Faculty of Electrical Engineering and Computing, Department of Applied Computing

M. Brčić, D. Krleža

Advanced Algorithms and Data Structures

Lecture script

PUBLISHED BY FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING, DEPARTMENT OF APPLIED COMPUTING

First printing, January 2022

Contents

Introduction

The page is intentionally left blank

12 Approximation algorithms

12.1 Basics

Often we have to solve NP-hard discrete optimization problems and we know that means exponential complexity if aiming for exact optimum (assuming $P \neq NP$).

Sure, we can approximately solve such problems with heuristics and metaheuristics in poly-time. However, the solutions gotten from such rule-of-thumb procedures don't have guarantees on the sub-optimality gap (i.e. worst-case distance from the optimum).

Some problems allow for poly-time **approximation algorithms**, which have strict guarantees on a sub-optimality gap in a form of a constant factor.

In this short chapter, we shall see several examples of problems that allow for such algorithms. We shall also mention a few examples of problems that have no such property.

These methods usually rely heavily on algorithmic toolkit that includes:

- greedy algorithms
- local search
- dynamic programming
- randomization
- rounding fractional values
  - basic
  - adaptive
  - randomized
- convex optimization (e.g. linear programming, semidefinite programming etc.)
- reductions

Elements from this toolkit are combined in elaborate ways in order to design approximation algorithms. The performance analysis of these algorithms also consists of certain leitmotifs.

Definition 12.1. (**α-approximation algorithm**) An α-approximation algorithm for optimization is a polynomial-time algorithm that in the worst-case produces a solution with a value within a factor of α of the optimal value.

In the above definition, α is called performance guarantee, approximation ratio, or approximation factor.

We shall follow convention from Williamson and Shmoys [2011] that $\alpha > 1$ for minimization problems (as we approach optimum from above) and $\alpha < 1$ for maximization problems (since we approach optimum from below).

For some problems, there are even approximation meta-algorithms that provide a way to construct approximation algorithms that can come arbitrarily close to optimum.

Definition 12.2. (**Polynomial-time approximation scheme (PTAS)**) A polynomial-time approximation scheme (PTAS) is a family of algoritms $\{A_\epsilon\}$, where for each $\epsilon > 0$ there is an algorithm such that A_ϵ is an $(1+\epsilon)$-approximation algorithm (for minimization problems) or $(1-\epsilon)$-approximation algorithm (for maximization problems).

However, PTAS is efficient with the respect to the input problem, but possibly not with respect to the $1/\epsilon$. For that reason, there is an even more restrictive family of approximation meta-algorithms.

Definition 12.3. (**Fully polynomial-time approximation scheme (FPTAS)**) A fully polynomial-time approximation scheme (FPTAS) is a PTAS such that running time of A_ϵ is bounded by polynomial in $1/\epsilon$.

12.2 *Examples*

We shall present several examples that permit approximation algorithms and we will demonstrate the construction of such algorithms and proofs involved in showing performance guarantees of such algorithms.

Vertex cover

Vertex cover is a simple combinatorial optimization problem that is excellent for the introduction. It is NP-hard, but it allows for a simple factor 2 approximation algorithm that uses linear programming and simple deterministic rounding of fractional values.

Definition 12.4. (Vertex cover) We are given an undirected graph $G = (V, E)$ with a vertex cost-function $c : V \to \mathbb{R}^+$. Vertex cover is a subset $V' \subseteq V$ such that for every edge in G at least one of the vertices is within V'. Minimum cost vertex cover is such a set with a minimal sum of vertex costs. A special case of minimum cost vertex cover problem is in the case of unit-cost vertex function. Such a problem is called *cardinality vertex cover problem*.

Minimal cost vertex cover problem can be expressed as integer linear program (ILP). Let A be incidence matrix. Also, let the decision variables be $x_i \in \{0,1\}$ for each $i \in V$. These variables are "selectors" of vertices that make up the vertex cover:

$$x_i = \begin{cases} 0 & \text{if } i \notin V' \\ 1 & \text{if } i \in V' \end{cases}$$

Then, the ILP for vertex cover problem is:

$$\begin{aligned} \min \quad & c^T x \\ \text{subject to} \quad & A\,x \geq 1 \\ & x \in \{0,1\}^{|V|} \end{aligned} \tag{12.1}$$

The constraint says that the sum of selector variables for each edge must be greater than one. That is, at least one incident vertex must be selected in order to consider the edge covered.

