INF561: Using randomness in algorithms **Spring 2012**

Lecture 2 — 16th of January

Lecturer: Fréderic Magniez *Scribe: Yan Wang and Bart van Merriënboer*

2.1 Pattern Matching

Let $u \in \{0,1\}^m$ be a pattern and $w \in \{0,1\}^n$ a word where $m \leq n$. The task is to find the starting points i of occurences of the pattern u in w i.e. all the i such that $\{w(i), w(i+1), \ldots, w(i+m-1)\} = u$.

A trivial greedy algorithm consists of verifying bits at each position i. This deterministic algorithm has a complexity of $O((n-m+1)m)$. Using automata we can obtain a better deterministic algorithm with complexity $O(m+n)$. Alternatively we can use fingerprinting to construct a randomised algorithm with the same complexity.

We start by defining the polynomial P_u of the pattern u as

$$P_u = \sum_{j=1}^{m} u(j)\, x^{j-1}$$

which allows us to compute fingerprints of u. We can also compute the fingerprint of a subset $v_i = \{w(i), \ldots, w(i+m-1)\}$ of w to compare with P_u. Our algorithm makes use of the fact that given the polynomial P_{v_i} it is very easy to calculate the polynomial $P_{v_{i+1}}$ since

$$\begin{aligned} P_{v_i} &= w(i) + w(i+1)\, x + \ldots + w(i+m-1)\, x^{m-1} \\ P_{v_{i+1}} &= w(i+1) + w(i+1)\, x + \ldots + w(i+m)\, x^{m-1} \\ &= \frac{1}{x} P_{v_i} - w(i) + w(i+m)\, x^{m-1} \end{aligned}$$

For computational reasons we prefer to multiply instead of divide, so we use $P_{v_i} = P_{v_{i+1}} x + w(i) - w(i+m)\, x^{m-1}$ and scan the word in reverse. We pick a prime p such that $n^3 \leq p \leq 2n^3$ and test whether the fingerprints are identical by checking whether $(P_{v_i} - P_u)(a) \mod p = 0$ for random sampled values of $a \in_R \{0, \ldots, p-1\}$.

Algorithm 1 Pattern matching

Input $u \in \{0,1\}^m$ and $w \in \{0,1\}^n$ where $m \leq n$
Output $\{i \mid \{w(i), w(i+1), \ldots, w(i+m-1)\} = u\}$

```
p ← prime number between n^3 and 2n^3
a ← random element from S = {0, ..., p − 1}
h_u ← 0
b ← 1
for i = 1 to m do                          ▷ Calculate the fingerprint P_u(a) mod p
    h_u ← h_u + b · u(i) mod p
    b ← b · a mod p
end for
h_v ← 0
b ← 1
for i = 1 to m do                          ▷ Calculate the fingerprint P_{v_{n−m+1}}(a) mod p
    h_v ← h_v + b · w(n − m + i) mod p
    b ← b · a mod p
end for
if h_u = h_v then                          ▷ Check whether the fingerprints match
    return n − m + 1
end if
c ← a^{m−1} mod p
for i = 1 to n − m do                      ▷ Move to the left and update the fingerprint
    h_v ← (h_v − c · w(n − i + 1)) · a + w(n − m + 1 − i)
    if h_u = h_v then
        return n − m + 1 − i
    end if
end for
```

Looking at the size of the for-loops we can conclude that the complexity of the algorithm is $O(n+m) = O(n)$. It is a one-sided error algorithm because if there is a pattern occurence, the fingerprints will always match and the algorithm will return the position i of the match. If they do not match, the Schwartz-Zippel tells us that the probability of the fingerprints matching is $\mathbb{P}(h_u = h_v \mid u \neq v) \leq \frac{m}{p} \leq \frac{n}{n^3} = \frac{1}{n^2}$. The algorithm compares a total of $n-m+1$ fingerprints, so it follows from Boole's inequality that the probability of at least 1 of the returned values of i being incorrect is less than or equal to $\sum_{n-m+1} \frac{1}{n^2} = O\left(\frac{1}{n}\right)$.

