Cpt S 515 Homework #6

Instructor: Zhe Dang

Student: Sheng Guan

11593929

1. Google uses a Markov chain to mimic a person browsing webpages. Actually, markov chains have many other applications as well. However, a difficult part is, at first place, a markov chain is hard to obtain for a practical application. Suppose that I treat a coffee maker as a markov chain (to simulate how one would use the machine). Notice that a modern coffee maker is quite complex and full of buttons to control almost everything. Suppose now that I think the coffee maker as a program and I want to test it. Can you give me a "random" test sequence (of buttons)? This is a difficult program. Open your mind.

Answer:

Based on my understanding, this question requires us to design test cases for the coffee maker based on Markov chain. In our case, we focus on weighted graphs, where a weight can model cost, time, etc. for an event. The computed probability of a path is proportional to its weight. If all paths with the same length take the same weight, it degenerates to the maximal entropy model.

The solution I would like to use is refer to “Random Words in a (Weighted) Regular Language: a Free Energy Approach”.

Where a weighted regular language is the collection of trajectories moving along the weighted graph defining the minimal deterministic finite automaton accepting the language. When the graph is strongly connected, one can compute a random walk – a unique Markov chain μ_* – that achieves the maximal free energy.

We first simulate every button as a state and pressing the button will be treated as transition of the state.

Let M be a DFA on alphabet Σ and with states in Q where q_0 is the initial state and $F \subseteq Q$ is the set of accepting state. We assume that M is cleaned up. That is, every state in M is reachable from the initial state and every state can reach an accepting state.

The way to generate a random word in L(M) is based on Gurevich's Algorithm, where the generated random words achieve the maximal free energy. We first make the M be strongly connected. Let ♦ $\notin \Sigma$ be a new symbol.

For each accepting state q, we add a transition to the initial state q_0:

$q \xrightarrow{♦} q_0$.

The resulting DFA is written M♦.

Next, we convert M♦ into a graph $G_{\bar{M}}$ as follows. Notice that DFA M itself may not be a graph: there could be multiple transitions from a node to another. Each state in M♦ is also a node in $G_{\bar{M}}$. Additionally, $G_{\bar{M}}$ has some other nodes shown below. Initially, $G_{\bar{M}}$has no edges. For each transition $p \xrightarrow{a} q$, with $a \in \Sigma \cup \{♦\}$, in M♦, we add a new node <p, a, q> and add the following two edges to $G_{\bar{M}}$:

– the edge, with weight w(a), from node p to node <p, a, q>;

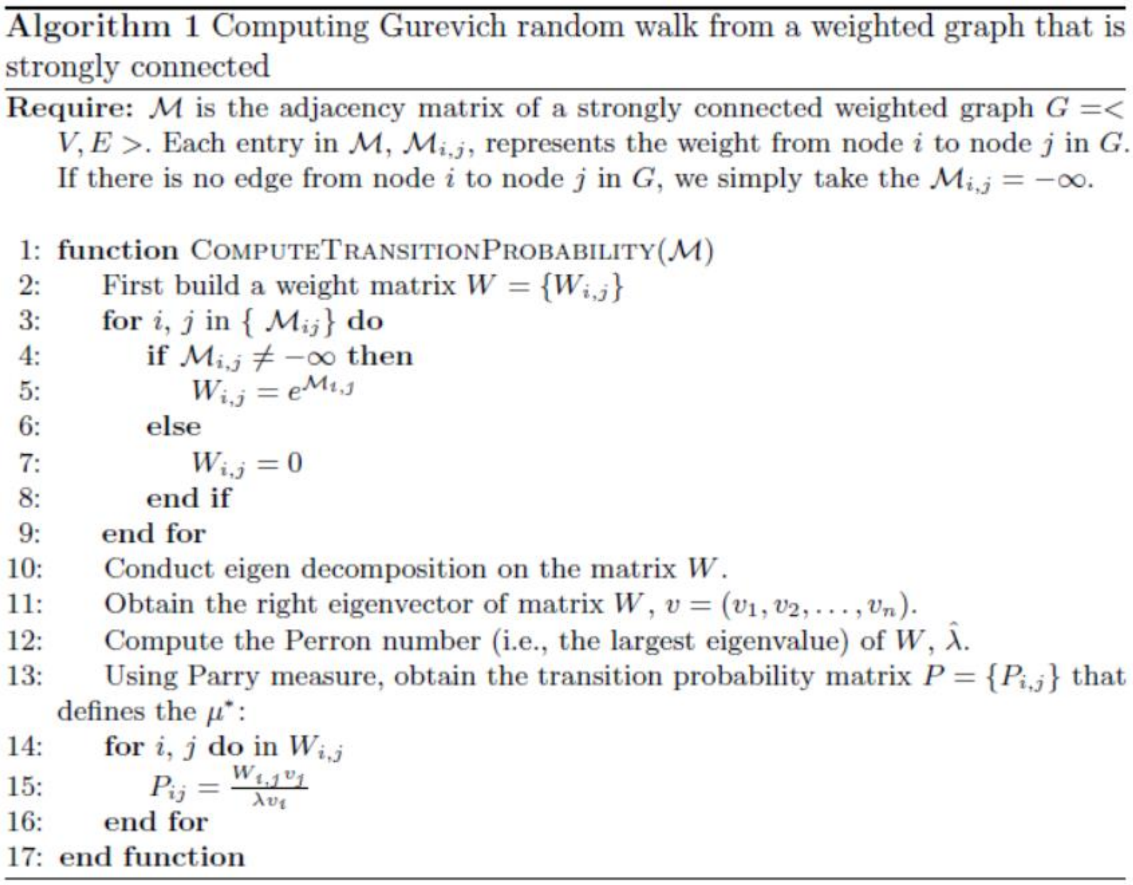
– the edge, with weight 0, from node <p, a, q> to node q.

