DM811
Heuristics for Combinatorial Optimization

Lecture 2
Introductory Topics

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Outline

1. Search Paradigms
   Construction Heuristics
   Local Search

2. Software Tools
   Constraint-Based Local Search with Comet^{TM}

Construction Heuristics

Construction heuristics

(aka, single pass heuristics or dispatching rules in scheduling)
They are closely related to tree search techniques but correspond to a single path from root to leaf

- search space = partial candidate solutions
- search step = extension with one or more solution components

Construction Heuristic (CH):
$s := \emptyset$
while s is not a complete candidate solution **do**
  choose a solution component $(X_i = v_j)$
  add the solution component to s

Designing Constr. Heuristics

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

- Select-Unassigned-Variable
  - *Static*: Degree heuristic (reduces the branching factor) also used as tie breaker
  - *Dynamic*: Most constrained variable = Fail-first heuristic = Minimum remaining values heuristic
- Order-Domain-Values
  eg, least-constraining-value heuristic (leaves maximum flexibility for subsequent variable assignments)

Designing Constr. Heuristics

- Ideas for variable selection
  - with smallest min value
  - with largest min value
  - with smallest max value
  - with largest max value
  - with smallest domain size
  - with largest domain size

  The degree of a variable is defined as the number of constraints it is involved in.
  - with smallest degree. In case of ties, variable with smallest domain.
  - with largest degree. In case of ties, variable with smallest domain.
  - with smallest domain size divided by degree
  - with largest domain size divided by degree

  The min-regret of a variable is the difference between the smallest and second-smallest value still in the domain.
  - with smallest min-regret: $i = \operatorname{argmin} \Delta f_i^{(2)} - \Delta f_i^{(1)}$
  - with largest min-regret: $i = \operatorname{argmax} \Delta f_i^{(2)} - \Delta f_i^{(1)}$
  - with smallest max-regret: $i = \operatorname{argmin} \Delta f_i^{(n)} - \Delta f_i^{(1)}$
  - with largest max-regret: $i = \operatorname{argmax} \Delta f_i^{(n)} - \Delta f_i^{(1)}$

Designing Constr. Heuristics

- Ideas for value selection
  - Select smallest value
  - Select median value
  - Select maximal value

  Look-ahead:
  - Select value that leaves the largest number of feasible values at to the other variables
  - Select value that leaves the smallest number of feasible values at to the other variables (fail early)

Greedy best-first search

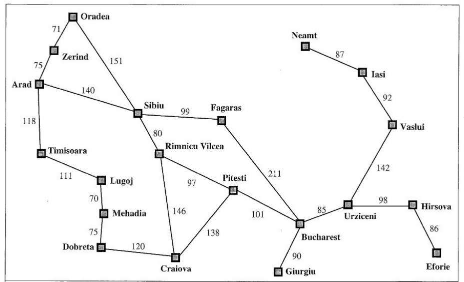


Figure 3.2 A simplified road map of part of Romania.

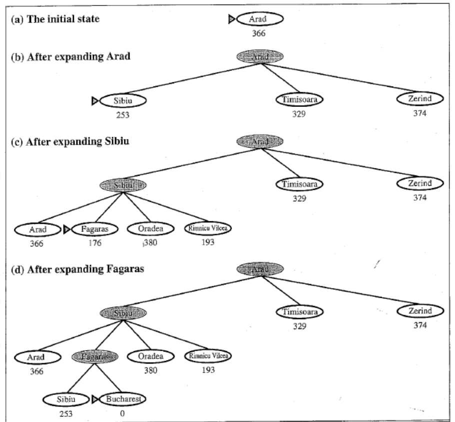


Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

- Sometimes greedy heuristics can be proved to be optimal
  - minimum spanning tree,
  - single source shortest path,
  - total weighted sum completion time in single machine scheduling,
  - single machine maximum lateness scheduling

- Other times an approximation ratio can be proved

Local Search Paradigm

- search space = complete candidate solutions
- search step = modification of one or more solution components
- neighborhood candidate solutions in the search space reachable in a step
- iteratively generate and evaluate candidate solutions
  - decision problems: evaluation = test if solution
  - optimization problems: evaluation = check objective function value

