# Programming Workshop
# Lecture 2 : Search

Carsten Schürmann
Alexandre Buisse

March 30, 2009

# Search in Computer Science

**Applications**
- Databases
- Games (Chess, Sodoku)
- Planning
- Artificial intelligence

**Technique**
- Recursive data structures
- Depth-first search
- Breadth-first search
- Iterative deepening

**Properties**
- Purely symbolic solution not known
- Speed
- Complexity
- Maintainability
- Correctness: Induction

# Sodoku

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 6 | | | 4 | | | 7 |
| 8 | 4 | | 2 | | | | 6 | |
| | | | | | | | | 2 |
| | | 2 | | 8 | | | 1 | |
| | | | 3 | | 5 | | | |
| | 5 | | | 7 | | 9 | | |
| 5 | | | | | | | | |
| | 2 | | | | 7 | | 5 | 6 |
| 1 | | | 8 | | | 3 | | |

# Basic Definitions

## Definition (Variable)

A variable is a place holder for an entity that is conjectured to exist.

## Definition (Configuration)

An instantiation of all variables defining a search problem.

## Definition (Operator)

A function that transforms one configuration into another.

## Definition (Expansion)

An expansion of a configuration is the largest set of all operators.

## Definition (Search)

The activity of constructing a sequence of configurations, $C_0 \rightarrow C_1 \ldots \rightarrow C_n$, such that each operator transforming $C_i$ to $C_{i+1}$ is in the expansion of $C_i$.

# Recursion

**Functional**

- Configurations are passed as parameters.
- Activation Record maintains configurations.
- Method returns to previous configuration.

```
int fib (int n) {
  if (n<2) {return 1;}
  else {return (fib (n-1) + fib (n-2));}
}
```

**Imperative**

- Configurations are encoded as objects.
- Layout of objects in memory.
- Two options. State maintenance up to user or garbage collector.

```
class List<A> {
  A head;
  List<A> tail;
  void dosomething () { ... };
}
```

# Example 1

**Problem:** Is there a solution to $n^2 - 8n + 16 = 0$ where $0 \leq n \leq 9$, natural number.

**Solution:** Functional search.

```
boolean search (int n) {
  if (n <= 9) {
    if (n*n - 8*n + 16 == 0) {return true;}
    else {return (search (n+1));}
  }
  return false;
}
```

**Better:** Symbolic solution: Solve for $x$.

# Example 2

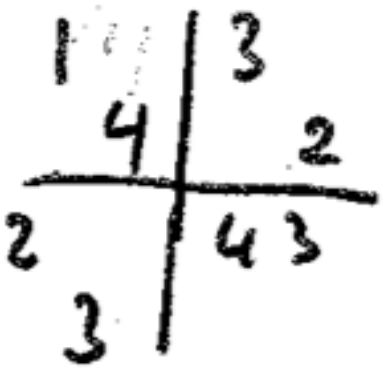
Problem: Is there a solution to $nm^2 + n^2m - 2 = 0$ where $0 \leq n \leq 9$, natural number.

Solution: Functional search.