Let G be a weighted graph. We assume that G is aperiodic and is an SCC. Let Gurevich Markov chain $\mu*$ obtained from running Algorithm 1 on the G be $p(\cdot)$, where for each edge t in G, p(t) is the transition probability of t, and w(t) is the given weight associated with t. Since the obtained $p(\cdot)$ is ergodic, we assume that $\pi(\cdot)$ is the unique stationary distribution where $\pi(v)$ is the stationary probability on node v in G, for each node v. We shall use $\pi(t)$ to denote $\pi(v)$ where v is the starting node of the edge t. A walk α of G is a sequence of edges

$\alpha = t_1 \cdot \cdot \cdot t_n$

for some n such that the ending node of t_i equals the starting node of t_{i+1} for each i. For the walk α, we use P(α) to denote the probability that α is actually walked; i.e., $P(\alpha) = \pi(t_1)p(t_1) \cdot \cdot \cdot p(t_n)$.

Let $\epsilon > 0$. We say that the walk α in is ϵ -typical (of G), if $|\lambda_\alpha - \lambda_G| \le \epsilon$, where λ_G is the free energy rate of G, and λ_α is the free energy rate of the walk α.



Algorithm 1 Computing Gurevich random walk from a weighted graph that is strongly connected

Require: $\mathcal{M}$ is the adjacency matrix of a strongly connected weighted graph $G = < V, E >$. Each entry in $\mathcal{M}$, $\mathcal{M}_{i,j}$, represents the weight from node i to node j in G. If there is no edge from node i to node j in G, we simply take the $\mathcal{M}_{i,j} = -\infty$.

```
1: function ComputeTransitionProbability($\mathcal{M}$)
2:     First build a weight matrix $W = \{W_{i,j}\}$
3:     for $i$, $j$ in $\{ \mathcal{M}_{ij} \}$ do
4:         if $\mathcal{M}_{i,j} \neq -\infty$ then
5:             $W_{i,j} = e^{\mathcal{M}_{i,j}}$
6:         else
7:             $W_{i,j} = 0$
8:         end if
9:     end for
10:    Conduct eigen decomposition on the matrix $W$.
11:    Obtain the right eigenvector of matrix $W$, $v = (v_1, v_2, \ldots, v_n)$.
12:    Compute the Perron number (i.e., the largest eigenvalue) of $W$, $\hat{\lambda}$.
13:    Using Parry measure, obtain the transition probability matrix $P = \{P_{i,j}\}$ that
   defines the $\mu^*$:
14:    for $i$, $j$ do in $W_{i,j}$
15:        $P_{ij} = \frac{W_{i,j} v_j}{\lambda v_i}$
16:    end for
17: end function
```

