DM841

Discrete Optimization

Lecture 3
Local Search and Metaheuristics Overview

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Last Time

1. Combinatorial Optimization and Terminology
2. Solution Methods
3. SAT Example: enumeration, MIP, local search, backtracking

Outline

1. Solution Methods & Examples
   - Knapsack
   - Enumeration, Branch & Bound
   - Dynamic Programming
   - Vertex Coloring
   - Constraint Programming

2. Heuristic Methods
   - Local Search

Outline

1. Solution Methods & Examples
   - Knapsack
   - Enumeration, Branch & Bound
   - Dynamic Programming
   - Vertex Coloring
   - Constraint Programming

2. Heuristic Methods
   - Local Search

Outline

1. Solution Methods & Examples
   - Knapsack
   - Enumeration, Branch & Bound
   - Dynamic Programming
   - Vertex Coloring
   - Constraint Programming

2. Heuristic Methods
   - Local Search

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

IP Model

Let x_i be a binary variable that denotes whether we include or not the item i

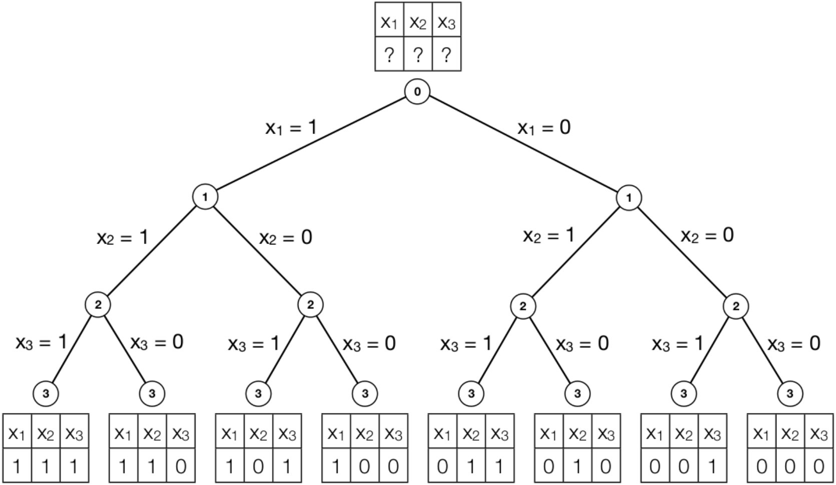
$$
\begin{aligned}
\max \quad & \sum_{i \in I} v_i x_i & \\
\text{s.t.} \quad & \sum_{i \in I} w_i x_i \leq K & \forall c \in C, \\
& x_i \in \{0, 1\}, & \forall i \in I
\end{aligned}
$$

Outline

1. Solution Methods & Examples
    - Knapsack
    - Enumeration, Branch & Bound
    - Dynamic Programming
    - Vertex Coloring
    - Constraint Programming

2. Heuristic Methods
    - Local Search

Enumeration



Branch and Bound

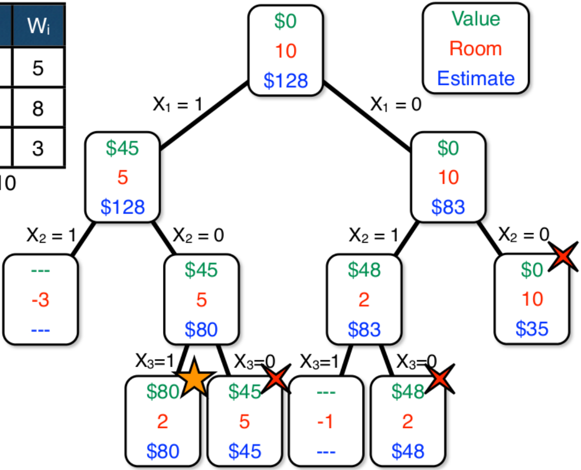
- Iterative two steps
  - branching
  - bounding
- Branching
  - split the problem into a number of subproblems
  - like in exhaustive search
- Bounding
  - find an optimistic estimate of the best solution to the subproblem
    maximization: upper bound
    minimization: lower bound

