Algorithms

Lecture Notes — Approximation Algorithms

Imdad ullah Khan

Contents

1 Hard Problems and Approximation Algorithms

A large number of optimization problems are known to be NP-hard, which we know from complexity theory are impossible to solve unless $P = NP$ (which is widely believed to be not true). We nonetheless need to solve these hard problems as they appear in many practical scenarios.

We make the following remarks about how to go about hard problems

- **Super polynomial time heuristics:** Some algorithms are just barely super-polynomial and run reasonably fast in most practical cases. For example in class we discussed a dynamic programming based solution for the Knapsack problem and proved that the runtime of that algorithms was pseudo-polynomial. For most practical cases this is good enough. But there aren't many problem for which we can find pseudo-polynomial time algorithms.
- **Probabilistic Analysis:** We can also strop focusing on worst case runtime analysis, instead we may ask what is the average case runtime of the algorithm. For some problems there are very few instances for which an algorithm takes exponential amount of time. For example for the Hamiltonian cycle problem, there is an algorithm that will find a Hamiltonian cycle in almost every Hamiltonian graph. Such results are usually derived by assuming a probability distribution on the input instances and then it shows that the algorithm will solve the problem with high probability.

  It is not very easy to come up with a reasonable probability distribution on the input instance.
- **Approximation Algorithms:** In this strategy we drop the requirement that the algorithm will find the optimal solution. Rather we relax the requirement by asking for a feasible solution that is some sense "pretty close" to the optimal solution. For a large body of NP-hard problems there are polynomial time algorithms that find solution that are nearly optimal (only slightly sub-optimal).

1.1 Preliminaries and examples of hard problems

An optimization problem P is characterized by the three things

- $\mathcal{I}$: Set of input instances
- $S(I)$: solution space, set of all feasible solutions for an instance $I \in \mathcal{I}$
- $f : S(I) \to \mathbb{R}$: A function f that maps solutions to a real number, this is called the value of a solution.

A **maximization problem** is given $I \in \mathcal{I}$, find a solution $s \in S(I)$ such that $f(s)$ is maximum, i.e.

$$\forall s' \in S(I), f(s) \geq f(s').$$

We will sometime refer the value of the optimal solution as $OPT(I) = f(s)$.

A **minimization problem** is defined analogously. Note that optimal solution need not be unique, but it will always exist.

In this part of the course we will consider NP-hard optimization problems and assume that $P \neq NP$. This basically assume that for all problems we will consider there is no polynomial time algorithm that finds an optimal solution for all instances of the problem. Examples of hard problems that we discussed in class are maximum independent set in a graph, maximum clique in a graph, Hamiltonian cycle in a graph, minimum set cover, set packing, minimum vertex cover in a graph, graph coloring, the knapsack problem, optimal scheduling on parallel machine. There are thousands of problems that are NP hard, we will define only those problems that we will study.

Definition 1. *An* **approximation algorithm** *A for an optimization problem P, is a* **polynomial time** *algorithm such that given an input instance I of P, it outputs a solution $s' \in S(I)$. Some time we will denote by $A(I)$ both the value of the solution $f(s')$ or the solution itself i.e. s' (it will be clear from the context).*

We need to extend this definition to be able to compare the value $A(I)$ with $OPT(I)$. So far this only asks for a feasible solution. For example for the maximum independent set problem in a graph G, since every vertex is an independent set of G, the algorithm that arbitrarily outputs a vertex of G satisfies this definition, which clearly is a trivial solution. We need to be able to measure the goodness of approximation algorithms, by asking for certain performance guarantees of approximation.

2 Absolute approximation guarantees

Definition 2. *An* **absolute approximation algorithm** *for a problem P is a polynomial time approximation algorithm such that there is some constant $k > 0$ such that*

$$\forall\ I \in \mathcal{I}, |A(I) - OPT(I)| \leq k.$$

Let us restate it for minimization problem

Definition 3. *An* **absolute approximation algorithm** *for a minimization problem P is a polynomial time approximation algorithm such that there is some constant $k > 0$ such that*

$$\forall\ I \in \mathcal{I}, A(I) \leq OPT(I) + k.$$

Equivalently it means that $f(s') \leq f(s) + k$.

This is clearly the best possible solution for a NP-hard minimization problem.

2.1 Planar graph coloring

The **graph coloring problem** is to color vertices of the graph with a minimum number of colors such that no adjacent vertices get the same color. More formally the graph coloring problem has

- $\mathcal{I}$: Graphs
- $S(I)$: An assignment of colors to vertices of input graph, in other words such that no two adjacent vertices have the same color.
- $f(s)$: number of colors used in the coloring s.

We denote by $\chi(G)$ the minimum number of colorings needed to color a graph G.

Following is an example graph, with two different colorings.

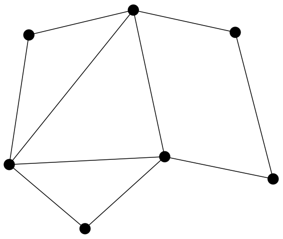


Figure 1: Original graph

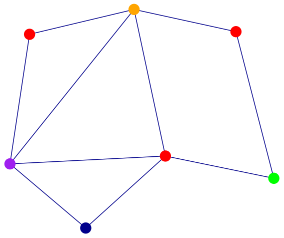


Figure 2: Graph colored with 4 colors

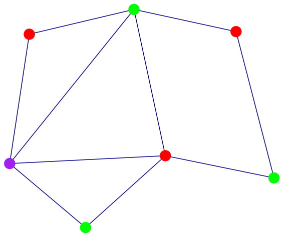
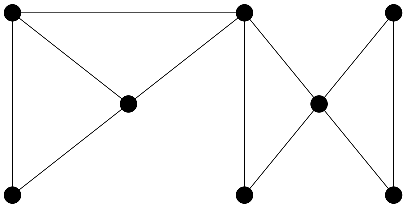
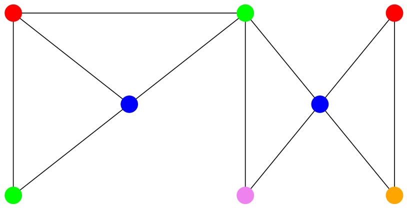


Figure 3: Graph colored with 3 colors

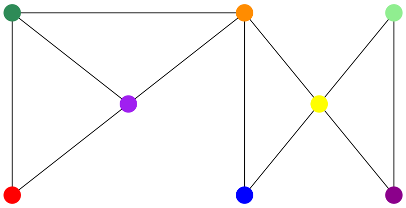
Here is another example from our slides



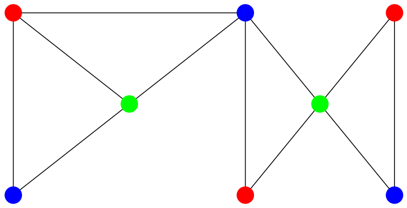
A graph G on 8 vertices



A coloring with 6 colors



A coloring with 8 colors



A coloring with (optimal) 3 colors

Definition 4. *A graph is planar, if it can be drawn on a plane without any edge crossing.*

Note that the definition says that it can be drawn, a planar graph could be drawn with edges crossing.

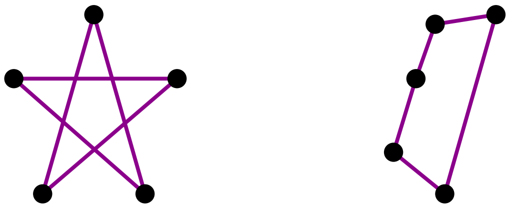


Figure 4: Planar graph with non-planar and planar drawing

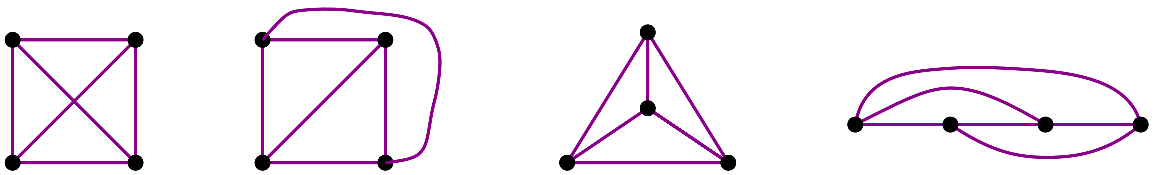


Figure 5: A planar graph K_4

Theorem 5 (Euler's formula)**.** *A planar graph with* $n \geq 3$ *vertices and* m *edges has* $m \leq 3n - 6$

An immediate corollary of the above fact using the handshaking lemma is that

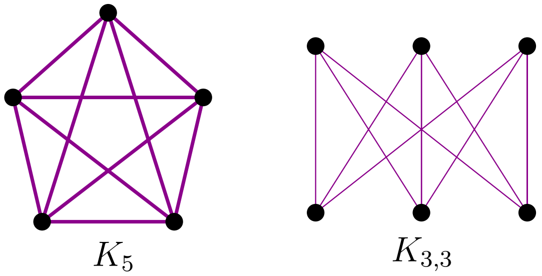


Figure 6: Non planar graphs K_5 and $K_{3,3}$

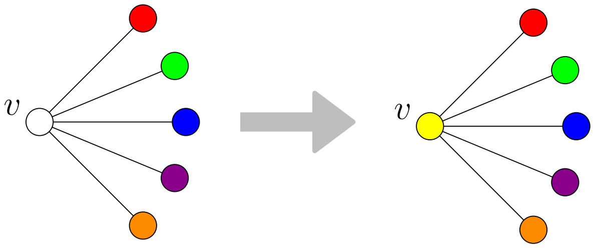
Corollary 6. *Every planar graph has a vertex with degree at most* 5*.*

A well known result about complexity of planar graph coloring is given in the following theorem.

Theorem 7. *The decision problem of whether or not a planar graph is* 3*-colorable is* NP*-complete.*

A very easy observation following from the above minimum degree condition is the following positive result about coloring planar graphs.

Theorem 8. *Any planar graph is* 6 *colorable.*



Proof. The proof is by induction with base case $|V| \leq 5$, is trivial. Let v be a vertex such that $deg(v) \leq 5$, by the above corollary there must exist such a vertex. Consider the graph $G - v$, Since G is planar, $G - v$ is also planar. By induction $G - v$ is 6 colorable. Consider a coloring of $G - v$ and add back to it v. Neighbors of v, $(N(v))$ are already colored, but the maximum number of color used for $N(v)$ is at most 5 and one of the 6 colors is available and legitimate to be used for v, we use that available color to cold v with and thus extend the coloring to the whole G. □

Note that the above consecutive proof readily gives us a recursive algorithm for 6 coloring of graph. Consider the following algorithm as our approximation algorithm to color planar graphs.

```
Algorithm APPROX-PLANAR-COLOR($\mathcal{G}$)
  if $G$ is bipartite then                          ▷ Easy to check with a BFS
      Color $G$ with the obvious 2 coloring
  else
      Color $G$ with the 6 coloring as in Theorem 8
```

Theorem 9. *The algorithm PlanarColor is a* 3*-absolute approximation algorithm. In other words*

Proof. If G is not bipartite, then the minimum colors needed to color G is at least 3 ($OPT(G) \geq 3$) and PlanarColor uses at most 6 colors ($PlanarColor(G) \leq 6$), hence the statement follows. □

Actually it is well known that any planar graph is 5 colorable. There is an easy algorithm for 5-coloring of planar graphs.

