Name : Pathri Vidya Praveen
Roll. No. : CS24BTECH11047
Coding + Theory Assignment 4

2. Theory Questions :

1. ⇒ Input : A weighted, directed graph $G = (V, E)$ with edge weights $w : E \rightarrow R$

⇒ Output : A list of vertices forming a negative weight cycle or "No negative cycle" if none exists.

Pseudocode :

⇒ Initialize : For each vertex $v \in V$:
$d[v] \leftarrow 0$
$\pi[v] \leftarrow$ NULL

⇒ For $i = 1$ to $|V|$:
For each edge $(u, v) \in E$:
if $d[u] + w(u, v) < d[v]$:
$d[v] \leftarrow d[u] + w(u, v)$
$\pi[v] \leftarrow u$

⇒ For each edge $(u, v) \in E$:
if $d[u] + w(u, v) < d[v]$:
let $x \leftarrow v$
For $i = 1$ to $|V|$:
$x \leftarrow \pi[x]$
let cycle $\leftarrow [\,]$
let $y \leftarrow x$
Repeat :
cycle.append(y)
$y = \pi[y]$
until $y = x$
Reverse cycle; return cycle

⇒ Return "No negative cycle"

Theorem statement and complexity: Given a weighted directed graph $G_1 = (V, E)$ that contains atleast 1 negative-weight cycle, the algorithm returns a list of vertices that forms a negative weight cycle. Algorithm runs in $O(|V||E|)$ time and $O(|V|)$ space.

Formal proof:

① Detection correctness: Algorithm initializes $d[v] = 0, \forall v \in V$. which is same as adding source vertex s with 0-weight edges to all vertices in V. this ensures any negative cycle reachable from s. After $|V|$ iterations of relaxation, if (u, v) satisfies $d[u] + w(u, v) < d[v]$, negative cycle exists. For contradiction, assume no negative cycle exists. Then all shortest paths from s have atmost $|V|$ edges so after $|V|$ iterations, no relaxation occurs.

② Cycle extraction correctness: let (u, v) be relaxable edge after $|V|$ iterations. Starting from v, we follow predecessor pointers $|V|$ times to obtain vertex x. Since there are $|V|$ vertices, by pigeonhole principle, this sequence must enter a cycle in the predecessor subgraph $G_\pi = (V, \{(\pi[v], v) : \pi[v] \neq NULL\})$

let $C = (v_0, \ldots, v_{k-1})$ be the cycle where $v_0 = x$,

$\pi[v_{(i+1)\%k}] = v_i \quad \forall i$.

③ Negative weight proof:

We prove $w(C) < 0$ by contradiction. By relaxation property, for each edge $(v_i, v_{(i+1) \bmod k})$ in C:

$$d[v_{(i+1)\%k}] \leq d[v_i] + w(v_i, v_{(i+1)\%k})$$

Assume $w(C) \geq 0$.

$$\sum_{i=0}^{k-1} d[v_{(i+1)\%k}] \leq \sum_{i=0}^{k-1} d[v_i] + w(C)$$

⇒ left, right sums of d are equal, so $W(C) \geq 0$.

If $W(C) \geq 0$, all inequalities will become

$$d[V_{(i+1)\%k}] = d[V_i] + W(V_i, V_{(i+1)\%k}) \quad \forall i$$

This implies d-values are stable on C. Consider path from v to x in G_π (traced by first |V| predecessor steps). For each edge (a,b) in P,,

$$d[b] \leq d[a] + W(a,b)$$

If $W(C) \geq 0$, then by induction, d-values along P must also be stable. Thus d-values of all vertices on P, C are stable after |V| iterations and this leads to contradiction since the relaxable edge (u,v) was found because $d[u] + W(u,v) < d[v]$ → instability. So, $W(C) < 0$.

Complexity:

⇒ Initialization : $O(|V|)$

⇒ |V| relaxation passes : $O(|V| \cdot |E|)$

⇒ Cycle detection and extraction : $O(|V|)$

⇒ Total time : $O(|V||E|)$

Space : $O(|V|)$ for d, π arrays.

Algorithm correctly outputs negative weight cycle when one exists with stated time, space complexity.

2. Algorithm Pseudocode:

⇒ Input : ① G is original graph (V, E)
② T is original MST (set of edges)
③ e : edge (u, v) not in T whose weight is decreased to w_new

⇒ Output : Updated MST T' for modified graph

⇒ ① Find unique path P in T from u to v using BFS/DFS
② let f be edge in P with maximum weight
③ if w_new < weight(f) :
$T' = (T \setminus \{f\}) \cup \{e\}$
else : $T' = T$
④ Return T'.