Branch and Bound

Optimistic estimate: Relaxing capacity constraint

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

K = 10

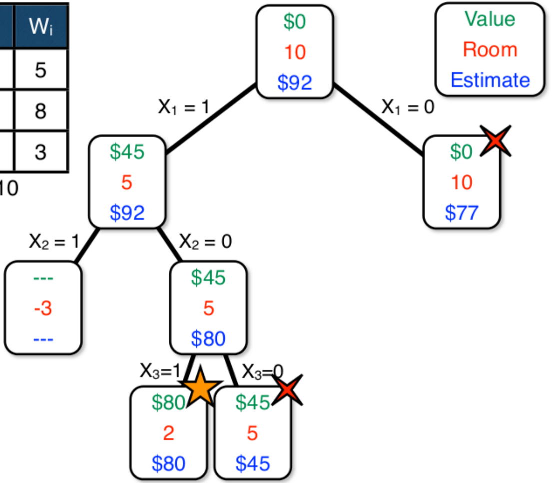


Branch and Bound

Optimistic estimation: Relaxing integrality

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

K = 10



Outline

1. Solution Methods & Examples
   - Knapsack
   - Enumeration, Branch & Bound
   - Dynamic Programming
   - Vertex Coloring
   - Constraint Programming
2. Heuristic Methods
   - Local Search

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

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

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.

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

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 $O(k, j-1)$

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 $v_j + O(k - w_j, j-1)$

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

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

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?

Outline

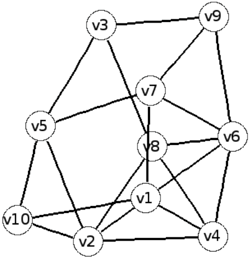
1. Solution Methods & Examples
   - Knapsack
   - Enumeration, Branch & Bound
   - Dynamic Programming
   - Vertex Coloring
   - Constraint Programming

2. Heuristic Methods
   - Local Search

The Vertex Coloring Problem

Given: A graph G and a set of colors Γ.

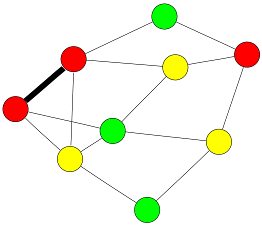
A proper coloring is an assignment of one color to each vertex of the graph such that adjacent vertices receive different colors.



The Vertex Coloring Problem

Given: A graph G and a set of colors Γ.

A proper coloring is an assignment of one color to each vertex of the graph such that adjacent vertices receive different colors.

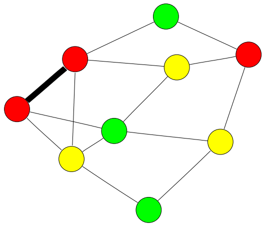


The Vertex Coloring Problem

Given: A graph G and a set of colors Γ.

A proper coloring is an assignment of one color to each vertex of the graph such that adjacent vertices receive different colors.

Decision version (k-coloring)
Task: Find a proper coloring of G that uses at most k colors.
Optimization version (chromatic number)
Task: Find a proper coloring of G that uses the minimal number of colors.

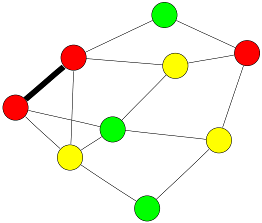


The Vertex Coloring Problem

Given: A graph G and a set of colors Γ.

A proper coloring is an assignment of one color to each vertex of the graph such that adjacent vertices receive different colors.

Decision version (k-coloring)
Task: Find a proper coloring of G that uses at most k colors.
Optimization version (chromatic number)
Task: Find a proper coloring of G that uses the minimal number of colors.



Design an algorithm for solving general instances of the graph coloring problem.

Exercise

Map coloring:

Outline

1. Solution Methods & Examples
   - Knapsack
   - Enumeration, Branch & Bound
   - Dynamic Programming
   - Vertex Coloring
   - Constraint Programming

2. Heuristic Methods
   - Local Search

Constraint Programming

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

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.

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).

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 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 $f : D(x_1) \times \cdots \times D(x_n) \rightarrow Q$ 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:

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

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.