Iterative Improvement (II):

determine initial candidate solution s
while s has better neighbors **do**
    choose a neighbor s' of s such that $f(s') < f(s)$
    $s := s'$

Local Search Algorithm

Basic Components:

- solution representation $\rightsquigarrow$ search space
- initial solution
- neighborhood relation (determines the move operator)
- evaluation function

Outline

Software Tools

- Modeling languages
  interpreted languages with a precise syntax and semantics
- Software libraries
  collections of subprograms used to develop software
- Software frameworks
  set of abstract classes and their interactions
  - *frozen spots* (remain unchanged in any instantiation of the framework)
  - *hot spots* (parts where programmers add their own code)

Software Tools

No well established software tool for Local Search:

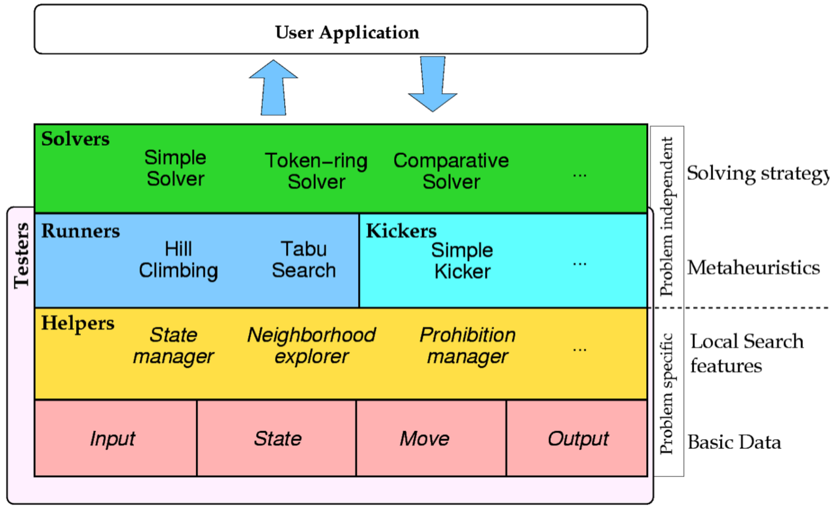
- the apparent simplicity of Local Search induces to build applications from scratch.
- crucial roles played by delta/incremental updates which is problem dependent
- the development of Local Search is in part a craft,
  beside engineering and science.
- lack of a unified view of Local Search.

Software Tools

EasyLocal++	C++, Java	Local Search
ParadisEO	C++	Local Search, Evolutionary Algorithm
OpenTS	Java	Tabu Search
Comet	–	Language

EasyLocal++	http://tabu.diegm.uniud.it/EasyLocal++/
ParadisEO	http://paradiseo.gforge.inria.fr
OpenTS	http://www.coin-or.org/Ots
Comet	http://dynadec.com/

A Framework



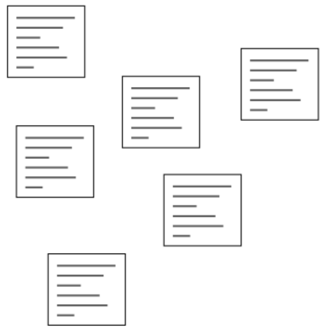
http://tabu.diegm.uniud.it/EasyLocal++/

Comet is

A programming language

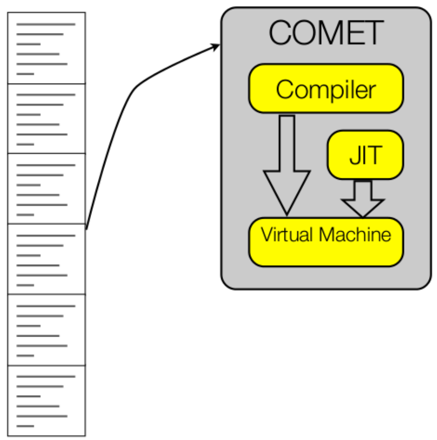
- Syntax inspired by C++
  - Object-oriented
  - Operator overloading
  - Filestreams
- Interpreted or Just-in-Time compiled
- Garbage collection
- High-level features
  - Invariants (one-way-constraints)
  - Closures
  - Functional programming-like constructions
    - List comprehension
    - `sum, select, selectMin, selectMax`
  - Sets, dictionaries, etc. are builtin types
  - Events

