Approximation Algorithms

Florent Bouchez Tichadou & Nicolas Fournel *

October 23, 2015

Objectives:

- Give acceptable solutions to NP-hard problems.

1 Course

1. It usually takes exponential time to find the optimal solution of NP-hard problems.
2. In many cases it is acceptable to obtain a sub-optimal solution if not too far from the optimal.
3. Some algorithms have a guarantee on the quality of the solutions they provide, for instance, if the solution is never more than α times the optimal, the algorithm is an α-approximation.
4. To find an approximation ratio you first need to find a bound on the value of the optimal solution.
5. Before starting a proof it is best to first find the worst case example.

1.1 NP-hardness

This is not the subject here, but you need basic knowledge about NP-hard problems in order to understand approximations algorithms. NP-hard problems are very difficult to solve optimally, unless using exponential-time algorithms. Actually, nobody knows whether if it is possible to find an optimal polynomial-time algorithm for NP-hard problems; So you can safely assume it is just impossible for you (and me, and everyone on this world).

Therefore, polynomial-time algorithms solving NP-hard problems are never optimal. However, they can sometimes give *guarantees* about the quality of the solution.

1.2 Approximation Algorithms

Some algorithms have a guarantee on the quality of the solutions they provide: if for all input $\mathcal{I}$, the value of the solution s given by algorithm A is within α times the value of s^* the optimal solution, then A is said to be an approximation algorithm of approximation ratio α, i.e., an α-approximation.

In principle, consider an algorithm A and an input I. Let C be the value of s the solution of $A(I)$, and $C*$ be the value of $s*$ an optimal solution for I. Then, if $\mathcal{I}$ is the set of all possible inputs, A is an α-approximation if and only if:

$$\begin{cases} \forall I \in \mathcal{I}, \frac{C}{C*} \leqslant \alpha \text{ for a minimization problem.} \\ \\ \forall I \in \mathcal{I}, \frac{C*}{C} \leqslant \alpha \text{ for a maximization problem.} \end{cases}$$

*Based on previous work by Mohammed Slim Bouguerra and Frédéric Wagner.

In practice, we don't know the value of the optimal solution. So the goal is to find a bound of the optimal (a lower bound for minimization problems, and an upper bound for maximization ones), and then show that our solution is withing α times this bound.

For instance, for a minimization problem, suppose you have a lower bound B on the cost of the optimal solution:

$$B \leqslant C^*$$

You then only need to show the following is always true:

$$C \leqslant \alpha \times B.$$

Not an approximation algorithm On the other end, proving that an algorithm is *not* an α-approximation is "easier" as it is sufficient to produce a counter example, i.e., an input I for which you show that the value of the solution produced C is more than $\alpha \times C^*$ (in fact, it doesn't even have to be the optimal C^*, just another solution). It is usually best to provide a *family* of counter-examples, i.e., a sequence of inputs I_n parametrized by n, with the size of I_n growing with n (this shows it is not just a problem with "small" instances).

Example

We consider the *Minimum Vertex Cover* problem: given a graph $G = (V, E)$ we want to find $C \subset V$ of smallest size such that $\forall (i, j) \in E$, either $i \in C$ or $j \in C$.

Vertex Cover is an NP-hard problem and we are therefore unable to find the optimal solution in a reasonable amount of time. We would therefore like to find some algorithm returning quickly a vertex cover even if the cover obtained is not of minimal size.

We propose the greedy algorithm on the right.

Data: a graph $G = (V, E)$
Result: $C \subset V$, a vertex cover
1 $C = \emptyset$;
2 **while** E is not empty **do**
3 pick $(i, j) \in E$;
4 remove all edges incident to i or j from E;
5 add i and j to C;
6 **return** C;

We want to prove this is an approximation algorithm.

Clearly, it cannot be better than a 2-approximation, as if we apply it on a graph with only two nodes linked by an edge, it will add both of them to C, hence providing a solution of size 2, while the optimal is only 1. More generally, we can construct a *family* of examples of size $2n$ as a collection of n independent edges for which the algorithm always give a solution twice worst as the optimal:

It seems hard to find worse examples, so we will try to prove it is a 2-approximation:

Proof. Let us consider S the set of ***edges*** selected during the execution of the greedy algorithm, and suppose we know C^* an optimal solution (i.e., a minimal cover).

All edges of S need to be covered by the vertices in C^*; Moreover, we know by construction of S that no common vertex is shared by the edges of S (i.e., there is no two edges (i, j) and (i, k) in S). Therefore, for each edge of S, at least one of its endpoints is in C^*. We deduce from this property a lower bound on the optimal: $|S| \leqslant |C^*|$.

Moreover, the solution C returned by the greedy algorithm contains all endpoints of edges of S and therefore the number of vertices in C is equal to $2 \times |S|$.

We can now conclude since:

$$|C| = 2 \times |S| \leqslant 2 \times |C^*|$$

So $\alpha = \frac{|C|}{|C^*|} \leqslant 2$ and the greedy algorithm is a 2-approximation. □

Exercises

Independent Set

We consider the *independent set* problem where given a graph $G = (V, E)$ we want to obtain a set S of vertices such that $\forall i$ and $j \in S, (i, j) \notin E$ and $|S|$ is maximal.

We consider here the greedy algorithm of Figure 1.

1. Find a family of instances on which the algorithm gives bad results.
2. Is this algorithm an approximation algorithm?
3. What modifications would you do on the algorithm to obtain better results?

Data: $G = (V, E)$ an undirected graph
Result: S: an independent set
1 $S = \emptyset$;
2 **while** V is not empty **do**
3 pick v a node in V;
4 add v to S;
5 remove v and all its neighbours from V;
6 **return** S

Figure 1: Greedy independent set algorithm

List Scheduling

We consider the scheduling problem where a set of heterogeneous tasks (of heterogeneous durations) has to be scheduled on a set of homogeneous machines. The problem is to find which machine should execute which task (only one task at a time) to minimize the last date of termination of a task. Figure 2 shows an example of this problem with 5 different tasks and the solution returned by the list scheduling algorithm with a completion time of 4 (here optimal).

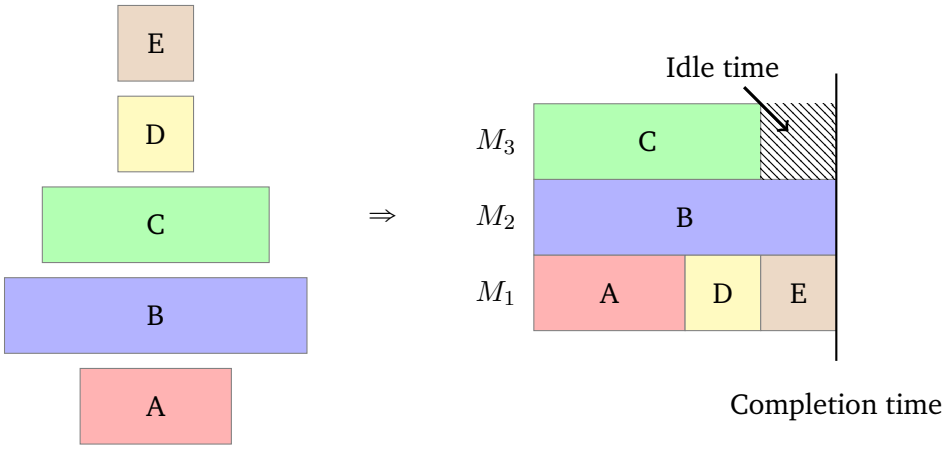

Figure 2: List scheduling example (3 machines). Tasks are on the left, and a Gantt diagram of a solution on the right.

The list-scheduling algorithm given in Figure 3 is a 2-approximation algorithm.

1. We consider as example a list of tasks of durations (1,1,1,1,1,1,1,1,1,3) on 3 machines

```
Data: a list of tasks, m the number of machines
Result: a schedule of the tasks
while all computations are not finished do
    while some machines are idle and there are tasks in the list do
        give the first task of the list to the first idle machine;
        remove first task from the list;
    advance in time until some machine becomes idle or the list of tasks becomes empty;
```

Figure 3: list scheduling algorithm

(a) Draw the Gantt diagram for this example.

(b) What is the optimal solution for this example?

(c) How far is the list scheduling from the optimal solution?

2. Build (on the same principle) a family of examples with a ratio list scheduling solution / optimal solution close to 2.

3. Prove that the list scheduling has an approximation ratio of 2 (hint: look at idle times).

Bin packing

We consider the bin packing problem where a set of n items of different sizes s_i ($\forall i \in 1, \ldots, n0 < s_i \leqslant 1$). A bin is a container providing a space of 1: the sum of the sizes of the items in a container cannot be more than 1. The goal of the bin packing is to pack all items in bins such that the total number of bins used is minimized.

Next Fit

We first consider the *next fit* algorithm of Figure 4.

```
Data: a list of items (a_1, ..., a_n) of sizes (s_1, ..., s_n)
Result: the contents of m bins used to pack the items
initially the current bin is empty;
foreach item a_i do
    if s_i is more than the free space in current bin then
        close current bin;
        create an empty new bin which will now be the current bin;
    put a_i in current bin;
    update the free space of current bin;
return the used bins;
```

Figure 4: Next Fit algorithm

1. Find a bad example for *next fit*.

2. Generalize to a family of examples where *next fit* gives a solution as close as you want from a factor 2 of the optimal number of bins.

3. Prove that *next fit* has an approximation ratio of 2.

First Fit

We can see that the *next fit* algorithm is not very efficient because only one bin is open at a time. To avoid such problems we now consider the *first fit* algorithm where all non-full bins can be considered for storage. The algorithm is displayed on Figure 5.

Data: a list of items $(a_1, \ldots, a_n)$ of sizes $(s_1, \ldots, s_n)$
Result: the contents of m bins used to pack the items
we have $B_1, B_2, B_3, \ldots$ a list of all possible bins;
initially all bins are empty;
foreach item a_i **do**
    Find the first bin B_j in the list of bins with enough space left for a_i;
    put a_i into B_j;
return all non-empty bins

Figure 5: First fit algorithm

1. Find the worst possible example for *first fit*.
2. Can you conjecture on the approximation ratio?

Knapsack

We consider the knapsack problem: we have a knapsack able to hold a weight W and a set of items $(o_i)_{1 \leqslant i \leqslant n}$. To each item o_i is assigned a value v_i and a weight w_i. As input of the problem all values and weights are given and the question is: how many of each kind of items should I take to maximize the value of the knapsack? If x_i is the number of times the object o_i is taken we try to maximize $\sum_i x_i \times v_i$ under the constraint $\sum_i x_i \times w_i \leqslant W$.

We propose the following greedy algorithm:

Data: a list of items with weights w_i and values v_i
Result: the number of times each item is selected
foreach item o_i **do**
    compute a value/weight ratio: $\frac{v_i}{w_i}$;
sort all items in decreasing ratios order;
let R be the remaining free weight in the knapsack;
initially $R = W$;
foreach item o_i in the sorted list **do**
    put $\left\lfloor \frac{R}{w_i} \right\rfloor$ times o_i in the knapsack;
    $R = R - \left\lfloor \frac{R}{w_i} \right\rfloor \times w_i$;
return the selected items

Figure 6: Greedy knapsack algorithm

1. Find the worst case example for this algorithm.
2. What would you suggest as an approximation ratio?
3. Prove the algorithm is an approximation algorithm.