CS250 Homework #5

David A. Mix Barrington, Ghazaleh Parvini

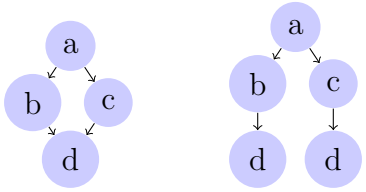
November 26, 2023

1: P9.5.2 [12 pts]

Suppose that a directed acyclic graph has maximum path length d and that no node has more than b neighbors. What is the largest number of node visits that could occur in a depth-first search of this graph? Show an example of such a graph that has only $bd + 1$ nodes and has the maximum number of node visits.

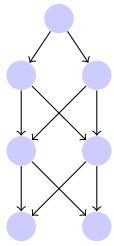
Solution:

For some given directed acyclic graph, we will define a traversal tree that represents the node visits during a depth first search. For some node r in the graph, we will define the traversal tree of r to have r as its root and the traversal trees of the neighbors of r as its subtrees. Note that a traversal tree of a directed acyclic graph can contain duplicate subtrees; this represents a node in the graph being visited multiple times.



A depth-first traversal of the graph starting at a certain node will be the same as a depth-first traversal of the traversal tree rooted at that node. However, the depth-first traversal of the traversal tree will visit each node in that tree exactly once, so we need only count the number of nodes in the traversal tree.
From now on we will be considering the traversal tree rooted at the beginning of the longest path through the original graph. The tree of depth d and maximum out-degree b with the most nodes is the full b-ary tree of depth d. From a similar argument as in P9.1.1, we know that a full b-ary tree of depth d has $b^{d+1} - 1$ nodes. Therefore, this is the greatest number of node visit for the original directed acyclic graph.

We will now construct a graph with only $bd+1$ nodes which achieves this maximum number of nodes visits. The graph will consist of the root node, followed by d layers of b nodes. Every node in the i^{th} layer will have an arc to every node in the $(i+1)^{\text{st}}$ layer. For example, with $d=3$ and $b=2$, the graph



will have 15 node visits during a depth-first traversal, even though it only has 7 nodes.

2: P9.6.8 [12 pts]

Let G be an undirected graph with n nodes that contains two nodes s and t, such that the shortest path from s to t has more than $\frac{n}{2}$ edges. Prove that there exists a node u, different from s and t, such that every path from s to t passes through u.

Solution:

Conduct a BFS from node s, so that every node is assigned to a level in the BFS. Since node t is more than $\frac{n}{2}$ steps by the shortest path, it is in a level d greater than $\frac{n}{2}$. We thus have levels $\{0, 1, ..., d\}$, with $d > \frac{n}{2}$. If levels 1 through $d-1$ each have two or more nodes, we would have a total of at least $2(d-1)+2 = 2d$ nodes, which is a contradiction if $d > \frac{n}{2}$. So there must exist a level e in the range from 1 through $d-1$ with only one node – let this node be u. Any path from s to t must pass through u, since no edge can pass from a node in a level before e to a node in a level after e.

3: P9.8.8 [10 pts]

Here is a single-step distance matrix for a weighted directed graph (shown in Figure 9-16) that indicates driving times among six locations in Massachusetts during Friday rush hour. The locations are Amherst (Amh), Hadley (Had), North Amherst (NAm), North Hadley (NHa), Northampton (Ntn), and Sunderland (Sun). An entry of `--` in the table indicates that there is no edge. (You may ignore the numbers next to each town name in Figure 9-16 – these will be used in Problem 9.9.7.)

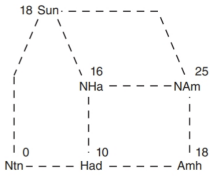


Figure 9-16: A directed graph representing some towns in Massachusetts.

