D* Lite Pathfinding

Gemmin Sugiura

April 2025

1 Introduction

This document provides a (hopefully simpler) explanation of the D* Lite pathfinding algorithm. It is based on the original research paper D* Lite by Sven Koenig and Maxim Likhachev. Readers interested in a more in-depth understanding are encouraged to refer to their published work.

D* Lite pathfinding addresses the problem with dynamic graphs (i.e. when vertices' traversability changes). It is the best algorithm if the target position does not change often.

2 Basic Definitions

- $G(V, E)$ denotes an undirected graph, where V denotes the set of vertices and E denotes the set of edges.
- $\text{NEIGH}(v) \subseteq V$ denotes the set of vertices that can be reached directly from v.
- $v_{\text{start}} \in V$ denotes the start vertex (i.e. the agent's current position). It is dynamic since the agent moves.
- $v_{\text{end}} \in V$ is the target vertex. It is static.
- $c(u, v) : E \to \mathbb{R}^+$ denotes the edge cost from vertex u to vertex v. G is undirected implying $c(u, v) = c(v, u)$.
- $u \in V$ is traversable iff $c(u, v) < \infty, \forall v \in \text{NEIGH}(u)$.
- $u \in V$ is intraversable iff $c(u, v) = \infty, \forall v \in \text{NEIGH}(u)$.
- $g(v) : V \to \mathbb{R}^+$ denotes the estimate length of the shortest path from v_{start} to v.
- $g^*(v) : V \to \mathbb{R}^+$ denotes the actual length of the shortest path from v_{start} to v.
- $h(v) : V \to \mathbb{R}^+$ is the heuristic function approximating the distance from v to v_{goal}. $h(v)$ is admissible and follows the triangle inequality.
- H denotes the heap (I use priority queue synonymously).

3 Lifelong Planning A* (LPA*) Pathfinding

Understanding LPA* is a prerequisite for learning D* Lite, as the latter builds upon the former's key concepts. There are three main ideas:

1. Right hand Side (RHS)
2. Local Consistency
3. Heap and Keys

3.1 Right Hand Side (RHS)

1. The $rhs(v)$ provides an alternative estimate of $g(v)$, computed using the vertex's local neighborhood information. This value serves as a "reality check." While $g(v)$ stores the current estimated cost from the start, $rhs(v)$ asks: Based on my neighbors' actual costs ($g(u)$) and their connection to me ($c(u, v)$), what should my cost reasonably be? Typically $g(v) \geq rhs(v)$, as the one-step look-ahead through neighbors often reveals better paths than the current $g(v)$ estimate. Formally, for an undirected graph:

$$rhs(v) = \begin{cases} \min\limits_{u \in \text{NEIGH}(v)} (g(u) + c(u, v)), & \text{if } v \neq v_{start} \\ 0 & \text{else} \end{cases}$$

3.2 Local Consistency

A vertex v is locally consistent when its stored cost $g(v)$ matches the best estimate of its neighbors: $g(v) = rhs(v)$. When all vertices are consistent, the g-values become true shortest-path costs (i.e. $g(v) = g^*(v)$). By induction from v_{goal} where $g(v_{\text{goal}}) = g^*(v_{\text{goal}}) = 0$:

$$\text{If } \forall u \in \text{NEIGH}(v)\ g(u) = g^*(u), \text{ then } g(v) = \min_u (c(v, u) + g^*(u)) = g^*(v)$$

Therefore, the optimal path can be found by using greedy from the start to the goal. There are two types of local inconsistency.

Type	**Condition**	**Meaning**
Overconsistent	$g(v) > rhs(v)$	Current cost estimate is too high (too pessimistic).
Underconsistent	$g(v) < rhs(v)$	Current cost is invalid.

