Chapter 7
Approximation Algorithms

Algorithm Theory
WS 2012/13

Fabian Kuhn
Approximation Algorithms

• Optimization appears everywhere in computer science

• We have seen many examples, e.g.:
  – scheduling jobs
  – traveling salesperson
  – maximum flow, maximum matching
  – minimum spanning tree
  – minimum vertex cover
  – ...

• Many discrete optimization problems are NP-hard

• They are however still important and we need to solve them

• As algorithm designers, we prefer algorithms that produce solutions which are provably good, even if we can’t compute an optimal solution.
Approximation Algorithms: Examples

We have already seen two approximation algorithms

• **Metric TSP**: If distances are positive and satisfy the triangle inequality, the greedy tour is only by a log-factor longer than an optimal tour

• **Maximum Matching and Vertex Cover**: A maximal matching gives solutions that are within a factor of 2 for both problems.
Approximation Ratio

An approximation algorithm is an algorithm that computes a solution for an optimization with an objective value that is provably within a bounded factor of the optimal objective value.

Formally:

- \( \text{OPT} \geq 0 \): optimal objective value
  \( \text{ALG} \geq 0 \): objective value achieved by the algorithm

- **Approximation Ratio** \( \alpha \):
  
  **Minimization:** \( \alpha := \max_{\text{input instances}} \frac{\text{ALG}}{\text{OPT}} \)

  **Maximization:** \( \alpha := \max_{\text{input instances}} \frac{\text{OPT}}{\text{ALG}} \)
Example: Load Balancing

We are given:

- $m$ machines $M_1, \ldots, M_m$
- $n$ jobs, processing time of job $i$ is $t_i$

Goal:

- Assign each job to a machine such that the makespan is minimized

**makespan**: largest total processing time of a machine

The above load balancing problem is NP-hard and we therefore want to get a good approximation for the problem.
Greedy Algorithm

There is a simple greedy algorithm:

- Go through the jobs in an arbitrary order
- When considering job \( i \), assign the job to the machine that currently has the smallest load.

**Example:** 3 machines, 12 jobs

| 3 | 4 | 2 | 3 | 1 | 6 | 4 | 4 | 3 | 2 | 1 | 5 |

**Greedy Assignment:**

\( M_1: \) 3 1 6 1 5
\( M_2: \) 4 4 3
\( M_3: \) 2 3 4 2

**Optimal Assignment:**

\( M_1: \) 3 4 2 3 1
\( M_2: \) 6 4 3
\( M_3: \) 4 2 1 5
Greedy Analysis

• We will show that greedy gives a 2-approximation

• To show this, we need to compare the solution of greedy with an optimal solution (that we can’t compute)

• Lower bound on the optimal makespan $T^*$:

$$T^* \geq \frac{1}{m} \cdot \sum_{i=1}^{n} t_i$$

• Lower bound can be far from $T^*$:
  – $m$ machines, $m$ jobs of size 1, 1 job of size $m$

  $$T^* = m, \quad \frac{1}{m} \cdot \sum_{i=1}^{n} t_i = 2$$
Greedy Analysis

• We will show that greedy gives a 2-approximation

• To show this, we need to compare the solution of greedy with an optimal solution (that we can’t compute)

• Lower bound on the optimal makespan $T^*$:

$$T^* \geq \frac{1}{m} \cdot \sum_{i=1}^{n} t_i$$

• Second lower bound on optimal makespan $T^*$:

$$T^* \geq \max_{1 \leq i \leq n} t_i$$
Greedy Analysis

**Theorem:** The greedy algorithm has approximation ratio \( \leq 2 \), i.e., for the makespan \( T \) of the greedy solution, we have \( T \leq 2T^* \).

**Proof:**

- For machine \( k \), let \( T_k \) be the time used by machine \( k \)
- Consider some machine \( M_i \) for which \( T_i = T \)
- Assume that job \( j \) is the last one schedule on \( M_i \):

\[
M_i: \begin{array}{c|c}
T & t_j \\
\hline
T - t_j & t_j \\
\end{array}
\]

- When job \( j \) is scheduled, \( M_i \) has the minimum load
Greedy Analysis

**Theorem:** The greedy algorithm has approximation ratio $\leq 2$, i.e., for the makespan $T$ of the greedy solution, we have $T \leq 2T^*$.

