McGill University COMP251: Assignment 3 Solution

Question 1 Idea: A number x in $\{0, 1, \ldots, n^3 - 1\}$ can be expressed in the form

$$a_2 n^2 + a_1 n + a_0$$

where $0 \leq a_2, a_1, a_0 \leq n - 1$. More precisely,

$$a_2 = \lfloor \frac{x}{n^2} \rfloor, a_1 = \lfloor \frac{x}{n} \rfloor - a_2 n, a_0 = x - n(a_2 n + a_1)$$

In other words, a_2, a_1, a_0 are the digits of x written in base n: $x = \overline{a_2 a_1 a_0}_n$.

Two numbers written in base n are compared just as in base 10: Suppose that $x = \overline{a_2 a_1 a_0}_n$ and $y = \overline{b_2 b_1 b_0}_n$ then $x > y$ if and only if

$$\begin{cases} a_2 > b_2 & \text{or} \\ a_2 = b_2 \text{ and } a_1 > b_1 & \text{or} \\ a_2 = b_2 \text{ and } a_1 = b_1 \text{ and } a_0 > b_0 \end{cases}$$

We can now use Radix sorting. Here the range of each digit is $k = n$, and each number has 3 digits. The running time for Radix sorting is $\Theta(d(n + k)) = \Theta(n)$ (d is the number of digits, here $d = 3$).

The algorithm: The algorithm Some-linear-sort below takes as input an array X of integers between 0 and $n^3 - 1$, sorts it in non-decreasing order, and store the result in X.

The algorithm consists of 3 main parts:

- Preparing for radix-sort (lines 1 to 9): here we compute the base-n representation for each element in X. Each $X[i]$ is represented as row $A[i]$ in a 2-dimensional array A.
- Radix sort the base-n representations (lines 10 to 37): There are 3 digits (thus the for-loop on line 11).
- Now convert the sorted base-n representations back and store in X (lines 38-41).

Some-linear-sort(X)

1. % initialization
2. $n \leftarrow length(X)$
3. A: an 2-dimensional array of size $n \times 3$
4. % first convert the numbers into base n and store the results in A. Suppose $X[i] = \overline{(a_2 a_1 a_0)}_n$, then $A[i][2] = a_2, A[i][1] = a_1, A[i][0] = a_0$
5. for i from 1 to n do
6. $\quad A[i][2] \leftarrow \lfloor \frac{X[i]}{n^2} \rfloor$
7. $\quad A[i][1] \leftarrow \lfloor \frac{X[i]}{n} \rfloor - nA[i][2]$

8. $A[i][0] \leftarrow X[i] - n(nA[i][2] + A[i][1])$
9. end for
10. % now radix sort
11. for *digit* from 0 to 2 do % counting sort on the *digit*-th digit
12. % initializes counting array C:
13. for j from 0 to $n-1$ do
14. $C[j] \leftarrow 0$
15. end for
16. % next make each $C[j]$ be the number of i such that $A[i][digit] = j$:
17. for i from 1 to n do
18. $C[A[i][digit]] \leftarrow C[A[i][digit]] + 1$
19. end for
20. % sum up: each $C[j]$ will be the number of i such that $A[i][digit] \leq j$:
21. for j from 1 to $n-1$ do
22. $C[j] \leftarrow C[j] + C[j-1]$
23. end for
24. % now put each row $A[i]$ into the right place (in B):
25. for i from n down to 1 do
26. $B[C[A[i]][0]] \leftarrow A[i][0]$
27. $B[C[A[i]][1]] \leftarrow A[i][1]$
28. $B[C[A[i]][2]] \leftarrow A[i][2]$
29. $C[A[i]] \leftarrow C[A[i]] - 1$
30. end for
31. % copy the array B back to A:
32. for i from 1 to n do
33. $A[i][0] \leftarrow B[i][0]$
34. $A[i][1] \leftarrow B[i][1]$
35. $A[i][2] \leftarrow B[i][2]$

