Optimization Methods Lecture 7

Jakub Uher

February 2020

Introduction

In this lecture we discuss the combinatorial optimisation problems. And their complexity classes P and NP.

Combinatorial optimisation problems

- We can view an instance of a combinatorial optimisation problem (COP) as a pair (R, C), where:
  - R is a finite set of configurations, given in some implicit way.
  - $C : R- > R$ is a function over R which assigns a cost $C(r)$ to every configuration $r \epsilon R$
- The most traditional objective of combinatorial optimisation problems is to find configuration $R \epsilon R$ which has the minimum (maximum) cost.

Shortest Path from s to t

- Problem input: A directed graph $G(V, E)$ with non-negative weights $w(x, y) \geq 0$ for all edges $(x, y) \epsilon E$ and two nodes $s, t \epsilon V$
- The set R of all possible configurations is defined as the set of all possible simple paths from s to t in G.
- A configuration $r \epsilon R$ is a simple path from s to t in G.
- The cost function C assigns a cost $C(r)$ to a configuration $r \epsilon R$ bu summing up the weights of the edges.
- Problem Output: A simple path r from s to t such that the $C(r) \leq C(r')$ where $r' \epsilon R$ is any simple path from s to t in G.

Travelling Salesman Problem as COP

- Problem input: A complete undirected graph $G(V, E)$ which edge weights $w(x, y) \geq 0$ for all edges $(x, y)\ \epsilon\ E$ and starting node $s\ \epsilon\ V$.
- Hamiltonian Path: A path that starts from s, visits each node exactly once and returns to s.
- The set R of all possible configurations is defined as the set of all Hamiltonian Paths starting from s.
- A configuration $r\ \epsilon\ R$ is a Hamiltonian Path starting from s.
- The cost function C assigns a cost $C(r)$ to a configuration $r\ \epsilon\ R$ by summing up the weights of the edges in r.

- Problem Output: A Hamiltonian path r such that the cost $C(r) \leq C(r')$ where $r'\ \epsilon\ R$ is any Hamiltonian path in G.

Decision problem

- Informally, A decision problem DP is a computational problem that can be posed as a yes-no question of the input values.
- Partition: Given a set A of $2n$ numbers $a_1, a_2, ..., a_{2n}$ such that $\sum$ of $2n$ starting at index $i = 1$. $a_i = 2B$ determine whether there is a subset A_1 of A such that $|A|$ = n and $\sum\ i\ \epsilon\ A$, a_i = B.
- Can we express a Combinatorial Optimisation Problem as a Decision tree?

Shortest Path from s to t as a DP

- Problem input: A directed graph $G(V, E)$ with non-negative weights $w(x, y) \geq 0$ for all edges $(x, y)\ \epsilon\ E$ and two nodes $s, t\ \epsilon\ V$ and a target cost k.
- Problem Output: YES if there is a simple shortest path from s to t in G such that $C(r) \leq k$ and NO otherwise.
- We can answer this problem in **Polynomial time, using Dijkstra's algorithm**

Travelling Salesman Problem as DP

- Problem input: A complete undirected graph $G(V, E)$ which edge weights $w(x, y) \geq 0$ for all edges $(x, y)\ \epsilon\ E$ and starting node $s\ \epsilon\ V$ and a target cost k.
- Problem output: YES if there is a Hamiltonian Path r starting from s such that $C(r) \leq k$ and NO otherwise.
- We can not answer this problem in **polynomial time, as there is no known algorithm to compute a minimum cost Hamiltonian Path**

A problem is in the class NPC if it is in NP and is as hard as any problem in NP. A problem is NP-hard if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself.

Complexity classes P and NP

Polynomial time Algorithm

- A polynomial time algorithm: The worst case running time of the algorithm is $O(n^k)$ where k is a constant number and n is the size of the input.
- Example: Bellman Ford Algorithm has worst case running time $O(n^3)$ for a graph with n nodes.
- Example: Merge Sort has a worst case running time $O(nlogn)$ to sort an array of n numbers.

