DM811

Heuristics for Combinatorial Optimization

Lecture 2

Introductory Topics

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Outline

Problem Solving

Problem solving is a mental process considered the most complex of all intellectual functions.

Move from a given state to a desired goal state using the knowledge we have but "inquiring in what we do not know" (Plato).
Often solutions seem to be original and creative.

Theories:

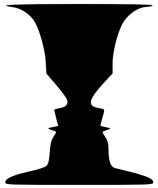
1. Gestalt approach
2. Problem space theory (Information-processing theory)

Gestalt Approach

The process of problem solving is

Behaviourists: reproduction of known responses, trial and error process

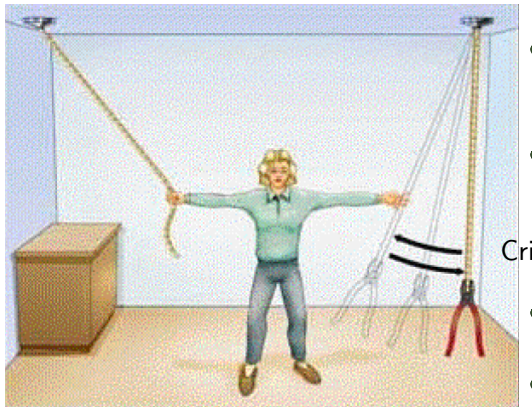
Gestalt school: (German psychologists in 20-30's concerned with experience as a whole rather than composed of parts)



- reproductive: draws on previous experience (may cause fixation and hinder the solution)
- productive: insight and problem restructuring

Gestalt Approach

Maier's Experiment (1931): pendulum problem



- Those who solved it rarely reported the clue
- Unconscious clue can lead to problem restructuring and insight

Criticism:

- unspecified and vague
- descriptive nature, not normative or explanatory (what processes are involved?)

Representational Theory

Incorporate Gestalt ideas into a working theory [Ohlsson, 1992]

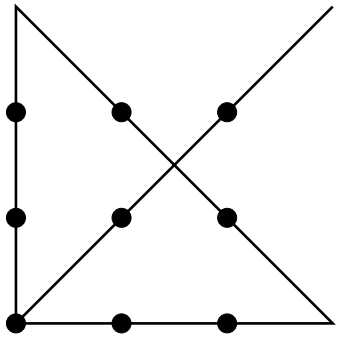
- A problem is represented in a certain way in the person's mind and this serves as a source of information from long-term memory
- The retrieval process spreads activation over relevant long term memory items
- A block occurs if the way a problem is represented does not lead to a helpful memory search
- The way the problem is represented changes and the memory search is extended, making new information available
- Representational change can occur due to elaboration (addition of new information) constraint relaxation (rules are reinterpreted) or re-encoding (fixedness is removed)
- Insight occurs when a block is broken and retrieved knowledge results in solution

Representational Theory

Draw four straight lines to join all the dots without taking the pen off the page

This problem was given to employees at Disney as is reportedly the origin of the expression "thinking outside the box"

Who failed probably did not consider extending the lines beyond the grid
➨ Constraint relaxation



Problem Space Approach

Information-processing theory:

> [A. Newell and H.A. Simon. *Computer science as empirical inquiry: symbols and search*. Communications of the ACM, 1976]

- human mind as symbolic system.
- generating problems states in the problem space using legal transition operators to go from an initial state to a goal state.
- limitations imposed by human processing system (limited short-term memory and speed).
- maximization heuristic: reduce difference between initial and goal state.
- progress monitoring: assessment of rate of progress

Further Elements

- Experience helps us since we can learn how to structure problems space and appropriate operators.
- Analogy (old knowledge is used to solve new problems)
- Domain knowledge and skill acquisition
  - Observation of expert vs novice in chess
    - chess masters remember board configurations
    - structure available to maintain configurations in short term memory
    - grouping of problems according to underlying conceptual similarities
    - better encoding of knowledge and easier information retrieval
  - skill acquisition:
    - general-purpose rules
    - rules to specific task
    - rules are tuned to speed up

Further reading:

- A. Newell and H.A. Simon. Computer science as empirical inquiry: symbols and search. Communications of the ACM, ACM, 1976, 19(3), 113-126
- A. Dix, J. Finlay, G.D. Abowd and R. Beale. Human-Computer Interaction. Pearson, Prentice Hall, 2004. (Chapter 1)
- Ormerod, T. MacGregor, J. Chronicle, E. (2002) Dynamics and Constraints in Insight Problem Solving. Journal of Experimental Psychology: Learning, Memory, and Cognition vol. 28 (4) pp 791-799

Outline

Constraint Satisfaction Problem

- **Input:**
  - a set of variables $X_1, X_2, \ldots, X_n$
  - each variable has a non-empty domain D_i of possible values
  - a set of constraints. Each constraint C_i involves some subset of the variables and specifies the allowed combination of values for that subset.

    [A constraint C on variables X_i and X_j, $C(X_i, X_j)$, defines the subset of the Cartesian product of variable domains $D_i \times D_j$ of the consistent assignments of values to variables. A constraint C on variables X_i, X_j is satisfied by a pair of values v_i, v_j if $(v_i, v_j) \in C(X_i, X_j)$.]
- **Task:**
  - find an assignment of values to all the variables $\{X_i = v_i, X_j = v_j, \ldots\}$
  - such that it is consistent, that is, it does not violate any constraint

If assignments are not all equally good but some are preferable this is reflected in an objective function.

