EECS3101 Algorithms (Summer'26)

Digest of Tutorial 8, July 9

Dijkstra's Algorithm: Time Complexity, Graph Representation Tradeoffs

Jackie Wang

How to use this Digest

This digest **is** meant to help you identify the main ideas, important examples, common pitfalls, and practice targets from the lecture. It is **not** a replacement for attending class, watching the lecture recording, reviewing the slides and iPad notes, or doing your own practice and reflection. The digest intentionally summarizes only the roadmap and highlights; **you are expected to fill in the details** by revisiting the lecture materials, tracing examples, writing code, solving problems, and checking your own understanding. A good way to use this digest is as a checklist: after reviewing the lecture, you should be able to explain and apply the items listed under "*What you should be able to do*".

This digest is an AI-assisted study roadmap reviewed by Jackie. It was produced from the lecture transcript, a carefully designed tool workflow, and Jackie's actual hand-written iPad notes. Although the digest is reviewed before being posted, it **might** still contain typos, inconsistencies, omissions, or errors, just like any manually prepared course material. Therefore, use it as a study guide, **not** as an unquestionable authority. If something seems unclear or inconsistent, you are responsible for asking for clarification and verifying your understanding. Possible errors in the digest should **not** be treated as permission to submit incorrect work, or as a substitute for understanding the course materials.

Table of Contents

Main Ideas

- This tutorial completed the heap-based running-time analysis of Dijkstra's algorithm and explained why the final bound is $O((n+m)\cdot\log n)$ in terms of vertices $n = |V|$ and edges $m = |E|$.
- The most important counting idea was that the inner neighbour-processing work is counted by summing vertex degrees, giving m for directed graphs and $2m$ for undirected graphs, not by blindly multiplying loop headers.
- Heap implementation details matter: reading the minimum at the root is $O(1)$, but insertion, deletion, and decrease-key can require bubbling and therefore cost $O(\log n)$.
- Jackie emphasized that auxiliary checks, such as testing whether a vertex is still in the priority queue, must be implemented efficiently; an $O(n)$ scan inside the inner loop would damage the intended analysis.
- The second half of the tutorial compared graph implementation strategies: edge list, adjacency list, and adjacency matrix.
- The adjacency-list strategy extends the edge-list idea by storing incident-edge lists at vertices and position links at edges, allowing efficient edge removal when all related lists are updated correctly.
- The adjacency-matrix strategy stores edge references in a graph-level 2D array indexed by vertices, making edge lookup direct but making dynamic vertex addition expensive.
- For Written Test 2, the graph lecture coverage was confirmed as up to and including the graph-representation comparison table on slide 75; later graph topics next week were excluded from that test coverage.

Roadmap

Dijkstra analysis setup

The tutorial began by returning to the graph lecture and using an alternative pseudocode form of Dijkstra's algorithm because it is easier to analyze line by line. The priority queue Q was assumed to be implemented by a heap, so the analysis reused the heap facts from Lecture 18: heap height is $O(\log n)$, the minimum is found at the root, and bubbling after insertion, deletion, or key change takes logarithmic time. Jackie also reminded students that the graph may be directed or undirected, but Dijkstra still requires non-negative edge weights.

Initialization and heap construction

The first part of the pseudocode initializes the distance array or map D and the ancestor array or map a. These are simple assignments over the vertices, so the initialization lines for

D and a contribute $O(n)$. The next step inserts all n vertices into the heap-backed priority queue. Since one insertion can require up-heap bubbling, a single insertion is $O(\log n)$, and inserting all vertices costs $O(n \cdot \log n)$.

The outer loop and the minimum vertex

The main loop continues while Q is not empty. Since all vertices are inserted and each vertex is eventually removed or finalized once, the outer loop runs n times. Jackie separated two related operations: simply reading the minimum at the heap root is $O(1)$, but removing the minimum and restoring the heap structure is $O(\log n)$. Therefore, across the whole run, the removals contribute $O(n \cdot \log n)$.