Exponential Time Algorithm

- A exponential time algorithm: The worst case running time of the algorithm is $o(2^d n)$ and $\Omega(2^c n)$ for some $0 < c \leq d$ where n is the size of the input.
- Example: An algorithm performing a simple (linear time) check for each subset of n elements has an exponential running time of $\Theta(n2^n)$

Complexity class P

- The class of all computation problems that are solvable in polynomial time (that is, there is an polynomial time algorithm which solves the problem)
- Example of problems in class P: $MinimumSpanningTree$, $Shortest-Paths$, $MinimumCostFlow$
- Exercise: Show that the problem of solving a 3x3 Rubik's Cube with the minimum number of moves can be solved by an algorithm which finds a simple shortest path on a graph with non-negative weights.

Complexity class NP

- The class of all computational problems such that given an input instance I, a configuration $r \epsilon R$ and a target cost k, we can check in polynomial time (in the size of the input instance I)
  - If r is a valid configuration.
  - If $C(r) \leq k$.
- The fact that we can check in polynomial time whether a configuration r has cost $C(r)$ less or equal to k does not imply that there is an polynomial algorithm which finds a configuration r satisfying $C(r) \leq k$.
- Exercise: Show that the Shortest Path Problem and the Travelling Salesman Problem are in class NP. Simple shortest path, check if there is a cycle (polynomial time), check all edges sum the weights and check if the sum is lower than k.
- Feasible path, is a path from s to t with no cycle.
- Any solution, doesn't have to be optimal, only feasible. Checking two things, for any solution, check if the constraints of the problem is satisfied and if the cost of the solution is smaller or equal to K.
- There exists algorithm to check the solution but not to come up with an optimal one.

Polynomial Time Reduction

- Consider a computational problem A, for which we do not know the complexity class it belongs to.
- We can reduce the problem into known problem. Like Vertex to 3SAT.
- Consider a computational problem B for which we do know the complexity class it belongs to.
- Polynomial Time Reduction is used to determine the complexity of a problem A with respect to the complexity of a problem B.
- Informally, a problem A can be reduced to another problem B if
  - Any instance of A can be modeled as an instance of B.
  - A solution to problem B provides a solution to a problem A and vice versa.
- Intuitively, this means that problem A is at most as hard as B and that B is at least as hard as A.

NP-Hard Problems

- Among all problems in NP there are some problems that are considered as the $Kings$ of the complexity class NP, these problems are called $NP\text{-}Hard$
- $NP\text{-}Hard$ problem is a problem that is as hard as the hardest problem in NP, so looking at all NP problems we pick the hardest one to solve, and our problem in order to be in class $NP\text{-}Hard$ must be as hard as the hardest one.
- $NP\text{-}Hard$ problems do NOT necessarily belong to NP.
- Why? Is it possible that given a configuration r for $NP\text{-}Hard$ problem and a target cost of k, we cannot check in polynomial time whether $C(r) \leq k$.

NP-Complete Problems

- NP-Complete: A **decision** problems which is in NP and is also $NP\text{-}Hard$.
- Each NP-Complete problem has to be in NP.
- What does this mean?
  - Any problem in NP can be reduced to a $NP - Complete$ problem in polynomial time.
  - This implies that the set $NP\text{-}C$ of NP-Complete problems is the set of the hardest problems in NP.
- If we can solve a NP-Complete problem in polynomial time:
  - We come the best Computer Scientists of this era.
  - Any NP-Complete problem can be solved in polynomial time by reducing the problem into the solvable one.

NP-Complete vs NP-Hard

- Putting everything together, a NP-Complete problem implies it being NP-Hard, but a NP-Hard problem does NOT imply it being NP-Complete.
- Consider the following problem Over a chess game between two players A and B.
- Given the current state of the chess board, determine if there is a sequence of at most k moves, which guarantees that player A win.
- Lets say someone provides u with a sequence of these moves, how do you check? for each move there are exponentially many i can make. Check if the move is valid configuration we have to consider all possible moves after the one given, but there are exponentially many we need exponential time to check feasibility therefore it cant be solved in polynomial time.

