Branch and Bound Algorithm and Heuristics for NP-Hard Optimization Problems

An Introduction

Martin Zachariasen

Department of Computer Science
University of Copenhagen

May 16, 2013

Some slides based on original material by:

Jesper Larsen and Jens Clausen

Outline

Outline

Outline

Outline

Outline

0-1 Knapsack Problem (KP)

Given n objects and a knapsack with capacity c. Object $i = 1, \ldots, n$ has weight w_i and returns a profit p_i if packed into the knapsack.

Choose a subset of objects such that

- the total weight of the chosen objects does not exceed the capacity of the knapsack
- the total profit is maximized

0-1 Knapsack Problem (KP)

Given n objects and a knapsack with capacity c. Object $i = 1, \ldots, n$ has weight w_i and returns a profit p_i if packed into the knapsack.

Choose a subset of objects such that

- the total weight of the chosen objects does not exceed the capacity of the knapsack
- the total profit is maximized

Integer programming formulation:

$$\begin{aligned} \max \quad & \sum_{i=1}^{n} p_i x_i \\ s.t. \quad & \sum_{i=1}^{n} w_i x_i \leq c \\ & x_i \in \{0,1\}, \ i = 1, \ldots, n \end{aligned}$$

Travelling Salesman Problem (TSP)

Given a set $\{c_1, c_2, \ldots, c_n\}$ of cities and distances $d(c_i, c_j)$ between every pair c_i, c_j of cities, find a shortest tour through all the cities, i.e., a permutation π such that

$$\sum_{i=1}^{n-1} d(c_{\pi(i)}, c_{\pi(i+1)}) \; + \; d(c_{\pi(n)}, c_{\pi(1)})$$

is minimized.

Equivalently:

Given an edge-weighted graph $G = (V, E)$, find an Hamiltonian cycle with minimum total weight in G.

TSP: Variants

General TSP: No restrictions on the distance function.

Symmetric TSP: All distances are symmetric: $d(c_i, c_j) = d(c_j, c_i),\ \forall c_i, c_j$.

Metric TSP: All distances are symmetric and fulfill the triangle inequality: $d(c_i, c_j) \leq d(c_i, c_k) + d(c_k, c_j),\ \forall c_i, c_j, c_k$.

Euclidean TSP: The cities are points in an Euclidean space and the distances are the Euclidean distances between the points.

In this talk we consider the symmetric TSP.

TSP: Integer Programming Formulation

x_{ij}: binary decision variable (1 if edge is chosen, 0 otherwise)

TSP: Integer Programming Formulation

x_{ij}: binary decision variable (1 if edge is chosen, 0 otherwise)

$$\min \quad \sum_{(i,j)\in E} d(c_i, c_j) x_{ij}$$

minimize tour length

TSP: Integer Programming Formulation

x_{ij}: binary decision variable (1 if edge is chosen, 0 otherwise)

$$\begin{array}{lll} \min & \sum_{(i,j)\in E} d(c_i, c_j)x_{ij} & \text{minimize tour length} \\ \text{s.t.} & \sum_j x_{ij} = 2 \qquad i \in \{1, 2, \ldots, n\} & \text{two edges per node} \end{array}$$

TSP: Integer Programming Formulation

x_{ij}: binary decision variable (1 if edge is chosen, 0 otherwise)

$$
\begin{array}{lll}
\text{min} & \sum_{(i,j)\in E} d(c_i, c_j) x_{ij} & \text{minimize tour length} \\
\text{s.t.} & \sum_j x_{ij} = 2 \qquad i \in \{1, 2, \ldots, n\} & \text{two edges per node} \\
 & \sum_{i,j\in Z} x_{ij} \leq |Z| - 1 \qquad \emptyset \subset Z \subset V & \text{no subtours}
\end{array}
$$

TSP: Integer Programming Formulation

x_{ij}: binary decision variable (1 if edge is chosen, 0 otherwise)

min	$\sum_{(i,j)\in E} d(c_i, c_j)x_{ij}$		minimize tour length
s.t.	$\sum_j x_{ij} = 2$	$i \in \{1, 2, \ldots, n\}$	two edges per node
	$\sum_{i,j\in Z} x_{ij} \leq \lvert Z\rvert - 1$	$\emptyset \subset Z \subset V$	no subtours
	$x_{ij} \in \{0, 1\}$	$(i, j) \in E$	binary decision variables

Outline

Combinatorial Optimization

Combinatorial optimization problems typically involve finding a selection, grouping, ordering, or assignment of a discrete, finite set of objects.

In addition, we have an objective function as well as logical conditions that solutions must satisfy.

Combinatorial Optimization

Combinatorial optimization problems typically involve finding a selection, grouping, ordering, or assignment of a discrete, finite set of objects.

In addition, we have an objective function as well as logical conditions that solutions must satisfy.

General definition:

- Solution space $\mathcal{S}$ and objective function $f : \mathcal{S} \longmapsto \mathbb{R}$.
- Find an optimal solution, that is, a solution $\mathbf{s}^* \in \mathcal{S}$ with minimum objective function value:

$$\forall \mathbf{s} \in \mathcal{S} \quad : \quad f(\mathbf{s}^*) \leq f(\mathbf{s})$$

Search in Combinatorial Optimization

Search: iteratively generate and evaluate (candidate) solutions from $\mathcal{S}$.

Note: evaluating candidate solutions is typically computationally much cheaper than finding (optimal) solutions.

Search in Combinatorial Optimization

Search: iteratively generate and evaluate (candidate) solutions from $\mathcal{S}$.