2.2 Commutativity testing

Let G be a finite group and $h_1, h_2, \ldots, h_n \in G$. We wish to find an algorithm to determine whether the group elements $h_1, h_2, \ldots, h_n \in G$ commute, minimising the number of group operations required to do so. Hence, the algorithm accepts if $h_i \circ h_j = h_j \circ h_i$ for $1 \leq i, j \leq n$ and rejects if not.

Let $H = \langle h_1, \ldots, h_n \rangle$ be the subgroup $H \subseteq G$ generated by the elements $h_1, h_2, \ldots, h_n \in G$. Note that H is abelian if and only if the generators $h_1, h_2, \ldots, h_n$ commute.

Lemma 2.1. *If H is non-abelian then*

$$\mathop{\mathbb{P}}_{h,k\in H}(h\circ k\neq k\circ h)\geq\frac{1}{4}$$

Proof: If H is non-abelian then the center $Z(H)=\{h\in H\mid\forall k\in H,h\circ k=k\circ h\}$ is a proper subgroup of H. Let $h\notin Z(H)$, then the centralizer $C_H(h)=\{k\in H\mid h\circ k=k\circ h\}$ is also a proper subgroup of H. By Lagrange's theorem the orders of $Z(H)$ and $C_H(h)$ must divide H, so they are at most $\frac{1}{2}|H|$. Hence

$$\begin{aligned}\mathop{\mathbb{P}}_{h,k\in H}(h\circ k\neq k\circ h)&=\mathop{\mathbb{P}}_{h,k\in H}(h\notin Z(H)\wedge k\notin C_H(h))\\&=\mathop{\mathbb{P}}_{h,k\in H}(k\notin C_H(h)\mid h\notin Z(H))\mathop{\mathbb{P}}_{h\in H}(h\notin Z(H))\\&\geq\frac{1}{2}\cdot\frac{1}{2}=\frac{1}{4}\end{aligned}$$

□

For our algorithm we need a method to to sample randomly from H, which is a difficult problem. We will prove here that for a random string of bits $u\in_R\{0,1\}^n$ the element $h(u)=h_1^{u_1}\circ\ldots\circ h_n^{u_n}$ is random in the sense that for any proper subgroup K of H

$$\mathop{\mathbb{P}}_{u\in\{0,1\}^n}\Pr(h(u)\in K)\leq\frac{1}{2}$$

in which case our lemma holds.

Proof: Since K is a proper subgroup there exists $h_i\notin K$. Let i be the smallest number such that $h_i\notin K$ and write

$$h(u)=\left(h_1^{u_1}\circ\ldots\circ h_{i-1}^{u_{i-1}}\right)\circ h_i^{u_i}\circ\left(h_{i+1}^{u_{i+1}}\circ\ldots\circ h_n^{u_n}\right)=l\circ h_i^{u_i}\circ t$$

Note that $l\in K$ by definition of i. We now have the two cases

$$\begin{aligned}t\in K\mathop{\mathbb{P}}_{u\in\{0,1\}}((l\circ h_i^{u_i}\circ t)\in K)&=\frac{1}{2}\\t\notin K\mathop{\mathbb{P}}_{u\in\{0,1\}}((l\circ h_i^{u_i}\circ t)\in K)&\leq\frac{1}{2}\end{aligned}$$

which allows us to conclude that $\mathop{\mathbb{P}}_{u_i\in\{0,1\}^n}(h(u)\in K)\leq\frac{1}{2}$.

□

With this sampling method we can now write an algorithm with a one-sided error of $\frac{1}{4}$ that requires an average of $\frac{n}{2}+2=O(n)$ binary operations, with an upper bound of $n+2$. In comparison, the deterministic algorithm requires $2\binom{n}{2}=n(n-1)=O(n^2)$ binary operations to compare all possible pairs.