**Proof:**
- For all machines $M_k$: load $T_k \geq T - t_j$
Can We Do Better?

The analysis of the greedy algorithm is almost tight:

• Example with $n = m(m - 1) + 1$ jobs
• Jobs 1, ..., $n - 1 = m(m - 1)$ have $t_i = 1$, job $n$ has $t_n = m$

**Greedy Schedule:**

$M_1$: 1111 ... 1 $t_n = m$

$M_2$: 1111 ... 1

$M_3$: 1111 ... 1

:    :

$M_m$: 1111 ... 1

Algorithm Theory, WS 2012/13

Fabian Kuhn
Improving Greedy

Bad case for the greedy algorithm: One large job in the end can destroy everything

**Idea:** assign large jobs first

**Modified Greedy Algorithm:**
1. Sort jobs by decreasing length s.t. \( t_1 \geq t_2 \geq \cdots \geq t_n \)
2. Apply the greedy algorithm as before (in the sorted order)

**Lemma:** \( T^* \geq t_m + t_{m+1} \geq 2t_{m+1} \)

**Proof:**
- Two of the first \( m + 1 \) jobs need to be scheduled on the same machine
- Jobs \( m \) and \( m + 1 \) are the shortest of these jobs
Analysis of the Modified Greedy Alg.

**Theorem:** The modified algorithm has approximation ratio \( \leq \frac{3}{2} \), i.e., we have \( T \leq \frac{3}{2} \cdot T^* \).

**Proof:**

- As before, choose machine \( M_i \) with \( T_i = T \)
- Job \( t_j \) is the last one scheduled on machine \( M_i \)
- If there is only one job \( t_j \) on \( M_i \), we have \( T_i = t_j = T^* \)
- Otherwise, we have \( j \geq m + 1 \)
  - The first \( m \) jobs are assigned to \( m \) distinct machines
Metric TSP

Input:
• Set $V$ of $n$ nodes (points, cities, locations, sites)
• Distance function $d: V \times V \to \mathbb{R}$, i.e., $d(u, v)$: dist. from $u$ to $v$
• Distances define a metric on $V$:
  \[
  d(u, v) = d(v, u) \geq 0, \quad d(u, v) = 0 \iff u = v
  \]
  \[
  d(u, v) \leq d(u, w) + d(v, w)
  \]

Solution:
• Ordering/permutation $v_1, v_2, \ldots, v_n$ of vertices
• Length of TSP path: $\sum_{i=1}^{n-1} d(v_i, v_{i+1})$
• Length of TSP tour: $d(v_n, v_1) + \sum_{i=1}^{n-1} d(v_i, v_{i+1})$

Goal:
• Minimize length of TSP path or TSP tour
Metric TSP

• The problem is **NP-hard**

• We have seen that the **greedy** algorithm (always going to the nearest unvisited node) gives an $O(\log n)$-**approximation**

• Can we get a constant approximation ratio?

• We will see that we can...
TSP and MST

**Claim:** The length of an optimal TSP path is upper bounded by the weight of a minimum spanning tree

**Proof:**
- A TSP path is a spanning tree, it’s length is the weight of the tree

**Corollary:** Since an optimal TSP tour is longer than an optimal TSP path, the length of an optimal TSP tour is less than the weight of a minimum spanning tree.
The MST Tour

Walk around the MST...

Cost: $< 2 \cdot \text{weight}(\text{MST})$
Approximation Ratio of MST Tour

**Theorem:** The MST TSP tour gives a 2-approximation for the metric TSP problem.

**Proof:**
- Triangle inequality $\implies$ length of tour is at most $2 \cdot \text{weight(MST)}$
- We have seen that $\text{weight(MST)} \leq \text{opt. tour length}$

Can we do even better?
Claim: Given a metric \((V, d)\) and \((V', d)\) for \(V' \subseteq V\), the optimal TSP path/tour of \((V', d)\) is at most as large as the optimal TSP path/tour of \((V, d)\).

Optimal TSP tour of nodes 1, 2, ..., 12

Induced TSP tour for nodes 1, 2, 5, 8, 10, 12

blue tour \(\leq\) green tour
TSP and Matching

• Consider a metric TSP instance \((V, d)\) with an even number of nodes \(|V|\)

• Recall that a perfect matching is a matching \(M \subseteq V \times V\) such that every node of \(V\) is incident to an edge of \(M\).

• Because \(|V|\) is even and because in a metric TSP, there is an edge between any two nodes \(u, v \in V\), any partition of \(V\) into \(|V|/2\) pairs is a perfect matching.

