Lecture 2

Introductory Topics

Marco Chiarandini

Deptartment 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:

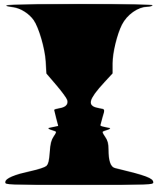
1. Gestalt approach
2. Problem space theory (Information-processing theory)

Gestalt Approach

The process of problem solving is

Behaviourists: reproduction of known responses, trial and error process

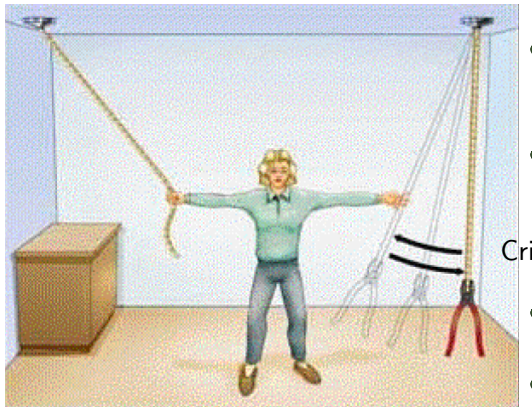
Gestalt school: (German psychologists in 20-30's concerned with experience as a whole rather than composed of parts)



- 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 cue
- 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]

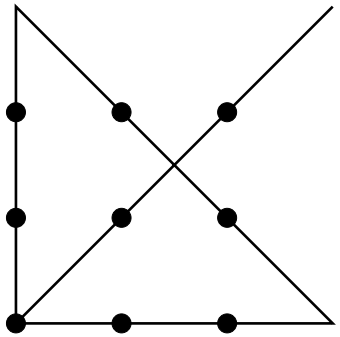
- A problem is represented in a certain way in the person's mind and this serves as a source of information from long-term memory
- The retrieval process spreads activation over relevant long term memory items
- A block occurs if the way a problem is represented does not lead to a helpful memory search
- 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, 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
- with largest min-regret
- with smallest max-regret
- with largest max-regret

Designing Constr. Heuristics

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

Greedy best-first search

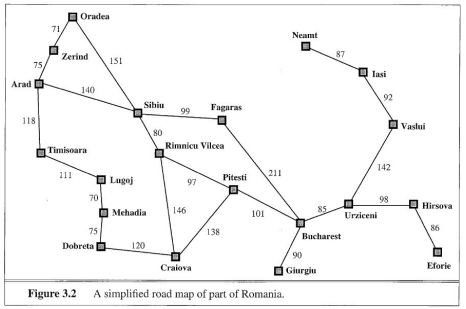


Figure 3.2 A simplified road map of part of Romania.

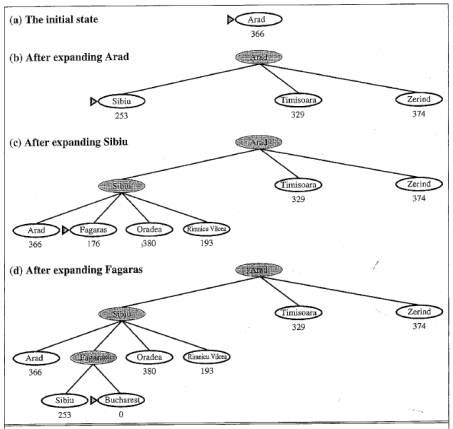


Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

- 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

Building a Work Environment

You will need these files for your project:

- The code that implements the algorithm. (Several versions.)
- The input:
  Instances for the algorithm, parameters to guide the algorithm, instructions for reporting.
- The output:
  The result, the performance measurements, perhaps animation data.
- The journal:
  A record of your experiments and findings.
- Analysis tools:
  statistics, data analysis, visualization, report.

How will you organize them? How will you make them work together?

Example

Input and reporting controls on command line

```
gcp -i instance.in -o output.sol -l run.log > data.out
```

Output on stdout self-describing

```
#stat instance.in 30 90
seed: 9897868
Parameter1: 30
Parameter2: A
Read instance. Time: 0.016001
begin try 1
best 0 col 22 time 0.004000 iter 0 par_iter 0
best 3 col 21 time 0.004000 iter 0 par_iter 0
best 1 col 21 time 0.004000 iter 0 par_iter 0
best 0 col 21 time 0.004000 iter 1 par_iter 1
best 6 col 20 time 0.004000 iter 3 par_iter 1
best 4 col 20 time 0.004000 iter 4 par_iter 2
best 2 col 20 time 0.004000 iter 6 par_iter 4
exit iter 7 time 1.000062
end try 1
```

Example

If one program that implements many heuristics

- re-compile for new versions but take old versions with a journal in archive.
- use command line parameters to choose among the heuristics
- C: `getopt`, `getopt_long`, `opag` (option parser generator)
  Java: package `org.apache.commons.cli`

```
gcp -i instance.in -o output.sol -l run.log --solver 2-opt > data.out
```

- use identifying labels in naming file outputs

Example

- So far: one run per instance. Multiple runs, multiple instances and multiple algorithms ➨ unix script (eg, bash one line program, perl, php)
- Data analysis: Select line identifier from output file, combine, send to grasp scripts.
  Example

```
grep #stat | cut -f 2 -d " "
```

- Data in form of matrix or data frame goes directly into R imported by `read.table()`, untouched by human hands!

```
alg instance     run sol time
ROS le450_15a.col 3 21 0.00267
ROS le450_15b.col 3 21 0
ROS le450_15d.col 3 31 0.00267
RLF le450_15a.col 3 17 0.00533
RLF le450_15b.col 3 16 0.008
...
```

- Visualization: Select animation commands from output file, send to animation tool (ex. graphviz, igraph).

Program Profiling

- Check the correctness of your solutions many times
- Plot the development of
  - best visited solution quality
  - current solution quality

  over time and compare with other features of the algorithm.

Code Optimization

- Profile time consumption per program components
  - under Linux: `gprof`
    1. add flag `-pg` in compilation
    2. run the program
    3. `gprof gmon.out > a.txt`
  - Java VM profilers (plugin for eclipse)

Software Development
Extreme Programming & Scrum

Planning

Release planning creates the schedule // Make frequent small releases // The project is divided into iterations

Designing

Simplicity // No functionality is added early // Refactor: eliminate unused functionality and redundancy

Coding

Code must be written to agreed standards // Code the unit test first // All production code is pair programmed // Leave optimization till last // No overtime

Testing

All code must have unit tests // All code must pass all unit tests before it can be released // When a bug is found tests are created

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:

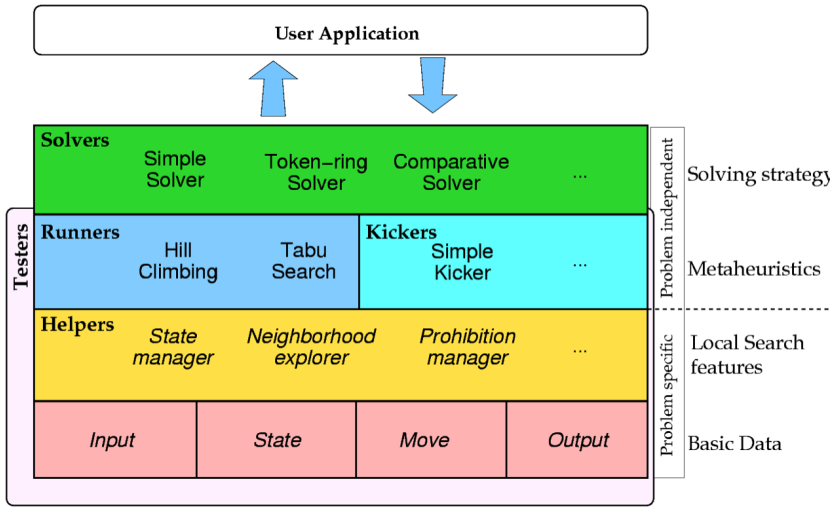
- the apparent simplicity of Local Search induces to build applications from scratch.
- crucial roles played by delta/incremental updates which is problem dependent
- the development of Local Search is in part a craft, beside engineering and science.
- lack of a unified view of Local Search.

Software Tools

EasyLocal++	C++, Java	LS
ParadisEO	C++	EA, LS
OpenTS	Java	TS
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++/

An Example

Input (util.h, util.c)

```
typedef struct {
  long int number_jobs; /* number of jobs in instance */
  long int release_date[MAX_JOBS];
  long int proc_time[MAX_JOBS];
  long int weight[MAX_JOBS];
  long int due_date[MAX_JOBS];
} instance_type;

instance_type instance;

void read_problem_size (char name[100])
void read_instances (char input_file_name[100])
```


State/Solution (util.h)

```
typedef struct {
  long int job_at_pos[MAX_JOBS]; /* Gives the job at a certain pos */
  long int pos_of_job[MAX_JOBS]; /* Gives the position of a specific job */
  long int completion_time_job[MAX_JOBS]; /* Gives C_j of job j */
  long int start_time_job[MAX_JOBS]; /* Gives start time of job j */
  long int tardiness_job[MAX_JOBS]; /* Gives T_j of job j */
  long int value; /* Objective function value */
} sol_representation;

sol_representation sequence;
```

Output (util.c)

```
void print_sequence (long int k)
void print_completion_times ()
```

State Manager (util.c)

```
void construct_sequence_random ()
void construct_sequence_canonical ()
long int evaluate ()
```


Random Generator (random.h, random.c)

```
void set_seed (double arg)
double MRG32k3a (void)
double ranU01 (void)
int ranUint (int i, int j)
void shuffle (int *X, int size)
```

Timer (timer.c)

```
double getCurrentTime ()
```

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 $P = < v_0, v_1, \ldots, v_k >$, $(v_0, v_1) \in E, \ldots, (v_{k-1}, v_k) \in E$, $< v_0, \ldots, v_1 >$ 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`)