Workflow



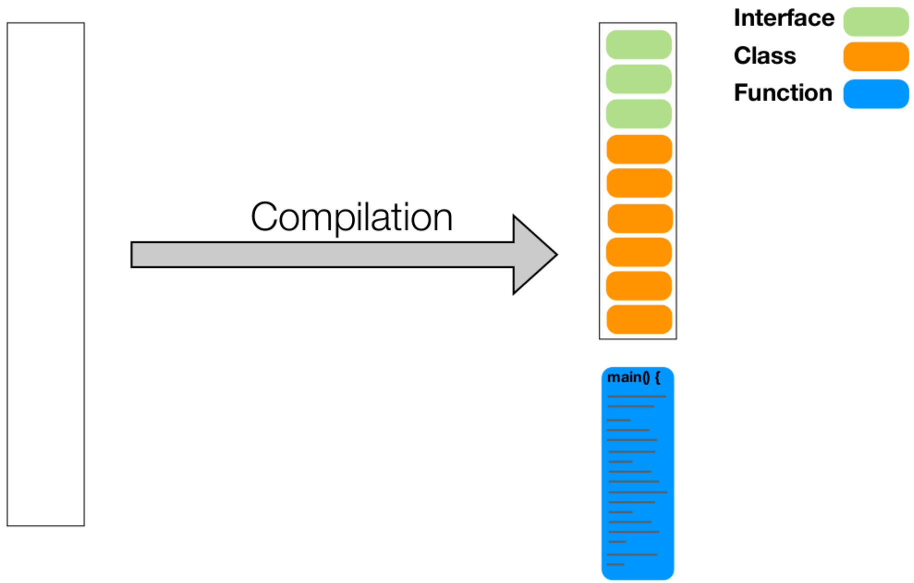
Workflow

Workflow

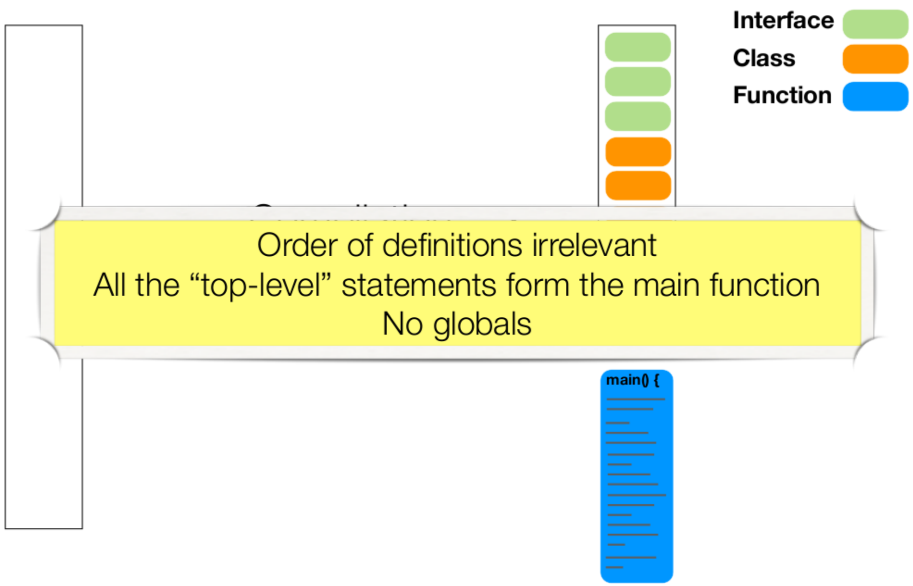


Source Organization

Source Organization



Source Organization



Comet is

A runtime environment

- With integrated optimization solvers
  - Constraint-Based Local Search
  - Constraint Programming
  - Linear Programming (COIN-OR CLP)
  - Mixed Integer Programming
- 2D graphics library
- Available for many platforms
  - Mac OS X (32 and 64 bit)
  - Windows
  - Linux (32 and 64 bit)
    - Ubuntu
    - SuSE
    - RedHat/Fedora

Comet is

Unfortunately not Open Source

Maintained and owned by Pascal Van Hentenryck (Brown University), Laurent Michel (University of Connecticut), Dynadec.