36. end for

37. end for

38. % now get back the values from the base-n representations and output in X

39. for i from 1 to n do

40. $X[i] \leftarrow n^2 A[i][2] + nA[i][1] + A[i][0]$

41. end for

Question 2 (a) **Data structure**: We are using a data structure for unbounded branching tree. Each node in the tree has the following fields (here we don't need the parent):

1. left-child: pointer to its leftmost child (NIL if this node is a leaf)
2. right-sibling: pointer to its right sibling (NIL if this node the the rightmost child of its parent)
3. constant: Boolean value, TRUE if this is a leaf and contains a constant value (as oppose to a variable)
4. value: value of the subformula rooted at this node
5. index: if this node is represents a variable (i.e., it is a leaf and constant = FALSE), then the variable represented is $X[index]$
6. operator: one of the values $\{+, -, \times, \div\}$

(b) We give a recursive algorithm $eval(x)$ for evaluating a formula represented by the tree rooted at x, and store the value in $x.value$. (So if x represents a constant, i.e., x is a leaf and $x.constant = TRUE$, then we don't have to do anything). Assuming all the children of x have been evaluated, then we simply go through the list of its children, extract their values, and apply the operator of x. If x is an $+$ or $\times$ node then it might have more than two children, and we need to loop through them. Otherwise x has only two children and we simply apply the operator to their values.

(It is a good excercise to write a non-recursive algorithm using stack, and you should try it out yourself.)

The algorithm: $eval(x, X)$ takes as input a node x and the array X that specifies the values of the variables (i.e., $x_i = X[i]$) and return the value of the expression represented by x.

eval(x, X):

1. % the first case is when x is a variable
2. if x.left-child $= NIL$ and $x.constant = FALSE$ then $x.value \leftarrow X[x.index]$
3. % second case: if x is not a constant nor a variable, compute value of x recursively
4. else if x.left-child $\neq NIL$
5. $y \leftarrow x$.left-child

```
6.    while y ≠ NIL do
7.        eval(y)
8.        y ← y.right-sibling
9.    end while
10.   % now use the operator of x
11.   y ← x.left-child
12.   if x.op = "+" then
13.       v ← 0   % v will be the value of x
14.       while y ≠ NIL do
15.           v ← v + y.value
16.           y ← y.right-sibling
17.       end while
18.       x.value ← v
19.   else if x.op = "×" then
20.       v ← 1   % v will be the value of x
21.       while y ≠ NIL do
22.           v ← v × y.value
23.           y ← y.right-sibling
24.       end while
25.       x.value ← v
26.   else if x.op = "−" then
27.       z ← y.right-sibling
28.       x.value ← y.value − z.value
29.   else if x.op = "÷" then
30.       z ← y.right-sibling
31.       x.value ← y.value ÷ z.value
32.   end if
33. end if
```

(c) Again, we give a recursive algorithm, and it's an exercise to write a non-recursive algorithm using stack. If x is a constant then we print its value, otherwise if x is a variable then we print a letter X followed by its $x.index$. Finally, for other cases we

- print a "(",
- then print x's children recursively (separated by x's operator),
- then a ")".

The algorithm:
print(x)

1. % if x is a constant
2. if x.left-child $= NIL$ and $x.constant = TRUE$ then print $x.value$
3. % if x is a variable
4. else if x.left-child $= NIL$ and $x.constant = FALSE$ then print "X"; print $x.index$
5. % now if x is not a constant nor a variable, print it recursively
6. else
7. print "("
8. $y \leftarrow x$.left-child
9. while $y \neq NIL$ do
10. print(y)
11. print $x.operator$
12. $y \leftarrow y$.right-sibling
13. end while
14. print ")"
15. end if

Question 3 Idea: We will construct an undirected graph whose nodes are wrestlers, and there is an edge between u and v if and only if they are rivals. The idea of the algorithm is as follows. We start with an arbitrary node s, assign it "good", and run BFS from there in order to classify all nodes reachable from s. It is necessary that all neighbors of s are "bad", then all neighbors of these neighbors are "good", etc. So if we detect a rivalry between two "good" or two "bad" nodes, then no designation is possible and we report error in this case.