This ILP is NP-hard and lets us assume its optimum is x^*_{ILP}. However, the given problem is **"almost"** a linear program that is efficiently solvable! Let us relax the problem (12.1) by changing the (binary-)integrality constraint with continuous range.

$$\begin{aligned} \min \quad & c^T x \\ \text{subject to} \quad & A\,x \geq 1 \\ & x \in [0,1]^{|V|} \end{aligned} \tag{12.2}$$

This problem can be solved efficiently to optimality, whereby we get optimal solution vector x^*. But, our selector variables x^*_i may be "soft", that is they can be fractional, "fuzzy" selectors of cover membership. Since LP relaxation is less constrained than ILP and it contains all the feasible points of ILP, it follows that $c^T x^* \leq c^T x^*_{ILP}$.

Let us define another variable by deterministic rounding of LP solutions:

$$z_i = \begin{cases} 1 & \text{if } x^*_i \geq 0.5 \\ 0 & \textit{otherwise} \end{cases} \tag{12.3}$$

Obviously our algorithm **DetRoundLP** looks like this:

Algorithm 1 2-approximation algorithm for vertex cover

Input: ILP: integer linear program for vertex cover
Output: worst-case 2-approximate vertex cover

```
1: function DetRoundLP($ILP$)
2:     Solve the LP relaxation of $ILP$ to get solution $x^*$
3:     $z \leftarrow round(x^*)$
4:     $V' \leftarrow \{i \in V | z_i = 1\}$
       return $V'$
```

We need to answer three questions regarding our proposed algorithm:

- Correctness - does it generate feasible solutions?
- Efficiency - does the algorithm run in poly-time?
- Quality - are there strict guarantees of closeness to the optimum?

Lemma 12.1. Algorithm 1 produces correct solutions to vertex cover problem.

Proof. For each edge sum of their respective two selector variables is 1. That means at least one of the selector variables is ≥ 0.5. Hence, rounding will for each edge select at least one incident vertex. $\square$

Solving LP can be done in poly-time and rounding procedure is in $O(V)$, hence Algorithm 1 runs in polynomial time.

Lemma 12.2. Algorithm 1 produces factor 2 solution.

The lemma 12.2 states that $c^T z \leq 2c^T x^*_{ILP}$.

Proof. From the rounding procedure, it follows that $z_i \leq 2x^*_i$, rounding property.

From the facts that:

- solution to LP is lower bound on the solution to ILP
- rounding property

follows:

$c^T x^* \leq c^T x^*_{ILP} \leq c^T z \leq 2c^T x^* \leq 2c^T x^*_{ILP}$.

The last element of inequality establishes factor 2 upper bound on solution quality of DetRoundLP. $\square$

Is factor 2 strict for DetRoundLP? Yes, there are problems for which DetRoundLP achieves the upper bound performance.

Metric traveling salesman problem

The metric traveling salesman problem is TSP in which triangle inequality holds between the cities. Let OPT be the optimal traversal in MTSP and z^* its length.

Both TSP and metric-TSP are NP-hard. Here we shall produce a 2-approximation algorithm. There are better-performing algorithms, but they are more complex (Christofides' 1.5-approximation algorithm and improvements).

Let $d : V \times V \to \mathbb{R}$ be the distance function that satisfies triangle inequality for the set of cities V.

The 2-approximation algorithm (2-MST):

Algorithm 2 2-MST heuristic for metric TSP

```
Input: $G = (V, V \times V)$: input graph
Output: worst-case 2-approximate MTSP traversal
1: function 2-MST($G$)
2:     Find minimal spanning tree (MST) in $G$.
3:     $x \leftarrow$ weight of the MST
4:     Do the depth-first search of MST passing through every edge
       twice. The traversed tour is $2x$.
5:     $L \leftarrow$ list of vertices from the above traversal
6:     $FL \leftarrow$ filtered L by keeping only first appearance of each ver-
       tex.
7:     $z \leftarrow$ length of traversal defined by FL
       return $FL$
```

Let us examine the 2-MST algorithm.

- Correctness - FL contains each vertex once and if we interpret it as a circular list (ends connect), we get the valid traversal.
- Efficiency - MST can be found in polynomial time. DFS double-traversal and filtering are in $O(V)$.
- Quality - The following lemmas address this question

Lemma 12.3. For the weight x of MST it holds $x \leq z$.

