E0 249: Approximation Algorithms January 7, 2022

Lecture 1

Instructor: Arindam Khan *Scribe: Sarvagya Jain, Prashant Gokhale*

1 Introductory Lecture

1.1 Motivation and Definitions

What is an approximation algorithm?

- Find a near-optimal solution,
- Efficient (polynomial time),
- Works for all instances.

Why study approximation algorithms?

- Fast solutions for practical problems,
- Provides mathematical rigour to study heuristics,
- Quantifies difficulty of different discretization problems,
- Leads to cool algorithms.

Some important definitions:

Definition 1 ($OPT_\Pi(I)$)**.** *Let Π be an optimization problem and I be an instance for Π. Then, $OPT_\Pi(I)$ is the value of the optimum solution for the instance I.*

Definition 2 (α-Approximation Algorithm)**.** *Let $\alpha \geq 1$. A is an α-Approximation Algorithm for a minimization problem Π if $A(I) \leq \alpha \cdot OPT_\Pi(I)\ \forall I$, where $A(I)$ is the value of the solution that A returns for I. Typical values for α are 1.5, $O(\log n)$. For a maximization problem: $A(I) \geq \frac{1}{\alpha} OPT_\Pi(I)$. This is also called absolute approximation algorithm.*

Remark

1. *Usually we omit Π and I in $OPT_\Pi(I)$ and just write OPT.*
2. *Sometimes in literature $\alpha < 1$ for maximization problem, with the definition modified appropriately.*

Surprisingly, NP-hard problems can be quite different in terms of approximately. For some problems, it is NP-hard to obtain even any $f(n)$ approximation (TSP) and for some we can get $(1+\varepsilon)$-approximation in $poly(n, \frac{1}{\varepsilon})$ time, for any $\varepsilon > 0$ (knapsack problem).

Definition 3 (PTAS)**.** *A_ε is a polynomial time approximation scheme (PTAS) for a minimization problem Π if $A_\varepsilon(I) \leq (1+\varepsilon)OPT(I)\ \forall I$ and for every fixed $\varepsilon > 0$, the running time of A_ε is polynomial in the input size.*
Typical running times are $O(\frac{n}{\varepsilon})$, $2^{\frac{1}{\varepsilon}} n^2 \log^2(B)$, $n^{(O(\frac{1}{\varepsilon}))^{O(\frac{1}{\varepsilon})}}$, where n is the number of objects in the instance and B is the biggest appearing number.

Definition 4 (EPTAS). *A_ε is a efficient polynomial time approximation scheme (EPTAS) for a minimization problem Π if for every $\varepsilon > 0$, $A_\varepsilon(I) \leq (1+\varepsilon)OPT(I)\ \forall I$ and the running time of A_ε is polynomial in the input size and $f(\frac{1}{\varepsilon})$ where f is some function. Typical running times are $2^{\frac{1}{\varepsilon}} n^{1000}$, $(\frac{7}{\varepsilon})^{(\frac{1}{\varepsilon})^{\frac{1}{\varepsilon}}}$ but not $n^{\log(\frac{1}{\varepsilon})}$.*

Definition 5 (FPTAS). *A_ε is a fully polynomial time approximation scheme (FPTAS) for a minimization problem Π if for every $\varepsilon > 0$, $A_\varepsilon(I) \leq (1+\varepsilon)OPT(I)\ \forall I$ and the running time of A_ε is polynomial in the input size and $\frac{1}{\varepsilon}$. A typical running time is $O(\frac{n^3}{\varepsilon^2})$.*

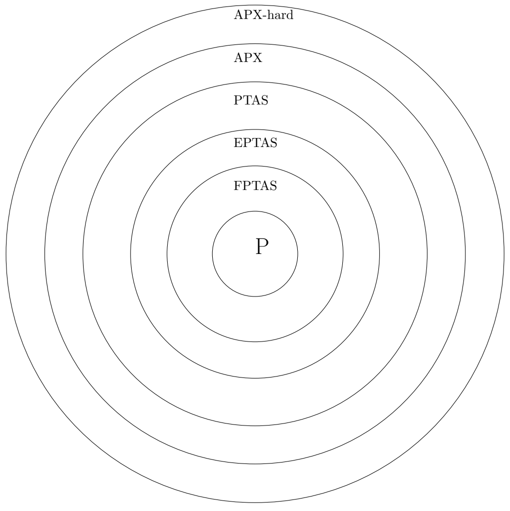
Note: If a problem is APX-hard then it admits no PTAS. If a problem is W[1]-hard then it admits no EPTAS. If a problem is strongly NP-hard then it admits no FPTAS.

