Motivation – Introduction to Complexity Theory

Celso C. Ribeiro (celso@ic.uff.br)

University of Vienna

Metaheuristics – 2017-10-11

Overview of talk

- Problem formulation
  - Definitions
  - Examples
    - Shortest path problem
    - Minimum spanning tree problem
    - Steiner tree problem in graphs
    - Knapsack problem
    - Traveling salesman problem
- Polynomial (efficient) algorithms
- Characterization of problems and instances (cases)
- One problem has three versions
  - Decision problem
  - Recognition problem
  - Optimization problem
- The classes *P* and *NP*
- Polynomial transformations and *NP*-complete problems
- *PSPACE* and the polynomial hierarchy
- Solution approaches
  - Superpolynomial algorithms
  - Approximation algorithms
  - Parallel processing
  - Heuristics
    - Constructive heuristics
    - Local search
    - Metaheuristics
  - Goals of algorithmic research in metaheuristics

Problem formulation - Definitions

An instance of a combinatorial optimization problem is defined by

- a finite ground set $E = \{1, \ldots, n\}$
- a set of feasible solutions $F \subseteq 2^E$
- and an objective function $f : 2^E \to \mathbb{R}$

Problem formulation - Definitions

An instance of a combinatorial optimization problem is defined by

- a finite ground set $E = \{1, \ldots, n\}$
- a set of feasible solutions $F \subseteq 2^E$
- and an objective function $f : 2^E \to \mathbb{R}$

The finite ground set E, the set of feasible solutions F, and the objective function f are defined for each specific problem.

Problem formulation - Definitions

An instance of a combinatorial optimization problem is defined by

- a finite ground set $E = \{1, \ldots, n\}$
- a set of feasible solutions $F \subseteq 2^E$
- and an objective function $f : 2^E \to \mathbb{R}$

The finite ground set E, the set of feasible solutions F, and the objective function f are defined for each specific problem.

For a minimization problem, we seek a global optimal solution $S^* \in F$ such that

$$f(S^*) \leq f(S),\ \forall S \in F.$$

Problem formulation - Definitions

An instance of a combinatorial optimization problem is defined by

- a finite ground set $E = \{1, \ldots, n\}$
- a set of feasible solutions $F \subseteq 2^E$
- and an objective function $f : 2^E \to \mathbb{R}$

The finite ground set E, the set of feasible solutions F, and the objective function f are defined for each specific problem.

For a minimization problem, we seek a global optimal solution $S^* \in F$ such that

$$f(S^*) \leq f(S), \ \forall S \in F.$$

For a maximization problem, we seek a global optimal solution $S^* \in F$ such that

$$f(S^*) \geq f(S), \ \forall S \in F.$$

Shortest path problem - Revisited

Let $G = (V, A)$ be a directed graph, where V is its set of nodes and A its set of arcs.

- The origin s and the destination t are two special nodes in V.
- For every pair of nodes $s, t \in V$ connected by a path $P_{st}(G)$ formed by a sequence of nodes $s = i_1, i_2, \ldots, i_{q-1}, i_q = t \in V$, the length of this path is given by

$$f(P_{st}(G)) = \sum_{k=1}^{q-1} d_{i_k, i_{k+1}},$$

  where $d_{i,j}$ is the length of arc $(i, j) \in A$ and q is the number of arcs in the path.
- The shortest path problem (SPP) is easy: it can be solved in $O(|V|^2)$ time (Dijkstra, 1959).

Shortest path problem - Revisited

Let $G = (V, A)$ be a directed graph, where V is its set of nodes and A its set of arcs.

- The origin s and the destination t are two special nodes in V.
- For every pair of nodes $s, t \in V$ connected by a path $P_{st}(G)$ formed by a sequence of nodes $s = i_1, i_2, \ldots, i_{q-1}, i_q = t \in V$, the length of this path is given by

$$f(P_{st}(G)) = \sum_{k=1}^{q-1} d_{i_k, i_{k+1}},$$

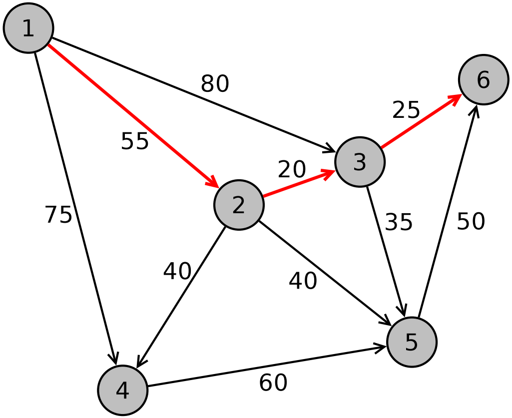
  where $d_{i,j}$ is the length of arc $(i, j) \in A$ and q is the number of arcs in the path.
- The shortest path problem (SPP) is easy: it can be solved in $O(|V|^2)$ time (Dijkstra, 1959).

In the case of the shortest path problem:

- The ground set consists of the arc set A.
- The set of feasible solutions F is formed by all subsets of arcs that are paths from s to t in G.
- The objective is to find a path $P^* \in F$ that minimizes the objective function $f(P)$ over all paths $P \in F$ from s to t in G.

Shortest path problem - Revisited

Consider the example in the figure, not drawn to scale.

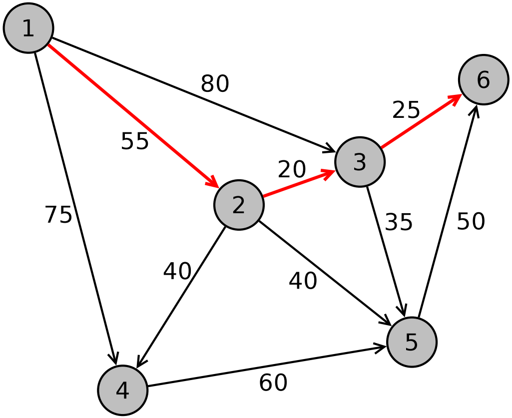


Shortest path problem - Revisited

Consider the example in the figure, not drawn to scale.

The shortest path from node 1 to node 6 is $1 - 2 - 3 - 6$ and is shown in red.

The length of this path is $55 + 20 + 25 = 100$.



Minimum spanning tree problem - Revisited

Let $G = (V, U)$ be a graph, where the node set V corresponds to points to be connected and its edge set U is formed by unordered pairs of points $i, j \in V$, with $i \neq j$.

- Let d_{ij} be the length (or weight) of edge $(i, j) \in U$.
- $T(G) = (V, U')$ is any spanning tree of graph G, i.e., a connected subgraph of G with the same node set V and whose edge set $U' \subseteq U$ has exactly $|V| - 1$ edges.
- The total weight of tree $T(G)$ is given by $f(T(G)) = \sum_{(i,j) \in U'} d_{ij}$.

Minimum spanning tree problem - Revisited

Let $G = (V, U)$ be a graph, where the node set V corresponds to points to be connected and its edge set U is formed by unordered pairs of points $i, j \in V$, with $i \neq j$.

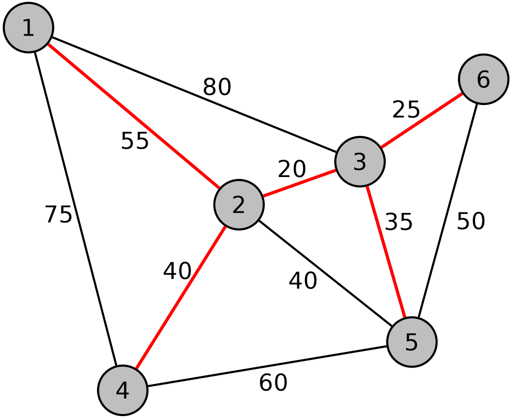
- Let d_{ij} be the length (or weight) of edge $(i, j) \in U$.
- $T(G) = (V, U')$ is any spanning tree of graph G, i.e., a connected subgraph of G with the same node set V and whose edge set $U' \subseteq U$ has exactly $|V| - 1$ edges.
- The total weight of tree $T(G)$ is given by $f(T(G)) = \sum_{(i,j)\in U'} d_{ij}$.
- The minimum spanning tree problem (MSTP) is easy: it can be solved in $O(|U| \log |V|)$ time (Kruskal, 1957).

In the case of the minimum spanning tree problem:

- The ground set consists of the edge set U.
- The set of feasible solutions F is formed by all subsets of edges that define spanning trees of G.
- The objective is to find a spanning tree $T^* \in F$ such that $f(T^*) \leq f(T)$ for all $T \in F$.

Minimium spanning tree problem - Revisited

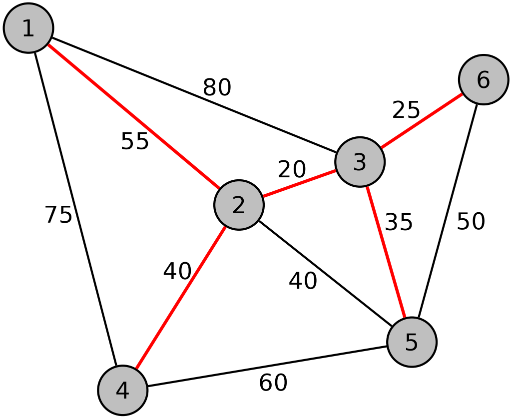
Consider the example in the figure, not drawn to scale.



