DM811
Heuristics for Combinatorial Optimization

Lecture 4

Solver Systems + Construction Heuristics and Metaheuristics

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Course Overview

- ✔ Combinatorial Optimization, Methods and Models
- ✔ CH and LS: overview
- ~ Working Environment and Solver Systems
- ~ Methods for the Analysis of Experimental Results
- Construction Heuristics
- Local Search: Components, Basic Algorithms
- Local Search: Neighborhoods and Search Landscape
- Efficient Local Search: Incremental Updates and Neighborhood Pruning
- Stochastic Local Search & Metaheuristics
- Configuration Tools: F-race
- Very Large Scale Neighborhoods

Examples: GCP, CSP, TSP, SAT, MaxIndSet, SMTWP, Steiner Tree, p-median, set covering

Outline

1. Software Tools
   - Constraint-Based Local Search with Comet^{TM}
2. Descriptions
3. Construction Heuristics
   - Complete Search Methods
   - Incomplete Search Methods
4. Metaheuristics
   - Random Restart
   - Rollout/Pilot Method
   - Beam Search
   - Iterated Greedy
   - GRASP

Outline

1. Software Tools
    - Constraint-Based Local Search with Comet™
2. Descriptions
3. Construction Heuristics
    - Complete Search Methods
    - Incomplete Search Methods
4. Metaheuristics
    - Random Restart
    - Rollout/Pilot Method
    - Beam Search
    - Iterated Greedy
    - GRASP

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:

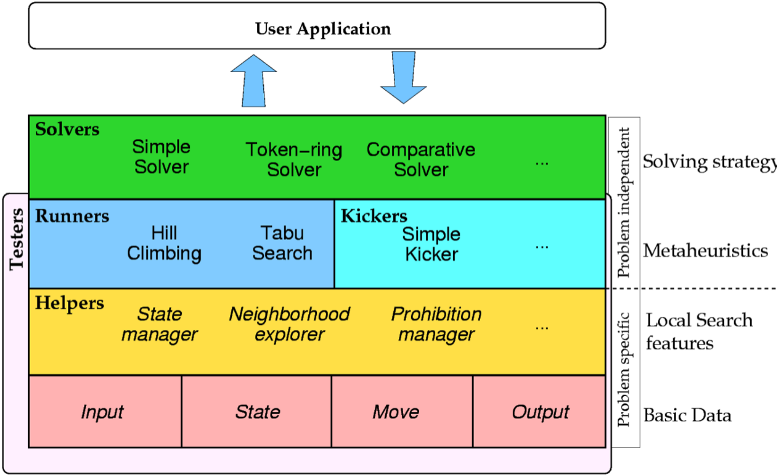
- the apparent simplicity of Local Search induces to build applications from scratch.
- the freedom of problem characteristics that can be tackled
- crucial roles played by delta/incremental updates which are highly problem dependent
- the development of Local Search is in part a craft, beside engineering 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	
LocalSolver	Language	
Google OR Tools	Libraries	

EasyLocal++	http://tabu.diegm.uniud.it/EasyLocal++/
ParadisEO	http://paradiseo.gforge.inria.fr
OpenTS	http://www.coin-or.org/Ots
Comet	http://dynadec.com/
LocalSolver	http://www.localsolver.com/
Google OR Tools	https://code.google.com/p/or-tools/

A Framework



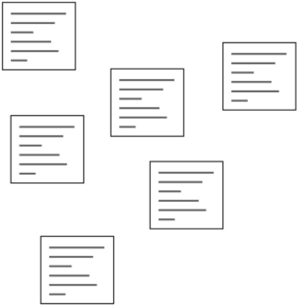
http://tabu.diegm.uniud.it/EasyLocal++/

Comet is

A programming language

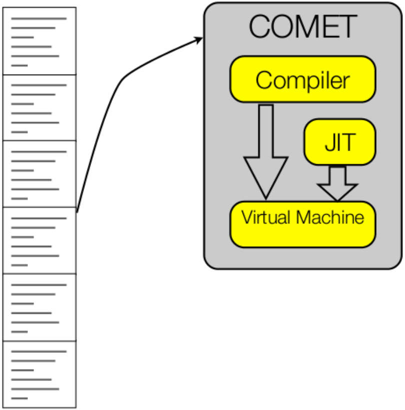
- Syntax inspired by C++
  - Object-oriented
  - Operator overloading
  - Filestreams
- Interpreted or Just-in-Time compiled
- Garbage collection
- High-level features
  - Invariants (one-way-constraints)
  - Closures
  - Functional programming-like constructions
    - List comprehension
    - `collect`,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