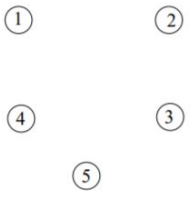
Solving NP-hard problems

- **Exhaustive search**: check all configurations, find optimal solution but always requires exponential time, **Very slow**.
- **Branch and Bound** method:improved exhaustive search, tries to avoid checking configurations which cannot be optimal solutions. Always finds an optimal solution. Requires exponential time in the worst case; an average computational time may be much better than the computational time of the exhaustive search. **Slow**.
- **Approximation algorithms** compute "good" solutions and give some guarantee how good the computed solutions are (give a bound on the difference between the cost of the computed solutions and the optimal cost) **Fast (polynomial)**
- **Heuristics**: find good solutions, but do not give any guarantee how good the computed solutions are. **Fast**
- **Meta-heuristics**: general heuristic techniques which can be applied to many different problems. For example simulated annealing and genetic algorithms.

Branch-and-bound method

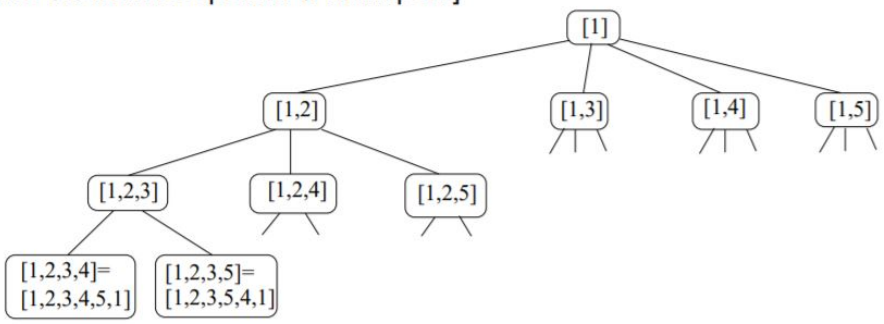
- Example TSP
- Consider the following input instance:

	1	2	3	4	5
1	0	14	4	10	20
2	14	0	7	8	7
3	4	5	0	7	16
4	11	7	9	0	2
5	18	7	17	4	0



Branch-and-bound method (cont)

[Figure 6.5 from Neapolitan & Naimipour]



Each leaf in this tree corresponds to one tour (one complete configuration).
The number of leaves for a n-city instance of TSP:

$$(n-1)(n-2)\cdots 2 = (n-1)!$$

For example: $4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24, \quad 15! \approx 1.3 \cdot 10^{12}.$

Partial configurations and branching procedure

- A **Partial configuration**: a partially specified configuration, which can be extended to a number of complete configurations. Expanding the configuration when branching.
- The **Initial partial configuration**: can be extended to any configuration. Example: Example starting from root.
- A **branching procedure**: takes as input a partial configuration P (which is not a complete configuration) and produces partial configurations $P_1, P_2, ..., P_k$ which extend P.

Each complete configuration which can be obtained from the partial configuration P, can be obtained from exactly one of the partial configurations $P_1, P_2, ..., P_k$

Some of the partial configurations $P_1, P_2, ..., P_k$ (or even all of them) can be a complete configuration. Example a tree with one root node and four leafs representing a **complete configuration**.

Partial configurations and branching procedure: example

For TSP with cities 1,2, ..., n

- **Partial configuration**: a sequence of distinct cities starting with city 1. Formally, a sequence of integers $< i_1, i_2, ..., i_k >$ such that:

  $$1 \leq k \leq n-1; 1 \leq i_j \leq n$$

  for each $j = 1, 2..., k$ $i_1 = 1$; and all $i_1, i_2, ..., i_k$ are distinct. For example $< 1, 3, 5, 7 >$.

- **Initial partial configuration**: $< 1 >$.

- **Branching procedure**: for a partial incomplete configuration

  $$< i_1, i_2, i_3, ..., i_k >$$

  where $1 \leq k \leq n-2$, output for all partial configurations

  $$< i_1, i_2, i_3, ..., i_k, x >$$

  where $1 \leq x \leq n$ and x is different than any of $i_1, i_2, i_3, ..., i_k$.

  For example, if $n = 6$ and input is $< 1, 3, 4 >$, then the output is: $< 1, 3, 4, 2 >$, $< 1, 3, 4, 5 >$, $< 1, 3, 4, 6 >$

Partial configurations and bounding procedure

- A **bounding procedure** computes for a given partial configuration P, a **lower bound** on the cost of any configuration which can be obtained by extending configuration P.

- An example of a bounding procedure for TSP.

  If the input is the initial partial configuration $< 1 >$, then sum, over all cities, the minimum cost of a link outgoing from each city:

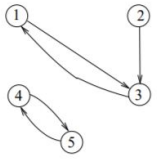
  $\sum_{i=1}^{n}$ min{ $c(i, x)$: $x \in$ { 1,2,...,n } - {i}}

  This gives a lower bound on the cost of any TSP tour; in particular, a lower bound on the minimum cost of a TSP tour.

Partial configurations and bounding procedure: example

In out example, this bounding procedure selects the following links:

	1	2	3	4	5
1	0	14	4	10	20
2	14	0	7	8	7
3	4	5	0	7	16
4	11	7	9	0	2
5	18	7	17	4	0

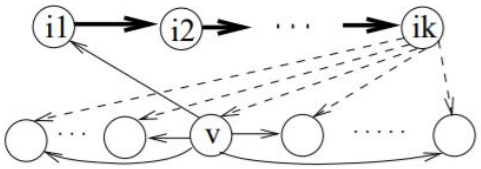


and computes the following lower bound on any TSP tour:

$c(1,3) + c(2,3) + c(3,1) + c(4,5) + c(5,4)$ =

$= 4 + 7 + 4 + 2 + 4 = 21 \leq$ [min cost of a TSP tour]

Partial configurations and bounding procedure: example (cont)

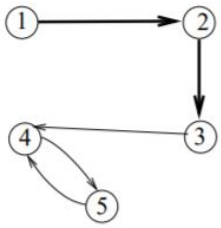


If the input is a partial tour $\langle i_1, i_2, \ldots, i_k \rangle$, $k \leq n-2$, then the bounding procedure sums together:

$c(i_1, i_2) + c(i_2, i_3) + \cdots + c(i_{k-1}, i_k)$,
$\min\{c(i_k, x) : x \notin \{i_1, i_2, \ldots, i_k\}\}$, and
$\min\{c(v, x) : x \notin \{i_2, \ldots, i_k, v\}\}$, for each city $v \notin \{i_1, i_2, \ldots, i_k\}$.

For our example, this bounding procedure would compute the following bound for the partial configuration $\langle 1, 2, 3 \rangle$:

$c(1,2) + c(2,3) + c(3,4) + c(4,5) + c(5,4)$
$= 14 + 7 + 7 + 2 + 4 = 34.$



Branch-and-bound method: general description

```
BRANCH_AND_BOUND   { the general Branch-and-bound method }
  opt_config ← not defined;  min_cost_seen ← ∞;
  Q: a priority queue holding pairs ⟨Partial_config, lower_bound⟩;
  P ← initial partial configuration;  lower_bound ← BOUNDPROC(P);
  INSERT(Q, ⟨P, lower_bound⟩);
  while Q is not empty do
    ⟨P, lower_bound⟩ ← EXTRACT_MIN(Q);
    if lower_bound ≥ min_cost_seen then break the loop else
      P_1, ..., P_k ← extensions of P generated by BRANCHPROC(P);
      for each P_i do
        if P_i is a complete configuration then
          if cost(P_i) < min_cost_seen then
            min_cost_seen ← cost(P_i);  opt_config ← P_i;
        else { P_i is not a complete configuration }
          lower_bound ← BOUNDPROC(P_i);
          if lower_bound < min_cost_seen then
            INSERT(Q, ⟨P_i, lower_bound⟩)
  return opt_config and min_cost_seen.
```

Branch-and-bound method for the weighted graph-bisection problem

Let $(G = (V, E), e)$ be an input to the weighted graph bisection problem, and let $V = 1, 2, ..., 2_n$

- **Partial configuration**:
  (L, R), where
  $L \subseteq V$; $R \subseteq V$; $L \cap R = \emptyset$; $|L| \leq n$; $|R| \leq n$; $L \cup R = \{1, 2, \ldots, k\}$, for some k, $1 \leq k \leq 2n$; and $1 \in L$.

  Initial partial configuration: $(\{1\}, \{\})$

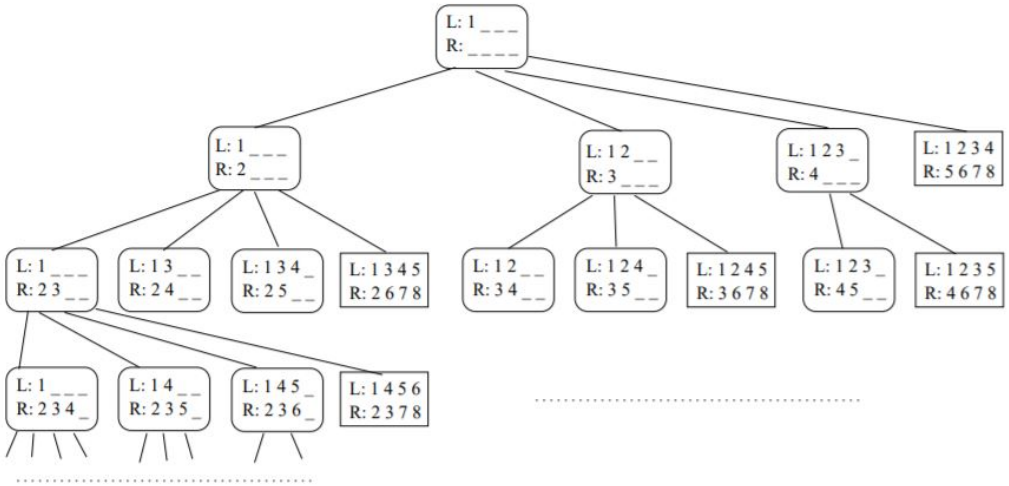
- **Branching procedure**:
  At a partial incomplete configuration (L, R), branch on the next possible element of R.

  If $L \cup R = \{1, 2, \ldots, k\}$, $|L| = l \leq n - 1$ and $|R| = r \leq n - 1$ (so $l + r = k$), then the next element of R can be one of $k + 1, k + 2, \ldots, n + r + 1$.

  If the next element of R is k', then this gives the partial configuration $(L \cup \{k + 1, \ldots, k' - 1\}, R \cup \{k'\})$.

Branching procedure for WGBP

If the input graph has 8 nodes, then the above branching procedure gives the following search tree:



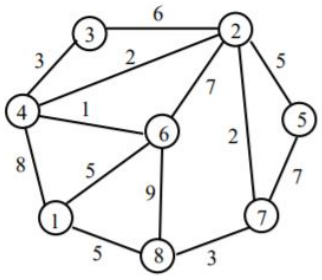
Branching procedure for WGBP (cont)

- A bound for a partial incomplete configuration (L, R) can be obtained by "relaxing" (discarding) the condition that the nodes in a complete configuration must be partitioned evenly.
- Compute the minimum capacity of a cut separating s and t in the undirected network G' obtained from network G by collapsing all nodes of set L into one node s and all nodes of set R into one node t.

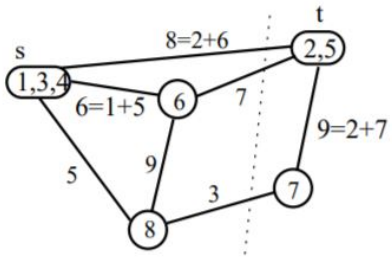
  This also works if both L and R are non-empty.
- The capacities of the edges in G' are the weights of the edges in G. Each set of parallel edges is replaced with one edge. The capacity of the new edge is equal to the sum of the capacities of the replaced edges.
- There are polynomial-time algorithms for computing the minimum capacity $s - t$ cut. (A polynomial-time algorithm for computing a maximum $s - t$ flow can be used).

Branching procedure for WGBP: example

Input graph:



Computing a lower bound for the partial configuration $(\{1, 3, 4\}, \{2, 5\})$



- The bound is 8 + 7 + 3 = 18
- The minimum capacity cut: ({[1,3,4],6,8}, {—2,5—, 7})

Notes produced from lecture slides @KCL given by Angelos Gkikas.