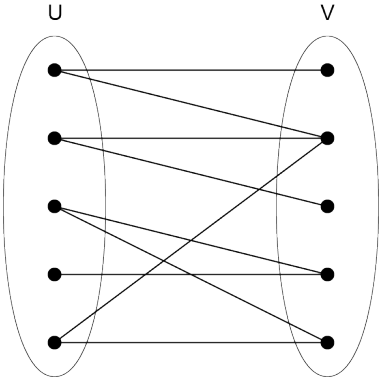
Algorithm 2 Commutativity testing

Input $h_1, \ldots, h_n \in G$
Output boolean indicating whether $h_i \circ h_j = h_j \circ h_i$ for all $1 \leq i, j \leq n$

```
u, v ← random {0,1}^n
h ← h_1^{u_1} ∘ ... ∘ h_n^{u_n}
k ← h_1^{v_1} ∘ ... ∘ h_n^{v_n}
if h ∘ k = k ∘ h then
    return true
else
    return false
end if
```

2.3 Perfect matching in bipartite graphs

Definition 2.2. *A* balanced bipartite graph $G = (U, V, E)$ *is specified by two disjoint sets of vertices,* U *and* V *where* $|U| = |V| = n$*, and a set of edges* E *between them.*



Definition 2.3. *A* perfect matching *is a subset of the set* E *such that every vertex of the graph is incident to exactly one edge of the matching.*

The best known deterministic algorithm to find a perfect matching of G is the Hopcroft–Karp algorithm which has a complexity of $O(\sqrt{|U \cup V|}\,|E|) \leq O\left(n^{5/2}\right)$. However, using matrix representation of the graph G we can find randomised algorithms which perform better. Let $U = \{1, \ldots, n\}$ and $V = \{1, \ldots, n\}$ then the biadjacency matrix A representing G has entries

$$A_{ij} = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases} \quad i \in U, j \in V$$

This matrix has the property that its permanent is equal to the number of perfect matchings of G.

$$\operatorname{perm}(A) = \sum_{\sigma \in S_n} \prod_{i=1}^{n} A_{i,\sigma(i)}$$

However, computing the permanent of a matrix is #P-complete, and the fastest deterministic algorithm to do so is Ryser's formula which requires $O\left(2^n n\right)$ operations. The fastest

known approximation algorithm due to Jerrum, Sinclair and Vigoda, has a complexity of approximately $O(n^{10})$. So instead, we consider the determinant of A, which is easier to compute

$$\det(A) = \sum_{\sigma \in S_n} \operatorname{sgn}(\sigma) \prod_{i=1}^{n} A_{i,\sigma(i)}$$

From the definition we can see that if $\det(A) \neq 0$ there exists at least one perfect matching. However, the converse is not true, so instead of considering the biadjacency matrix we define the Tutte matrix T of G as the $n \times n$ matrix with the entries

$$T_{i,j} = \begin{cases} x_{ij} & \text{if } (i,j) \in E \\ 0 & \text{if } (i,j) \notin E \end{cases} \quad 1 \leq i,j \leq n$$

Now $\det(T)$ is a polynomial with $|E|$ variables $\mathbf{x} = \{x_{ij} \mid 1 \leq i,j \leq n\}$, a degree $d \leq n$, and coefficients of 0 and 1. Moreover, we have

$$\det(T) = 0 \iff \nexists \text{ perfect matching of } G$$

where 0 is the zero polynomial.

Proof: If $\det(T) \neq 0$ then $\exists\, \sigma \in S_n$ such that $\prod_{i=1}^{n} T_{i,\sigma(i)} \neq 0$. This means that for this permutation $(i, \sigma(i)) \in E$ for all $1 \leq i \leq n$. Hence the set $M = \{(i, \sigma(i))\}$ is a perfect matching.

