Fall 2016 CS 374 Final Exam Note

Xiaoyan Wang

- The set of all programs is countable, the set of all problems is uncountable.
- P computes F is for every x, P(x) output F(x) and halts.
- Strings (definition: 1) empty string, 2) ax, where a is an element of Σ and x is a string

Length of string	Concatenation of string
$\lvert w\rvert := \begin{cases} 0 & \text{if } w = \varepsilon, \\ 1 + \lvert x\rvert & \text{if } w = ax. \end{cases}$	$w \bullet z := \begin{cases} z & \text{if } w = \varepsilon, \\ a \cdot (x \bullet z) & \text{if } w = ax. \end{cases}$

Proof: Let w be an arbitrary string.
Assume, for every string x such that $|x| < |w|$, that x is perfectly cromulent.
There are two cases to consider.

- Suppose $w = \varepsilon$.

  Therefore, w is perfectly cromulent.
- Suppose $w = ax$ for some symbol a and string x.
  The induction hypothesis implies that x is perfectly cromulent.

  Therefore, w is perfectly cromulent.

In both cases, we conclude that w is perfectly cromulent.

- Languages

Lemma 2.1: for all languages A, B and C

(a) $\emptyset A = A\emptyset = \emptyset$.

(b) $\varepsilon A = A\varepsilon = A$.

(c) $A + B = B + A$.

(d) $(A + B) + C = A + (B + C)$.

(e) $(AB)C = A(BC)$.

(f) $A(B + C) = AB + AC$.

Lemma 2.2. *The following identities hold for every language L:*

(a) $L^* = \varepsilon + L^+ = L^*L^* = (L + \varepsilon)^* = (L \setminus \varepsilon)^* = \varepsilon + L + L^+L^+$.

(b) $L^+ = L^* \setminus \varepsilon = LL^* = L^*L = L^+L^* = L^*L^+ = L + L^+L^+$.

(c) $L^+ = L^*$ *if and only if* $\varepsilon \in L$.

Lemma 2.3 (Arden's Rule). *For any languages A, B, and L such that $L = AL + B$, we have $A^*B \subseteq L$. Moreover, if A does not contain the empty string, then $L = AL + B$ if and only if $L = A^*B$.*

- Regular Language (Note: $L^+ = LL^*$)

Definition

- L is empty;
- L contains a single string (which could be the empty string ε);
- L is the union of two regular languages;
- L is the concatenation of two regular languages; or
- L is the Kleene closure of a regular language.

Regular Expression Tree

- A leaf node labeled $\emptyset$.
- A leaf node labeled with a string in Σ^*.
- A node labeled $+$ with two children, each of which is the root of a regular expression tree.
- A node labeled $\bullet$ with two children, each of which is the root of a regular expression tree.
- A node labeled $*$ with one child, which is the root of a regular expression tree.

Proof: Let R be an arbitrary regular expression.
Assume that every proper subexpression of R is perfectly cromulent.
There are five cases to consider.

- Suppose $R = \varepsilon$.

  Therefore, R is perfectly cromulent.
- Suppose R is a single string.

  Therefore, R is perfectly cromulent.
- Suppose $R = S + T$ for some regular expressions S and T.
  The induction hypothesis implies that S and T are perfectly cromulent.

  Therefore, R is perfectly cromulent.
- Suppose $R = S \bullet T$ for some regular expressions S and T.
  The induction hypothesis implies that S and T are perfectly cromulent.

  Therefore, R is perfectly cromulent.
- Suppose $R = S^*$ for some regular expression. S.
  The induction hypothesis implies that S is perfectly cromulent.

  Therefore, w is perfectly cromulent.

In both cases, we conclude that w is perfectly cromulent.

- DFA/NFA $M = (\Sigma, Q, \delta, s, F)$ Note: for DFA $\delta = Q \times \Sigma \rightarrow Q$, for NFA $\delta = Q \times \Sigma \rightarrow 2^Q = \mathbb{P}(Q)$

For two regular languages L and L'

- $\overline{L} = \Sigma^* \setminus L$ *is regular.*
- $L \cap L'$ *is regular.*
- $L \setminus L'$ *is regular.*
- $L \oplus L'$ *is regular.*