Theorem statement:

Given a graph $G = (V, E)$ with edge weights and an MST T of G, if the weight of an edge $e = (u, v) \notin T$ is decreased to w_new, then algorithm correctly computes MST for modified graph.

Complexity:

⇒ $O(|V|)$ time - dominated by finding path in tree
⇒ $O(|V|)$ space for storing path during BFS/DFS

Proof: Let G' be modified graph where $w(e) = w\_new$.

Case 1 : w_new ≥ weight(f):

We prove T remains MST for G' using cycle property.
A spanning tree is minimum iff for every edge not in tree, it is heaviest edge in the cycle it forms with tree.

⇒ For any edge $g \neq e$ not in T : Fundamental cycle of g with T is unchanged from G to G' and since T was MST for G, g was heaviest edge in its cy

⇒ For edge e : Cycle is $C = P \cup \{e\}$. Since W_new ≥ weight(f), f was maximum weight edge in P, e is heaviest edge in C. By cycle property, T is MST for G'.

Case 2: W_new < weight(f):-

⇒ $T' = (T \setminus \{f\}) \cup \{e\}$

⇒ Removing f from T disconnects T into 2 components Since P was the u-v path in T and $f \in P$, u, v are in different components after removing f. Adding e reconnects components. $|T'| = |T| = |V| - 1$ and T' is connected so T' is tree

⇒ Suppose some MST M of G' does not contain e. Then M is spanning tree of G (original weights) with weights w(M). $w(M) \geq w(T)$ because T was MST of G.

$w(T') = w(T) - \text{weight}(f) + W\_new < w(T)$

$w(M) \geq w(T) > w(T')$

↳ contradicts M being MST of G'.

- so, all MST of G' must contain e

⇒ Any spanning tree containing e must be of the form $(T \setminus \{x\}) \cup \{e\}$ for some x on the u-v path P. The weight is $w(T) - \text{weight}(x) + W\_new$. This is minimized when weight(x) is maximized which is exactly f.

⇒ T' is MST for G'.

3. Algorithm Pseudocode: Prim-MST (G, w)

⇒ Input: Connected graph G = (V, E) with edge weights w

⇒ Output: A MST T of G

⇒ Initialize: T = ϕ

Let r be arbitrary vertex in G.

key[r] = 0

∀ v ∈ V \ {r}: key[v] = ∞

Q ← V (priority queue using key values)

⇒ While Q ≠ ϕ:

① u = extract-min(Q)

② For each vertex v adjacent to u:

If v ∈ Q, w(u,v) < key[v]:

key[v] = w(u,v)

π[v] = u

③ If u ≠ r, add edge (π[u], u) to T

⇒ Return T

Theorem statement: If for every cut of graph, there is unique light edge crossing the cut then Prim's algorithm outputs unique MST of graph.

Complexity: Prim's algo runs in O(E log V) time with binary heap, O(V+E) space to store graph and auxiliary data structures.

Formal proof: We prove by induction that at every step of Prim's algorithm, set T is subset of every MST of G.

⇒ Base case: Initially T = ϕ is trivially subset of every MST

⇒ Inductive step: Assume after k steps, T is subset of every MST. Consider (k+1)th step.

⇒ Prim's algo maintains cut $(S, V \setminus S)$ where S is set of vertices already in tree. Algo selects min. weight edge e crossing this cut. By hypothesis, there is unique light edge crossing this cut, so e must be this unique light edge.

⇒ Assume, ∃ MST T' that does not contain e. Since T' is spanning tree, it must contain atleast 1 edge f crossing the cut $(S, V \setminus S)$. By hypothesis, e is unique light edge so $w(e) < w(f)$. Now consider tree $T'' = T' \setminus \{f\} \cup \{e\}$. $w(T'') = w(T') - w(f) + w(e)$ $< w(T')$ this contradicts T' is minimal. So, e must be in every MST. $T \cup \{e\}$ is subset of every MST.

⇒ By induction, final T is subset of every MST. Since T is spanning tree, it is MST. Since it is contained in every MST, it is unique MST.