Minimium spanning tree problem - Revisited

Consider the example in the figure, not drawn to scale.

- The minimum spanning tree of this graph is shown in red and has five edges: $(1, 2)$, $(2, 3)$, $(2, 4)$, $(3, 5)$, and $(3, 6)$.
- Its total weight is $55 + 20 + 40 + 35 + 25 = 175$.



Steiner tree problem in graphs - Revisited

Let $G = (V, U)$ be a graph, where the node set is $V = \{1, \ldots, n\}$ and the edge set U is formed by unordered pairs of nodes $i, j \in V$, with $i \neq j$.

- Let d_{ij} be the length of edge $(i, j) \in U$.
- $T \subseteq V$ is a subset of terminal nodes that have to be connected.
- A Steiner tree $S = (V', U')$ of G is a subtree of G that connects all nodes in T.
- The cost of the Steiner tree S is $f(S) = \sum_{(i,j) \in U'} d_{ij}$.

Steiner tree problem in graphs - Revisited

Let $G = (V, U)$ be a graph, where the node set is $V = \{1, \ldots, n\}$ and the edge set U is formed by unordered pairs of nodes $i, j \in V$, with $i \neq j$.

- Let d_{ij} be the length of edge $(i, j) \in U$.
- $T \subseteq V$ is a subset of terminal nodes that have to be connected.
- A Steiner tree $S = (V', U')$ of G is a subtree of G that connects all nodes in T.
- The cost of the Steiner tree S is $f(S) = \sum_{(i,j) \in U'} d_{ij}$.

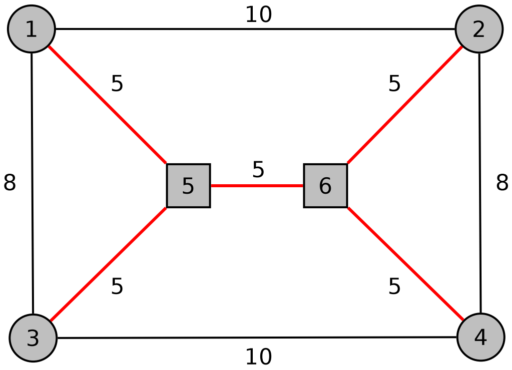
In the case of the Steiner tree problem in graphs:

- The ground set consists of the edge set U.
- The set of feasible solutions F is formed by all subsets of edges that define Steiner trees of G.
- The objective is to find a Steiner tree $S^* \in F$ such that $f(S^*) \leq f(S)$ for all $S \in F$.

We shall see that the Steiner tree problem (STP) in graphs is intractable or NP-hard (Karp, 1972).

Steiner tree problem in graphs - Revisited

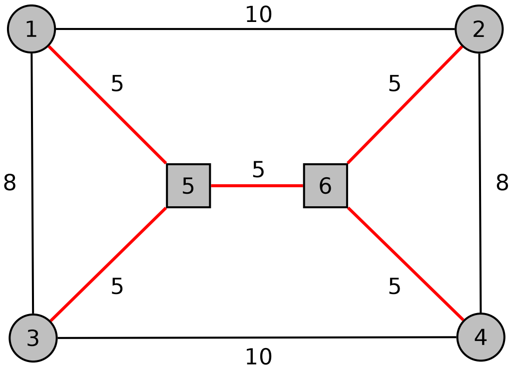
Consider the example in the figure, not drawn to scale.



Steiner tree problem in graphs - Revisited

Consider the example in the figure, not drawn to scale.

- The terminal nodes are represented by circles, while the optional nodes correspond to squares.
- The minimum Steiner tree is shown in red and makes use of the optional nodes 5 and 6.
- Its total cost is $5 + 5 + 5 + 5 + 5 = 25$.
- The nonterminal (optional) nodes in $V \setminus T$ that are effectively used to connect the terminal nodes in T are called Steiner nodes: nodes 5 and 6 in this example.
- The Steiner tree problem in graphs reduces to a shortest path problem when $|T| = 2$ (easy).
- It reduces to a minimum spanning tree problem when $T = V$ (also easy).



Knapsack problem - Revisited

Let b be an integer representing the maximum weight that can be taken in a hiker's knapsack and suppose the hiker has a set $I = \{1, \ldots, n\}$ of items to be placed in the knapsack.

- Let a_i be an integer number representing the weight of each item $i \in I$.
- Let c_i be an integer number representing the utility of each item $i \in I$.
- A subset of items $K \subseteq I$ is feasible if $\sum_{i \in K} a_i \leq b$.
- The utility of this subset K of items is $f(K) = \sum_{i \in K} c_i$.

Knapsack problem - Revisited

Let b be an integer representing the maximum weight that can be taken in a hiker's knapsack and suppose the hiker has a set $I = \{1, \ldots, n\}$ of items to be placed in the knapsack.

- Let a_i be an integer number representing the weight of each item $i \in I$.
- Let c_i be an integer number representing the utility of each item $i \in I$.
- A subset of items $K \subseteq I$ is feasible if $\sum_{i \in K} a_i \leq b$.
- The utility of this subset K of items is $f(K) = \sum_{i \in K} c_i$.

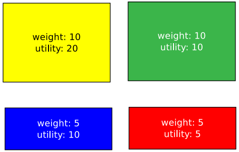
In the case of the knapsack problem:

- The ground set consists of the set I of items to be packed.
- The set of feasible solutions F is formed by all subsets of items $K \subseteq I$ for which $\sum_{i \in K} a_i \leq b$.
- The objective of the knapsack problem is to find a set of items $K^* \in F$ such that $f(K^*) \geq f(K)$ for all $K \in F$.

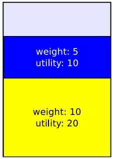
We shall see that the knapsack problem (KS) is also intractable or NP-hard.

Knapsack problem - Revisited

Consider the example in the figure, where four items are available to be placed in a knapsack of capacity 19.



(a) Items

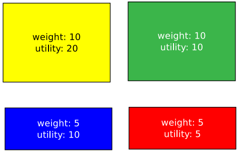


(b) Knapsack of capacity 19 with two items

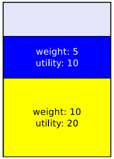
Knapsack problem - Revisited

Consider the example in the figure, where four items are available to be placed in a knapsack of capacity 19.

- The weights of the yellow and green items are equal to 10 and those of the blue and red items are equal to 5: therefore, only two of the four items fit together.
- The two heaviest items have utilities 20 and 10, while the two others have utilities 10 and 5. Since both large items cannot be placed together, the hiker will need to select a large and a small item.
- Of each group, the hiker selects the item with maximum utility: yellow and blue items are placed in the knapsack, with a weight of $5 + 10 = 15$ and a maximum utility of $10 + 20 = 30$.



(a) Items



(b) Knapsack of capacity 19 with two items

Traveling salesman problem - Revisited

Consider the graph $G = (V, U)$ with non-negative lengths d_{ij} associated with each existing edge $(i, j) \in U$, and let $V = \{1, \ldots, n\}$ be the set of cities a traveling salesman has to visit.

- A feasible solution to the traveling salesman problem is a tour defined by a circular permutation $\pi = (i_1, i_2, \ldots, i_n, i_1)$ of the n cities, with $i_j \neq i_k$ for every $j \neq k \in V$.
- This permutation is associated with the Hamiltonian cycle $H = \{(i_1, i_2), (i_2, i_3), \ldots, (i_{n-1}, i_n), (i_n, i_1)\}$ in G, i.e. $(i_n, i_1) \in U$ and $(i_k, i_{k+1}) \in U$, for $k = 1, \ldots, n-1$.
- The total length of this tour is given by $f(H) = \sum_{k=1}^{n-1} d_{i_k, i_{k+1}} + d_{i_n, i_1}$.

Traveling salesman problem - Revisited

Consider the graph $G = (V, U)$ with non-negative lengths d_{ij} associated with each existing edge $(i, j) \in U$, and let $V = \{1, \dots, n\}$ be the set of cities a traveling salesman has to visit.

- A feasible solution to the traveling salesman problem is a tour defined by a circular permutation $\pi = (i_1, i_2, \dots, i_n, i_1)$ of the n cities, with $i_j \neq i_k$ for every $j \neq k \in V$.
- This permutation is associated with the Hamiltonian cycle $H = \{(i_1, i_2), (i_2, i_3), \dots, (i_{n-1}, i_n), (i_n, i_1)\}$ in G, i.e. $(i_n, i_1) \in U$ and $(i_k, i_{k+1}) \in U$, for $k = 1, \dots, n-1$.
- The total length of this tour is given by $f(H) = \sum_{k=1}^{n-1} d_{i_k, i_{k+1}} + d_{i_n, i_1}$.

