Approximation Algorithms

CS 3AC3

Ryszard Janicki

Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Acknowledgments: Material based on *Algorithm Design* by Jon Kleinberg and Éva Tardos (Chapter 11)

Coping with NP-completeness

Q. Suppose I need to solve an NP-complete problem. What should I do?

A. Theory says you are unlikely to find poly-time algorithm.

We must sacrifice one of three desired features.

- Solve problem to optimality.
- Solve problem in polynomial time.
- Solve arbitrary instances of the problem.

ρ-approximation algorithm.

- Guaranteed to run in polynomial time.
- Guaranteed to solve arbitrary instance of the problem
- Guaranteed to find solution within ratio ρ of true optimum.

Challenge. Need to prove a solution's value is close to optimum, without even knowing what optimum value is.

Load balancing

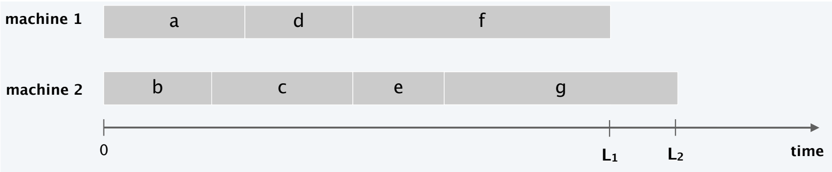
Input. m identical machines; n jobs, job j has processing time t_j.

- Job j must run contiguously on one machine.
- A machine can process at most one job at a time.

Definition (Load and Makespan)

1. Let $J(i)$ be the subset of jobs assigned to machine i.
   The load of machine i is $L_i = \sum_{k \in J(i)} t_j$.
2. The makespan is the maximum load on any machine $L = \max_i\{L_i\}$.

Load balancing. Assign each job to a machine to minimize makespan.



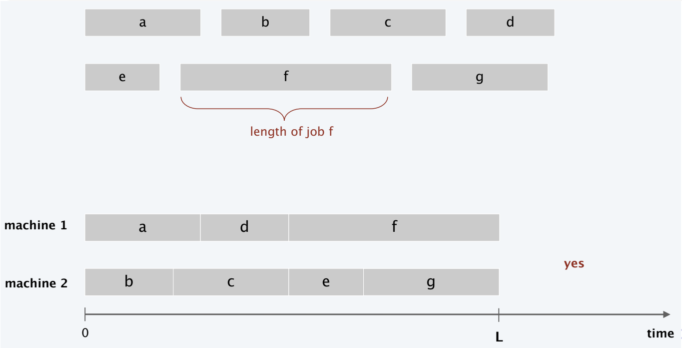
Load balancing on 2 machines is NP-hard

Proposition

Load balancing is hard even if only 2 machines.

Proof.

SUBSET-SUM $\leq_P$ NUMBER-PARTITIONING $\leq_P$ LOAD-BALANCE, where NUMBER-PARTITIONING is considered in Exercise 8.26 on page 518 of the textbook. We can prove directly SUBSET-SUM $\leq_P$ LOAD-BALANCE, but this way it is easier. □



Greedy List Scheduling Algorithm.

- Consider n jobs in some fixed order.
- Assign job j to machine whose load is smallest so far.

```
List-Scheduling(m, n, t_1,t_2,…,t_n) {
   for i = 1 to m {
      L_i ← 0              ← load on machine i
      J(i) ← ∅             ← jobs assigned to machine i
   }

   for j = 1 to n {
      i = argmin_k L_k         ← machine i has smallest load
      J(i) ← J(i) ∪ {j}        ← assign job j to machine i
      L_i ← L_i + t_j          ← update load of machine i
   }
   return J(1), …, J(m)
}
```

Implementation. $O(n \log m)$ using a priority queue (c.f. CS/SE 2C03 course).

Load balancing: Greedy list scheduling analysis

Theorem

Greedy list scheduling algorithm is a 2-approximation.

Proof.

- First worst-case analysis of an approximation algorithm.
- Need to compare resulting solution with optimal makespan L^*.
- Details in Kleinberg-Tardos.

