DM841

Discrete Optimization

Part I

Lecture 4

Introduction to Gecode

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Outline

1. Introduction to Gecode
   n-Queens, Grocery, Magic Squares

2. Solving CSP – Overview

Resume

- CP modeling examples
  - Coloring with consecutive numbers
  - Send More Money
- Constraint programming:
  representation (modeling language) + reasoning (propagation + search)
  - propagate, filtering, pruning
  - search = backtracking + branching
- Gecode: model in Script class implementation
  - Variables
    declare as members
    initialize in constructor
    update in copy constructor
  - Posting constraints (in constructor)
  - Create branching (in constructor)
  - Provide copy constructor (recomputation) and copy function (cloning)

Solving Scripts

Available Search Engines

- Returning solutions one by one for script
  - `DFS` depth-first search
  - `BAB` branch-and-bound
  - `Restart, LDS`
- Interactive, visual search
  - `Gist`

Main Method: First Solution

```
…

int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  DFS<SendMoreMoney> e(m);
  delete m;
  if (SendMoreMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

Main Method: First Solution

create root
space for
search

…

```
int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  DFS<SendMoreMoney> e(m);
  delete m;
  if (SendMoreMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

Main Method: First Solution

…

```
int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  DFS<SendMoreMoney> e(m);
  delete m;
  if (SendMoreMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

> create search engine (takes clone of `m`)

Main Method: First Solution

root space not any longer needed

…

```
int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  DFS<SendMoreMoney> e(m);
  delete m;
  if (SendMoreMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

Main Method: First Solution

```
…

int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  DFS<SendMoreMoney> e(m);
  delete m;
  if (SendMoreMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

search first solution and print it

Main Method: All Solutions

```
…

int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  DFS<SendMoreMoney> e(m);
  delete m;
  while (SendMoreMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

Gecode Gist

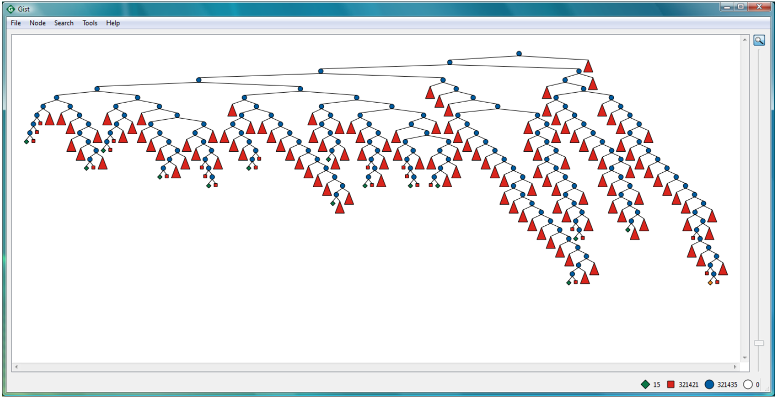
- A graphical tool for exploring the search tree
  - explore tree step by step
  - tree can be scaled
  - double-clicking node prints information: inspection
  - search for next solution, all solutions
  - …
- Best to play a little bit by yourself
  - hide and unhide failed subtrees
  - …

Main Function: Gist

```
#include <gecode/gist.hh>

int main(int argc, char* argv[]) {
  SendMoreMoney* m = new SendMoreMoney;
  Gist::dfs(m);
  delete m;
  return 0;
}
```

Gist Screenshot



Best Solution Search

Reminder: SMM++

- Find distinct digits for letters, such that

```
      SEND
+     MOST
= MONEY
```

and `MONEY` maximal

Script for SMM++

- Similar, please try it yourself at home
- In the following, referred to by `SendMostMoney`

Solving SMM++: Order

- Principle
  - for each solution found, constrain remaining search for better solution
- Implemented as additional method

```
virtual void constrain(const Space& b) {
  …
}
```

- Argument b refers to so far best solution
  - only take values from b
  - never mix variables!
- Invoked on object to be constrained

Order for SMM++

```
virtual void constrain(const Space& _b) {
  const SendMostMoney& b =
    static_cast<const SendMostMoney&>(_b);

  IntVar e(l[1]), n(l[2]), m(l[4]), o(l[5]), y(l[8]);

  IntVar b_e(b.l[1]), b_n(b.l[2]), b_m(b.l[4]),
         b_o(b.l[5]), b_y(b.l[8]);

  int money = (10000*b_m.val()+1000*b_o.val()+100*b_n.val()+
               10*b_e.val()+b_y.val());
   rel
  post(*this, 10000*m+1000*o+100*n+10*e+y > money);
}
```

value of any next solution

value of current best solution b

Main Method: All Solutions

```
…

int main(int argc, char* argv[]) {
  SendMostMoney* m = new SendMostMoney;
  BAB<SendMostMoney> e(m);
  delete m;
  while (SendMostMoney* s = e.next()) {
    s->print(); delete s;
  }
  return 0;
}
```

Main Function: Gist

```
#include <gecode/gist.hh>

int main(int argc, char* argv[]) {
  SendMostMoney* m = new SendMostMoney;
  Gist::bab(m);
  delete m;
  return 0;
}
```

Summary: Solving

- Result-only search engines
  - `DFS, BAB`
- Interactive search engine
  - `Gist`

- Best solution search uses constrain-method for posting constraint
- Search engine independent of script and constrain-method

Exercise

- Solve in Gecode the problem:

  $$\texttt{send} + \texttt{more} = \texttt{money}$$

  What is the solution that maximizes money? How many solutions are there for the decision version? Compare using lexicographic and first-fail search. Which of the two search strategeis is the best?