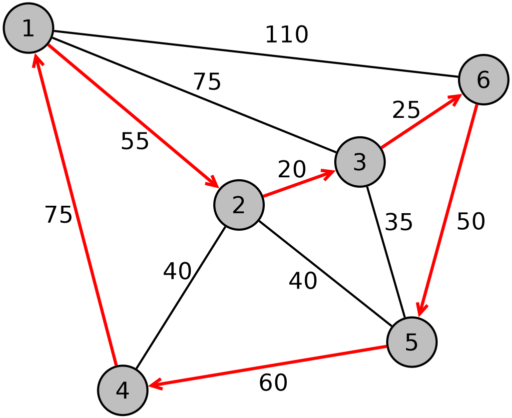
In the case of the traveling salesman problem:

- The ground set E consists of the set U of edges.
- The set of feasible solutions F is formed by all subsets of edges that correspond to Hamiltonian cycles in G.
- The objective of the traveling salesman problem is to find a Hamiltonian cycle $H^* \in F$ such that $f(H^*) \leq f(H)$ for all $H \in F$.

We shall see that the traveling salesman problem (TSP) is also intractable or NP-hard (Karp, 1972).

Traveling salesman problem - Revisited

Consider the example in the figure, not drawn to scale.



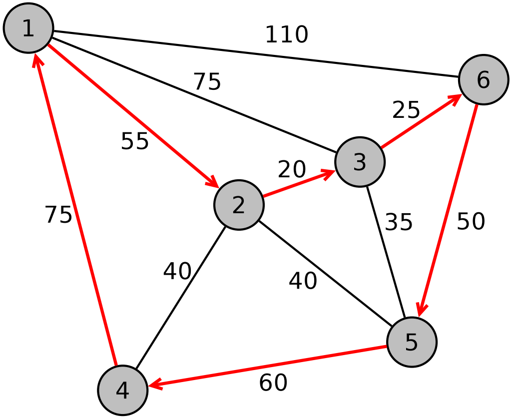
Traveling salesman problem - Revisited

Consider the example in the figure, not drawn to scale.

- An optimal tour

  $$H = \{(1,2),(2,3),(3,6),(6,5),(5,4),(4,1)\}$$

  is shown in red.
- It visits cities $1-2-3-6-5-4-1$ in this order.
- Its total length is $55+20+25+50+60+75=285$.



Polynomial-time algorithms

A problem is considered as well-solved if there exists an efficient algorithm for its exact solution.

Polynomial-time algorithms

A problem is considered as well-solved if there exists an efficient algorithm for its exact solution.

- Efficient algorithms are those that are not too time-consuming and whose computation times do not grow excessively fast with the problem size.
- The rate of growth of the time taken by an algorithm is the main limitation for its use in practice: algorithms with fast increasing computation times quickly become useless.
- An algorithm is said to be efficient (and therefore useful) for solving a problem whenever its running time (or time complexity) grows polynomially with the number of its variables.

Polynomial-time algorithms

A problem is considered as well-solved if there exists an efficient algorithm for its exact solution.

- Efficient algorithms are those that are not too time-consuming and whose computation times do not grow excessively fast with the problem size.
- The rate of growth of the time taken by an algorithm is the main limitation for its use in practice: algorithms with fast increasing computation times quickly become useless.
- An algorithm is said to be efficient (and therefore useful) for solving a problem whenever its running time (or time complexity) grows polynomially with the number of its variables.
- Polynomial algorithms are known for the shortest path problem and the minimum spanning tree problem.
- The Steiner tree problem in graphs, the maximum clique problem, the knapsack problem, and the traveling salesman problem are typical examples of hard problems for which, to date, no polynomial algorithm is known: hard optimization problems in this category are those that benefit from metaheuristics for their solution.

Characterization of problems and instances (cases)

Each instance (case) of a combinatorial optimization problem is characterized by:

- Finite ground set $E = \{1, \ldots, n\}$
- Set F of feasible solutions
- Cost function $f : F \to \mathbb{R}$ that associates a real value $f(S)$ with each feasible solution $S \in F$

Characterization of problems and instances (cases)

Each instance (case) of a combinatorial optimization problem is characterized by:

- Finite ground set $E = \{1, \ldots, n\}$
- Set F of feasible solutions
- Cost function $f : F \rightarrow \mathbb{R}$ that associates a real value $f(S)$ with each feasible solution $S \in F$

The set F of feasible solutions is implicitly given by a recognition algorithm $\mathcal{A}_F$:

- Given an object $S \in 2^E$ and a set P_F of parameters, the recognition algorithm determines if object S belongs to F, i.e., if S is a feasible solution.

Characterization of problems and instances (cases)

Each instance (case) of a combinatorial optimization problem is characterized by:

- Finite ground set $E = \{1, \ldots, n\}$
- Set F of feasible solutions
- Cost function $f : F \rightarrow \mathbb{R}$ that associates a real value $f(S)$ with each feasible solution $S \in F$

The set F of feasible solutions is implicitly given by a recognition algorithm $\mathcal{A}_F$:

- Given an object $S \in 2^E$ and a set P_F of parameters, the recognition algorithm determines if object S belongs to F, i.e., if S is a feasible solution.

The cost function $f(S)$ is implicitly given by a cost calculation algorithm $\mathcal{A}_f$:

- Given a feasible solution $S \in F$ and a set P_f of parameters, the cost calculation algorithm computes the cost function value $f(S)$ of a feasible solution S.

Characterization of problems and instances (cases)

Each instance (case) of a combinatorial optimization problem is characterized by:

- Finite ground set $E = \{1, \dots, n\}$
- Set F of feasible solutions
- Cost function $f : F \to \mathbb{R}$ that associates a real value $f(S)$ with each feasible solution $S \in F$

The set F of feasible solutions is implicitly given by a recognition algorithm $\mathcal{A}_F$:

- Given an object $S \in 2^E$ and a set P_F of parameters, the recognition algorithm determines if object S belongs to F, i.e., if S is a feasible solution.

The cost function $f(S)$ is implicitly given by a cost calculation algorithm $\mathcal{A}_f$:

- Given a feasible solution $S \in F$ and a set P_f of parameters, the cost calculation algorithm computes the cost function value $f(S)$ of a feasible solution S.

Therefore, each problem is characterized by the recognition algorithm $\mathcal{A}_F$ and the cost calculation algorithm $\mathcal{A}_f$, while each instance (case) is characterized by a pair of parameter sets P_F and P_f.

Characterization of problems and instances (cases)

Each instance (case) of a combinatorial optimization problem is characterized by:

- Finite ground set $E = \{1, \ldots, n\}$
- Set F of feasible solutions
- Cost function $f : F \to \mathbb{R}$ that associates a real value $f(S)$ with each feasible solution $S \in F$

Shortest path problem – Characterization

- Parameters for feasibility: directed graph $G = (V, A)$ with node set V and arc set A, source and destination nodes $s, t \in V$
- Parameters for cost function calculation: arc lengths d_{ij}, for every arc $(i, j) \in A$

Shortest path problem – Characterization

- Parameters for feasibility: directed graph $G = (V, A)$ with node set V and arc set A, source and destination nodes $s, t \in V$
- Parameters for cost function calculation: arc lengths d_{ij}, for every arc $(i, j) \in A$
- Ground set: arc set A
- Candidate object to be a feasible solution: any subset P of the arcs in A

Shortest path problem – Characterization

- Parameters for feasibility: directed graph $G = (V, A)$ with node set V and arc set A, source and destination nodes $s, t \in V$
- Parameters for cost function calculation: arc lengths d_{ij}, for every arc $(i, j) \in A$
- Ground set: arc set A
- Candidate object to be a feasible solution: any subset P of the arcs in A
- Recognition algorithm checks if candidate object P is a path from s to t in G.
- Cost calculation algorithm adds up the lengths of all arcs in P to compute the cost function value $f(P)$.

Minimum spanning tree problem – Characterization

- Parameters for feasibility: undirected graph $G = (V, U)$ with node set V and edge set U
- Parameters for cost function calculation: edge lengths d_{ij}, for every edge $(i, j) \in U$

Minimum spanning tree problem – Characterization

- Parameters for feasibility: undirected graph $G = (V, U)$ with node set V and edge set U
- Parameters for cost function calculation: edge lengths d_{ij}, for every edge $(i, j) \in U$
- Ground set: edge set U
- Candidate object to be a feasible solution: any subset T of the edges in U

Minimum spanning tree problem – Characterization

- Parameters for feasibility: undirected graph $G = (V, U)$ with node set V and edge set U
- Parameters for cost function calculation: edge lengths d_{ij}, for every edge $(i, j) \in U$
- Ground set: edge set U
- Candidate object to be a feasible solution: any subset T of the edges in U
- Recognition algorithm checks if candidate object T is a spanning tree in G.
- Cost calculation algorithm adds up the lengths of all edges in T to compute the cost function value $f(T)$.