Theorem 10. *Any planar graph is* 5*-colorable.*

Proof. Please check out its proof online, I may include it in an appendix. □

This theorem immediately gives us a 2-absolute approximation algorithm. In fact the famous 4 color theorem gives us a 1-absolute approximation algorithm. Read online about the 4 color theorem.

Figure 7: UIUC postage stamps in honor the 4 color theorem

2.2 Negative result for absolute approximation algorithm by scaling

Generally absolute approximation algorithms exists for problems where one can find a rather small range of values where the optimal value lies. The hardness of such problems is determining the exact value of the optimum solution within this range.

Such an absolute approximation algorithm merely uses the fact that the range is very small to come up with a tight absolute approximation guarantee.

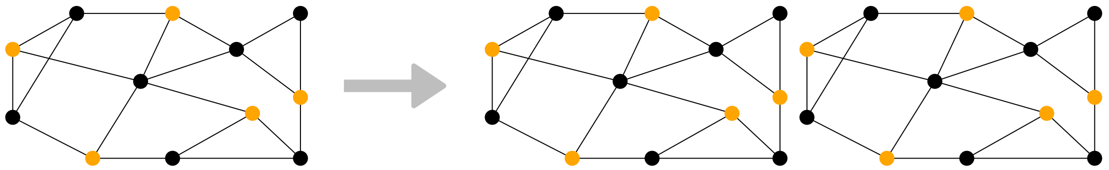
Not many hard problems have an absolute approximation algorithm. Typically such impossibility results use the technique of scaling. The broad idea is as follows: We first scale up certain parameter(s) associate with the instance, secondly we show that if there is an absolute approximation algorithm for the sacred up instance, then the solution can be rescaled to get an optimum solution for the original instance. But this imply an efficient algorithm to solve the optimization problem, which by our assumption of $P \neq NP$ is not possible.

2.2.1 Impossibility result for the Max Independent set problem

The maximum independent set problem is that of finding the largest subset of vertices in the graph such that no pair of them is adjacent. As we saw earlier this an NP-hard problem.

Theorem 11. *If $P \neq NP$, then there is no k-absolute approximation algorithm for the maximum independent set (MIS) problem.*

The proof uses the so-called scaling technique. We first give a specific example on how the scaling technique works, later we give a general proof.

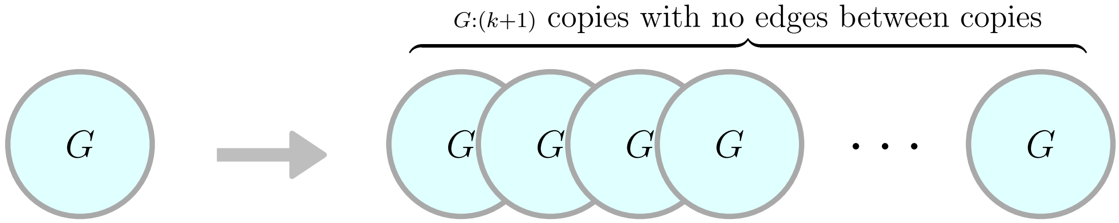


In I Max Ind. set is of size 5 In $2I$ Max Ind. set is of size 10

Note: $f(OPT(2\text{I})) = 2f(OPT(\text{I}))$

We assume that there is a k-absolute approximation algorithm. We scale the original instance I by a factor of 2 (call this instance $2I$). Note that $OPT(2I) = 2OPT(I)$. We run the above assumed algorithm on the instance $2I$ to get an approximate result of size at least $OPT(2I) - k = 2OPT(I) - k$. We divide this solution by 2 to get a solution for I of size at least $\frac{2OPT(I)-k}{2} = OPT(I) - \frac{k}{2}$. Hence we got a even better approximation algorithm, namely a $\frac{k}{2}$-absolute approximation algorithm. Now it is easy to see that if we keep on applying the scaling trick at some point the approximation guarantee will drop below 1. At this point typically we appeal to the integrality of the optimal solution and argue that we actually found an optimal solution.

Proof. Assume the contrary, that is assume that there is k-absolute approximation algorithm A for MIS. For any $G = (V, E)$, let G' be made of $k+1$ disjoint copies of G (there are no edges between them). It is easy to see that the MIS of G' is composed of one MIS in each copy of G. This implies that $OPT(G') = (k+1)OPT(G)$.



Note: $f(OPT(G')) = (k+1)f(OPT(G))$

Now run A on G', it will return an independent set of size at least $OPT(G') - k$ (by its quality guarantee), which is composed of independent sets in each copy of G. By the pigeon hole principle, at least one copy of G must contain an independent set of size at least

$$\frac{OPT(G') - k}{k+1} = \frac{(k+1)OPT(G) - k}{k+1} \geq OPT(G) - \frac{1}{k} \geq OPT(G).$$

Hence we can find the optimum solution in G in polynomial time, which by our assumption ($P \neq NP$), is not possible. □

2.2.2 Impossibility result for the knapsack problem

In this section we show that it is impossible to come up with an absolution approximation algorithm for the knapsack problem. We didn't cover this in the class, but since we discuss it in great details in terms of positive result, it is worth noting why we didn't give an absolute approximation algorithm for it.

Recall the knapsack problem, where an instance of it consists of

- Items $U = \{1, 2, \ldots, n\}$
- Weights (or sizes) $W = \{w_1, w_2, \ldots, w_n\}$, where w_i is the weight of item $i \in U$.
- Values (or profits) $V = \{p_1, p_w, \ldots, p_n\}$, where v_i is the value of item $i \in U$.
- Knapsack capacity (or budget) C.

A feasible solution to the problem is a subset $U' \subseteq U$, such that $\sum_{i \in U'} w_i \leq C$. Our goal is to maximize $f(U') = \sum_{i \in U'} v_i$.

Informally we would like to pack some items of different sizes into a knapsack of fixed capacity so as to maximize the total profits from the packed items.

This problem is NP-hard. There is no k-absolute approximation algorithm for the knapsack problem.

Theorem 12. *If $P \neq NP$, then there is no k-absolute approximation algorithm for the knapsack problem.*

Proof. Suppose there is a k-absolute approximation algorithm A for the knapsack problem. Given an instance I of the knapsack problem (where all sizes and profits are integers), let $(2k)I$ be the instance such that everything remains the same as I excepts profits are scaled up by a factor of $2k$, i.e. $p_i' = 2k \cdot p_i$. It is easy to see that $OPT((2k)I) = 2kOPT(I)$, because same capacity and same weights implies we can take the same items as in optimal solution to I, the only difference is that the value of solution for $(2k)I$ is $2k$ times $OPT(I)$.

Running algorithm A on $(2k)I$ by its performance guarantee gives us a solution such that $A((2k)I) \geq OPT((2k)I) - k$. Dividing value of each selected item in this solution by $2k$ we get a solution s' to the original instance I, whose value is at least

$$\frac{OPT((2k)I) - k}{2k} = \frac{2kOPT(I) - k}{2k} \geq OPT(I) - \frac{1}{2}.$$

Since by our assumption I has integer weights and values, the maximum achievable value ($OPT(I)$) is also an integer, hence $f(s') \geq OPT(I) - \frac{1}{2}$ must be equal to $OPT(I)$. This clearly gives us a polynomial time algorithm to solve optimally any integer instance of the knapsack problem, contradicting our assumption that $P \neq NP$. □

3 Relative approximation algorithms

Given that we can't always find the most desirable absolute approximation algorithm (indeed very few problems admit absolution approximation guarantees as discussed above), the reasonable thing to do is to relax the definition of "pretty good" requirement for approximation algorithm.

Definition 13. *A* **relative approximation algorithm** *A for a maximization problem P is a polynomial time approximation algorithm such that there is some constant $\alpha > 0$*

such that

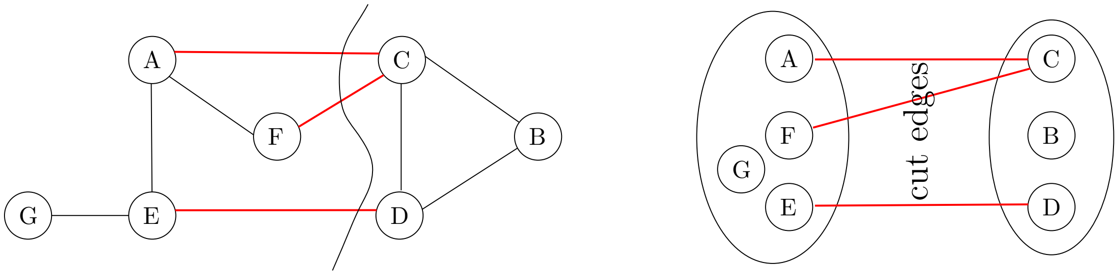
$$\forall\ I \in \mathcal{I}, |A(I) \geq \frac{OPT(I)}{\alpha}.$$

In other words if s' is the solution given by A and s is the optimal solution, then $f(s') \geq f(s)/\alpha$.

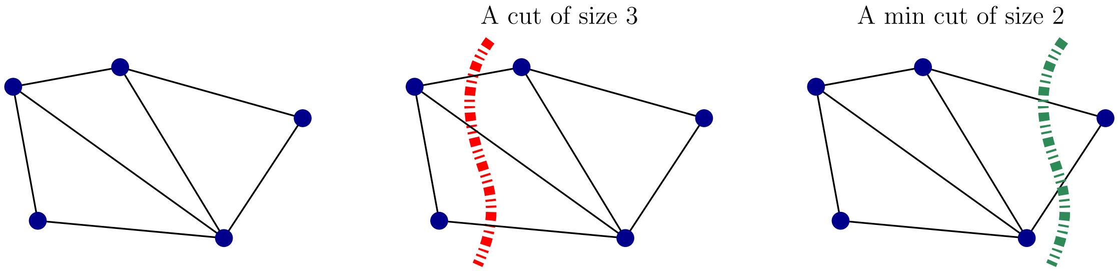
For a minimization problem it is analogously defined, except for the requirement is $f(s') \leq \alpha f(s)$. We call such algorithms α-approximate algorithm.

3.1 Maximum Cut Problem

Given an undirected graph $G = (V, E)$. A cut in G is a subset $S \subset V$, it is denoted by $[S, \overline{S}]$. The size (or cost) of a cut in the number of crossing or cut edges. If the graph has weights on its edges, then the cost of the cut is defined to be the sum of weights of cut edges.



The maximum cut problem is to partition V into two subsets S and $\overline{S} = V \setminus S$ such that the number of edges in the cut $[S, \overline{S}]$ is the maximum.



The problem is NP-hard; we give an approximation algorithm for it. The algorithm is very simple, in iterates over all the vertices in an arbitrary order and keep each vertex either in A or $B = V \setminus A$ that are initially empty. In each step if the current vertex v has more neighbors in A, then it keeps v in B and vice-versa. If v has no neighbors in either or has equal number of neighbors in both, then it arbitrarily keeps v in either A or B.

```
Algorithm  Greedy-Max-Cut($G = (V, E)$)
  $A \leftarrow \emptyset$
  $B \leftarrow \emptyset$
  for $v \in V$ do
      if $deg_A(v) \geq deg_B(v)$ then                    $\triangleright$ $deg_X(v) = N(v) \cap X$
          $A \leftarrow A \cup \{v\}$
      else
          $B \leftarrow B \cup \{v\}$
  return $[A, B]$
```

The main problem about max cut is that it is not easy to come up with upper bounds on the optimum max cut.