Counterexample to converse:

⇒ G → vertices = $\{1, 2, 3\}$ and edges are

① (1,2) → weight 1 ② (1,3) → weight 1 ③ (2,3) → weight 2

⇒ This graph has unique MST $T = \{(1,2), (1,3)\}$. However the cut $(\{1\}, \{2,3\})$ has 2 light edges: (1,2), (1,3) both of weight 1. Thus the condition "for every cut, there is a unique light edge" is not satisfied, showing converse is false.

4. Algorithm Pseudocode: Update-MST(T, v, E•v)

⇒ Input: ① T is MST of original graph G (set of edges) ② v = new vertex to be added

③ E•v = set of edges incident to v (connecting v to vertices in G)

⇒ Output : T' → updated MST of $G \cup \{v\}$

⇒ ① Construct graph H with : Vertices $V \cup \{v\}$ where V are vertices from G, Edges : $T \cup E_v$

② Run Kruskal's algorithm on H to compute MST

③ Return resulting MST as T'.

Theorem: MST of full updated graph $G' = (V \cup \{v\}, E \cup E_v\}$ is identical to MST of subgraph $H = (V \cup \{v\}, T \cup E_v)$.

Complexity:

⇒ $n = |V|$ = no. of vertices in G

$k = |E_v|$ = no. of new edges incident to v

⇒ Graph H has $|V_H| = n+1$ vertices;

$|E_H| = |T| + |E_v| = n-1+k$ edges

⇒ Kruskal's algo on H takes $O(|E_H| \log |E_H|)$

$= O((n+k) \log (n+k))$.

⇒ Since $k \leq n$, this simplifies to $O(n \log n)$

⇒ Space complexity $= O(|V_H| + |E_H|) = O(n+k) = O(n)$

Formal proof: Prove that $MST(G') = MST(H)$

⇒ $G' = (V \cup \{v\}, E \cup E_v)$ is full updated graph

⇒ $H = (V \cup \{v\}, T \cup E_v)$ is constructed subgraph

Part 1: Weight (MST(G')) ≤ Weight (MST(H))

⇒ Since H is subgraph of G', any spanning tree of H is also spanning tree of G'. So, MST(G') must have ~~less w~~ weight ≤ weight of any spanning tree of H (MST(H)).

Part 2: Weight (MST(G')) ≥ Weight (MST(H))

⇒ Let T' be any MST(G'). Show that we can transform T' into spanning tree that uses only edges from H without increasing the weight.

⇒ Assume T' contains edge $e = (u, w)$ not in H. So, $e \in E \setminus T$. Consider adding e to original MST T. This creates unique cycle C in $T \cup \{e\}$. By cycle property of MSTs. —— Every edge f on cycle C $(f \neq e)$ must satisfy weight $(f) \leq$ weight (e). All such edges f are in T and therefore are in H.

⇒ Remove e from T'. This disconnects T' into 2 components creating a cut. Cycle C must cross this cut atleast twice. — once at e and atleast once more. Let f be edge from C $(f \in T)$ that also crosses this cut.

⇒ Since T' is MST, e must be min. weight edge crossing this cut so weight $(e) \leq$ weight (f).
By cycle prop., weight $(f) \leq$ weight (e)

⇒ weight (f) = weight (e).

⇒ We can create new spanning tree $T'' = (T' \setminus \{e\}) \cup \{f\}$

⇒ This tree has same weight as T' (weight of $f = e$), uses one more edge from H since $f \in T \subseteq H$.

⇒ We can repeat this process $\forall$ edge in T' not in H, eventually obtaining MST T'_{final} of G' that uses only edges from H.

⇒ Since T_{final} is spanning tree of H, MST(H),

Weight(MST(H)) $\leq$ Weight (T'_{final}) = Weight (MST(G'))

⇒ By Parts ① and ②

Weight (MST(G')) $\leq$ Weight (MST(H))

Weight (MST(G')) $\geq$ Weight (MST(H))

⇒ Weight (MST(G')) = Weight (MST(H))

Algorithm correctly computes MST(G').

5. Example of a directed graph with negative weight edges where Dijkstra's algorithm fails:

⇒ Graph with vertices S, A, B and edges S → A with weight 3, S → B with weight 2, A → B with weight −5. Shortest path from S to B:

S → A → B has 3 − 5 = −2 (correct shortest path)

⇒ By Dijkstra's algorithm:

① Initialize distance[S] = 0, distance[A] = distance[B] = ∞

② Process S: Update neighbors dist[A] = 3, distance[B] = 2

③ Extract min. from queue: B (distance 2) → here B has incorrect distance 2 and is finalized.