We only discuss one special case here that the minimal DFA M of the regular language L is already strongly connected and aperiodic, generate a long typical word from the prefix-closed L as follows:

Step 1. Construct the Markov chain $G_{\overline{M}}$(with transition probabilities in μ^*),
from the minimal automaton M for the L;
Step 2. Treat $G_{\overline{M}}$ as a probabilistic program and starts to walk from the
node q_0;
Step 3. When the program hits state F, goto Step 2. When the program hits
the length N, if the walk so far is indeed ($\in$, $\hat{\in}$,N), then return the word (in L) recovered from the walk as a typical word, else goto Step 2.

Typical words in the language are algorithmically generated which have applications in software engineering (test case generation). Here we only discuss one simple coffee maker program scenario that the DFA is already minimal, strongly connected and aperiodic. We can directly compute the transition probabilities, using the Markov chain that achieves the free energy rate of L(M), on transitions in M without introducing any $\hat{\epsilon}$-edges, which is a probabilistic program denoted as $\widehat{M}$. From this, we can see the test cases can be generated automatically when the $\widehat{M}$ is run as a probabilistic program.

2. Show that the following problem is NP-complete:
Given: a Boolean formula F
Question: Does F have even number of satisfying assignments?

Answer:
We can use the template introduced in the class to show that this problem is NP first.

```
Boolean myAlg(input F){
    y=Guess(F);
    //Guess an assignment for F with length=n (# of assignments in F)
    //Clearly, |y|≤P(|F|)
    If check(F,y)
    //When we get a set of satisfying assignments, to plug in the Boolean formula F
    //and verify the satisfying assignments is an even number **takes polynomial time and is deterministic,**
    //we just need to scan all given assignments and count the satisfied ones as a
    //counter, in the end to determine whether this counter is an even number or not.
        Return true;
    Crash;
}
```

Because the above problem satisfies this template, we can say the above problem has a Nondeterministic Algorithm. The problem is NP problem.
Next, we call this problem is an EvenNumber-SAT problem and reduction of 3SAT to EvenNumber-SAT: Given a 3cnf-function φ, create a new Boolean function φ' by adding

a new clause ($x \cup \bar{x}$) to φ, where x is a new variable not in φ. Then check if φ' ∈ EvenNumber-SAT.

This reduction clearly works in polynomial time.

We now need to prove that the original φ ∈ 3SAT iff the φ' ∈ EvenNumber-SAT. If the original φ is unsatisfiable, then the φ' is unsatisfiable; If φ ∈ 3SAT, then use the same assignment of literals in φ we will get x=0 and x=1 both work for φ' ∈ EvenNumber-SAT. Hence, there are even number of satisfying assignments of the φ' to make φ' ∈ EvenNumber-SAT. Similarly, when φ' ∈ EvenNumber-SAT, the same assignment of literals in φ′ except of x must make φ ∈ 3SAT. The proof is completed.

Hence, the problem is NP-complete.

3. Show that #3SAT is #P-complete.

Answer:

We first need to show that f ϵ # P.

1) #3SAT is to compute a function f:
   3SAT F → number of "solutions" to F.
2) 3SAT is an NP-complete problem.
   ∀ L ϵ NP, $L \leq_m$ 3SAT
   By #P definition, we can show #3SAT is the number of accepting runs of a NTM M' running in poly-time. Using Cook's reduction technique and we know we can do a polynomial time translation from SAT to 3SAT by treating a series of literals in one clause as one single literal to form 3-literals-only in a single clause, we will have a Q(x,y) where Q(x,y) is deterministic polynomial time computable and wrt the "input" x, the "output" y is not too long ($\widehat{\mathrm{M}}$ says yes/no on (x,y) in poly-time). Thus, Q(x,y) is an NP-predicate, #3SAT (that is f) ϵ # P

Then we need to show ∀ f' ϵ # P, $f' \leq_m f$. We first prove $f' \leq_m f_2$ where f_2 = #SAT, then we prove that f(x) = $M^{f_2}(x)$ (M is poly-time deterministic TM).