3.3 Min Heap and Keys

Only locally inconsistent vertices need processing. Thus, the heap should consist of vertices that are locally inconsistent, that is, $g(v) \neq rhs(V), \forall v \in H$. Each vertex in the heap should have a priority, which we will call the key. Each key is a pair of two values $[k_1, k_2] = [\min(g(v), rhs(v)) + h(v, v_{\text{goal}}), \min(g(v), rhs(v))]$. The best way to understand the key is to first understand that $\min(g(v), rhs(v))$ is just an estimate of the length of the shortest path from v_{start} to v. Then k_1 is equal to the key used in A*, which is $f(v) = g(v) + h(v)$ (but slightly better). k_2 serves as a tiebreaker in the case where $k_1(v) = k_1(u)$; the keys are lexicographically ordered. The priority queue must maintain Invariant 3: $k(v) = \text{calculateKey}(v), \forall v \in H$, where $k(v)$ is the stored key, and calculateKey(v) is the actual priority. The heap must support the following $O(1)$ or $O(\log n)$ operations (we will look into the actual implementation of the heap later):

- `top()`: Returns a vertex with minimal key (k_1, k_2).
- `topKey()`: Returns a minimal key value ($[\infty, \infty]$ if empty).
- `contains(v)`: Returns $v \in H$.
- `pop()`: Removes and returns minimal vertex.
- `insert(v, k)`: Adds vertex v with key k.
- `update(v, k)`: Updates v's key to k (if changed).
- `remove(v)`: Removes vertex v from H.

3.4 Pseudocode

```
procedure CalculateKey(v)
    return [min(g(v), rhs(v)) + h(v, v_goal); min(g(v), rhs(v))]
procedure Initialize
    H = ∅
    for all v ∈ V do                                  ▷ In practice, use map
        rhs(v) = g(v) = ∞
    rhs(v_start) = 0
    H.Insert(v_start, CalculateKey(s_start))
procedure UpdateVertex(v)
    if v ≠ v_start then
        rhs(v) = min_{v'∈NEIGH(v)}(g(v') + c(v', v))
    if v ∈ H then
        H.Remove(v)
    if g(v) ≠ rhs(v) then
        H.Insert(v, CalculateKey(v))
procedure ComputeShortestPath
    while H.TopKey() < CalculateKey(s_goal) OR rhs(s_goal) ≠ g(s_goal) do
        v = H.Pop()
        if g(v) > rhs(v) then
            g(v) = rhs(v)
            for all s ∈ NEIGH(v) do
                UpdateVertex(s)
        else
            g(v) = ∞
            for all s ∈ NEIGH(v) ∪ {v} do
                UpdateVertex(s)
procedure Main
    Initialize()
    loop
```

ComputeShortestPath()
Wait for changes in edge costs
for all directed edges (u, v) with changed edge costs **do**
  Update the edge cost $c(u, v)$
  UpdateVertex(v)

4 D* Lite Pathfinding

Unlike LPA*, D* Lite performs a backward search from v_{goal}, making g(v) represent the estimated shortest path cost from v to v_{goal}. Like LPA*, I divided the learning into three sections, in which the supertopic is heap reordering.

1. Key Mechanics
2. Error Analysis
3. Correction Method

4.1 Key Mechanics

Recall the first element of any key: $k_1(v) = \min(g(v), rhs(v)) + h(v_{\text{start}}, v) + k_m$, where v_{start} is the *current position* of the agent. Since $\min(g(v), rhs(v))$ represents the length of the shortest path from v_{goal} to v, even if the agent moves $\min(g(v), rhs(v))$ is a *constant*. However, the heuristic $h(v_{\text{start}}, v)$ changes with respect to the agent's position. Consequently, the keys stored in the priority queue become obsolete because the stored keys were calculated using the old agent's position (i.e. $h(v_{\text{old start}}, v)$), but the actual priorities should be calculated using the current agent's position (i.e. $h(v_{\text{current start}}, v)$). Consequently, the stored keys violate Invariant 3. The naive fix is to recalculate the priorities of all vertices every time the agent moves.

4.2 Error Analysis

Intuitively, as the agent moves along or near the optimal path toward the target vertex, $h(v_{\text{start}}, v)$ should decrease, which implies that the stored keys become overestimates of the true priorities. Let $\varepsilon = \triangle k_1$ denote this error. Since the stored keys were calculated using $h(v_{\text{old start}}, v)$, and the actual priorities use $h(v_{\text{current start}}, v)$, $\varepsilon = \triangle k_1 = \triangle h = h(v_{\text{old start}}, v) - h(v_{\text{current start}}, v)$. ε can be upper-bounded by using the triangle inequality.

$$
\begin{aligned}
& h(v_{\text{old}}, v) \leq h(v_{\text{old}}, v_{\text{curr}}) + h(v_{\text{curr}}, v) && \text{(Triangle inequality)} \\
\implies & h(v_{\text{old}}, v) - h(v_{\text{curr}}, v) \leq h(v_{\text{old}}, v_{\text{curr}}) && \text{(Rearranged form)}
\end{aligned}
$$

