A D*-Lite Variant Incorporating Risk Awareness for Dynamic Pursuit–Evasion Scenario on Grid Graphs

Muhammad Akmal – 13524099
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha No. 10 Bandung
muhammad.akmal.3806@gmail.com, 13524099@std.stei.itb.ac.id

***Abstract*—In a pursuit–evasion scenario, path planning has always been the main challenge for both the evader and the pursuer. By modelling the environment in a grid-based graph, the problem can be simplified into find the shortest path dynamically for both agents to achieve their goals. Various algorithms have been combined to simulate how the agents move and interact, each with its own heuristic and trade-off. This paper proposes a novel point of view in designing a dynamic path planning algorithm for the evader, incorporating risk awareness to avoid nearby chasers while also maintaining the shortest path to the goal. Implemented on a 64×64 grid and evaluated across nine penalty scales and eleven avoidance radii, this method increases the evader's success rate from approximately 25 percent (with vanilla D* Lite) to over 75 percent when the avoidance radius is at least three cells, with peak performance reaching 100 percent success at a radius of eight cells. These gains incur a modest path-cost increase of up to 45 percent and planning times around 500 milliseconds per step in Python. This variant retains all the efficiency advantages of D* Lite while substantially improving survivability in adversarial environments. On the broader field, the author hope these findings could be utilized in various scenarios, including UAV, robotics, game AI, and other applications for the greater good.**

***Index Terms*—D* Lite, dynamic path planning, incremental search, real-time replanning, risk awareness.**

I. Introduction

Dynamic path planning involves finding optimal or safe routes when the environment or objectives change over time. In robotics and navigation, algorithms like A* and its incremental variants: LPA* and D*-Lite, are widely used for real-time replanning. The A* algorithm is a standard heuristic search that finds shortest paths using admissible heuristics (e.g. Manhattan or Chebyshev distance for grid graph). However, A* assumes a static environment and must re-plan from scratch if the world changes [1]. Lifelong Planning A* (LPA*) is an incremental form of A* that reuses previous search effort when edge costs change [2]. D* Lite, introduced by Koenig and Likhachev [3], builds on LPA* to enable efficient replanning in unknown or changing terrain. Koenig and Likhachev note that "incremental heuristic search methods use heuristics to focus their search and reuse information from previous searches to find solutions ... faster than solving each task from scratch" [2]. D* Lite "implements the same behavior as Stentz' Focussed Dynamic A*" but with a simpler implementation [3]. These algorithms have proven effective in robotic navigation and have applications in military strategy: for instance, missile guidance systems use similar replanning methods to evade interceptors in dynamic engagement scenarios.

In adversarial pursuit–evasion scenario, an evader (target) must reach a goal while avoiding a pursuer (chaser). Incorporating *risk awareness* into the evader's planning can significantly improve survivability: the evader should prefer paths that both reach the goal and maintain distance from the pursuer. Risk aware planning has been studied in contexts like UAV navigation, where "dynamic path planning for UAVs ... takes into account static and dynamic threats" using risk maps [4]. Generally, risk-aware planners minimize both travel cost and exposure to danger (e.g. threats, obstacles, or adversaries).

This paper proposes a variant of the D*-Lite Algorithm that incorporates risk awareness for the evader in a pursuit–evasion scenario. The evader's motion is planned on a discrete grid using an incremental D*-Lite search that dynamically incorporates a risk penalty based on proximity to the chaser. The author chose to focus on the evader: it must reach its goal while maximizing separation from the chaser. The contributions include (a) a formal model of grid-based pursuit–evasion, (b) a D*-Lite variant that adds a distance-based risk cost to the evader's search, and (c) an empirical evaluation (in Python) measuring path cost, planning runtime, and success rate.

II. Theoretical Basis

A. Graph Theory and Terminology

A *graph* $G = (V, E)$ is formally defined by a vertex set V and an edge set $E \subseteq [V]^2$ where each edge is a 2-element subset of V [5]. Here, $[V]^2$ denotes all unordered pairs of vertices. Graphs can be **undirected** (edges have no orientation) or **directed** (each edge is an ordered pair of vertices). For example, in an undirected graph, an edge $\{u, v\}$ connects u and v symmetrically, whereas in a directed graph, an edge is written (u, v) meaning "from u to v" [5]. Graphs may also be **weighted**, meaning each edge (u, v) carries a numerical cost $w(u, v)$. Graphs that may have multiple edges connecting the same vertices are called **multigraphs**, [6]. When there are m different edges associated to the same unordered pair of vertices $\{u, v\}$, we also say that $\{u, v\}$ is an edge of multiplicity m. That is, we can think of this set of edges as m different copies of an edge $\{u, v\}$.

Several graph terminologies heavily used in this paper includes adjacency, incidency, and degree. Two vertices u and

v in an undirected graph G are called **adjacent** (or neighbors) in G if u and v are endpoints of an edge e of G. Such an edge e is called **incident** with the vertices u and v and e is said to connect u and v [6]. The set of all neighbors of a vertex v of $G = (V, E)$, denoted by $N(v)$, is called the **neighborhood** of v. If A is a subset of V, we denote by $N(A)$ the set of all vertices in G that are adjacent to at least one vertex in A [6]. So,

$$N(A) = \bigcup_{v \in A} N(v).$$

The degree of a vertex in an undirected graph is the number of edges incident with it, except that a loop at a vertex contributes twice to the degree of that vertex [6]. The degree of the vertex v is denoted by $deg(v)$.