④ Process B: No outgoing edges

⑤ Extract min. from queue: A (distance 3)

⑥ Process A: discover path to B with 3 − 5 = −2 weight. As B is finalized already, no update

⑦ Finally distance[B] = 2 (which should be −2)

Dijkstra's algorithm pseudocode: Dijkstra(G, S)

```
distance[S] = 0
prev[S] = ∞
for each vertex v in G:
    if v ≠ source:
        distance[v] = ∞
        prev[v] = ∞
    add v to priority queue Q with key distance[v]
while Q is not empty:
    u = extract-min(Q)
    for each neighbor v of u:
        alt = distance[u] + weight(u, v)
        if alt < distance[v]:
            distance[v] = alt ; prev[v] = u
            decrease key(Q, v, alt)
return distance, prev
```

Theorem: If $G = (V, E)$ is a directed graph with non negative weight function $w: E \rightarrow R_+$ and source vertex s, Dijkstra's algorithm computes shortest path from s to every vertex $v \in V$.

Complexity:

⇒ Time: $O((|V|+|E|)\log|V|)$ using binary heap

⇒ Space: $O(|V|)$

Formal proof:

⇒ Let S be set of vertices whose shortest paths from s have been determined. We prove that whenever a vertex u is added to S, distance[u] is shortest path distance from s to u.

⇒ Base case: $S = \{s\}$, distance[s] = 0 is true

⇒ Inductive step: Assume all vertices in S have correct distances. Let u be the next vertex added to S (minimal distance[u] in Q). Assume, by contradiction, ∃ shorter path P from s to u.

⇒ Let y be 1st vertex on P not in S, x be its predecessor in S.

⇒ ① distance[y] ≤ distance[x] + w(x, y)

② By greedy choice: distance[u] ≤ distance[y] as u was chosen before y

③ Critical step which makes failure for negative weights is that length(P) ≥ distance[y] holds because the subpath from y to u has non negative weight, so length(P)

= (s → y) + (y → u) ≥ (s → y) ~~+ dist~~

≥ distance[y]

⇒ Contradiction:

distance[u] ≤ distance[y] ≤ length(P) < distance[u]

so, distance[u] < distance[u] → contradiction

⇒ Proof fails with negative weights because assumption that length(P) ≥ distance[y] fails because:

① Subpath y to u may have negative weight

② ∴, length(P) = (s → y) + (y → u) < (s → y)

③ length(P) ≥ distance[y] doesnot hold

④ Contradiction argument does not valid.

⇒ With -ve weights, a subpath of shortest path is not necessarily shortest path itself. Dijkstra algo's greedy algo of expanding minimum distance vertex because path from higher distance vertex may result in lower cost due to negative edges later in path.

---

6. Algorithm pseudocode: Dijkstra_modified(G, w, s)

```
V = G.V
E = G.E
max_dist = (V-1) * w
distance[0..V-1] = ∞
distance[s] = 0.
Create array B[0..max_dist] of empty lists.
B[0].append(s)
min = 0
while min ≤ max_dist
    while B[min] is not empty
        u = B[min].removeFront()
        if dist[u] < min then continue
        for each vertex v in G.adj[u]
            new_d = distance[u] + w(u,v)
            if new_d < distance[v]
                distance[v] = new_d
                if new_d ≤ max_dist
                    B[new_d].append(v)
    min++
return distance
```

Theorem: Given a weighted directed graph $G = (V, E)$ with non-ve edge weights from $\{0, 1, \ldots, W\}$, modified Dijkstra's algo correctly computes shortest path distances from source vertex s to all other vertices in $O(VW + E)$ time, $O(VW + E)$ space.

Complexity: Algo achieves $O(VW + E)$ time by using bucket based priority queue operations. Space complexity is $O(VW + E)$ due to bucket array of size $O(VW)$ and total storage of $O(E)$ elements across all buckets.

Formal proof:

⇒ **Processing order:** Vertices are processed in non-decreasing order of their shortest path distances. This is ensured because we process buckets in increasing order. We completely exhaust each bucket before moving to next lines ⑧ inside outer while. When relaxing edge (u, v), if distance[v] is updated, v is placed in bucket B[new_d] where new_d ≥ distance[u] since $w(u, v) \geq 0$

⇒ Only vertices with current distance equal to bucket index are processed. If distance[u] < min, it means u was re inserted into lower numbered bucket due to better path found later, this stale entry could be skipped.