Note: evaluating candidate solutions is typically computationally much cheaper than finding (optimal) solutions.

Most combinatorial optimization problems are $\mathcal{NP}$-hard, and for these problems complete search/enumeration of $\mathcal{S}$ is (in principle) necessary.

Complete enumeration is very time-consuming if the solution space $\mathcal{S}$ is large.

Outline

Exact and Heuristic Search Algorithms

Exact algorithm: Systematically search through the whole search space

Heuristic algorithm: Only visit (hopefully) promising parts of the search space

Exact and Heuristic Search Algorithms

Exact algorithm: Systematically search through the whole search space

Heuristic algorithm: Only visit (hopefully) promising parts of the search space

Exact algorithm (systematic search) is often better suited when ...

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

Heuristic algorithm (incomplete search) is often better suited when ...

- reasonably good solutions are required within a short time,
- parallel processing is used,
- problem-specific knowledge is rather limited.

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

1. Solution quality: How good are the solutions obtained? (Only really relevant for heuristic algorithms.)

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

1. Solution quality: How good are the solutions obtained? (Only really relevant for heuristic algorithms.)
2. Running time: What is the running time of the algorithm as a function of problem instance size?

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

1. Solution quality: How good are the solutions obtained? (Only really relevant for heuristic algorithms.)
2. Running time: What is the running time of the algorithm as a function of problem instance size?

Three possible approaches:

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

1. Solution quality: How good are the solutions obtained? (Only really relevant for heuristic algorithms.)
2. Running time: What is the running time of the algorithm as a function of problem instance size?

Three possible approaches:

1. Worst-case analysis: Give a bound on the worst possible performance (quality and/or running time) of the algorithm.

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

1. Solution quality: How good are the solutions obtained? (Only really relevant for heuristic algorithms.)
2. Running time: What is the running time of the algorithm as a function of problem instance size?

Three possible approaches:

1. Worst-case analysis: Give a bound on the worst possible performance (quality and/or running time) of the algorithm.
2. Average analysis: Assume that the instances (or their respective parameters) are distributed according to a probability function. Compute the average behaviour of the algorithm.

Evaluation and Comparison of Search Algorithms

Two interesting parameters:

1. Solution quality: How good are the solutions obtained? (Only really relevant for heuristic algorithms.)
2. Running time: What is the running time of the algorithm as a function of problem instance size?

Three possible approaches:

1. Worst-case analysis: Give a bound on the worst possible performance (quality and/or running time) of the algorithm.
2. Average analysis: Assume that the instances (or their respective parameters) are distributed according to a probability function. Compute the average behaviour of the algorithm.
3. Empirical/experimental analysis: Implement the algorithm and investigate how it performs on a suitable set of problem instances.

Outline

Branch and Bound

Main idea:

Perform an implicit enumeration of the search space $\mathcal{S}$, but avoid visiting (hopefully) large parts of $\mathcal{S}$ by proving non-optimality of solutions in these parts of $\mathcal{S}$.

Branch and Bound

Main idea:

Perform an implicit enumeration of the search space $\mathcal{S}$, but avoid visiting (hopefully) large parts of $\mathcal{S}$ by proving non-optimality of solutions in these parts of $\mathcal{S}$.

Core elements of branch and bound:

Branch and Bound

Main idea:

Perform an implicit enumeration of the search space $\mathcal{S}$, but avoid visiting (hopefully) large parts of $\mathcal{S}$ by proving non-optimality of solutions in these parts of $\mathcal{S}$.

Core elements of branch and bound:

- Branch: Divide the problem (search space) recursively into subproblems

Branch and Bound

Main idea:

Perform an implicit enumeration of the search space $\mathcal{S}$, but avoid visiting (hopefully) large parts of $\mathcal{S}$ by proving non-optimality of solutions in these parts of $\mathcal{S}$.

Core elements of branch and bound:

- Branch: Divide the problem (search space) recursively into subproblems
- Bound: Provide lower bounds on the quality of solutions for subproblems

Branch and Bound

Main idea:
Perform an implicit enumeration of the search space $\mathcal{S}$, but avoid visiting (hopefully) large parts of $\mathcal{S}$ by proving non-optimality of solutions in these parts of $\mathcal{S}$.

Core elements of branch and bound:

- Branch: Divide the problem (search space) recursively into subproblems
- Bound: Provide lower bounds on the quality of solutions for subproblems
- Select: Select a subproblem to process: bound and (if necessary) branch

Outline

TSP: Bounds

- One way to identify a bound for the TSP is by relaxing constraints. This could be to allow subtours. This bound is although known to be rather weak.
- An alternative is the **1-tree relaxation**.

