Mathematical Structures CS

Final Exam: (100 pts)

Student Name: Amrit Singh

1. Question 1: Explain the concept of a context-free grammar (CFG). Provide an example of a CFG and demonstrate how it can be used to parse a simple arithmetic expression. (20pts)

A sentence is a string of characters over some alphabet. A language a set of sentences according to the rules of the language. Context-free grammar (CFG) is a language generator that describes the syntax of languages. It's a set of rules that can be applied to regardless of context. An equivalence is Backus-Naur form (BNF), a notation for CFGs in computer science.

Example:

1. <expression> → <expression> + <term> // addition
2. <expression> → <expression> - <term> // subtraction
3. <expression> → <term> // expression is a term
4. <term> → <term> * <factor> // multiplication
5. <term> → <term> / <factor> // division
6. <term> → <factor> // term is a factor
7. <factor> → (<expression>) // parentheses
8. <factor> → number // factor is a number

Using these CFGs, we're able to do arithmetic.

2. What is a parse tree? Construct a parse tree for the expression (a + a) * a using the context-free grammar provided in Question 1. (20 pts)

A parse-tree is a hierarchical representation of a derivation. It is a syntax tree diagram that show how an expression is derived from the rules of a CFG. Using the rules from #1, we can construct the parse tree for the expression (a + a) * a as the following (assuming a is number):

```
    <expression[(a + a) * a]>
            |
       <term[(a + a) * a]>
     |                      |
<term[(a + a)]>  *   <factor[a]>
           |                    |
<factor[(a + a)]             a
            |
<( expression )[(a + a)]>
           |                  \
<expression[a]>    +    <term[a]>
            |                     |
      <term[a]>              <term[a]>
            |                     |
      <factor[a]>           <factor[a]>
             |                     |
             a                     a
```

3. Given the sequence $a_n=2^n$ for $n \geq 0$, find the generating function A(x) for the sequence. Show all steps in your calculation. (20pts)

A generating function is a function that encodes a sequence as a serious of coefficients.

If n = 0; $a_0 = 2^0 = 1$
If n = 1; $a_1 = 2^1 = 2$
If n = 2; $a_2 = 2^2 = 4$
If n = 3; $a_3 = 2^3 = 8$

If n = 3; $a_4 = 2^4 = 16$
If n = 3; $a_5 = 2^5 = 32$

So, we have a sequence that looks like 1, 2, 4, 8, 16, 32, …. Our generating function should be:

$$A(x) = \sum_{k=0}^{\infty} c_k x^k = \sum_{k=0}^{\infty} 2^k x^k$$

This can be reduced to a geometric series. Observe the following for when |x| < ½ :

$$A(x) = \sum_{k=0}^{\infty} 2^k x^k = \sum_{k=0}^{\infty} (2x)^k = \frac{1}{1-2x}$$

Therefore, the generating function is $\frac{1}{1-2x}$.

4. Graph (20 pts)

a. How does graph coloring apply to bipartite graphs?

A graph is bipartite if the vertices can be divided into two disjoint sets, *A* and *B*, with no two vertices in *A* adjacent and no two vertices in *B* adjacent. There are no edges between any two vertices in the same set.

If we place a vertex in center of a region of a map, connect them, and assume the regions share a border, we get a graph. Coloring the regions of this map corresponds to coloring the vertices of that graph.

Graph coloring is related to bipartite graphs because you can assign two colors to each set of the graph as a bipartite graph is divided between two disjoint sets. Using two colors, we can color the graph such that no two adjacent vertices share the same color. This means that a bipartite graph is 2-colorable.

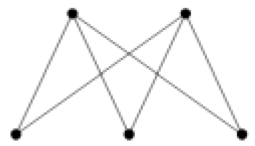
b. Prove that a bipartite graph is 2-colorable?

A bipartite graph, *G* is defined as a graph is bipartite if the vertices can be divided into two disjoint sets, *A* and *B*, with no two vertices in *A* adjacent and no two vertices in *B* adjacent. There are no edges between any two vertices in the same set.