- Repeat the analysis on this other instance of the problem:

  $$\texttt{ten} + \texttt{ten} + \texttt{forty} = \texttt{sixty}$$

  Is the conclusion the same as in the point above?

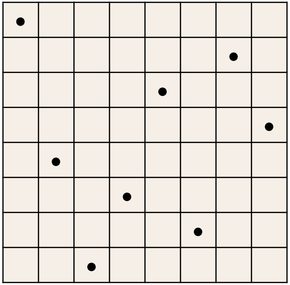
Outline

1. Introduction to Gecode
   n-Queens, Grocery, Magic Squares

2. Solving CSP – Overview

8-Queens

Problem Statement



- Place 8 queens on a chess board such that the queens do not attack each other
- Straightforward generalizations
  - place an arbitrary number: *n* Queens
  - place as closely together as possible

What Are the Variables?

- Representation of position on board
- First idea: two variables per queen
  - one for row
  - one for column
  - $2 \cdot n$ variables
- Insight: on each column there will be a queen!

Fewer Variables…

- Have a variable for each column
  - value describes row for queen
  - *n* variables
- Variables: $x_0, \ldots, x_7$
  where $x_i \in \{0, \ldots, 7\}$

Other Possibilities

- For each field: number of queen
  - which queen is not interesting, so…
  - n^2 variables
- For each field on board: is there a queen on the field?
  - 8×8 variables
  - variable has value 0: no queen
  - variable has value 1: queen
  - n^2 variables

Outline

General Purpose Algorithms

Search algorithms

organize and explore the search tree

- Search tree with branching factor at the top level nd and at the next level $(n-1)d$. The tree has $n! \cdot d^n$ leaves even if only d^n possible complete assignments.
- Insight: CSP is commutative in the order of application of any given set of action (the order of the assignment does not influence final answer)
- Hence we can consider search algs that generate successors by considering possible assignments for only a single variable at each node in the search tree.
  The tree has d^n leaves.

Backtracking search

depth first search that chooses one variable at a time and backtracks when a variable has no legal values left to assign.

Backtrack Search

function BACKTRACKING-SEARCH(*csp*) **returns** a solution, or failure
  return RECURSIVE-BACKTRACKING({ }, *csp*)

function RECURSIVE-BACKTRACKING(*assignment*, *csp*) **returns** a solution, or failure
  if *assignment* is complete **then return** *assignment*
  var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[*csp*], *assignment*, *csp*)
  for each *value* **in** ORDER-DOMAIN-VALUES(*var*, *assignment*, *csp*) **do**
    if *value* is consistent with *assignment* according to CONSTRAINTS[*csp*] **then**
      add {*var* = *value*} to *assignment*
      result ← RECURSIVE-BACKTRACKING(*assignment*, *csp*)
      if *result* ≠ *failure* **then return** *result*
      remove {*var* = *value*} from *assignment*
  return *failure*

Backtrack Search

- No need to copy solutions all the times but rather extensions and undo extensions
- Since CSP is standard then the alg is also standard and can use general purpose algorithms for initial state, successor function and goal test.
- Backtracking is uninformed and complete. Other search algorithms may use information in form of heuristics

General Purpose Backtracking

Implementation refinements

1) [Search] Which variable should we assign next, and in what order should its values be tried?

2) [Propagation] What are the implications of the current variable assignments for the other unassigned variables?

3) [Search] When a path fails – that is, a state is reached in which a variable has no legal values can the search avoid repeating this failure in subsequent paths?

Search

1) Which variable should we assign next, and in what order should its values be tried?

- Select-Initial-Unassigned-Variable
  degree heuristic (reduces the branching factor) also used as tie breaker
- Select-Unassigned-Variable
  Most constrained variable (DSATUR); fail-first heuristic;
  Minimum remaining values (MRV) heuristic (speeds up pruning)
- Order-Domain-Values
  least-constraining-value heuristic (leaves maximum flexibility for subsequent variable assignments)

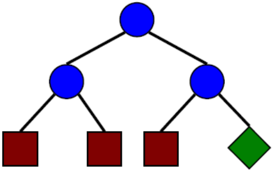
NB: If we search for all the solutions or a solution does not exists, then the ordering does not matter.

Search
Branching (aka, Labelling)

1. Pick a variable x with at least two values
2. Pick value v from $D(x)$
3. Branch with

$$x = v \qquad x \neq v$$
$$x \leq v \qquad x > v$$

The constraints for branching become part of the model in the subproblems generated



The inner nodes (blue circles) are choices, the red square leaf nodes are failures, and the green diamond leaf node is a solution.