Definition 6 (QPTAS). *A_ε is a quasi polynomial time approximation scheme (QPTAS) for a minimization problem Π if for every $\varepsilon > 0$, $A_\varepsilon(I) \leq (1+\varepsilon)OPT(I)\ \forall I$ and the running time of A_ε is quasi polynomial (a running time that is between polynomial and exponential time). A typical running time is $n^{(\log n)^{O(1)}}$.*

Definition 7 (PPTAS). *A_ε is a pseudo polynomial time approximation scheme (PPTAS) for a minimization problem Π if for every $\varepsilon > 0$, $A_\varepsilon(I) \leq (1+\varepsilon)OPT(I)\ \forall I$ and the running time of A_ε is pseudo polynomial (a running time that is polynomial in the numeric data present in the input, but not necessarily in the number of bits required to represent it). A typical running time is $(nB)^{O(1)}$ where B is the biggest numeric data.*

We can divide the problems into various classes based on the approximation scheme possible. For example, $PTAS$ is the class of problems that admit a polynomial time approximation scheme and APX is the class of problems that have a constant-factor approximation. $PTAS \subsetneq APX$ as bin packing has no PTAS under the assumption $P \neq NP$. A problem Π_1 is said to be APX-hard if there is a PTAS-reduction from every problem $\Pi_2 \in APX$ to Π_1, and to be APX-complete if Π_1 is APX-hard and also in APX.
An APX-hard problem does not admit a PTAS unless $P = NP$. In fact there is no quasi-polynomial time approximation scheme (QPTAS) for any APX-hard problem unless NP $\subseteq$ QP.
We can summarise these classes by the following diagram:

Containments are strict-assuming $P \neq NP$



Definition 8 (Asymptotic Approximation Ratio). *The asymptotic approximation ratio ρ_A^∞ of an algorithm A is defined as $\lim_{n\mapsto\infty} \sup \rho_A^n$, where $\rho_A^n = \sup_{I\in l}\left(\frac{A(I)}{OPT(I)} | OPT(I) = n\right)$.*

Consider two algorithms $A_1 = OPT(I) + 1$ and $A_2 = 2OPT(I)$. Both attain absolute approximation ratio of 2. But as OPT gets larger A_1 performs much better in terms of ratio.

Definition 9 (APTAS)**.** *A_ε is a asymptotic polynomial time approximation scheme (APTAS) if for every $\varepsilon > 0$ there is a poly-time algorithm with asymptotic approximation ratio of $(1+\varepsilon)$. Typical running times are $(1+\varepsilon)OPT + O(1)$.*

1.2 Different Inapproximability

Recent inapproximability results divide problems into four broad classes, based on the approximability ratio that is provably hard to achieve. These approximation ratios are respectively, $1+\varepsilon$ for some fixed $\varepsilon > 0$, $\Sigma(\log n)$, $2^{(\log n)^{1-\gamma}}$ for every fixed $\gamma > 0$, and n^δ for some fixed $\delta > 0$ (n is input size throughout). The corresponding classes of problems are called Classes I, II, III, and IV respectively. Inapproximability results for problems within a class share common ideas. Representative problems for each class are MAX-3SAT, SETCOVER, LABELCOVER and CLIQUE, respectively.

1.3 Why optimization problems are hard?

Optimization problems can be hard due to multiple reasons:

- **Intractability:** Computational hardness such as NP-hardness. For example, finding best route to deliver all packages (travelling salesman problem) or finding optimal schedule of jobs into servers (bin packing). Approximation algorithms are useful in such cases.
- **Uncertainty:** May not have complete data. For example, online ad allocation (matching) or online taxi allocation (k-server). Online algorithms are useful in such cases.
- **Dynamic Input:** Data changes over time and algorithm needs to maintain a good solution with minimum update (dynamic set cover). Dynamic algorithms are useful in such cases.
- **Big Data:** Data arrives too fast or is too big to store. Streaming algorithms are useful in such cases.

Approximation algorithms go beyond P!=NP problem. Approximation is also very useful in the context of fine-grained algorithms (say, for algorithms with runtime $O(n^2)$), parameterized algorithms, distributed algorithms, etc.

