Lecture 8: Las Vegas Algorithms, Approximation Algorithms

Las Vegas Algorithms

- When defining the class RP, the calculations of an algorithm depend on random variables. The running time is always polynomial, the output can be incorrect with a certain probability.
- Now we consider randomized decision algorithms that do not make a false statement for languages $L \subseteq \Sigma^*$, i.e. neither in the case $w \in L$ nor in the case $w \notin L$ for all $w \in \Sigma^*$.
- However, we do not always get a reasonable, i.e. polynomial time an answer - what do we want to note with the output "?".
- We consider the running time $time_A$ of a randomized algorithm A as a random variable. We are interested in the probability distribution over the running time of words of length $n\ (\leq n)$

Expected Polynomial (EP):

Definition: Let Σ be an alphabet. $L \subseteq \Sigma^*$belongs to the class EP iff there is a randomized decision algorithm A for L with a polynomial worst expected running time. Such algorithms are called Las Vegas algorithms.

Density function: $time_A(w)\ \ |w| = n\ (\leq n)$

$sup\{E(time_A(w)\ )\ |\ |w| \leq n\} = O\big(p(n)\big)$ for a polynom p

Zero-Error Probabilistic Polynomial (ZPP)

ZPP algorithms are also Las Vegas algorithms that do not make false statements, but that can also stop without making a decision - with the output "?" ("No idea").

Definition: $\varepsilon: \mathbb{N} \to [0,1),\ L \subseteq \Sigma^*, L \in ZPP(\varepsilon(n))$ iff there is a probabilistic algorithm A with: $w \in \Sigma^*$

(1) $Prob_A[A\ accepts\ w \mid w \notin L] = 0$

(2) $Prob_A[A\ doesn't\ accept\ w \mid w \in L] = 0$

(3) $Prob_A[A\ outputs\ "?"] < \varepsilon(n)$

(4) $sup\{E(time_A(w)) \mid |w| \leq n\} = O(p(n))$ for a polynom p

A simple example:

```
1  // Las Vegas algorithm
2  repeat:
3      k = RandInt(n)
4      if A[k] == 1,
5          return k;
```

- A variable k is generated randomly; after k is generated, k is used to index the array A.
- If this index contains the value 1, then k is returned; otherwise, the algorithm repeats this process until it finds 1.
- Although this Las Vegas algorithm is guaranteed to find the correct answer, it does not have a fixed runtime; due to the randomization (in line 3 of the above code), it is possible for arbitrarily much time to elapse before the algorithm terminates.

Here is a table comparing Las Vegas and Monte Carlo algorithms:

	Running Time	**Correctness**
Las Vegas Algorithm	probabilistic	certain
Monte Carlo Algorithm	certain	probabilistic

The Bounded error Probabilistic Polynomial (BPP)

So far, we have considered randomized algorithms with one-sided error (i.e., Monte Carlo algorithms) and those that do not make an error but also cannot make a decision (i.e., Las Vegas algorithms). Now we consider classes of languages for which two-sided errors are allowed in the decision.

The language $L \subseteq \Sigma^*$ belongs to the class BPP if and only if there exists a polynomial, randomized algorithm A and an $\epsilon, 0 < \epsilon < \frac{1}{2}$, with

(1) for all $w \notin L$: Prob $[A(w) = 0] \geq \frac{1}{2} + \epsilon$,
(2) for all $w \in L$: Prob $[A(w) = 1] \geq \frac{1}{2} + \epsilon$,
(3) for all $w \in \Sigma^*$: time $_A(w) = O($ poly $(|w|))$.

A BPP algorithm is called an Atlantic City algorithm.

Approximation Algorithms

- An approximation algorithm returns a solution to a combinatorial optimization problem that is **provably** close to optimal (as opposed to a *heuristic* that may or may not find a good solution).
- Approximation algorithms are typically used when finding an optimal solution is intractable.
- But they can also be used in some situations where a near-optimal solution can be found quickly and an exact solution is not needed.

Definition: Let P be a problem, then:

1) A_P: Problem instances

undirected Graphs $G = (V, E)$

$V = \{1,2,3, \dots, n\}, n \geq 0$
$E \subseteq V\ x\ V \quad \{x, y\} \in E, x \rightarrow y$

with $l: A_P \rightarrow \mathbb{N}_0 \quad l(V, E) = |V|$

2) $S_P(a)$: Set of valid solutions for $a \in A_P$

3) $m_P^a: S_P(a) \rightarrow \mathbb{N}_0$

4) Now we can distinguish between 2 types of optimization problems, namely a minimization problem and a maximization problem.

If P is a minimization problem, then a solution $x \in S_P(a)$ with $m_P^a(x) = min\{m_P^a(y) \mid y \in S_P(a)\}$ should be found for $a \in A_P$.

If P is a maximization problem, then a solution $x \in S_P(a)$ with $m_P^a(x) = max\{m_p^a(y) \mid y \in s_P(a)\}$ should be found for $a \in A_P$.

Example: P: The graph coloring problem with the constraint that no adjacent nodes have the same color. Determine the minimum number of colors!

1) A_P is the set of all undirected graphs $G = (V, E)$ as well $l(G) = |V|$.

2) $S_P(G)$ is the set of all valid coloring solutions f of G.

3) $m_P^G(f)$ is the number of colors that f assigns to the nodes.

4) It is a minimization problem.

Quality criteria:

Let $A(a)$ be the value of the solution that Algorithm A delivers for problem instance a. Furthermore, let $OPT(a)$ be the value of the optimal solution $s \in S_P(a)$.

Absolute Error: $|OPT(a) - A(a)|$
Relative Error: $\frac{|OPT(a)-A(a)|}{OPT(a)}$

If there is an ε> 0 with

$$\frac{|OPT(a)-A(a)|}{OPT(a)} \leq \varepsilon, \quad \varepsilon \geq 0$$

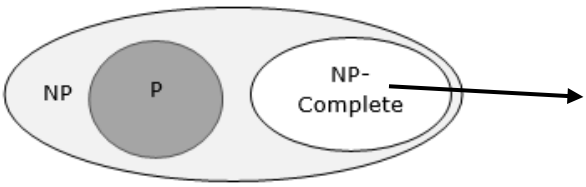
for all a ∈ A_P, then A is called **ε-approximation** for P.

➔ There exist 3 classes for a problem $P \in NPC$ in terms of approximation.

(1) Fully approximable, i.e. there is a polynomial ε-approximation for P for all ε > 0

(2) Partially approximable, i.e. polynomial ε-approximation is only possible for particular ε-values

(3) Not approximable, i.e. no polynomial ε-approximation is possible => ∄ε



NP-complete problems are divided into 3 approximation classes.

Induced Subgraph (G_{IS}):

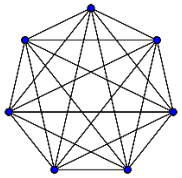
$$G = (V, E)$$

$$U \subseteq V, E_U \subseteq E$$

$G_{IS} = (U, E_U)$ is a subgraph induced by U with $E_U = E \cap UxU$

Complete graph with n nodes (K_n):

$$K_n = (\{1, \dots, n\}, \{1, \dots, n\} x \{1, \dots, n\})$$



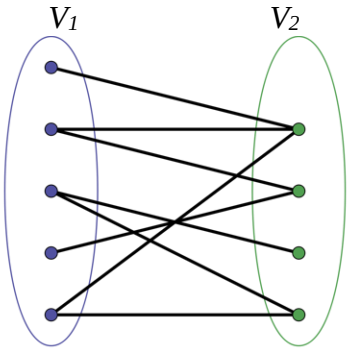
Example of a complete graph

Bipartite Graph:

$$V_1, V_2 \subset V$$

$$E \subseteq V_1 x V_2$$

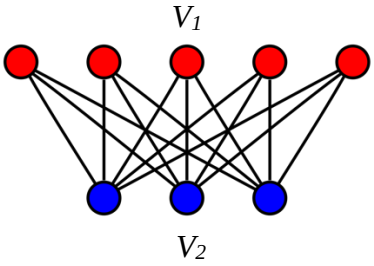
$$G = (V_1 \; and \; V_2, E) \text{ with } V_1 \cap V_2 = \emptyset$$



Example of a bipartite graph

Complete Bipartite Graph $(K_{n,m})$:

$$V_1 = \{1, \dots, n\}, V_2 = \{1, \dots, m\}, E = V_1 x V_2$$



Example of a complete bipartite graph with n = 5 and m = 3

Circle Graph (C_n):

$$C_n = (\{1, \dots, n\}, E)$$
$$E = \{(i, i+1) \mid 1 \leq i \leq n-1\} \cup \{(n, 1)\}$$

Clique Number of a Graph ω(G):

$G = (V, E)$, $U \subseteq V$ is clique iff G_{IG} is complete.