If M is a perfect matching, then consider the permutation $\sigma = \{\sigma(i) = j \mid (i,j) \in M\}$ where $\sigma \in S_n$. Since $M \subseteq E$ this implies that $\prod_{i=1}^{n} T_{i,\sigma(i)} \neq 0$ and hence $\det(T) \neq 0$ because of the unique decomposition of $\det(T)$.

□

We can design a decision algorithm that determines whether there exist perfect matchings of G by checking whether the determinant of its Tutte matrix is identical to the zero polynomial. To do so, we use the Schwartz-Zippel lemma with $S = \{0, \ldots, p-1\}$ where p is prime and $n^2 \leq p \leq 2n^2$. We reduce the space complexity of the algorithm by testing whether $\det(T) \mod p = 0$ instead of $\det(T) = 0$. We use $\det(T(\mathbf{a}))$ to denote $\det(T|_{\mathbf{x}=\mathbf{a}})$.

Algorithm 3 Perfect matching of a bipartite graph

Input $E \subseteq \{1, \ldots, n\} \times \{1, \ldots, n\}$
Output boolean indicating whether there exists a perfect matching of G

```
p ← prime number between n^2 and 2n^2
T ← Tutte matrix of G
a ← |E| random elements from S = {0, ..., p − 1}
if det(T(a)) mod p = 0 then
    return false
else
    return true
end if
```

By the Schwartz-Zippel lemma we know that the chance of rejecting even though the determinant is equal to the zero polynomial is less than or equal to $\frac{d}{|S|} \leq \frac{n}{n^2} = \frac{1}{n}$. Furthermore, the complexity of this algorithm depends on calculating the determinant of an $n \times n$ matrix, requiring the same time as matrix multiplication, which Coppersmith-Winograd algorithms can do in $O(n^{2.3727})$.

2.4 *st*-connectivity

Undirected *st*-onnectivity (USTCON) is the decision problem asking whether two vertices $(s,t) \in V^2$ in an undirected graph $G = (V, E)$ are connected by a path. Let $|V| = n$ and $|E| = m$. Note that if we assume the graph is connected we have $n - 1 \leq m \leq \frac{n(n-1)}{2}$.

The deterministic depth-first search (DFS) and breadth-search first (BFS) algorithms can solve this problem in linear time complexity $O(m+n) = O(m)$ because in the worst case scenario they will visit every edge and vertex. Their space complexity is $O(n)$ as the algorithms need to store their past and future search paths in the graph. The space complexity of these algorithms can be problematic for very large graphs (for example, the Internet graph). Therefore, we wish to find an algorithm with a smaller space complexity.

Definition 2.4. L *(logarithmic-space, also known as* LSPACE*) is the complexity class containing decision problems which can be solved by a deterministic Turing machine using a logarithmic amount of memory space.*

Definition 2.5. NL *(non-deterministic logarithmic-space, also known as* NSPACE*) is the complexity class containing decision problems which can be solved by a non-deterministic Turing machine using a logarithmic amount of memory space.*

Definition 2.6. RL (randomized logarithmic-space), *sometimes called* RLP (randomized logarithmic-space polynomial-time), *is the complexity class of computational complexity theory problems solvable in logarithmic space and polynomial time with probabilistic Turing machines with one-sided error.*

Note that L $\subseteq$ RL $\subseteq$ NL. Considering the space complexity of the DFS and BFS algorithms we know that USTCON $\in$ NL. In fact, it has been shown in 2005 by Reingold that USTCON $\in$ L. However, here we will only prove the following theorem :

Theorem 2.7. *USTCON* $\in$ *RL.*

To do so we consider a randomised algorithm.