1.4 Different Approximation Techniques and some examples

- Greedy: set cover, vertex set cover, 3-SAT,
- Local search: max cut, facility location,
- Combinatorial Algorithm: TSP, Steiner Tree,
- Rounding Data: Bin packing, Scheduling jobs on identical parallel machines,
- Dynamic Programming: Knapsack, maximum independent set of rectangles,
- Cuts and metrics/embeddings: min multicut, multiway cut, sparsest cut,
- Linear Programming: will be a major part of this course (rounding, dual-fitting, primal-dual),
- Semidefinite programs: max-cut.

1.5 Brief History

One of the first problems that involved approximation algorithm was the problem of Multiprocessor Scheduling with Precedence Constraints, studied by Ron Graham in 1966.
The problem statement goes as follows:
We are given n jobs $J_1, J_2, \dots, J_n$ with job J_i having a processing time p_i, a precedence relation $\prec$ defined on the jobs ($J_i \prec J_\ell$ means that J_i has to be finished, before J_ℓ is allowed to start) and m identical machines. We can represent the precedence constraints using a Directed Acyclic Graph (DAG). The goal is to find a non-preemptive schedule of the jobs on m machines respecting the precedence order that minimizes the makespan (time taken to finish the jobs).
Graham's list scheduling provides us a 2-approximation to the makespan. The algorithm goes as follows:

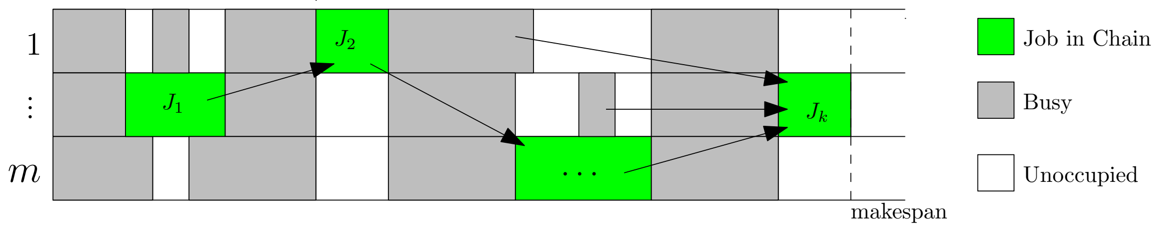
```
Algorithm 1: Grahams-List-Scheduling
  Data: Jobs $J_1, J_2, \dots, J_n$, precedence relation $\prec$, $m$ machines
  Result: An 2-approximation to the makespan
  for $t = 1, \dots$ do
      if machine $j \in \{1, \dots, m\}$ idle at $t$ AND all predecessors of some unfinished job $J_i$ are finished then
          schedule $J_i$ on machine $j$ starting from $t$;
      end
      if All jobs finished then
          return t;
      end
  end
```

In words, at any time t, just start a job whenever possible. Finally, return the time when all jobs are finished.
Analysis of the algorithm:

Theorem 10. *The makespan of the produced algorithm is at most 2 times the optimal makespan.*

Proof. Start by considering a sequence of jobs (w.l.o.g. after reordering) $J_1, \dots, J_k$ such that:

- J_k is the last job of the whole schedule that finishes
- $J_1 \prec J_2 \prec \dots \prec J_k$ (chain in the partial order $\prec$)
- J_i is the predecessor of J_{i+1} that is finished last



After J_i finished J_{i+1} is started as soon as a machine is available. Hence between J_i is finished and J_{i+1} begins, all machines must be fully busy. Thus, length of all busy periods $\leq \frac{1}{m}\sum_{i=1}^{n} p_i \leq OPT$. Length of time taken by the chain $J_1, \dots, J_k$ is $\leq OPT$. Combining, makespan $\leq$ length chain + busy period $\leq 2 \cdot OPT$. □

Remark We can further tighten the analysis. Let us denote the time taken by the chain by T, then the length of all busy periods $\leq \frac{1}{m}\left(\sum_{i=1}^{n} p_i - T\right) \leq OPT - \frac{T}{m}$. Thus, makespan $\leq T + OPT - \frac{T}{m} = OPT + \left(1 - \frac{1}{m}\right)T \leq \left(2 - \frac{1}{m}\right)OPT$.