Search

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

Outline

1. Solution Methods & Examples
    - Knapsack
    - Enumeration, Branch & Bound
    - Dynamic Programming
    - Vertex Coloring
    - Constraint Programming

2. Heuristic Methods
    - Local Search

Outline

1. Solution Methods & Examples
    - Knapsack
    - Enumeration, Branch & Bound
    - Dynamic Programming
    - Vertex Coloring
    - Constraint Programming

2. Heuristic Methods
    - Local Search

Local Search

Main idea for combinatorial optimization

- Sequential modification of a small number of decisions
- Incremental evaluation of solutions, generally in O(1) time
  - Lazy propagation of constraints
  - Usage of invariants

$\leadsto$ Small improvement probability but small time and space complexity
$\leadsto$ Millions of moves per minute

- (Meta)heuristic rules to drive the search

Metaheuristics

- Variable Neighborhood Search and Large Scale Neighborhood Search
  diversified neighborhoods + incremental algorithmics ("diversified" $\equiv$ multiple, variable-size, and rich).

- Tabu Search: Online learning of moves
  Discard undoing moves,
  Discard inefficient moves
  Improve efficient moves selection

- Simulated annealing
  Allow degrading solutions

- "Restart" + parallel search
  Avoid local optima
  Improve search space coverage

Local Search Modeling

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.

Local Search

Given a (combinatorial) optimization problem Π and one of its instances π:

- search space $S(\pi)$
  specified by candidate solution representation:
  discrete structures: sequences, permutations, graphs, partitions
  (*e.g.*, for SAT: array, sequence of all truth assignments
  to propositional variables)

  Note: solution set $S'(\pi) \subseteq S(\pi)$
  (*e.g.*, for SAT: models of given formula)

Local Search

Given a (combinatorial) optimization problem Π and one of its instances π:

- search space $S(\pi)$
  specified by candidate solution representation:
  discrete structures: sequences, permutations, graphs, partitions
  (*e.g.*, for SAT: array, sequence of all truth assignments
  to propositional variables)

  Note: solution set $S'(\pi) \subseteq S(\pi)$
  (*e.g.*, for SAT: models of given formula)

- evaluation function $f_\pi : S(\pi) \to \mathbf{R}$
  (*e.g.*, for SAT: number of false clauses)

Local Search

Given a (combinatorial) optimization problem Π and one of its instances π:

- search space $S(\pi)$
  specified by candidate solution representation:
  discrete structures: sequences, permutations, graphs, partitions
  (*e.g.*, for SAT: array, sequence of all truth assignments
  to propositional variables)

  Note: solution set $S'(\pi) \subseteq S(\pi)$
  (*e.g.*, for SAT: models of given formula)

- evaluation function $f_\pi : S(\pi) \to \mathbf{R}$
  (*e.g.*, for SAT: number of false clauses)

- neighborhood function, $\mathcal{N}_\pi : S \to 2^{S(\pi)}$
  (*e.g.*, for SAT: neighboring variable assignments differ
  in the truth value of exactly one variable)

Local Search Algorithm

Further components [according to [HS]]

- set of memory states $M(\pi)$
  (may consist of a single state, for LS algorithms that do not use memory)

Local Search Algorithm

Further components [according to [HS]]

- set of memory states $M(\pi)$
  (may consist of a single state, for LS algorithms that do not use memory)
- initialization function $\mathtt{init} : \emptyset \to S(\pi)$
  (can be seen as a probability distribution $\Pr(S(\pi) \times M(\pi))$ over initial search positions and memory states)

Local Search Algorithm

Further components [according to [HS]]

- set of memory states $M(\pi)$
  (may consist of a single state, for LS algorithms that do not use memory)
- initialization function $\mathtt{init} : \emptyset \to S(\pi)$
  (can be seen as a probability distribution $\Pr(S(\pi) \times M(\pi))$ over initial search positions and memory states)
- step function $\mathtt{step} : S(\pi) \times M(\pi) \to S(\pi) \times M(\pi)$
  (can be seen as a probability distribution $\Pr(S(\pi) \times M(\pi))$ over subsequent, neighboring search positions and memory states)

Local Search Algorithm

Further components [according to [HS]]

