CHECKLIST

Space-Efficient Dijkstra Variants: Production Checklist

A practical checklist for evaluating and deploying memory-conscious Dijkstra implementations on large graphs.

Space-Efficient Dijkstra Variants for Large-Scale Graphs

Production Checklist

Use this checklist to decide whether a space-efficient Dijkstra variant is appropriate, and to validate it before production deployment.

---

1) Confirm the operational need

- [ ] The graph workload is failing or nearing failure due to memory pressure, not CPU alone.
- [ ] Peak memory is a clearer risk than raw runtime.
- [ ] The graph is large enough that adjacency storage, distance tables, predecessor tracking, and heap state are all meaningful contributors.
- [ ] The deployment environment has a fixed memory ceiling, such as:
- [ ] container memory limit
- [ ] VM memory cap

- [ ] embedded/runtime limit
- [ ] shared host constraints
- [ ] The team has a reason to trade some CPU time or implementation simplicity for a smaller working set.

---

2) Define the actual query shape

- [ ] The problem is single-source shortest path.
- [ ] The problem is single-source to single-target.
- [ ] The problem is bounded-radius or cutoff-based exploration.
- [ ] The problem requires full-source exploration.
- [ ] The expected query shape is documented before choosing the implementation.

Notes:

- Single-target and bounded queries often benefit most from frontier reduction.
- Full-source workloads may need stronger storage optimization to remain feasible.

---

3) Decide what output you really need

- [ ] The business requirement is distance only.
- [ ] The business requirement includes the actual path.
- [ ] The team has confirmed whether predecessor tracking is mandatory.
- [ ] If paths are required, a compact reconstruction strategy is defined.
- [ ] If paths are not required, predecessor storage is omitted entirely.

Rule of thumb:

- Distance-only workloads can save memory by dropping path reconstruction state.
- Path-required workloads must preserve enough information to rebuild the route correctly.

---

4) Measure the baseline before changing the algorithm

- [ ] Baseline peak memory is measured on a representative graph.
- [ ] Baseline runtime is measured on the same data set.
- [ ] Baseline results are captured for correctness comparison.
- [ ] The measurement includes the reachable subgraph, not just total graph size.
- [ ] The largest production-like graph is included in validation.
- [ ] The measurement environment matches production as closely as practical.

Record these values:

- Baseline peak memory: _____
- Baseline runtime: _____
- Baseline output verified against: _____
- Largest tested graph size: _____

---

5) Evaluate graph representation options

- [ ] The current representation uses pointer-heavy objects and may be memory-inefficient.
- [ ] A compact adjacency array representation has been considered.
- [ ] Node identifiers can be stored as integers instead of objects or strings where appropriate.
- [ ] Edge data can be packed into contiguous structures.
- [ ] The representation improves cache locality as well as memory use.
- [ ] The cost of converting from source format to compact format is acceptable.

Prefer compact storage when:

- the graph is large

- traversal is frequent
- memory overhead per node or edge is significant
- cache locality matters in the hot path

---

6) Evaluate distance storage options

- [ ] A dense distance array is truly necessary.
- [ ] A sparse map or lazy materialization strategy would be sufficient.
- [ ] The number of reachable nodes is often far smaller than the total number of vertices.
- [ ] Distance values fit into the narrowest safe numeric type.
- [ ] Overflow behavior is understood and tested.
- [ ] Unreachable vertices are handled explicitly.

Check carefully:

- Smaller integer types reduce memory only if they cannot overflow on valid paths.
- Sparse tracking is most effective when exploration touches a limited subset of the graph.

---

7) Evaluate frontier / priority-queue strategy

- [ ] The implementation can tolerate stale heap entries.
- [ ] Decrease-key support is not required for correctness.
- [ ] Duplicate queue entries are acceptable if they reduce implementation complexity.
- [ ] Queue growth has been profiled under worst-case conditions.
- [ ] Heap churn does not exceed the memory budget.
- [ ] Queue cleanup or stale-entry checks are implemented correctly.

Validation point:

- When a node is popped from the queue, the implementation must verify that the popped distance is still the best known distance.

---

8) Confirm algorithmic correctness assumptions

- [ ] All edge weights are non-negative.
- [ ] The chosen variant preserves Dijkstra?s correctness conditions.
- [ ] Settled-node behavior is unchanged.
- [ ] Early exit logic, if used, is valid for the query shape.
- [ ] Any pruning or bounding rule is safe and documented.
- [ ] No optimization breaks shortest-path guarantees.