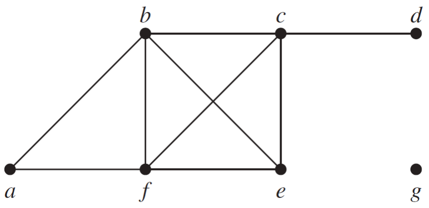

Fig. 1. A simple undirected graph with 7 vertices and 9 edges. (Source: [6])

Let's look at Fig. 1 to illustrate the definitons. The neighborhoods of vertices are $N(a) = \{b, f\}$, $N(b) = \{a, c, e, f\}$,$N(c) = \{b, d, e, f\}$, $N(d) = \{c\}$, $N(f) = \{a, b, c, e\}$, and $N(g) = \emptyset$. Since $a \in N(b)$ and $b \in N(a)$, then meaning a and b are adjacent. The degree of vertices are $deg(a) = 2$, $deg(b) = deg(c) = deg(f) = 4$, $deg(d) = 1$, $deg(e) = 3$, and $deg(g) = 0$. Therefore, it can be concluded that for a simple undirected graph, $deg(v) = |N(v)|$.

B. Graph Representation

In algorithms, we represent $G = (V, E)$ via either **adjacency lists** or an **adjacency matrix** [7]. In the adjacency-list representation, we have an array (or map) `Adj` of length $|V|$, where `Adj[u]` is a list of all vertices v such that $(u, v) \in E$ (for directed graphs) or $\{u, v\} \in E$ (undirected) [7]. Thus, `Adj[u]` stores pointers or references to u's neighbors. Fig. 1 could be represented by lists with mapping the letter label to number (e.g. $a = 1$), `Adj[1]={2,6}`, `Adj[2]={1,3,5,6}`, etc. In the adjacency-matrix representation, we index vertices from 1 to $|V|$ and use a $|V| \times |V|$ matrix $A = (a_{ij})$ with

$$a_{ij} = \begin{cases} 1 & \text{if } (i, j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

For undirected graphs, A is symmetric. In a weighted graph, we store the weight $w(i, j)$ instead of 1 (or in place of 0/1). Adjacency lists are space-efficient for sparse graphs ($O(V + E)$ memory), while an adjacency matrix uses $O(V^2)$ space [7]. Both representations readily extend to weighted graphs by storing each edge's weight in the list or matrix entry.

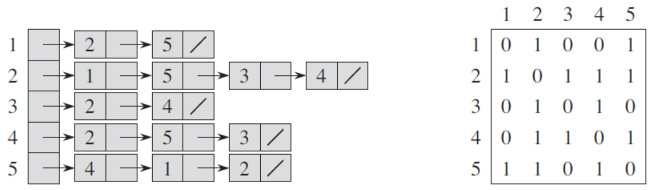

Fig. 2. Standard graph representations for a simple graph. Adjacency-list form (left); adjacency-matrix form (right). (Source: [7])

C. Grid Connectivity and Chebyshev Distance

In grid-based path planning, we often use an **8-connected** grid graph where each cell is connected (adjacent) to its 8 neighbors (including diagonal). The cost (weight) of moving between two adjacent cells (x, y) and (x', y') is given by the **Chebyshev distance**:

$$d_{cheb}\big((x, y), (x', y')\big) \;=\; \max\big(|x - x'|,\; |y - y'|\big). \quad (1)$$

Thus a diagonal move $|x - x'| = |y - y'| = 1$ has cost 1 (same as a horizontal or vertical move). This corresponds to the minimum number of king's moves on a chessboard. In effect, an 8-connected grid with unit Chebyshev weights makes all immediate moves cost 1.

D. Path Planning Algorithm

To compute shortest paths in a graph, classic algorithms follow a similar pattern of exploring nodes and relaxing edge costs. Below, the author outline three key methods: Breadth-First Search (for unweighted paths), Dijkstra's algorithm (for weighted graphs), and A* search (for weighted graphs with heuristics). This part describe their general steps, pseudocode, and relevant properties, citing standard algorithm texts.

1) Breadth First Search: BFS finds shortest paths (in edge-count) from a node source s in an unweighted graph. It uses a FIFO (first in, first out) queue to explore vertices in order of increasing distance. Initially, all vertices are marked "white" (undiscovered) and assigned distance $d[v] = \infty$; the source s is colored "gray", $d[s] = 0$, and enqueued. Then BFS repeatedly dequeues a vertex u, and for each neighbor $v \in \text{Adj}[u]$ that is still white, it "discovers" v by coloring it gray, setting $d[v] = d[u]+1$, recording u as v's predecessor, and enqueuing v. Once all neighbors are processed, u is colored "black" to mark it finished. This level-by-level exploration guarantees $d[v]$ is the minimum number of edges from s to v. In fact, BFS runs in $O(|V| + |E|)$ time by scanning each adjacency list once [8], as can be seen in Algorithm 1.

2) Djikstra's Algorithm: For graphs with non-negative edge weights, Dijkstra's algorithm finds the shortest-path distances from a source s. It maintains a set S of vertices whose final distances are known, and a min-priority queue Q (typically a binary heap) of the remaining vertices keyed by their current distance estimate $d[u]$. Initially all $d[v] = \infty$ except $d[s] = 0$. Then in each iteration, the algorithm extracts the vertex u with minimum $d[u]$ from Q, adds u to S, and "relaxes" each edge (u, v) out of u: if $d[u] + w(u, v) < d[v]$, it updates