Workflow



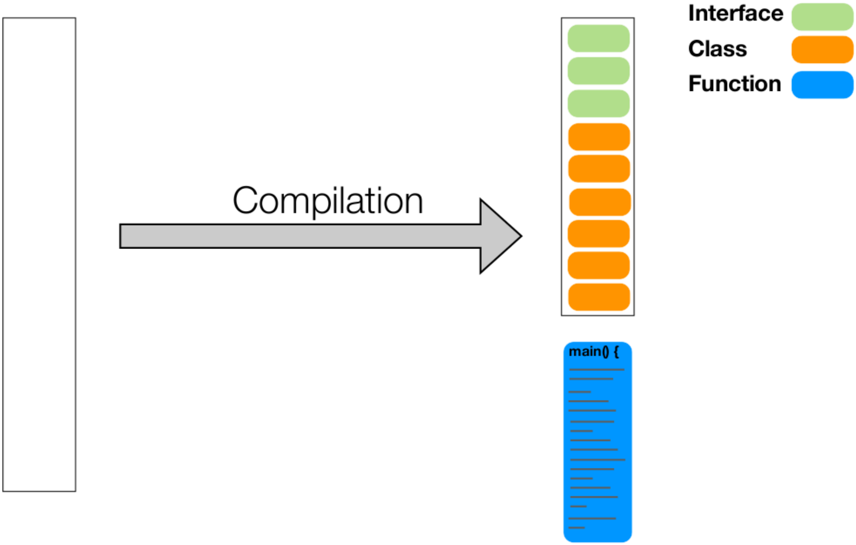
Workflow

Workflow

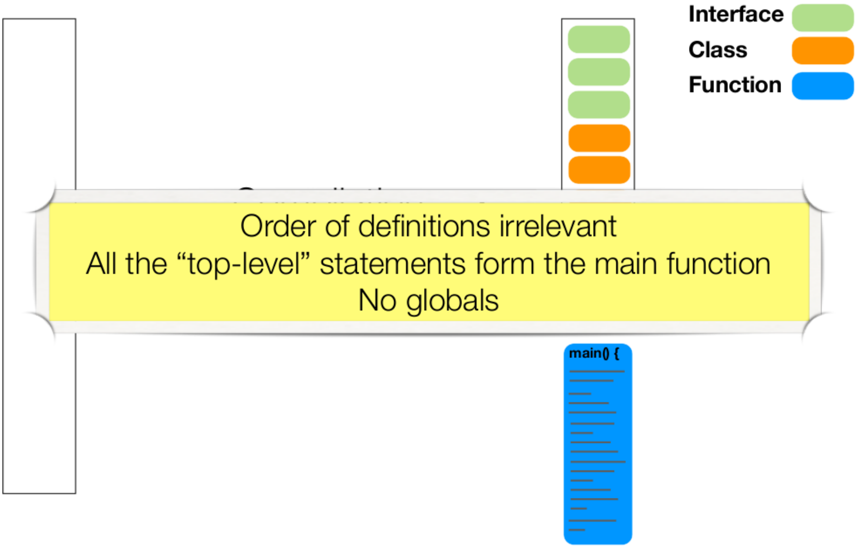


Source Organization

Source Organization



Source Organization



Order of definitions irrelevant
All the “top-level” statements form the main function
No globals

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

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

Not anymore in active development

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{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)
```

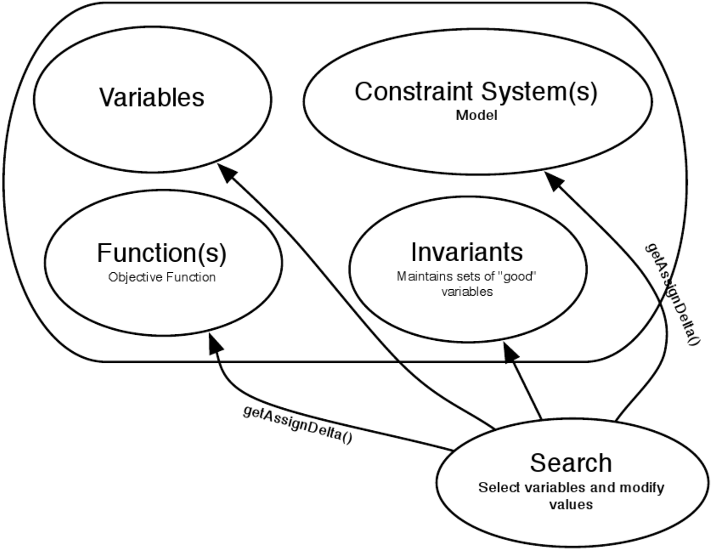
ConstraintSystem<LS> extends Constraint<LS>

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));
```