Clique number of G: $\omega(G) = max\{|U| \mid U \text{ is clique of } G\}$

Independence Number of a Graph $\alpha(G)$:

$G = (U, E)$, $U \subseteq V$ is independent iff $\forall i, j \in U, i \neq j: (i, j) \notin E$

Independence number of G: $\alpha(G) = max\{|U| \mid U \text{ is independent in } G\}$

Clique Number	Independence Number
$\omega(K_n) = n$	$\alpha(K_n) = 1$
$\omega(C_n) = 2, n = 2 \text{ and } n \geq 4$ $\omega(C_n) = 3, n = 3$	$\alpha(C_n) = \left\lfloor \frac{n}{2} \right\rfloor$
$\omega(K_{n,m}) = 2, m, n \geq 1$	$\alpha(K_{n,m}) = max\{m, n\}$
$\alpha(G) = \omega(\bar{G})$	

Graph Coloring

$G = (V, E)$ is k-colorable iff there is a mapping $c: V \to \{1, \dots, k\}$ with $(i, j) \in E$ and $c(i) \neq c(j)$

$$\chi(G) = min\{k \mid G \text{ is } k - colourable\}$$

⇨ $\chi(G)$ is the chromatic number of G

(1) $\chi(G) = 1$, if $E = \emptyset, |V| \geq 1$

(2) $\chi(G) \leq d_G + 1$

a. The degree of a node is the number of edges that are incident on the node.

b. **The degree of a graph** d_G is the maximum over all node degrees.

c. Examples:

(3) $\chi(G) \geq \omega(G)$

(4) $\chi(K_n) = n$

(5) G bipartite: $\chi(G) \leq 2$

(6) $\chi(C_n) = \begin{cases} 2, & n\ even \\ 3, & n\ odd \end{cases}$

Estimating the chromatic number of G, i.e. $\chi(G)$:

$$G = (V, E), |V| = n$$

$$\frac{n}{\alpha(G)} \leq \chi(G) \leq n - \alpha(G) + 1$$

Question: Is this estimation useful to us in terms of determining an approximation? Although this estimation limits the interval, it does not really help us for an acceptable approximation. In the following we consider special examples where this interval is arbitrarily filled.

Complete bipartite Graph $K_{n,n}$:

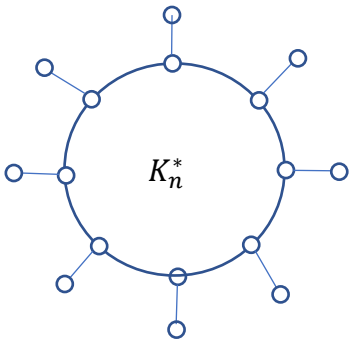
$$\alpha(K_{n,n}) = n$$
$$\chi(K_{n,n}) = 2$$

$$\frac{2n}{\alpha(K_{n,n})} \leq \chi(K_{n,n}) \leq 2n - \alpha(K_{n,n}) + 1$$

$$\frac{2n}{n} \leq 2 \leq 2n - n + 1$$

$$2 \leq 2 \leq n + 1$$

(Modified) Complete Graph K_n^*:



$$\alpha(K_n^*) = n$$

$$\chi(K_n^*) = n$$

$$\frac{2n}{n} \leq \mathrm{n} \leq 2n - n + 1$$

$$2 \leq n \leq n + 1$$

Question: Can the coloring problem be approximated at all?

Theorem: Let A be a polynomial algorithm to approximate $\chi(G)$ of an undirected graph G with $|OPT(G) - A(G)| \leq k$, then it follows that $P = NP$

<u>Assuming the reverse of the theorem, i.e. P ≠ NP,</u> it follows that there can be no polynomial algorithm for approximating $\chi(G)$.