Algorithm 4 USTCON Las Vegas Algorithm

Input $s, t \in V$
Output boolean indicating whether there exists a path from s to t

```
u ← s
while u ≠ t do
    v ← random element from {v | (u, v) ∈ E}
    u ← v
end while
return true
```

The space complexity of this algorithm is $O(\log n)$ because it suffices to keep the current vertex in memory, which requires a maximum of $\log_2(n)$ bits. If s and t are not connected then the algorithm never terminates. If there exists a path from s to t in G then the expected number of steps the algorithm requires is clearly less than or equal to the cover time $C(G, u)$, which is defined as the expected number of steps it takes to visit every vertex in the graph, starting at u.

Theorem 2.8. *Define $C(G)$ as $\max_{u \in V} C(G, u)$. We have $C(G) \leq 4|V||E| = 4nm$*

Proof: To prove this we shall consider the random walk on a graph as a Markov chain with a state space V and a transition matrix P where

$$P_{ij} = \begin{cases} \frac{1}{\deg(i)} & (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Let T be a totally ordered set (the steps in the random walk) and $X_t \in V$ the position of the random walk at step $t \in T$. Since all states $v \in V$ are positive recurrent (every vertex will be visited an infinite number of times, and the expected time to do so is finite) there exists a unique stationary distribution, described by a vector $\pi = (\pi_1, \ldots, \pi_n)$ such that $0 \leq \pi_i \leq 1$ for all $i \in V$ and

$$\sum_{i \in V} \pi_i = 1$$

$$\pi_i = \sum_{(i,j) \in E} \pi_j \frac{1}{\deg(j)}$$

Which is equivalent to $\pi P = P$. In fact, the stationary distribution is given by $\pi_i = \frac{\deg(i)}{2m}$.

For $u, v \in G$ we define the hitting time (or first hit time) as the average time it takes for a random walk on G starting at u to reach v :

$$h_{u,v} = \mathbb{E}(\inf\{t \in T \mid X_t = v, X_0 = u\})$$

The return time of u, defined as $h_{u,u}$, is actually related to the stationary distribution π_u and satisfies

$$h_{u,u} = \frac{1}{\pi_u} = \frac{2m}{\deg(u)}$$

The intuition behind this lies in the probability of $X_t = u$ being constant, and hence resembling a series of Bernoulli trials (biased coin tosses). The expected number of steps needed to get one success is then given by the expected value of the geometric distribution, which is $\frac{1}{p}$.

We can establish an upper bound on the hitting time $h_{u,v}$ where $(u, v) \in E$.

$$\begin{aligned} h_{v,v} &= 1 + \frac{1}{\deg(v)} \sum_{(w,v) \in E} h_{w,v} \\ &\geq 1 + \frac{1}{\deg(u)} h_{u,v} \\ h_{u,v} &\leq (h_{v,v} - 1)\deg(v) \\ &\leq 2m \end{aligned}$$

Now fix $u \in V$. By performing a depth-first search of graph G starting a u we obtain a spanning tree of G which has n vertices and $n-1$ edges. We can construct a tour $T = (u_1 = u, u_2, \ldots, u_N = u)$ of the tree such that $(u_i, u_{i+1}) \in E$ and so that it covers all the vertices of G. This tour passes each edge of the spanning tree twice, therefore $N \leq 2(n-1)$.

Note that $C(G)$ is less than or equal to the average time it takes to travel from u_1 to u_2, then from u_2 to u_3, etc. So $C(G) \leq h_{u_1,u_2} + \ldots + h_{u_{N-1},u_N}$. For all i we have $h_{u_i,u_{i+1}} \leq 2m$, and thus

$$C(G) \leq (N-1) \cdot (2m) < 4nm$$

□

We can use this upper bound of the expected runtime of algorithm 4 to convert it into a Monte Carlo algorithm.