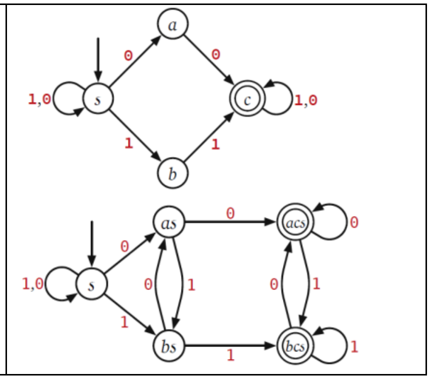
- Fooling set example

Lemma 3.7. *The language $L = \{ww^R \mid w \in \Sigma^*\}$ of even-length palindromes is not regular.*

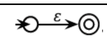
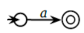
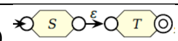
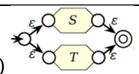
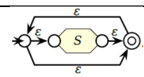
Proof: Let x and y be arbitrary distinct strings in 0^*1. Then we must have $x = 0^i1$ and $y = 0^j1$ for some integers $i \neq j$. The suffix $z = 10^i$ distinguishes x and y, because $xz = 0^i110^i \in L$, but $yz = 0^i110^j \notin L$. We conclude that 0^*1 is a fooling set for L. Because 0^*1 is infinite, L cannot be regular. $\square$

- NFA to DFA – subset construction example

q'	ε-reach(q')	$q' \in A'$?	$\delta'(q', 0)$	$\delta'(q', 1)$
s	s		as	bs
as	as		acs	bs
bs	bs		as	bcs
acs	acs	✓	acs	bcs
bcs	bcs	✓	acs	bcs



- <u>Regular Expression to DFA</u>

$R = \emptyset, L(R) = \emptyset$ 	$R = \epsilon, L(R) = \{\epsilon\}$ 
$R = a, L(R) = \{a\}$ 	$R = ST, L(R) = L(S) \cdot L(T)$ 
$R = S + T, L(R) = L(S) \cup L(T)$ 	$R = S^*, L(R) = L(S)^*$ 

- Context Free Grammar $G = (\Sigma, V, P, S)$ (Σ terminals, V non-terminals, P production rules, S start symbol)
- A string w is <u>ambiguous</u> with respect to a grammar if there is more than one parse tree for w, and a grammar G is <u>ambiguous</u> is some string is ambiguous with respect to G.
- A context-free language L is <u>inherently ambiguous</u> if every context-free grammar that generates L is ambiguous.
- Proof of correctness of grammar, example

> In fact, it is not hard to *prove* by induction that $L(C) = \{0^n 1^n \mid n \geq 0\}$ as follows. As usual when we prove that two sets X and Y are equal, the proof has two stages: one stage to prove $X \subseteq Y$, the other to prove $Y \subseteq X$.
>
> - First we prove that $C \rightsquigarrow^* 0^n 1^n$ for every non-negative integer n.
>
>   Fix an arbitrary non-negative integer n. Assume that $C \rightsquigarrow^* 0^k 1^k$ for every non-negative integer $k < n$. There are two cases to consider.
>   - If $n = 0$, then $0^n 1^n = \varepsilon$. The rule $C \to \varepsilon$ implies that $C \rightsquigarrow \varepsilon$ and therefore $C \rightsquigarrow^* \varepsilon$.
>   - Suppose $n > 0$. The inductive hypothesis implies that $C \rightsquigarrow^* 0^{n-1} 1^{n-1}$. Thus, the rule $C \to 0C1$ implies that $C \rightsquigarrow 0C1 \rightsquigarrow^* 0(0^{n-1}1^{n-1})1 = 0^n 1^n$.
>
>   In both cases, we conclude that that $C \rightsquigarrow^* 0^n 1^n$, as claimed.
>
> - Next we prove that for every string $w \in \Sigma^*$ such that $C \rightsquigarrow^* w$, we have $w = 0^n 1^n$ for some non-negative integer n.
>
>   Fix an arbitrary string w such that $C \rightsquigarrow^* w$. Assume that for any string x such that $|x| < |w|$ and $C \rightsquigarrow^* x$, we have $x = 0^k 1^k$ for some non-negative integer k. There are two cases to consider, one for each production rule.
>   - If $w = \varepsilon$, then $w = 0^0 1^0$.
>   - Suppose $w = 0x1$ for some string x such that $C \rightsquigarrow^* x$. Because $|x| = |w| - 2 < |w|$, the inductive hypothesis implies that $x = 0^k 1^k$ for some integer k. Then we have $w = 0^{k+1} 1^{k+1}$.
>
>   In both cases, we conclude that that $w = 0^n 1^n$ for some non-negative integer n, as claimed.