• The weight of a matching \(M\) is the total distance of the edges in \(M\).
Lemma: Assume, we are given a metric TSP instance \((V, d)\) with an even number of nodes. The length of an optimal TSP tour of \((V, d)\) is at least twice the weight of a minimum weight perfect matching of \((V, d)\).

Proof:
• The edges of a TSP tour can be partitioned into 2 perfect matchings
Minimum Weight Perfect Matching

**Claim:** A minimum weight perfect matching of \((V, d)\) can be computed in polynomial time

**Proof Sketch:**

- We have seen that a maximum matching in an unweighted graph can be computed in polynomial time
- With a more complicated algorithm, also a maximum weighted matching can be computed in polynomial time
- In a complete graph, a maximum weighted matching is also a (maximum weight) perfect matching
- Define weight \(w(u, v) := D - d(u, v)\)
- A maximum weight perfect matching for \((V, w)\) is a minimum weight perfect matching for \((V, d)\)
Algorithm Outline

Problem of MST algorithm:
• Every edge has to be visited twice

Goal:
• Get a graph on which every edge only has to be visited once
  (and where still the total edge weight is small compared to an
  optimal TSP tour)

Euler Tours:
• A tour that visits each edge of a graph exactly once is called an
  Euler tour
• An Euler tour in a (multi-)graph exists if and only every node
  of the graph has even degree
• That’s definitely not true for a tree, but can we get it?
Euler Tour

**Theorem:** A connected graph $G$ has an Euler tour if and only if every node of $G$ has even degree.

**Proof:**

1. If $G$ has an odd degree node, it clearly cannot have an Euler tour
2. If $G$ has only even degree nodes, a tour can be found recursively

1. Start at some node
2. As long as possible, follow an unvisited edge
   - Gives a partial tour, the remaining graph still has even degree
3. Solve problem on remaining components recursively
4. Merge the obtained tours into one tour that visits all edges
TSP Algorithm

1. Compute MST $T$
2. $V_{\text{odd}}$: nodes that have an odd degree in $T$ ($|V_{\text{odd}}|$ is even)
3. Compute min weight maximum matching $M$ of $(V_{\text{odd}}, d)$
4. $(V, T \cup M)$ is a (multi-)graph with even degrees
TSP Algorithm

5. Compute Euler tour on $(V, T \cup M)$

6. Total length of Euler tour $\leq \frac{3}{2} \cdot \text{TSP}_{\text{OPT}}$

7. Get TSP tour by taking shortcuts wherever the Euler tour visits a node twice
TSP Algorithm

• The described algorithm is by Christofides

**Theorem:** The Christofides algorithm achieves an approximation ratio of at most $\frac{3}{2}$.

**Proof:**

• The length of the Euler tour is $\leq \frac{3}{2} \cdot \text{TSP}_{\text{OPT}}$
• Because of the triangle inequality, taking shortcuts can only make the tour shorter
Knapsack

• \( n \) items 1, \( ..., n \), each item has weight \( w_i > 0 \) and value \( v_i > 0 \)
• Knapsack (bag) of capacity \( W \)
• Goal: pack items into knapsack such that total weight is at most \( W \) and total value is maximized:

\[
\max \sum_{i \in S} v_i \\
\text{s.t. } S \subseteq \{1, ..., n\} \text{ and } \sum_{i \in S} w_i \leq W
\]

• E.g.: jobs of length \( w_i \) and value \( v_i \), server available for \( W \) time units, try to execute a set of jobs that maximizes the total value
Knapsack: Dynamic Programming Alg.

We have shown:

- If all item weights $w_i$ are integers, using dynamic programming, the knapsack problem can be solved in time $O(nW)$
- If all values $v_i$ are integers, there is another dynamic progr. algorithm that runs in time $O(n^2V)$, where $V$ is the max. value.