Theorem 14. *The above greedy algorithm is a 2-approximation algorithm for max cut.*

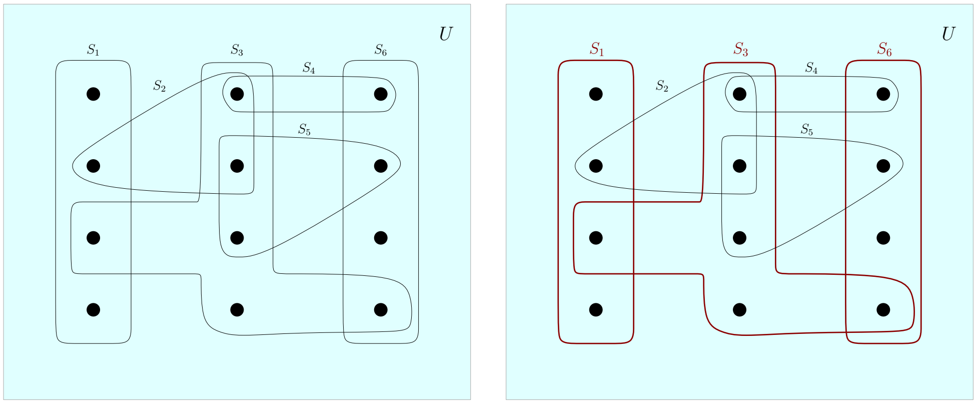
Proof. We prove that in every iteration the number of cut edges (among the vertices already assigned to A and B) is at least as large as the number of uncut edges. Initially both of these numbers are 0 and at every step some more edges are now cut and uncut, meaning some non-negative integers are added to both the number of cut edges and the number of uncut edges. By the greedy choice the number of cut edges always remain larger than the number of uncut edges.

The optimal solution might include at most all the edges of G, i.e. $|E|$. Our algorithm has the guarantee of number of cut edges is at least as large as the number of uncut edges, i.e. $f(s') \geq |E| - f(s') \implies f(s') \geq \frac{|E|}{2}$. Which proves the statement.

□

3.2 Set Cover Problem

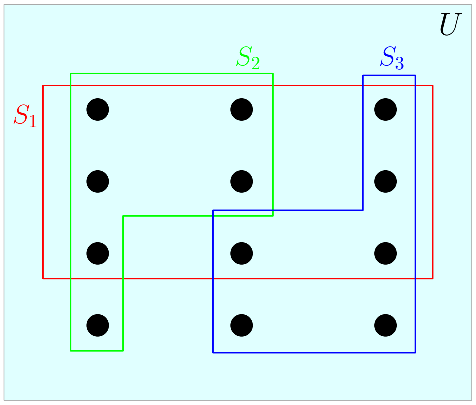
An instance of the set cover problem consists of a set U of n elements, a collection of subsets of U and $\mathcal{S} = \{S_1, S_2, \ldots, S_m\}$, where each $S_i \subseteq U$. A cover is a collection of subsets from $\mathcal{S}$ whose union is equal to U. Our goal is to find a cover with minimum number of subsets.



This is a very general optimization problem, that models for example the following scenario. Suppose we have m application softwares with different capabilities, U is the set of n capabilities we must have in our system. We want to choose the smallest (least cost) set of softwares that in total will meet our requirement specs.

Consider the following simple greedy algorithm for this problem. It Iterates over sets until all elements of U are covered. While there is an uncovered element choose a set S_i from $\mathcal{S}$ which covers the most number of (yet) uncovered elements.

```
Algorithm GREEDY-SET-COVER($U, \mathcal{S}$)
  $X \leftarrow U$                                          ▷ Yet uncovered elements
  $C \leftarrow \emptyset$
  while $X \neq \emptyset$ do
      Select an $S_i \in \mathcal{S}$ that maximizes $|S_i \cap X|$        ▷ Cover most elements
      $C \leftarrow C \cup S_i$
      $X \leftarrow X \setminus S_i$
  return $C$
```



- The algorithm will select S_1, S_2, and S_3. While optimal is S_2 and S_3

Here is another run of the GREEDY-SET-COVER on the following instance.

- $U = \{1, 2, 3, 4, 5\}$ and
- $\mathcal{S} = \{\{1, 2\}, \{1\}, \{1, 4\}, \{4\}, \{1, 2, 3, 5\}, \{4, 5\}\}$

We will first pick $\{1, 2, 3, 5\}$ as this cover 4 elements, next we can pick $\{1, 4\}$ or $\{3, 4\}$ or $\{4, 5\}$. Either of them will cover all elements of U. Hence this algorithm picks two sets to cover all element of U, which clearly is the best possible solution.

Consider the following example.

- $U = \{1, 2, 3, 4, 5, 6, 7, 8\}$
- $\mathcal{S} = \{\{1, 2, 3, 4, 5\}, \{6\}, \{7, 8\}, \{1, 2, 3, 8\}, \{4, 5, 6, 7\}\}$

Algorithm A will pick $\{1, 2, 3, 4, 5\}$, $\{4, 5, 6, 7\}$ and $\{1, 2, 3, 8\}$. while the best solution clearly is $\{1, 2, 3, 8\}$ and $\{4, 5, 6, 7\}$.

Hence GREEDY-SET-COVER might not always produce an optimal solution. We prove a performance guarantee for GREEDY-SET-COVER.

Theorem 15. *The above greedy algorithm is a* $\log n$*-approximation algorithm for the set cover problem (where* $n = |U|$*).*

Proof. Let $f(s) = OPT = k$, i.e. there exists k sets in $\mathcal{S}$ that cover all elements of U.

By the pigeon-hole principles, this immediately implies that there exists a set $S_i \in \mathcal{S}$ that covers at least $\frac{n}{k}$ elements. Hence the first set that our algorithm will pick have at least $\frac{n}{k}$ elements. Let n_1 be the number of elements that remain uncovered after the first set is picked. We get that $n_1 \leq n - \frac{n}{k} = n(1 - \frac{1}{k})$. Again by the pigeon-hole principle one of the remaining sets in $\mathcal{S}$ must contain at least $\frac{n_1}{k-1}$ of the uncovered elements, as otherwise the optimal solution would have to contain more than k sets. Our greedy algorithm will pick the largest such set and the number of remaining uncovered elements is at most $n_2 \le n_1 - \frac{n_1}{k-1} = n_1(1 - \frac{1}{k-1}) \le n(1 - \frac{1}{k})(1 - \frac{1}{k-1}) \le n(1 - \frac{1}{k})^2$.

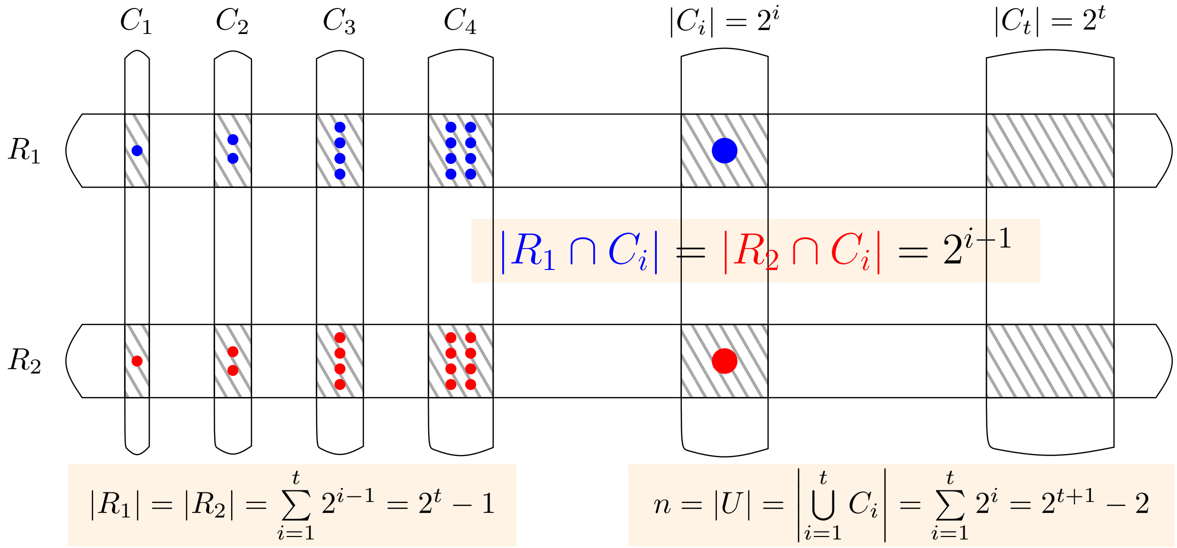
In general after the iteration, for the number of remaining uncovered elements we have $n_i \le n(1 - \frac{1}{k})^i$.

Let us see in how many iterations does this number goes below 1 (at which point we have a cover) and this is the number of sets we have. The following calculation gives us a bound on the number of iterations.

$$\begin{aligned} n_i \le n\left(1 - \frac{1}{k}\right)^i &< 1 \\ \left(1 - \frac{1}{k}\right)^{\frac{ki}{k}} &< \frac{1}{n} \\ e^{-\frac{i}{k}} &< \frac{1}{n} && \text{using the fact that } (1-x)^{\frac{1}{x}} \sim e^{-1} \\ \frac{i}{k} &< \ln n \\ i &< k \ln n \end{aligned}$$

This implies that the number of set we choose $i \le k \ln n = \ln n f(s)$, hence this algorithm is $\ln n$-approximate. □

This analysis is tight in the sense, that there are instances on which the GREEDY-SET-COVER indeed selects a cover of size $\log n \cdot \text{OPT}$.


C_1
C_2
C_3
C_4
$|C_i| = 2^i$
$|C_t| = 2^t$
R_1
R_2
$|R_1 \cap C_i| = |R_2 \cap C_i| = 2^{i-1}$
$|R_1| = |R_2| = \sum_{i=1}^{t} 2^{i-1} = 2^t - 1$
$n = |U| = \left|\bigcup_{i=1}^{t} C_i\right| = \sum_{i=1}^{t} 2^i = 2^{t+1} - 2$

- GREEDY-SET-COVER selects $C_t, C_{t-1}, \ldots, C_1$
- The optimal solution is R_1 and R_2
- On this example, the algorithm achieves approximation factor is $O(\log n)$

The natural question that every algorithm designer ought to ask at this point, is can we do better? Meaning can we design another algorithm that will outperform GREEDY-SET-COVER. Unfortunately the answer to this question is No, i.e. a lower bound is known for this problem. The theorem is

Theorem 16. *Unless* P = NP*, there is no approximation algorithm for* SET-COVER *problem that achieves an approximation ratio better than* $\log n$

We discuss inapproximability results later. However we can do much better for a special instance of set cover, namely the vertex cover problem.

3.3 Vertex Cover

In the vertex cover problem, given a graph $G = (V, E)$, out goal is to find the smallest subset of vertices such that for every edge at least one end point is selected. We studied it earlier and show that it was a special case of set cover problem. We can of course use the above set cover algorithm (at each step choose a vertex that covers the maximum number of remaining uncovered edges) to get a performance guarantee of $\ln |E| = O(\log n)$.

Consider the following instance of set cover problem (which is exactly the vertex cover problem for a given graph $G = (V, E)$.

- $U = \{e_1, \ldots, e_m\}$, the set of edges in G that we want to cover.
- $\mathcal{S} = \{E_1, E_2, \ldots, E_n\}$ where each $E_i \subset U$, is the set of edge incident on v_i. i.e. $E_i = \{e : e \text{ is incident on } v_i\}$. One can think of them as the set of edges covered by v_i.