- Turing Machine $M = (Q, \Sigma, \Gamma, B, \delta, q_{start}, q_{accept}, q_{reject})$ (Note B or ☐ is the blank symbol, $\Sigma = \Gamma \backslash B$, $\delta = Q \times \Gamma(\text{read}) \to Q \times \Gamma(\text{write}) \times \{L, R\}$)
- M recognizes or accepts L if and only if M accepts every string in L but nothing else. (recursively enumerable language)
- M decides L if and only if M accepts every string in L and rejects every string in $\Sigma^* \backslash L$. A language is decidable (or computable or recursive) if it is decided by some Turing machine. (recursive language)

- MergeSort O(n log n), Median of Median O(n) (with large constant)
- Divide and conquer: split into n/c. Backtracking: split into n – c
- Dynamic Programming Rubric

> - 6 points for a correct recurrence, described either using mathematical notation or as pseudocode for a recursive algorithm.
>   - + 1 point for a clear **English** description of the function you are trying to evaluate. (Otherwise, we don't even know what you're *trying* to do.) **Automatic zero if the English description is missing.**
>   - + 1 point for stating how to call your function to get the final answer.
>   - + 1 point for base case(s). −½ for one *minor* bug, like a typo or an off-by-one error.
>   - + 3 points for recursive case(s). −1 for each *minor* bug, like a typo or an off-by-one error. **No credit for the rest of the problem if the recursive case(s) are incorrect.**
> - 4 points for details of the dynamic programming algorithm
>   - + 1 point for describing the memoization data structure
>   - + 2 points for describing a correct evaluation order; a clear picture is usually sufficient. If you use nested loops, be sure to specify the nesting order.
>   - + 1 point for time analysis

- Greedy Algorithm always need to be proved

> - Assume that there is an optimal solution that is different from the greedy solution.
> - Find the "first" difference between the two solutions.
> - Argue that we can exchange the optimal choice for the greedy choice without degrading the solution.

- Graph

Comparison of different representations

	Adjacency matrix	Standard adjacency list (linked lists)	Adjacency list (hash tables)
Space	$\Theta(V^2)$	$\Theta(V+E)$	$\Theta(V+E)$
Time to: Test if $uv \in E$	$O(1)$	$O(1 + \min\{\deg(u), \deg(v)\}) = O(V)$	$O(1)$
Test if $u \to v \in E$	$O(1)$	$O(1 + \deg(u)) = O(V)$	$O(1)$
List v's neighbors	$O(V)$	$O(1 + \deg(v))$	$O(1 + \deg(v))$
List all edges	$\Theta(V^2)$	$\Theta(V+E)$	$\Theta(V+E)$
Insert edge uv	$O(1)$	$O(1)$	$O(1)^*$
Delete edge uv	$O(1)$	$O(\deg(u) + \deg(v)) = O(V)$	$O(1)^*$

Traverse (with remembering)

```
TRAVERSE(s):
  put s into the bag
  while the bag is not empty
      take v from the bag
      if v is unmarked
          mark v
          for each edge vw
              put w into the bag
```

```
TRAVERSE(s):
  put (∅, s) in bag
  while the bag is not empty
      take (p, v) from the bag          (⋆)
      if v is unmarked
          mark v
          parent(v) ← p
          for each edge vw              (†)
              put (v, w) into the bag   (⋆⋆)
```

Stack: LIFO (DFS) $O(|V| + |E|)$. If graph is connected $O(|E|)$
Queue: FIFO (BFS) $O(|V| + |E|)$. If graph is connected $O(|E|)$
Priority queue: lightest out (shortest first search) $O(|V| + |E| \log|E|)$.
If graph is connected $O(|E| \log|E|)$

DFS $O(|V| + |E|)$ (for directed graph)

```
DFS(v):
  mark v
  PreVisit(v)
  for each edge vw
    if w is unmarked
      parent(w) ← v
      DFS(w)
  PostVisit(v)
```

```
DFSAll(G):
  for all vertices v
    unmark v
  for all vertices v
    if v is unmarked
      DFS(v)
```

```
DFS(v):
  mark v
  PreVisit(v)
  for each edge v→w
    if w is unmarked
      DFS(w)
  PostVisit(v)
```

