Benchmarking of Priority Queue Implementations in the Dynamic D*Lite Algorithm

Milena Pellegrino[1]

[1] Universidad Nacional de Córdoba - Facultad de Matemática, Astronomía, Física y Computación, Córdoba, Argentina
milenapellegrino@mi.unc.edu.ar
ORCID: 0009-0001-5434-9238

Abstract. This work experimentally evaluates how the choice of priority queue affects D* Lite performance for path planning in dynamic graphs. We implemented a single algorithmic baseline and compared three data structures: Binary Heap, Fibonacci Heap, and Bucket Queue. To ensure statistical robustness, the evaluation comprised 10 independent repetitions per scenario, aggregating execution times and analyzing variance. The study covers 12 synthetic benchmarks on grids ranging from 10×10 to 500×500, addressing static maps, random obstacles, and dynamic updates. Additionally, we relate the structural mechanics of the queues to the algorithm's runtime profile (specifically the ratio of `DecreaseKey` to `PopMin` operations). Results show that Binary Heap provides the best practical performance, consistently outperforming Fibonacci Heap and Bucket Queue. Although Fibonacci Heap has better theoretical amortized bounds, the high frequency of `PopMin` operations on low-connectivity grids causes its constant overhead to dominate. While providing a solid baseline, our reliance on synthetic benchmarks and the lack of comprehensive memory profiling point to future opportunities for validating these findings on real-world datasets. Overall, Binary Heap currently stands as the most robust option for D* Lite in large-scale dynamic navigation tasks.

Keywords: D* Lite, priority queues, dynamic graphs, benchmarking, shortest path algorithms

Benchmarking de Implementaciones de Colas de Prioridad en el Algoritmo Dinámico D*Lite

Resumen. Este trabajo evalúa de forma experimental cómo la elección de la cola de prioridad en el algoritmo D* Lite impacta en su rendimiento para la planificación de rutas en grafos dinámicos. Se implementó una base algorítmica única, comparando tres estructuras: Binary Heap, Fibonacci Heap y Bucket Queue. Para brindar validez estadística a los resultados y mitigar el ruido del sistema, se ejecutaron 10 repeticiones independientes por escenario, reportando su desviación estándar. La evaluación empleó 12 benchmarks sintéticos sobre grillas desde 10×10 hasta 500×500 (incluyendo obstáculos, laberintos y cambios dinámicos), y se correlacionó las mediciones con la estructura interna de las operaciones (`DecreaseKey` y `PopMin`). Los hallazgos sitúan al Binary Heap como el paradigma de mejor desempeño práctico; si bien el Fibonacci Heap tiene cotas amortizadas teóricas superiores, la reestructuración ante continuas

extracciones penaliza excesivamente sus tiempos. Por otro lado, la Bucket Queue demostró ser competitiva sólo ante rangos acotados de prioridades. Queda asentado el requerimiento futuro de trasladar estos análisis a entornos del mundo real (no sintéticos) y de instrumentar un profiling de memoria profundo para consolidar de forma empírica las hipótesis de localidad espacial planteadas.

Palabras clave: D* Lite, colas de prioridad, grafos dinámicos, benchmarking, algoritmos de ruta más corta

1 Introducción

La planificación de rutas en grafos dinámicos es un problema fundamental en ciencias de la computación con aplicaciones en robótica móvil, navegación de vehículos autónomos, enrutamiento en redes, videojuegos, entre otros. A diferencia de los problemas clásicos de ruta más corta donde el grafo es estático (como en los algoritmos de Dijkstra (Dijkstra, 1959) o A* (Hart et al., 1968)), en escenarios dinámicos los pesos de las aristas pueden cambiar durante la ejecución, requiriendo una replanificación eficiente sin tener que recalcular todo desde cero.

El algoritmo D* Lite (Koenig & Likhachev, 2002) se ha establecido como uno de los métodos más eficientes y más utilizados actualmente para resolver este tipo de problemas, manteniéndose relevante en aplicaciones modernas como la robótica híbrida (Zhang et al., 2024). Combina las ventajas de la búsqueda heurística con técnicas incrementales que reutilizan información de búsquedas previas, logrando replanificaciones hasta órdenes de magnitud más rápidas que ejecutar A* repetidamente. Sin embargo, el rendimiento práctico de D* Lite depende críticamente de la implementación de su componente principal: la **cola de prioridad**.

1.1 Motivación

Una cola de prioridad es una estructura de datos en la que cada elemento tiene asociada una prioridad. Para extraer un elemento de la cola, siempre se elige el de menor (o mayor, dependiendo de la implementación) prioridad. En D* Lite, la cola de prioridad determina cuáles van a ser los nodos siguientes que se van a procesar, siendo esencial para la eficiencia del algoritmo.