We state the following theorem due to Svensson without proof:

Theorem 11. *For every fixed $\varepsilon > 0$, there is no $(2-\varepsilon)$-approximation unless a variant of the Unique Games Conjecture is false.*

Finding the complexity status of $P3 \mid p_i = 1, \text{prec} \mid C_{\max}$ (i.e. PRECSCHEDULING with unit processing times and 3 machines) is still an open problem. It is known that there is a $\frac{4}{3}$-approximation for the problem and that $P2 \mid p_i = 1, \text{prec} \mid C_{\max}$ is polynomial-time solvable.

1.6 Timeline

Some examples of using problems in complexity class P to give approximation to other hard problems:

- 1968: Moore showed optimal minimum spanning tree can be used to give a solution to travelling salesman problem of cost at most two times the optimal cost.
- 1969: Graham studied scheduling with no precendence constraints and proposed longest processing time algorithm with makespan at most $\left(\frac{4}{3} - \frac{1}{3m}\right)$ times the optimal makespan. He also devised the first PTAS (makespan $\leq (1+\varepsilon)OPT$ in run time $O(n \log m + m^{(m-1-\varepsilon m)/\varepsilon})$).

Some examples of Theory of NP-completeness and reductions:

- 1970s: Boolean satisfiability is NP-complete (Cook-Levin '71). 21 famous problems! (Karp '72). Most of the practical problems are NP-hard (Garey-Johnson).
- 1975: David Johnson in his thesis on bin packing problem coined the term *approximation algorithms* .
- 1975: Ibarra and Kim gave FPTAS for knapsack.
- 1976: $\frac{3}{2}$ approx for TSP.
- 1976: Sahni-Gonzalez show that for some problems (such as TSP), obtaining constant ratio approximation is equivalent to obtain the optimal solution.

Some examples of theory of inapproximability:

- 1977: Approx preserving reductions, APX and other classes (Ausellio et al.).
- 1970s and 1980s: Development of polynomial time algorithms for linear programs. Some examples include ellipsoid method (Khachiyan '79), interior point method (Karmakar '84), and optimization=separation (GLS'81: LP with exponential number of constraints can be solved using separation oracle).

Integer programming is NP-hard. Linear programs can be good approximation of integer programs:

- 1970s: Set cover $O(\log n)$-approximation
- 1982: Set cover f-approximation using deterministic rounding (Hochbaum '82)
- 1987: Randomized rounding (Raghavan and Thompson)
- 1982: Asymp. FPTAS (Karmakar-Karp)

Some examples of semidefinite programming.

- 1990s: 0.878 approx algo for MAX-CUT

The 2001 Godel prize was awarded for work on the PCP theorem and its connection to the hardness of approximations.

2 Set Cover

2.1 Problem Statement

Definition 12 (Set Cover)**.** Let E be a set of elements and $S = \{S_1, S_2, S_3, ..., S_m\}$ be a collection of subsets of E. A set cover is a sub collection $S' \subseteq S$ whose union is E. That is,

$$\bigcup_{s \in S'} s = E$$

For example, if $E = \{1, 2, 3, 4, 5, 6, 7, 8\}$ and $S_1 = \{1, 2, 3\}$, $S_2 = \{1, 4, 5, 8\}$, $S_3 = \{2, 3, 4\}$, $S_4 = \{6, 7\}$, $S_5 = \{2, 4, 6\}$ and $S_6 = \{2, 3, 5, 8\}$, then $\{S_2, S_3, S_4\}$ is a set cover but $\{S_1, S_3, S_6\}$ is not a set cover.

In the set cover problem, we are given a ground set of n elements $E = \{e_1, e_2, ..., e_n\}$ and a collection of m subsets of E which is $S = \{S_1, S_2, ..., S_m\}$ and a non-negative weight function $cost : S \mapsto \mathbb{Q}^+$. The goal is to find a minimum weight collection of subsets (the elements of S) that covers all elements in E. If all weights are equal, then it is called the unweighted set cover problem.

Set cover problem is a generalisation of many other important problems like vertex cover and edge cover. Set cover also has many applications in real life like antivirus products and VLSI design.

2.2 The Greedy Paradigm

A greedy paradigm is a general algorithmic paradigm, in which we construct a solution iteratively via a sequence of myopic decisions, and hope that everything works at the end. In other words, we select the best augmentation that minimizes the ratio of cost to advantage. Greedy algorithms are easy to design and analyse, but often do not lead to good approximations for many problems.

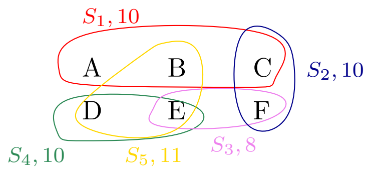
2.3 Greedy Algorithm

For the set cover problem, a "good strategy" for picking the sets can be, pick the set that costs the least per unit uncovered element in each iteration until every element is covered by the union of the picked sets. Using this naive strategy, we arrive at the following algorithm:

```
Algorithm 2: Greedy-Set-Cover($E, S, cost$)
  Data: universe set $E$, collection of sets $S$ and cost function $cost$.
  Result: An $O(\log n)$ approximation to the Set Cover problem
  $C \leftarrow \{\}$;
  while $C$ is not a set cover for $E$ do
      Define cost effectiveness of each set $x \in S$ as $\alpha_x = \frac{cost(S)}{|S \backslash C|}$;
      Find $y$, the most cost effective (that is $\alpha_y$ is the least) set in the current iteration;
      Pick $y$, and for all newly covered elements $e \in y \setminus C$, set $price = \alpha_y$;
      $C \leftarrow C \cup \{y\}$;
  end
  return $C$;
```

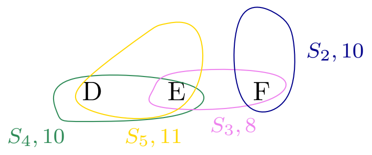
2.4 Example run of greedy algorithm

$$S_1 = \{A, B, C\},\ S_2 = \{C, F\},\ S_3 = \{E, F\},\ S_4 = \{D, E\},\ S_5 = \{B, D, E\},$$



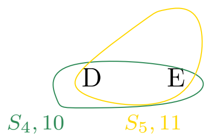
Step 0

$\boxed{\alpha_{S_1} = \frac{10}{3}}$, $\alpha_{S_2} = \frac{10}{2}$, $\alpha_{S_3} = \frac{8}{2}$, $\alpha_{S_4} = \frac{10}{2}$, $\alpha_{S_5} = \frac{11}{3}$



Step 1

$\alpha_{S_1} = \frac{10}{1}$, $\boxed{\alpha_{S_3} = \frac{8}{2}}$, $\alpha_{S_4} = \frac{10}{2}$, $\alpha_{S_5} = \frac{11}{2}$



Step 2

$\boxed{\alpha_{S_4} = \frac{10}{1}}$, $\alpha_{S_5} = \frac{11}{1}$

$$ALGO = c(S_1 \cup S_3 \cup S_4) = 10+8+10 = 28$$

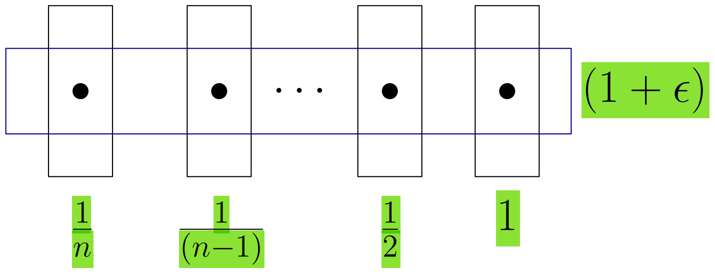
2.5 Analysis of Greedy Algorithm

- This algorithm returns a valid set cover in polynomial time since it needs at most $O(n)$ steps and each step takes at most $O(m \log m + n)$ time.
- In any iteration, left over sets of the optimal solution can cover the remaining elements $E \setminus C$ at a cost of at most OPT. Thus, at least one of the remaining sets must have cost effectiveness of at most $\frac{OPT}{|E \setminus C|}$ by an averaging argument.
- WLOG assume that the elements are numbered in the order in which they were covered, resolving ties arbitrarily. Let $e_1, e_2, \cdots, e_n$ be this numbering.
- Assume element e_k was covered by the most cost effective set at some iteration $i (\leq k)$. At most $(k-1)$ items were covered before the iteration i. Thus, at least $n - (k-1)$ elements were not covered before the iteration i and hence $|E \setminus C| \geq (n - k + 1)$ at the beginning of iteration i.
- Now using these observations, $price(e_k) \leq \frac{OPT}{|E \setminus C|} \leq \frac{OPT}{n-k+1} = \frac{OPT}{p}$ where $p = (n - k + 1)$
- *price* of elements is obtained by distributing set weights into the items. Therefore, $\sum_{S_j \in C} cost(S_j) = \sum_{e_i \in E} price(e_i)$
- Now, $\sum_{e_k \in E} price(e_k) \leq \sum_{k=1}^{n} \frac{OPT}{n-k+1} \leq \sum_{p=1}^{n} \frac{OPT}{p} \leq H_n \times OPT$

- Thus, the greedy algorithm has H_n or $O(\log n)$ approximation ratio, where H_n is the n^{th} harmonic number.