Overview

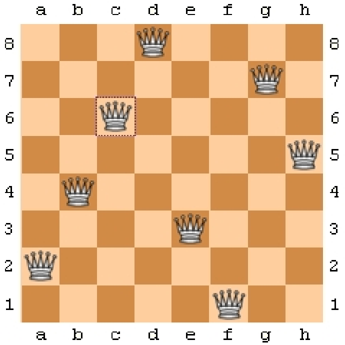

Variables
Constraint System(s)
Model
Function(s)
Objective Function
Invariants
Maintains sets of "good" variables
getAssignDelta()
getAssignDelta()
Search
Select variables and modify values

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<<"chng @ "<<it<<": 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

Outline

Guidelines for Text Writing

- Outline:
    1. word (discursive) description
    2. precise algorithm using mathematical notation and pseudo-code
    3. implementation details, ie, abstract data structures
    4. computational (runtime, space) analysis
- Refer to floating environments like Algorithms and Figures that you present in the text
- Cite your sources in a proper and detailed way, they must be retrievable by the reader. If you do not do it then you are committing plagiarism.
- Before submitting: run spell checker and *then* read again and again and again
- Mathematical notation makes things clearer and precise and the overall descriptions more concise. (but use latex!)
- As a reader you should ask yourself whether you would be able to reproduce the algorithm in exactly the same way as described.

Outline

Complete Search Methods

Tree search:

Uninformed Search

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional Search

Informed Search

- best-first search, aka, greedy search
- A^* search
- Iterative Deepening A^*
- Memory bounded A^*
- Recursive best first

Constraint Satisfaction and Backtracking

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

- Select-Initial-Unassigned-Variable
- Select-Unassigned-Variable
  - most constrained first = fail-first heuristic
    = Minimum remaining values (MRV) heuristic
    (tend to reduce the branching factor and to speed up pruning)
  - least constrained last

  Eg.: max degree, farthest, earliest due date, etc.
- Order-Domain-Values
  - greedy
  - least constraining value heuristic
    (leaves maximum flexibility for subsequent variable assignments)
  - maximal regret
    implements a kind of look ahead

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

Propagating information through constraints:

- Implicit in Select-Unassigned-Variable
- Forward checking (coupled with Minimum Remaining Values)
- Constraint propagation in CSP
  - arc consistency: force all (directed) arcs uv to be consistent:
    $\exists$ a value in $D(v)$: $\forall$ values in $D(u)$, otherwise detects inconsistency

    can be applied as preprocessing or as propagation step after each assignment (Maintaining Arc Consistency)

    Applied repeatedly

    [Can you find preprocessing rules for the graph coloring problem?]

Propagation: An Example

	WA	*NT*	*Q*	*NSW*	*V*	*SA*	*T*
Initial domains	R G B	R G B	R G B	R G B	R G B	R G B	R G B
After *WA*=*red*	R	G B	R G B	R G B	R G B	G B	R G B
After *Q*=*green*	R	B	G	R B	R G B	B	R G B
After *V*=*blue*	R	B	G	R	B		R G B

Figure 5.6 The progress of a map-coloring search with forward checking. $WA = red$ is assigned first; then forward checking deletes *red* from the domains of the neighboring variables NT and SA. After $Q = green$, *green* is deleted from the domains of NT, SA, and NSW. After $V = blue$, *blue* is deleted from the domains of NSW and SA, leaving SA with no legal values.

3) 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?

Backtracking-Search

- chronological backtracking, the most recent decision point is revisited
- backjumping, backtracks to the most recent variable in the conflict set (set of previously assigned variables connected to X by constraints).

Dealing with Objectives
Optimization Problems

A* search

- The priority assigned to a node x is determined by the function

$$f(x) = g(x) + h(x)$$

  $g(x)$: cost of the path so far
  $h(x)$: heuristic estimate of the minimal cost to reach the goal from x.

- It is optimal if $h(x)$ is an
  - admissible heuristic: *never overestimates* the cost to reach the goal
  - consistent: $h(n) \leq c(n, a, n') + h(n')$

A* search

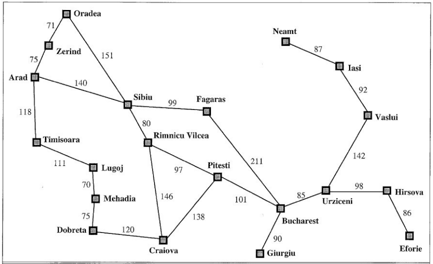


Figure 3.2 A simplified road map of part of Romania.

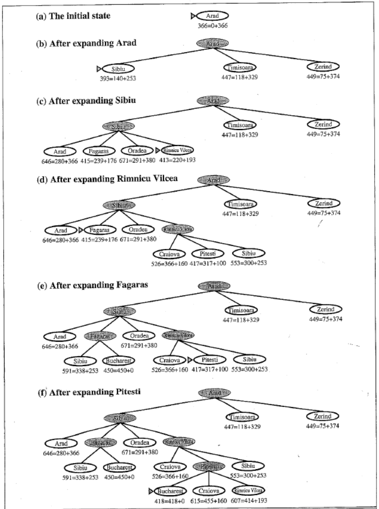


Figure 4.3 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

A* search

Possible choices for admissible heuristic functions

- optimal solution to an easily solvable **relaxed problem**
- optimal solution to an easily solvable **subproblem**
- learning from experience by gathering statistics on state features
- preferred heuristics functions with higher values (provided they do not overestimate)
- if several heuristics available $h_1, h_2, \ldots, h_m$ and not clear which is the best then:

$$h(x) = \max\{h_1(x), \ldots, h_m(x)\}$$

A* search

Drawbacks

- Time complexity: In the worst case, the number of nodes expanded is exponential,
  (but it is polynomial when the heuristic function h meets the following condition:
  $$|h(x) - h^*(x)| \leq O(\log h^*(x))$$
  h^* is the optimal heuristic, the exact cost of getting from x to the goal.)
- Memory usage: In the worst case, it must remember an exponential number of nodes.
  Several variants: including iterative deepening A* (IDA*), memory-bounded A* (MA*) and simplified memory bounded A* (SMA*) and recursive best-first search (RBFS)

Incomplete Search

Complete search is often better suited when ...

- proofs of insolubility or optimality are required;
- time constraints are not critical;
- problem-specific knowledge can be exploited.

Incomplete search is the necessary choice when ...

- non linear constraints and non linear objective function;
- reasonably good solutions are required within a short time;
- problem-specific knowledge is rather limited.

Greedy algorithms

Greedy algorithms (derived from best-first)

- Strategy: always make the choice that is best at the moment
- Single descent in the search tree
- They are not generally guaranteed to find globally optimal solutions (but sometimes they do: Minimum Spanning Tree, Single Source Shortest Path, etc.)

We will see problem sepcific examples

Outline

Metaheuristics

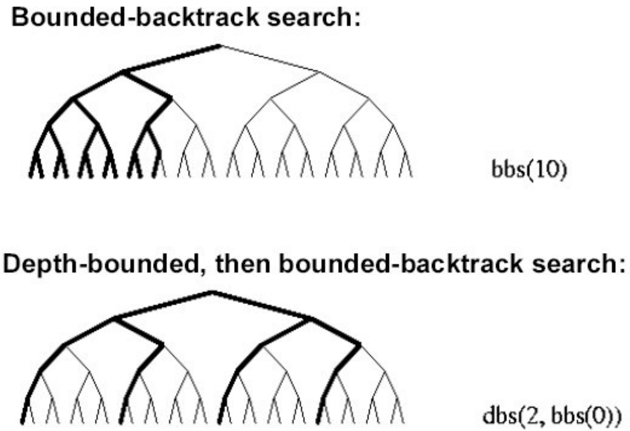
On backtracking framework
(beyond best-first search)

- Random Restart
- Bounded backtrack
- Credit-based search
- Limited Discrepancy Search
- Barrier Search
- Randomization in Tree Search

Outside the exact framework
(beyond greedy search)

- Random Restart
- Rollout/Pilot Method
- Beam Search
- Iterated Greedy
- GRASP
- (Adaptive Iterated Construction Search)
- (Multilevel Refinement)

Bounded backtrack

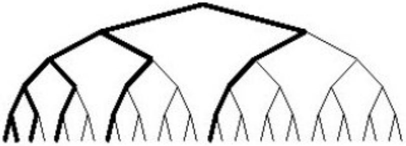


http://4c.ucc.ie/~hsimonis/visualization/techniques/partial_search/main.htm

Limited Discrepancy Search

Limited Discrepancy Search (LDS)