In active development

- Syntax is changing (faster than the documentation)
- Small bugs will be fixed fast
- Large bugs will be fixed
- Feature requests are always considered

Constraint Programming is

- Model
  - Variables
    - Domains
  - Objective Function
  - Constraints
- Search
  - Branching
    - Variable selection
    - Value selection
  - Search strategy
    - BFS
    - DFS
    - LDS

Constraint-Based Local Search is

- Model
  - Incremental variables
  - Invariants
  - Differentiable objects
    - Functions
    - Constraints
    - Constraint Systems
- Search
  - Local Search
    - Iterative Improvement
    - Tabu Search
    - Simulated Annealing
    - Guided Local Search

Incremental variables

- `var{int}, var{float}, var{bool}, var{set{int}}, ...`
- Attached to a model object
- Has a domain
- Has a value

Examples

```
Solver<LS> m();

var{int} x(m, 1..100);
var{bool} b[1..7](m);
var{set{int}} S(m);

x := 7;
S := {1,3,6,8};
```

Invariants

- *var* <- *expr*
- Also known as one-way constraints
- Defined over incremental variables
- Implicitly attached to a model object
- LHS variable value is maintained incrementally under changes to RHS variable values
- Can be user defined (by implementing Invariant<LS>)

Examples

```
var{int} x(m) := 7
var{int} y(m) <- (x+5)*x;
x <- y;                          // not allowed!!!
y := 3;                          // not allowed!!!
var{int} c[i in 1..n](m) := (i % 6);
var{int} C(m) <- sum(i in 1..n)(c[i]);
var{set{int}} Z(m) <- collect(i in n : c[i] == 0)(i);
var{int} q(m) <- c[x];
```


Differentiable objects

- Constraint<LS>
- ConstraintSystem<LS>
- Function<LS>

- Defined over incremental variables
- Implicitly attached to a model object
- Has a value (or a number of violations)
- Maintains value incrementally under changes to variable values
- Supports delta evaluations
- Can be user defined (by extending UserConstraint<LS>)

Constraint<LS>

Interface

```
int getAssignDelta(var{int},int)
int getAssignDelta(var{int}[],int[])
int getSwapDelta(var{int},var{int})
var{int}[] getVariables()
var{boolean} isTrue()
var{int} violations()
var{int} violations(var{int})
```


ConstraintSystem<LS> extends Constraint<LS>

A conjunction of constraints

Interface

```
Constraint<LS> post(expr{boolean})
Constraint<LS> post(expr{boolean},int)
Constraint<LS> post(Constraint<LS>)
Constraint<LS> post(Constraint<LS>,int)
```

ConstraintSystem<LS> extends Constraint<LS>

Examples

```
Solver<LS> m();
var{int} x[1..10](m);
var{int} y[1..10](m, 1..2);
int w[i in 1..10] = 2*i;
int C[1..2] = 95;

ConstraintSystem<LS> S(m);
S.post(x[1] >= 7);
S.post(sum(i in 3..7)(x[i]*x[i] <= x[10]);
S.post(AllDifferent<LS>(x));
S.post(Knapsack<LS>(y, w, C));
```


Function<LS>

Interface

```
int getAssignDelta(var{int},int)
int getSwapDelta(var{int},var{int})
var{int} flipDelta(var{boolean})
var{int} evaluation()
var{int} value()
var{int}[] getVariables()
var{int} increase(var{int})
var{int} decrease(var{int})
```