TSP: The 1-Tree Bound

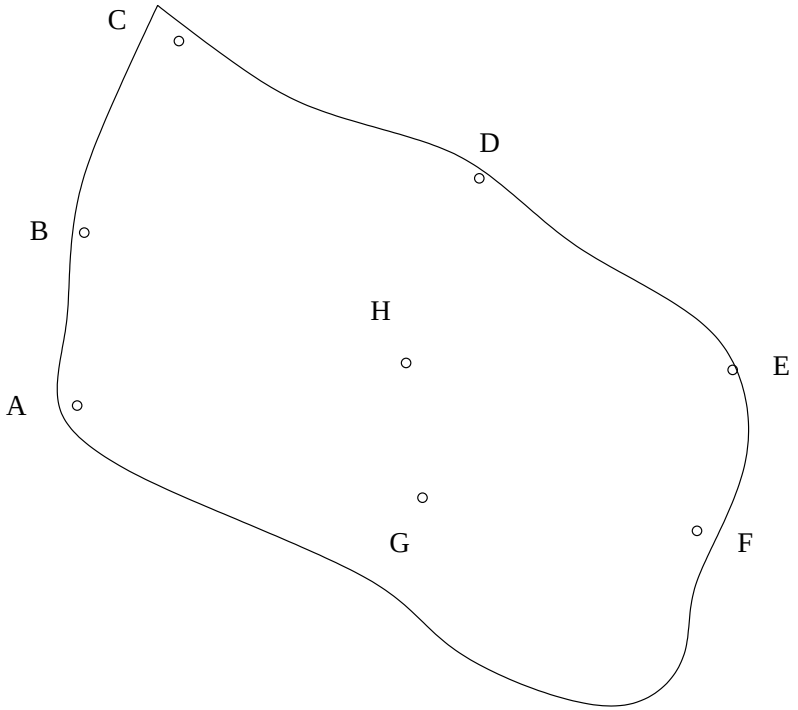
- Identify a special vertex **1** (this can be any vertex of the graph).
- **1** and all edges incident with **1** are removed from G.
- For the remaining graph determine a minimum spanning tree T.
- Now the two smallest edges e_1 and e_2 incident with **1** are added to T producing T_1 (called a **1-tree**)

TSP: Why is T_1 a Bound?

We need to convince ourselves that the total cost of T_1 is a lower bound of the value of an optimal tour.

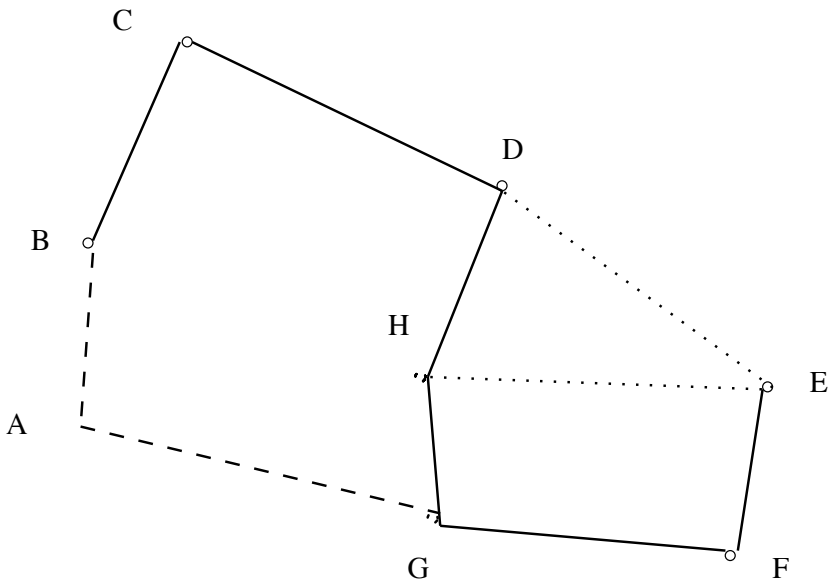
- Note that a Hamiltonian tour can be divided into two edges e_1' and e_2' that are incident with $\mathbf{1}$ and the rest of the tour (let us call it T').
- So the set of Hamiltonian tours is a subset of 1-trees of G.
- Since e_1, e_2 are the two smallest edges incident to $\mathbf{1}$ $d_{e_1} + d_{e_2} \leq d_{e_1'} + d_{e_2'}$. Furthermore as T' is a tree $d(T) \leq d(T')$.
- So the cost of T_1 is less that or equal to the cost of any Hamiltonian tour.
- In the case T_1 is a tour we have found the optimal solution and can prune by bounding — otherwise we need to strengthen the bound or branch.

TSP: Bornholm Example



	A	B	C	D	E	F	G	H
A	0	11	24	25	30	29	15	15
B	11	0	13	20	32	37	17	17
C	24	13	0	16	30	39	29	22
D	25	20	16	0	15	23	18	12
E	30	32	30	15	0	9	23	15
F	29	37	39	23	9	0	14	21
G	15	17	29	18	23	14	0	7
H	15	17	22	12	15	21	7	0

