DM841

Heuristics for Combinatorial Optimization

Metaheuristics to Enhance Construction Heuristics

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

Outline

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
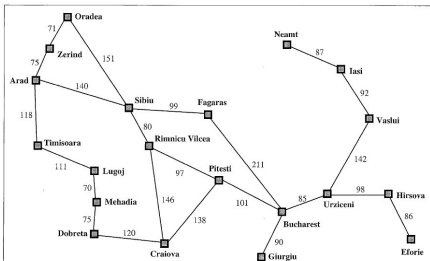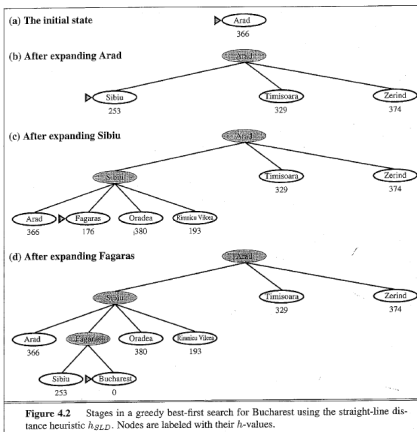
Best-first search

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.

A* search

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: $h(n) \leq c(n, a, n') + h(n')$

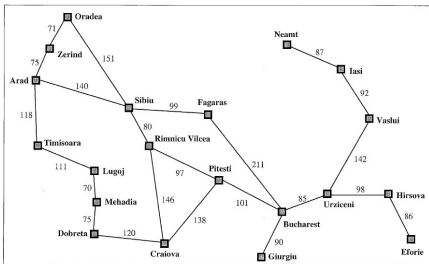# A* search

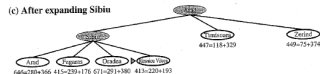**Figure 3.2** A simplified road map of part of Romania.
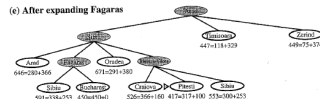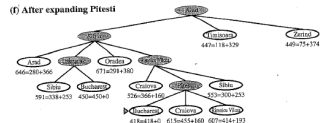
Figure 4.3 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

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)

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?]

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).

Outline

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

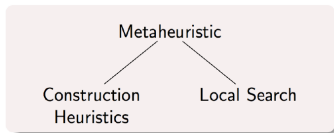Greedy algorithms (derived from best-first)

- ▶ 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

Metaheuristics

On backtracking framework
(beyond best-first search)

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

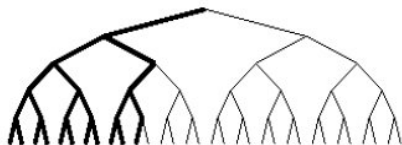Outside the exact framework
(beyond greedy search)

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

Outline

Bounded backtrack

Bounded-backtrack search:

bbs(10)

Depth-bounded, then bounded-backtrack search:

dbs(2, bbs(0))

Outline

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

Outline

Randomization in Tree Search

The idea comes from complete search: the important decisions are made up in the search tree (backdoors - set of variables such that once they are instantiated the remaining problem simplifies to a tractable form)
$\rightsquigarrow$ 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: $S_u = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 4, 8, 1, \ldots)$

35

Outline

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

Outline

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

Outline

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**

$\quad$ $r := s$
$\quad$ (randomly or heuristically) destruct part of s
$\quad$ greedily reconstruct the missing part of s
$\quad$ based on acceptance criterion,
$\quad\quad$ keep s or revert to $s := r$

Outline

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

Outline

Guidelines for Text Writing
From common bad practice in this course

- ▶ 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.

- Algorithmic sketches in pseudo-code must be code independent

- Complexity analysis is relevant: it helps to understand the algorithm and gives idea about how things can be implemented efficiently

- Aim at beauty, eg, general approaches rather than problem dependent.

- Reason on the problem, do not do things mechanically, every problem is a different story.

- Originality counts

- Language, choose the one you prefer

- Avoid self-pietism: Do not write "I did not have time to..."

- Focus on efficency, aim at the Pareto frontier.

- See also Comment List and examples of past final projects from course web page