Function<LS>

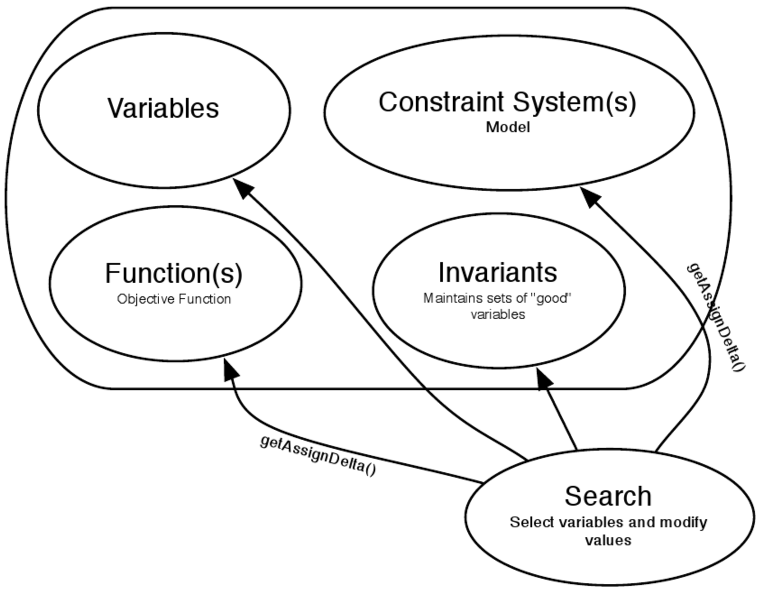
Examples

```
Solver<LS> m();

var{int} x(m, 1..10);

FunctionWrapper<LS> f1(x[1]*(7-x[2]);
FunctionWrapper<LS> f2(x[5]);
FunctionPower<LS>   f3(f2, 3);
FunctionTimes<LS>   f4(f2, f3);
FunctionSum<LS>     f5(m);
F.post(f1);
F.post(f2);
F.post(f3, 17);
F.post(x[10]-10);
F.close();
MinNbDistinct<LS>   f6(x);
```


Overview

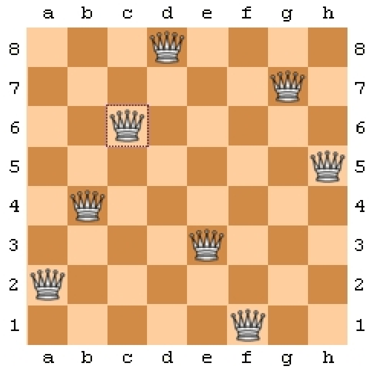


Example

N-Queens problem

Input: A chessboard of size $N \times N$

Task: Find a placement of n queens on the board such that no two queens are on the same row, column, or diagonal.



A CP Example

```
import cotfd;

int t0 = System.getCPUTime();
Solver<CP> m();
int n = 8;
range S = 1..n;
var<CP>{int} q[i in S](m,S);
Integer c(0);
solve<m> {
  m.post(alldifferent(all(i in S) q[i] + i));
  m.post(alldifferent(all(i in S) q[i] - i));
  m.post(alldifferent(q));
} using {
  forall(i in S : !q[i].bound()) by (q[i].getSize())
    tryall<m>(v in S : q[i].memberOf(v))
      m.post(q[i] == v);
  onFailure m.post(q[i]!=v);
  cout << q << endl;
  c := c + 1;
}

cout << "Nb␣=␣" << c << endl;
cout << "Time␣=␣" << System.getCPUTime() - t0 << endl;
cout << "#choices␣=␣" << m.getNChoice() << endl;
cout << "#fail␣=␣" << m.getNFail() << endl;
```


An LS Example

```
import cotls;
int n = 16;
range Size = 1..n;
UniformDistribution distr(Size);

Solver<LS> m();
var{int} queen[Size](m,Size) := distr.get();
ConstraintSystem<LS> S(m);

S.post(alldifferent(queen));
S.post(alldifferent(all(i in Size) queen[i] + i));
S.post(alldifferent(all(i in Size) queen[i] - i));
m.close();

int it = 0;
while (S.violations() > 0 && it < 50 * n) {
  select(q in Size,v in Size : S.getAssignDelta(queen[q],v) < 0) {
    queen[q] := v;
    cout<<"change:␣queen["<<q<<"]␣:=␣"<<v<<"␣viol:␣"<<S.violations() <<endl;
  }
  it = it + 1;
}
cout << queen << endl;
```


How to learn more

Comet Tutorial in the Comet distribution	*Constraint-Based Local Search* P. Van Hentenryck, L. Michel MIT Press, 2005 ISBN-10: 0-262-22077-6

- Implement, experiment, fail, think, try again!
- See: http://www.imada.sdu.dk/ marco/Misc/comet.html
- Ask: http://forums.dynadec.com

Summary

- Modeling (from previous lecture)
- (High level) Construction Heuristics
- (High level) Local Search
- Development framework
- Comet

Outlook

- Working Environment
- Construction Heuristics
- Examples for the TSP