Algorithm 5 USTCON Monte Carlo Algorithm

Input $s, t \in V$, $T_{max} \in \mathbb{Z}^+$
Output boolean indicating whether there exists a path from s to t

```
u ← s
while u ≠ t and number of iterations ≤ T_max do
    v ← random element from {v | (u, v) ∈ E}
    u ← v
end while
if u = t then
    return true
else
    return false
end if
```

If there is no path between s and t, the algorithm will always return false. As a corollary to theorem 2.8 we have

Corollary 2.9. *If $T \geq 8knm$, then algorithm 5 has a one-sided error less than or equal to 2^{-k}*

Proof: Using Markov's inequality and

$$C(G) = \mathbb{E}\left(\max_{u \in V} \min_{t \in T} \left\{ t \in T \mid \bigcup_{s=0}^{t} X_s = V, X_0 = u \right\}\right)$$

we can see that the probability of the algorithm returning false because it has not found t yet, even though there is a path between s and t at step $T \geq 8nm$ is

$$\mathbb{P}\left(\max_{u \in V} \min_{t \in T} \left\{ t \in T \mid \bigcup_{s=0}^{t} X_s = V, X_0 = u \right\} \geq 8nm\right) \leq \frac{C(G)}{8nm} < \frac{1}{2}$$

We can run our algorithm for a longer time $T \geq 8knm$, giving us a one-sided error of 2^{-k}, since

$$\begin{aligned}\mathbb{P}(t \text{ not reached at run } k) = \mathbb{P}(t \text{ not reached at run } k \mid t \text{ not reached at run } k-1) \cdot \ldots \\ \mathbb{P}(t \text{ not reached at run } 2 \mid t \text{ not reached at run } 1) \cdot \\ \mathbb{P}(t \text{ not reached at run } 1) \\ \leq \left(\frac{1}{2}\right)^k\end{aligned}$$

□

2.4.1 Examples

Linear Graph

Let $V = \{1, \ldots, n\}$ and $E = \{(i, i+1) \mid i \in \{1, \ldots, n-1\}\}$. By theorem 2.8 we have $C(G) \leq 4(n-1)n = O(n^2)$.

Complete Graph

In a complete graph, we have $C(G) \leq 4n \cdot \frac{n(n-1)}{2} = O(n^3)$. But in fact, we can prove a tighter bound of $C(G) \sim n \log n$.

Proof: In a complete graph the cover time is closely related to the coupon collector's problem. Let τ_i denote the first step at which i vertices have been visited. The number of steps it takes to reach a new vertex is

$$\tau_{i+1} - \tau_i = \frac{n-i}{n-1}$$

Since these events are independent we have

$$\mathbb{E}(\tau_{i+1} - \tau_i) = \frac{n-1}{n-i}$$

and we can use the approximation of the harmonic series by the natural logarithm to show that