Algorithm 1 Breadth-First Search (BFS)

```
Require: Graph $G = (V, E)$, starting vertex $s \in V$
Ensure: Distances $d[v]$ and parents parent$[v]$ for all $v \in V$
 1: for all $u \in V$ do
 2:     color$[u] \leftarrow$ WHITE
 3:     $d[u] \leftarrow \infty$
 4:     parent$[u] \leftarrow$ NIL
 5: end for
 6: color$[s] \leftarrow$ GRAY
 7: $d[s] \leftarrow 0$
 8: parent$[s] \leftarrow$ NIL
 9: $Q \leftarrow$ empty queue
10: ENQUEUE$(Q, s)$
11: while $Q \neq \emptyset$ do
12:     $u \leftarrow$ DEQUEUE$(Q)$
13:     for all $v \in$ Adj$[u]$ do
14:         if color$[v]$ = WHITE then
15:             color$[v] \leftarrow$ GRAY
16:             $d[v] \leftarrow d[u] + 1$
17:             parent$[v] \leftarrow u$
18:             ENQUEUE$(Q, v)$
19:         end if
20:     end for
21:     color$[u] \leftarrow$ BLACK
22: end while
```

Algorithm 2 Dijkstra's Algorithm

```
Require: Graph $G = (V, E)$ with non-negative edge weights
    $w(u, v)$, source vertex $s \in V$
Ensure: Shortest path distances $d[v]$ and parent pointers
    parent$[v]$ for all $v \in V$
 1: for all $u \in V$ do
 2:     $d[u] \leftarrow \infty$
 3:     parent$[u] \leftarrow$ NIL
 4: end for
 5: $d[s] \leftarrow 0$
 6: $Q \leftarrow$ priority queue containing all $u \in V$ with key $d[u]$
 7: while $Q \neq \emptyset$ do
 8:     $u \leftarrow$ EXTRACT-MIN$(Q)$
 9:     for all $v \in$ Adj$[u]$ do
10:         if $d[u] + w(u, v) < d[v]$ then
11:             $d[v] \leftarrow d[u] + w(u, v)$
12:             parent$[v] \leftarrow u$
13:             DECREASE-KEY$(Q, v, d[v])$
14:         end if
15:     end for
16: end while
```

$d[v] = d[u] + w(u, v)$ and sets u as v's predecessor. Cormen, Leiseron, Rivest, and Stein [7] describe this as "repeatedly select the vertex $u \in V - S$ with minimum shortest-path estimate, add u to S, and relax all edges leaving u". When the queue is empty, Dijkstra's algorithm has $d[v] = \delta(s, v)$ (the true shortest distance) for every v, provided all weights are non-negative. With a binary-heap implementation, the time is $O((|V| + |E|) \log |V|)$. See Algorithm 2 for implementation of the algorithm using priority queue.

3) A Search:* A* (pronounced as A-star) is a best-first search algorithm that uses a heuristic to guide the search towards a goal node t. It generalizes Uniform-Cost Search by using a priority function

$$f(n) = g(n) + h(n) \tag{2}$$

instead of just $g(n)$, where $g(n)$ is the cost from the start to n and $h(n)$ is an estimate of the cost from n to the goal [8]. In *graph-search* form, A* maintains an *open set* (priority queue) of frontier nodes ordered by f; each iteration pops the node n with smallest $f(n)$. If n is the goal, the shortest path has been found. Otherwise, n is expanded and its neighbors are "relaxed" in the same way as Dijkstra's algorithm, except using f-values. When a neighbor m is discovered or improved, we set $g(m)$, compute $f(m) = g(m) + h(m)$, and insert (or update) m in the open set. If the heuristic h is **admissible** (never overestimates the true cost to the goal), A* is guaranteed to find an optimal path. As noted by Russell and Norvig, the A* pseudocode (shown in Algorithm 3) is identical to Uniform-Cost Search except for the priority computation: use $f(n) = g(n) + h(n)$ instead of $g(n)$ [8].

E. Incremental Search Algorithm

1) LPA Algorithm:* Lifelong Planning A* (LPA*) extends A* to dynamic graphs [1], [2]. It maintains for each vertex v a pair of values $(g(v), rhs(v))$, where g is the current best-known distance from the start, and rhs is a one-step lookahead cost (like a local target). The rhs-values are computed by (3), where $s \in S$ representing the finite set of vertices of the graph, and $pred(s)$ means the predecessor of vertex s [2]; while $c(s', s)$ is the cost (weight) to get from s' to s.

$$rhs(s) = \begin{cases} 0 & \text{if } s = s_{start}, \\ min_{s' \in pred(s)}(g(s') + c(s', s)) & \text{otherwise.} \end{cases} \tag{3}$$

Initially, LPA* runs a standard A* to compute a path. When an edge cost changes (due to an obstacle added or removed), only affected vertices have their and updated, and the search re-expands necessary nodes. The key idea is that LPA* "reuses those parts of the previous search tree that are identical to the new one" [1], [2], so replanning is faster than planning from scratch if changes are small or near the goal. LPA* repeatedly finds shortest paths from the start to the goal while edge costs change or vertices are added or deleted.

2) D Algorithm:* D* (Dynamic A*) is a similar incremental planner originally designed for mobile robots in unknown or changing terrain [9]. D* computes an initial optimal path from the start to goal, then when new information (e.g. an obstacle) is discovered, it efficiently repairs the existing path instead of replanning entirely. Stentz showed that "D* is far more efficient than the brute-force path planner" and guarantees an optimal traverse as costs change [9]. D* repeatedly