Maximum clique problem – Characterization

- Parameters for feasibility: undirected graph $G = (V, U)$ with node set V and edge set U
- Parameters for cost function calculation:

Maximum clique problem – Characterization

- Parameters for feasibility: undirected graph $G = (V, U)$ with node set V and edge set U
- Parameters for cost function calculation:
- Ground set: edge set V
- Candidate object to be a feasible solution: any subset C of the nodes in V

Maximum clique problem – Characterization

- Parameters for feasibility: undirected graph $G = (V, U)$ with node set V and edge set U
- Parameters for cost function calculation:
- Ground set: edge set V
- Candidate object to be a feasible solution: any subset C of the nodes in V
- Recognition algorithm checks if candidate object C is a clique in G.
- Cost calculation algorithm counts the number of nodes in C to compute the cost function value $f(C)$.

Knapsack problem – Characterization

- Parameters for feasibility: maximum weight b of the knapsack and weights a_i of each item $i \in I$
- Parameters for cost function calculation: utilities c_i, for each item $i \in I$

Knapsack problem – Characterization

- Parameters for feasibility: maximum weight b of the knapsack and weights a_i of each item $i \in I$
- Parameters for cost function calculation: utilities c_i, for each item $i \in I$
- Ground set: item set I
- Candidate object to be a feasible solution: any subset K of the item set I

Knapsack problem – Characterization

- Parameters for feasibility: maximum weight b of the knapsack and weights a_i of each item $i \in I$
- Parameters for cost function calculation: utilities c_i, for each item $i \in I$
- Ground set: item set I
- Candidate object to be a feasible solution: any subset K of the item set I
- Recognition algorithm checks if candidate object K is a subset of I and if $\sum_{i \in K} a_i \leq b$.
- Cost calculation algorithm computes the total utility $f(K) = \sum_{i \in K} c_i \; K$ of the selected objects.

Traveling salesman problem – Characterization

- Parameters for feasibility: undirected graph $G = (V, U)$ with node set V representing the cities and edge set U, number of cities to be visited $|V|$
- Parameters for cost function calculation: edge lengths d_{ij}, for every edge $(i, j) \in U$

Traveling salesman problem – Characterization

- Parameters for feasibility: undirected graph $G = (V, U)$ with node set V representing the cities and edge set U, number of cities to be visited $|V|$
- Parameters for cost function calculation: edge lengths d_{ij}, for every edge $(i, j) \in U$
- Ground set: node set V
- Candidate object to be a feasible solution: any circular permutation of all cities in the node set V

Traveling salesman problem – Characterization

- Parameters for feasibility: undirected graph $G = (V, U)$ with node set V representing the cities and edge set U, number of cities to be visited $|V|$
- Parameters for cost function calculation: edge lengths d_{ij}, for every edge $(i, j) \in U$
- Ground set: node set V
- Candidate object to be a feasible solution: any circular permutation of all cities in the node set V
- Recognition algorithm checks if there is an edge between each pair of consecutive cities.
- Cost calculation algorithm computes the total length of the edges between each pair of consecutive cities in the circular permutation.

Traveling salesman problem – Characterization

- Parameters for feasibility: undirected graph $G = (V, U)$ with node set V representing the cities and edge set U, number of cities to be visited $|V|$
- Parameters for cost function calculation: edge lengths d_{ij}, for every edge $(i, j) \in U$
- Ground set: node set V
- Candidate object to be a feasible solution: any circular permutation of all cities in the node set V
- Recognition algorithm checks if there is an edge between each pair of consecutive cities.
- Cost calculation algorithm computes the total length of the edges between each pair of consecutive cities in the circular permutation.

OR, ALTERNATIVELY:

Traveling salesman problem – Characterization

- Parameters for feasibility: undirected graph $G = (V, U)$ with node set V representing the cities and edge set U, number of cities to be visited $|V|$
- Parameters for cost function calculation: edge lengths d_{ij}, for every edge $(i, j) \in U$
- Ground set: node set V
- Candidate object to be a feasible solution: any circular permutation of all cities in the node set V
- Recognition algorithm checks if there is an edge between each pair of consecutive cities.
- Cost calculation algorithm computes the total length of the edges between each pair of consecutive cities in the circular permutation.

OR, ALTERNATIVELY:

- Ground set: edge set U
- Candidate object to be a feasible solution: any subset H of the edge set U

Traveling salesman problem – Characterization

- Parameters for feasibility: undirected graph $G = (V, U)$ with node set V representing the cities and edge set U, number of cities to be visited $|V|$
- Parameters for cost function calculation: edge lengths d_{ij}, for every edge $(i, j) \in U$
- Ground set: node set V
- Candidate object to be a feasible solution: any circular permutation of all cities in the node set V
- Recognition algorithm checks if there is an edge between each pair of consecutive cities.
- Cost calculation algorithm computes the total length of the edges between each pair of consecutive cities in the circular permutation.

OR, ALTERNATIVELY:

- Ground set: edge set U
- Candidate object to be a feasible solution: any subset H of the edge set U
- Recognition algorithm checks if edges in the candidate subset H define a Hamiltonian cycle of G visiting every node in V exactly once.
- Cost calculation algorithm computes the sum of the lengths of the edges in the Hamiltonian cycle H.

One problem has three versions

A combinatorial optimization problem can be alternatively stated as:

Optimization problem

Given representations for the parameter sets P_F and P_f for algorithms $\mathcal{A}_F$ and $\mathcal{A}_f$, respectively, find an optimal feasible solution.

One problem has three versions

A combinatorial optimization problem can be alternatively stated as:

Optimization problem

Given representations for the parameter sets P_F and P_f for algorithms $\mathcal{A}_F$ and $\mathcal{A}_f$, respectively, find an optimal feasible solution.

If instead of finding an optimal solution itself, we are only interested in finding its value, then we have a more relaxed evaluation form of the problem:

Evaluation problem

Given representations for the parameter sets P_F and P_f for algorithms $\mathcal{A}_F$ and $\mathcal{A}_f$, respectively, find the cost of an optimal feasible solution.

One problem has three versions

A combinatorial optimization problem can be alternatively stated as:

Optimization problem

Given representations for the parameter sets P_F and P_f for algorithms $\mathcal{A}_F$ and $\mathcal{A}_f$, respectively, find an optimal feasible solution.

If instead of finding an optimal solution itself, we are only interested in finding its value, then we have a more relaxed evaluation form of the problem:

Evaluation problem

Given representations for the parameter sets P_F and P_f for algorithms $\mathcal{A}_F$ and $\mathcal{A}_f$, respectively, find the cost of an optimal feasible solution.

If the value of any solution can be efficiently computed, the evaluation version cannot be harder than the optimization version: once the optimization version has been solved and its optimal solution is known, its value can be easily computed by the cost calculation algorithm $\mathcal{A}_f$.

One problem has three versions

A third problem version is particularly important in the context of complexity theory.

The decision version of a minimization problem is simply a question requiring a "yes" or "no" answer:

Decision problem

Given representations for parameter sets P_F and P_f for algorithms $\mathcal{A}_F$ and $\mathcal{A}_f$, respectively, and an integer number B that represents a bound, is there a feasible solution $S \in F$ such that $f(S) \leq B$?

One problem has three versions

A third problem version is particularly important in the context of complexity theory.

The decision version of a minimization problem is simply a question requiring a “yes” or “no” answer:

Decision problem

Given representations for parameter sets P_F and P_f for algorithms $\mathcal{A}_F$ and $\mathcal{A}_f$, respectively, and an integer number B that represents a bound, is there a feasible solution $S \in F$ such that $f(S) \leq B$?

The decision version of a maximization problem asks for the existence of a feasible solution with cost greater than or equal to B.

The decision version of a combinatorial optimization problem cannot be harder than its evaluation version: once the optimal value has been obtained as the solution of the evaluation version, we can just compare it with the value of B to give a “yes” or “no” answer to the decision version.

One problem has three versions

A third problem version is particularly important in the context of complexity theory.

The decision version of a minimization problem is simply a question requiring a "yes" or "no" answer:

Decision problem

Given representations for parameter sets P_F and P_f for algorithms $\mathcal{A}_F$ and $\mathcal{A}_f$, respectively, and an integer number B that represents a bound, is there a feasible solution $S \in F$ such that $f(S) \leq B$?

The decision version of a maximization problem asks for the existence of a feasible solution with cost greater than or equal to B.

The decision version of a combinatorial optimization problem cannot be harder than its evaluation version: once the optimal value has been obtained as the solution of the evaluation version, we can just compare it with the value of B to give a "yes" or "no" answer to the decision version.

We have therefore established a problem hierarchy:

- The decision version is not harder than the evaluation version.
- The evaluation version is not harder than the optimization version.

One problem has three versions

Algorithm TSPOPT(n, d) for the optimization version of the traveling salesman problem.

begin TSPOPT(n, d);

$LB \leftarrow 0$;

$UB \leftarrow n \cdot \max_{i,j \in V : i \neq j}\{d_{ij}\}$;

$BIG \leftarrow UB + 1$;

while $UB \neq LB$ **do**

  if $\text{TSPDEC}(n, d, \lfloor (LB + UB)/2 \rfloor) = \text{"yes"}$

  then $UB \leftarrow \lfloor (LB + UB)/2 \rfloor$;

  else $LB \leftarrow \lfloor (LB + UB)/2 \rfloor$;

  end-if;

end-while;

$OPT \leftarrow UB$;

...

