TEMPLATE

Efficient Dijkstra Variants Selection Template

A practical, vendor-neutral worksheet for choosing the right Dijkstra variant based on graph density, weight characteristics, and operational constraints.

Efficient Dijkstra Variants Selection Template

Use this worksheet to choose a Dijkstra implementation that fits your graph, workload, and operational constraints.

---

1) Workload summary

Use case: _____

System or domain: _____

Primary query pattern:

- [ ] One source to all reachable nodes
- [ ] One source to one target
- [ ] Repeated queries from the same source
- [ ] Repeated queries across many sources
- [ ] Dynamic / frequently changing graph

Latency target: _____

Memory limit: _____

Expected graph size:

- Vertices: _____
- Edges: _____

---

2) Graph characteristics

Graph density:

- [ ] Sparse
- [ ] Moderately dense
- [ ] Dense

Edge weights:

- [ ] All weights are non-negative
- [ ] Some weights may be negative ? Dijkstra is not suitable

Weight type:

- [ ] Floating point
- [ ] Integer
- [ ] Mixed

Weight range:

- [ ] Small bounded integers
- [ ] Wide integer range
- [ ] Unbounded / unknown
- [ ] Narrow and stable

Topology stability:

- [ ] Static
- [ ] Occasionally updated
- [ ] Frequently updated
- [ ] Highly dynamic

---

3) Candidate variant decision

Start here

- [ ] Binary heap Dijkstra
- Best default for sparse graphs and general non-negative weights
- Easier to implement, test, and maintain
- Recommended as the first production candidate
- [ ] Array-based Dijkstra
- Consider for dense graphs
- Useful when heap overhead outweighs the benefit of priority queue operations
- Simpler runtime behavior, but worse asymptotic performance on sparse graphs
- [ ] Bucket-based Dijkstra / Dial-style queue
- Consider when weights are small, non-negative, and bounded integers
- Strong candidate when queue operations dominate runtime
- Verify memory use and bucket range before adoption
- [ ] Radix heap / integer priority structure
- Consider for integer weights with larger ranges
- Good when monotonic distance behavior fits the workload
- Requires careful implementation and profiling
- [ ] Alternative approach
- [ ] A* search if a valid admissible heuristic exists
- [ ] Dynamic shortest-path strategy if the graph changes frequently
- [ ] Other: _____

---

4) Suitability checks

Non-negativity check

- [ ] All edge weights are confirmed non-negative
- [ ] Input validation rejects negative weights
- [ ] Data pipeline cannot introduce negative corrections silently

Performance check

Estimate what dominates runtime:

- [ ] Queue operations
- [ ] Edge relaxations
- [ ] Memory access
- [ ] Graph construction / preprocessing

If queue operations dominate, a specialized queue may help. If edge relaxations dominate, changing the queue may not matter much.

Operational check

- [ ] Implementation is easy to explain to on-call engineers
- [ ] Correctness is easy to test with known shortest-path cases
- [ ] Memory usage is acceptable under peak load
- [ ] Tail latency is acceptable on production-like data
- [ ] Failure modes are understood

---

5) Recommended decision path

- [ ] Confirm all weights are non-negative.
- [ ] Classify graph density.
- [ ] Check whether weights are bounded integers.
- [ ] Identify whether you need all-pairs-like repeated runs or a single run.
- [ ] Start with the simplest workable implementation.
- [ ] Profile on representative data.
- [ ] Switch only if the measured bottleneck justifies the complexity.

---

6) Production validation checklist

Before release, verify the following:

- [ ] Results match a trusted reference on small graphs
- [ ] Edge cases are covered:
- [ ] Disconnected nodes
- [ ] Zero-weight edges
- [ ] Duplicate edges
- [ ] Self-loops
- [ ] Large graphs near memory limits
- [ ] Queue behavior is stable under load
- [ ] No integer overflow in distance calculations
- [ ] Path reconstruction works if required
- [ ] Monitoring captures runtime, memory, and failure rates
- [ ] Rollback plan exists if performance regresses

---

7) Quick recommendation guide

Graph / workload profile	Preferred starting point	Notes
Sparse graph, general non-negative weights	Binary heap Dijkstra	Best default choice
Dense graph	Array-based Dijkstra	May reduce queue overhead
Small bounded integer weights	Bucket-based Dijkstra	Often fastest in practice
Integer weights with wider range	Radix heap	Profile carefully
Frequent topology changes	Dynamic shortest-path strategy	Reuse may be unsafe
Heuristic available, target-focused search	A*	Can outperform plain Dijkstra

---

8) Implementation notes to document

Record these decisions in your design notes:

- Selected variant: _____
- Why this variant fits the workload: _____
- Known limitations: _____
- Validation data set: _____
- Monitoring signals: _____
- Re-evaluation trigger: _____

---

9) Final sign-off

Chosen approach is safe for production?

- [ ] Yes
- [ ] No

Reviewer: _____

Date: _____

Comments:

_____

_____

_____