⇒ When min exceeds max_dist, all reachable vertices have been processed with correct shortest distances.

⇒ Algo correctly computes shortest paths.

Time complexity:

⇒ Outer loop iterates from 0 to max_dist = $O(VW)$ iterations

⇒ Each vertex is processed atmost once with final distance. Due to lazy deletion, vertex may be removed from buckets multiple times. Each edge relaxation that successfully updates distance[v] causes exactly 1 insertion. Total no. of successful relaxations = total insertions = total removals = $O(E)$

⇒ Each edge relaxed exactly once when its tail u is processed with its final distance. O(E) work

⇒ Total time = O(VW + E)

Space complexity:

⇒ Bucket array B has size max-dist + 1 → O(VW)

⇒ O(E) total elements across all buckets → O(E)

⇒ O(V) space for distance[]

⇒ Total space = O(VW + E + V) = O(VW + E)

---

7. Heapsort Algorithm:

```
⇒ Heapsort(A):
      n = A.length
      Build-max-heap(A)
      for i = n to 2:
          swap(A[1], A[i])
          max-heapify(A, 1, i-1)

⇒ Build-max-heap(A):
      n = A.length
      for i = [n/2] to 1:
          max-heapify(A, i, n)

⇒ Max-heapify(A, i, heap_size):
      left = 2 * i
      right = 2 * i + 1
      largest = i
      if left ≤ heap_size and A[left] > A[largest]:
          largest = left
      if right ≤ heap_size and A[right] > A[largest]:
          largest = right
      if largest ≠ i:
          swap(A[i], A[largest])
          Max-heapify(A, largest, heap_size)
```

Theorem: Best case running time of heapsort on array A of n distinct elements is $\Omega(n \log n)$.

Complexity: Heapsort uses $O(1)$ additional space.

Lower bound $\Omega(n \log n)$ is in best case due to comparisons.

Formal proof:

① Build-max-heap: Construct max-heap in $O(n)$ time

② Repeated extraction: Removes max element and restores heap property $n-1$ times.

⇒ In max-heap of n distinct elements, $\lceil n/2 \rceil$ leaves are smallest elements by max-heap property

⇒ During extraction phase, for each i from n to 2, we swap root with element at position i. Element at position i is initially leaf, one of smallest elements in current heap.

⇒ After swap, max-heapify is called on new root. Since new root is one of smallest elements, children of root are from set of larger elements (non leaves), new root must be smaller than atleast one child. So, atleast one swap must occur, element must sort down through heap

Lower bound analysis:

⇒ $T(n)$ = no. of comparisons during extraction phase

$$\Rightarrow T(n) \geq \sum_{i=2}^{n} \lceil \log (i-1) \rceil = \sum_{k=1}^{n-1} \lceil \log k \rceil$$

no. of comparisons required by max-heapify is atleast height of heap.

$$\Rightarrow T(n) \geq \sum_{k=\frac{n}{2}}^{n-1} \lceil \log k \rceil$$

⇒ For $k \geq \frac{n}{2}$, $\lceil \log k \rceil \geq \lceil \log(\frac{n}{2}) \rceil = \lceil \log$

$T(n) \geq \frac{n}{2}\lceil \log n - 1 \rceil = \Omega(n \log n)$

$O(n)$ comparisons $\longrightarrow \Omega(n \log n)$

Best case running time of heapsort is $\Omega(n \log n)$

---

8. Algorithm:

⇒ Input: Directed graph $G = (V, E)$ with no self loops

⇒ Output: $|V|^2 \times |V|$ matrix BB^T

Pseudocode:

⇒ ① $n = |V|$

② Initialize $n \times n$ matrix A with all entries 0

③ For each edge e in E with tail u, head v

$A[u][u] \leftarrow A[u][u] + 1$

$A[v][v] \leftarrow A[v][v] + 1$

$A[u][v] \leftarrow A[u][v] - 1$

$A[v][u] \leftarrow A[v][u] - 1$

④ Return A

Theorem: Let $G = (V, E)$ be directed graph with no self loops, B is incidence matrix. $A = BB^T$.

① For each vertex i, $A[i][i]$ = degree of vertex i i.e total no. of edges incident to i

② For $i \neq j$, off diagonal entry $A[i][j]$ is negative of no. of edges b/w vertices i and j.