Counting the inner loop correctly

The key technical point was the inner loop over the vertices adjacent to the current vertex u. A tempting but misleading approach is to treat the nested loop as automatically n^2. Jackie instead counted how often the inner body runs over the entire algorithm: for each finalized vertex u, the loop visits the incident or outgoing edges of u. Summing these degrees over all vertices gives m in a directed graph, or $2m$ in an undirected graph, which is still $O(m)$. This is the reason the edge term appears in the final bound.

Decrease-key and membership checks

Inside the inner loop, updating $D[v]$ is not just a constant-time assignment if $D[v]$ is the key stored in a heap. Decreasing a key may break the Heap Order Property, so the heap may need up-heap bubbling, giving $O(\log n)$ for that update. Jackie also discussed the cost of checking whether v is still in Q: this membership test must be $O(\log n)$ or better, for example by maintaining an appropriate auxiliary balanced search structure, rather than by doing a linear scan.

Putting the Dijkstra bound together

Combining the pieces gives $O(n)$ for simple initialization, $O(n \cdot \log n)$ for building the heap, $O(n \cdot \log n)$ for removals, and $O(m \cdot \log n)$ for edge relaxations and related efficient checks. The resulting general bound is

$$O((n + m) \cdot \log n).$$

Jackie then explained that if the graph is dense, so m can be on the order of n^2, this may be written as $O(n^2 \cdot \log n)$ for that special case. The more accurate general form remains $O((n + m) \cdot \log n)$.

Moving from Dijkstra to graph representations

After finishing the Dijkstra analysis, Jackie introduced the main exercise for Tutorial 8: compare the running times of graph operations under three implementation strategies. Students had already worked with edge lists through the assignments. The tutorial then introduced

adjacency lists and adjacency matrices and pointed students to the feature summary and time-complexity table in the graph slides.

Adjacency list strategy

The adjacency-list strategy keeps the familiar lists of vertices and edges, but each vertex additionally stores a list of incident edges. Each edge also stores positions in the main edge list and in the incident-edge lists of its endpoints. The visualization with vertices A, B, C and edges O and P showed how vertex objects, edge objects, list positions, and incident-edge lists point to one another at run time. This is the hardest diagram in the tutorial, and Jackie explicitly encouraged students to map each arrow back to a Java attribute.

Removing an edge from an adjacency list

For an edge such as $O = (B, A)$ in the adjacency-list diagram, removal must happen in three places: the origin vertex's incident-edge list, the destination vertex's incident-edge list, and the main edge list. Because stored positions point directly to the relevant list nodes, each removal can be $O(1)$. The main lesson is not only the final bound, but also the need to update every related structure so the graph representation does not leave stale references behind.

Adjacency matrix strategy

The adjacency-matrix strategy stores a graph-level 2D array of edge references. Each vertex has an index used to locate the corresponding row and column. In the undirected example, the entries for (A, B) and (B, A) point to the same edge object, so the matrix is symmetric through aliasing; a null entry means that no edge is currently present. Rows or columns can be used to inspect incident edges, and edge existence checks can be direct.

Tradeoffs and the Tutorial 8 comparison table

Jackie closed by directing students to slides 74 and 75: slide 74 summarizes the ingredients of the three graph strategies, and slide 75 gives the comparison table for operations such as `numVertices()`, `vertices()`, `getEdge(u,v)`, `insertVertex(x)`, `removeVertex(v)`, `insertEdge(u,v,x)`, and `removeEdge(e)`. Students should not only memorize the table; they should be able to derive entries from the underlying representation.

Key Terms, Notation, and Structures

- $n = |V|$ **and** $m = |E|$: n denotes the number of vertices and m denotes the number of edges in the input graph.
- **Heap-backed priority queue** Q: The priority queue used by Dijkstra stores vertices using their current D values as keys.