Important:

- Space optimization must not change the mathematical result.
- If negative weights are possible, Dijkstra is not the right tool.

---

9) Decide whether bidirectional or bounded search fits

- [ ] The query has a known target.
- [ ] The graph semantics support bidirectional search.
- [ ] The search can be bounded by radius, cost, or hop limit.
- [ ] A reduced frontier will materially lower memory use.
- [ ] The implementation team understands the limitations of the chosen bound.
- [ ] The stopping condition is tested and documented.

Use these only when appropriate:

- Single-source to single-target queries
- Controlled search spaces

- Workloads where full graph exploration is unnecessary

---

10) Test memory peak under realistic worst cases

- [ ] Peak memory is tested on the largest expected graph.
- [ ] Peak memory is tested on a graph with a large reachable component.
- [ ] Peak memory is tested under a stressed frontier expansion pattern.
- [ ] The implementation stays below the production limit with safety margin.
- [ ] The process does not page, thrash, or trigger OOM eviction.
- [ ] Memory usage remains stable across repeated runs.

Recommended margin:

- Keep peak memory comfortably below the hard limit, not just barely under it.

---

11) Validate correctness against a trusted baseline

- [ ] Outputs match a known-correct reference implementation.
- [ ] Distances match for all tested nodes.
- [ ] Paths match where path output is required.
- [ ] Unreachable nodes are represented consistently.
- [ ] Tie-breaking behavior is understood if multiple equal shortest paths exist.
- [ ] Results are compared across several graph shapes:
- [ ] sparse graphs
- [ ] dense graphs
- [ ] disconnected graphs
- [ ] long-chain graphs
- [ ] graphs with many equal-weight edges

---

12) Profile CPU trade-offs explicitly

- [ ] The CPU cost of the memory-saving variant has been measured.
- [ ] The runtime increase is acceptable for the use case.
- [ ] Any extra preprocessing is justified.
- [ ] Any additional conversion step is amortized or acceptable.
- [ ] The team has confirmed that memory savings matter more than maximum throughput.

Remember:

- Space-efficient variants often trade memory for CPU.
- That trade is good only when the operational constraints demand it.

---

13) Check production readiness

- [ ] Memory profile is documented.
- [ ] Runtime profile is documented.
- [ ] Correctness tests pass.
- [ ] Failure modes are understood.
- [ ] Observability exists for memory, queue growth, and execution time.
- [ ] Rollback plan is available.
- [ ] The implementation choice is recorded with rationale.
- [ ] The team knows when to revisit the design.

---

14) Red flags that usually mean ?do not deploy yet?

- [ ] The graph still fits only in development, not in production.
- [ ] Peak memory has not been measured on production-like data.
- [ ] Path reconstruction was removed but downstream consumers still need paths.
- [ ] Negative or unknown-weight edges might exist.
- [ ] Queue behavior has not been tested at scale.
- [ ] The optimization was chosen for elegance rather than an actual memory problem.
- [ ] No correctness baseline exists.

---

15) Final go/no-go decision

Use this quick decision gate before deployment:

- [ ] Memory pressure is the primary constraint.
- [ ] The selected variant reduces working-set size in a measurable way.
- [ ] The variant preserves correctness for non-negative weights.
- [ ] The output requirements are still satisfied.
- [ ] Peak memory is safely below the limit.
- [ ] Runtime remains acceptable.
- [ ] The implementation has been validated on production-like graphs.

Decision

- [ ] Go ? the space-efficient variant is appropriate for production.
- [ ] No-go ? revisit graph representation, query scope, or output requirements.

Quick implementation summary

If you need a compact checklist for the engineering ticket, use this:

- [ ] Measure baseline memory and runtime
- [ ] Identify query shape: full-source, single-target, or bounded
- [ ] Confirm whether path reconstruction is required
- [ ] Use compact graph storage where possible
- [ ] Use sparse distance tracking when reachable nodes are limited
- [ ] Avoid unnecessary predecessor storage
- [ ] Accept stale heap entries if correctness checks are in place
- [ ] Validate peak memory against production limits
- [ ] Compare outputs with a trusted baseline
- [ ] Deploy only after confirming the memory trade-off is worth the CPU cost

---

Notes for the team

Implementation rationale:

Graph characteristics:

Production constraints:

Validation results: