DM811

Heuristics for Combinatorial Optimization

Lecture 2
Heuristics: basic ideas

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Summary

1. Course Introduction
2. Combinatorial Optimization
   - Combinatorial Problems, Terminology
   - Solution Methods, Overview
   - Travelling Salesman Problem
3. Problem Solving
   - Example: Graph Coloring Problem
   - Polya's view about Problem Solving
4. Basic Concepts from Algorithmics
   (Review slides and Cormen, Leiserson, Rivest and Stein. *Introduction to algorithms*. 2001)

Outline

1. Modelling and Search
   - IP-models
   - CP-models
   - Modeling for Heuristics
   - Search

2. Search Paradigms
   - Construction Heuristics
   - Local Search

Outline

1. Modelling and Search
   - IP-models
   - CP-models
   - Modeling for Heuristics
   - Search

2. Search Paradigms
   - Construction Heuristics
   - Local Search

solution algorithm = model + search

Mathematical Programming Models

- How to model an optimization problem
  - choose some decision variables
    they typically encode the result we are interested into
  - express the problem constraints in terms of these variables
    they specify what the solutions to the problem are
  - express the objective function
    the objective function specifies the quality of each solution
- The result is an optimization model
  - It is a declarative formulation
    specify the "what", not the "how"
  - There may be many ways to model an optimization problem

IP-models

Standard IP formulation: Let x_{vk} be a 0–1 variable equal to 1 whenever the vertex v takes the color k
and y_k be 1 if color k is used and 0 otherwise

$$
\begin{aligned}
\min \quad & \sum_{k \in K} y_k & \\
\text{s.t.} \quad & \sum_{k \in K} x_{vk} = 1, & \forall v \in V, \\
& x_{vk} + x_{uk} \leq y_k, & \forall (u, v) \in E(G), \forall k \in K, \\
& x_{vk} \in \{0, 1\}, & \forall v \in V, \forall k \in K \\
& y_k \in \{0, 1\}, & \forall k \in K.
\end{aligned}
$$

Column generation formulation

- Notation
  - Independent set s, with cardinality c_s
  - $\mathcal{S}$: Collection of every maximal independent set of G
  - $\mathcal{S}_v$: subset of S that contains v
  - λ_s: 0-1 variable equal to 1 if independent set s is used

$$
\begin{aligned}
\min \quad & \sum_{s \in \mathcal{S}} \lambda_s & \\
\text{s.t.} \quad & \sum_{s \in \mathcal{S}_v} \lambda_s \geq 1, & \forall v \in V, \\
& \lambda_s \in \{0, 1\}, & \forall s \in \mathcal{S}.
\end{aligned}
$$

Constraint Programming

The **domain** of a variable x, denoted $D(x)$, is a finite set of elements that can be assigned to x.

A **constraint** C on X is a subset of the Cartesian product of the domains of the variables in X, i.e., $C \subseteq D(x_1) \times \cdots \times D(x_k)$ (extensional form). A tuple $(d_1, \ldots, d_k) \in C$ is called a solution to C.
Equivalently, we say that a solution $(d_1, ..., d_k) \in C$ is an assignment of the value d_i to the variable $x_i, \forall 1 \leq i \leq k$, and that this assignment satisfies C (intentional form). If $C = \emptyset$, we say that it is inconsistent.

Constraint Programming

Constraint Satisfaction Problem (CSP)

A CSP is a finite set of variables X, together with a finite set of constraints C, each on a subset of X. A **solution** to a CSP is an assignment of a value $d \in D(x)$ to each $x \in X$, such that all constraints are satisfied simultaneously.

Constraint Optimization Problem (COP)

A COP is a CSP P defined on the variables $x_1, \ldots, x_n$, together with an objective function $f : D(x_1) \times \cdots \times D(x_n) \to Q$ that assigns a value to each assignment of values to the variables. An **optimal solution** to a minimization (maximization) COP is a solution d to P that minimizes (maximizes) the value of $f(d)$.

