DM811

Heuristics for Combinatorial Optimization

Lecture 5

Metaheuristics based on Construction Heuristics

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

1. Metaheuristics
   - Rollout/Pilot Method
   - Beam Search
   - Iterated Greedy
   - GRASP
   - Adaptive Iterated Construction Search
   - Multilevel Refinement
2. Work Environment
   - Organization
3. Bin Packing

Outline

1. Metaheuristics
   - Rollout/Pilot Method
   - Beam Search
   - Iterated Greedy
   - GRASP
   - Adaptive Iterated Construction Search
   - Multilevel Refinement
2. Work Environment
   - Organization
3. Bin Packing

Metaheuristics

On backtracking framework
(beyond best-first search)

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

Outside the exact framework
(beyond greedy search)

- Rollout/Pilot Method
- Beam Search
- Iterated Greedy
- GRASP
- Adaptive Iterated Construction Search
- Multilevel Refinement

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

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 $s := r$

Extension: 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

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)

Adaptive Iterated Construction Search

Key Idea: Alternate construction and local search phases as in GRASP, exploiting experience gained during the search process.

Realisation:

- Associate *weights* with possible decisions made during constructive search.
- Initialize all weights to some small value τ_0 at beginning of search process.
- After every cycle (= constructive + local local search phase), update weights based on solution quality and solution components of current candidate solution.

Adaptive Iterated Construction Search (AICS):
initialise weights
while *termination criterion* is not satisfied: **do**
  generate candidate solution s using
    subsidiary randomized constructive search
  perform subsidiary local search on s
  adapt weights based on s

Subsidiary constructive search:

- The solution component to be added in each step of *constructive search* is based on i) *weights* and ii) heuristic function h.
- h can be standard heuristic function as, *e.g.*, used by greedy heuristics
- It is often useful to design solution component selection in constructive search such that any solution component may be chosen (at least with some small probability) irrespective of its weight and heuristic value.

Subsidiary local search:

- As in GRASP, local search phase is typically important for achieving good performance.
- Can be based on Iterative Improvement or more advanced LS method (the latter often results in better performance).
- Tradeoff between computation time used in construction phase *vs* local search phase (typically optimized empirically, depends on problem domain).

Weight updating mechanism:

- Typical mechanism: increase weights of all solution components contained in candidate solution obtained from local search.
- Can also use aspects of search history;
  e.g., *current candidate solution* can be used as basis for weight update for additional intensification.

Example: A simple AICS algorithm for the TSP (1/2)

[Based on Ant System for the TSP, Dorigo et al. 1991]

- Search space and solution set as usual (all Hamiltonian cycles in given graph G). However represented in a construction tree T.
- Associate weight τ_{ij} with each edge (i, j) in G and T
- Use heuristic values $\eta_{ij} := 1/w_{ij}$.
- Initialize all weights to a small value τ_0 (parameter).
- *Constructive search* start with randomly chosen vertex and iteratively extend partial round trip ϕ by selecting vertex not contained in ϕ with probability

$$\frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N'(i)} [\tau_{il}]^\alpha \cdot [\eta_{ij}]^\beta}$$

Example: A simple AICS algorithm for the TSP (2/2)

- *Subsidiary local search* = typical iterative improvement
- *Weight update* according to

  $$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \Delta(ij, s')$$

  where $\Delta(i, j, s') := 1/f(s')$, if edge ij is contained in the cycle represented by s', and 0 otherwise.
- Criterion for weight increase is based on intuition that edges contained in short round trips should be preferably used in subsequent constructions.
- Decay mechanism (controlled by parameter ρ) helps to avoid unlimited growth of weights and lets algorithm forget past experience reflected in weights.
- (Just add a population of cand. solutions and you have an Ant Colony Optimization Algorithm!)

Multilevel Refinement

Key idea: make the problem recursively less refined creating a hierarchy of approximations of the original problem.