Is it a DAG (directed acyclic graph)? Time $O(|V| + |E|)$

```
IsAcyclic(G):
  add vertex s
  for all vertices v ≠ s
    add edge s→v
    status(v) ← New
  return IsAcyclicDFS(s)
```

```
IsAcyclicDFS(v):
  status(v) ← Active
  for each edge v→w
    if status(w) = Active
      return False
    else if status(w) = New
      if IsAcyclicDFS(w) = False
        return False
  status(v) ← Done
  return True
```

Topological Sort (The order by which vertices are DONE in DFS is a reverse topological order).

```
ProcessBackward(G):
  add vertex s
  for all vertices v ≠ s
    add edge s→v
    status(v) ← New
  ProcessPostorderDFS(s)
```

```
ProcessPostorderDFS(v):
  status(v) ← Active
  for each edge v→w
    if status(w) = New
      ProcessPostorderDFS(w)
    else if status(w) = Active
      fail gracefully
  status(v) ← Done
  Process(v)
```

If known DAG

```
ProcessDagPostorder(G):
  add vertex s
  for all vertices v ≠ s
    add edge s→v
    unmark v
  ProcessDagPostorderDFS(s)
```

```
ProcessDagPostorderDFS(v):
  mark v
  for each edge v→w
    if w is unmarked
      ProcessDagPostorderDFS(w)
  Process(v)
```

Longest path $O(|V| + |E|)$ (on DAG)

```
LongestPath(s, t):
  for each node v in reverse topological order
    if v = t
      v.LLP ← ∞
    else
      v.LLP ← ∞
      for each edge v→w
        v.LLP ← max{v.LLP, ℓ(v→w) + w.LLP}
  return s.LLP
```

SCC (the vertex DONE in DFS is a source component of scc(G)) $O(|V| + |E|)$

```
KosarajuSharir(G):
  ⟨⟨Phase 1: Push in DFS finishing order⟩⟩
  unmark all vertices
  for all vertices v
    if v is unmarked
      RevPushDFS(v)
  ⟨⟨Phase 2: WFS in stack order⟩⟩
  unmark all vertices
  count ← 0
  while the stack is non-empty
    v ← Pop
    if v is unmarked
      count ← count + 1
      LabelOneWFS(v, count)
```

```
RevPushDFS(v):
  mark v
  for each edge v→u in rev(G)
    if u is unmarked
      RevPushDFS(u)
  Push(v)
```

```
LabelOneWFS(v, count):
  put v into the bag
  while the bag is not empty
    take v from the bag
    mark v
    label(v) ← count
    for each edge v→w in G
      if w is unmarked
        put w into the bag
```

Single source shortest path (SSSP)

```
InitSSSP(s):
  dist(s) ← 0
  pred(s) ← Null
  for all vertices v ≠ s
    dist(v) ← ∞
    pred(v) ← Null
```

```
GenericSSSP(s):
  InitSSSP(s)
  put s in the bag
  while the bag is not empty
    take u from the bag
    for all edges u→v
      if u→v is tense
        Relax(u→v)
        put v in the bag
```

Dijkstra's: using priority heap: $O(|E|\log|V|)$, Fibonacci heap $O(|E| + |V|\log|V|)$ (but no negative edge)

Shimbel-Bellman-Ford $O(|V||E|)$

```
ShimbelSSSP(s)
  InitSSSP(s)
  repeat V times:
    for every edge u→v
      if u→v is tense
        Relax(u→v)
  for every edge u→v
    if u→v is tense
      return "Negative cycle!"
```

All pair shortest path: Floyd-Warshall $O(|V|^3)$

```
FloydWarshall2(V, E, w):
  for all vertices u
    for all vertices v
      dist[u, v] ← w(u→v)
  for all vertices r
    for all vertices u
      for all vertices v
        if dist[u, v] > dist[u, r] + dist[r, v]
          dist[u, v] ← dist[u, r] + dist[r, v]
```

- P is the set of decision problems that can be solved in polynomial time
- NP is the set of decision problems with the following property: If the answer is Yes, then there is a proof of this fact that can be checked in polynomial time

- co-NP is essentially the opposite of NP. If the answer to a problem in co-NP is No, then there is a proof of this fact that can be checked in polynomial time.
- Every decision problem in P is also in NP and co-NP
- Π is NP-hard $\Leftrightarrow$ If Π can be solved in polynomial time, then P=NP
- a problem is NP-complete if it is both NP-hard and an element of NP
- any algorithm that runs on a random-access machine in $T(n)$ time can be simulated by a single-tape, single-track, single-head Turing machine that runs in $O(T(n)^4)$ time
- To prove X is NP-hard