Consider a function f' ϵ # P. According to the definition, f' is the number of accepting runs of a non-deterministic poly-time TM M'

To show $f' \leq_m f_2$, we use $\forall x \in \sum^*$

$f'(x)$ = the number of y such that $\underline{\text{y is an accepting run of M' on x}}$

= the number of y such that Q(x,y)

We notice that $Q \in \mathrm{P}$, and using the Cook's reduction technique, there's a Boolean formula $\hat{Q}(\hat{x}, \hat{y})$ such that Q(x,y) iff $\hat{Q}(\hat{x}, \hat{y})$ and the mapping from Q to $\hat{Q}$ is deterministic poly-time and the translation x,y ->$\hat{x}, \hat{y}$ is 1-1.

$\forall x, |\{y:Q(x,y)\}| = |\{\hat{y} : \hat{Q}(\hat{x}, \hat{y})\}|$

f'(x) can be computed in terms of #3SAT($\hat{Q}(\hat{x},.)$) = #3SAT(g(x)) where g(x) = $\hat{Q}(\hat{x},.)$ a Boolean formula. g runs in poly-time.

Let M be a non-deterministic polynomial-time TM, then $\forall x \in \sum^*$,

f_M = the number of accepting runs of M on x

is in #P

We will get $f_M \leq_m f_2$.

Consider an example $(a_1 \lor a_2 \lor a_3 \lor a_4)$, to translate it as a 3SAT problem, we need to add additional Boolean literals to get $(a_1 \lor a_2 \lor y_1) \land (\overline{y_1} \lor a_3 \lor a_4)$. This translation will be done through polynomial time and y_i can be uniquely decided by a_i if all the clauses are satisfied. When we map clause $(u \lor v)$ to $(u \lor v \lor a)(u \lor v \lor \bar{a})$ we also choose some 3CNF formula containing a that has precisely one satisfying assignment and add it to F. Hence, there's a poly-time deterministic TM M that calls f_2 such that

f(x) = $M^{f_2}(x)$.

Hence, #3SAT is #P-complete.

4. (easy) Consider a 3SAT Boolean formula F which has variables x1, … ,xn. Recall that the F is a conjunction of clauses, and each clause is a disjunctions of three literals. A Boolean variable a shows up in a clause if either a or $\bar{a}$ is in the clause. A set C of variables is a cover of F if for each clause in F, there is a variable in C that shows up in the clause. Let k be the size of a cover C of a given F. Prove that the problem of 3SAT is fixed parameter tractable wrt k.

Answer:

We will change this 3SAT problem into a graph vertex cover problem.

Problem: A graph G=(V,E) and an integer k.

Question: Is there a vertex-cover C ⊆ V such that |C| = k and for each edge (u,v) ∈ E, at least either u or v in C ?

For each variable such as x in F, we will use the variable gadget described in the following: x ~ ¬ x

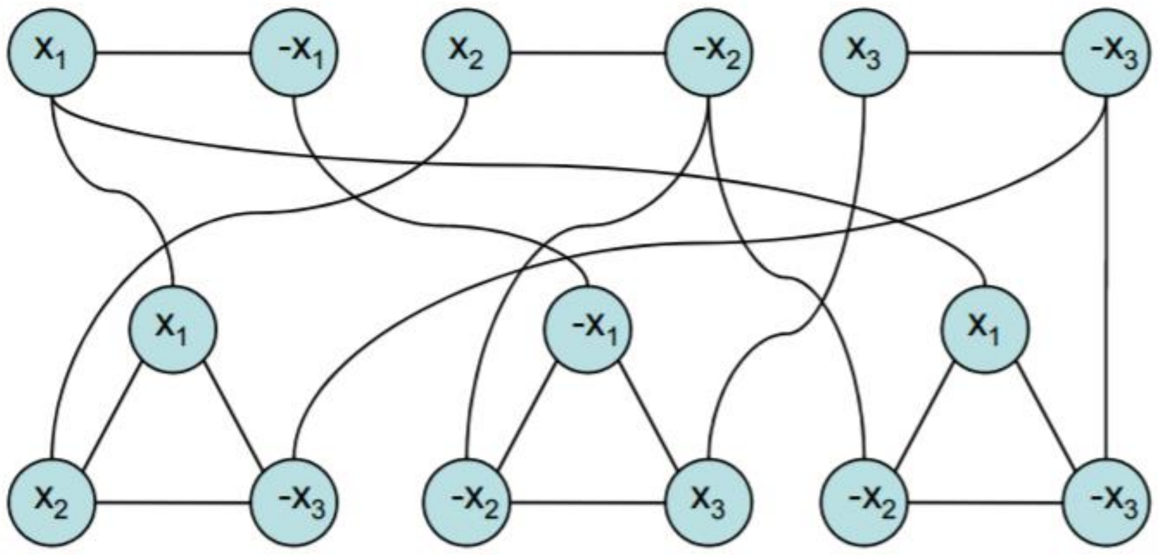
For each clause in F, we will connect the literals in the variable gadgets with an edge.