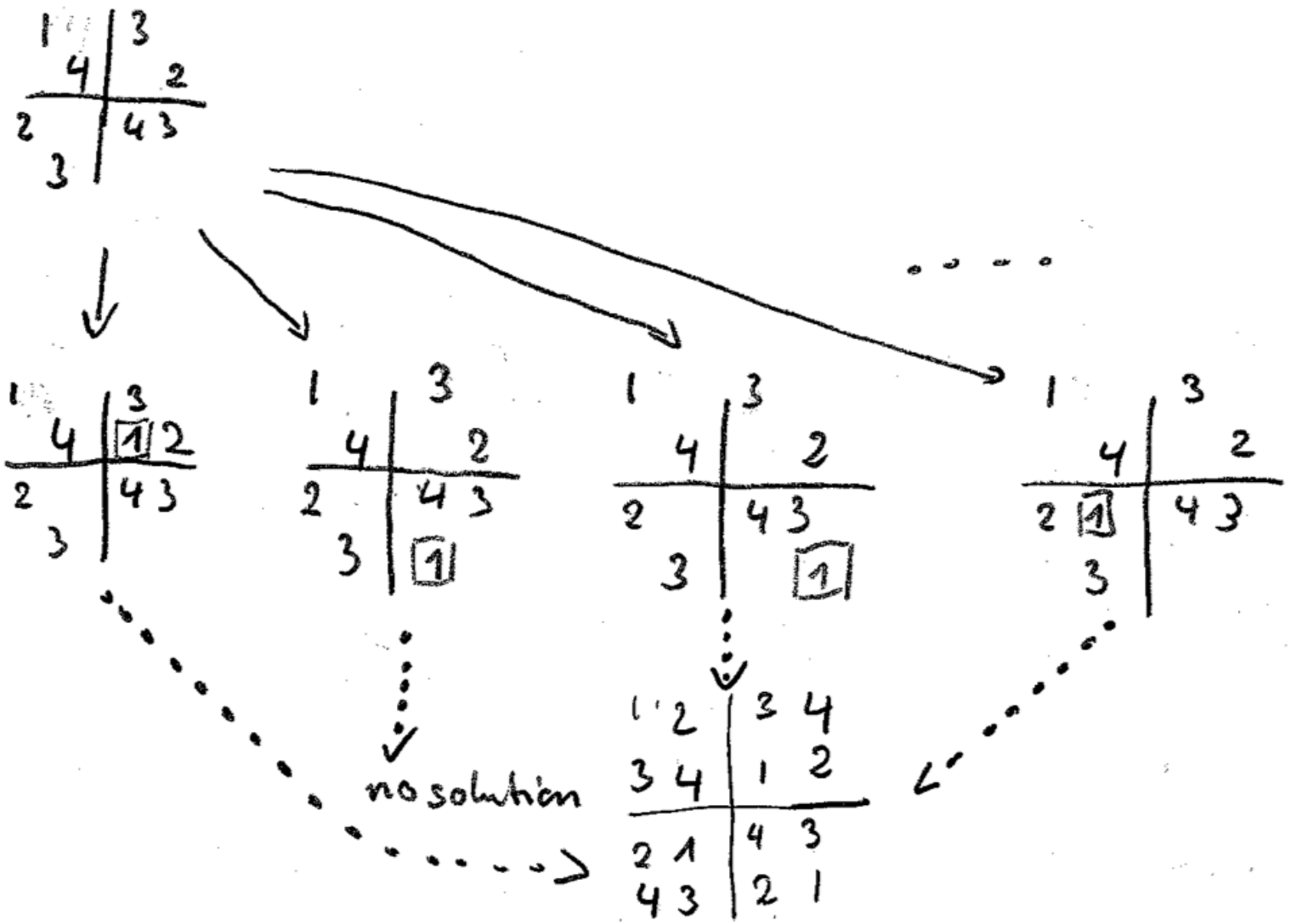
```
boolean search (int n, int m) {
  if (n <= 9 && m <=9) {
    if (n*m*m + n*n*m -2 == 0) {return true;}
    else {if (m == 9) {return (search (n+1, 0));
          else {return (search (n, m+1));}}
  }
  return false;
}
```

# Example 3

Problem: Is it possible to solve the 2 by 2 Sodoku?



# Search Tree



# Search Tree Traversal Strategies

**DFS** Depth First-Search Strategy

- Choose operator and apply
- return if and only if solution found

**BFS** Breadth First-Search Strategy

- Maintain a list of all *open* configurations
- Pick a configuration expand it to all possibilities and append
- return if and only if solution found

**ID** Iterative Deepening

- DFS, depth limited
- If no solution found, increase depth limit
- Try again

# Functional Depth First Search

Condition: Configuration is effect-free.

Code:

```
boolean dfs (Configuration c) {
  if (c.success ()) {return true;}
  else {
    for (Operator op in c.iterator();
         op.hasNext()) {
      if (dfs (op.next ())) {return true;}
    }
  }
  return false;
}
```

Bad: Works only for few examples

Good: Fastest

# Imperative Depth First Search [Do it yourself]

Code:

```
Configuration c;
boolean dfs () {
  boolean result = false;
  if (c.success ()) {return true;}
  else {
    for (Operator op in c.iterator();
         op.hasNext()) {
      c.apply (op.next());
      result = dfs ();
      c.revert ();
      if (result) {return true;}
    }
  }
  return false;
}
```

Bad: Error prone

Good: Fast

# Imperative Depth First Search [Garbage Collector]

Code:

```
class Configuration {
  boolean dfs () {
    if (this.success ()) {return true;}
    else {
      for (Operator op in this.iterator();
           op.hasNext()) {
        Configuration c  = this.clone ()
        c.apply (op.next ());
        if (c.dfs ()) {return true;}
      }
    }
    return false;
  }
}
```

Bad: Slow. Lots of copying

Good: Easy code

Questions and Answers