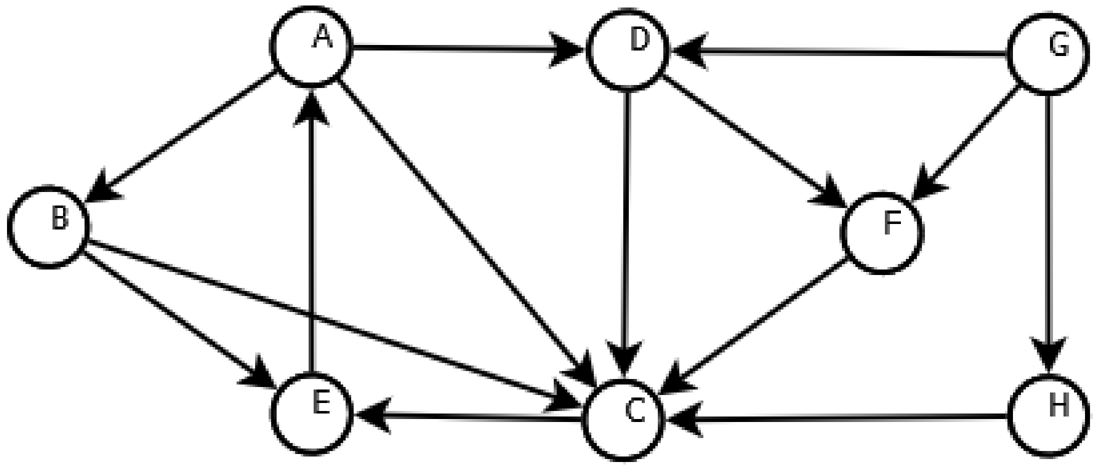
Let's assign a color, like red, to all the vertices of set *A* and assign another color, like blue, to all the vertices of set *B*. Because all the edges go between the two sets, no two adjacent vertices will be in the same set. Therefore, no edges connect two vertices of the same color.

Example:

- Top 2 vertices = red
- Bottom 3 vertices = blue



5. Graph (20 pts)



Breadth first search (BFS) is when you visit all vertices in the same generation, i.e visit all siblings of the root vertex, before any vertices of the next generation, then repeat.

Dept first search (DFS) is visiting one child of the root, then visit a child of that vertex, then visit its child. When you get to a vertex with no children, retreat to its parent and see if the parent has any other children, and repeat.

a. Do BFS for the following graph and find minimum spanning tree.

   Suppose we have a queue Q that's empty. To do BFS for this graph, let's start at vertex A (i.e the root) and enqueue it into our queue (Q: A). Dequeue A from the queue. Visit its unvisited siblings, B, C, and D. Let's add them to the queue (Q: B, C, D). Mark A as visited. Dequeue B from the queue. Visit its unvisited sibling, E, and add it to the queue (Q: C, D, E). Mark it as visited. Dequeue C from the queue, visit its unvisited siblings (which are none since we already visited C) so the queue is now Q: D, E. Dequeue D. Visit its unvisited sibling, F, and add it to the queue (Q: E, F). Mark it as visited. Dequeue E. Add its unvisited siblings (which are none!) and mark it as visited (Q: F). Dequeue F. Visit its unvisited siblings (none!). Mark it as visited.

   Visit order: A, B, C, D, E, F. We never visited G or H from root A.
   The minimum spanning tree does not exist for a directed, unweighted graph.

b. Do DFS for the following graph and find minimum spanning tree. (20 pts)

   Suppose we have a stack, S, that's empty. To do DFS for this graph, let's start at vertex A (i.e the root), and push it into our stack (S: A). Pop out A from the stack. Visit its sibling, B, and add to stack (S: B), marking A as visited. Pop out B from the stack, visit its unvisited sibling C and add to stack (S: C), mark B as visited. Pop out C from the stack, visit its unvisited sibling E and add to stack (S: E), mark C as visited. Pop out E from the stack, visit its unvisited sibling (none!) and add to stack (S: empty). Now let’s backtrack to the parent, which was A. Add another sibling to the stack (S: D). Pop out D from the stack, visit its unvisited sibling F and add to stack (S: F), mark D as visited. Pop out F from the stack, visit its unvisited sibling (none!) and add to stack (S: empty), mark F as visited. Backtrack, no more siblings to visit. We’re finished.

Visit order: A, B, C, E, D, F. We never visited G or H from root A.

The minimum spanning tree does not exist for a directed, unweighted graph.