4.3 Correction Method

Suppose that an agent moves from v to v' following an optimal path and detects changes in the edge cost(s), forcing new priorities to be computed. Then, from the result above, the new priorities will have decreased by at most $h(v_{\text{old start}}, v_{\text{current start}})$. In other words, when edge cost(s) change, the new priorities are $h(v, v')$ too small relative to $k_1(v), \forall v \in H$. To account for this error, one solution is to subtract $h(v, v'), \forall v \in H$. This solution leads to the keys becoming underestimates of the true priorities because we may subtract too much. However, underestimates of the true priorities are okay (lines 12-15). Thus, $v \in H$ are either *only* underestimates or have correct keys; there should be no vertices in H with keys that overestimate its priority.

Now, note that subtracting each $\forall v \in H$ by $h(v, v')$ does not change the order of the vertices in H (similar to subtracting a constant for every element in a sorted array). Thus, instead, equivalently, we could add $k_m = h(v, v')$ to the new priorities, which avoids reordering H (line 2). Note that adding $h(v, v')$ does not make the new priorities an overestimate, since $k_m = \sum \triangle h \leq h^*$ by the triangle inequality. In addition, adding $h(v, v')$ makes $v \in H$ underestimates.

D* Lite leverages the property that the stored k_1s in the priority queue are lower bounds on LPA*'s true priorities (plus k_m). When expanding the vertex v with minimum priority, we compare its heap key k_{old} with its recalculated key k_{new}. Since these stored keys are guaranteed to be underestimates, we check whether $k_{\text{old}} < k_{\text{new}}$ (line 14). This conservative approach ensures that we never expand vertices too late (which could compromise path optimality), while allowing safe early expansions. When an underestimate is detected, we simply reinsert v with its updated key k_{new}, maintaining the correct processing order relative to the true path costs.

4.4 Pseudocode

1: **procedure** CALCULATEKEY(v)
2:     **return** $[\min(g(v), rhs(v)) + h(v, v_{\text{start}}) + k_m, \min(g(v), rhs(v))]$
3: **procedure** UPDATEVERTEX(v)
4:     **if** $v \in H$ **and** $g(v) \neq rhs(v)$ **then**
5:         H.update(v, calculateKey(v))
6:     **else if** $v \notin H$ **and** $g(v) \neq rhs(v)$ **then**
7:         H.insert(v, calculateKey(v))
8:     **else if** $v \in H$ **and** $g(v) = rhs(v)$ **then**
9:         H.remove(v)
10: **procedure** COMPUTESHORTESTPATH
11:     **while** H.topKey() < calculateKey(v_{start}) **or** $rhs(v_{\text{start}}) > g(v_{\text{start}})$ **do**
12:         $\{v, k_{\text{old}}\} \leftarrow H$.top(), H.topKey()
13:         $k_{\text{new}} \leftarrow$ calculateKey(v)
14:         **if** $k_{\text{old}} < k_{\text{new}}$ **then** ▷ Underestimate
15:             H.update(v, k_{new})
16:         **else if** $g(v) > rhs(v)$ **then** ▷ Locally Overconsistent
17:             $g(v) \leftarrow rhs(v)$
18:             H.remove(v)
19:             **for all** $u \in$ NEIGH(v) **do**
20:                 **if** $u \neq v_{\text{goal}}$ **then**
21:                     $rhs(u) \leftarrow \min(rhs(u), c(u, v) + g(v))$
22:                 updateVertex(u)
23:         **else** ▷ Locally Underconsistent