CP-model

CP formulation:

$$
\begin{array}{rll}
\textit{variables}: & \mathtt{domain}(\mathtt{y_i}) = \{1, \ldots, K\} & \forall i \in V \\
\textit{constraints}: & y_i \neq y_j & \forall ij \in E(G) \\
 & \mathtt{alldifferent}(\{\mathtt{y_i} \mid \mathtt{i} \in \mathtt{C}\}) & \forall C \in \mathcal{C}
\end{array}
$$

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.

Constraint based Modelling

Can be done within the same framework of Constraint Programming.
See Constraint Based Local-Search (Hentenryck and Michel) [B4].

- Decide the variables.
  An assignment of these variables should identify a candidate solution or a candidate solution must be retrievable efficiently
  Must be linked to some Abstract Data Type (arrays, sets, permutations).

- Express the constraints on these variables

No restrictions are posed on the language in which the above two elements are expressed.

Search

- Backtracking (complete)
- Branch and Bound (complete)
- Local search (incomplete)

Example: Knapsack problem

Knapsack problem

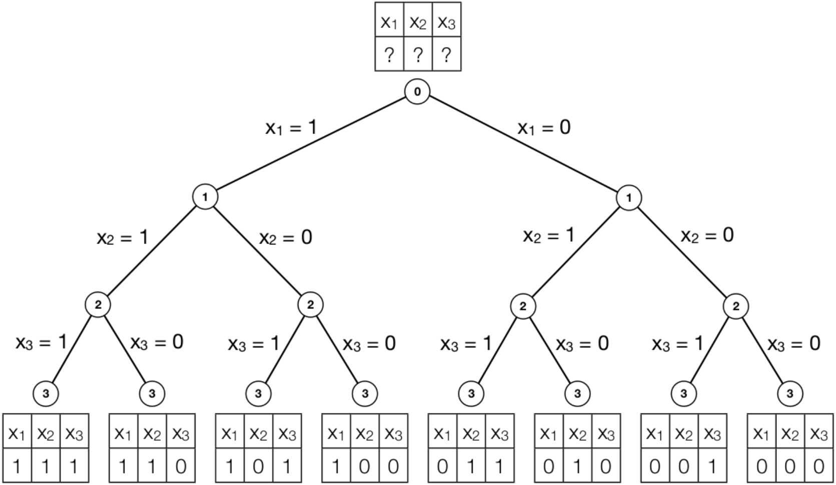
Given: a set of items I, each item $i \in I$ characterized by

- its weight w_i
- its value v_i
- and a capacity K for a knapsack

Task: find the subset of items in I

- does not exceed the capacity K of the knapsack
- that has maximum value

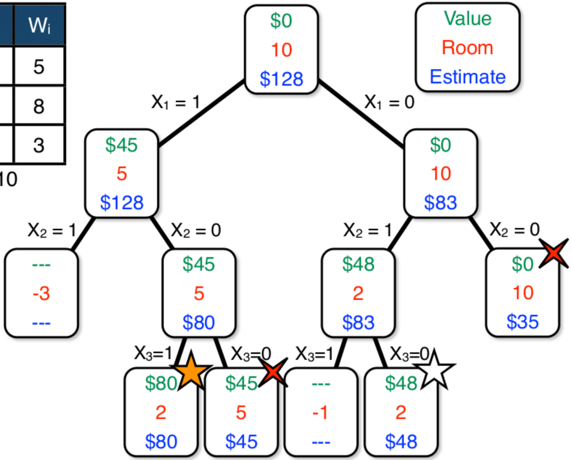
Branch and Bound



Relaxing integrality

i	V_i	W_i
1	45	5
2	48	8
3	35	3

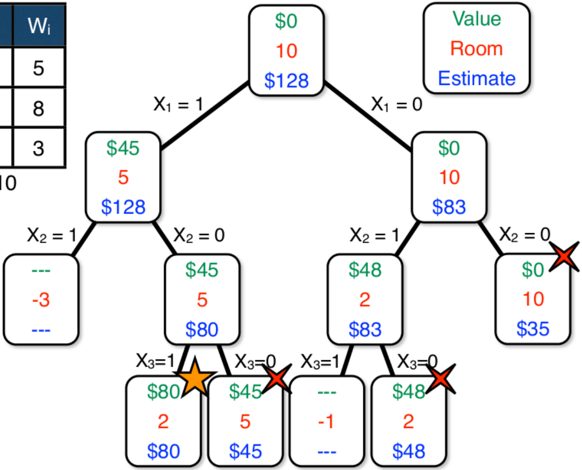
K = 10



Relaxing capacity constraint

i	V_i	W_i
1	45	5
2	48	8
3	35	3

K = 10



Dynamic Programming

Notation:

- assume that $I = 1, 2, ..., n$
- $O(k, j)$ denotes the optimal solution to the knapsack problem with capacity k and items $[1..j]$

We are interested in finding out the best value $O(K, n)$

Recurrence relation

- Assume that we know how to solve

$$O(k, j-1) \text{ for all } k \in 0..K$$

- We want to solve $O(k, j)$: We are just considering one more item, i.e., item j.
- If $w_j \leq k$, there are two cases
  - Either we do not select item j, then the best solution we can obtain is $O(k, j-1)$
  - Or we select item j and the best solution is $v_j + O(k - w_j, j-1)$
- In summary

$$O(k, j) = \begin{cases} max\{O(k, j-1), vj + O(k - wj, j-1)\} & \text{if} w_j \leq k \\ O(k, j-1) & \text{otherwise} \end{cases}$$

- Initial conditions:

$$O(k, 0) = 0 \text{ for all } k$$

Compute the recurrence relation bottom up

```
int O(int k,int j) {
  if (j == 0)
    return 0;
  else if (wj <= k)
    return max(O(k,j-1),vj + O(k-wj,j-1));
  else
    return O(k,j-1)
}
```

How efficient is this approach?

Outlook

To come:

- Construction Heuristics
- High level description of Local Search
- Solver Systems
- Setting up the Working Environment

Outline

1. Modelling and Search
   - IP-models
   - CP-models
   - Modeling for Heuristics
   - Search
2. Search Paradigms
   - Construction Heuristics
   - Local Search

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 candidate 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 infeasibility risk)
    (The degree of a variable is defined as the number of constraints it is involved in)
  - *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

  - 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 to the other variables
  - Select value that leaves the smallest number of feasible values to the other variables (fail early)

Example: Knapsack

Greedy best-first search

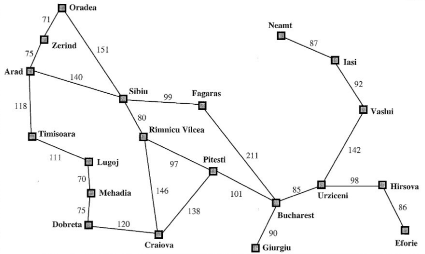


Figure 3.2 A simplified road map of part of Romania.

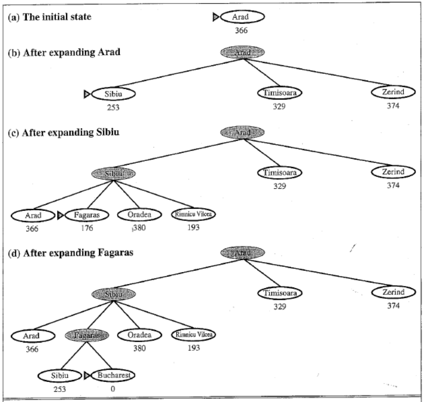


Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

- 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:

- solution representation $\rightsquigarrow$ search space
- initial solution
- neighborhood relation (determines the move operator)
- evaluation function

Course Overview

✔ Combinatorial Optimization, Methods and Models

1. CH and LS: overview
2. Working Environment and Solver System
3. Methods for the Analysis of Experimental Results
4. Construction Heuristics
5. Local Search: Components, Basic Algorithms
6. Local Search: Neighborhoods and Search Landscape
7. Efficient Local Search: Incremental Updates and Neighborhood Pruning
8. Stochastic Local Search & Metaheuristics
9. Configuration Tools: F-race
10. Very Large Scale Neighborhoods

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