One problem has three versions

Algorithm TSPOPT(n, d) for the optimization version of the traveling salesman problem.

- Algorithm is based on the repeated execution of algorithm TSPDEC(n, d, B) for the decision version.
- First part: solve the evaluation version by computing the cost OPT of the optimal solution.
- $O(\log UB)$ iterations.

begin TSPOPT(n, d);
$LB \leftarrow 0$;
$UB \leftarrow n \cdot \max_{i,j \in V : i \neq j}\{d_{ij}\}$;
$BIG \leftarrow UB + 1$;
while $UB \neq LB$ **do**
    if $\mathrm{TSPDEC}(n, d, \lfloor (LB + UB)/2 \rfloor) =$ "yes"
    then $UB \leftarrow \lfloor (LB + UB)/2 \rfloor$;
    else $LB \leftarrow \lfloor (LB + UB)/2 \rfloor$;
    end-if;
end-while;
$OPT \leftarrow UB$;
...

Algorithm TSPOPT(n, d) for the optimization version of the traveling salesman problem.

$OPT \leftarrow UB$;
for $j = 1, \ldots, n$ **do**
  for $i = 1, \ldots, n$ with $i \neq j$ **do**
    $TMP \leftarrow d_{ij}$;
    $d_{ij} \leftarrow BIG$;
    if $\text{TSPDEC}(n, d, OPT) = \text{“no”}$
    then $d_{ij} \leftarrow TMP$;
  end-for;
end-for;
$S^* \leftarrow \varnothing$;
for $j = 1, \ldots, n$ **do**
  for $i = 1, \ldots, n$ with $i \neq j$ **do**
    if $d_{ij} \neq BIG$ **then** $S^* \leftarrow S^* \cup \{(i, j)\}$;
  end-for;
end-for;
return S^*, OPT;
end TSPOPT.

Algorithm TSPOPT(n, d) for the optimization version of the traveling salesman problem.

- Algorithm is based on the repeated execution of algorithm TSPDEC(n, d, B) for the decision version.
- Second part: solve the optization version from the ptrviously computed cost *OPT* of the optimal solution.
- $O(n^2)$ iterations.

$OPT \leftarrow UB$;
for $j = 1, \dots, n$ **do**
    for $i = 1, \dots, n$ with $i \neq j$ **do**
        $TMP \leftarrow d_{ij}$;
        $d_{ij} \leftarrow BIG$;
        if $\mathrm{TSPDEC}(n, d, OPT) = \text{“no”}$
        then $d_{ij} \leftarrow TMP$;
    end-for;
end-for;
$S^* \leftarrow \emptyset$;
for $j = 1, \dots, n$ **do**
    for $i = 1, \dots, n$ with $i \neq j$ **do**
        if $d_{ij} \neq BIG$ **then** $S^* \leftarrow S^* \cup \{(i, j)\}$;
    end-for;
end-for;
return S^*, OPT;
end TSPOPT.

One problem has three versions

Algorithm TSPOPT(n, d) for the optimization version of the traveling salesman problem.

- Algorithm is based on the repeated execution of algorithm TSPDEC(n, d, B) for the decision version.
- Second part: solve the optization version from the ptrviously computed cost OPT of the optimal solution.
- $O(n^2)$ iterations.
- Overall complexity: $O((\log UB + n^2) \cdot T(n))$, where $T(n)$ is the complexity of solving TSPDEC(n, d, B).
- Similar constructions available to most problems.

```
OPT ← UB;
for j = 1, ..., n do
    for i = 1, ..., n with i ≠ j do
        TMP ← d_ij;
        d_ij ← BIG;
        if TSPDEC(n, d, OPT) = "no"
        then d_ij ← TMP;
    end-for;
end-for;
S* ← ∅;
for j = 1, ..., n do
    for i = 1, ..., n with i ≠ j do
        if d_ij ≠ BIG then S* ← S* ∪ {(i, j)};
    end-for;
end-for;
return S*, OPT;
end TSPOPT.
```

Maximum clique problem – Problem versions

Optimization version

Given a graph $G = (V, U)$, find a maximum cardinality clique of G.

Maximum clique problem – Problem versions

Optimization version

Given a graph $G = (V, U)$, find a maximum cardinality clique of G.

Evaluation version

Given a graph $G = (V, U)$, find the number of nodes in a maximum cardinality clique of G.

Maximum clique problem – Problem versions

Optimization version

Given a graph $G = (V, U)$, find a maximum cardinality clique of G.

Evaluation version

Given a graph $G = (V, U)$, find the number of nodes in a maximum cardinality clique of G.

Decision version

Given a graph $G = (V, U)$ and an integer number B, is there a clique in G with at least B nodes?

Knapsack problem – Problem versions

Optimization version

Given a set $I = \{1, \ldots, n\}$ of items, integer weights a_i and utilities c_i associated with each item $i \in I$, and a maximum weight capacity b, find a subset $K^* \subseteq I$ of items such that $\sum_{i \in K^*} c_i = \max_{K \subseteq I} \{\sum_{i \in K} c_i : \sum_{i \in K} a_i \leq b\}$.

Knapsack problem – Problem versions

Optimization version

Given a set $I = \{1, \ldots, n\}$ of items, integer weights a_i and utilities c_i associated with each item $i \in I$, and a maximum weight capacity b, find a subset $K^* \subseteq I$ of items such that
$\sum_{i \in K^*} c_i = \max_{K \subseteq I} \{\sum_{i \in K} c_i : \sum_{i \in K} a_i \leq b\}$.

Evaluation version

Given a set $I = \{1, \ldots, n\}$ of items, integer weights a_i and utilities c_i associated with each item $i \in I$, and a maximum weight capacity b, find $c^* = \max_{K \subseteq I} \{\sum_{i \in K} c_i : \sum_{i \in K} a_i \leq b\}$.

Knapsack problem – Problem versions

Optimization version

Given a set $I = \{1, \ldots, n\}$ of items, integer weights a_i and utilities c_i associated with each item $i \in I$, and a maximum weight capacity b, find a subset $K^* \subseteq I$ of items such that $\sum_{i \in K^*} c_i = \max_{K \subseteq I}\{\sum_{i \in K} c_i : \sum_{i \in K} a_i \leq b\}$.

Evaluation version

Given a set $I = \{1, \ldots, n\}$ of items, integer weights a_i and utilities c_i associated with each item $i \in I$, and a maximum weight capacity b, find $c^* = \max_{K \subseteq I}\{\sum_{i \in K} c_i : \sum_{i \in K} a_i \leq b\}$.

Decision version

Given a set $I = \{1, \ldots, n\}$ of items, integer weights a_i and utilities c_i associated with each item $i \in I$, a maximum weight capacity b, and an integer B, is there $K \subseteq I$ such that $\sum_{i \in K} a_i \leq b$ and $\sum_{i \in K} c_i \geq B$?

Traveling salesman problem – Problem versions

Optimization version

Given a complete graph $G = (V, U)$ with non-negative distances d_{ij} between every pair of nodes $i, j \in V$, find a shortest Hamiltonian cycle in G.

Traveling salesman problem – Problem versions

Optimization version

Given a complete graph $G = (V, U)$ with non-negative distances d_{ij} between every pair of nodes $i, j \in V$, find a shortest Hamiltonian cycle in G.

Evaluation version

Given a complete graph $G = (V, U)$ with non-negative distances d_{ij} between every pair of nodes $i, j \in V$, compute the length of a shortest Hamiltonian cycle in G.

Traveling salesman problem – Problem versions

Optimization version

Given a complete graph $G = (V, U)$ with non-negative distances d_{ij} between every pair of nodes $i, j \in V$, find a shortest Hamiltonian cycle in G.

Evaluation version

Given a complete graph $G = (V, U)$ with non-negative distances d_{ij} between every pair of nodes $i, j \in V$, compute the length of a shortest Hamiltonian cycle in G.

Decision version

Given a complete graph $G = (V, U)$ with non-negative distances d_{ij} between every pair of nodes $i, j \in V$ and an integer B, is there a Hamiltonian cycle in G of length less than or equal to B?

One problem has three versions

There is a problem hierarchy:

- The decision version is not harder than the evaluation version.
- The evaluation version is not harder than the optimization version.

One problem has three versions

There is a problem hierarchy:

- The decision version is not harder than the evaluation version.
- The evaluation version is not harder than the optimization version.

Under very reasonable assumptions, the three versions of any combinatorial problem have roughly the same computational complexity:

- If we have a polynomial-time algorithm to solve the decision version of a combinatorial problem, then in general we can also construct polynomial-time algorithms for solving the evaluation and the optimization versions.

Decision problems offer a simpler and more structured framework for the study of complexity theory.

If a decision problem cannot be solved in polynomial time, then its corresponding optimization version cannot be solved in polynomial time as well.

