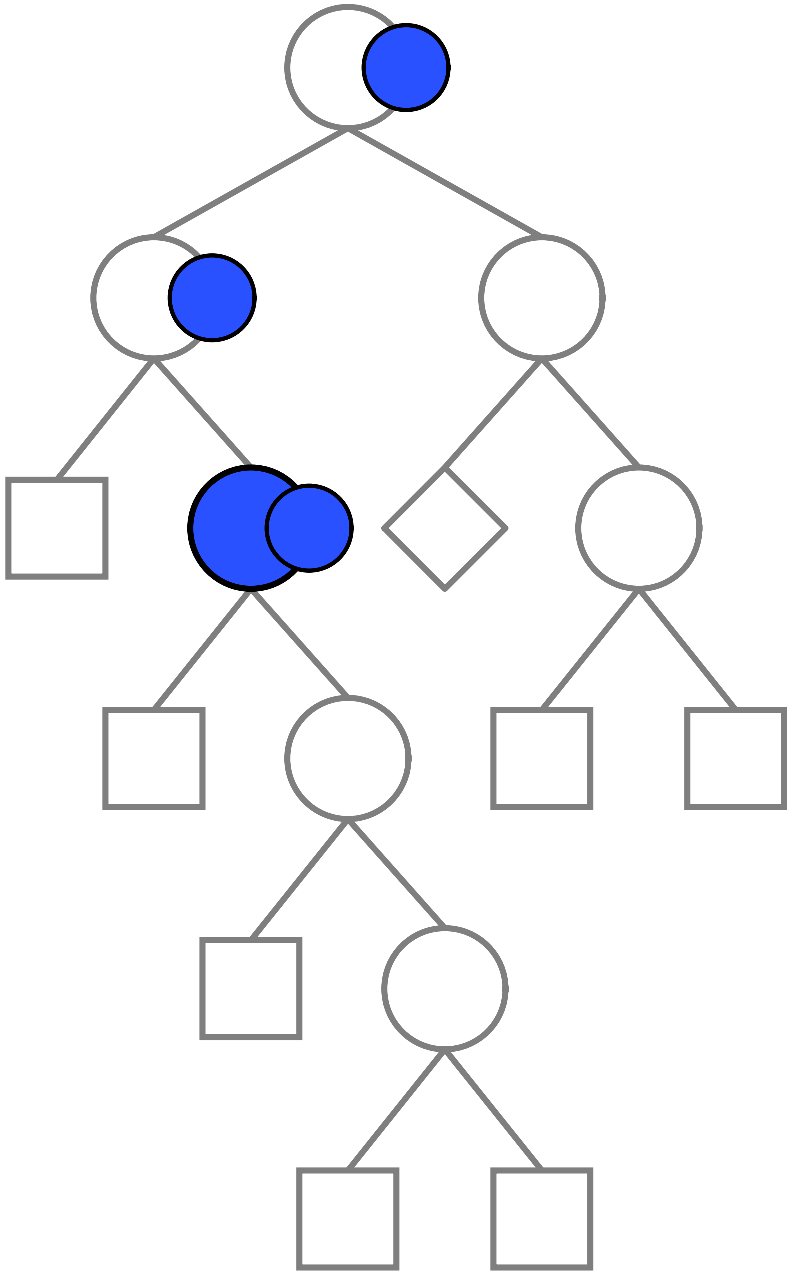
Copying



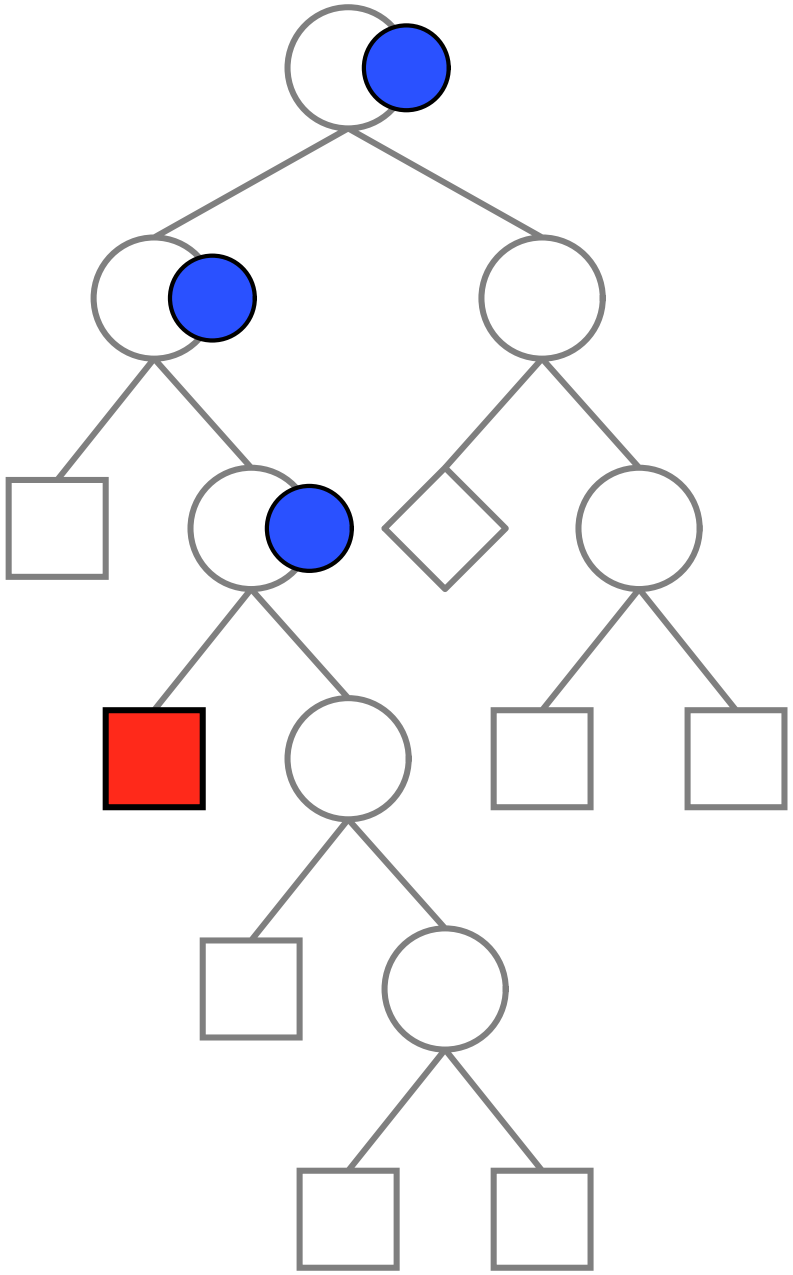
Copying



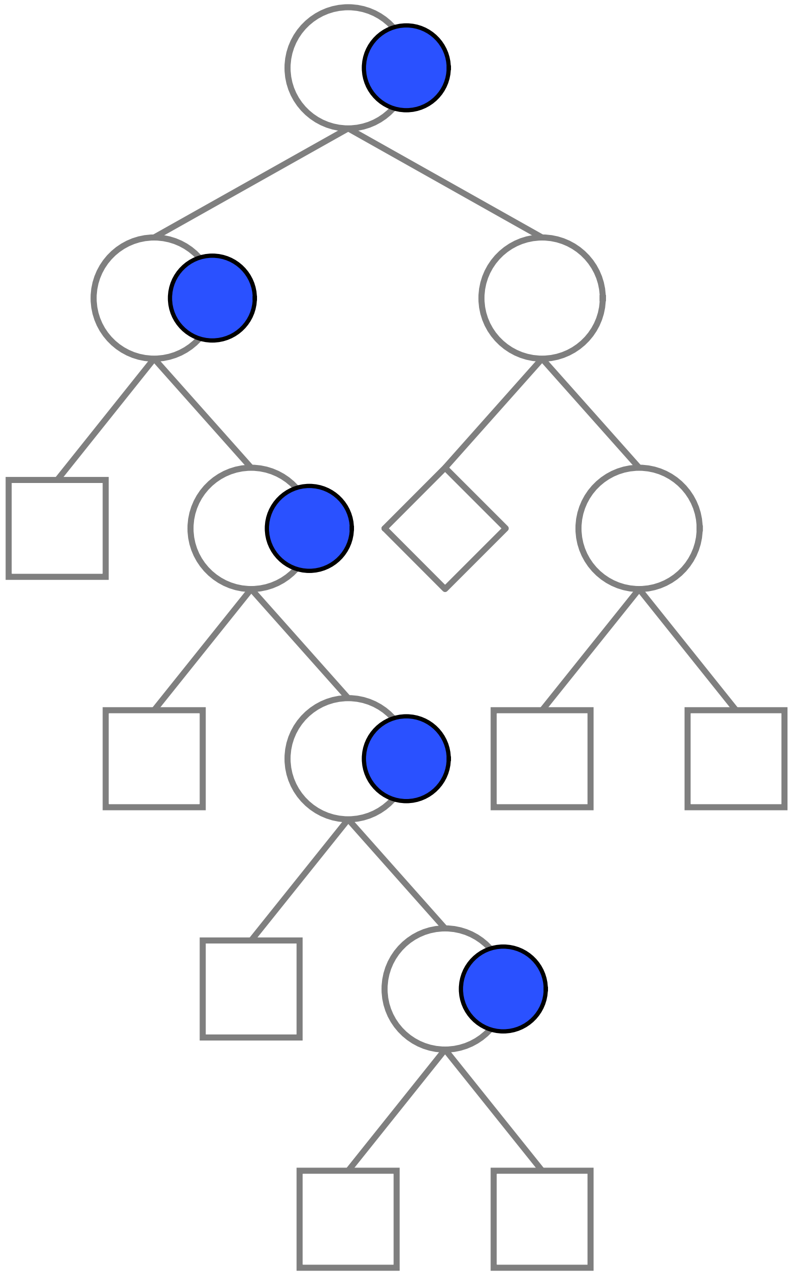
Copying



Copying



Copying



Disadvantages of copying

- **contents of a space**