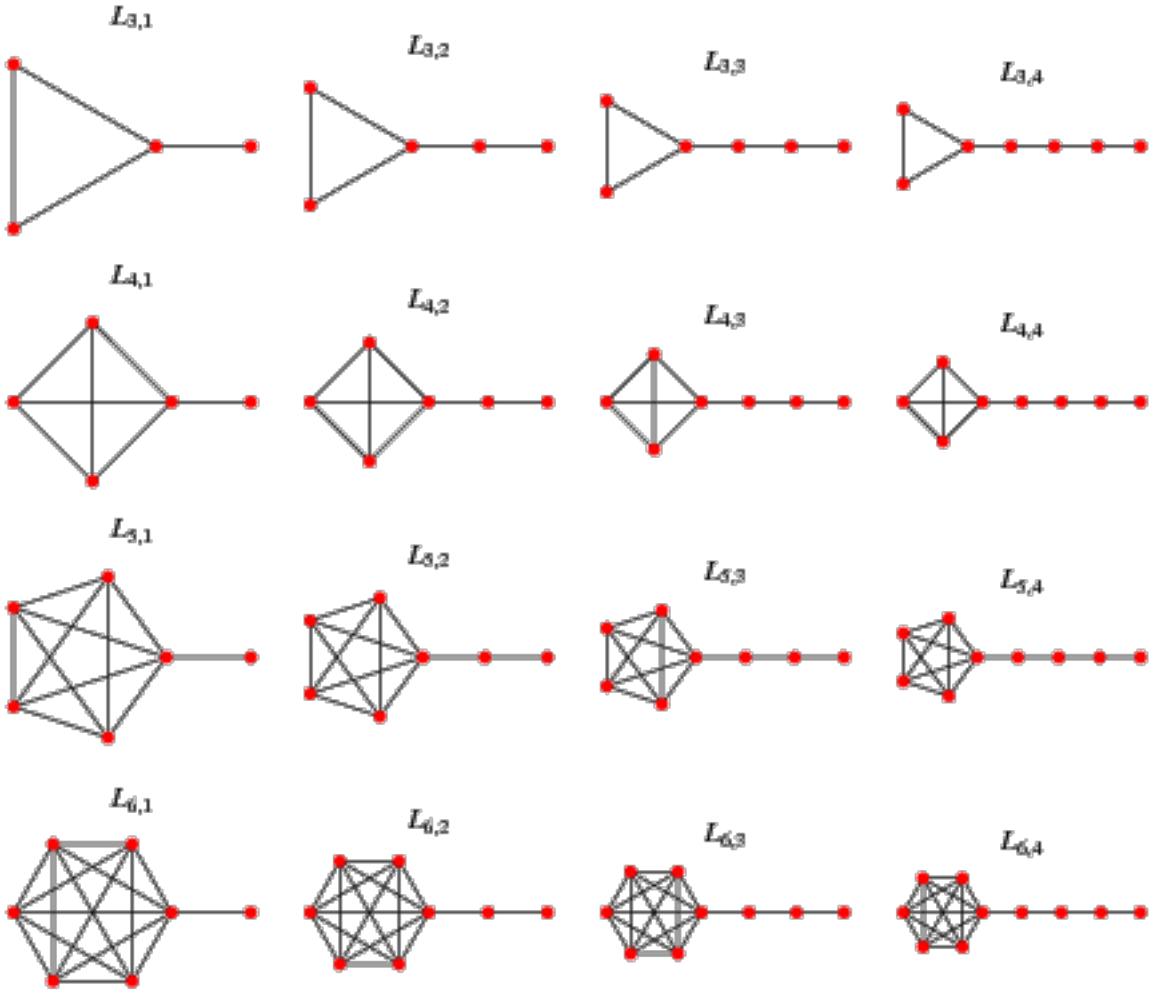
$$\mathbb{E}(\tau_n) = \mathbb{E}(\tau_1) + \sum_{i=1}^{n-1} \mathbb{E}(\tau_{i+1} - \tau_i) = 1 + \sum_{i=1}^{n-1} \frac{n-1}{n-i} = 1 + (n-1)\sum_{i=1}^{n-1} \frac{1}{i} \approx n \log n \text{ as } n \to \infty$$

□

Lollipop Graph

Lollipop graph is a graph as the conjunction of a linear graph (with $\frac{n}{2}$ vertices) and a complete graph (also with $\frac{n}{2}$ vertices). We deduce from the last section that : $C(G) \leq O(n^3)$.

The following graph is examples of Lollipop graphs.



Let s and t be left-most and right-most vertices of the linear graph. We can show that $h(s,t) = O(n^3)$ and $h(t,s) = O(n^2)$, which shows an interesting property of asymmetry in this graph.

2.5 Randomized algorithm for satisfiability : SAT

Definition 2.10. *Let $X_1, X_2, ..., X_n$ be $n \geq 1$ logic variables. A* literal *l is of the form X_i or $\overline{X_i}$ i.e. a literal is either a variable or the negation of a variable. A* clause *C is a disjunction of literals, for example $C = X_1 \vee X_2 \vee \overline{X_3}$. A clause C containing at most k variables is also called a* k-clause. *A* SAT formula *ϕ is of the form $\phi = C_1 \wedge C_2 \wedge ... \wedge C_m$ where C_i are clauses. A SAT formula ϕ is called a* k-SAT formula *if ϕ contains only k-clauses.*

The k-SAT is a decision problem to decide if for a given k-SAT formula there exist values of the boolean variables for which the formula is true.

