Study probs

­­­­­­1) Adjacency List - For sparse graphs, has space and time advantage, dynamic size.

Adjacency Matrix - For dense graphs, has space advantage but no time advantage or disadvantage, always takes up the same amount of space (n^2) even if little of it is used

2) IN DEGREE 🡪 List- O(n^2) Matrix – O(n)

OUT DEGREE 🡪 List – O(n) Matrix – O(n)

3) Time complexity: L 🡪 O(|V| + |E|log|V|) M🡪O(V^2)

For sparse graphs (|E| = C|V|), list is better

For dense graphs (|E| = C|V|^2), matrix is better.

4) S A B C

AFTER FIRST: DIST: 0, 7, 8, 2 QUEUE: CAB FINAL: S

AFTER SECOND: DIST: 0, 6, 4, 2 QUEUE: BA FINAL: SC

AFTER THIRD: DIST: 0, 5, 4, 2 QUEUE: A FINAL: SCA

AFTER FOURTH: DIST: 0, 5, 4, 2 QUEUE: FINAL: SCAB

5)DFS: O(V+E)

ADD START NODE TO STACK, THEN TAKE TOP NODE FROM STACK, VISIT FIRST AVAILABLE NEIGHBOR. FOR ANY NEIGHBOR OF TOP NODE, IF IT IS NOT VISITED, ADD TO TOP OF STACK AND REPEAT (since you want to go deeper). IF THE NODE ON TOP OF STACK HAS NO NEIGHBORS, POP THE TOP NODE AND REPEAT UNTIL NO MORE NODES IN STACK.

1. STACK: S VISITED: 2) STACK: SA VISITED: S 3) STACK: SB VISITED: SA

STACK: SC VISITED: SAB STACK: S VIS: SABC STACK: VIS: SABC

BFS: O(V+E)

ADD START NODE TO QUEUE, TAKE VERTEX FROM FRONT OF Q AND EXPLORE IT, ADDING NEIGHBORS TO QUEUE BEFORE GOING ON. THEN REPEAT, DEQUEUING THE NEXT VERTEX AND EXPLORING TO FIND ALL NEIGHBORS, IF NO NEIGHBORS FOUND, THEN MOVE ON TO NEXT VERTEX IN QUEUE. REPEAT UNTIL QUEUE IS EMPTY.

1. QUEUE: S VISITED: 2) QUEUE: A VISITED: S 3) QUEUE: AB VISITED: S 4) QUEUE: ABC VISITED: S 5) QUEUE: BC VISITED: SA 6) QUEUE: C VISITED: SAB 7) QUEUE: VISITED: SABC

6) MIN SPANNING TREE IS A SET OF EDGES that connects every vertex in the graph with the lowest total sum of weights without cycles

KRUSKAL: (O(nlogn)) USE MINHEAP TO STORE EDGES, EXTRACT MIN TO GET THE MINIMUM EDGE IN THE GRAPH. IF ADDING THE EXTRACTED EDGE WOULD RESULT IN A CYCLE, SKIP THAT EDGE. ADD THESE SMALLEST EDGES TO THE GROWING MST UNTIL MINHEAP IS EMPTY.

G = {(s, b), (f, e), (b, c), (s, f), (b, d)}

Cost = 1+2+3+6+11 = 23

PRIM: USE MINHEAP TO KEEP TRACK OF CONNECTED VERTICES. SET ALL PRIORITIES TO INFINITY EXCEPT FOR SOURCE, WHICH = 0. ALL NODES ARE WHITE TO START. EXTRACT THE NODE WITH THE MINIMUM DISTANCE FROM THE CONNECTED COMPONENTS OF THE MST AND MAKE IT BLACK, ADD THE EDGE WITH THIS MINIMUM NODE AND THE TARGET OF THE CONNECTED EDGE TO THE MST. FOR EACH EDGE THAT USES THE EXTRACTED MIN AS THE SOURCE NODE, IF THE NEIGHBOR IS BLACK, SKIP IT. IF THE WEIGHT OF THE EDGE IS ABOVE THE TARGET’S PRIORITY IN THE MINHEAP, SKIP IT AS WELL. ELSE UPDATE THE PRIORITY OF THE TARGET NODE TO REFLECT THE UPDATED SHORTEST DISTANCE IN THE MST. REPEAT UNTIL MINHEAP IS EMPTY.

1. BLACK: S MST: {} 2) BLACK: SB MST: {(S, B) }

3) BLACK: SBC MST: {(S, B), (B, C)}

4) BLACK: SBCF MST: {(S, B), (B, C), (S, F)}

5) BLACK: SBCFE MST: {(S, B), (B, C), (S, F), (F, E)}

6) BLACK: SBCFED MST: {(S,B), (B,C), (S,F), (F,E), (B,D)}

COST = 1+3+6+2+11 = 23

7) FIND ALL-PAIRS SHORTEST PATHS.  It gradually allows more and more vertices to be included as intermediate points in paths, updating shortest path so far for each vertex as needed.