⇒ Time complexity = $O(|V|^2 + |E|)$ → pass over edges.
↳ matrix initialization

⇒ Space complexity = $O(|V|^2)$ → matrix A.

Formal proof: $A = BB^T$.

$\Rightarrow A[i][j] = \sum_{e \in E} B[i][e]\, B[j][e]$

Case 1: $i = j$: $L[i][i] = \sum_{e \in E} (B[i][e])^2$

$(B[i][e])^2 = 1$ iff edge e is incident to vertex i

$L[i][i]$ = no of edges incident to i = degree of vertex i

Case 2: $i \neq j$: $L[i][j] = \sum_{e \in E} B[i][e] \cdot B[j][e]$

$\Rightarrow B[i][e] \cdot B[j][e]$ is non-zero iff both i, j are incident to edge e; e connects i and j.

① e directed from i to j. $B[i][e] = -1$. $B[j][e] = 1$

product $= -1$

② e directed from j to i. $B[i][e] = 1$, $B[j][e] = -1$.

product $= -1$

$\Rightarrow L[i][j] = -a_{ij}$, a_{ij} = no. of edges b/w i, j

9. Adjacency list representation algorithm:

$\Rightarrow n = |V|$

create new adjacency list G^2 with n empty nodes.

for each vertex $u \in V$

create a set $S = \phi$

for each vertex w in $G.adj[u]$

$S = S \cup \{w\}$

for each vertex v in $G.adj[w]$

$S = S \cup \{v\}$

$G^2.adj[u]$ = list of vertices in S

return G^2

Adjacency matrix representation algorithm:

⇒ $n = |V|$

let A be G_1's $n \times n$ adjacency matrix

Create $n \times n$ matrix G_1^2 initialized to 0

```
for i = 1 to n
    for j = 1 to n
        if A[i][j] = 1, then G1^2[i][j] = 1
        else
            for k = 1 to n
                if A[i][k] == 1 and A[k][j] == 1:
                    G1^2[i][j] = 1
                    break
return G1^2
```

Theorem: Given directed graph $G_1 = (V, E)$, algorithms correctly compute square graph $G_1^2 = (V, E^2)$ $(u, v) \in E^2$ iff G_1 contains path from u to v with atmost 2 edges.

⇒ Complexity of adjacency list:

time $= O(V + E + \sum_{(u,w) \in E} \deg(w)) = O(VE)$ in worst case

space $= O(V+E)$ input, $O(V+E^2)$ output

⇒ Complexity of adjacency matrix:

time $= O(V^3)$ space $= O(V^2)$

Proof of adjacency list algorithm:

⇒ (u, v) be any pair of vertices. We show $(u, v) \in E^2$ iff algo adds v to G_1^2.adj[u].

⇒ Suppose $(u, v) \in E^2$. ∃ path from u to v with atmost 2 edges.

⇒ ① If path is $u \to v$, $v \in G_1.adj[u]$. When processing u, we iterate through each w in $G_1.adj[u]$. When $w = v$, algorithm adds v to S.

② If path is $u \to w \to v$ for some w, $w \in G.adj[u]$, $v \in G.adj[v]$. When processing u and iterating through $w \in G.adj[u]$, we find w. We iterate through $G.adj[w]$, find v, add it to S.

⇒ Suppose algo adds v to $G^2.adj[u]$. v was added to S either

① $v \in G.adj[u]$

② $w \in G.adj[u]$, $v \in G.adj[w]$

⇒ $(u,v) \in E^2$ by definition

⇒ Time = $O(V + E + \sum_{w \in adj[u] \text{ (or) } (u,w) \in E} deg(w))$

Worst case → $deg(w) = O(V)$ → time = $O(VE)$

Proof of adjacency matrix algorithm:

⇒ (u,v) are any pair of vertices. $G^2[u][v] = 1$ iff $(u,v) \in E^2$

⇒ If $G^2[u][v] = 1$ then either $A[u][v] = 1$, $(u,v) \in E$ (1-edge path) or $\exists\ k$, $A[u][k] = 1$, $A[k][v] = 1$, $u \to k \to v$

⇒ If $(u,v) \in E^2$ then either $(u,v) \in E$, $A[u][v] = 1$, $G^2[u][v] = 1$ or $\exists\ u \to k \to v$, $A[u][k] = 1$, $A[k][v] = 1$ for some k, $G^2[u][v] = 1$

⇒ Algorithms correctly compute G^2.