Existen muchas implementaciones de colas de prioridad. Si bien el *Fibonacci Heap* (Fredman & Tarjan, 1987) es una de las más conocidas y representa la solución teórica óptima para D* Lite gracias a su eficiencia $O(1)$ en operaciones clave (inserción y `DecreaseKey`), posee una gran complejidad estructural en comparación con la simplicidad del *Binary Heap*.

No he encontrado estudios previos que comparen el rendimiento práctico de diferentes implementaciones de colas de prioridad específicas para D* Lite sobre grafos y grillas.

Por lo tanto, me pregunto:

¿Cómo afecta la elección de la estructura de datos para la cola de prioridad (Binary Heap, Fibonacci Heap, Bucket Queue) al rendimiento práctico del algoritmo D Lite, medido en tiempo de replanificación, bajo escenarios de actualización incremental en grafos masivos?*

Específicamente, nos interesa determinar:

1. Si las ventajas teóricas del *Fibonacci Heap* se materializan en la práctica para D* Lite.
2. Cómo escala el rendimiento de cada implementación con el tamaño del grafo.
3. Qué implementación es más eficiente bajo diferentes patrones de cambio en el grafo.

2 Fundamentos Teóricos

2.1 Colas de Prioridad

Como ya se mencionó, una cola de prioridad es una estructura de datos que contiene un conjunto de elementos, con la particularidad de que cada elemento tiene asignada una prioridad.

2.1.1 Definición y Operaciones Básicas

Formalmente, una cola de prioridad soporta las siguientes operaciones básicas:

- `Insert(elemento, clave)`: Inserta un elemento con una clave de prioridad asociada
- `FindMin()`: Retorna el elemento con la menor clave de prioridad sin eliminarlo
- `DeleteMin()`: Extrae y retorna el elemento con la menor clave de prioridad
- `DecreaseKey(elemento, nueva_clave)`: Disminuye la clave de prioridad de un elemento ya presente en la cola
- `Delete(elemento)`: Elimina un elemento específico de la cola

La eficiencia de estas operaciones varía según la implementación utilizada. En el contexto de algoritmos de búsqueda de caminos, las operaciones más frecuentes son `Insert`, `DeleteMin` y `DecreaseKey`, por lo que la elección de la estructura de datos tiene un impacto significativo en el rendimiento global del algoritmo.

2.1.2 Importancia en Algoritmos de Grafos

Las colas de prioridad son fundamentales en algoritmos de grafos como Dijkstra, A*, Prim, y en nuestro caso, D* Lite. En estos algoritmos, la cola de prioridad mantiene un conjunto de nodos "candidatos" ordenados por su distancia estimada o costo acumulado. El algoritmo procesa repetidamente el nodo de menor costo, actualiza los costos de sus vecinos, y potencialmente modifica sus prioridades en la cola.

La frecuencia y el patrón de estas operaciones determinan qué implementación de cola de prioridad será más eficiente en la práctica. Por ejemplo, en D* Lite, la operación `DecreaseKey` ocurre con frecuencia durante la fase de propagación de cambios en el grafo, lo que hace crucial su eficiencia.

El algoritmo de Dijkstra encuentra el camino más corto desde un nodo origen a todos los demás nodos (con la condición de que el peso de las aristas sea no negativo). Su complejidad es $O((|V|+|E|)\log|V|)$ usando una cola de prioridad eficiente. Sin embargo, si el peso de una arista cambia, Dijkstra debe ejecutarse nuevamente desde cero, lo cual es ineficiente en grafos con cambios dinámicos.

A* mejora sobre Dijkstra usando una heurística para guiar la búsqueda hacia el objetivo, pero tampoco maneja cambios dinámicos de una manera eficiente.

El algoritmo D* (Stentz, 1994) fue el primero en abordar este problema para navegación robótica, pero su implementación es compleja.

Aquí es donde entra en juego el algoritmo D* Lite (Koenig & Likhachev, 2002), el cual es un algoritmo incremental que combina las ideas de A* con búsqueda incremental. Fue diseñado específicamente para escenarios donde:

- El agente se mueve hacia un objetivo en un grafo inicialmente desconocido o parcialmente conocido
- Los pesos de las aristas pueden cambiar durante la ejecución
- Es necesario re-planificar eficientemente cuando se detectan cambios
- El agente no puede retroceder instantáneamente al punto de inicio

La ventaja clave de D* Lite sobre ejecutar A* repetidamente es que reutiliza gran parte de la información de la búsqueda anterior y solo recalcula las partes que cambiaron del grafo; por lo tanto, las replanificaciones son más rápidas en la práctica.

Se decidió utilizar el algoritmo D* Lite como base para los experimentos por ser más directo de implementar e integrar con nuestras versiones de las colas de prioridad. Era fundamental evitar la complejidad excesiva y los casos especiales que posee D*; además, hoy en día se utiliza predominantemente D* Lite, mientras que D* ha quedado relegado a un contexto más teórico o histórico.