We will pose the following theorems without proving them.

Theorem 2.11. *2-SAT $\in P$ and k-SAT is NP-complete for all $k \geq 3$*

Theorem 2.12. *2-SAT can be solved by a deterministic algorithm with complexity $O(n+m)$ where n is the number of variables and m is the number of clauses.*

The algorithm of the above theorem is first to construct a graph with literals of all variables, and then to check whether a variable x_i and $\overline{x_i}$ are contained in the same strongly connected component. Note that the second step can be completed within linear time using Tarjan's algorithm.

We are interested in designing a probabilistic algorithm to solve k-SAT which works for all k.

Algorithm 6 Random k-SAT algorithm

Input a k-SAT formula ϕ, clauses $C_1, \ldots, C_m$, and literals $(\neg)X_1, \ldots, (\neg)X_n$ ▷ $n = km$
Output a boolean indicating whether there exists an interpretation that satisfies ϕ
  $a \leftarrow$ any value in $\{0,1\}^n$ ▷ $a = (X_1, \ldots, X_n)$
  while $\phi(a) = 0$ **do**
    $j \leftarrow$ any integer such that $C_j(a) = 0$
    $i \leftarrow$ random integer from $\{k \mid X_k \text{ is a variable of } C_j\}$
    $X_k \leftarrow 1 - X_k$ ▷ Flip the bit of this variable in a
  end while
  return true

This algorithm will never terminate if there is no interpretation that satisfies ϕ i.e. $\phi(a) = 0$ for all a. If there is an interpretation satisfying ϕ, the algorithm will always find it, but there is no upper bound to its running time. Note that each iteration of the while-loop has a complexity of $O(mk)$ since it requires the evaluation of the entire formula, which has km literals.

Theorem 2.13. *If ϕ is 2-SAT and there exists an interpretation a such that $\phi(a) = 1$, then the average number of iterations needed to find a is $\leq 4n^2$.*

Corollary 2.14. *There exists an algorithm for 2-SAT with one-sided error 2^{-k} and running time $8kn^2$.*

Proof: For a 2-SAT problem let $s \in \{0,1\}^n$ such that $\phi(s) = 1$. Define $d(a,s) = |\{a_i \neq s_i \mid 1 \leq i \leq n\}|$. If $a = s$ then $d(a,s) = 0$ and in general $d(a,s) \in \{0, 1, ..., n\}$. Let $X_i = d(a,s)$ after i iterations.

If the algorithm has not stopped, we have

$$\mathbb{P}(X_{i+1} = n - 1 \mid X_i = n) = 1$$
$$\mathbb{P}(X_{i+1} = j - 1 \mid X_i = j) \geq \frac{1}{2} \quad \text{for} \quad 1 \leq j < n$$

The first statement is obvious because if all literals have the wrong value, changing one will always decrease the distance. For the case where $1 \leq j < n$ we can consider two cases :

– If C is a single literal, e.g. $C = \overline{X}_5 \vee \overline{X}_5$, the distance decreases with probability 1.

– If C has two literals, e.g. $C = X_2 \vee \overline{X}_7$, the distance decreases with probability $\geq \frac{1}{2}$, because at least 1 of the 2 bits is wrong, and there is a probability of $\frac{1}{2}$ that we flip the right one.

□

Note that the algorithm we have constructed is similar to a random walk on a line, where the upper bound of probability $\frac{1}{2}$ is the case where either direction is equally likely. We know that $h_{n,0}$ is upper bounded by $C(G) \leq 4n^2$.

As we did with the USTCON algorithms, we can bound the runtime of the 2-SAT algorithm by a time $8kn^2$ to construct an algorithm with a one-sided error of 2^{-k}.