Create a 2D VxV distance matrix dist (where V is the # of vertices). Set all initial values in it to infinity. Set all diagonal values to 0 (a vertices shortest path to itself is 0). For each existing edge, add its weight to the corresponding place in adjacency matrix

|  |  |  |  |
| --- | --- | --- | --- |
| D(1) | 1 | 2 | 3 |
| 1 | 0 | 3 | -1 |
| 2 | 3 | 0 | -1 |
| 3 | 3 | 3 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| D(2) | 1 | 2 | 3 |
| 1 | 0 | 3 | -1 |
| 2 | 3 | 0 | -1 |
| 3 | 3 | 3 | 0 |

Add an intermediate vertex v. Create an auxiliary distance matrix dist\_a. For each spot in dist\_a that isn’t in the row and/or column of v, compare dist

|  |  |  |  |
| --- | --- | --- | --- |
| D(0) | 1 | 2 | 3 |
| 1 | 0 | 3 | -1 |
| 2 | 3 | 0 | -1 |
| 3 | 3 | 3 | 0 |

|  |  |  |  |
| --- | --- | --- | --- |
| D(3) | 1 | 2 | 3 |
| 1 | 0 | 2 | -1 |
| 2 | 2 | 0 | -1 |
| 3 | 3 | 3 | 0 |

8) MST. USE KRUSKAL WITH A MINHEAP IMPLEMENTATION 🡪 O(ElogV)

WITHOUT MINHEAP, K HAS O(V^2)

PRIM HAS O(V + (V+E)LOGV) FOR ALIST, O(V^2) FOR MATRIX

9) CREATE RESIDUAL GRAPH, FIND AUGMENTING PATH; IF CANNOT FIND ANY PATHS, STOP. MAXIMUM FLOW HAS BEEN FOUND. ADD BOTTLENECK CAPACITY TO TOTAL FLOW. MAX FLOW IS THE SUM OF BOTTLENECKS OF ALL AUGMENTING PATHS. THE EDGES IN THE PATHS CAN BE FORWARD OR BACKWARD, IF A BACKWARD EDGE IS TAKEN, SUBTRACT FLOW FROM THAT EDGE, AND IF FORWARD EDGE IS TAKEN, ADD FLOW TO THAT EDGE. EDGE CANNOT BE TAKEN IF CAPACITY IS REACHED. REPEAT UNTIL NO AUGMENTING PATHS CAN BE FOUND.

s 🡪 a🡪d🡪t flow of 1

s🡪b🡪c🡪e🡪f🡪t flow of 1

s🡪a🡪e🡪f🡪t flow of 1

s🡪b🡪c🡪d (residual edge) 🡪a🡪e🡪f🡪t flow of 1

total of 4

10) 10

2 20

1 7 13

8

INSERT 4.

10

2 20

1 7 13

4 8

DELETE 2.

10

4 20

1 7 13

8

DELETE 10.

13

4 20

1 7

8

11) BST HAS MIN, MAX, PRED, SUCC WHILE HASH DOES NOT

TASK A: USE HASH TABLE DUE TO CONSTANT LOOKUP TIME

TASK B: BST DUE TO EFFICIENT LOOKUP TIME && FIND NEIGHBORS (BY BIRTHDAY)

12) FOR LARGE SETS, B-TREES ARE MORE EFFICIENT DUE TO THEIR USAGE OF CONTIGUOUS STORAGE. THEY GROUP ELEMENTS CLOSELY IN MEMORY, AND THE MEMORY USAGE IS SUPERIOR TO BST FOR LARGE SETS.

13) A: 10 IS BIGGER THAN 5, MOVE RIGHT. 10 IS LESS THAN 13, MOVE LEFT. FOUND A LEAF CELL THAT IS NOT FULL, INSERT 10 INTO THE CELL WITH 6 AND 8.

B: 10 IS LESS THAN 11, MOVE LEFT. CELL IS FULL, SO WE SPLIT IT, MAKING 4 THE ROOT AND 2 AND 6 THE LEFT AND RIGHT CHILDREN, RESPECTIVELY. 10 IS GREATER THAN 4 AND 6, SO MOVE RIGHT TWICE. WE HIT ANOTHER FULL CELL. SPLIT THE LEAF CELL, WITH 8 BEING MOVED INTO THE 6 CELL AND 7 AND 9 BEING CHILDREN NODES. 10 IS GREATER THAN 8, SO MOVE RIGHT, AND NOW IT CAN BE INSERTED INTO THE NODE WITH 9.

14)

A) DEPENDS ON DEFINITION OF CONNECTED

B) FALSE, IF THE SHORTEST PATH CONSISTS OF 3 EDGES, AND ANOTHER PATH CONSISTS OF A SINGLE DIRECT EDGE, THE PROPOSED INCREASE ADDS 3 TO THE FIRST PATH AND 1 TO THE SECOND PATH, POTENTIALLY CHANGING WHICH PATH IS ACTUALLY SHORTEST.

C) FALSE, NOT O(VLOGE), ITS O(ELOGV)

D) TRUE

E) TRUE ROOT CAN BE EXCEPTION, AND CHILDREN FOLLOW T-1<N<2T-1

F) TECHNICALLY FALSE, SINCE FW IS O(V^3), SO IT SHOULD NOT BE <=. \*ORDER IS O(E)<O(V+ELOGV) OR O(V^2) < O(V^3)

G) TRUE FOR PRE AND POST ORDER, BUT NOT IN-ORDER.