Instead of approximation algorithms, we must use heuristics and analyze them in terms of their goodness.

➔ 2 well-known classical heuristic algorithms: Greedy and Johnson Coloring Algorithms

Greedy Coloring Algorithm:

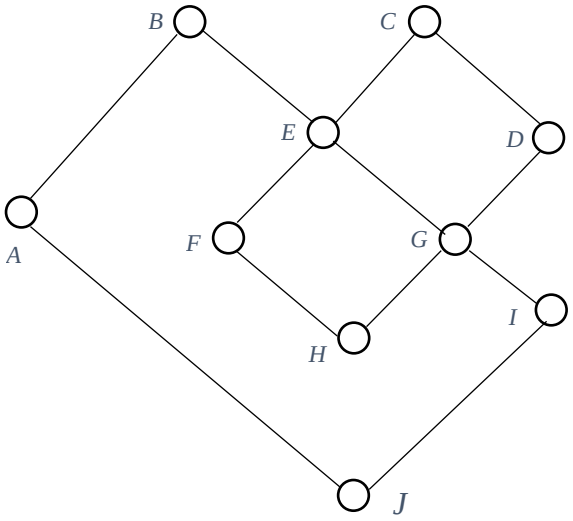
```
algorithm greedyColoring;
    input G = (V, E) : Graph; //V = {1, 2, ..., n}

    output f: [1...max] of int;
           color: int;

    var counter, color, i, j: int;
    f := 0; color := 0; counter := 0;

    while counter < n do
       color := color + 1; for all i ∈ V do
          if f[i] < 1 and f[i] ≠ −color then
             f[i] := color;
             counter := counter + 1;
             for all j ∈ NG(i) do
                if f[j] < 1 then
                   f[j] := −color
                endif
             endfor
          endif
       endfor
    endwhile
    return f;
    return color;
endalgorithm greedyColoring.
```

Colors = { R: Red, G: Green, B: Blue}



Let $G = (V, E)$ be a graph with $|E| = m$, then it applies

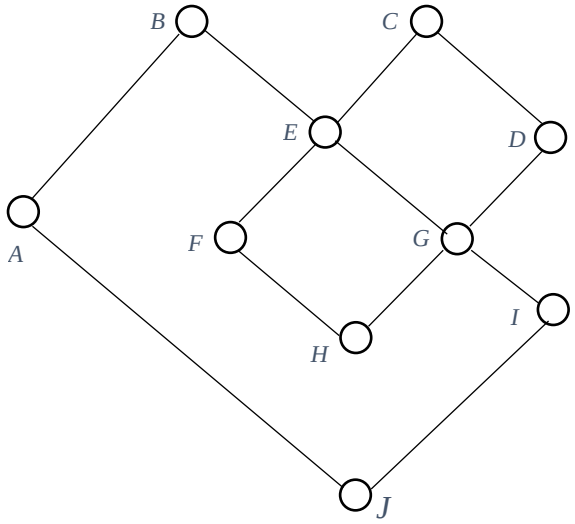
`greedyColoring(G).farbe` $\leq \lceil \sqrt{2m} \rceil$

Johnson Coloring Algorithm:

```
algorithm johnsonColoring;
    input G = (V, E) : Graph;
    output f : [1 ... max] of int;
           color : int;
    var U, W : subset of V;
    f := 0; color := 0; W := V;

    repeat
        U := W;
        color := color + 1;
        while U ≠ ∅ do
            Determine node u in GIS with minimal grade;
            f[u] := color;
            U := U − {u} − NU(u);      // NU(u): Neighbors of u
            W := W − {u};
        endwhile
    until W = ∅;
    return f;
    return color;
endalgorithm johnsonColoring.
```

Colors = { R: Red, G: Green, B: Blue}



Let $G = (V, E)$ be a graph with $|V| = n$, then it applies

`johnsonColoring(G).farbe` $\leq \left\lfloor \frac{4n * \log(\chi(G))}{\log n} \right\rfloor$

Some examples for approximation problems:

(1) Fully approximable:

a. Triangle Traveling Salesman Problem

b. 2-Processor-Scheduling

(2) Partially approximable:

a. Vertex/Node Cover

(3) Not approximable:

a. Traveling Salesman Problem

b. Coloring Problem