Problems:

- If $W$ and $V$ are large, the algorithms are not polynomial in $n$
- If the values or weights are not integers, things are even worse (and in general, the algorithms cannot even be applied at all)

Idea:

- Can we adapt one the algorithms to at least compute an approximate solution?
Approximation Algorithm

- The algorithm has a parameter $\varepsilon > 0$
- We assume that each item alone fits into the knapsack
- We define:
  \[
  V := \max_{1 \leq i \leq n} v_i, \quad \forall i: \hat{v}_i := \left\lfloor \frac{v_i n}{\varepsilon V} \right\rfloor, \quad \hat{V} := \max_{1 \leq i \leq n} \hat{v}_i
  \]
- We solve the problem with values $\hat{v}_i$ and weights $w_i$ using dynamic programming in time $O(n^2 \cdot \hat{V})$

**Theorem:** The described algorithm runs in time $O(n^3 / \varepsilon)$.

**Proof:**

\[
\hat{V} = \max_{1 \leq i \leq n} \hat{v}_i = \max_{1 \leq i \leq n} \left\lfloor \frac{v_i n}{\varepsilon V} \right\rfloor = \left\lfloor \frac{V n}{\varepsilon V} \right\rfloor = \left\lfloor \frac{n}{\varepsilon} \right\rfloor
\]
Approximation Algorithm

**Theorem:** The approximation algorithm computes a feasible solution with approximation ratio at most $1 + \varepsilon$.

**Proof:**

- Define the set of all feasible solutions

$$S := \left\{ S \subseteq \{1, \ldots, n\} : \sum_{i \in S} w_i \leq W \right\}$$

- Let $S^*$ be an optimal solution and $\hat{S}$ be the solution computed by the approximation algorithm.

- We have

$$S^* = \max_{S \in \mathcal{S}} \sum_{i \in S} v_i, \quad \hat{S} = \max_{\bar{S} \in \mathcal{S}} \sum_{\bar{S} \in \mathcal{S}} \bar{v}_i$$

- Hence, $\hat{S}$ is a feasible solution
Approximation Algorithm

**Theorem:** The approximation algorithm computes a feasible solution with approximation ratio at most $1 + \varepsilon$.

**Proof:**

- Because every item fits into the knapsack, we have
  \[ \forall i \in \{1, \ldots, n\}: v_i \leq \sum_{j \in S^*} v_j \]

- For the solution of the algorithm, we get
  \[ \hat{v}_i = \left\lfloor \frac{v_in}{\varepsilon V} \right\rfloor \Rightarrow v_i \leq \frac{\varepsilon V}{n} \cdot \hat{v}_i \]

- Therefore
  \[ \sum_{i \in S^*} v_i \leq \frac{\varepsilon V}{n} \cdot \sum_{i \in S^*} \hat{v}_i \leq \frac{\varepsilon V}{n} \cdot \sum_{i \in \hat{S}} \hat{v}_i \leq \frac{\varepsilon V}{n} \cdot \sum_{i \in \hat{S}} \left( \frac{v_in}{\varepsilon V} + 1 \right) \]

Algorithm Theory, WS 2012/13

Fabian Kuhn

32
Approximation Algorithm

**Theorem:** The approximation algorithm computes a feasible solution with approximation ratio at most $1 + \varepsilon$.

**Proof:**

- We have
  \[
  \sum_{i \in S^*} v_i \leq \frac{\varepsilon V}{n} \cdot \sum_{i \in S^*} \hat{v}_i \leq \frac{\varepsilon V}{n} \cdot \sum_{i \in \hat{S}} \hat{v}_i \leq \frac{\varepsilon V}{n} \cdot \sum_{i \in \hat{S}} \left( \frac{v_i n}{\varepsilon V} + 1 \right)
  \]

- Therefore
  \[
  \sum_{i \in S^*} v_i \leq \sum_{i \in \hat{S}} v_i + \frac{\varepsilon V}{n} \cdot |\hat{S}| \leq \varepsilon V + \sum_{i \in \hat{S}} v_i
  \]

- Because $V$ is a lower bound on the optimal solution:
  \[
  \sum_{i \in S^*} v_i \leq (1 + \varepsilon) \cdot \sum_{i \in \hat{S}} v_i
  \]
Approximation Schemes

• For every parameter $\epsilon > 0$, the knapsack algorithm computes a $(1 + \epsilon)$-approximation in time $O(n^3/\epsilon)$.

• For every fixed $\epsilon$, we therefore get a polynomial time approximation algorithm

• An algorithm that computes an $(1 + \epsilon)$-approximation for every $\epsilon > 0$ is called an approximation scheme.

• If the running time is polynomial for every fixed $\epsilon$, we say that the algorithm is a polynomial time approximation scheme (PTAS)

• If the running time is also polynomial in $1/\epsilon$, the algorithm is a fully polynomial time approximation scheme (FPTAS)

• Thus, the described alg. is an FPTAS for the knapsack problem