1. Pick a known NP-hard problem Y
2. Assume for the sake of argument, a polynomial time algorithm for X
3. Derive a polynomial time algorithm for Y, using algorithm X as subroutine
4. Contradiction

- A clique is another name for a complete graph, that is, a graph where every pair of vertices is connected by an edge
- A vertex cover of a graph is a set of vertices that touches every edge in the graph.
- A Hamiltonian cycle in a graph is a cycle that visits every vertex exactly once
- definition of several languages

• The *accepting* language $\text{Accept}(M) := \{w \in \Sigma^* \mid M \text{ accepts } w\}$
• The *rejecting* language $\text{Reject}(M) := \{w \in \Sigma^* \mid M \text{ rejects } w\}$
• The *halting* language $\text{Halt}(M) := \text{Accept}(M) \cup \text{Reject}(M)$
• The *diverging* language $\text{Diverge}(M) := \Sigma^* \setminus \text{Halt}(M)$

- M accepts L: ACCEPT(M) = L
- M decides L: ACCEPT(M) = L and DIVERGE(M) = Ø

- Useful properties

Lemma 1. *Let M be an arbitrary Turing machine.*
(a) *There is a Turing machine M^R such that $\text{Accept}(M^R) = \text{Reject}(M)$ and $\text{Reject}(M^R) = \text{Accept}(M)$.*
(b) *There is a Turing machine M^A such that $\text{Accept}(M^A) = \text{Accept}(M)$ and $\text{Reject}(M^A) = \varnothing$.*
(c) *There is a Turing machine M^H such that $\text{Accept}(M^H) = \text{Halt}(M)$ and $\text{Reject}(M^H) = \varnothing$.*

Lemma 2. *If L and L' are decidable, then $L \cup L'$, $L \cap L'$, $L \setminus L'$, and $L' \setminus L$ are also decidable.*

Corollary 3. *The following hold for all languages L and L'.*
(a) *If $L \cap L'$ is undecidable and L' is decidable, then L is undecidable.*
(b) *If $L \cup L'$ is undecidable and L' is decidable, then L is undecidable.*
(c) *If $L \setminus L'$ is undecidable and L' is decidable, then L is undecidable.*
(d) *If $L' \setminus L$ is undecidable and L' is decidable, then L is undecidable.*

Lemma 4. *For all acceptable languages L and L', the languages $L \cup L'$ and $L \cap L'$ are also acceptable.*

Lemma 5. *An acceptable language L is decidable if and only if $\Sigma^* \setminus L$ is also acceptable.*

- Nature properties of encoding Turing machines: unique, modifiable, executable
- To prove that a language L is undecidable, reduce a known undecidable language to L (see example below)

Theorem 12. *Halt is undecidable.*

Proof: Suppose to the contrary that there is a Turing machine H that decides Halt. Then we can use H to build another Turing machine SH that decides the language SelfHalt. Given any string w, the machine SH first verifies that $w = \langle M \rangle$ for some Turing machine M (rejecting if not), then writes the string $ww = \langle M, M \rangle$ onto the tape, and finally passes control to H. But SelfHalt is undecidable, so no such machine SH exists. We conclude that H does not exist either. □

- Rice's theorem

Rice's Theorem. *Let $\mathcal{L}$ be any set of languages that satisfies the following conditions:*
• *There is a Turing machine Y such that $\text{Accept}(Y) \in \mathcal{L}$.*
• *There is a Turing machine N such that $\text{Accept}(N) \notin \mathcal{L}$.*

The language $\text{AcceptIn}(\mathcal{L}) := \{\langle M \rangle \mid \text{Accept}(M) \in \mathcal{L}\}$ is undecidable.

- The set L in the statement of Rice's Theorem is often called a property of languages
- NP-rubric

Rubric (for all undecidability proofs, out of 10 points):
Diagonalization:
+ 4 for correct wrapper Turing machine
+ 6 for self-contradiction proof (= 3 for ⇐ + 3 for ⇒)

Reduction:
+ 4 for correct reduction
+ 3 for "if" proof
+ 3 for "only if" proof

Rice's Theorem:
+ 4 for positive Turing machine
+ 4 for negative Turing machine
+ 2 for other details (including using the correct variant of Rice's Theorem)