A natural greedy algorithm for the vertex cover problem would be as follows: While there is an uncovered edge, choose one of its two endpoints to cover it. Clearly one of the two endpoints is in the optimal cover, but we can be unfortunate at every step and choose the wrong one at every step. Consider the following graph.

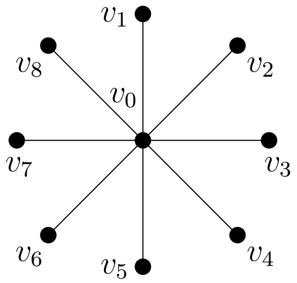


Figure 8: The optimal vertex cover is $\{v_0\}$ while to cover each edge we might choose all other vertices.

But we can modify the first greedy algorithm to improve the approximation guarantee. This algorithm proceeds as follows. If there is an uncovered edge $e = (u, v)$ choose both of its endpoints to the cover.

```
Algorithm  VERTEX-COVER($G$)
  $C \leftarrow \emptyset$
  while $E \neq \emptyset$ do
      pick any $\{u, v\} \in E$
      $C \leftarrow C \cup \{u, v\}$
      Remove all edges incident to either $u$ or $v$
  return $C$
```

Theorem 17. *The greedy algorithm that repeatedly picks both endpoints of an uncovered edge that remains is* 2*-approximate.*

Proof. Since for each edge $e = (u, v)$, the optimal solution must include either u or v, while this algorithm at worst picks both u and v. Hence if s' is solution by this algorithm and s is the optimal solution then $f(s') \leq 2f(s)$. □

Actually we might actually do this bad, consider the example in Figure 9

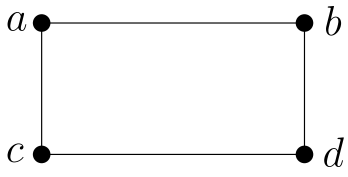


Figure 9: To cover the edge (a, b) we will choose a and b, while to cover (c, d), we will choose c and d, while an optimal solution is $\{a, d\}$

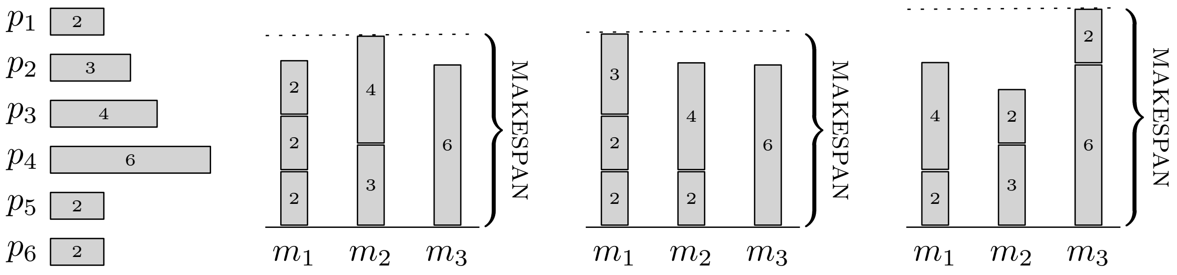
Remark 18. *The best known algorithm for vertex cover has an approximation guarantee of* $2 - O(\log\log n/\log n)$*, while the best known lower bound is* $4/3$*. To close the gap is an open problem.*

3.4 Scheduling on Parallel Machines

This is a general problem of load balancing a classic NP-hard problem that has many applications. An instance of such a scheduling problem consists of

- P: Set of n jobs (processes) $\{p_1, p_2, \ldots, p_n\}$
- Each job p_i has a time t_i
- M: Set of k parallel machines $\{m_1, \ldots, m_m\}$

We want to schedule these processes, (assign jobs to machines). Let $A(j)$ be the set of jobs assigned to m_j. The load of a machine T_j is the total time of processes assigned to it, i.e. $T_j = \sum_{p_i \in A(j)} t_i$. The makespan of a schedule (an assignment of jobs to machine), $T(P, M)$ is the maximum load of any machine, i.e. $T(P, M) = \max_j T_j = \max_j \sum_{p_i \in A(j)} t_i$.



3.4.1 List scheduling algorithm

We give a simple greedy algorithm for this problem. This algorithm iterates over each process and assign p_i to a machine that currently has the lowest load, i.e. assign each job to a least loaded machine.

```
Algorithm GREEDY LOAD BALANCE($\mathcal{G}$)
  for $j = 1$ to $k$ do
      $A(j) \leftarrow \emptyset$
      $T_j \leftarrow 0$
  for $i = 1$ to $n$ do
      Let $m_j$ be a machine that has the minimum load at this time, i.e. $m_j = \min_k T_k$
      $A(j) \leftarrow A(j) \cup p_i$
      $T_j \leftarrow T_j + t_i$
```

Consider the following example of 6 jobs and 3 machines illustrated in Figure 10. If the order of jobs was $2, 3, 4, 6, 2, 2$ the above greedy algorithm will schedule them as in Figure 11 for a total makespan of 8 ($T_1 = 8$). This is clearly not optimal if the job order was $6, 4, 3, 2, 2, 2$ the scheduling would have been as in Figure 12 for a makespan of 7, which is optimal.

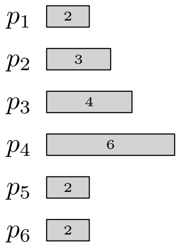


Figure 10: 6 jobs with times $2, 3, 4, 6, 2, 2$

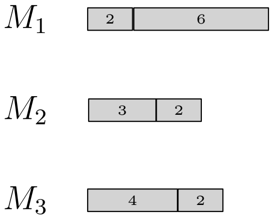


Figure 11: Schedule for order $2, 3, 4, 6, 2, 2$

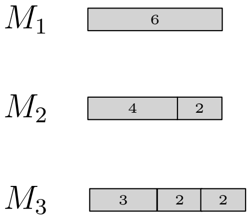


Figure 12: Schedule for order $6, 4, 3, 2, 2, 2$

Theorem 19. *The list scheduling algorithm is* $(2 - \frac{1}{k})$*-approximate, where* k *is the number of machines.*

Proof. First we need to derive a lower bound on $OPT(I)$ for a general instance I.

- We can lower bound $OPT(I)$ by considering the total processing time $\sum_i t_i$. By the pigeon-hole principle one of the k machines must do at least $\frac{\sum_i t_i}{k}$ amount of total work (if every one does less than $1/k$ fraction of total work, then the total work done will be less than $\sum_i t_i$. So we get that

$$OPT(I) \geq \frac{\sum_i t_i}{k}.$$

- We also have the following obvious lower bound on $OPT(I)$. Since the machine to which the maximum time consuming process t_{max} will take at least t_{max} time to finish, hence the makespan will be at least t_{max}. We get

  $$OPT(I) \geq \max_i t_i = t_{max}.$$

  In other words it gives us that

  $$\forall\, p_i \;\; OPT(I) \geq t_i.$$

WLOG assume that by the above algorithm, machine m_1 has the maximum load, let this load be $c_{max} = T_1$. Let p_i be the last job placed at machine m_1. At the time p_i was assigned to m_1, by design the load of m_1 at that time was the minimum across alll machines at that time. Let L_1 be the load of m_1 at that time. Since we know that p_i is the last job placed at m_1, we get that $L_1 = T_1 - t_i$. Since this is the least loaded machine, at that time all other machines must have load at least $T_1 - t_i$ too.

Adding up the load of all machines we get that

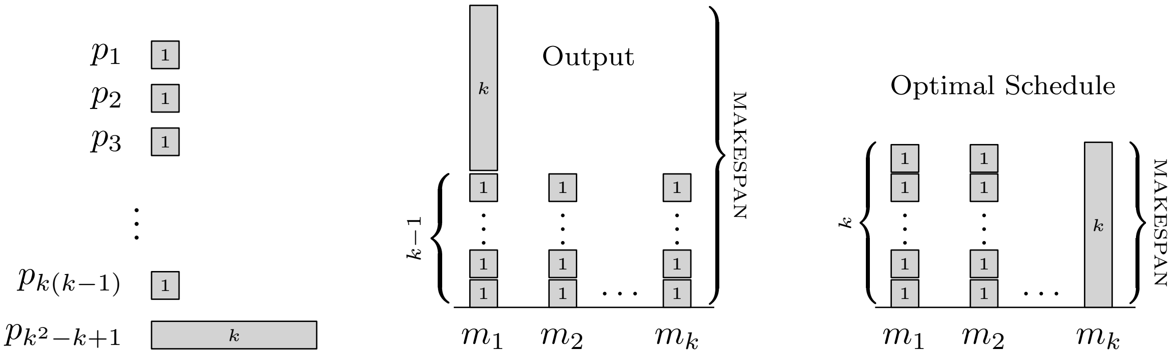
$$\sum_j T_j \geq k(T_1 - t_i) + t_i.$$

The quantity on the right hand side is exactly the total time of all the jobs (since very job is assigned to exactly one machine), combining it with our first lower bound we get that

$$
\begin{aligned}
kOPT(I) &\geq \sum_i t_i = \sum_j T_j \geq k(T_1 - t_i) + t_i \\
kOPT(I) &\geq k(T_1 - t_i) + t_i \\
OPT(I) &\geq (T_1 - t_i) + \frac{t_i}{k} \\
OPT(I) &\geq T_1 - \left(1 - \frac{1}{k}\right) t_i \\
OPT(I) &\geq T_1 - \left(1 - \frac{1}{k}\right) OPT(I) \qquad \text{using the second lower bound} \\
T_1 &\leq \left(2 - \frac{1}{k}\right) OPT(I)
\end{aligned}
$$

Hence the approximation ratio of the above list scheduling algorithm is $\left(2 - \frac{1}{k}\right)$, where k is the number of machines. □

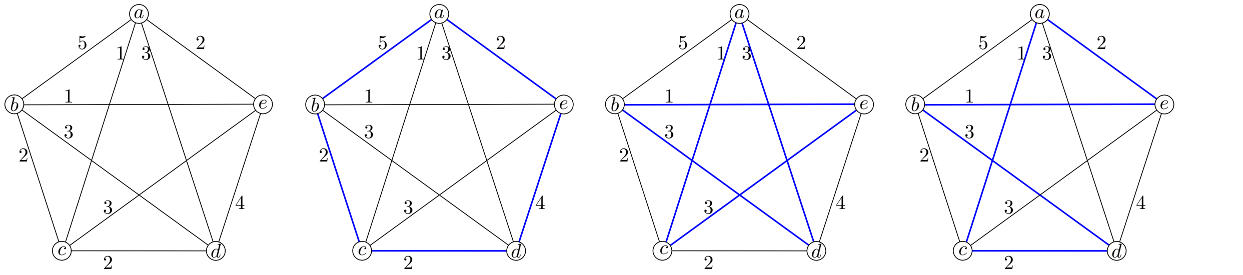
We give an example to show that the above approximation guarantee is tight for this algorithm.



Let $n = k(k-1)+1$, let the first $n-1$ jobs have runtime of 1 and the last job job has runtime k. In other words for $1 \leq i \leq n-1$, $t_i = 1$, while $t_n = k$. It is easy to see that $OPT(I)$ for this I is k, (i.e. assign p_n to m_1 and distribute the remaining $k(k-1)$ jobs equally among the remaining $k-1$ machines.) Think about what our algorithm will do in this case, it balances the first $n-1$ jobs among the k machines and then assign the giant job to one of the machines, resulting in makespan of $2k-1$. This achieves equality in the above upper bound.