- **Find minimum versus remove minimum**: Reading the minimum at the heap root is $O(1)$, while removing it and restoring the heap is $O(\log n)$.
- **Decrease-key**: The operation of lowering a vertex's key $D[v]$ in the heap; it may require up-heap bubbling and therefore costs $O(\log n)$ in this analysis.
- **Degree**: The number of incident edges of a vertex. In this tutorial, a lowercase d in the graph-representation table means degree, not Dijkstra's true-distance notation from previous lectures.
- **Sum of degrees**: For directed graphs, summing the relevant outgoing degrees across all vertices gives m; for undirected graphs, summing degrees gives $2m$, which is still $O(m)$.
- **Dense graph**: A graph whose number of edges can be on the order of n^2; in this case $O((n+m)\cdot\log n)$ can specialize to $O(n^2\cdot\log n)$.
- **Edge list**: A graph representation with a list of vertices and a list of edges; it is the strategy students had already used in the assignments.
- **Adjacency list**: A graph representation that stores, for each vertex, a list of its incident edges, in addition to global vertex and edge lists.
- **Incident-edge list**: The per-vertex list that stores the edges touching that vertex in the adjacency-list strategy.
- **Position pointers**: Stored references to positions in linked lists; they allow constant-time removal once the relevant object is known.
- **Adjacency matrix**: A graph representation using a graph-level 2D array of edge references, with vertex indices used as row and column indices.
- **Aliasing in an undirected matrix**: In an undirected graph, `matrix[i][j]` and `matrix[j][i]` may refer to the same edge object.
- **Null matrix entry**: A null entry in the adjacency matrix means there is no edge between the corresponding pair of vertices.

Key Examples and Demonstrations

Line-by-line Dijkstra running-time analysis

Setup: The pseudocode uses a graph $G = (V, E)$, a source vertex s, distance values D, ancestor values a, and a heap-backed priority queue Q.

Main point: The final complexity comes from combining several different costs: simple vertex-wide initialization, heap insertions, n removals, and edge-based inner-loop work. The important result is $O((n+m)\cdot\log n)$, not a generic nested-loop guess.

What to practise: Recreate the analysis line by line: label which lines are $O(n)$, which are $O(n\cdot\log n)$, where $O(m)$ inner-loop executions come from, and why each key update can add a $\log n$ factor.

Counting inner-loop work by degrees

Setup: For each removed vertex u, the algorithm examines vertices or edges adjacent to u.

Main point: Across the whole algorithm, these examinations correspond to the graph's edges. In a directed graph the total is m; in an undirected graph it may be $2m$, which is still asymptotically $O(m)$.

What to practise: Draw a small directed graph and a small undirected graph. List each vertex degree and sum the values. Then connect that sum to how many times Dijkstra's inner-loop body can execute.

Adjacency-list object-and-pointer diagram

Setup: The notes use vertices A, B, C and edges O and P to show global vertex and edge lists, vertex objects, edge objects, incident-edge lists, and stored positions.

Main point: Adjacency lists are faster than pure edge lists for many local operations because each vertex directly knows its incident edges. The price is a more sophisticated object structure with multiple pointers that must remain consistent.

What to practise: Redraw the diagram with one additional edge. For each edge object, mark its origin, destination, main edge-list position, and positions in the endpoint incident-edge lists.

Constant-time edge removal in an adjacency list

Setup: To remove an edge, the representation must remove it from the origin's incident-edge list, the destination's incident-edge list, and the global edge list.

Main point: With stored positions and doubly linked lists, each of the three removals can be $O(1)$. The correctness of the representation depends on doing all three updates, not just deleting the edge from one visible place.

What to practise: Pick one edge in the adjacency-list diagram and write the three conceptual removal actions. Then explain why each one is constant time only because a position is already stored.

Adjacency matrix visualization and vertex insertion

Setup: The matrix example maps vertices such as B, A, and C to indices 0, 1, and 2, and stores edges in a 2D array of edge references.