- Key observation that often the heuristic used in the search is nearly always correct with just a few exceptions.
- Explore the tree in increasing number of discrepancies, modifications from the heuristic choice.
- Eg: count one discrepancy if second best is chosen
  count two discrepancies either if third best is chosen or twice the second best is chosen
- Control parameter: the number of discrepancies



Randomization in Tree Search

The idea comes from complete search: the important decisions are made up in the search tree (backdoors) $\rightsquigarrow$ random selections + restart strategy

Random selections

- randomization in variable ordering:
  - breaking ties at random
  - use heuristic to rank and randomly pick from small factor from the best
  - random pick among heuristics
  - random pick variable with probability depending on heuristic value
- randomization in value ordering:
  - just select random from the domain

Restart strategy in backtracking

- Example: $S_u = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 4, 8, 1, \ldots)$

Rollout/Pilot Method

Derived from A*

- Each candidate solution is a collection of m components $S = (s_1, s_2, \ldots, s_m)$.
- Master process adds components sequentially to a partial solution $S_k = (s_1, s_2, \ldots s_k)$
- At the k-th iteration the master process evaluates feasible components to add based on an heuristic look-ahead strategy.
- The evaluation function $H(S_{k+1})$ is determined by sub-heuristics that complete the solution starting from S_k
- Sub-heuristics are combined in $H(S_{k+1})$ by
  - weighted sum
  - minimal value

Speed-ups:

- halt whenever cost of current partial solution exceeds current upper bound
- evaluate only a fraction of possible components

Beam Search

Again based on tree search:

- maintain a set B of bw (beam width) partial candidate solutions
- at each iteration extend each solution from B in fw (filter width) possible ways
- rank each $bw \times fw$ candidate solutions and take the best bw partial solutions
- complete candidate solutions obtained by B are maintained in B_f
- Stop when no partial solution in B is to be extended

Iterated Greedy

(aka, Adaptive Large Neighborhood Search, see later)

Key idea: use greedy construction

- alternation of construction and deconstruction phases
- an acceptance criterion decides whether the search continues from the new or from the old solution.

Iterated Greedy (IG):
determine initial candidate solution s
while termination criterion is not satisfied **do**
  $r := s$
  (randomly or heuristically) destruct part of s
  greedily reconstruct the missing part of s
  based on acceptance criterion,
    keep s or revert to $s := r$

GRASP

Greedy Randomized Adaptive Search Procedure

Key Idea: Combine randomized constructive search with subsequent local search.

Motivation:

- Candidate solutions obtained from construction heuristics can often be substantially improved by local search.
- Local search methods often require substantially fewer steps to reach high-quality solutions when initialized using greedy constructive search rather than random picking.
- By iterating cycles of constructive + local search, further performance improvements can be achieved.

Greedy Randomized "Adaptive" Search Procedure (GRASP):

while *termination criterion* is not satisfied **do**
  generate candidate solution s using
    subsidiary greedy randomized constructive search
  perform subsidiary local search on s

- Randomization in *constructive search* ensures that a large number of good starting points for *subsidiary local search* is obtained.
- Constructive search in GRASP is 'adaptive' (or dynamic): Heuristic value of solution component to be added to a given partial candidate solution may depend on solution components present in it.
- Variants of GRASP without local search phase (aka *semi-greedy heuristics*) typically do not reach the performance of GRASP with local search.

Restricted candidate lists (RCLs)

- Each step of *constructive search* adds a solution component selected uniformly at random from a restricted candidate list (RCL).
- RCLs are constructed in each step using a *heuristic function* h.
  - RCLs based on cardinality restriction comprise the k best-ranked solution components. (k is a parameter of the algorithm.)
  - RCLs based on value restriction comprise all solution components l for which $h(l) \leq h_{min} + \alpha \cdot (h_{max} - h_{min})$,
    where h_{min} = minimal value of h and h_{max} = maximal value of h for any l. (α is a parameter of the algorithm.)
  - Possible extension: reactive GRASP (*e.g.*, dynamic adaptation of α during search)

Example: Squeaky Wheel

Key idea: solutions can reveal problem structure which maybe worth to exploit.

Use a greedy heuristic repeatedly by prioritizing the elements that create troubles.

Squeaky Wheel

- Constructor: greedy algorithm on a sequence of problem elements.
- Analyzer: assign a penalty to problem elements that contribute to flaws in the current solution.
- Prioritizer: uses the penalties to modify the previous sequence of problem elements. Elements with high penalty are moved toward the front.

Possible to include a local search phase between one iteration and the other