Remark As was done above, finding a good lower bound on OPT is a general strategy used in proving approximation bounds in minimisation problems.

This analysis is actually tight. Consider the following example:



- Optimal solution has only one set of cost $(1+\varepsilon)$, where $0 < \varepsilon \ll 1$.
- The greedy algorithm will return n singleton sets with total cost $= \frac{1}{n} + \frac{1}{n-1} + ... + 1 = H_n$
- So, the approximation ratio for this example is $\frac{H_n}{1+\varepsilon} \approx H_n$ as ε can be taken to be arbitrarily small.
- Thus, our analysis is tight as we have both upper and lower bounds.

Remark A further result by P. Slavic (STOC '96) shows that the performance ratio of greedy algorithm is exactly $\ln n - \ln\ln n + \Theta(1)$.

Dinur and Steurer in 2014 showed that it is NP-hard to approximate the optimum for set cover within $(1-\varepsilon)\ln n$ factor, $\forall$ constant $\varepsilon > 0$.

2.6 Vertex Cover Problem

In this section, we state the vertex cover problem.

The vertex cover problem is as follows, we are given a undirected graph $G = (V, E)$ with node weights $c : V \mapsto \mathbb{Q}^+$. We want to find a subset $U \subseteq V$ such that every edge is incident to at least one node in U and $\sum_{v \in U} c(v)$ is minimised.

Vertex cover is a special case of set cover. To see this, simply take the universe in set cover to be set of all edges and available subsets to be all the edges touched by a particular vertex, i.e., S_i is the set of edges incident on the i^{th} vertex.

In this case, a simple 2-approximation algorithm is possible and we leave at as an exercise for the reader to find one.

Exercise 13. *Find a 2-approximation algorithm to the vertex cover problem.*

3 Max-Cut

3.1 Problem Statement

We will start by defining two quantities essential for stating the problem of finding a Max-Cut.

Definition 14 (Cut)**.** Given an undirected graph $G(V, E)$, a cut $[S, V \setminus S]$ is a partition of V.

Definition 15 (Separated Edges)**.** Given an undirected graph $G(V, E)$ and a set $S \subseteq V$, the separated edges, denoted by $\delta(S)$, refers to the set of edges having one endpoint in S and the other in $V \setminus S$.

We are finally ready to define what we mean by a Max-Cut.

Definition 16 (Max-Cut)**.** Given a complete undirected graph $G(V, E)$ and edge weights $w : E \to \mathbb{Q}_+$, Max-Cut refers to a cut maximising the weight of separated edges. Mathematically, $[\mathcal{S}, V \setminus \mathcal{S}]$ is a Max-Cut if

$$\mathcal{S} = \operatorname*{argmax}_{S \subseteq V} \left\{ \sum_{e \in \delta(S)} w(e) \right\}.$$

Intuitively, in order to find a Max-Cut of a graph $G(V, E)$, we need to find a set $S \subseteq V$ such that the sum of weights of edges from S to $\overline{S} := V \setminus S$ is maximised.

Unlike Min-Cut, finding a Max-Cut is an NP-hard problem. In order to find a good approximation to the Max-Cut, we look for a cut for which the weight of separated edges is close to OPT, where

$$OPT = \max_{S \subseteq V} \left\{ \sum_{e \in \delta(S)} w(e) \right\}.$$

3.2 A Randomized Algorithm

In this section, we present a simple randomized approximation algorithm. We will prove that the approximation factor of the algorithm is 2 in expectation.

```
Algorithm 3: Randomized-Max-Cut($G(V, E)$)
  Data: An unweighted graph $G(V, E)$
  Result: A 2-approximation in expectation to the Max-Cut for $G(V, E)$
  For each vertex $v \in V$, add $v$ to set $S$ with probability 1/2.;
  $\overline{S} := V \setminus S$;
  return The set of edges between $S$ and $\overline{S}$;
```