The classes P and NP

The decision version of a combinatorial optimization problem amounts to a question that can be answered by either “yes” or “no”. Some examples:

The classes P and NP

The decision version of a combinatorial optimization problem amounts to a question that can be answered by either "yes" or "no". Some examples:

SHORTEST PATH

Given a directed graph $G = (V, A)$, an origin node $s \in V$, a destination node $t \in V$, lengths d_{ij} associated with every arc $(i, j) \in A$, and an integer B, is there a path from s to t in G whose length is less than or equal to B?

The classes P and NP

The decision version of a combinatorial optimization problem amounts to a question that can be answered by either "yes" or "no". Some examples:

SHORTEST PATH

Given a directed graph $G = (V, A)$, an origin node $s \in V$, a destination node $t \in V$, lengths d_{ij} associated with every arc $(i, j) \in A$, and an integer B, is there a path from s to t in G whose length is less than or equal to B?

MINIMUM SPANNING TREE

Given a graph $G = (V, U)$, a weight d_{ij} associated with each edge $(i, j) \in U$, and an integer B, is there a spanning tree of G such that the sum of the weights of its edges is less than or equal to B?

The classes P and NP

STEINER TREE IN GRAPHS

Given a graph $G = (V, U)$, lengths d_{ij} associated with each edge $(i, j) \in U$, a subset $T \subseteq V$, and an integer B, is there a subtree of G that connects all nodes in T and such that the sum of its edge lengths is less than or equal to B?

The classes P and NP

STEINER TREE IN GRAPHS

Given a graph $G = (V, U)$, lengths d_{ij} associated with each edge $(i, j) \in U$, a subset $T \subseteq V$, and an integer B, is there a subtree of G that connects all nodes in T and such that the sum of its edge lengths is less than or equal to B?

CLIQUE

Given a graph $G = (V, U)$ and an integer B, is there a clique in G with at least B nodes?

The classes P and NP

STEINER TREE IN GRAPHS

Given a graph $G = (V, U)$, lengths d_{ij} associated with each edge $(i, j) \in U$, a subset $T \subseteq V$, and an integer B, is there a subtree of G that connects all nodes in T and such that the sum of its edge lengths is less than or equal to B?

CLIQUE

Given a graph $G = (V, U)$ and an integer B, is there a clique in G with at least B nodes?

KNAPSACK

Given a set $I = \{1, \ldots, n\}$ of items, integer weights a_i and utilities c_i associated with each item $i \in I$, a maximum weight capacity b, and an integer B, is there a subset of items $K \subseteq I$ such that $\sum_{i \in K} a_i \leq b$ and $\sum_{i \in K} c_i \geq B$?

The classes P and NP

TRAVELING SALESMAN PROBLEM (TSP)

Given a set $V = \{1, \ldots, n\}$ of cities and non-negative distances d_{ij} between every pair of cities $i, j \in V$, with $i \neq j$, and an integer B, is there a tour visiting every city of V exactly once with length less than or equal to B?

The classes P and NP

TRAVELING SALESMAN PROBLEM (TSP)

Given a set $V = \{1, \ldots, n\}$ of cities and non-negative distances d_{ij} between every pair of cities $i, j \in V$, with $i \neq j$, and an integer B, is there a tour visiting every city of V exactly once with length less than or equal to B?

INDEPENDENT SET

Given a graph $G = (V, U)$ and an integer B, is there an independent set of nodes in G (i.e., a subset of mutually nonadjacent nodes) with at least B nodes?

The classes P and NP

TRAVELING SALESMAN PROBLEM (TSP)

Given a set $V = \{1, \ldots, n\}$ of cities and non-negative distances d_{ij} between every pair of cities $i, j \in V$, with $i \neq j$, and an integer B, is there a tour visiting every city of V exactly once with length less than or equal to B?

INDEPENDENT SET

Given a graph $G = (V, U)$ and an integer B, is there an independent set of nodes in G (i.e., a subset of mutually nonadjacent nodes) with at least B nodes?

GRAPH COLORING

Given a graph $G = (V, U)$ and an integer B, is it possible to color the nodes of G with at most B colors, such that adjacent nodes receive different colors?

The classes P and NP

LINEAR PROGRAMMING

Given an $m \times n$ matrix A of integer numbers, an integer m-vector b, an integer n-vector c, and an integer B, is there an n-vector $x \geq 0$ of rational numbers such that $A \cdot x = b$ and $c \cdot x \leq B$?

The classes P and NP

LINEAR PROGRAMMING

Given an $m \times n$ matrix A of integer numbers, an integer m-vector b, an integer n-vector c, and an integer B, is there an n-vector $x \geq 0$ of rational numbers such that $A \cdot x = b$ and $c \cdot x \leq B$?

INTEGER PROGRAMMING

Given an $m \times n$ matrix A of integer numbers, an integer m-vector b, an integer n-vector c, and an integer B, is there an n-vector $x \geq 0$ of integer numbers such that $A \cdot x = b$ and $c \cdot x \leq B$?

The classes P and NP

There are other decision problems that have not been originally cast as optimization problems. Some examples:

The classes P and NP

There are other decision problems that have not been originally cast as optimization problems. Some examples:

HAMILTONIAN CYCLE

Given a graph $G = (V, U)$, is there a Hamiltonian cycle in G visiting all its nodes exactly once?

The classes P and NP

There are other decision problems that have not been originally cast as optimization problems. Some examples:

HAMILTONIAN CYCLE

Given a graph $G = (V, U)$, is there a Hamiltonian cycle in G visiting all its nodes exactly once?

GRAPH PLANARITY

Given a graph $G = (V, U)$, is it planar?

The classes P and NP

There are other decision problems that have not been originally cast as optimization problems. Some examples:

HAMILTONIAN CYCLE

Given a graph $G = (V, U)$, is there a Hamiltonian cycle in G visiting all its nodes exactly once?

GRAPH PLANARITY

Given a graph $G = (V, U)$, is it planar?

GRAPH CONNECTEDNESS

Given a graph $G = (V, U)$, is it connected?

The classes P and NP

SATISFIABILITY (SAT)

Given m disjunctive clauses $C_1, \ldots, C_m$ involving the Boolean variables $x_1, \ldots, x_n$ and their complements, is there a truth assignment of 0 (false) and 1 (true) values to these variables such that the formula $C_1 \wedge C_2 \wedge \cdots \wedge C_m$ is satisfiable?

SATISFIABILITY (SAT)

Given m disjunctive clauses $C_1, \ldots, C_m$ involving the Boolean variables $x_1, \ldots, x_n$ and their complements, is there a truth assignment of 0 (false) and 1 (true) values to these variables such that the formula $C_1 \wedge C_2 \wedge \cdots \wedge C_m$ is satisfiable?

$$(x_1 \vee x_2 \vee x_5) \wedge (\bar{x}_2 \vee \bar{x}_5) \wedge (\bar{x}_1 \vee x_3 \vee x_4 \vee x_5)$$

$$x_1 = x_3 = x_4 = 1, \quad x_2 = x_5 = 0 \longrightarrow \text{TRUE}$$

The classes P and NP

SATISFIABILITY (SAT)

Given m disjunctive clauses $C_1, \dots, C_m$ involving the Boolean variables $x_1, \dots, x_n$ and their complements, is there a truth assignment of 0 (false) and 1 (true) values to these variables such that the formula $C_1 \wedge C_2 \wedge \dots \wedge C_m$ is satisfiable?

$$(x_1 \vee x_2 \vee x_5) \wedge (\bar{x}_2 \vee \bar{x}_5) \wedge (\bar{x}_1 \vee x_3 \vee x_4 \vee x_5)$$

$$x_1 = x_3 = x_4 = 1, \quad x_2 = x_5 = 0 \longrightarrow \text{TRUE}$$

$$(x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_2)$$

always FALSE

The classes P and NP

Class P

A decision problem $\mathcal{P}$ belongs to the class P if there exists an algorithm $\mathcal{A}$ that solves any of its instances in polynomial time.

Class P is formed by "easy" decision problems that can be efficiently solved by polynomial-time algorithms.

The classes P and NP

Class P

A decision problem $\mathcal{P}$ belongs to the class P if there exists an algorithm $\mathcal{A}$ that solves any of its instances in polynomial time.

Class P is formed by "easy" decision problems that can be efficiently solved by polynomial-time algorithms.

Examples of problems in this class:

- SHORTEST PATH
- MINIMUM SPANNING TREE
- GRAPH CONNECTEDNESS
- LINEAR PROGRAMMING
- 2-SAT (special case of SATISFIABILITY, in which every clause has exactly two variables or their complements),

The classes P and NP

Certificate

Given a decision problem $\mathcal{P}$ and a “yes” instance $\mathcal{J}$, a certificate $c(\mathcal{J})$ is a string that encodes a solution and makes it possible to reach the “yes” answer for instance $\mathcal{J}$.

Certificate

Given a decision problem $\mathcal{P}$ and a "yes" instance $\mathcal{J}$, a certificate $c(\mathcal{J})$ is a string that encodes a solution and makes it possible to reach the "yes" answer for instance $\mathcal{J}$.