Algorithm 3 A* Search Algorithm

```
Require: Graph $G = (V, E)$, start vertex $start$, goal vertex $goal$, heuristic function $h : V \to \mathbb{R}$
Ensure: Path from $start$ to $goal$ minimizing $g(n) + h(n)$
1: for all $u \in V$ do
2:     $g[u] \leftarrow \infty$
3:     parent$[u] \leftarrow$ NIL
4: end for
5: $g[start] \leftarrow 0$
6: $f[start] \leftarrow h(start)$
7: $open\_set \leftarrow$ priority queue containing $start$, keyed by $f[\,]$
8: while $open\_set \neq \emptyset$ do
9:     $n \leftarrow$ EXTRACT-MIN$(open\_set)$    ▷ node with lowest $f$
10:    if $n = goal$ then
11:        return construct path from $start$ to $goal$ via parent[ ]
12:    end if
13:    for all $m \in$ Adj$[n]$ do
14:        $cost \leftarrow g[n] + w(n, m)$
15:        if $cost < g[m]$ then
16:            $g[m] \leftarrow cost$
17:            parent$[m] \leftarrow n$
18:            $f[m] \leftarrow g[m] + h(m)$
19:            if $m \notin open\_set$ then
20:                INSERT$(open\_set, m, \text{key} = f[m])$
21:            end if
22:        end if
23:    end for
24: end while
```

propagates cost changes from the changed edge back through the open list of vertices, effectively performing a focused incremental update near the change. This makes D* well-suited to robotic navigation where new obstacles are sensed during execution.

3) D Lite Algorithm:* D* Lite simplifies D* by leveraging LPA*. Koenig and Likhachev proved that D* Lite "implements the same behavior as Stentz' Focused Dynamic A* but is algorithmically different" [3]. In practice, D* Lite runs LPA* on a graph where the roles of start and goal are exchanged. This method allows replanning from the current robot position back to the (fixed) goal, as can be seen in Algorithm 4. When an edge cost changes, only local consistency is checked. As the authors note, when the start moves, the heuristic values decrease by a constant offset and all keys can be updated by that offset, avoiding a full priority-queue rebuild. The result is an algorithm that can replan very quickly when unexpected obstacles appear, making it effective in cluttered and dynamic environments. The main algorithm can be seen in Algorithm 5.

Algorithm 4 Compute Shortest-Path Procedure for D* Lite

```
1: procedure CALCULATEKEY($s$)
2:     return $[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m;\ \min(g(s), rhs(s))]$
3: end procedure
4: procedure UPDATEVERTEX($u$)
5:     if $u \neq s_{goal}$ then
6:         $rhs(u) \leftarrow \min_{s' \in Succ(u)}(c(u, s') + g(s'))$
7:     end if
8:     if $u \in U$ then
9:         $U$.Remove$(u)$
10:    end if
11:    if $g(u) \neq rhs(u)$ then
12:        $U$.Insert$(u, \text{CalculateKey}(u))$
13:    end if
14: end procedure
15: procedure COMPUTESHORTESTPATH
16:    while $U$.TopKey() $<$ CalculateKey$(s_{start})$ or $rhs(s_{start}) \neq g(s_{start})$ do
17:        $k_{old} \leftarrow U$.TopKey()
18:        $u \leftarrow U$.Pop()
19:        if $k_{old} <$ CalculateKey$(u)$ then
20:            $U$.Insert$(u, \text{CalculateKey}(u))$
21:        else if $g(u) > rhs(u)$ then
22:            $g(u) \leftarrow rhs(u)$
23:            for all $s \in Pred(u)$ do
24:                UPDATEVERTEX($s$)
25:            end for
26:        else
27:            $g(u) \leftarrow \infty$
28:            for all $s \in Pred(u) \cup \{u\}$ do
29:                UPDATEVERTEX($s$)
30:            end for
31:        end if
32:    end while
33: end procedure
```

III. Design of Algorithm

A. Environment Model

The environment the agents live is defined as a 2D grid of cells, represented by a matrix

$$A = (a_{ij})_{0 \leq i < m,\ 0 \leq j < n},$$

Then, two cells $p = a_{i_1 j_1}$ and $p = a_{i_2 j_2}$ can be defined to be adjacent if and only if

$$max(|i_1 - i_2|, |j_1 - j_2|) = 1.$$

Then, the author model the environment as a grid-based graph, defined as a pair $G = (V, E)$ of sets satisfying $E \subseteq [V]^2$ where each vertex corresponds to a cell in the 2D grid and edges $e = (u, v) \in E$ connect adjacent cells, meaning an eight-connected neighborhood. In an obstacle-free "open" field, all edges are present except those going off the grid. A base weight $w(u, v)$ is assigned to each edge, representing

Algorithm 5 Main Procedure of D* Lite Algorithm

```
1: procedure INITIALIZE
2:     $U \leftarrow \emptyset$
3:     $k_m \leftarrow 0$
4:     for all $s \in S$ do
5:         $rhs(s) \leftarrow g(s) \leftarrow \infty$
6:     end for
7:     $rhs(s_{goal}) \leftarrow 0$
8:     $U$.Insert($s_{goal}$, CalculateKey($s_{goal}$))
9: end procedure
10: procedure MAIN
11:     $s_{last} \leftarrow s_{start}$
12:     INITIALIZE
13:     COMPUTESHORTESTPATH
14:     while $s_{start} \neq s_{goal}$ do          ▷ If $g(s_{start}) = \infty$, no known path
15:         $s_{start} \leftarrow \arg\min_{s' \in Succ(s_{start})}(c(s_{start}, s') + g(s'))$
16:         Move to $s_{start}$
17:         Scan for changed edge costs
18:         if any edge cost changed then
19:             $k_m \leftarrow k_m + h(s_{last}, s_{start})$
20:             $s_{last} \leftarrow s_{start}$
21:             for all directed edges $(u, v)$ with changed edge costs do
22:                 Update edge cost $c(u, v)$
23:                 UPDATEVERTEX($u$)
24:                 UPDATEVERTEX($v$)
25:             end for
26:             COMPUTESHORTESTPATH
27:         end if
28:     end while
29: end procedure
```

the travel cost. With equal-cost movement, one can set for $w(u, v) = 1$ orthogonal neighbors, which corresponds to the Chebyshev distance metric as seen in (1).