4 The TSP Problem

Recall that, given a complete graph G on n vertices with edge weights $w : E \mapsto \mathbb{R}$, a TSP tour is a Hamiltonian cycle in G. The TSP problem is to find a minimum length TSP tour in G.



K_5 with edge weights

A TSP tour of length 15

A TSP tour of length 11

A TSP tour of length 9

Figure 13: Examples of TSP tours

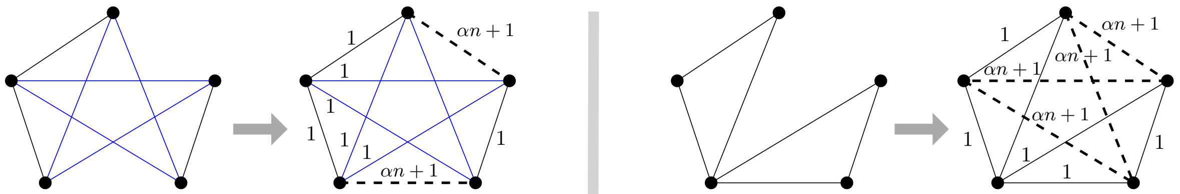
4.1 Impossibility of Relative Approximation

Theorem 20. *If* P $\neq$ NP*, then for any* $\alpha > 1$*, there is no* α*-approximation for* TSP

Proof. We prove this by reducing the HAMILTONIAN-CYCLE(G) to the α-APPROXIMATE-TSP(G) problem. Consider an instance $G = (V, E)$ of HAM-CYCLE(G), $|V| = n$. Next, construct a complete graph G' on n vertices with weights as follows:

$$w(v_i, v_j) \;=\; \begin{cases} 1 & \text{if } (v_i, v_j) \in E(G) \\ \alpha n + 1 & \text{else} \end{cases}$$

Now, if there is a Hamiltoninan Cycle in G, then the same cycle is a TSP tour T in G'. Note that T uses all edges (from G) of weight 1 and is of length n. Thus, T is an α-approximate tour in G'. If there is no Hamiltonian Cycle in G, then any TSP tour T in G' must use an edge of weight $\alpha n + 1$. Thus, T is an α-approximate tour in G'.



Hamiltonian cycle in G shown in blue

TSP tour in G' of length $5 = n$ shown in blue

No Hamiltonian cycle in G

Any TSP tour must use an edge of weight $\alpha n + 1$

Figure 14: Reducing the HAMILTONIAN-CYCLE(G) to the α-APPROXIMATE-TSP(G) problem

Suppose there is an algorithm $\mathcal{A}$ for the α-APPROXIMATE-TSP(G) problem. For an instance $G = (V, E)$ of HAM-CYCLE(G), $|V| = n$, make $K_n = G'$ with

$$w(v_i, v_j) \;=\; \begin{cases} 1 & \text{if } (v_i, v_j) \in E(G) \\ \alpha n + 1 & \text{else} \end{cases}$$

If G has a HAM-CYCLE, then OPT-TSP(G') $=$ n. If G has no HAM-CYCLE, then OPT-TSP(G') $\geq$ $\alpha n+1$ We have used $\mathcal{A}$ to solve HAM-CYCLE(G) in polynomial time, which implies $P = NP$. This contradicts out assumption that $P \neq NP$.

□

4.2 Metric TSP

Before we define METRIC-TSP, recall the properties of the distance metric and metric space.

A distance $d(u, v)$ is a **distance metric** if it satisfies the following 4 axioms

1. Non-negativity: $d(u, v) \;\geq\; 0$
2. Indiscernibility: $d(u, v) \;=\; 0 \;\;\Leftrightarrow u = v$
3. Symmetry: $d(u, v) \;=\; d(v, u)$
4. Triangle Inequality: $d(u, w) \;\leq\; d(u, v) + d(v, w)$

Now we can define Metric TSP as: Given a complete graph G on n vertices with **metric edge weights** $w : E \mapsto \mathbb{R}^+$, a TSP tour is a Hamiltonian cycle in G. Note that for all vertices $x, y, z \quad w(x, z) \leq w(x, y) + w(y, z)$ For a Euclidean TSP vertices are points in a plane and distance is the Euclidean distance

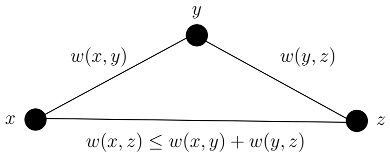
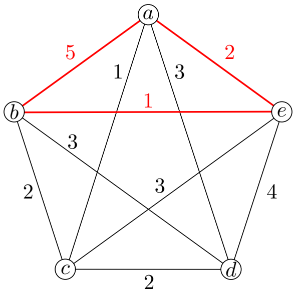
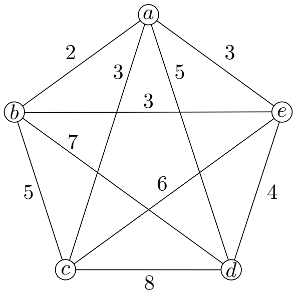


Figure 15: Direct distance is shorter than the distance via an intermediate point



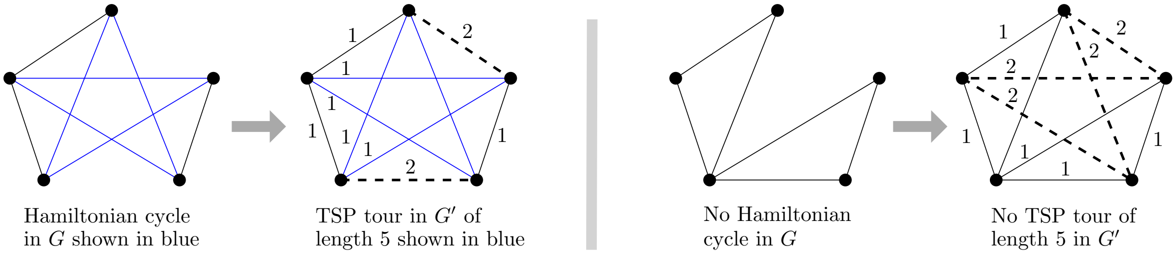
Not a METRIC-TSP instance



A METRIC-TSP instance

We can show that metric TSP is NP-HARD by reducing HAM-CYCLE(G) to METRIC-TSP(G', k) where G' is a weighted graph and k is the required length of the TSP tour. We construct G' as follows. Given an instance $G = (V, E)$ of HAM-CYCLE(G) where $|V| = n$, make a complete graph on n vertices G' with weights defined such that they induce a distance metric.

$$w(v_i, v_j) = \begin{cases} 1 & \text{if } (v_i, v_j) \in E(G) \\ 2 & \text{else} \end{cases}$$



Hamiltonian cycle in G shown in blue

TSP tour in G' of length 5 shown in blue

No Hamiltonian cycle in G

No TSP tour of length 5 in G'

Now, if there is a Hamiltoninan Cycle in G, then the same cycle is a TSP tour in G' which uses all edges (from G) of weight 1 and is of length n. If there is no Hamiltonian Cycle in G, then any TSP tour T in G' must use an edge of weight 2 and the length of the tour is greater than n. Therefore, G has a Hamiltonian cycle if and only if G' has a TSP tour of length $k = n$.

4.3 2-approximation for Metric TSP

We observe a simple lower bound on METRIC-TSP:

Theorem 21. *If C is* HAM-CYCLE *and T^* is a* MST *in G, then $w(T^*) \leq w(C)$*

Proof. Let $e \in C$ be any edge. Then, $T = C \setminus e$ is a spanning tree in $G \triangleright C$ is a HAM-CYCLE. Since T^* is a minimum spanning tree, $w(T^*) \leq w(T) \leq w(C \setminus e) \leq w(C)$.

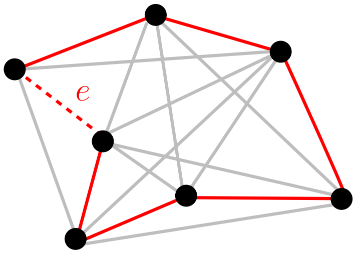


Figure 16: A Hamiltonian Cycle C can be obtained by adding an edge to the MST T^*

.

□

Before we use this to show a 2-approximation algorithm for TSP, let's recall Eulerian Graphs. A **Euler Circuit:** is a closed walk in graph G containing every edge of G. A **Euler Path** is a walk in G containing every edge of G.

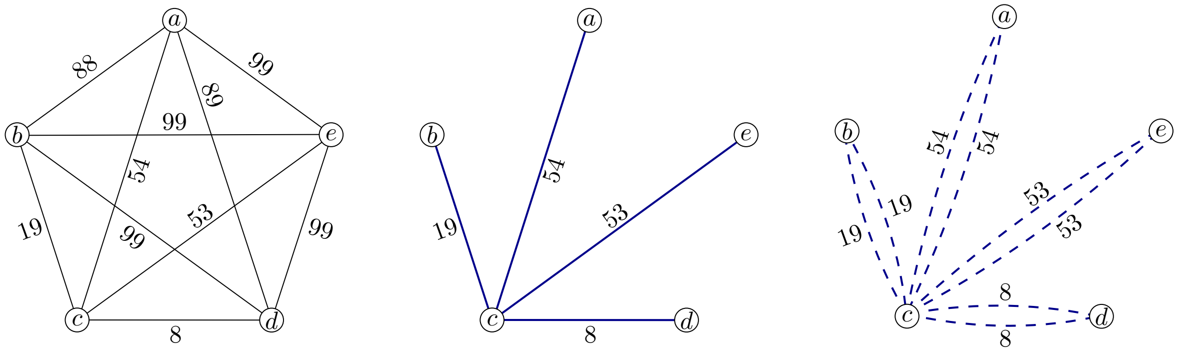
Theorem 22. *G has an Euler circuit if and only if every vertex has even degree*

Theorem 23. *G has an Euler path if and only if it has exactly two vertices of odd degree*

We use a spanning tree T to find a TSP tour C on a METRIC-TSP instance G. Suppose each edge in T is duplicated, i.e. can be used twice. Consider some vertex s to be the root of the T.

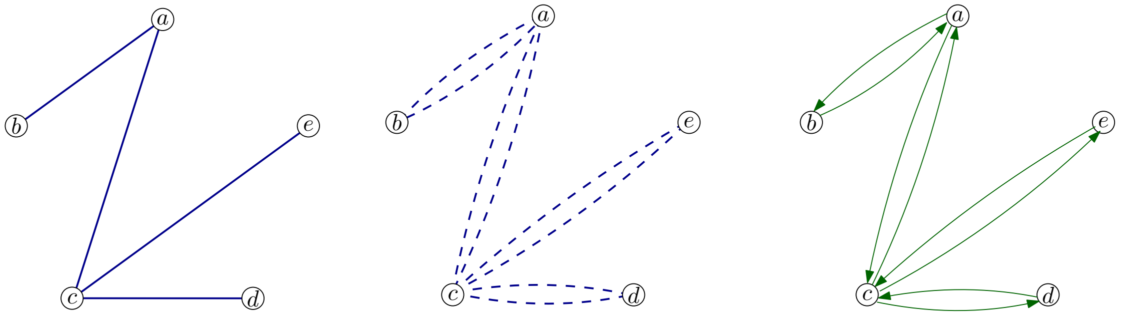
Let L be an Euler tour on $T^{(*)}$ starting from s (the root). We list vertices in order of L including repetitions. The length of L is $w(L) = \sum_{e \in L} w(e)$.