After doing BFS at s, if there are nodes in G unvisited we choose one of them and repeat the process.

(a) We will use the adjacency list representation for the graph. So associated with each node v is a linked list $Adj[v]$ that contains the neighbors of v. We will also use a field $label[v]$ for the designation of node v. These are either 0 (for "Good") or 1 (for "Bad"). To facilitate BFS, we have a color field $color[v]$ for each node v. $color[v]$ can have two values: White and Black. At the beginning all vertices are White. Once a vertex is visited in a BFS it's changed to Black.

(b1) Parsing the input. We construct the adjacency list for each vertex. Simply go through the array R, for each pair (u, v) in the array add v to $Adj[u]$ and add u to $Adj[v]$.

1. for i from 1 to length(R) do
2. $\quad (u, v) \leftarrow R[i]$
3. $\quad$ insert v to $Adj[u]$
4. $\quad$ insert u to $Adj[v]$
5. end for

(b2) We do BFS starting at node 1, compute the distance from 1 to other vertices reachable from it. If during the search we see an edge between two vertices of the same distance to v then we know that no designation is possible. After BFS at 1, if there are White vertices left then we choose the one White vertex and do the same, and so on. At the end, the designation for a node v is determined by the parity of $d[v]$.

1. % initialization: start with White vertices
2. $n \leftarrow length(W)$
3. for i from 1 to n do
4. $\quad color[i] \leftarrow White$
5. end for
6. % main for-loop: do BFS while there are White vertices left
7. for i from 1 to n do
8. $\quad$ if $color[i] = White$ do
9. $\quad\quad label[i] \leftarrow 0$
10. $\quad\quad$ % now do BFS at i
11. $\quad\quad$ Queue $Q \leftarrow \varnothing$
12. $\quad\quad color[i] \leftarrow Black$ % mark that i has been visited
13. $\quad\quad$ Enqueue(Q, i)
14. $\quad\quad$ while Q is not empty do
15. $\quad\quad\quad v \leftarrow Dequeue(Q)$ % take an element from the queue

16. for each u in $Adj[v]$ do
17. if $color[u] = White$ do
18. $label[u] \leftarrow 1 - label[v]$ % give u different label from v
19. $color[u] \leftarrow Black$
20. Enqueue(Q, u)
21. else do % if $color[u] = Black$
22. if $label[u] = label[v]$ % two neighbors with the same label
23. print "NO DESIGNATION POSSIBLE"; return
24. end if
25. end if
26. end for
27. end while
28. end if
29. end for

(b3) Now we print out the labels next to the names of the wrestlers.

1. for i from 1 to n do
2. print $W[i]$
3. if $label[i] = 0$ print "Good"
4. else print "Bad"
5. end for

(c) The procedure in (b1) has $length(R) = |E|$ iterations. Each iteration take constant time, so in total it takes $\mathcal{O}(|E|)$ time.

For the procedure in (b2): the initialization requires time $\mathcal{O}(|V|)$. The main part (lines 7 to 29) is complicated (as DFS), but the analysis can be done in similar, simple way. Instead of looking at the loops, we look at the amount of work done for each node and each edge in the graph. Here each node v is checked colored Black once, enqueued and dequeued once, gets a label once, and its color is checked $deg(v) + 1$ times (one on line 8, the other on line 17). Each edge of the graph is crossed at most once (comparing the labels of its endpoints, line 22). So in total this part takes time $\mathcal{O}(|V| + |E|)$.

The procedure in (b3) requires time $\mathcal{O}(|V|)$.

As a result, the total running time is $\mathcal{O}(|V| + |E|)$.