Main point: Matrix entries make edge existence checks direct, but adding a new vertex to a primitive array-based matrix requires allocating a larger matrix and copying the old entries, giving $O(n^2)$ work.

What to practise: Draw the 3×3 matrix for the example, then add a fourth vertex. Mark which entries are copied, which entries are new nulls, and why the copy step is quadratic.

Graph representation comparison table

Setup: The tutorial exercise asks students to compare operations under edge list, adjacency list, and adjacency matrix strategies.

Main point: Each table entry should be derived from the representation, not memorized. For example, `getEdge(u,v)` depends on whether you scan all edges, scan a smaller incident list, or index directly into a matrix.

What to practise: Choose three operations from slide 75 and write the step-by-step procedure for each representation before checking the table or the solution walkthrough.

Common Pitfalls and Misconceptions

Blindly treating the nested loops in Dijkstra as n^2

Pitfall: Assuming that an outer loop over vertices and an inner loop over adjacent vertices automatically costs $O(n^2)$.

Why it causes problems: This ignores the graph structure. The inner loop work is governed by the total number of incident edges examined, which sums to m or $2m$, not necessarily n^2.

How to avoid it: Whenever a graph algorithm loops over neighbours, try to rewrite the total count as a sum of degrees before deciding the asymptotic bound.

Forgetting that decrease-key is not just an assignment

Pitfall: Treating a change to $D[v]$ as $O(1)$ even though $D[v]$ is the heap key.

Why it causes problems: Changing a heap key can violate the Heap Order Property, so the heap may need up-heap bubbling to restore its relational structure.

How to avoid it: Each time an algorithm changes a value used as a priority-queue key, ask what repair operation the data structure needs.

Confusing root access with heap deletion

Pitfall: Saying that `extractMin` is always $O(1)$ or always $O(\log n)$ without clarifying whether it only reads the root or removes it.

Why it causes problems: The root can be read in constant time, but removing it and restoring completeness and the Heap Order Property takes logarithmic time.

How to avoid it: In complexity explanations, state the operation semantics: reading or peeking at the minimum is $O(1)$; deleting or finalizing it is $O(\log n)$.

Using a linear membership check inside Dijkstra's inner loop

Pitfall: Checking whether v is still in Q by scanning a linear collection.

Why it causes problems: An $O(n)$ membership check inside edge processing can dominate the intended heap-based analysis.

How to avoid it: Use an auxiliary structure with $O(\log n)$ or better membership checks, and include that cost explicitly in the reasoning.

Memorizing the graph-representation table without derivation

Pitfall: Treating slide 75 as a table to memorize mechanically.

Why it causes problems: Jackie may ask for the reasoning behind a complexity, not just the final entry. Memorization also makes it easier to mix up edge list, adjacency list, and adjacency matrix costs.

How to avoid it: For each operation, first write what the representation must touch: all vertices, all edges, one incident list, one matrix row or column, or the whole matrix.

Removing an adjacency-list edge from only one place

Pitfall: Deleting an edge from the global edge list but forgetting its endpoint incident lists, or deleting it from one incident list but not the other.

Why it causes problems: The representation will contain dangling or inconsistent references, so later degree, incident-edge, or traversal operations can return wrong results.

How to avoid it: For edge removal, use the checklist: origin incident list, destination incident list, and global edge list.

Assuming the adjacency matrix is always the best representation

Pitfall: Choosing the adjacency matrix only because edge existence checks are direct.

Why it causes problems: A primitive array-based matrix can be expensive for dynamic vertex operations, especially adding a vertex, which requires creating a larger matrix and copying n^2 entries.

How to avoid it: Choose the graph representation according to the operations your application performs most frequently.