TSP: 1-tree Bound of Bornholm



Tree in rest of G

Edge left out by Kruskal's MST algorithm

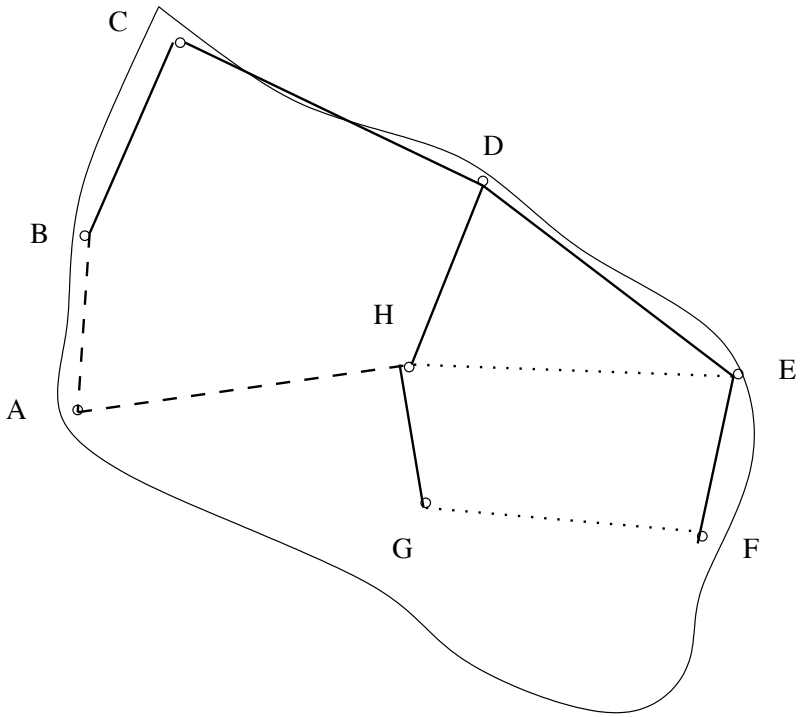
1–tree edge

Cost of 1–tree = 97

TSP: Strengthening the Bound

- Idea: Vertices of T_1 with high degree are incident with too many attractive edges. Vertices of degree 1 have on the other hand too many unattractive edges.
- Define π_i as the degree of vertex i minus 2.
- Note that $\sum_{i \in V} \pi_i$ equals 0 since T_1 has n edges and therefore the degree sum is $2n$.
- For each edge $(i, j) \in E$ we transform the cost to $d'_{ij} = d_{ij} + \pi_i + \pi_j$.

TSP: Strengthen the Bound



Cost of 1–tree = 97

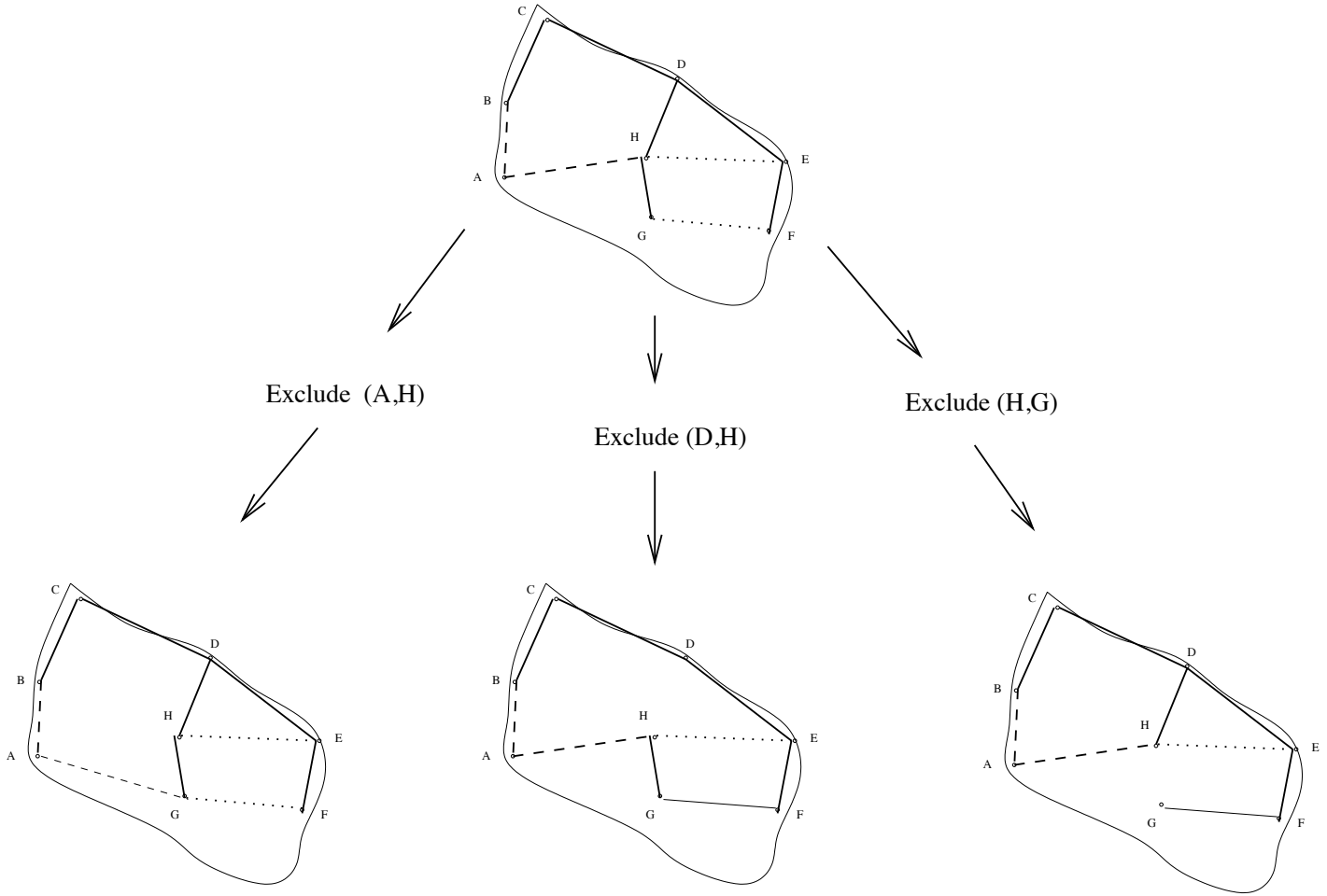
Modified distance matrix:

	A	B	C	D	E	F	G	H
A	0	11	24	25	29	29	16	15
B	11	0	13	20	31	37	18	17
C	24	13	0	16	29	39	30	22
D	25	20	16	0	14	23	19	12
E	29	31	29	14	0	8	23	14
F	29	37	39	23	8	0	15	21
G	16	18	30	19	23	15	0	8
H	15	17	22	12	14	21	8	0

TSP: How do we Branch?

- Observe that in the case our 1-tree is **not** a tour at least one vertex has degree 3 or more.
- So choose a vertex v with degree 3 or more.
- For each edge (u_i, v) generate a subproblem where (u_i, v) is excluded from the set of edges.

TSP: Branching on Bornholm



Select: Choose the Next Subproblem to Process

Critical subproblem: The lower bound is strictly smaller than the value of an optimal solution

Therefore: branching is necessary for critical subproblems independent on the selection strategy

Select: Choose the Next Subproblem to Process

Critical subproblem: The lower bound is strictly smaller than the value of an optimal solution

Therefore: branching is necessary for critical subproblems independent on the selection strategy

Common selection strategies:

Select: Choose the Next Subproblem to Process

Critical subproblem: The lower bound is strictly smaller than the value of an optimal solution

Therefore: branching is necessary for critical subproblems independent on the selection strategy

Common selection strategies:

- Best-first: Select the subproblem with the smallest lower bound

Select: Choose the Next Subproblem to Process

Critical subproblem: The lower bound is strictly smaller than the value of an optimal solution

Therefore: branching is necessary for critical subproblems independent on the selection strategy

Common selection strategies:

- Best-first: Select the subproblem with the smallest lower bound
- Breadth-first: Process all nodes at one level of the search tree before any nodes deeper in the tree

Select: Choose the Next Subproblem to Process

Critical subproblem: The lower bound is strictly smaller than the value of an optimal solution

Therefore: branching is necessary for critical subproblems independent on the selection strategy

Common selection strategies:

- Best-first: Select the subproblem with the smallest lower bound
- Breadth-first: Process all nodes at one level of the search tree before any nodes deeper in the tree
- Depth-first: Choose the deepest node in the search tree

Outline

Branch and Bound: Algorithm

1. $\textsc{Incumbent} := \infty$; $\textsc{Live} := \{(P_0, \infty)\}$
2. **repeat**
3. Select the node P from $\textsc{Live}$ to be processed
4. $\textsc{Live} := \textsc{Live} \setminus \{P\}$
5. Compute lower bound $LB(P)$
6. **if** $LB(P) = f(\mathbf{s})$ for some $\mathbf{s} \in \mathcal{S}$ **and** $f(\mathbf{s}) < \textsc{Incumbent}$ **then**
7. $\textsc{Incumbent} := f(\mathbf{s})$
8. $\textsc{OptimalSolution} := \mathbf{s}$
9. **else if** $LB(P) \geq \textsc{Incumbent}$ **then**
10. fathom P
11. **else**
12. Branch on P generating $P_1, \ldots, P_k$
13. **for** $1 \leq i \leq k$ **do**
14. $\textsc{Live} := \textsc{Live} \cup \{(P_i, LB(P))\}$
15. **until** Live $= \emptyset$

Outline

Neighbourhoods and Smoothness

Main idea:

Define a neighbourhood function $\mathcal{N}(\mathbf{s}) \subseteq \mathcal{S}$ for all $\mathbf{s} \in \mathcal{S}$

Defines a relation between candidate solutions: Neighbouring solutions are "similar" to each other

Neighbourhoods and Smoothness

Main idea:
Define a neighbourhood function $\mathcal{N}(\mathbf{s}) \subseteq \mathcal{S}$ for all $\mathbf{s} \in \mathcal{S}$

Defines a relation between candidate solutions: Neighbouring solutions are "similar" to each other

Goal: Neighbouring solutions should also have similar objective values, so that the solution space becomes "smooth"

Neighbourhoods and Smoothness

Main idea:
Define a neighbourhood function $\mathcal{N}(\mathbf{s}) \subseteq \mathcal{S}$ for all $\mathbf{s} \in \mathcal{S}$

Defines a relation between candidate solutions: Neighbouring solutions are "similar" to each other

Goal: Neighbouring solutions should also have similar objective values, so that the solution space becomes "smooth"

Local search: Traverse through the solution space using the neighbourhood relation

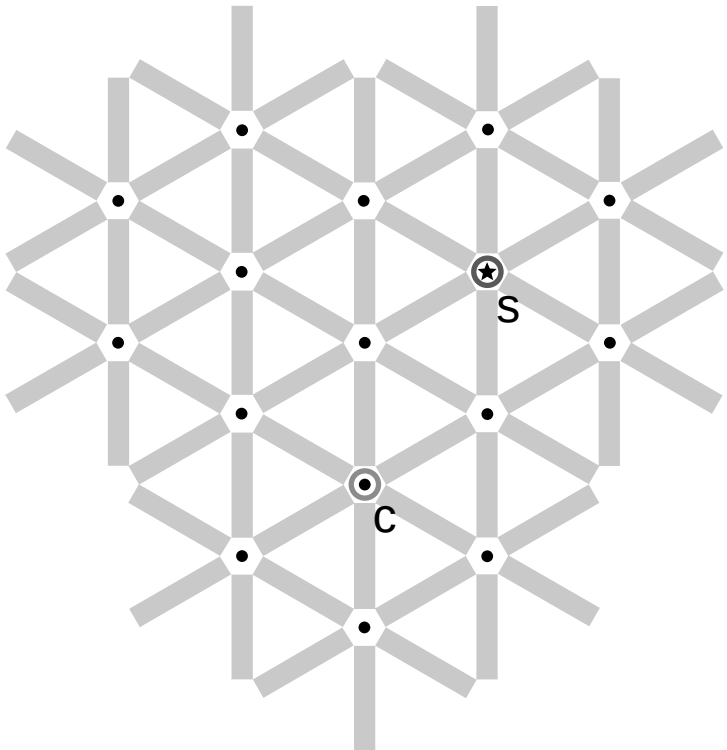
Local Search: General Algorithm

1. Choose an initial solution $\mathbf{s} \in \mathcal{S}$
2. Let $\mathbf{s}^* := \mathbf{s}$ be the *best solution* so far
3. **repeat**
4. Choose a solution $\mathbf{t} \in \mathcal{N}(\mathbf{s})$
5. **if** $f(\mathbf{t}) < f(\mathbf{s}^*)$ **then** $\mathbf{s}^* := \mathbf{t}$
6. **if** $\mathbf{t}$ is accepted **then** $\mathbf{s} := \mathbf{t}$
7. **until** stopping condition is met
8. **return** $\mathbf{s}^*$

Local Search: Global View

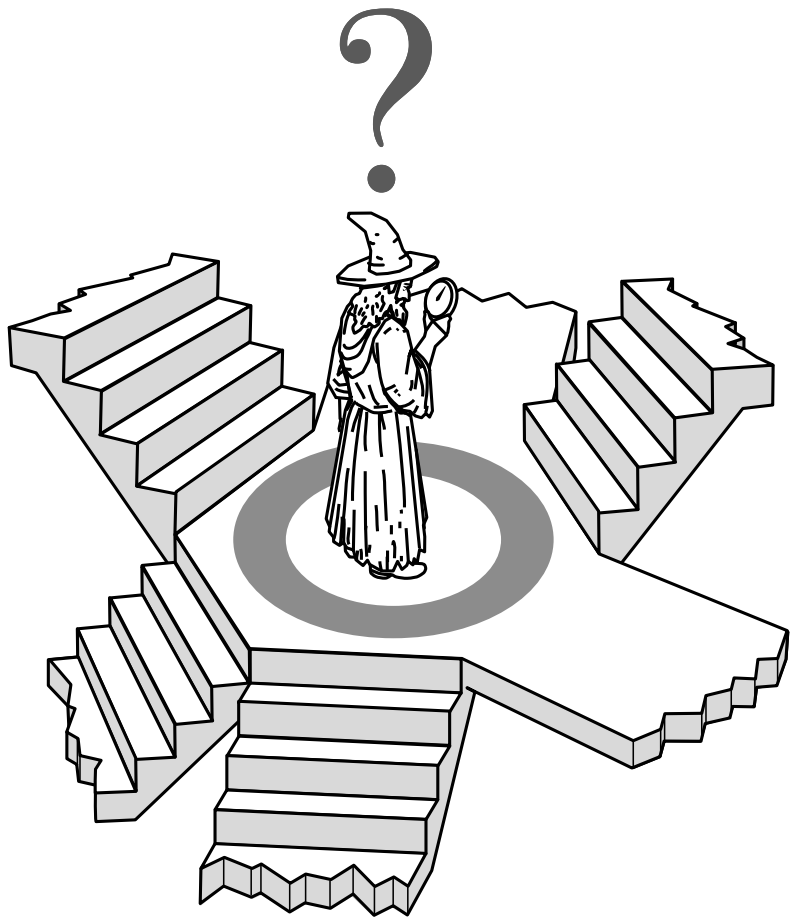
Vertices: candidate solutions (search positions)

Edges: connect neighbouring positions



Local Search: Local View

Next search position is selected from local neighbourhood based on local information, e.g., heuristic values.



Exploitation versus Exploration

Need to balance two contradicting search principles:

Exploitation versus Exploration

Need to balance two contradicting search principles:

Exploitation (or intensification): Search for solutions that are similar to the current solution → local search

Intuition: Assuming that the search space is smooth, good solutions can be found in the vicinity of other good solutions

Exploitation versus Exploration

Need to balance two contradicting search principles:

Exploitation (or intensification): Search for solutions that are similar to the current solution → local search

Intuition: Assuming that the search space is smooth, good solutions can be found in the vicinity of other good solutions

Exploration (or diversification): Search for solutions in the whole search space → random search

Intuition: Assuming that the search space has many local optima, it is necessary to make significant "jumps" in the search space to hit other good solutions

Representation of Solutions and Neighbourhoods

A candidate solution can be represented by, e.g.,

- a vector of bits, integers, reals...
- a permutation
- a subset of a ground set
- a rooted or unrooted tree
- an undirected or directed graph
- ...

Representation of Solutions and Neighbourhoods

A candidate solution can be represented by, e.g.,

- a vector of bits, integers, reals...
- a permutation
- a subset of a ground set
- a rooted or unrooted tree
- an undirected or directed graph
- ...

A neighbour is obtained by making a small modification in the underlying representation — a tweak of the corresponding candidate solution

Neighbourhood Structures using Sets

The neighbourhood design depends heavily on the solution representation

Assume the following set-representation:

- A solution is represented by elements in a ground set $E = \{1, 2, \ldots, m\}$.
- Solution space: $\mathcal{S} \subseteq 2^E$
- Thus for any solution $\mathbf{s} \in \mathcal{S}$ we have $\mathbf{s} \subseteq E$.

Task: Find a feasible subset of E with minimum objective value.

Canonical Neighbourhoods

Solution distance:

$$d(\mathbf{s}, \mathbf{t}) = |\mathbf{s} \setminus \mathbf{t}| + |\mathbf{t} \setminus \mathbf{s}|$$

(#elements that appear in $\mathbf{s}$ or $\mathbf{t}$ but not in both)

Distance-k neighbourhood:

$$\mathcal{N}_k(\mathbf{s}) = \{\mathbf{t} \in \mathcal{S} \ : \ d(\mathbf{s}, \mathbf{t}) \leq k\}$$

k-exchange neighbourhood:

$$\mathcal{N}^k(\mathbf{s}) = \{\mathbf{t} \in \mathcal{S} \ : \ |\mathbf{s} \setminus \mathbf{t}| = |\mathbf{t} \setminus \mathbf{s}| \leq k\}$$

If all feasible solutions have the same cardinality, then

$$\mathcal{N}^k(\mathbf{s}) = \mathcal{N}_{2k}(\mathbf{s})$$

Neighbourhood Size

The size of a neighbourhood is

$$\max_{\mathbf{s}\in\mathcal{S}} |\mathcal{N}(\mathbf{s})|$$

Usually the size is polynomial in $m = |E|$.

For fixed k, the size of $\mathcal{N}_k(\mathbf{s})$ and $\mathcal{N}^k(\mathbf{s})$ is polynomial.

Neighbourhood Size

The size of a neighbourhood is

$$\max_{\mathbf{s}\in\mathcal{S}} |\mathcal{N}(\mathbf{s})|$$

Usually the size is polynomial in $m = |E|$.

For fixed k, the size of $\mathcal{N}_k(\mathbf{s})$ and $\mathcal{N}^k(\mathbf{s})$ is polynomial.

Elementary observations:

- Larger neighbourhoods take longer to search
- Larger neighbourhoods usually result in better local optima

Thus the size and the quality of local optima should be well balanced in order to get an efficient neighbourhood.

Outline

Basic Hill-Climbing

Main idea:

Only make strictly improving moves.

1. Choose an initial solution $\mathbf{s} \in \mathcal{S}$
2. **repeat**
3. Choose a solution $\mathbf{t} \in \mathcal{N}(\mathbf{s})$
4. **if** $f(\mathbf{t}) < f(\mathbf{s})$ **then** $\mathbf{s} := \mathbf{t}$
5. **until** **s** is a local optimum
6. **return** **s**

A solution **s** is a local optimum (minimum of maximum) if

$$\forall \mathbf{t} \in \mathcal{N}(\mathbf{s}) \ : \ f(\mathbf{s}) \leq f(\mathbf{t})$$

Steepest Descent Hill-Climbing

Main idea:
Only make best possible improving moves.

1. Choose an initial solution $\mathbf{s} \in \mathcal{S}$
2. **repeat**
3. Choose the best solution $\mathbf{t} \in \mathcal{N}(\mathbf{s})$
4. $\mathbf{s} := \mathbf{t}$
5. **until** **s** is a local optimum
6. **return s**

Steepest Descent Hill-Climbing is sometimes denoted local optimization.

Iterated Local Search

Main idea:
Use two types of local search steps — one for intensification and one for diversification

- subsidiary steps for reaching local optima as efficiently as possible (intensification)
- perturbation steps for effectively escaping from local optima (diversification).

Also: Use acceptance criterion to control diversification vs intensification behaviour.

Outline

Metaheuristics: General Heuristic Methods

- General heuristics/methods/ideas/frameworks that can be used on several problems — in contrast to approximation algorithms which usually are very problem specific.
- Many metaheuristics are based on local search.
- Performance usually investigated empirically, since normally no useful worst-case guarantees can be given — a side effect of the general nature of the metaheuristics.
- However, in practice metaheuristics are powerful and easy to implement.

Simulated Annealing (SA)

Main idea:
Always perform improving moves, but also accept non-improving moves with some probability that decreases as the search progresses.

Corresponds to the physical process of cooling material in a heat bath — a process known as annealing.

Uses a parameter called the temperature T that controls the acceptance of non-improving moves.

Simulated Annealing: Basic Algorithm

1. Choose an initial solution $\mathbf{s} \in \mathcal{S}$
2. Let $\mathbf{s}^* := \mathbf{s}$ be the best solution so far
3. Choose a starting temperature T
4. **repeat**
5. **repeat**
6. Choose a *random* neighbour $\mathbf{t} \in \mathcal{N}(\mathbf{s})$
7. $\Delta := f(\mathbf{t}) - f(\mathbf{s})$
8. **if** $\Delta \leq 0$ **then**
9. $\mathbf{s} := \mathbf{t}$ (update)
10. **if** $f(\mathbf{s}) < f(\mathbf{s}^*)$ **then** $\mathbf{s}^* := \mathbf{s}$
11. **else**
12. Choose a random number r from [0,1]
13. **if** $r < e^{-\Delta/T}$ **then** $\mathbf{s} := \mathbf{t}$ (update)
14. **until** *equilibrium* for this temperature
15. Decrease the temperature T
16. **until** the system is *frozen*
17. **return** $\mathbf{s}^*$

Simulated Annealing: Generic Decisions

Generic decisions:

- initial solution
- starting temperature
- definition of frozen state
- definition of equilibrium
- function used for decreasing the temperature (cooling schedule)

Simulated Annealing: Cooling Schedule

Typical Cooling Schedule:

Initial temperature: Accept a certain percentage of candidate solutions.

Geometric cooling: $T := \alpha \cdot T$ where $0 < \alpha < 1$

Termination: No improvement over a number of iterations or acceptance ratio below some threshold.

Simulated Annealing: Cooling Schedule

Typical Cooling Schedule:

Initial temperature: Accept a certain percentage of candidate solutions.

Geometric cooling: $T := \alpha \cdot T$ where $0 < \alpha < 1$

Termination: No improvement over a number of iterations or acceptance ratio below some threshold.

Convergence property:

Under certain conditions (extremely slow cooling), any sufficiently long trajectory of SA is guaranteed to end in an optimal solution [Geman and Geman, 1984; Hajek, 1998].

Note: Practical relevance for combinatorial problem solving is very limited (necessary conditions not practical).

Tabu Search (TS)

Main idea:

Choose a most improving *or least non-improving* move in every step.
Avoid cycles by forbidding (or penalizing) certain solutions/moves,
so-called tabu solutions/moves.

Tabu Search (TS)

Main idea:
Choose a most improving *or least non-improving* move in every step.
Avoid cycles by forbidding (or penalizing) certain solutions/moves, so-called tabu solutions/moves.

In contrast to "simple" local search methods, tabu search

- collects information and uses memory while searching
- has no compact (or simple) definition
- requires more adaption to each problem type and neighbourhood structure
- is heavily influenced by perspectives from artificial intelligence (AI)

Presented in current form by Fred Glover in 1986.

Tabu Search: Implementing Tabu Memory

Tabu memory can be implemented in various ways:

- Strict (forbid all previous solutions)
- Fixed length solution list
- Fixed length move list
- Attribute based

Tabu Search: Basic Algorithm

1. Choose an initial solution $\mathbf{s} \in \mathcal{S}$
2. Let $\mathbf{s}^* := \mathbf{s}$ be the best solution so far
3. Initialize *tabu memory*
4. $k := 1$ (number of iterations)
5. **repeat**
6. Generate a *candidate set* $V \subseteq \mathcal{N}(\mathbf{s}, k) \subseteq \mathcal{N}(\mathbf{s})$
7. Find the *best* solution $\mathbf{t} \in V$
8. $\mathbf{s} := \mathbf{t}$ (update)
9. **if** $f(\mathbf{s}) < f(\mathbf{s}^*)$ **then** $\mathbf{s}^* := \mathbf{s}$
10. Update tabu memory
11. $k := k + 1$
12. **until** stopping condition is met
13. **return** $\mathbf{s}^*$

Tabu Search: Generic Decisions

Generic decisions:

- initial solution
- definition of tabu memory
- definition of tabu-based neighbourhood $\mathcal{N}(\mathbf{s}, k) \subseteq \mathcal{N}(\mathbf{s})$
- definition of candidate set $V \subseteq \mathcal{N}(\mathbf{s}, k)$
- stopping condition

Guided Local Search (GLS)

Main idea:
Perform local optimization iteratively — escape local minima by modifying the objective function

Local search algorithm:

- Based on iterative improvement (hill climbing)
- Experience from search history used to escape local optima; memory guides the search
- The objective function is modified

Only need to define a set of features and feature costs for the specific problem considered.

Features in GLS $\leftrightarrow$ Attributes in TS

Guided Local Search: Definition of Features

1. Features A set $M = \{1, \ldots, m\}$. The presence of a feature $i \in M$ in a solution $\mathbf{s} \in \mathcal{S}$ is represented by an indicator function:

$$I_i(\mathbf{s}) = \begin{cases} 1 & \text{iff } \mathbf{s} \in \mathcal{S} \text{ has feature } i \\ 0 & \text{otherwise} \end{cases}$$

2. Feature costs A cost c_i is associated to each feature $i \in M$. A feature of high cost should be penalized.

3. Augmented objective function

$$h(\mathbf{s}) = f(\mathbf{s}) + \lambda \cdot \sum_{i=1}^{M} p_i \cdot I_i(\mathbf{s})$$

where p_i denotes penalty. Initially $p_i = 0$.

Guided Local Search: Algorithm

1. Choose an initial solution $\mathbf{s} \in \mathcal{S}$
2. Let $\mathbf{s}^* := \mathbf{s}$ be the best solution so far
3. Set $p_i = 0$ for all $i \in M$
4. **repeat**
5. $\quad \mathbf{s} = \textsc{LocalOpt}_h(\mathbf{s})$
6. $\quad$ **if** $f(\mathbf{s}) < f(\mathbf{s}^*)$ **then** $\mathbf{s}^* := \mathbf{s}$
7. $\quad \mu_i(\mathbf{s}) = \frac{c_i}{1+p_i} \cdot I_i(\mathbf{s})$ for all $i \in M$
8. $\quad$ for each i such that $\mu_i(\mathbf{s})$ is maximum
   $\quad\quad$ set $p_i = p_i + 1$ (penalize)
9. **until** stopping condition is met
10. **return** $\mathbf{s}^*$

Guided Local Search: Generic Decisions

Generic decisions:

- initial solution
- definition of features
- definition of local optimization procedure $\textsc{LocalOpt}_h$
- stopping condition

Fast Local Search (FLS)

GLS dominated by the time to perform local optimizations ($\textsc{LocalOpt}_h$ procedure)

Finding improving neighbours in $\mathcal{N}(\mathbf{s})$ is time-consuming.

1. Divide neighbourhood into sub-neighbourhoods
2. Initially all sub-neighbourhoods are active
3. Scan sub-neighbourhoods in round-robin manner
4. If no improvement $\rightarrow$ deactivate sub-neighbourhood
5. Repeat until all sub-neighbourhoods are inactive

Activate sub-neighbourhoods for penalized features

Literature and Exercises

Literature

J. Clausen, *Branch and Bound Algorithms – Principles and Examples*, 1999. Sections 1, 2 and 4

Exercises

See Exercise Sheet on Absalon