To simulate the dynamic condition of the map, the grid weight can be increased around an area for indicating a "ground elevation". This condition will realistically simulate real-world scenario with uneven terrain in ground path-planning. For the wall, fully increasing the weight up to infinity instead of deleting the edges connecting both cell. This model could be represented compactly by assigning a value C_v to each cell (vertex) v then compute the difference instead of storing it in the edge. With the model in mind, it can be generalized that is for every adjacent two vertices $u, v \in V$, the weight of edge $e = (u, v)$ is

$$w(u, v) = |C_u - C_v|. \tag{4}$$

The author will use Python's library Pygame to display the environment as a 2D Gridmap with grayscale gradient indicating the value assigned to the cells, where the darker cell correspond to greater value or "higher elevation". White and black cells correspond to normal cell with weight 1 and a

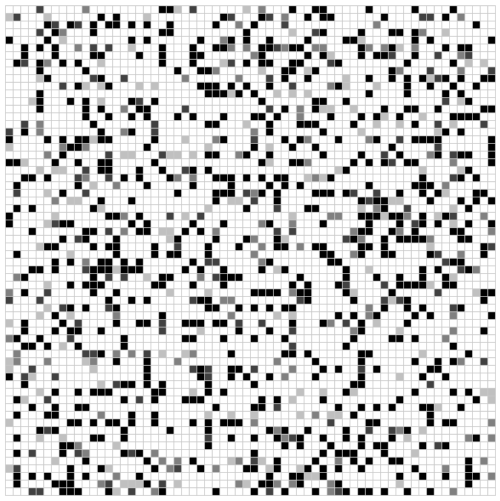

Fig. 3. Initially generated 64×64 grid map with grayscale marks.

wall with infinite cost, repectively. The map is initialized with the 64×64 grid cells and each cell filled with random number to simulate the initial condition of the map. The result of the generated map can be seen in Fig. 3. The full simulation can be acessed in Appendix.

B. *Evader's Initial Algorithm*

The author will implement the vanilla D* Lite algorithm as the evader's initial path planning algorithm. The main advantage of D* Lite is that it can efficiently replan in dynamic environments by incrementally updating the shortest path as the agent progresses and the map changes.

D* Lite is initialized with the goal as the anchor point and uses the Chebyshev distance as the consistent heuristic function, defined as

$$h(u, v) = \max\left(|i_u - i_v|,\ |j_u - j_v|\right), \tag{5}$$

where (i_u, j_u) and (i_v, j_v) denote the coordinates of nodes u and v respectively.

At each time step t, the algorithm does the following:

1) Updates edge weights if the environment has changed.
2) Performs consistency check on affected nodes.
3) Replans the shortest path from the current position to the goal.
4) Moves one step along the path.

This algorithm only takes into account environmental cost changes but assumes no knowledge of a dynamic pursuer. Thus, it is susceptible to being intercepted by the pursuer, particularly in open environments without obstacles.

C. *Evader's Variant Algorithm: Risk-Aware D* Lite*

To enhance the survivability of the evader, this paper introduce a variant of D* Lite that incorporates a dynamic risk penalty into the path cost function. The key idea is to bias the evader's path away from the pursuer, based on their real-time location, while still maintaining an efficient route to the goal.

1) Risk Penalty Function: Let p_t denote the current position of the pursuer at time t. Define a time-varying risk penalty function $\rho_t(v)$ for any cell $v \in V$ as:

$$\rho_t(v) = \lambda \cdot d_{cheb}(e_t, g) \cdot \max\{0,\ r - d_{\text{cheb}}(v, p_t)\}, \quad (6)$$

where d_{cheb} is the Chebyshev distance between cell v and the pursuer p_t as defined in (1), r is the risk radius threshold, and λ is a penalty scaling factor. The $d_{cheb}(e_t, g)$ is used to decrease the risk when the evader e_t is near the goal g, encouraging the evader to get to ignoring the pursuer when its almost reached the goal.

This function adds a repulsive cost around the pursuer, with higher penalties for cells closer than r to the pursuer. This transforms the map into a risk-augmented terrain that discourages paths passing near the chaser.

2) Weight Update Rule: Modify the edge weight computation to include the risk penalty as follows:

$$w'_t(u, v) = |C_u - C_v| + \rho_t(v), \quad (7)$$

where $|C_u - C_v|$ accounts for the terrain elevation difference, and $\rho_t(v)$ penalizes proximity to the pursuer.

3) Algorithmic Procedure: At each time step t, the evader executes the following Risk-Aware D* Lite procedure:

1) Observe the current position of the pursuer p_t.
2) For all $v \in V$ such that $d_{\text{cheb}}(v, p_t) \leq r$, compute $\rho_t(v)$ using (6).
3) Update edge weights to use $w'_t(u, v)$ as in (7).
4) Mark affected nodes as inconsistent and enqueue them for priority queue update.
5) Execute the `ComputeShortestPath` function of D* Lite to replan.
6) Move one step along the computed path.

The pseudocode implementation of this routine is provided in Algorithm 6.

Algorithm 6 Risk-Aware D* Lite Path Planning

```
Require: Initial state $s$, goal state $g$, risk threshold $r$, penalty factor $\lambda$
1:  INITIALIZE    ▷ Initialize D* Lite with $s$, $g$, base costs $w(u,v)$
2:  while $s \neq g$ and $s \neq p_t$ do
3:      Observe pursuer position $p_t$
4:      for all $v \in V$ such that $d_{\text{cheb}}(v, p_t) \leq r$ do
5:          for all neighbors $u$ of $v$ do
6:              Update $w'_t(u,v) \leftarrow |C_u - C_v| + \lambda \cdot \max(0, r - d_{\text{cheb}}(v, p_t))$
7:          end for
8:      end for
9:      COMPUTESHORTESTPATH
10:     Move $s$ to the next step on the new path
11: end while
```

Remarks: This variant encourages the evader to avoid capture zones dynamically, producing more intelligent paths that consider both elevation and threat. It remains computationally efficient due to D* Lite's incremental nature, even as risk values change frequently.