24: oldG $\leftarrow g(v)$
25: $g(v) \leftarrow \infty$
26: **for all** $u \in \text{pred}(v) \cup \{v\}$ **do**
27: **if** $rhs(u) = c(u, v) + \text{oldG}$ **then**
28: $rhs(u) \leftarrow \text{computeRhs}(u)$
29: updateVertex(u)
30: **procedure** MAIN
31: $v_{\text{last}} \leftarrow v_{\text{start}}$
32: initialize()
33: computeShortestPath()
34: **while** $v_{\text{start}} \neq v_{\text{goal}}$ **do**
35: $v_{\text{start}} \leftarrow \arg\min_{v' \in \text{NEIGH}(v_{\text{start}})}(c(v_{\text{start}}, v') + g(v'))$
36: Move to v_{start}
37: **if** edge costs changed **then**
38: $k_m \leftarrow k_m + h(v_{\text{last}}, v_{\text{start}})$
39: $v_{\text{last}} \leftarrow v_{\text{start}}$
40: **for all** (u, v) with changed edge costs **do**
41: $c_{\text{old}} \leftarrow c(u, v)$
42: Update $c(u, v)$
43: **if** $c_{\text{old}} > c(u, v)$ **then**
44: **if** $u \neq v_{\text{goal}}$ **then**
45: $rhs(u) \leftarrow \min(rhs(u), c(u, v) + g(v))$
46: **else if** $rhs(u) = c_{\text{old}} + g(v)$ **then**
47: **if** $u \neq v_{\text{goal}}$ **then**
48: $rhs(u) \leftarrow \min_{u' \in \text{NEIGH}(u)}(c(u, u') + g(u'))$
49: updateVertex(u)
50: computeShortestPath()

4.4.1 Explanation of computeShortestPath()

When a vertex v is popped from the priority queue, there are three cases:

1. $k(v)$ is an underestimate of the true priority. Therefore, the vertex should be reinserted into the priority queue with its corrected key (line 15).

2. v is locally overconsistent meaning $g(v) > rhs(v)$. Simply, update g(v) with $rhs(v)$ to make v locally consistent (lines 16). Since changing g(v) can affect the rhs values of its neighbors, fix the neighbors' rhs values to reflect the updated g(v) (lines 20-21).

3. v is locally underconsistent meaning $g(v) < rhs(v)$ (lines 23). Case 3 will only happen when v becomes intraversable. Since $g(v) < rhs(v)$, g(v) is too optimistic. Therefore, g(v) is reset to ensure the recomputation of g(v), that is, to make v locally overconsistent (line 25). Since $g_{\text{old}}(v)$ is outdated, the rhs values of its neighbors may be wrong. Thus, for each of the neighboring vertices, we ask if $g_{\text{old}}(v)$ was used to calculate rhs (line 27). If so, fix rhs using its definition (lines 27-28).

4.4.2 Explanation of main()

While the agent is not at v_{goal}, the agent moves greedily toward v_{goal} (line 35). If traversability changes, we use the heap ordering trick to account for new priorities that must be computed (lines 38-39). Some edge costs can change, indicating changes in traversability. Therefore, when some edge costs change, the *rhs* values should be updated to reflect these changes. Changes in the *rhs* values mean that new priorities must be calculated. Thus, the offset value k_m should also be updated before calling updateVertex(v) (line 49). For each changed edge costs, there are two cases. If the edge cost decreased,

4.5 Practical Implementation

One question that may arise is how the heap should be implemented since the heap must support insert, update, and delete operations, which are uncommon. When implementing the min heap, to make the min heap more efficient, use a heap/map, where the map maps vertices to indices in the heap. This approach allows quick logarithmic time for insertion, deletion, and update instead of linear time.

The main trick using a heap/map is to find the indices of the target vertex and swap (if needed). In the case of deletion, swap the vertex to be deleted with the last vertex in the heap. Now, deleting the vertex to be deleted is the last element in the H, and, therefore, easy to delete. However, the vertex that was swapped may violate the heap property. Thus, call siftUp *and* siftDown on the swapped vertex to fix the heap, which only takes logarithmic time.

If the edge costs are used strictly for classifying traversability, we could optimize lines 40-49 in the main function, since the graph is undirected and traversability determines the values of edge costs. If a vertex v becomes traversable, update $rhs(v)$ using its definition. If a vertex v becomes intraversable, update $rhs(v) = \infty$. Afterwards, call updateVertex(v) to address changes in the rhs(s) costs.

procedure UPDATEVERTEX(v, traversability)
    if v.traversability = traversability **then return**
    $k_m \leftarrow k_m + h(v_{\text{last}}, v_{\text{start}})$
    $v_{\text{last}} \leftarrow v_{\text{start}}$
    if traversability **then** ▷ v is now traversable
        $v \leftarrow$ computeRhs(v)
    !traversability ▷ v is not traversable
    $v \leftarrow \infty$

    updateVertex(v)