(d) There are two parts here. First we show that if the algorithm prints the error message "NO DESIGNATION POSSIBLE", then indeed no designation is possible. Because every two BFS trees

are not connected, the designation for each tree is independent of each other. So without loss of generality, suppose that the message is printed during BFS starting at vertex 1. We can assume that 1 is designated "Good" (by reversing the labels of all wrestlers if necessary). So all neighbors of 1 can only be designated "Bad", and so on. In other words, the designation of each node visited in the BFS starting at 1 is uniquely determined by the designation of 1. Therefore if there is an edge between two nodes with the same designation, no designation is possible.

Now we show that if no error message is printed, then the output is correct. This is because right after designating a node we color it Black, and for each edge with both Black endpoints its endpoints do not have the same designation. For an edge (u, v):

- if (u, v) is an edge in the BFS tree, then they are assigned different labels (line 18)
- otherwise, their labels are compared on line 22.

Question 4 Idea: The problem is to find for each vertex v the vertex with smallest label that is reachable from v. Put it another way, suppose that we reverse the edges in the graph and obtain a new graph G'. Then for each vertex v we want to find a vertex s with the smallest label such that there is a path (in G') from s to v. This suggests that we sort the vertex of G' in non-decreasing order of their labels, then perform DFS in this order. The value of a vertex v is the label of vertex s if v is visited during a call to DFS(s). So as we do the DFS, we will propagate the label of the root of the DFS tree and make it the value of all nodes in the tree.

Reversing the edges in the graph takes time $\mathcal{O}(|V| + |E|)$. The DFS algorithm takes time $\mathcal{O}(|V| + |E|)$. Because the labels are integer between 1 and $5|V|$, the sorting mentioned above can be done using counting sort, which takes time $\mathcal{O}(|V|)$. So in total the running time is $\mathcal{O}(|V| + |E|)$.

Data structure: Each node u has a color $color[u]$. Here for simplicity we use only two colors: White and Gray (as pointed out in class, Black is not necessary). We use an array $value$, $value[u]$ will be the value of u.

The algorithm:

1. Sort V in non-decreasing order of the labels using counting sort, store result in array A
2. % next, reverse the edges of G: $B[v]$ is the adjacency list of v in the new graph G'
3. for v in V do
4. for u in $Adj[v]$ do
5. insert v into $B[u]$
6. end for
7. end for
8. % now use DFS using the ordering in A, on G'
9. % initialization
10. for each u in A do
11. $color[u] \leftarrow White$

12. end for
13. % now the main loop
14. for each u in A do
15. if $color[u] = White$ do
16. $Visit(B, u)$
17. end if
18. end for

The code for Visit: we perform the DFS visit, propagating the value of the root of the tree to all nodes in the DFS tree.

Visit(B, u):

1. stack $S \leftarrow \varnothing$ % initialize S to the empty stack
2. $push(S, u)$
3. while S is not empty do
4. $x \leftarrow pop(S)$
5. $value[x] \leftarrow L[u]$ % propagate the label of u
6. if $color[x] = White$ do
7. $color[x] \leftarrow Gray$
8. $push(S, x)$
9. for each v in $B[x]$ do
10. if $color[v] = White$ do
11. $push(S, v)$
12. end if
13. end for
14. end if
15. end while

Running time: The procedure for reversing the graph visits each edge once, each vertex once, so it takes time $\mathcal{O}(|V|+|E|)$. Sorting using counting sort takes time $\mathcal{O}(|V|)$. Performing DFS (and propagating value) take time $\mathcal{O}(|V| + |E|)$.

Correctness: It suffices to show that for any vertex v: if s is the vertex with smallest label such that there is a path from s to v in G', then Visit(B, s) will be called (on line 16 of the main loop), and hence $value[v] = L[s]$ as required.

For any vertex u with $L[u] < L[s]$, there is no path from u to v. As a result, there is also no path from u to s. (Otherwise concatenate a path from u to s and a path from s to v we get a path from u to v.) Thus the DFS tree rooted at u cannot contain s nor v. As a result, v is visited during a call to Visit(B, s). QED.