D. Pursuer's Algorithm

The pursuer's will use A* Algorithm as shown in Algorithm 3 with a modification on choosing the target cell. For simplicity, it is assumed that the agent know the position of both the evader and its goal. When the target is afar (the distance is large enough), the algorithm will try to "intercept" the evader by setting a "middle point" between the evader's current position and the goal cell position and try to move there as fast as possible. Otherwise, the Chebyshev Distance between the evader and pursuer is close enough (i.e. $d_{cheb}(p_t, e_t) < R$ for some radius R) , so the pursuer will use the original A* algorithm to "chase" the evader. We will use the value $R = r + 1$ for the evader's algorithm by reusing r previously defined in section III-C1. This decision is made to simplify the model and surpress the number of variables used.

More formally, the target cell for the pursuer's algorithm at a discrete time state t when the current distance between pursuer p_t and evader e_t is $d_{cheb}(p_t, e_t) \geq r$, is $d = a[i][j]$, where:

$$(i, j) = \left(\left\lfloor \frac{i_{\text{evader}}(t) + i_{\text{goal}}}{2} \right\rfloor, \left\lfloor \frac{j_{\text{evader}}(t) + j_{\text{goal}}}{2} \right\rfloor \right) \quad (8)$$

The behaviour of the agent is that for every discrete time step t, it will update the evader's posititon, then immediately recompute the shortest path and then move one step at a time. Sure, this behaviour is computationally expensive and not resembling a good incremental search algorithm. But it will be a good algorithm to oppose the original D* Lite Algorithm without the awareness of pursuer's posititon. This way, the evader cannot just "blindly" move to the goal, since it will likely that the pursuer may meet it on the way.

E. Game State

A game state is determined to be as follows. For every discrete time step t, the game will check if one of these conditions satisfied, in order:

1) **Lose**: The pursuer caught the evader when $d_{\text{cheb}}(e_t, p_t) \leq 1$, where e_t and p_t are the evader's and pursuer's position at t, respectively.
2) **Win**: The evader get to the goal cell without getting caught, when $d_{\text{cheb}}(e_t, g) \leq 1$, where e_t is the evader's position and g is the goal cell.
3) **Running**: Otherwise the game is running, both the evader and pursuer may choose to either move to any valid cell, or stay in the currently occupied cell.

Fig. 4 llustrate an example of each game state. For all three figures, the pursuer's currently occupied cell is highlighted with red, the evader's cell with green, and the goal cell with blue.

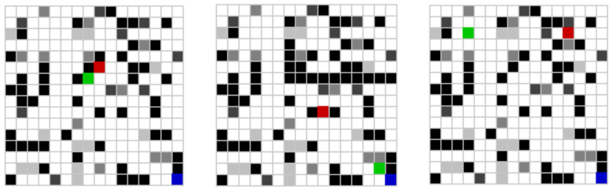

Fig. 4. Illustration of game state with different position for both agents. A "lose" condition (left); "win" condition (middle); "running" contidion (right).

IV. Result and Analysis

A. Experiment Setup

To evaluate the effectiveness of our Risk-Aware D* Lite algorithm, we conducted experiments on a 64×64 grid environment. Each trial was initialized with:

- Randomly assigned elevation values C_i for each cell from $[1, 10]$.
- Randomly placed walls with $C_i = \infty$.
- Evader and goal initialized at $(0, 0)$ and $(63, 63)$ respectively.
- Pursuer initialized at $(63, 0)$ to chase the evader.

Each experiment ran for up to 128 discrete time steps or until termination. We tested the evader's behavior under multiple risk-penalty settings:

- $\lambda = \{0.00, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0\}$
- $r = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

The algorithms were implemented in Python with visualization using Pygame. Each configuration was run over 4 different map. The athor used the following metrics to see how the algorithm performs under variable risk-penalty settings:

- **Success Rate**: Percentage of runs where the evader reached the goal without being caught.
- **Path Cost**: Sum of all weighted steps taken by the evader.
- **Computation Time**: Average planning time per step (in milliseconds).

B. Results and Interpretation

TABLE I
EVADER'S WIN RATE (%) FOR VARIOUS PENALTY AND RISK RADIUS

λ	Risk Radius r										
	0	1	2	3	4	5	6	7	8	9	10
0.00	25	25	25	75	50	50	75	75	75	75	75
0.25	25	25	25	100	50	75	75	75	100	50	50
0.50	25	25	25	75	50	50	75	50	100	50	50
0.75	25	25	25	75	50	50	75	50	100	50	50
1.00	25	25	25	75	50	50	75	50	100	25	50
1.25	25	25	25	75	50	50	75	50	100	25	50
1.50	25	25	25	75	50	50	75	50	100	25	50
1.75	25	25	25	75	50	50	75	50	100	25	50
2.00	25	25	25	75	50	50	75	50	100	25	50

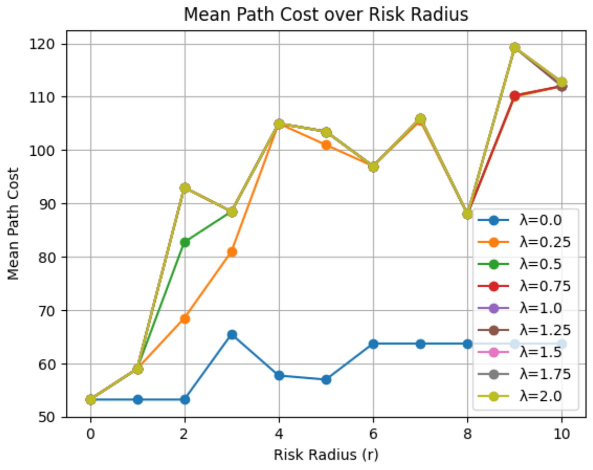

Fig. 5. Line chart of mean path cost over radius in various λ value.