	Amh	Had	NAm	HNa	Ntn	Sun
Amh	0	15	15	--	--	--
Had	12	0	--	8	30	--
NAm	19	--	0	10	--	14
NHa	--	10	9	0	--	12
Ntn	--	15	--	--	0	20
Sun	--	--	11	12	22	0

Use a uniform-cost search (with a closed list, recognizing previously seen nodes) to determine the fastest driving route and shortest-path distance from North Amherst to Northampton.

Solution:

Notation: (source, distance, destination)

Pop	**Put**	**Priority Queue**
∅	(NAm,0)	(NAm,0)
(NAm,0)	(NAm,19,Amh), (NAm,10,Nha), (NAm,14,Sun)	(NAm,10,Nha),(NAm,14,Sun), (NAm,19,Amh)
(NAm,10,Nha)	(Nha,20, Had),(Nha,22,Sun)	(NAm,14,Sun),(NAm,19,Amh), (Nha,20, Had),(Nha,22,Sun)
(NAm,14,Sun)	(Sun,36,Ntn)	(NAm,19,Amh), (Nha,20, Had),(Nha,22,Sun),(Sun,36,Ntn)
(NAm,19,Amh)	(Amh,34,Had)	(Nha,20, Had),(Nha,22,Sun), (Amh,34,Had),(Sun,36,Ntn)
(Nha,20, Had)	(Had,50,Ntn)	(Nha,22,Sun),(Amh,34,Had), (Sun,36,Ntn),(Had,50,Ntn)
(Nha,22,Sun)	(Sun,44,Ntn)	~~(Amh,34,Had)~~, (Sun,36,Ntn), (Sun,44,Ntn),(Had,50,Ntn)
(Sun,36,Ntn)	∅	∅

Shortest path: North Amherst → Sunderland → Northampton. It will take 36 min.

4: P9.8.9 [10 pts]

Our uniform-cost search is very similar to **Dijkstra's Algorithm**, the best-known method of solving the single-source shortest path problem. In one formulation, Dijkstra's Algorithm keeps an array D indexed by the nodes of the graph, so that $D(x)$ indicates the shortest *so far known* distance from s to x. Initially we set $D(s)$ to 0 and $D(x)$ to ∞ for all other nodes x. Another boolean array classifies each node as "explored" or "unexplored", with all nodes initially unexplored.
A step of the algorithm is as follows:

- Find the unexplored node x with the smallest value of $D(x)$.
- For every edge out of x, to node y with cost $c(x, y)$, compute $D(x) + c(x, y)$.
- If any of these values are smaller than the corresponding $D(y)$, reset $D(y)$ to the new value.
- Mark x as explored.

The algorithm ends if a goal node is marked explored, or if $D(x)$ for every unexplored x is ∞.

(a) Explain how this algorithm corresponds to a uniform-cost search of the same graph, in particular how each step of one corresponds to a step of the other.

(b) Uniform-cost search uses a priority queue. Explain how a priority queue can be used to improve the running time of Dijkstra's Algorithm.

Solution:

(a) The difference between the two is Dijkstra's Algorithm first assigning $D(s) = 0$ and $D(x) = \infty$ to the source and every other node, repeatedly accessing a node's neighbors and finding the node with the smallest $D(x)$, calculating $D(x) + c(x, y)$, and assigning the smallest distance to the node, after the assignment of the smallest weight, it is marked as explored. This will be repeated for all the nodes until the goal state is reached and assign the weight from the source node to that goal state.
However, UCS starts with the source nodes, it doesn't need to access all the neighbor nodes. Each time it will visit the node that has the smallest distance value $D(x)$, put node x into the closed list, add the neighbor nodes of node x to the priority queue, and compute $D(x) + c(x, y)$. This will be repeated until the goal state is reached.

(b) For Dijkstra's Algorithm, the computation for the minimum distance from a node s to s' requires checking all of its neighbor nodes which requires O(n) time. However, for the UCS algorithm, finding the minimum distance from node s to s' is implemented using a heap. RemoveMin() and add() operation only cost O(log(n)) time. This is

logarithmic faster than Dijkstra's Algorithm. In summary, using a heap prevents the algorithm from checking all the neighbors of a node and instead, it returns a minimum D(x) from the node that has been explored.