2.2 Implementaciones de Colas de Prioridad

Se implementaron 3 variantes diferentes para la cola de prioridad: *Binary Heap*, *Fibonacci Heap* y *Bucket Queue*. Todas las variantes fueron evaluadas utilizando el mismo algoritmo D* Lite.

2.2.1 Binary Heap

Es la clásica cola de prioridad, basado en un árbol binario completo almacenado en un arreglo dinámico, donde cada nodo cumple la propiedad de heap mínimo respecto de su clave.

Con respecto a las complejidades teóricas en esta cola de prioridad, son las siguientes:

- `top`/`topKey`: $O(1)$.
- `insert`: $O(\log n)$.
- `pop` (extraer mínimo): $O(\log n)$.
- `update`: $O(\log n)$ promedio (localización por hash + *sift-up*/*sift-down*).
- `remove`: $O(\log n)$ promedio.

2.2.2 Fibonacci Heap

Fibonacci Heap también se implementa como un árbol, pero a diferencia del *Binary Heap*, se compone de un conjunto de árboles de orden parcial, enlazados mediante listas doblemente enlazadas circulares. Además, mantiene un puntero al nodo mínimo y utiliza operaciones amortizadas para mejorar el costo de actualización de claves.

Complejidades amortizadas:

- `insert`: $O(1)$.
- `top`/`topKey`: $O(1)$.
- `update` (cuando disminuye clave): $O(1)$ amortizado.
- `pop` y `remove`: $O(\log n)$ amortizado.

2.2.3 Bucket Queue

La última variante analizada fue la *Bucket Queue*, la cual organiza los elementos en cubetas (*buckets*) según rangos de prioridad. En este caso, al implementarla para el algoritmo D* Lite, se utiliza principalmente el valor k_1 de la clave. Cada bucket contiene una lista de nodos con prioridades similares, y se mantiene un índice del bucket mínimo no vacío.

Complejidades en la implementación:

- `insert`: $O(1)$.
- `update`: $O(1)$ amortizado.
- `top`/`pop`: depende del contenido del bucket mínimo y del escaneo de buckets vacíos.

El peor caso de `pop` aparece cuando se deben recorrer múltiples buckets vacíos para encontrar el siguiente no vacío, por lo que el rendimiento depende de la parametrización (ancho de bucket y número total de buckets) y de la distribución real de claves.

3 El Algoritmo D* Lite

D* Lite (Koenig & Likhachev, 2002) es un algoritmo de búsqueda incremental para caminos mínimos en entornos dinámicos. En lugar de recomputar todo el camino nuevamente cada vez que sucede un cambio, utiliza la información que se calculó previamente y actualiza solo la porción que cambió.

Durante la ejecución, la búsqueda se realiza desde el objetivo hacia el inicio; esto permite que cuando aparezcan o desaparezcan obstáculos cerca de la posición del agente, la actualización se concentre solo en un subconjunto local de nodos.

El algoritmo mantiene dos estimaciones por nodo s:

- $g(s)$: costo actualmente conocido desde s hasta la meta.
- $rhs(s)$: estimación de un paso (*one-step lookahead*) definida como

$$rhs(s) = \min_{s' \in Succ(s)} \left(c(s,s') + g(s') \right)$$

3.1 Cola de prioridad

Cada nodo se ordena en la cola con una clave en una 2-upla:

$$k(s) = (k_1(s), k_2(s))$$

donde

$$k_1(s) = \min(g(s), rhs(s)) + h(s_{start}, s) + k_m, \qquad k_2(s) = \min(g(s), rhs(s)).$$

La comparación de claves es lexicográfica, es decir primero se compara k_1 y, en caso de empate, k_2.

La cola de prioridad es la estructura que mantiene los nodos que necesitan ser actualizados (es decir, los que cumplen $g(s) \neq rhs(s)$), ordenándolos según su clave $k(s) = (k_1, k_2)$ para procesar siempre primero el que más nos beneficia. Cuando el algoritmo se está ejecutando, se extrae el nodo con menor clave de la cola, se ajusta su valor de g para que coincida con *rhs* (si corresponde) y se propagan estos cambios a sus vecinos, los cuales pueden volverse inconsistentes e ingresar a la cola. De esta forma, la cola es la que lleva el control de manera eficiente de qué partes del grafo deben recalcularse, permitiendo actualizar el camino óptimo sin recomputar todo desde cero.

3.2 Complejidad

El costo total depende de cuántos nodos se vuelven inconsistentes tras cada cambio y del costo de las operaciones de la cola de prioridad. Por eso, en este trabajo comparamos el rendimiento de D* Lite con las distintas implementaciones de cola comentadas anteriormente (*Binary Heap*, *Fibonacci Heap* y *Bucket Queue*).