Let C^* be an optimal TSP tour in G. L is not equal to C^* since C^* must visit each vertex only once except the first. To convert L to a TSP tour C, we remove duplicate vertices, while the retaining the first and last vertex, by **short-circuiting** L. We do this by traversing L and keeping only the first occurrence of a vertex. When a vertex



A metric-TSP instance An MST T T rooted at c and edges duplicated

Figure 17: Duplicating each edge in MST T in G



An MST T T rooted at d and edges duplicated L d, c, e, c, a, b, a, c, d

Figure 18: An Euler tour on MST $T^{(*)}$ in G

is about to be revisited, we skip it and simply visit the last vertex. Only the repeated root at the last vertex is retained to complete the cycle C.

The algorithm can be summarized as follows:

```
Algorithm DOUBLE-TREE-TSP($G$)
  $T \leftarrow$ MST($G$)                              ▷ e.g. KRUSKAL algorithm
  $T^{(*)} \leftarrow$ DUPLICATE edges of $T$            ▷ every vertex has even degree
  $L \leftarrow$ EULER tour of $T^{(*)}$                 ▷ Fleury or Hierholzer algorithm
  $C \leftarrow$ SHORT-CIRCUIT($L$)
  return $C$
```

Runtime of each step in the above algorithm is polynomial: Kruskal's algorithm takes

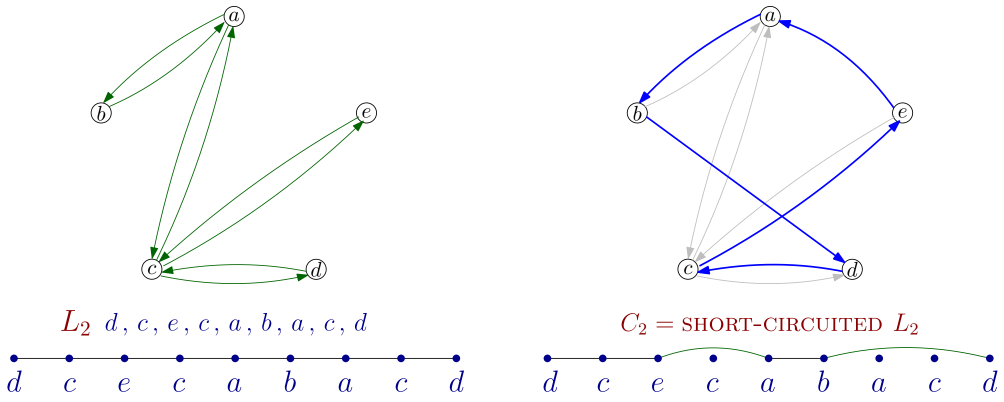

a
b
e
c
d
L_2 d, c, e, c, a, b, a, c, d
d c e c a b a c d
a
b
e
c
d
C_2 = SHORT-CIRCUITED L_2
d c e c a b a c d

Figure 19: Short-Circuiting L to obtain C

$O(|E| \log n)$, duplicating edges is $O(n)$ while Euler tour can be obtained in $O(|E|^2)$ using Fleury's algorithm. and in $O(|E|)$ using Hierholzer's algorithm. Finally, short circuiting takes $O(n)$. Therefore, the overall runtime is clearly polynomial.

Theorem 24. *The* DOUBLE-TREE-TSP *is a* 2*-approximation for* METRIC-TSP

Proof. Since edges were duplicated in T to obtain a Euler tour L on T, $w(L) = 2w(T)$. Then, since $w(T) \leq w(C)$, $w(L) \leq 2w(C^*)$. Since edges were only removed during short circuiting, and edge weights in T (or G) are a distance metric, $w(C) \leq w(L)$.

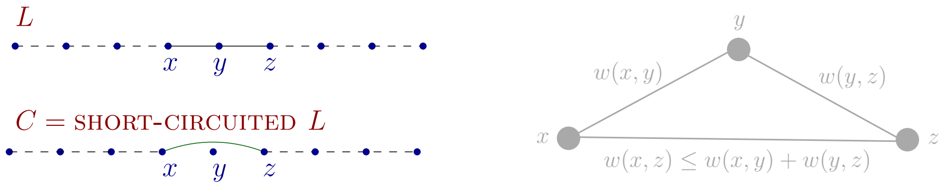


Figure 20: Removing duplicate vertices during short-circuiting

Therefore, we can conclude that

$$w(C) \;\leq\; w(L) \;=\; 2w(T) \;\leq\; 2w(C^*)$$

□

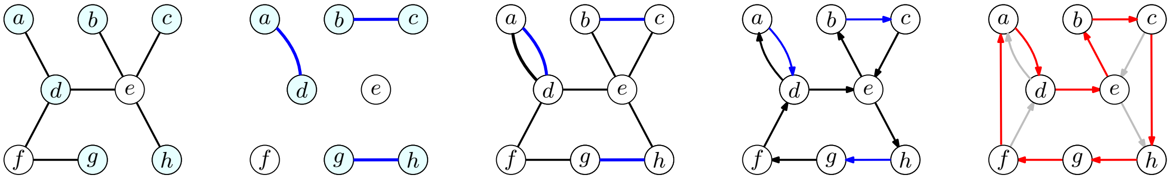
Can we do any better? The factor 2 in the DOUBLE-TREE-TSPalgorithm appeared because of duplicating all edges of T since we needed all vertices to have an even degree for an Euler tour. Therefore, we do not need to alter vertices which already have an even degree. The interesting question now is, is there a less costly way to make degrees of select vertices even? We see that we can indeed do so in the Christofides' Algorithm which gives us a 1.5-approximation for METRIC-TSP.

4.4 Christofides Algorithm: 1.5-approximation for Metric TSP

We make the following observation about the MST T in G.

Theorem 25. *The number of odd-degree vertices in T is even.*

Let O be the set of odd degree vertices in T. By the above theorem, $|O|$ is even. Let M be the minimum cost perfect matching in subgraph in G induced by vertices in O. M can be obtained using Micali and Vazirani algorithm in $O(n^{2.5})$. In order to make all vertex degrees even, combine edges of M and T to get a multigraph H. Then, a Euler tour in L can be found and C can be obtained by short-circuiting L, similar to the DOUBLE-TREE-TSP algorithm.



Spanning tree T

Matching M among odd degree vertices

$H = M \cup T$

EULER tour L

C : SHORT-CUT L

Figure 21: Example: Obtaining HAM-CYCLE C from MST T in Christofides' Algorithm

```
Algorithm  CHRISTOFIDES-ALGO-TSP($G$)
  $T \leftarrow$ MST($G$)                                          ▷ e.g. KRUSKAL algorithm
  $G' \leftarrow$ Subgraph in $G$ induced by odd degreed vertices in $T$.
  $T^{(')} \leftarrow$ MIN-COST-PERFECT-MATCHING($G'$)  ▷ Micali & and Vazirani's Algorithm
  $O(n^{2.5})$
  $H \leftarrow M \cup T$                                ▷ every vertex has even degree in $H$
  $L \leftarrow$ EULER tour of $H$                          ▷ Fleury or Hierholzer algorithm
  $C \leftarrow$ SHORT-CIRCUIT($L$)
  return $C$
```

The runtime of above algorithm is evidently polynomial.

Theorem 26. *The Christofides' algorithm is a 1.5-approximation for* METRIC-TSP.

Proof. The length of Euler tour L on H is $w(L) = w(T) + w(M)$. Since $w(C) \leq w(L)$ and $w(T) \leq w(C*)$, $w(C) \leq w(C^*) + w(M)$.

First, we show that, given a HAM-CYCLE C in $G = (V, E)$ and $U \subseteq V$ with $|U|$ is even, for a min-cost perfect matching M on U, $w(M) \leq {}^1\!/_2 w(C)$.

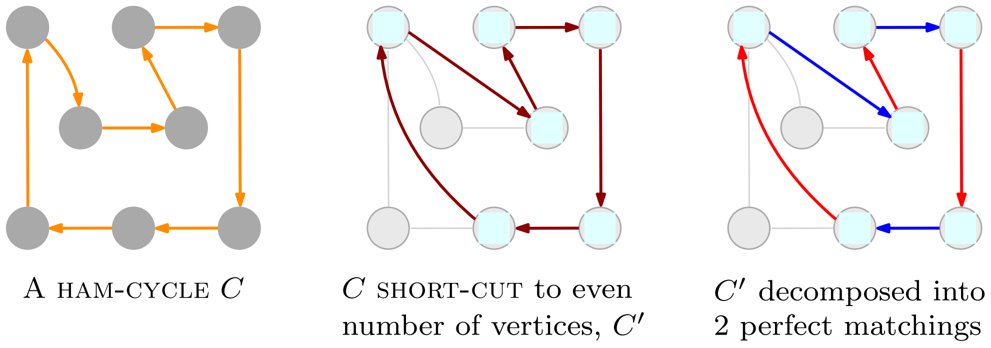

A HAM-CYCLE C
C SHORT-CUT to even number of vertices, C'
C' decomposed into 2 perfect matchings

Figure 22: If C is short-circuit C' on vertices in U, C' can be decomposed into two perfect matchings on U

Since M is a min-cost perfect matching $w(M) \leq {}^1\!/_2 w(C')$. Also note that $w(C') \leq w(C)$ by the triangle inequality. The above two facts imply that $w(M) \leq {}^1\!/_2 w(C)$.Furthermore, since we earlier saw that $w(C) \leq w(C^*) + w(M)$ and now given $w(M) \leq {}^1\!/_2 w(C')$, we can conclude that $w(C) \leq (1 + {}^1\!/_2) w(C^*)$.

□

5 The Knapsack Problem

Recall the knapsack problem, where an instance of it consists of

- Items $U = \{1, 2, \ldots, n\}$
- Weights (or sizes) $W = \{w_1, w_2, \ldots, w_n\}$, where w_i is the weight of item $i \in U$.
- Values (or profits) $V = \{p_1, p_w, \ldots, p_n\}$, where v_i is the value of item $i \in U$.
- Knapsack capacity (or budget) C.

A feasible solution to the problem is a subset $U' \subseteq U$, such that $\sum_{i \in U'} w_i \leq C$. Our goal is to maximize $f(U') = \sum_{i \in U'} v_i$.

Informally we would like to pack some items of different sizes into a knapsack of fixed capacity so as to collect the maximum total profit from the packed items.

In the following we will assume that all weights and values are integers.

5.1 A Greedy Algorithm for the Knapsack problem

Earlier we looked at some greedy approaches but they could result in arbitrarily bad solutions. A greedy approach could be to take the highest value item as long as the total does not go over capacity. Consider the instance Consider $C = 100$ and $V = W = \{51, 50, 50\}$; In this instance, we would select $\{51\}$ and stop, while clearly a better answer would be to take $\{50, 50\}$.

Another greedy approach would be to take the lowest weight items, so as to use up the least capacity. Consider the instance $C = 100$ and $V = W = \{1, 50, 50\}$; , in this case, we would select $\{1, 50\}$, whereas a better option was to take $\{50, 50\}$.

Yet another greedy algorithm is given as follows.