  - variables and their current domains
  - propagator queue, modified variables
- **memory consumption**
  - not unusual: 1000 variables, 10,000 propagators
  - **several MB per space!**

Backtracking strategies

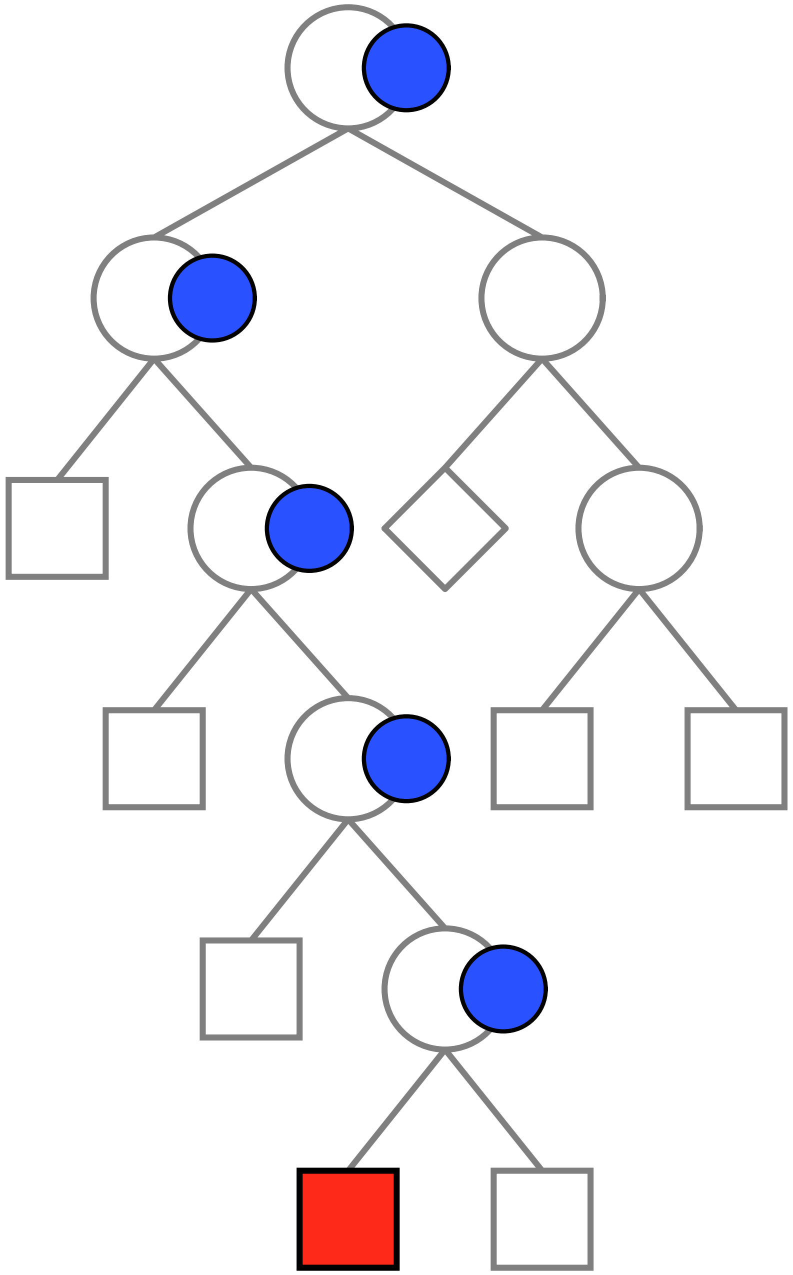
- **copying:**
  backup the state of the system
  before making a choice
- **trailing:**
  remember an undo action for the choice
- **recomputation:**
  recompute the state of the system
  before the choice was made