Concise certificate

A certificate is said to be concise if the length of its encoding is polynomial in the amount of memory that is used to encode instance $\mathcal{J}$.

The classes P and NP

Certificate

Given a decision problem $\mathcal{P}$ and a "yes" instance $\mathcal{J}$, a certificate $c(\mathcal{J})$ is a string that encodes a solution and makes it possible to reach the "yes" answer for instance $\mathcal{J}$.

Concise certificate

A certificate is said to be concise if the length of its encoding is polynomial in the amount of memory that is used to encode instance $\mathcal{J}$.

Class NP

A decision problem $\mathcal{P}$ belongs to the class NP if there exists a certificate-checking algorithm $\mathcal{A}'$ such that, for any "yes" instance of $\mathcal{P}$, there is a concise certificate $c(\mathcal{J})$ with the property that algorithm $\mathcal{A}'$ applied to instance $\mathcal{J}$ and certificate $c(\mathcal{J})$ reaches the answer "yes" in polynomial time.

The classes P and NP

For a problem to be in NP, it is not required that there exists an algorithm that computes an answer in polynomial time for every instance of this problem.

All that is required for a problem to be in NP is that there exists a concise certificate for any “yes” instance that can be checked for validity in polynomial time.

The classes P and NP

For a problem to be in NP, it is not required that there exists an algorithm that computes an answer in polynomial time for every instance of this problem.

All that is required for a problem to be in NP is that there exists a concise certificate for any "yes" instance that can be checked for validity in polynomial time.

Maximum clique problem – Concise certificate and membership in NP

- A certificate for the maximum clique problem is an encoding of a list of nodes.
- This certificate is concise, because it cannot have more than $|V|$ nodes.
- The certificate-checking algorithm is polynomial. It starts by checking whether the certificate corresponds to a subset of the nodes of the graph $G = (V, U)$, then verifying if there is an edge in G for every pair of nodes in the certificate. Next, it counts the number of nodes in the certificate, which is compared with the parameter B.
- Therefore, the decision problem CLIQUE belongs to NP.

Knapsack problem – Concise certificate and membership in NP

- A certificate for the knapsack problem is an encoding of a subset of the n available items.
- This certificate is concise, because it cannot have more than n items.
- The certificate-checking algorithm is polynomial. It starts by adding up the weights of the items in the certificate and comparing the total weight with the maximum weight capacity b. Next, it adds up the utilities of the items in the certificate and their total utility is compared with the parameter B.
- Consequently, the decision problem KNAPSACK belongs to NP.

The classes P and NP

Knapsack problem – Concise certificate and membership in NP

• A certificate for the knapsack problem is an encoding of a subset of the n available items.
• This certificate is concise, because it cannot have more than n items.
• The certificate-checking algorithm is polynomial. It starts by adding up the weights of the items in the certificate and comparing the total weight with the maximum weight capacity b. Next, it adds up the utilities of the items in the certificate and their total utility is compared with the parameter B.
• Consequently, the decision problem KNAPSACK belongs to NP.

Traveling salesman problem – Concise certificate and membership in NP

• A certificate for the traveling salesman problem is an encoding of a permutation of the n cities or nodes in the graph $G = (V, U)$.
• This certificate is concise, because it must have exactly $|V|$ nodes.
• The certificate-checking algorithm is polynomial. It starts by checking if every city appears exactly once. Next, it adds up the lengths of the edges defined by the certificate and the total length is compared with the parameter B.
• Therefore, the decision problem TSP also belongs to NP.

The classes P and NP

Examples of other problems in this class:

- STEINER TREE IN GRAPHS
- GRAPH PLANARITY
- GRAPH COLORING
- INTEGER PROGRAMMING
- HAMILTONIAN CYCLE
- SATISFIABILITY

The classes *P* and *NP*

Examples of other problems in this class:

- STEINER TREE IN GRAPHS
- GRAPH PLANARITY
- GRAPH COLORING
- INTEGER PROGRAMMING
- HAMILTONIAN CYCLE
- SATISFIABILITY

To prove that a problem is in *NP*, one is not required to provide an efficient algorithm to compute the certificate:

- One has only to prove the existence of at least one concise certificate for each "yes" instance.
- Nothing is required for the "no" instances: concise certificates should exist only for "yes" instances.
- It works as if an external oracle was able to provide the certificate.
- The acronym *NP* stands for *nondeterministic polynomial*, and not for *nonpolynomial*.

The classes P and NP

Suppose there exists a polynomial-time algorithm $\mathcal{A}$ for solving some decision problem $\mathcal{P}$ in P.

- In other words, algorithm $\mathcal{A}$ is able to provide the appropriate "yes" or "no" answer for every instance of $\mathcal{P}$.
- The steps of algorithm $\mathcal{A}$ applied to any "yes" instance provide a concise certificate for this instance.
- The existence of a concise certificate that can be checked in polynomial time for any "yes" instance shows that $\mathcal{P}$ is also in NP.
- Therefore, whenever a decision problem $\mathcal{P} \in P$, it also holds that $\mathcal{P} \in NP$.

The classes P and NP

Suppose there exists a polynomial-time algorithm $\mathcal{A}$ for solving some decision problem $\mathcal{P}$ in P.

- In other words, algorithm $\mathcal{A}$ is able to provide the appropriate "yes" or "no" answer for every instance of $\mathcal{P}$.
- The steps of algorithm $\mathcal{A}$ applied to any "yes" instance provide a concise certificate for this instance.
- The existence of a concise certificate that can be checked in polynomial time for any "yes" instance shows that $\mathcal{P}$ is also in NP.
- Therefore, whenever a decision problem $\mathcal{P} \in P$, it also holds that $\mathcal{P} \in NP$.

Consequently, $P \subseteq NP$.

Polynomial transformations and *NP*-complete problems

Polynomial-time transformation

Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two decision problems. We say that there is a polynomial-time transformation from problem $\mathcal{P}_1$ to problem $\mathcal{P}_2$ if an instance $\mathcal{J}_2$ of $\mathcal{P}_2$ can be constructed in polynomial time from any instance $\mathcal{J}_1$ of $\mathcal{P}_1$, such that $\mathcal{J}_1$ is a "yes" instance of $\mathcal{P}_1$ if and only if $\mathcal{J}_2$ is a "yes" instance of $\mathcal{P}_2$.

CLIQUE polynomially transforms to INDEPENDENT SET

CLIQUE

Given a graph $G = (V, U)$ and an integer B, is there a clique in G with at least B nodes?

INDEPENDENT SET

Given a graph $G = (V, U)$ and an integer B, is there an independent set of nodes in G (i.e., a subset of mutually nonadjacent nodes) with at least B nodes?

CLIQUE polynomially transforms to INDEPENDENT SET

CLIQUE

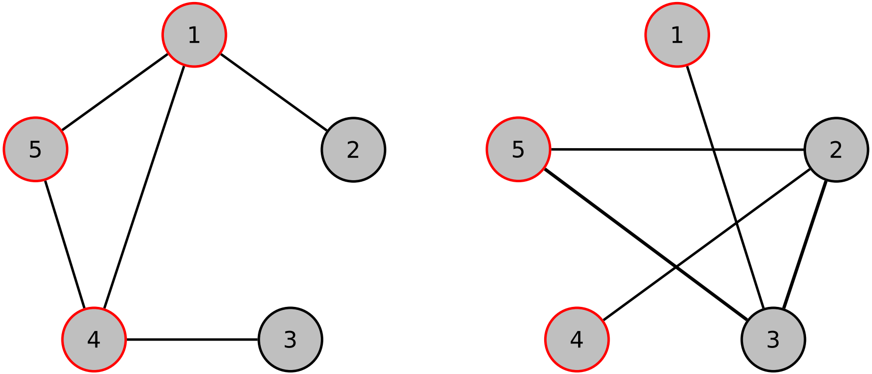
Given a graph $G = (V, U)$ and an integer B, is there a clique in G with at least B nodes?

INDEPENDENT SET

Given a graph $G = (V, U)$ and an integer B, is there an independent set of nodes in G (i.e., a subset of mutually nonadjacent nodes) with at least B nodes?

• Let an instance $\mathcal{J}_1$ of CLIQUE be defined by a graph $G = (V, U)$ and an integer B.
• Let $\bar{G} = (V, \bar{U})$ be the complement of G: for every pair of nodes $i, j \in V$, there is an edge $(i, j) \in \bar{U}$ if and only if the pair i, j does not constitute an edge in U.
• An instance $\mathcal{J}_2$ of INDEPENDENT SET defined by the complement of G and the same integer B can be constructed in time $O(|V|^2)$ such that $\mathcal{J}_1$ is a "yes" instance of CLIQUE if and only if $\mathcal{J}_2$ is a "yes" instance of INDEPENDENT SET.

CLIQUE polynomially transforms to INDEPENDENT SET



(a) Maximum clique in the original graph G (b) Maximum independent set in $\bar{G}$

Figure: Polynomial transformation from CLIQUE to INDEPENDENT SET: Nodes 1, 4, and 5 form a maximum clique of the original graph G in (a), while the same nodes correspond to a maximum independent set of the complement $\bar{G}$ of G in (b). The instances defined by G and $\bar{G}$ are "yes" instances for any $B \leq 3$ and "no" instances for any $B > 3$.

Polynomial transformations and *NP*-complete problems

NP-complete problems

A decision problem $\mathcal{P} \in NP$ is said to be *NP-complete* if every other problem in *NP* can be transformed to it in polynomial time.

If there is a polynomial-time algorithm for any *NP*-complete problem, then there are also polynomial-time algorithms for all other problems in *NP*.

Polynomial transformations and *NP*-complete problems

NP-complete problems

A decision problem $\mathcal{P} \in NP$ is said to be *NP-complete* if every other problem in *NP* can be transformed to it in polynomial time.

If there is a polynomial-time algorithm for any *NP*-complete problem, then there are also polynomial-time algorithms for all other problems in *NP*.

The proof that a problem is *NP*-complete involves two main steps:

1. Proving that it is in *NP*.
2. Showing that all other problems in *NP* can be transformed to it in polynomial time.

The second part is often the hardest and is usually proved by showing that another problem already proved to be *NP*-complete is polynomially transformable to the problem on hand.

Polynomial transformations and *NP*-complete problems

SATISFIABILITY was the first problem to be explicitly proved to be *NP*-complete (Cook, 1971).

Polynomial transformations and *NP*-complete problems

SATISFIABILITY was the first problem to be explicitly proved to be *NP*-complete (Cook, 1971).

Other *NP*-completeness results followed by polynomial transformations originating with SAT:

- 3-SAT (special case of SATISFIABILITY, in which every clause has exactly three variables or their complements)
- KNAPSACK
- CLIQUE
- INDEPENDENT SET
- TSP
- STEINER TREE IN GRAPHS
- INTEGER PROGRAMMING
- HAMILTONIAN CYCLE
- GRAPH COLORING
- GRAPH PLANARITY
- and many others.

Polynomial transformations and *NP*-complete problems

NP-hard problems

A problem $\mathcal{P}$ is *NP-hard* if all problems in *NP* are polynomially transformable to $\mathcal{P}$, but its membership to *NP* cannot be established.

This definition includes not only decision problems that are not proved to be in *NP*, but also refers to the optimization problems whose decision versions are *NP*-complete.

The maximum clique problem, the knapsack problem, and the traveling salesman problem introduced as combinatorial optimization problems are all *NP*-hard, since the decision problems CLIQUE, KNAPSACK, and TSP are *NP*-complete, respectively.

PSPACE and the polynomial hierarchy

Consider the requirements of space or memory that are needed for solving a decision problem:

Class *PSPACE*

A decision problem $\mathcal{P}$ belongs to the class *PSPACE* if there exists an algorithm $\mathcal{A}$ that solves any of its instances using a polynomial amount of space (or memory).

PSPACE and the polynomial hierarchy

Consider the requirements of space or memory that are needed for solving a decision problem:

Class *PSPACE*

A decision problem $\mathcal{P}$ belongs to the class *PSPACE* if there exists an algorithm $\mathcal{A}$ that solves any of its instances using a polynomial amount of space (or memory).

Any polynomial-time algorithm cannot consume more than a polynomial amount of space: $P \subseteq PSPACE$

Since the certificate-checking for problems in *NP* is polynomial: $NP \subseteq PSPACE$

Even problems that take an exponential amount of time to be solved by repeated enumeration make use of polynomial space.

PSPACE and the polynomial hierarchy

Consider the requirements of space or memory that are needed for solving a decision problem:

Class *PSPACE*

A decision problem $\mathcal{P}$ belongs to the class *PSPACE* if there exists an algorithm $\mathcal{A}$ that solves any of its instances using a polynomial amount of space (or memory).

Any polynomial-time algorithm cannot consume more than a polynomial amount of space: $P \subseteq PSPACE$

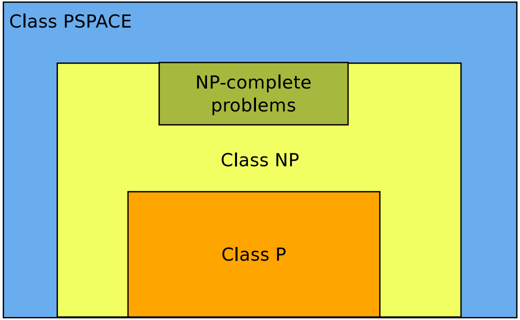
Since the certificate-checking for problems in *NP* is polynomial: $NP \subseteq PSPACE$

Even problems that take an exponential amount of time to be solved by repeated enumeration make use of polynomial space.

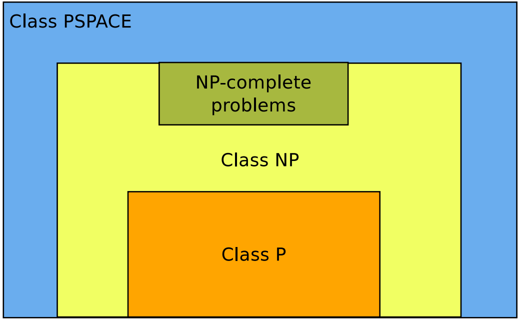
Since polynomiality is considered as a limitation for any scarce resource such as time or space, we can say that time requirements become critical (i.e., superpolynomial) before space does.

Time is the main and critical scarce resource considered in the analysis and design of computer algorithms, which in practice very rarely involve space considerations.

PSPACE and the polynomial hierarchy



PSPACE and the polynomial hierarchy



Open question – P vs. NP:

- $P \subset NP$?
- $P = NP$?

Solution approaches

Most optimization problems of practical relevance are *NP*-hard.

- Being considered as computationally intractable does not preclude the need for their solution.

Solution approaches

Most optimization problems of practical relevance are *NP*-hard.

- Being considered as computationally intractable does not preclude the need for their solution.

Solution approaches to exactly solve or to efficiently find high-quality solutions:

- Super-polynomial exact algorithms: Theoretical developments in polyhedral theory, combined with efficient algorithm design and data structures and advances in computer hardware, have made it possible to solve very large instances of some *NP*-hard problems.
- Parallel processing: Parallel/distributed algorithms and new architectures (clusters, grids, clouds) with a limited number of processors are able to speedup sequential algorithms, but do not change problem complexity.
- Approximation algorithms: Algorithms that build feasible solutions that are not necessarily optimal, but whose objective function value can be shown to be within a guaranteed difference from the exact optimal value (not very useful results in practice).
- Heuristics: A heuristic (or approximate algorithm) is essentially any algorithm that provides a feasible solution for a given problem, without necessarily providing a guarantee of performance in terms of solution quality or computation time.

Heuristics

Heuristic methods can be classified into three main groups:

Heuristics

Heuristic methods can be classified into three main groups:

- Constructive heuristics are those that build a feasible solution from scratch. Greedy and semi-greedy algorithms are examples of constructive heuristics.
- Local search or improvement procedures start from a feasible solution and improve it by successive small modifications until a locally optimal solution is found. They can become prematurely stuck in low-quality locally optimal solutions.
- Metaheuristics are general high-level procedures that coordinate simple heuristics and rules to find good-quality solutions to computationally difficult optimization problems: simulated annealing, tabu search, greedy randomized adaptive search procedures (GRASP), genetic algorithms, scatter search, variable neighborhood search (VNS), ant colonies, and others.

Metaheuristics

Metaheuristics are based on distinct paradigms and offer different mechanisms to escape from locally optimal solutions.

- Trajectory-based: one single solution is progressively improved.
- Population-based: a family of solutions is improved as a whole.

Metaheuristics

Metaheuristics are based on distinct paradigms and offer different mechanisms to escape from locally optimal solutions.

- Trajectory-based: one single solution is progressively improved.
- Population-based: a family of solutions is improved as a whole.

They are among the most effective solution strategies for solving combinatorial optimization problems in practice and very often produce much better solutions than those obtained by the simple heuristics and rules they coordinate.

Metaheuristics have been applied to a wide array of academic and real-world problems.

Metaheuristics

Metaheuristics are based on distinct paradigms and offer different mechanisms to escape from locally optimal solutions.

- Trajectory-based: one single solution is progressively improved.
- Population-based: a family of solutions is improved as a whole.

They are among the most effective solution strategies for solving combinatorial optimization problems in practice and very often produce much better solutions than those obtained by the simple heuristics and rules they coordinate.

Metaheuristics have been applied to a wide array of academic and real-world problems.

Algorithmic research in metaheuristics:

- Solve larger problems
- Solve problems in smaller computation times
- Find better solutions

Concluding remarks

The material in this talk is taken from

- Chapter 2 – A short tour of combinatorial optimization and computational complexity

of our book, *Optimization by GRASP: Greedy Randomized Adaptive Search Procedures* (Resende & Ribeiro, 2016).

and from the book

- C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization*, 1982.

Short video presentation of the *P* vs. *NP* question.