Algorithm GreedyByRation(G)

```
if $\sum_{i=1}^{n} w_i \leq C$ then                ▷ If all items fit in the sack, then take all
    $S \leftarrow U$
    return $S$
Sort items by $\frac{v_i}{w_i}$ into an array $S$      ▷ WLOG assume that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$
$Weight \leftarrow 0$                        ▷ We store the total weight collected so far in $Weight$
$Value \leftarrow 0$                          ▷ We store the total value collected so far in $Value$
$S \leftarrow \emptyset$                                        ▷ Initially the knapsack is empty
for $i = 1$ to $n$ do
    if $Weight + w_i < C$ then
        $S \leftarrow S \cup a_i$
        $Value \leftarrow Value + v_i$
        $Wt \leftarrow Weight + w_i$
```

It turns out that this algorithm too can be arbitrarily bad. Consider the following instance $W = \{1, C\}$ and $V = \{2, C\}$, so items have value to weight ratio as 2 and 1. So we will pick item item 1 first as its ratio is 2, but then there is no more capacity for the second item. While the optimal solution clearly is to take item 2. The ration v_i/w_i is called the density of items. The problem is that density is not necessarily a good measure of profitability. In the above example more dense item blocks the more profitable item.

We can fix this with an extremely simple trick. We also run another simple greedy algorithm that chooses the first item that GreedyByRatio algorithm misses. We return the best of the two algorithms. Following is a pseudocode for this algorithm.

Algorithm ModifiedGreedyByRation(G)

```
if $\sum_{i=1}^{n} w_i \leq C$ then                ▷ If all items fit in the sack, then take all
    $S \leftarrow U$
    return $S$
Sort items by $\frac{v_i}{w_i}$ into an array $S$    ▷ WLOG assume that $\frac{v_1}{w_1} \geq \frac{v_2}{w_2} \geq \ldots \geq \frac{v_n}{w_n}$
$Weight \leftarrow 0$                 ▷ We store the total weight collected so far in $Weight$
$Value \leftarrow 0$                  ▷ We store the total value collected so far in $Value$
$S \leftarrow \emptyset$                          ▷ Initially the knapsack is empty
for $i = 1$ to $n$ do
    if $Weight + w_i < C$ then
        $S \leftarrow S \cup a_i$
        $Value \leftarrow Value + v_i$
        $Wt \leftarrow Weight + w_i$
$k \leftarrow |S|$
if $Value \geq v_k$ then
    return $S$
else
    return $\{a_k\}$
```

Theorem 27. *The above algorithm is* 2*-approximate.*

Proof. Let $f(s')$ be the value of solution by this algorithm. Let k be the last item added to S by the algorithm (as in the code). Clearly we have that $\sum_{i=1}^{k} v_i \leq OPT$.

Let $c = \dfrac{C - (w_1 + w_2 + \ldots + w_k)}{w_{k+1}}$. The numerator is the remaining capacity of the knapsack after packing the first k items. So c is the fraction of the $(k+1)$st item that can be packed (if we were allowed to take fractions of item). **Note that** $c < 1$, because otherwise all of item $k+1$ can be packed and the algorithm would have packed it.

We get that

$$\sum_{i=1}^{k} v_i + c \cdot v_{k+1} \geq OPT, \tag{1}$$

by explanation of c above, (if we were allowed fractional packing) this packing consumes all the capacity of the Knapsack since and uses the capacity optimally (if fractional items were allowed), as we always selected items with largest density (largest value per unit capacity).

Actually we are going to refer to this fact later (a few times) so lets give it a name.

Lemma 28. *If a_k is the last item chosen by the ModifiedGreedyByRatio algorithm, then*

$$\sum_{i=1}^{k} v_i + c \cdot v_{k+1} \geq OPT,$$

where $c = \dfrac{C - (w_1 + w_2 + \ldots + w_k)}{w_{k+1}}$.
As an immediate corollary we get that $\sum_{i=1}^{k+1} v_i \geq OPT$*, as* $c < 1$.

Now our approximation guarantee follows. Since $f(s') = \max \left\{ \sum_{i=1}^{k} v_i, v_{k+1} \right\}$, if $\sum_{i=1}^{k} v_i < \frac{OPT}{2}$, then $v_{k+1} \geq \frac{OPT}{2}$, because otherwise using that fact $c < 1$, we get a contradiction to (1). □

This analysis is tight. Consider the following instance where the performance guarantee is matched. Let $U = \{a_1, a_2, a_3\}$, $V = \{1 + \frac{\epsilon}{2}, 1, 1\}$, $W = \{1 + \frac{\epsilon}{3}, 1, 1\}$, and $C = 2$.

The densities are given as $\{(1 + \epsilon/2)/(1 + \epsilon/3), 1/1, 1/1\}$. The above greedy algorithm will choose a_1 as this is equal to S (by the algorithm) and $v_1 > v_2$. While the optimal solution clearly is $\{a_2, a_3\}$. $f(s') = (1 + \epsilon/2)$, so $f(s) = 2 = 2f(s') - \epsilon$, which is arbitrarily close to 2 by choosing a sufficiently small ϵ.

Note that the runtime of this algorithm is $n \log n$ for sorting (Each of the n divisions takes time proportional to $\log(P.C)$, where P is the sum of alll values). So if we sacrifice quality of solution we brought down runtime to $n \log n$ from the dynamic programming pseudo-polynomial time algorithm that took $O(nC)$.

6 Polynomial Time Approximation Scheme

How good are the approximation algorithm we have seen so far. Consider the 2-approximate algorithm for vertex cover, while this is much better than applying the more general set cover approximate algorithm, but think about it, the solution we give is a 100% more than the optimal. The set cover solution is even worse $O(\log n)$ times more than the optimal.

An important fact about approximation algorithms and approximability is that while all NP-Complete problem are equivalent in terms of polynomial time solvability. But if $P \neq NP$, they differ substantially in terms of approximability. For instance we could find a 1-absolute approximate algorithm for planar graph coloring, while we proved that the independent set problem cannot have a k-absolute approximate algorithm.

While we didn't cover any inapproximability results for relative approximation algorithm, it can be shown that the set cover $O(\log n)$-approximate algorithm is essentially the best possible (assuming $P \neq NP$). Similarly, as we remarked earlier that vertex cover doesn't admit an approximation ratio better than 4/3. Inapproximability results are known for many other NP-complete problems too. Such results are part of a fascinating research area called hardness of approximation or lower bounds on approximability.

In the next section we will discuss algorithms that achieve any desired approximation ratio guarantee. Meaning the user inputs in addition to the problem instance the desired precision required and the algorithm guarantees to outputs a result within that error bound.

Definition 29. *A* **Polynomial Time Approximation Scheme (PTAS)** *is an approximation algorithm that takes as input in addition to the problem instance a parameter $\epsilon > 0$, and produces a solution that is $(1 \pm \epsilon)$-approximate. The running is polynomial in the size of the problem instance (generally n) its dependency on ϵ can be exponential however.*

- $(1 \pm \epsilon)$-approximate means that for minimization problem the value of solution is at most $(1 + \epsilon) \cdot OPT$ and that for maximization problem is at least $(1 - \epsilon) \cdot OPT$.
- Runtime of such algorithm could be for example $O(2^{1/\epsilon} n^3)$, or $O(n^{1/\epsilon})$, or

Definition 30. *A* **Fully Polynomial Time Approximation Scheme (PTAS)** *is a PTAS whose running time is polynomial in both n and $1/\epsilon$.*

6.1 PTAS for the Knapsack problem

First we make a few observation about the ModifiedGreedyByRatio algorithm and Lemma 28. Above we gave an instance where this algorithm actually about 1/2 the optimal solution. We argue that in some cases the solution by the algorithm is not very bad, we actually identify when is the solution bad.

Lemma 31. *If there is an $0 < \epsilon < 1/2$, such that for every items $w_i \leq \epsilon C$, then ModifiedGreedyByRatio algorithm gives a $(1-\epsilon)$ approximation.*

Proof. First since we sorted items by value to weight ratio, we have that
$\forall\ 1 \leq i \leq k+1,\ \dfrac{v_i}{w_i} \geq \dfrac{v_{k+1}}{w_{k+1}} \implies v_i \geq w_i \dfrac{v_{k+1}}{w_{k+1}}$.

Adding up all these inequalities we get that

$$v_1 + v_2 + \ldots + v_{k+1} \geq (w_1 + w_2 + \ldots + w_{k+1}) \frac{v_{k+1}}{w_{k+1}}$$
$$w_{k+1} \cdot \frac{v_1 + v_2 + \ldots + v_{k+1}}{w_1 + w_2 + \ldots + w_{k+1}} \geq v_{k+1}$$

By definition of k (the last item that the algorithm chose, actually a_{k+1} is first item that the algorithm rejected is the more important fact), we have that $w_1 + w_2 + \ldots + w_{k+1} > C$, plugging this in the above inequality we get that

$$v_{k+1} \leq \frac{w_{k+1}}{C} \cdot v_1 + v_2 + \ldots + v_{k+1}.$$

Using every $w_i \leq \epsilon C$, pluggin in $w_{k+1} \leq \epsilon C$, in the above inequality we get

$$\begin{aligned} v_{k+1} &\leq \epsilon \cdot (v_1 + v_2 + \ldots + v_{k+1}) \\ &\leq \epsilon \cdot (v_1 + v_2 + \ldots + v_k)/(1-\epsilon) \end{aligned}$$

Now if $(v_1 + v_2 + \ldots + v_k) \geq (1-\epsilon) \cdot OPT$, then we are done (got a $(1-\epsilon)$-approximation). If $(v_1 + v_2 + \ldots + v_k) < (1-\epsilon) \cdot OPT$, then we get from the above $v_{k+1} \leq \epsilon \cdot OPT$ (just substitute this in the above inequality for v_{k+1}.

Combining these two implies that $v_1 + v_2 + \ldots + v_k + v_{k+1} < (1-\epsilon) \cdot OPT + \epsilon \cdot OPT < OPT$, contradicting Lemma 28 (the corollary). Hence at least one of them must be at least $(1-\epsilon) \cdot OPT$.

□

Lemma 32. *If there is an $0 < \epsilon < 1/2$, such that for every items $v_i \leq \epsilon OPT$, then ModifiedGreedyByRatio algorithm gives a $(1 - \epsilon)$ approximation.*

Proof. By Lemma 28 (the corollary) we have that $(v_1 + v_2 + \ldots + v_{k+1}) \geq OPT$, while by the premise of this statement we have that $v_{k+1} \leq \epsilon OPT$, hence we must have that $(v_1 + v_2 + \ldots + v_k) \geq (1 - \epsilon) \cdot OPT$. Hence in this case ModifiedGreedyByRatio algorithm gives a $(1 - \epsilon)$ approximation.

□

We will next design a PTAS for the Knapsack problem (using ideas from the above two lemmas). First we state a simple but nonetheless very useful fact.

Fact 33. *In any optimal solution with total value OPT and any $0 < \epsilon < 1$, there are at most $\lceil \frac{1}{\epsilon} \rceil$ items with values at least $\epsilon \cdot OPT$.*

The above fact and lemma gives us the following idea for designing a PTAS. We will first try to guess the heavier items (values larger than $\epsilon \cdot OPT$) in the optimal solution, (by the above fact there aren't too many), then for the remaining items we will use our old ModifiedGreedyByRatio algorithm, by the lemma it will give us quite good solution (because the remaining items have values at most $\epsilon \cdot OPT$). The problem is how to guess the heavier items (actually since we don't know OPT, we can't even define heavy items). All we know is a bound on their number. But that's good enough information, since there can only be $\lceil \frac{1}{\epsilon} \rceil$ of them. We will try all subsets of U of sizes at most $\lceil \frac{1}{\epsilon} \rceil$. There are at most $n^{\lceil \frac{1}{\epsilon} \rceil + 1}$ subsets of U of size at most $\lceil \frac{1}{\epsilon} \rceil$.