5: P9.9.7 [12 pts]

Conduct an A^* search of the weighted directed graph of Problem 9.8.8, with start node North Amherst and goal node Northampton. Use the heuristic function h defined by $h(\text{Ntn}) = 0$, $h(\text{Had}) = 10$, $h(\text{NHa}) = 16$, $h(\text{Amh}) = 18$, $h(\text{Sun}) = 18$, and $h(\text{NAm}) = 25$. (This heuristic represents the time needed to drive to Northampton with no traffic.) Figure 9-16 shows the heuristic value for each node, and the driving times are given in a table in Problem 9.8.8. Has the traffic in the original problem affected the optimal route from North Amherst to Northampton?

Solution:

Pop	**Put**	**Priority Queue**
$\emptyset$	(NAm,25)	(NAm,25)
(NAm,25)	(NAm,26,Nha),(NAm,32,Sun), (NAm,37,Amh)	(NAm,26,Nha),(NAm,32,Sun), (NAm,37,Amh)
(NAm,26,Nha)	(Nha,30,Had),(Nha,40,Sun)	(Nha,30,Had),(NAm,32,Sun), (NAm,37,Amh),(Nha,40,Sun)
(Nha,30,Had)	(Had,50,Amh),(Had,50,Ntn)	(NAm,32,Sun),(NAm,37,Amh), (Nha,40,Sun),(Had,50,Amh), (Had,50,Ntn)
(NAm,32,Sun)	(Sun,36,Ntn)	(Sun,36,Ntn),(NAm,37,Amh), (Nha,40,Sun),(Had,50,Amh), (Had,50,Ntn)
(Sun,36,Ntn)	$\emptyset$	$\emptyset$

Shortest path: North Amherst $\rightarrow$ Sunderland $\rightarrow$ Northampton. It will take 36 min.

The optimal route stays the same. For A* algorithm, it requires less iterations to reach the goal state.

6: P5.1.6 [12 pts]

Let A be any finite alphabet. The language $A^{\leq k}$ is defined to be the union $A^0 + A^1 + \ldots + A^k$, the set of all strings over A with length at most k. Prove by induction on all naturals k that the languages $(\emptyset^* + A)^k$ and $A^{\leq k}$ are equal.

Solution:

Let the predicate $P(k)$ be true if $A^{\leq k} = (\emptyset^* + A)^k$, otherwise is false.
Base Case: k = 0

$$\begin{aligned} A^{\leq 0} &= (\emptyset^* + A)^0 \\ A^0 &= \lambda \\ \lambda &= \lambda \end{aligned}$$

Inductive Hypothesis: $A^{\leq k} = (\emptyset^* + A)^k$
Inductive Step: $\forall k : P(k) \longrightarrow P(k+1)$

$$\begin{aligned} (\emptyset^* + A)^{k+1} &= (\emptyset^* + A)(\emptyset^* + A)^k \\ &= (\emptyset^* + A)(\emptyset^* + A)^k \\ &= (\emptyset^* + A)A^{\leq k} && \text{I.H.} \\ &= \emptyset^* A^{\leq k} + AA^{\leq k} \\ &= A^{\leq k} + A^{\leq k+1} \\ &= (A^0 + A^1 + A^2 + \cdots + A^k) + (A^0 + A^1 + A^2 + \cdots + A^{k+1}) \\ &= (A^0 + A^1 + A^2 + \cdots + A^{k+1}) \\ &= A^{\leq k+1} \end{aligned}$$

7: P5.2.4 [10 pts]

Construct a regular expression denoting the language of strings in $\{a, b\}^*$ that have a number of b's that is divisible by 3.

Solution:

A regular expression for the language of strings with the number of b's divisible by 3 is $(a^*ba^*ba^*ba^*)^* + a^*$.