Proof. Proof by contradiction. Assume $z < x$. OPT is a cycle. If we remove an edge from OPT it becomes a spanning tree T (acyclic + passing through all vertices). Therefore $c(T) < z < x$, which contradicts the assumption that x is the weight of a minimal spanning tree. □

Lemma 12.4. 2-MST is 2-approximation algorithm.

Proof. $z \leq 2x$ follows from the triangle inequality that holds in MTSP. During filtering, deletion of every vertex in L amounts to a substitution of two edges in traversal by one direct (those three edges make up a triangle). Hence, the filtering operation does not increase the length of traversal.

Using lemma 12.3, and substituting, we get: $x \leq 2z \leq 2x$. □

0-1 Knapsack

We have already introduced the knapsack problem in the dynamic programming chapter. It is NP-hard problem and dynamic programming is a pseudo-polynomial algorithm. Here, we shall find a polynomial-time approximation algorithm, and in the end, even PTAS.

As a reminder, in knapsack we are given a set of items $I = \{1, ..., n\}$, where each item has size $s_i \in \mathbb{N}$ and value $v_i \in \mathbb{N}$. The knapsack has capacity $B \in \mathbb{N}$. The goal is to find the subset of items $S \subseteq I$

that satisfies capacity constraint ($\sum_{i \in S} v_i \leq B$) and maximizes sum of values of items ($\sum_{i \in S} v_i$).

We have seen how dynamic programming can be used to solve knapsack problems, but that approach is pseudo-polynomial complexity $O(nB)$. The reason is that size of the input B is $log_2 B$, hence that complexity is exponential in the size of one of the inputs.

Let us introduce *unary* encoding, where number 5 is encoded by 5 concatenated ones (11111).

Definition 12.5. An algorithm is pseudo-polynomial if its running time is polynomial in the size of input when the numeric part is encoded in unary (as opposed to binary).

Notice! If the numeric parameters in the knapsack problem are polynomial in n, then we can get a polynomial-time algorithm. That is the basis of $FPTAS$ that we are going to construct: aggregation of numeric input parameters by rounding into a number of buckets dependent on n. Unfortunately, numeric parameters into the problem are given and are not under our control. The procedure we will develop will aim to substitute input parameters with some parameters under our control that we can tune.

However, that approach will not work for (items, capacities) DP table! Work out why, by following the procedure described below. Instead, we shall exchange the "axes" of the DP table. We shall use (items, values) table and cell at $A[i, v]$ has the meaning as the minimal size achievable for the subset of first i items of value v. Let N be the maximal value among the items.

The DP algorithm would in that case look like following:

Algorithm 3 DP-by-value algorithm for 0-1 knapsack

```
Input: v,s: items data, B: knapsack capacity
Output: decoded data
1: function DPKnapsackByValue($v, s, B$)
2:     $N \leftarrow max_i v_i$
3:     $A[:, 1] \leftarrow B + 1$
4:     $A[0, 1] \leftarrow 0, A[v_1, 1] \leftarrow min(A[v_1, 1], s_1)$
5:     for each $i \in [2, n]$ do
6:         $A[:, i] \leftarrow A[:, i-1]$
7:         for each $v \in [v_i, ..., n \cdot N]$ do
8:             $A[v, i] \leftarrow min(A[v, i-1], A[v - v_i, i-1] + s_i)$
       return $max\{v : A[v, n] \leq B\}$
```

Obviously, Algorithm 3 is in $O(n^2 N)$, again pseudo-polynomial (due to N, this time, instead of B). If N would be polynomial in n, then the running time would be polynomial in the input size. The approximation scheme is based on this fact: we will make it so! Let us coarse-grain (or bucketize) the values of items in order to bound the number of buckets polynomially with n. Let us find a unit-measure μ in which we shall measure the values (with rounding-down to closest integer). If we make $\frac{N}{\mu}$ dependent on n, we shall achieve a polynomial-time algorithm.

What about the error we are introducing, can it be bound within some constant factor? The upper bound on rounding-error of each element is μ itself, hence through rounding all the items we make an error of at most $n\mu$ in our final solution. Assuming a specific instance of a knapsack problem with optimal solution value OPT, we get the relative error to be $\epsilon = \frac{n\mu}{OPT}$.

The lower bound on the optimal solution in all knapsack problems is the value of its most valuable item N. That lower bound is tight, as we can construct problems where it is an optimal solution. The upper bound on the optimal solution is simply nN. For all these cases of the optimal solution $[N, nN]$ we must not have an error beyond some constant factor ϵ.