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 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

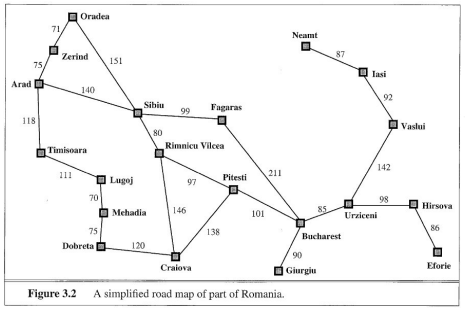


Figure 3.2 A simplified road map of part of Romania.

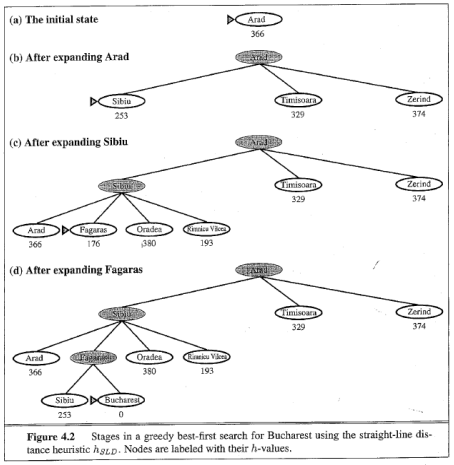


Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

- 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 (given problem instance π):

- search space (solution representation) $S(\pi)$
- initialization function $\texttt{init} : \emptyset \mapsto \mathcal{P}(S(\pi))$
- neighborhood relation $\mathcal{N}(\pi) \subseteq S(\pi) \times S(\pi)$
  (determines the move operator)
- evaluation function $f(\pi) : S \mapsto \mathbf{R}$

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/

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

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

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];
```

Diffentiable 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)
Constraint<LS> satisfy(expr{boolean})
Constraint<LS> satisfy(expr{boolean},int)
Constraint<LS> satisfy(Constraint<LS>)
Constraint<LS> satisfy(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));
```

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})
```

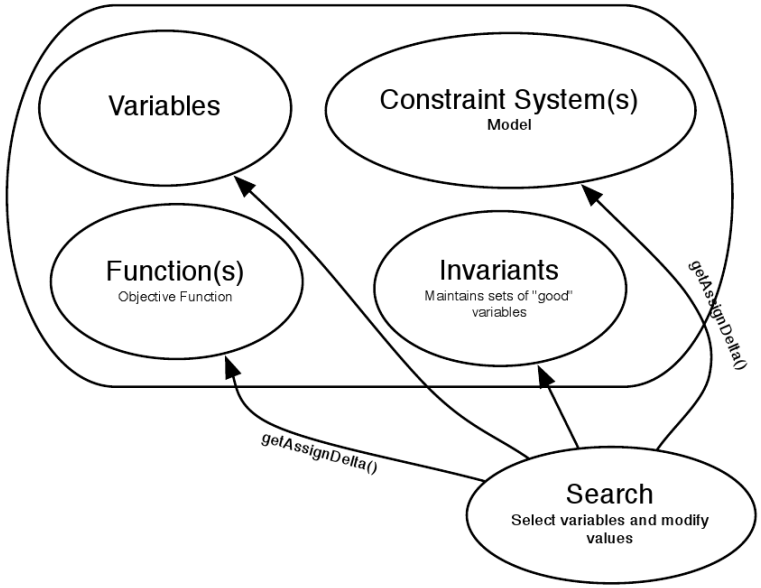

Examples

```
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



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/Teaching/-Fall2010/DM811/resources.html
- Ask: http://forums.dynadec.com

Outline

Graphs

Graphs are combinatorial structures useful to model several applications

Terminology:

- $G = (V, E)$, $E \subseteq V \times V$, vertices, edges, $n = |V|, m = |E|$, digraphs, undirected graphs, subgraph, induced subgraph
- $e = (u, v) \in E$, e incident on u and v; u, v adjacent, edge weight or cost
- particular cases often omitted: self-loops, multiple parallel edges
- degree, δ, Δ, outdegree, indegree
- path $P =< v_0, v_1, \ldots, v_k >$, $(v_0, v_1) \in E, \ldots, (v_{k-1}, v_k) \in E$, $< v_0, v_1 >$ has length 2, $< v_0, v_1, v_2, v_0 >$ cycle, walk, path
- directed acyclic digraph
- digraph strongly connected ($\forall u, v \ \exists (uv)$-path), strongly connected components
- G is a tree ($\exists$ path between any two vertices) $\iff$ G is connected and has $n - 1$ edges $\iff$ G is connected and contains no cycles.
- parent, children, sibling, height, depth

Representing Graphs

Operations:

- Access associated information (NodeArray, EdgeArray, Hashes)
- Navigation: access outgoing edges
- Edge queries: given u and v is there an edge?
- Update: add remove edges, vertices

Data Structures:

- Edge sequences
- Adjacency arrays
- Adjacency lists
- Adjacency matrix

How to choose?

- it depends on the graphs and the application
- if time and space not crucial no need to customize the structures
- use interfaces that make easy to change the data structure
- libraries offer different choices (LEDA, Java `jdsl.graph`)

Summary

- (Graphs)
- Constraint Satisfaction Probelm
- Construction Heuristics and Local Search
- Working environment organization
- Comet

Outlook

- How to analyse results
- Short introduction to R