$\square$

Greedy with Longest Processing Time (LPT). Sort n jobs in descending order of processing time, and then run list scheduling algorithm.

```
LPT-List-Scheduling(m, n, t1,t2,…,tn) {
   Sort jobs so that t1 ≥ t2 ≥  … ≥ tn

   for i = 1 to m {
      Li ← 0                 ← load on machine i
      J(i) ← ∅               ← jobs assigned to machine i
   }

   for j = 1 to n {
      i = argmink Lk             ← machine i has smallest load
      J(i) ← J(i) ∪ {j}          ← assign job j to machine i
      Li ← Li + tj               ← update load of machine i
   }
   return J(1), …, J(m)
}
```

Load balancing: Greedy with LPT rule

Theorem

Greedy with LPT rule is a $4/3$-approximation.

- Complexity is $O(n \log n)$ because of sorting.
- $4/3$-approximation is tight.

Polynomial-time approximation scheme

- **PTAS.** $(1+\varepsilon)$-approximation algorithm for any constant $\varepsilon > 0$.
- Consequence. PTAS produces arbitrarily high quality solution, but trades off accuracy for time.
- We will show PTAS for knapsack problem.

Knapsack (simple) problem

Knapsack problem.

- Given n objects and a knapsack.
- Item i has value $v_i > 0$ and weighs $w_i > 0$. ← we assume $w_i \leq W$ for each i
- Knapsack has weight limit W.
- Goal: fill knapsack so as to maximize total value.

Ex: $\{ 3, 4 \}$ has value 40.

item	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

original instance (W = 11)

Knapsack problem

Definition (Knapsack problem)

Given a set X, weights $w_i \geq 0$, values $v_i \leq 0$, a weight limit W, and a target value V, is there a subset $S \subseteq X$ such that:

$$\begin{aligned} \sum_{i \in S} w_i &\leq W \\ \sum_{i \in S} v_i &\geq V \end{aligned}$$

Knapsack is NP-complete

Definition (Knapsack problem)

Given a set X, weights $w_i \geq 0$, values $v_i \leq 0$, a weight limit W, and a target value V, is there a subset $S \subseteq X$ such that:

$$\sum_{i \in S} w_i \leq W \wedge \sum_{i \in S} v_i \geq V$$

SUBSET-SUM. Given a set X, values $u_i \geq 0$, and an integer U, is there a subset $S \subseteq X$ whose elements sum to exactly U ?

Theorem

SUBSET-SUM $\leq_P$ *KNAPSACK.*

Proof.

Given instance $(u_1, \ldots, u_n, U)$ of SUBSET-SUM, create KNAPSACK instance:

$$\begin{array}{ll} v_i = w_i = u_i & \sum_{i \in S} u_i \leq U \\ V = W = U & \sum_{i \in S} u_i \geq U \end{array}$$

$\square$

Knapsack problem: dynamic programming I

Def. $OPT(i, w)$ = max value subset of items $1,\ldots, i$ with weight limit w.

Case 1. OPT does not select item i.

- OPT selects best of $1, \ldots, i-1$ using up to weight limit w.

Case 2. OPT selects item i.

- New weight limit $= w - w_i$.
- OPT selects best of $1, \ldots, i-1$ using up to weight limit $w - w_i$.

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1,w) & \text{if } w_i > w \\ \max\{\, OPT(i-1,w),\ \ v_i + OPT(i-1,w-w_i)\,\} & \text{otherwise} \end{cases}$$

Theorem. Computes the optimal value in $O(n\,W)$ time.

- Not polynomial in input size.
- Polynomial in input size if weights are small integers.

Knapsack problem: dynamic programming II

Def. $OPT(i, v)$ = min weight of a knapsack for which we can obtain a solution of value $\geq v$ using a subset of items $1,..., i$.

Note. Optimal value is the largest value v such that $OPT(i, v) \leq W$.

Case 1. OPT does not select item i.

- OPT selects best of $1, \ldots, i-1$ that achieves value v.

Case 2. OPT selects item i.