Theorem 17. *The above algorithm is a* 2*-approximation randomized algorithm.*

Proof. Let X_i be the indicator random variable for the event that both endpoints of the edge e_i belongs to different sets, i.e., one of the endpoints belongs to S and the other to $\overline{S}$. Now we note that

$$\mathbb{E}[X_i] = \frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 0 = \frac{1}{2}$$

By the linearity of expectation and the fact that $|E|$ is an upper bound on OPT, we get

$$\mathbb{E}[|[S, \overline{S}]|] = \mathbb{E}[\sum_{i \in E} X_i] = \sum_{i \in E} \mathbb{E}[X_i] = \frac{|E|}{2} \geq \frac{OPT}{2}.$$

Here, $|[S, \overline{S}]|$ denotes the number of edges having one endpoint in S and other in $\overline{S}$.

This shows that the proposed algorithm outputs a $2-$approximation in expectation. □

3.3 Derandomization

In this section, we will use a technique referred to as derandomization to derandomize the algorithm in the previous section so that we can always guarantee a 2−approximation. Later in the section, we will use the insights gained from the derandomization procedure to give two more 2-approximation algorithms.

Let us consider the vertices of the graph one by one in some sequence, say $v_1, v_2, \cdots, v_n$, where $n := |V|$.

Let x_i be the value assigned to the vertex v_i. We will construct two disjoint sets A and B of vertices using the following rules:

- if $x_i = 0$, then $v_i \in A$
- if $x_i = 1$, then $v_i \in B$

Suppose we have assigned values $x_1, \cdots, v_k$ to vertices $v_1, \cdots, v_k$.

Now we consider the expected value of the cut if the remaining vertices $v_{k+1}, \cdots, v_n$ are independently assigned values from the set $\{0, 1\}$ uniformly at random. i.e., we consider the conditional expectation $\mathbb{E}[\text{cut}(A,B) \mid v_1 = x_1, \cdots, v_k = x_k]$, where $\text{cut}(A,B) := |[A,B]|$ (the number of edges between A and B). The strategy is to inductively assign a value x_{k+1} to v_{k+1} such that the following holds:

$$\mathbb{E}[\text{cut}(A,B) \mid v_1 = x_1, \cdots, v_k = x_k] \le \mathbb{E}[\text{cut}(A,B) \mid v_1 = x_1, \cdots, v_k = x_k, v_{k+1} = x_{k+1}].$$

The above condition implies the following:

$$E[\text{cut}(A,B) \mid v_i = x_i, \cdots, v_n = x_n] \ge E[\text{cut}(A,B)] \ge \frac{|E|}{2} \ge \frac{OPT}{2}$$

Thus, it suffices to show the existence of such an inductive procedure.

- Base Case ($k = 1$):
  Since it doesn't matter whether we add the first vertex to A or B, hence
  $$\mathbb{E}[\text{cut}(A,B) \mid v_1 = x_1] = \mathbb{E}[\text{cut}(A,B)]$$
- Inductive Step:
  Since v_{k+1} is assigned either 0 or 1 with probability 1/2, hence
  $$\begin{aligned} E\left[\text{cut}(A,B) \mid v_1 = x_1, \ldots, v_k = x_k\right] &= \frac{1}{2} \cdot E\left[\text{cut}(A,B) \mid v_1 = x_1, \ldots, v_k = x_k, v_{k+1} = 0\right] \\ &+ \frac{1}{2} \cdot E\left[\text{cut}(A,B) \mid v_1 = x_1, \ldots, v_k = x_k, v_{k+1} = 1\right] \\ &= \frac{Q_1 + Q_2}{2} \end{aligned}$$
  where Q_1 and Q_2 are the first and the second terms of the right hand side of the equation respectively. Also,
  $$\max(Q_1, Q_2) \ge \frac{Q_1 + Q_2}{2} = E\left[\text{cut}(A,B) \mid v_1 = x_1, \ldots, v_k = x_k\right]$$
  Thus, we will use the following assignment scheme:
  $$v_{k+1} = \begin{cases} 0, & \text{if } Q_1 > Q_2 \\ 1, & \text{otherwise} \end{cases}$$

The only thing that remains is the computation of Q_1 and Q_2. To compute Q_1, assign the vertices $v_1, \cdots, v_k$ the values $x_1, \cdots, x_k$ respectively and assign v_{k+1} the value 0. Now count the number of edges in the graph restricted to $v_1, \cdots, v_{k+1}$ that have different valued endpoints. To this, add 0.5 times the number of edges in the original graph that were not in the restricted graph to obtain the value of Q_1 (such an edge contributed to the value of Q_1 with probability 1/2). Similarly, one can compute the value of Q_2. It is easy to see that the time required is polynomial in the number of edges and vertices.
This completes our inductive procedure.