From Table I, it is evident that the risk radius r shows a far stronger influence on the evader's success than the penalty scaling factor λ. When r is very small (≤ 2), the win-rate remains low (approximately 25%), regardless of the value of λ, indicating that a narrow avoidance zone fails to divert the pursuer. As r expands into the mid-range ($3 \leq r \leq 7$), the evader's success climbs sharply into the 50–75% band, demonstrating that a broader repulsive field provides substantially greater protection. The best success rate would be when $r = 8$, where the evader almost always reach the goal (100% win rate), with the exception when $\lambda = 0$ (i.e. normal D*-Lite). Beyond this, at large radii (> 8), reached-rates fall to 50-75%, and can plumbed to 25%, showing that further widening of the avoidance zone do not guaranteed success.

In contrast, increasing λ beyond a modest threshold (around 0.5) produces only marginal improvements in win-rate, curves for $\lambda = 0.5$, 1.0, and 2.0 are nearly indistinguishable. The reason the penalty's value λ does not affect the the success rate may be because it was too late for the evader to flee when the pursuer is near. Other reason proposed by the author is that the the value's range ($0 \leq \lambda \leq 2$) may not wide enough. Further experiment may debunked this assumptions, but with current set up, the conclusion is that the λ is not as impactful as r. These observations suggest a clear tuning strategy: first select a sufficiently large risk radius to secure the majority of benefits, and then choose a moderate penalty factor to fine-tune performance, since further increases in λ offer diminishing returns.

From only Table I, a conclusion may be made that it is better to increase r with low λ since it will guarantee the best survival rate. However, as shown in Fig. 5, this comes at a cost: path lengths increase due to wider detours. For instance, with $\lambda = 2.0$ and $r = 9$, total path costs averages to almost 120 weighted steps. As a saying "don't miss the forest for the trees", the algorithm should not prioritize evading

TABLE II
Mean Planning Time (milliseconds) for Each Penalty and Risk Radius

Penalty	**Risk Radius** r											**Mean**	**Std Dev**
λ	0	1	2	3	4	5	6	7	8	9	10		
0.00	567	578	577	588	624	625	573	581	586	584	587	588	18
0.25	642	605	595	576	587	597	563	591	582	597	585	593	19
0.50	591	580	544	568	598	589	572	520	521	530	581	563	28
0.75	586	587	588	523	528	529	576	597	567	529	548	560	27
1.00	517	504	539	575	562	538	518	513	505	592	596	542	33
1.25	546	511	529	515	495	544	547	571	543	549	535	535	20
1.50	489	549	573	539	524	510	491	507	547	596	557	535	33
1.75	512	525	500	490	552	561	534	541	505	530	523	525	21
2.00	551	600	540	512	526	505	496	556	570	549	557	542	29
Mean	556	560	554	543	555	555	541	553	547	562	563	**554**	
Std Dev	44	37	29	33	39	39	31	32	30	28	24	**34**	

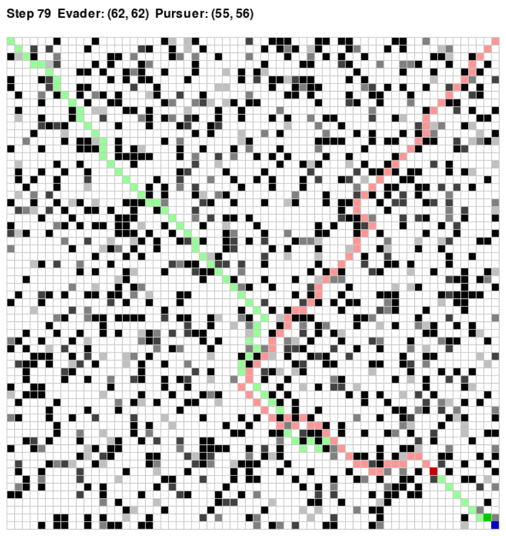

Fig. 6. The trace path of the evader (green) and pursuer (red) at $r = 8$, $\lambda = 1.25$, and $t = 79$.

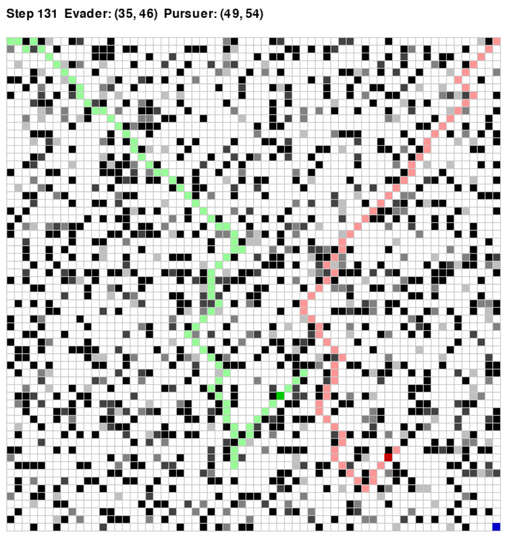

Fig. 7. The trace path between the evader (green) and pursuer (red) at $r = 9$, $\lambda = 1.75$, and $t = 131$.

from pursuer over its main objective: get to the goal cell with the shortest path possible. With this in mind, the author propose that the best configuration for current environment model would be $r = 8$ with any nonzero λ value. One of the best demonstration of such configuration can be seen in Fig. 6 with $\lambda = 1.25$ and only requires 79 steps.