- an initial solution is found on the original problem or at a refined level
- solutions are iteratively refined at each level
- use of projection operators to transfer the solution from one level to another

Multilevel Refinement
while Termination criterion is not satisfied **do**
  coarse the problem π_0 into π_i, $i = 0, \ldots, k$ successive non degenerate problems
  $i = k$
  determine an initial candidate solution for π_k
  repeat
    $i = i - 1$
    extend the solution found in π_{i+1} to π_i
    use subsidiary local search to refine the solution on π_i
  until $i \geq 0$;

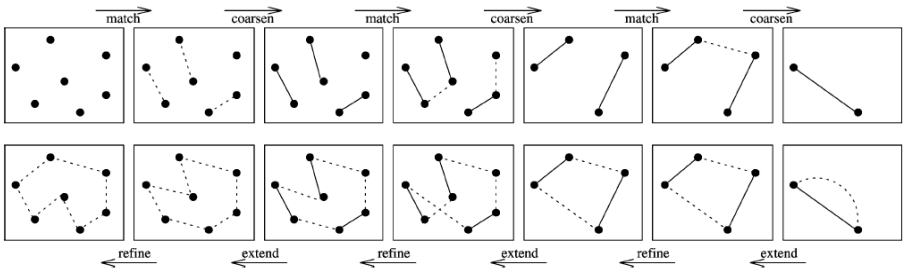
Example: Multilevel Refinement for TSP

Coarsen: fix some edges and contract vertices

Solve: matching
(always match vertices with the nearest unmatched neighbors)

Extend: uncontract vertices

Refine: LK heuristic



Note

- crucial point: the solution to each refined problem must contain a solution of the original problem (even if it is a poor solution)

Applications to

- Graph Partitioning
- Traveling Salesman
- Graph Coloring

Outline

1. Metaheuristics
   - Rollout/Pilot Method
   - Beam Search
   - Iterated Greedy
   - GRASP
   - Adaptive Iterated Construction Search
   - Multilevel Refinement
2. Work Environment
   - Organization
3. Bin Packing

Building a Work Environment

You will need these files for your project:

- The code that implements the algorithm (likely, 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?

Suggested organization

```
root\
 |   src\
 |   data\
 |   results\
 |   log\
 |   bin\
 |-  README
```


Example

Input controls on command line

```
comet queens.co -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`
  Comet: see example provided `loadDIMACS.co`

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

- use identifying labels in naming file outputs
  Example:
  `c0010.i0002.t0001.s02010.log`

Example

- You will need Multiple runs, multiple instances and multiple algorithms. Arrange this outside of your program: ➨ unix scripts (eg, bash one line program, perl, php)
- Parse outputfiles:
  Example

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

  See `http://www.gnu.org/software/coreutils/manual/` for shell tools.
- 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
...
```

Graphics

Visualization helps understanding

- Problem visualization (graphviz, igraph)
- Algorithm animation: (comet visualize)
- Results visualization: recommended R (more on this later)

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

Outline

1. Metaheuristics
   - Rollout/Pilot Method
   - Beam Search
   - Iterated Greedy
   - GRASP
   - Adaptive Iterated Construction Search
   - Multilevel Refinement

2. Work Environment
   - Organization

3. Bin Packing

Knapsack, Bin Packing, Cutting Stock

Knapsack

Given: a knapsack with maximum weight W and a set of n items $\{1, 2, \ldots, n\}$, with each item j associated to a profit p_j and to a weight w_j.
Task: Find the subset of items of maximal total profit and whose total weight is not greater than W.

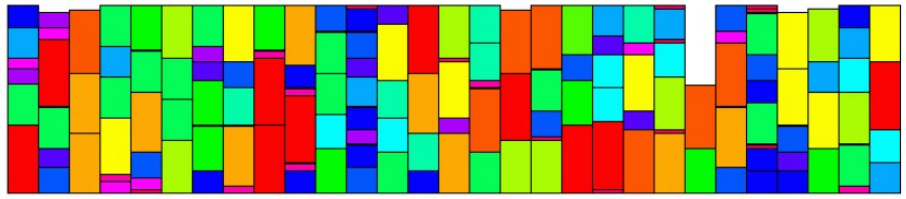
One dimensional bin packing

Given: A set $L = (a_1, a_2, \ldots, a_n)$ of *items*, each with a size $s(a_i) \in (0, 1]$ and an unlimited number of unit-capacity bins $B_1, B_2, \ldots, B_m$.
Task: Pack all the items into a minimum number of unit-capacity bins $B_1, B_2, \ldots, B_m$.

Cutting stock

Given: ... each item (paper roll) has a profit $p_j > 0$ and a number of times it must appear q_i.
Task: determine the patterns of items to be packed (cut) in a single finite bin (eg, paper strip) that minimizes the total waste.

Bin Packing



Cutting Stock