- set of memory states $M(\pi)$
  (may consist of a single state, for LS algorithms that do not use memory)
- initialization function $\texttt{init} : \emptyset \to S(\pi)$
  (can be seen as a probability distribution $\Pr(S(\pi) \times M(\pi))$ over initial search positions and memory states)
- step function $\texttt{step} : S(\pi) \times M(\pi) \to S(\pi) \times M(\pi)$
  (can be seen as a probability distribution $\Pr(S(\pi) \times M(\pi))$ over subsequent, neighboring search positions and memory states)
- termination predicate $\texttt{terminate} : S(\pi) \times M(\pi) \to \{\top, \bot\}$
  (determines the termination state for each search position and memory state)

Example: Local Search for SAT

Example: Uninformed random walk for SAT (1)

- **search space S:** set of all truth assignments to variables in given formula F
  (**solution set S':** set of all models of F)

Example: Local Search for SAT

Example: Uninformed random walk for SAT (1)

- **search space S:** set of all truth assignments to variables in given formula F
  (**solution set S':** set of all models of F)
- **neighborhood relation $\mathcal{N}$:** *1-flip neighborhood*, *i.e.*, assignments are neighbors under $\mathcal{N}$ iff they differ in the truth value of exactly one variable
- **evaluation function** not used, or $f(s) = 0$ if model $f(s) = 1$ otherwise

Example: Local Search for SAT

Example: Uninformed random walk for SAT (1)

- **search space S:** set of all truth assignments to variables in given formula F (**solution set S':** set of all models of F)
- **neighborhood relation $\mathcal{N}$:** *1-flip neighborhood*, *i.e.*, assignments are neighbors under $\mathcal{N}$ iff they differ in the truth value of exactly one variable
- **evaluation function** not used, or $f(s) = 0$ if model $f(s) = 1$ otherwise
- **memory:** not used, *i.e.*, $M := \{0\}$

Example: Uninformed random walk for SAT (2)

- **initialization:** uniform random choice from S, *i.e.*, $\mathtt{init}(,\{a', m\}) := 1/|S|$ for all assignments a' and memory states m

Example: Uninformed random walk for SAT (2)

- **initialization:** uniform random choice from S, *i.e.*, $\mathtt{init}(, \{a', m\}) := 1/|S|$ for all assignments a' and memory states m
- **step function:** uniform random choice from current neighborhood, *i.e.*, $\mathtt{step}(\{a, m\}, \{a', m\}) := 1/|N(a)|$ for all assignments a and memory states m, where $N(a) := \{a' \in S \mid \mathcal{N}(a, a')\}$ is the set of all neighbors of a.

Example: Uninformed random walk for SAT (2)

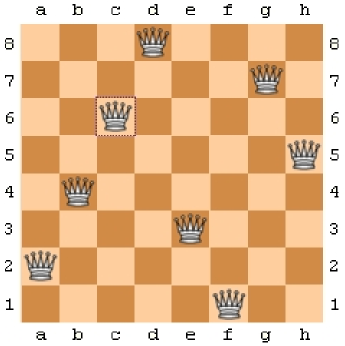
- **initialization:** uniform random choice from S, *i.e.*, $\mathtt{init}(,\{a', m\}) := 1/|S|$ for all assignments a' and memory states m
- **step function:** uniform random choice from current neighborhood, *i.e.*, $\mathtt{step}(\{a, m\}, \{a', m\}) := 1/|N(a)|$
  for all assignments a and memory states m,
  where $N(a) := \{a' \in S \mid \mathcal{N}(a, a')\}$ is the set of all neighbors of a.
- **termination:** when model is found, *i.e.*, $\mathtt{terminate}(\{a, m\}, \{\top\}) := 1$ if a is a model of F, and 0 otherwise.

N-Queens Problem

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.



queensLS0a.co

```
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) {
    queen[q] := v;
    cout<<"chng @ "<<it<<": queen["<<q<<"]:="<<v<<" viol: "<<S.violations() <<endl;
  }
  it = it + 1;
}
cout << queen << endl;
```

Local Search Modeling
Another Random Walk

queensLS1.co

```
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 : S.violations(queen[q])>0, v in Size) {
    queen[q] := v;
    cout<<"chng @ "<<it<<": queen["<<q<<"]:="<<v<<" viol: "<<S.violations()<<endl;
  }
  it = it + 1;
}
cout << queen << endl;
```