For the range of optimal values, the absolute error always has a greater relative share in problems with optimal solutions closer to the lower part of the range. For such problems, we must look to tame (bound) the error. Constructing the worst case we get: maximal error $n\mu$ for problem with optimal solution N, that is error $\epsilon = \frac{n\mu}{N}$. All other problems with identical N will have lower relative error since the denominator increases. From that, it follows that if we commit to a specific relative error ϵ, we know to select the right bucket size:

$$\mu = \frac{\epsilon N}{n} \tag{12.4}$$

Hence, we propose the following approximation scheme for knapsack problem:

Algorithm 4 Bucketized DP for 0-1 knapsack

Input: v, s: items data, B: knapsack capacity, ϵ: error parameter
Output: worst-case $(1-\epsilon)$-approximate solution

```
1: function BucketizedDP($v, s, B, \epsilon$)
2:     $N \leftarrow max_i v_i$
3:     $\mu \leftarrow \frac{\epsilon N}{n}$
4:     $v'_i \leftarrow \lfloor \frac{v_i}{\mu} \rfloor, \forall i \in I$
       return DPKnapsackByValue($v', s, B$)
```

As always, let us analyze the 3 aspects of approximation algorithm: correctness, efficiency, quality.

It generates **correct** selection, due to the definition of bucketized DP that selects a solution that satisfies the capacity constraint. The elements that make up that solution can be recovered from the DP table with a value of at least $v^*\mu$.

Regarding the **efficiency**, after scaling N is transformed to $\frac{n}{\epsilon}$. Hence the complexity of bucketized DP becomes $O(n^3/\epsilon)$, which is polynomial in n and $epsilon$.

The question of quality is settled by the following lemma:

Lemma 12.5. Algorithm 4 is FPTAS for the 0-1 knapsack problem.

Proof. The proof is based on the way we constructed bucket sizing.

We need to show that returned solution is with a value at least $(1-\epsilon)$ of optimal value.

Let S be the set of items returned by the algorithm. Let O be the optimal set of items.

$$\begin{aligned}
\textstyle\sum_{i\in S} v_i &\geq \mu \textstyle\sum_{i\in S} v'_i \\
&\geq \mu \textstyle\sum_{i\in O} v'_i \\
&\geq \textstyle\sum_{i\in O} v_i - \mu|O| \\
&\geq \textstyle\sum_{i\in O} v_i - \mu n \\
&= \textstyle\sum_{i\in O} v_i - \epsilon M \\
&\geq OPT - \epsilon OPT = (1-\epsilon)OPT
\end{aligned}$$

The inequality chain follows from:

1. introduced scaling error
2. O possibly not being optimal under bucketized DP
3. using maximal possible rounding error for elements in O
4. $|O| \leq n$
5. using bucket sizing formula
6. $M \leq OPT$ (using lower end for range of optimal values)

□

12.3 *Conclusion*

What we have seen is the compositionality of already known algorithmic blocks into algorithmic designs that make-up fast approximation algorithms with strict performance guarantees.

The common used blocks are

- greedy algorithms
- local search
- dynamic programming
- randomization
- rounding fractional values
  - basic
  - adaptive
  - randomized
- convex optimization (e.g. linear programming, semidefinite programming etc.)
- reductions (see chapter ...)

In this short chapter, we have covered the following blocks: greedy algorithms(MTSP), local search (MTSP), linear programming (vertex cover), basic rounding (vertex cover), dynamic programming (knapsack), adaptive rounding (knapsack).

The main approaches presented here include:

- find superoptimal (better than optimal, but infeasible) solution (LP relaxation for vertex cover, 2*MST for MTSP)
- construct a feasible solution from infeasible through a corrective procedure by losing as least as possible solution quality in the process
  - clever rounding of solutions that may be fractional (LP outputs in vertex cover and problem inputs in knapsack problem)
  - use greedy procedure using problem properties (DFS with filtering in MTSP)
- simplify problematic inputs to control complexity (scaling in a knapsack)

The proofs for guarantees of performance always followed the used procedure, obviously. They were based on using the scaling of superoptimal solution through a corrective procedure to bound the final solution within some factor of optimal quality (for vertex cover and MTSP). And for knapsack, they were based on bounding relative error on problem instances with smaller optimal values by using absolute error.

Bibliography

David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms.* Cambridge University Press, 2011. ISBN 978-0-521-19527-0.