Remark In the above algorithm, one should note that the value assigned to v_{k+1} depends on whether v_{k+1} has more 0 neighbours or 1 neighbours as the edges not having v_{k+1} as an endpoint contribute the same quantity to Q_1 and Q_2. Using this observation, we will present a simple greedy algorithm next.

Algorithm 4: Greedy-Max-Cut($G(V, E)$)

```
Data: An unweighted graph $G(V, E)$
Result: A 2-approximation to the Max-Cut for $G$
Consider vertices in some order $v_1, \cdots, v_n$, where $n = |V|$;
$A = \{v_1\}$;
$B = \{v_2\}$;
for each successive vertex do
    if $v_i$ has more neighbours in $A$ than $B$ then
        Put $v_i$ in $B$;
    end
    else
        Put $v_i$ in $A$;
    end
end
return The set of edges between $A$ and $B$, i.e., $[A, B]$;
```

3.4 Local Search Based Algorithm

Given instance I of a problem, let $\mathcal{S}(I)$ be the set of all feasible solutions for I. For $S \in \mathcal{S}(I)$, let $N(S)$ (the neighbourhood of S) be the set of all solutions S' that can be obtained from S via some local moves.

Now, the general scheme of local search algorithms can be summarised by the following pseudocode:

```
Algorithm 5: General-Local-Search
  Find a good initial solution $S_0 \in \mathcal{S}(I)$;
  $S \leftarrow S_0$;
  while True do
      if $\exists S' \in N(S)$ such that $val(S')$ is strictly better than $val(S)$ then
          $S \leftarrow S'$;
      end
      else
          S is a local optimum;
          return S;
      end
  end
```

Here *val* of a set represents the number corresponding to that set, which is use to compare it with other sets in order to progress towards a better solution.

For minimisation (resp. maximisation) problems, S' is strictly better than S if $val(S') < val(S)$ (resp. $val(S') > val(S)$).

In these algorithms, we look for quick termination and good solution guarantee of local optima.

Based on the property we remarked below the inductive procedure for derandomization, we will now present a local search algorithm for the max-cut problem (let $n := |V|$ and $m := |E|$).

```
Algorithm 6: General-Local-Search
  Data: An unweighted graph $G(V, E)$
  Result: A set $S \subseteq V$
  Intialise $S$ arbitrarily;
  Find a good initial solution $S_0 \in \mathcal{S}(I)$;
  $S \leftarrow S_0$;
  while $\exists u \in V$ with more edges to the same side than across do
      move u to the other side;
  end
  return S;
```

Theorem 18. *The above algorithm runs in polynomial time.*

Proof. The proof follows from following observations:

- Checking if $\exists u \in V$ with more edges to the same side than across takes $O(mn)$ time.
- Now, we will bound the number of iterations. Initially, $|E(S, \overline{S})| \geq 0$. Every iteration, $|E(S, \overline{S})|$ increases by at least 1. Also, clearly $|E(S, \overline{S})| \leq m \leq n^2$. Hence, there are at most n^2 iterations.
- Combining, the total run time is $O(mn^3)$.

□

Theorem 19. *The above algorithm is a 2-approximation algorithm.*

Proof. Let S be a local optimum, i.e., $\forall u \in V$, there are more $\{u, v\}$ edges across the cut than the ones connecting vertices in the same side.
The algorithm always returns such an S.

$$\therefore |E(S, \overline{S})| = \frac{1}{2} \sum_{u \in V} d_{across}(u) \geq \frac{1}{2} \sum_{u \in V} \frac{1}{2} d(u) = \frac{1}{4} 2m = \frac{m}{2} \geq \frac{OPT}{2}.$$

Here, $d_{across}(u)$ is the number of edges from S to $\overline{S}$ that have u as one of the endpoints. □

We leave the generalisation to the weighted case as an exercise for the reader.

Exercise 20. *Extend the local search algorithm to account for weighted edges.*