Here is the algorithm. For a set $S \subseteq U$ we define $w(S) = \sum_{i \in S} w_i$ and $v(S) = \sum_{i \in S} v_i$ (the total weight and value of items in S).

Algorithm KnapsackPTAS(G)

```
$h \leftarrow \lceil \frac{1}{\epsilon} \rceil$
$currentMax \leftarrow 0$
for each $H \subseteq U$, such that $|H| \leq h$ do
    Pack $H$ in knapsack with capacity $C$ (if possible)
    Let $v_m$ be the minimum value of any item in $H$
    Let $H'$ be items in $U \setminus H$ with value greater than $v_m$
    Run ModifiedGreedyByRatio algorithm on $U \setminus \{H \cup H'\}$ with capacity $C - w(H)$
    Let $S$ be the solution returned
    if $currentMax < v(H) + v(S)$ then
        $currentMax \leftarrow v(H) + v(S)$
```

We can easily keep track of the best solution too, by keeping the current best pair of H and S.

Theorem 34. *The above algorithm is a PTAS for the knapsack problem, with runtime* $n^{\lceil \frac{1}{\epsilon} \rceil + 1}$

Proof. There are $O(n^{h+1})$ subsets of U of size at lest h. For each subset we do a linear amount of work and then call the ModifiedGreedyByRatio algorithm. Note that we do sorting only once, but that is immaterial as it is dominated by the above $O(n^h)$ term. $O(n^{h+1}) = O(n^{\lceil \frac{1}{\epsilon} \rceil + 1})$, hence it is polynomial in n (and exponential in $1/\epsilon$).

For the approximation ratio, note that since we are iterating over all subsets of size at most h, so in one of these iterations we will actually consider the correct set H (which the optimal solution has) (we know it cannot have more than h items of value more than $\epsilon \cdot OPT$). Consider that iteration. $U' = U \setminus \{H \cup H'\}$, Let OPT' be the optimal way to pack items in U'. So $OPT = v(H) + OPT'$. In this iteration we removed all the items that have value at least larger than $\epsilon \cdot OPT$, because by the fact above they cannot be part of the solution (OPT'). Since for every item in U', $v_i \leq \epsilon \cdot OPT$, by the above lemma we can get a solution that is at least $(1 - \epsilon)OPT'$. Hence our solution is $v(H) + (1 - \epsilon)OPT' \geq (1 - \epsilon)OPT$ ($v(H)$ could be 0, the optimal solution might not include any heavier item).

□

7 FPTAS for the Knapsack problem

We first develop a dynamic programming solution to solve the knapsack problem, which we use to develop a FPTAS for the problem.

7.1 Dynamic programming solution

We discussed a dynamic programming solution in great length in the class, see lecture notes for dynamic programming. We quickly review that solution and identify problems with that we solve in the next section, by developing a more scaling friendly dynamic programming solution.

Given an instance (U, V, W, C) of the knapsack problem, where $|U| = |V| = |W| = n$, $V, W \subset \mathbb{Z}$ and $C \in \mathbb{Z}$. Let $OPT(i, c)$ and $\mathcal{O}(i, c)$ be the value of the optimal solution and the optimal solution (the actual subset) respectively, of the (sub)problem $U = \{a_1, \dots, a_i\}$, $W = \{w_1, \dots, w_i\}$ and $V = \{v_1, \dots, v_i\}$ and capacity c. We discussed the optimal substructure property and got the following recurrence

$$OPT(i, c) = \begin{cases} 0 & \text{if } c \leq 0 \\ 0 & \text{if } i = 0 \\ \max\{OPT(i-1, c-w_i) + v_i, OPT(i-1, c)\} & \text{else} \end{cases}$$

As we discussed earlier we can find $OPT(n, C)$ (and $\mathcal{O}(n, C)$), which is our goal exactly in time $O(nC)$. This is not polynomial in the size of the input as unless we are using a unary system C can be described in $O(\log n)$ bits.

But can we use this solution to design a FPTAS? Earlier we saw that if all weights are not very large ($\leq \epsilon C$) then we get $(1-\epsilon)$-approximation. So we can scale down all the weights and B and solve the problem exactly using this algorithm and then scale up the solution. There could be a problem; during scaling up we might end up with an infeasible solution (violating the capacity constraint), which is a hard constraint. In the following we give another dynamic programming formulation which is more scaling friendly. That solution actually scales all the values (and use the second lemma, when all values are small), here we can scale up with care so as to not end up with infeasible solution, but value is kind of a soft constraint (it is in our objective function and we are allowed to make bounded error).

7.2 Scaling Friendly Dynamic Programming

We want to develop an algorithm with running time depending on n and $P = \sum_i v_i$ and as we mentioned earlier scaling P up and down is relatively in our control, so it will be easy to use it as a subroutine for a FPTAS. Recall that all of the values are all integers. For the previous dynamic programming solution we essentially answered the question "what is the maximum value that we can gain if capacity is c"? Here we turn this question to "what is the minimum weight that we need if want to gain a value of p"? Let's define $\overline{OPT}(i,v)$ to be the smallest capacity needed to get the target value v from the sub-instance $U = \{a_1,\ldots,a_i\}$, $W = \{w_1,\ldots,w_i\}$ and $V = \{v_1,\ldots,v_i\}$. The maximum target one can achieve is at most $P = \sum_i v_i$, we we have to compute a $\overline{OPT}(i,v)$ for each $0 \le i \le n$ and $0 \le v \le P$. Let v_m be the maximum value of any item, then $P \le nv_m$, the total number of subproblems are at most $O(n \cdot nv_m)$.

Remark 35. *Remember none of these subproblems is our actual goal, but if we have solution to these subproblems, then we can compute a solution to our problem instance. The solution to our instance is the largest v such that $\overline{OPT}(i,v) \le C$.*

It is easy to see that the following recurrence can be used to solve these subproblems.

$$\overline{OPT}i,v = \begin{cases} 0 & \text{if } v = 0 \\ \infty & \text{if } i = 0 \text{ and } v > 0 \\ \overline{OPT}i-1,v & \text{if } i \ge 1 \text{ and } 1 \le v < v_i \\ \min\{\overline{OPT}(i-1,v), \overline{OPT}(i-1,p-v_i)+w_i & \text{if } i \ge 1 \text{ and } v \ge v_i \end{cases}$$

This recurrence might not be clear immediately so think about it until it is. Try to formally prove it, it just uses the definition of $\overline{OPT}(i,v)$. You may consult your textbook for a formal proof.

As usual this immediately gives us a dynamic programming algorithm, (a bottom-up) iterative procedure that runs in $O(n^2 v_m)$ that in view of the above remark computes $OPT(n,C)$. Note that this too is a pseudo-polynomial time algorithm but as we will see below we have achieved a lot.

7.3 Scaling and Rounding

Now we use the above dynamic programming solution to design a FPTAS for the knapsack problem. We cannot directly use the above procedure if the values are too large (or they are not integers). Here are some immediate observations.

- If in our instance v_m is small (polynomial in n, say $v_m = n^k$), this means all values are small, then we can run the above dynamic programming solution to solve the problem optimally in polynomial time $O(n^2 v_m) = O(n^2 n^k)$.

- If the values are large, then we will scale them down, as we don't have to deal with them exactly, since we are only seeking an approximate solution.

We are going to scale down all the values so they are not too large and also round them so they are integral. This scaling and rounding will introduce some error, because the algorithm may prefer some subset over another erroneously as the exact values are unknown. We, therefore, want to do scaling and rounding such that the error is small (bounded). Since we want to get a $(1-\epsilon)$-approximation we want the error to be no more than $\epsilon \cdot OPT$.

Let $b = \frac{\epsilon}{n} \cdot OPT$. Suppose we change every value v_i to v_i' such that v_i' is the smallest integers such that $v_i \leq v_i' \cdot b$. The number v_i' satisfying the above bound is given by

$$v_i' = \left\lceil \frac{v_i}{b} \right\rceil .$$

There are two things to notice about this transformation.

1. if $v_i \leq v_j$, then $v_i' < v_j'$ for all $1 \leq i, j \leq n$
2. lets see how large can the new values be? let v_m' be the maximum new values, (By the above property we know that this corresponds to v_m). We know that $OPT \geq v_m$, so

   $$v_m' \leq \left\lceil \frac{v_m}{b} \right\rceil = \left\lceil \frac{v_m}{\epsilon/n \cdot OPT} \right\rceil \leq \left\lceil \frac{n \cdot v_m}{\epsilon \cdot v_m} \right\rceil = \left\lceil \frac{n}{\epsilon} \right\rceil .$$

   So all new values are at most $\lceil n/\epsilon \rceil$, a good news as now the total profit can't be much ($\leq \sum_i v_i' \leq n v_m'$). Since running the scaling friendly dynamic programming we will give us an optimal solution with v_i' in time $O(n \cdot n \cdot v_m) = O(n^2 \cdot v_m) = O(n^3 \cdot \frac{1}{\epsilon})$ (polynomial in n and $1/\epsilon$).
3. Lets see what is the error in using v_i' instead of v_i in the dynamic programming run. Let S' be the solution returned by this dynamic programming algorithm with values v_i'. Let S be the optimal solution to the original instance, i.e $OPT = \sum_{i \in S} v_i$. Since we didn't change the weights w_i or the capacity C we have the $w(S') < C$. We will show that $v(S') \geq (1-\epsilon)v(S)$. Note that since S' is optimal w.r.t v_i', we have that $v'(S') \geq v'(S)$ (as S is a feasible solution in the scaled down version too). One last thing and the rest is calculations, we have by construction we have

   $$\frac{v_i}{b} \leq v_i' \leq \frac{v_i}{b} + 1.$$

Hence we get

$$
\begin{aligned}
OPT &= \sum_{i \in S} v(i) \\
&\leq \sum_{i \in S} b \cdot v'_i \\
&\leq b \cdot \sum_{i \in S} v'_i \\
&\leq b \cdot v'(S) \\
&\leq b \cdot v'(S') \\
&\leq b \cdot \sum_{i \in S'} v'_i \\
&\leq b \cdot \sum_{i \in S'} (\frac{v_i}{b} + 1) \\
&= b \cdot \sum_{i \in S'} \frac{v_i + b}{b} \\
&= b \cdot \frac{1}{b} \sum_{i \in S'} (v_i + b) \\
&= \sum_{i \in S'} v_i + b \cdot |S'| \\
&\leq v(S') + n \cdot b \\
&= v(S') + \epsilon \cdot OPT
\end{aligned}
$$

Hence $v(S') \geq (1-\epsilon) \cdot OPT$, therefore, S', the solution given by the scaling friendly dynamic programming with scaled down values is at least $(1-\epsilon)$-approximate.

Here is a **huge** problem. How do we know the value of OPT, that we used throughout in the form of b. Actually we don't know value of OPT, but we know a good lower bound on that i.e. $OPT \geq v_m$ (assuming all items have weight at most C, since larger weighing items can be ignored in the pre-processing). So we take $b = \frac{\epsilon}{n} \cdot v_m$ and do all the calculations with this new b. All the above three points go through with this new b, in the third point at the end of the chain of inequalities we get that $OPT \leq v(S') + \epsilon \cdot v_m$, so we plug in $v_m \leq OPT$ there and get $OPT \leq v(S) + \epsilon \cdot OPT$.