Let's prove this in both directions, which means the number of b's of every string in the language of the regular expression is divisible by 3 and every string with a number of b's that's divisible by 3 is in the language of the regular expression.

The first part of the regular expression ensures that there will always be exactly 3 b's added, and note that if there are 0 b's, we must also allow for any number of a's.

Given any string with 3k b's, k is a positive integer, we can divide the string into substrings with either 0 or 3 b's in it, and every such string is in the expressions that we star in our expression.

8: P5.4.7 [10 pts]

Prove that for any two languages S and T, $(ST)^*S = S(TS)^*$. Use induction on the definition of the Kleene star languages.

Solution:

Let's define predicate $P(n)$: true if $(ST)^nS = S(TS)^n$. Otherwise, is false.
Base case: $n = 0$ for the concatnation of the Kleene star.

$$\begin{aligned}(ST)^0S &= S(TS)^0\\ \lambda S &= S\lambda\\ S &= S\end{aligned}$$

Inductive Hypothesis: Assume a arbitrary positive natural number k such that:

$$(ST)^kS = S(TS)^k$$

Inductive Step: $\forall k : P(k) \longrightarrow P(k+1)$

$$\begin{aligned}(ST)^{k+1}S &= (ST)(ST)^kS\\ &= (ST)S(TS)^k && \text{I.H.}\\ &= S(TS)(TS)^k && \text{Associativity}\\ &= S(TS)^{k+1}\end{aligned}$$

EC: P5.5.6 [10 pts]

If L is any language, we define its **substring language** $Sub(L)$ to be the set of all strings y such that y is a substring of any string $x \in L$. Prove that if S is any regular expression, then $Sub(L(S))$ is a regular language. Give a recursive algorithm to produce a regular expression for this language.

Solution:

We can start out to do a standard induction proof on all regular expressions:

$$\begin{aligned} \text{Sub}(\emptyset) &= \emptyset \\ \text{Sub}(a) &= a + \lambda \\ \text{Sub}(R + S) &= \text{Sub}(R) + \text{Sub}(S) \end{aligned}$$

$\text{Sub}(\emptyset) = \emptyset$ is simple, but the other two induction steps are much more complicated. If R and S are languages, there are three ways a string y to be a substring of a string in $L(RS)$:

- y could be in Sub(R), if $L(S)$ is not the empty language
- y could be in Sub(S) if $L(R)$ is not the empty language y could be in Suff(R)Pref(S)
- y could be in Suff(R)Pref(S)

where Suff(R) is the set of all suffixes of $L(R)$, and Pref(S) is the set of all prefixes of $L(S)$. This means that to complete our proof and algorithm for Sub, we also need to solve the similar problem for Pref and Suff.

Fortunately, E5.5.3 does this for Pref, and the proof for Suff is similar. So all we need to do to complete the Sub proof is to finish the star case, by getting an expression for Sub(R^*). For a string y to be a substring of $L(R^*)$, it could be:

- y could be in Sub(R)
- y could be in Suff(R)Pref(R)
- y could be in Suff(R)R^*Pref(R)

Code:

```
public RegExp substring (RegExp r){
    if (r.isEmptySet()){
        return new RegExp();
    }
    if (r.isZero()){
        return plus(star(new RegExp()), new RegExp('0'));
    }
    if (r.isOne()){
        return plus(star(new RegExp()), new RegExp('1'));
    }

    RegExp s = r.firstArg();
    if (r.isStar()) {
        if (!emptyLanguage(s)){
            return plus(substring(s), cat(suffix(s), cat(star(s), \
            prefix(s))));
        }else{
            return star(new RegExp());
        }
    }

    RegExp t = r.secondArg();
    if (r.isUnion()) {
        return plus(substring(s), substring(t));
    }

    if (emptyLanguage(s) || emptyLanguage(t)){
        return new RegExp();
    }else{
        return plus(plus(substring(s), substring(t)), \
        cat(suffix(s),prefix(t)))
    }
}
```