Backtracking strategies

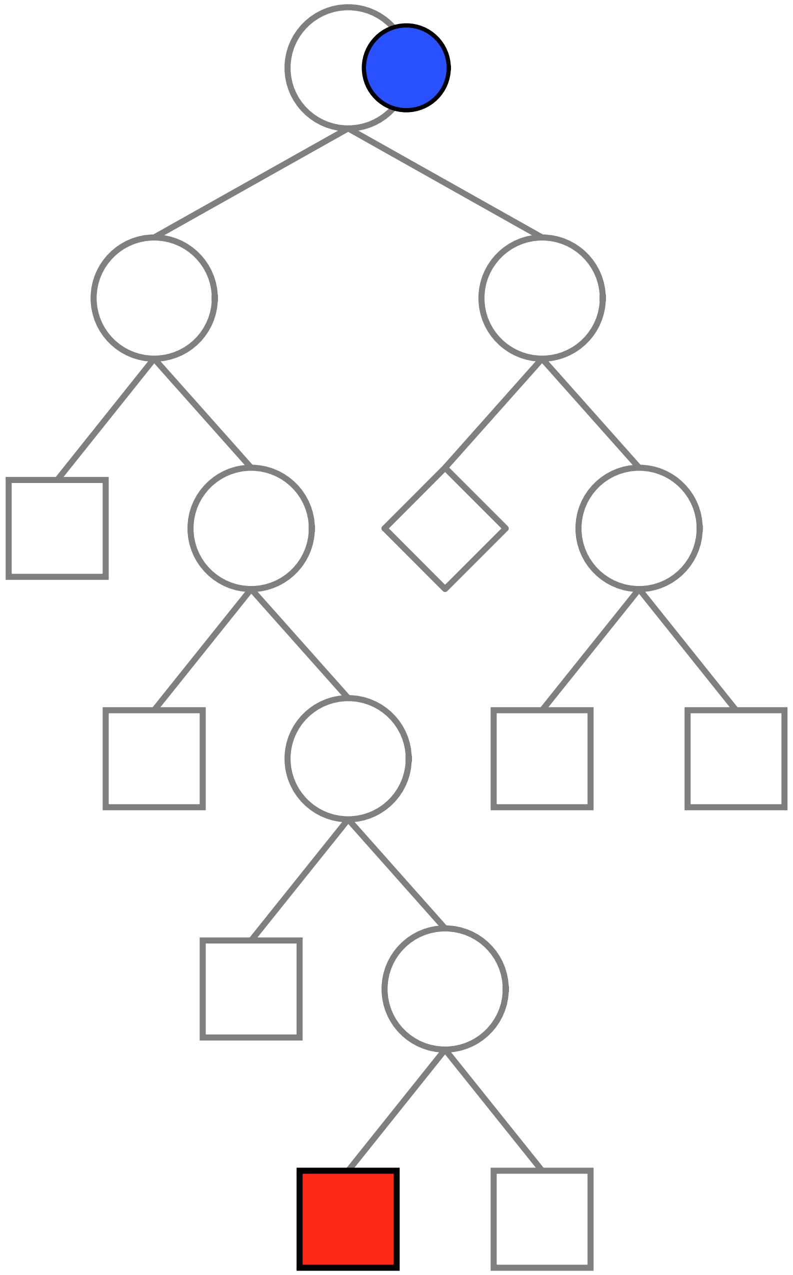
- **copying:**
  backup the state of the system
  before making a choice
- **trailing:**
  remember an undo action for the choice
- **recomputation:**
  recompute the state of the system
  before the choice was made

trade space for time

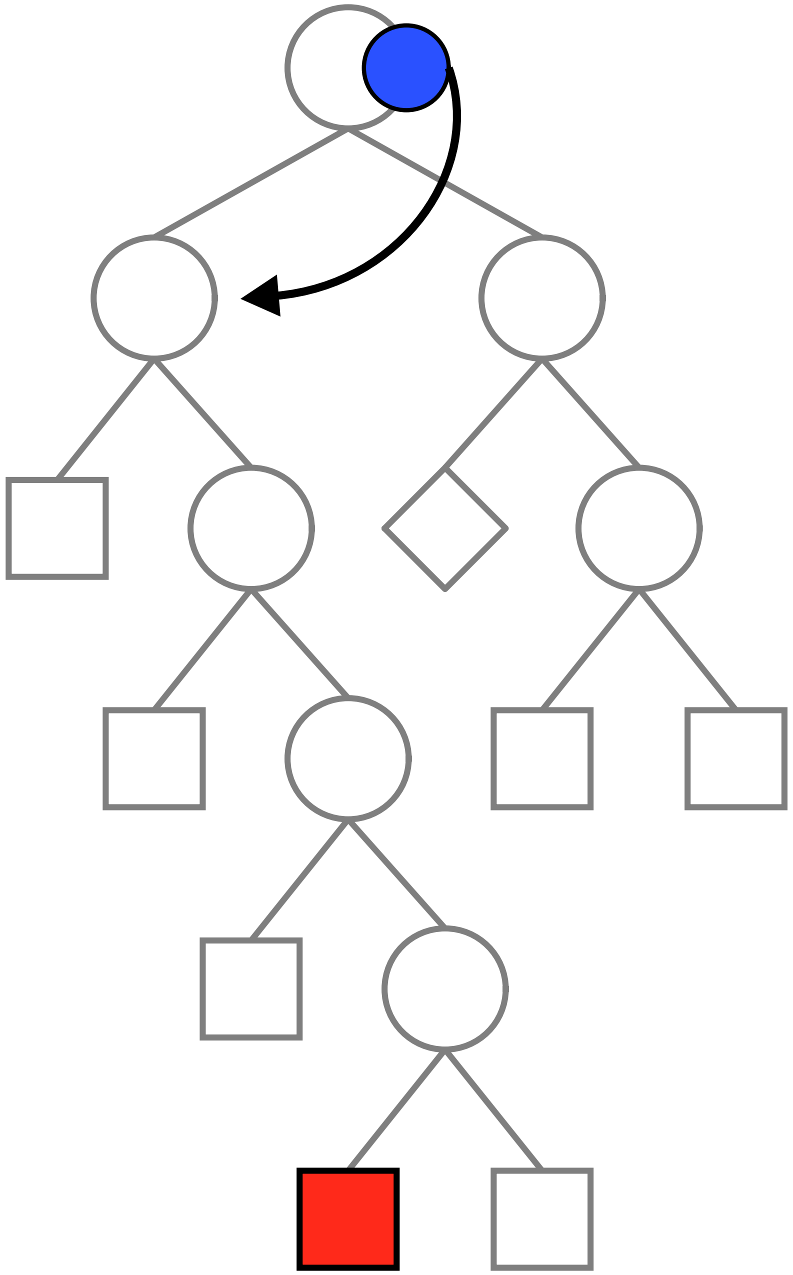
Full recomputation



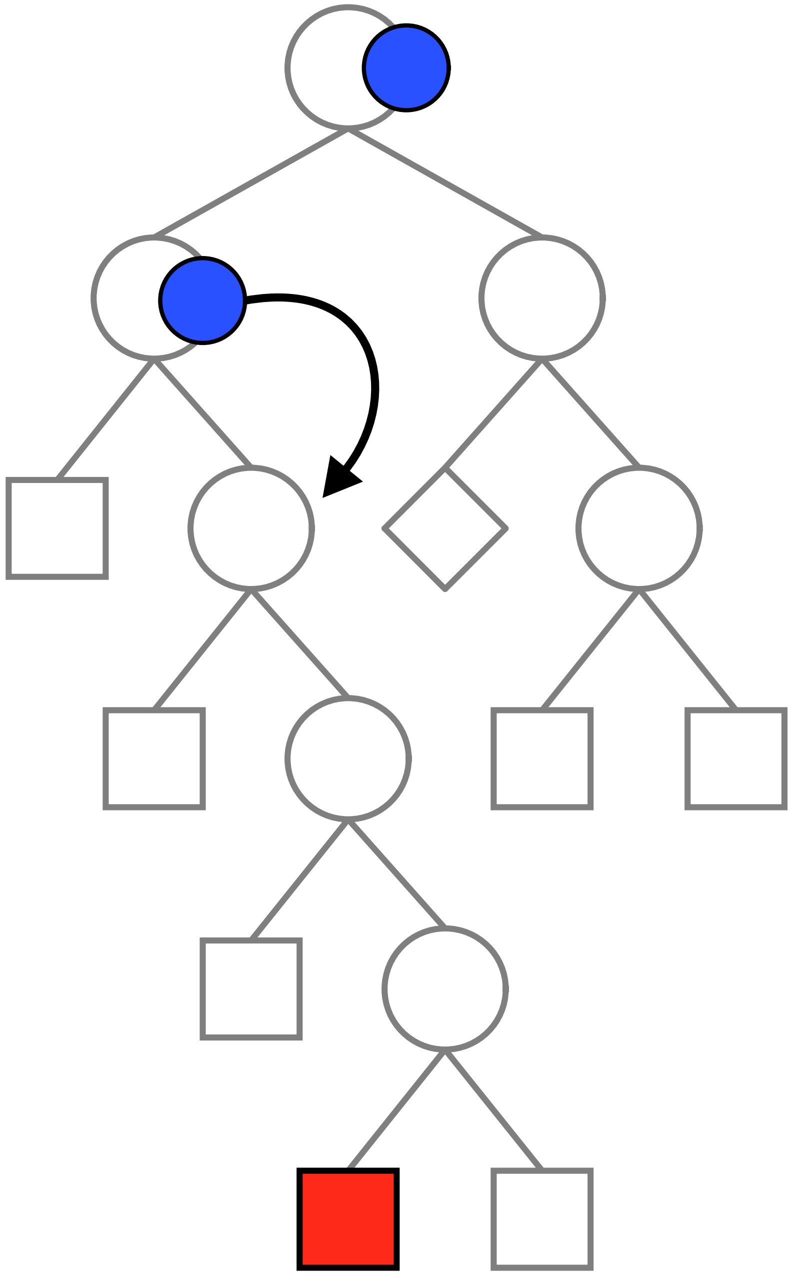
Full recomputation



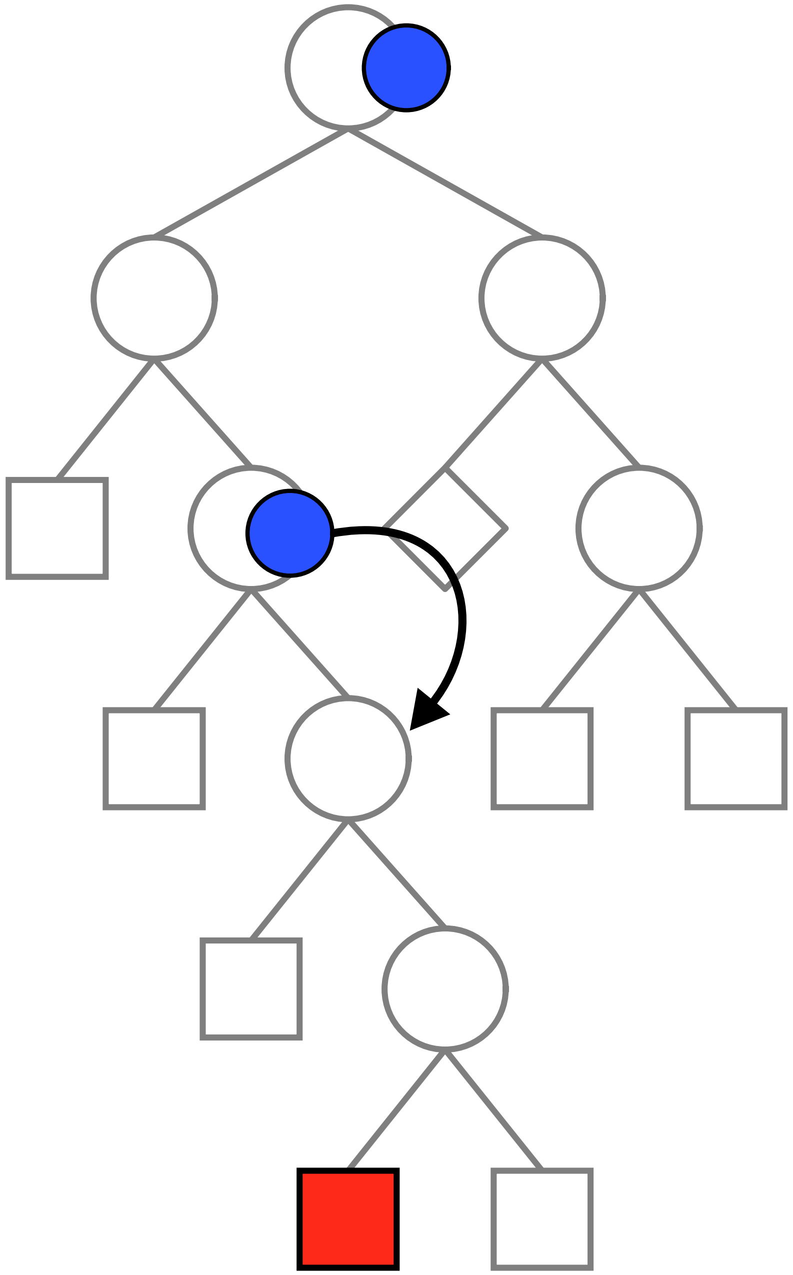
Full recomputation



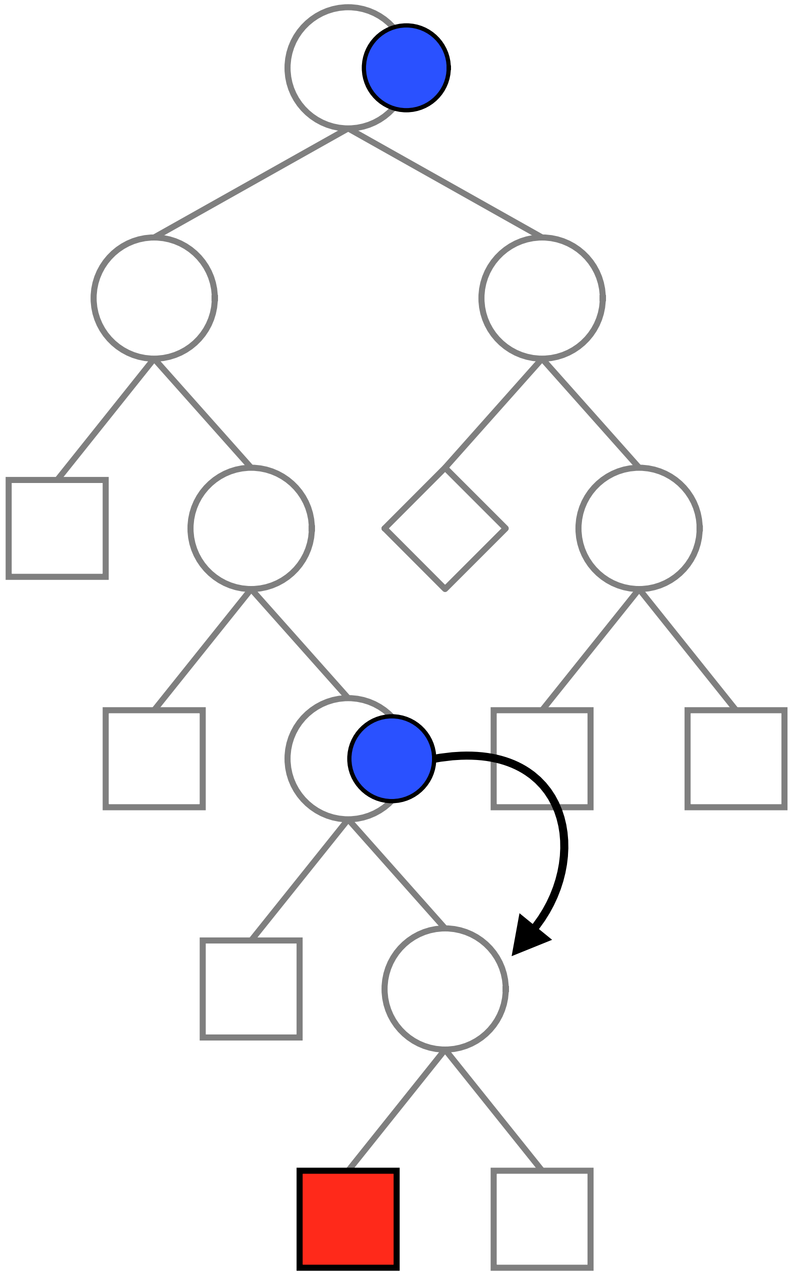
Full recomputation



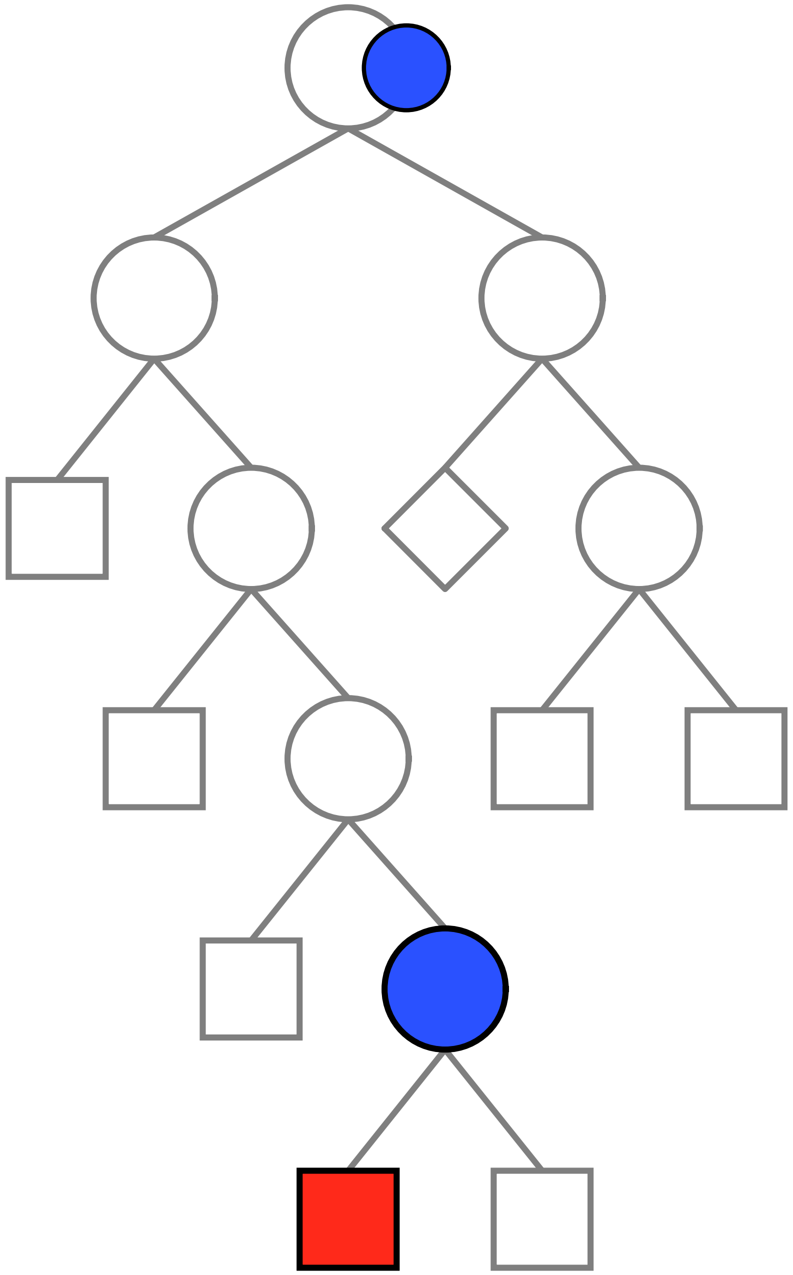
Full recomputation



Full recomputation



Full recomputation



DFS + Recomputation (1)

```
public Space recompute(Path path) {
  Space result = cloneSpace(this);
  for Long i in path {
    result.commit(i);
  }
  return result;
}
```

DFS + Recomputation (2)

```
public static Space dfs(Space space, Space root, Path path) {
  switch (space.status()) {
    case SS_FAILED: return null;
    case SS_SOLVED: return space;
    case SS_BRANCH:
      Path pc = new Path(path); pc.add(1);
      space.commit(0); path.add(0);
      Space s = dfs(space, root, path);
      if (s != null) {
        return s;
      } else {
        Space c = root.recompute(pc);
        return dfs(c, root, pc);
      }}}
```

Recomputation strategies

- **full recomputation**
  no copying at all
- **fixed recomputation**
  keep a copy every **n** nodes
- **adaptive recomputation**
  during recomputation, place a copy
  on the middle of the path to the last copy

Batch Recomputation

- Before we commit to an alternative in a branching, we need to compute the fixed point.
- Therefore, recomputation of a node on a path of length **n** computes **n** fixed points.
- Idea behind **batch recomputation**:
  record what propagators are used along a path, and
  compute only one fixed point per recomputation

Operations on spaces

- **SpaceStatus status()**
- **BranchingDesc description()**
  get a description of the constraints added by status()
- **Space cloneSpace()**
- **void commit(BranchingDesc d, long alternative)**
  post the constraints described by **d**, and
  then commit to one of the alternatives

DFS + Batch Recomputation (1)

```
public Space recompute(Path path) {
  Space result = cloneSpace(this);
  for Pair<BranchingDesc,Long> item in path {
    BranchingDesc d = item.getFirst();
    Long i = item.getSecond();
    result.commit(d, i);
  }
  return result;
}
```

DFS + Batch Recomputation (2)

```
public static Space dfs(Space space, Space root, Path path) {
  switch (space.status()) {
    case SS_FAILED: return null;
    case SS_SOLVED: return space;
    case SS_BRANCH:
      BranchingDesc d = space.description();
      Path pc = new Path(path); pc.add(d, 1);
      space.commit(d, 0); path.add(d, 0);
      Space s = dfs(space, root, path);
      if (s != null) {
        return s;
      } else {
        Space c = root.recompute(pc);
        return dfs(c, root, pc);
      }}}
```

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