The finding from Table I would implies that the evader get captured at large radii ($r > 8$). However, Fig. 5 may argues that it is not the case. Recall at section IV-A that we only run the simulation at 128 discrete steps. At $r = 9$ and $r = 10$, the mean path cost skyrockets to around 110-120 weighted steps, implies that may be the simulation terminates in "running" state where both agents still free to move. In fact, that is what actually happens when the author tries to run the simulation manually. After some time ($t > 80$), the pursuer acts like a "goal-keeper", preventing the evader to go to the goal cell. Since the evader "repels" the pursuer while the it is stand in its path, the resultant move would be a back-and-forth dance between the both agents at distance r. Fig. 7 shows the trace path between both agents. This phenomenon may happens because the "intercepting" behaviour we embed in the pursuer's AI back at section III-D utilize the similar variable as the evader ($R = r - 1$), making it almost always keep a stable distance between each other.

Table II shows that the computation time not differ much as λ and r changes, with the average standard deviation only 34 milliseconds. For a Python code, the author argues that it is still acceptable for offline simulation. However, optimizations may still need to be done to decrease the computation time in real-time planning. With around 500 milliseconds of compute time per steps, a real-time dynamic path planning would be extremely slow unless the algorithm is run sparsely. The author suggest that optimization could be done by switching to a compiled language like C++ or Java, reduce the number of recomputing path, or changes the data structure used in the algorithm. The current model uses a dictionary to store the risk penalty functions, which could be replaced by a matrix or 2D array to remove overhead time of hashing the key with dictionary.

C. Limitations and Future Work

Our model assumes both agent knows perfect knowledge of the other's location and moves with constant velocity. This model is far from the actual condition in dynamic pursuit-evasion scenarion where an agent may moves with different speeds and have limited knowledge of the suroundings. The quantitative result also did not incorporate a dynamic obstacle condition due to limited resource to simulate the condition. The author hopes that future work may incorporate uncertainty models with limited knowledge, dynamic velocity settings, and tests in dynamic obstacle environments. Additionally, integrating machine learning to tune λ and r adaptively during runtime remains an open direction.

V. Conclusion

This work introduced a *Risk-Aware D* Lite* algorithm, which integrates a dynamically computed penalty into the classic D* Lite framework to steer the evader away from a moving pursuer. In a grid-world experiments modelled via graph theory with 8-connected neighborhood with, the size of the avoidance region is the primary determinant of success: small radii yielded only 25 percent success, while radii of three to seven cells increased success rates to over 75 percent, a radius of eight cells achieved 100 percent win rate, and increasing the radius more only reduces the rate. In contrast, increasing the penalty scale beyond a moderate value offered only marginal improvements, indicating insufficient parameter space exploration due to range restriction. The evader secured near-optimal performance with risk radius eight and penalty scale between 0.5 and 1.5, balancing survivability against a path-cost increase of less than 50 percent. Planning times averaged around 500 milliseconds per step. It is suitable for offline analysis but highlighting the need for optimization in real-time applications.

Future research will explore the integration of *uncertainty* in pursuer position estimates, the extension of this approach to cluttered environments with dynamic obstacles and heterogeneous agent speeds, and the application of learning-based techniques for *adaptive* parameter tuning. The author believe Risk-Aware D* Lite offers a simple yet powerful enhancement for evader planning in robotics, unmanned aerial systems, and game AI.

Appendix

For further study, the author put a Python implementation of the algorithm designed below. Feel free to check out this repository and experiment more. To get more interactive explanation, the author also provide a YouTube video explaining the topics in Bahasa Indonesia in this link.

Acknowledgment

The author would like to express gratitude to Allah SWT for His blessings and guidance in completing this paper. Appreciation is also extended to Ir. Rinaldi, M.T., Lecturer of IF1220 Discrete Mathematics, for his dedication and the knowledge shared throughout the course. The author hopes this paper may serve as a foundation for further development beyond its initial submission.

References

[1] N. Sharma, C. Dharmatti, and J. E. Siegel, "A Survey of Path Planning Algorithms for Mobile Robots," Vehicles, vol. 3, no. 3, pp. 448–468, 2021.

[2] S. Koenig and M. Likhachev, "Lifelong Planning A* and Dynamic D* Lite: The proofs," Int. J. Robotics Res., vol. 20, pp. 405-425, 2005.

[3] S. Koenig and M. Likhachev, "D* Lite," in Proc. AAAI, 2002, pp. 476-483.

[4] G. Primatesta et al., "A Risk-Aware Path Planning Strategy for UAVs in Urban Environments," J. Int. Robot. Syst., vol. 88, no. 4, pp. 627-643, 2018.

[5] R. Diestel, *Graph Theory*, 6th ed., vol. 173, Graduate Texts in Mathematics. Berlin, Heidelberg: Springer, 2025. doi: 10.1007/978-3-662-70107-2.

[6] K. H. Rosen, Discrete Mathematics and Its Applications, 8th ed. New York, NY, USA: McGraw-Hill, 2024.

[7] T. H.Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms*, 4th ed. Cambridge, MA, USA: MIT Press, Apr. 2022.

[8] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Boston, MA, USA: Pearson, 2020.

[9] A. Stentz, "The D* Algorithm for Real-Time Planning of Optimal Traverses," Tech. Rep. CMU-RI-TR-94-37, Robotics Institute, Carnegie Mellon Univ., 1994.

Statement

I hereby declare that the paper I wrote is my own writing, not an adaptation or translation of someone else's paper, and is not plagiarized.

Bandung, June 23rd, 2025

[illegible]

Muhammad Akmal
13524099