- Consumes weight w_i, need to achieve value $v - v_i$.
- OPT selects best of $1, \ldots, i-1$ that achieves value $v - v_i$.

$$OPT(i, v) = \begin{cases} 0 & \text{if } v \leq 0 \\ \infty & \text{if } i = 0 \text{ and } v > 0 \\ \min\{OPT(i-1, v),\ w_i + OPT(i-1, v - v_i)\} & \text{otherwise} \end{cases}$$

Knapsack problem: dynamic programming II

Theorem. Dynamic programming algorithm II computes the optimal value in $O(n^2\, v_{max})$ time, where v_{max} is the maximum of any value.

Pf.

- The optimal value $V^* \leq n\, v_{max}$.
- There is one subproblem for each item and for each value $v \leq V^*$.
- It takes $O(1)$ time per subproblem. ▪

Remark 1. Not polynomial in input size!

Remark 2. Polynomial time if values are small integers.

Knapsack problem: Knapsack problem: polynomial-time approximation scheme

Intuition for approximation algorithm.

- Round all values up to lie in smaller range.
- Run dynamic programming algorithm II on rounded/scaled instance.
- Return optimal items in rounded instance.

item	value	weight
1	934221	1
2	5956342	2
3	17810013	5
4	21217800	6
5	27343199	7

original instance (W = 11)

item	value	weight
1	1	1
2	6	2
3	18	5
4	22	6
5	28	7

rounded instance (W = 11)

Knapsack problem: Knapsack problem: polynomial-time approximation scheme

Round up all values:

- $0 < \varepsilon \le 1$ = precision parameter.
- v_{max} = largest value in original instance.
- θ = scaling factor = $\varepsilon\, v_{max} / 2n$.

$$\bar{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil \theta\,, \quad \hat{v}_i = \left\lceil \frac{v_i}{\theta} \right\rceil$$

Observation. Optimal solutions to problem with $\bar{v}$ are equivalent to optimal solutions to problem with $\hat{v}$.

Intuition. $\bar{v}$ close to v so optimal solution using $\bar{v}$ is nearly optimal; $\hat{v}$ small and integral so dynamic programming algorithm II is fast.

Knapsack problem: Knapsack problem: polynomial-time approximation scheme

Theorem. If S is solution found by rounding algorithm and S^* is any other feasible solution, then $(1+\epsilon)\sum_{i \in S} v_i \geq \sum_{i \in S^*} v_i$

Pf. Let S^* be any feasible solution satisfying weight constraint.

$$
\begin{aligned}
\sum_{i \in S^*} v_i &\leq \sum_{i \in S^*} \bar{v}_i && \text{always round up} \\
&\leq \sum_{i \in S} \bar{v}_i && \text{solve rounded instance optimally} \\
&\leq \sum_{i \in S} (v_i + \theta) && \text{never round up by more than } \theta \\
&\leq \sum_{i \in S} v_i + n\theta && |S| \leq n \\
&= \sum_{i \in S} v_i + \tfrac{1}{2}\,\epsilon\, v_{max} && \theta = \varepsilon v_{max} / 2n \\
&= (1+\epsilon)\sum_{i \in S} v_i && v_{max} \leq 2\,\Sigma_{i \in S}\, v_i
\end{aligned}
$$

choosing $S^* = \{\, max \,\}$

$$
\begin{aligned}
v_{max} &\leq \sum_{i \in S} v_i + \tfrac{1}{2}\,\epsilon\, v_{max} \\
&\leq \sum_{i \in S} v_i + \tfrac{1}{2}\, v_{max}
\end{aligned}
$$

thus

$$
v_{max} \leq 2 \sum_{i \in S} v_i
$$

Knapsack problem: Knapsack problem: polynomial-time approximation scheme

Theorem. For any $\varepsilon > 0$, the rounding algorithm computes a feasible solution whose value is within a $(1+\varepsilon)$ factor of the optimum in $O(n^3 / \varepsilon)$ time.

Pf.

- We have already proved the accuracy bound.
- Dynamic program II running time is $O(n^2\, \hat{v}_{\max})$, where

$$\hat{v}_{\max} = \left\lceil \frac{v_{\max}}{\theta} \right\rceil = \left\lceil \frac{n}{\varepsilon} \right\rceil$$