Now we do the reduction of 3SAT to vertex-cover: For the reduction, we are going to reduce an instance of 3SAT (a Boolean formula) to a vertex cover instance that has a cover if and only if the 3SAT formula has a satisfying assignment.

Suppose F contains m variables and l clauses. For the vertex cover, we will use k = m + 2l.

To illustrate our idea, we use the below example:

F = (x1vx2v¬x3) ∧(¬x1v¬x2vx3) ∧(x1 v¬x2v¬x3) then we are looking for a vertex cover of size k = 3 + 2*(3) = 9



To make 3SAT satisfiable, for each variable, we add the node corresponding to the true version of the literal for that variable to the vertex cover, we will get m vertices in our cover set first. Then, for each clause, select one true literal in the clause and add the remaining two literals to the vertex cover. We have used 2 vertices in each clause gadget and 1 vertex in each variable gadget for this cover, which meets the size requirement for the cover. Each edge in a variable gadget is covered by the node selected from that gadget. All three edges in each clause are covered by the two nodes we selected in that clause gadget. One true literal in each clause gadget is left out of the cover, but because it is a true literal it is connected by an edge to the node corresponding to the true literal in a variable clause. For example, in our case we select x_1, x_2, and x_3 as true literal, then for triangle gadget 1we will select x_2 and $\neg x_3$, for triangle gadget 2 we will select $\neg x_1$ and $\neg x_2$, for triangle gadget 3 we will select $\neg x_2$ and $\neg x_3$. We can easily verify that the 3SAT assignment satisfies and the nodes we select construct as a valid vertex cover.
If we have a vertex cover of size k for this graph, that cover must contain one node in each variable gadget and two nodes in each clause gadget to cover the edges in those gadgets. This requires exactly k = m+2l nodes. Suppose we take literal corresponding to a covered node in a variable gadget to be true. This assignment satisfies F because, for each clause gadget, each of the three edges connecting the clause gadget to the variable gadgets is covered, and only two nodes in the clause gadget are in the cover. Therefore one of these clause-gadget edges must be covered by a node in a variable gadget, and so the assignment of that covered variable gadget literal to be true in F will satisfy the clause. This holds for every clause gadget and clause, so this assignment satisfies. For example, in clause $(x_1 \vee x_2 \vee \neg x_3)$, the vertex cover chooses x_2, $\neg x_3$ in the cover, then to satisfy vertex cover in order to cover edge (x_1, x_1), the x_1 must be set as true which will satisfy clause $(x_1 \vee x_2 \vee \neg x_3)$. In the end, the F will satisfy iff the vertex cover holds.
Therefore a k-covering of this graph corresponds to a satisfying assignment for F, while a satisfying assignment for F corresponds to a k-covering of this graph, and this reduction is correct. Suppose F has m variables and l clauses, then k = m + 2l.
The problem of 3SAT is fixed parameter tractable wrt k is equivalent to the vertex-cover

problem is fixed parameter tractable wrt k.

Use the algorithm taught in the class:

```
Alg: Boolean VC(G,k)
    If G has no edges, return true;
    Take an arbitrary edge (u,v) in G
    Try VC(G-u, k-1) and VC(G-v,k-1)
    If one of the two returns true,
    Then return true;
    Else  return false;
```

Constructing G-u takes O(n)

Recursive tree size = O(2^k)

Hence, we have O(2^k*n). 3SAT is fixed parameter tractable wrt k.