What You Should Be Able to Do

After this lecture, you should be able to:

- explain why Dijkstra's algorithm with a heap-backed priority queue has running time $O((n+m)\cdot\log n)$;
- identify where the $O(n\cdot\log n)$ and $O(m\cdot\log n)$ terms come from in the pseudocode;
- justify why the total neighbour-processing count is $O(m)$ using the sum of degrees;
- distinguish reading the minimum heap root from removing the minimum and restoring the heap;
- explain why decreasing a heap key requires up-heap bubbling in the worst case;
- describe why a membership check such as $v \in Q$ must not be implemented by an $O(n)$ scan inside Dijkstra's inner loop;
- draw the main object structure of an adjacency-list graph, including vertex lists, edge lists, incident-edge lists, and position links;

- explain the three updates needed to remove an edge from an adjacency-list representation;
- draw an adjacency matrix for a small undirected graph and explain how symmetric entries can alias the same edge object;
- explain why adding a vertex to an array-based adjacency matrix costs $O(n^2)$;
- compare edge list, adjacency list, and adjacency matrix strategies by deriving selected operation costs rather than only recalling the final table;
- locate the graph-lecture coverage for Written Test 2 and use slide 75 as a derivation target, not just a memorization target.

Suggested Practice

- Reproduce the Dijkstra running-time calculation from the pseudocode without looking at the digest: initialization, heap construction, outer loop count, inner-loop count, key updates, removals, and final simplification.
- Draw one directed and one undirected graph, compute each vertex degree, and verify the appropriate sum-of-degrees count used in the Dijkstra analysis.
- Review heap insertion, deletion, and decrease-key until you can explain which operation uses up-heap bubbling and which uses down-heap bubbling.
- Complete the Tutorial 8 graph-representation exercise: for each operation, write the steps under edge list, adjacency list, and adjacency matrix before checking slide 75.
- Redraw the adjacency-list runtime diagram with vertices and edges as objects. Label which arrows are ordinary object attributes and which are stored positions for efficient removal.
- Practise removing one edge from an adjacency-list diagram by crossing it out from all three required places.
- Redraw the adjacency-matrix example and add a new vertex. Show the new matrix size and explain where the $O(n^2)$ copy cost appears.
- Use the Assignment 2 solution and the newly released Assignment 3 material to connect the graph representations back to the programming-test preparation path.
- For Written Test 2, practise explaining at least a few table entries in words, especially `getEdge(u,v)`, `removeVertex(v)`, `insertVertex(x)`, and `removeEdge(e)`.

Where to Review

- Review the iPad notes page on Dijkstra's time complexity, especially the annotated pseudocode, the $n = |V|$, $m = |E|$ notation, the heap facts, and the final $O((n + m) \cdot \log n)$ bound.
- Revisit the part of the recording where Jackie explains why the inner loop contributes $O(m)$ work. This is the key reasoning step students found less intuitive during class.
- Review the adjacency-list strategy pages in the notes, including the Java class attributes and the runtime object diagram with vertices A, B, C and edges O, P.
- Review the adjacency-matrix strategy pages, especially the graph-level 2D edge array, vertex index attributes, null entries, and undirected aliasing.
- Study slides 74 and 75 of the graph slides: slide 74 summarizes the ingredients of the graph strategies, and slide 75 is the comparison table for operation running times.
- Use the Tutorial 8 instruction and solution walkthrough video if your derived table entries do not match slide 75.
- Review the heap insertion-versus-deletion exercises from Lecture 18, because the Dijkstra analysis depends on the $O(\log n)$ heap-operation costs.
- For Written Test 2, use Jackie's posted guide and the confirmed graph-lecture coverage up to and including slide 75; topics introduced after this point next week were not part of that written-test coverage.

Closing Reminder

This digest gives you the roadmap, but the learning work is in the tracing and derivation. Revisit the Dijkstra pseudocode, redo the degree-sum argument, and practise explaining why each heap operation costs what it costs. Then use the graph-representation diagrams and slide 75 table as active exercises: for each operation, identify what data structure is touched and derive the time bound yourself before checking the posted solution or recording.