4 Metodología Experimental

4.1 Diseño del Benchmark

El objetivo es medir el impacto que tiene utilizar diferentes implementaciones de la cola de prioridad para una misma implementación de D* Lite. Para ello, se mantuvo fija la lógica del algoritmo y se reemplazó únicamente la cola por tres variantes: *Binary Heap*, *Fibonacci Heap* y *Bucket Queue*.

Cada experimento se ejecutó sobre una grilla 4-conexa rectangular. Por lo tanto, los movimientos se limitan a las direcciones cardinales. Se definió un costo unitario por arista válida y un costo infinito cuando hay obstáculos. Con respecto al nodo objetivo y al nodo de inicio, son fijados para cada escenario diferentes, y no cambian durante la ejecución del algoritmo.

Cada escenario tiene dimensiones diferentes, obstáculos iniciales y cambios dinámicos; también se realizó la evaluación en escenarios donde no había cambios.

Para cada corrida se registran las siguientes métricas:

- `initTimeMs`: tiempo de inicialización más la primera planificación.
- `totalTimeMs`: tiempo total de ejecución incluyendo re-planificaciones.
- `pathLength`: cantidad de nodos del camino final reconstruido.
- `pathCost`: costo total estimado por D* Lite desde el inicio.
- `replanifications`: número de eventos de replanificación dinámica.
- `pathFound`: indicador de éxito (camino encontrado/no encontrado).

Por lo tanto, la comparación más importante entre colas se centra en los tiempos `initTimeMs` y `totalTimeMs`, ya que esto nos permite ver de manera clara cómo es el comportamiento incremental en escenarios dinámicos.

4.2 Repeticiones y agregación estadística

Cada combinación *benchmark–cola de prioridad* se ejecutó $N = 10$ veces de forma independiente. Los tiempos pueden variar entre repeticiones por factores del sistema operativo (planificación de procesos, caché, interrupciones), por lo que un único valor no es suficiente para sustentar conclusiones sólidas.

Para cada métrica de tiempo se reportan:

- **media aritmética** sobre las N repeticiones;
- **desvío estándar muestral** (s, con $N-1$ en el denominador);
- **intervalo de confianza del 95 %** para la media, calculado como $\bar{x} \pm t_{0,975,N-1}\, s/\sqrt{N}$, donde $t_{0,975,N-1}$ es el cuantil de la distribución t de Student.

Las métricas discretas (`pathLength`, `pathCost`, `replanifications`, `pathFound`) son deterministas dado el escenario y se registran una sola vez por celda, ya que no varían entre repeticiones cuando el mapa y la semilla son fijos.

La salida del harness exporta cada corrida en CSV (columna `run`) y un script de post-procesamiento agrega las estadísticas y genera gráficos con barras de error.

4.3 Implementación

La implementación se realizó en C++17. Los experimentos se ejecutaron sobre un procesador AMD Ryzen 7 250 w/ Radeon 780M Graphics a 3.6 GHz, con 32 GB de memoria RAM, utilizando el sistema operativo Fedora Linux 42 (Workstation Edition).

Los tiempos se miden con `chrono::high_resolution_clock`, expresados en milisegundos en doble precisión. La salida se presenta por cola de prioridad, manteniendo el mismo orden de ejecución para todas las variantes.

Para poder garantizar una comparabilidad justa:

- se utiliza exactamente el mismo conjunto de benchmarks para las tres colas,
- cada escenario fija posiciones de inicio/meta y patrón de cambios,
- no se modifica ninguna otra parte del algoritmo D* Lite entre corridas.

4.4 Datasets y Escenarios de Prueba

Se utilizaron 12 datasets, agrupados en tres familias:

Escenarios estáticos base. Se diseñaron utilizando grillas vacías de varias dimensiones (por ejemplo, 10×10, 100×100 y 500×500), un caso con pared y un único pasillo, así como un laberinto con paredes alternadas.

Escenarios con obstáculos aleatorios. Se consideran grillas de 50×50 con densidad de obstáculos del 20 % y 30 %, y una grilla de 100×100 con densidad del 15 %. Estos benchmarks miden el desempeño cuando la estructura del espacio libre es irregular.

Escenarios dinámicos incrementales. Se incluyen configuraciones donde:

- aparece una pared durante la ejecución,
- desaparece parcialmente una pared inicialmente bloqueante,
- ocurren múltiples cambios en distintos pasos,
- hay cambios frecuentes en una grilla grande (100×100).

Todos los escenarios son **sintéticos**, es decir, las grillas son generadas con sus dimensiones, obstáculos y cambios dinámicos predefinidos. No se incluyeron mapas del mundo real (robótica, videojuegos, redes viales), lo que facilita la comparación controlada entre colas pero limita la generalización a entornos con topologías y distribuciones de claves distintas. Esta limitación se discute en la Sección 5.3.

5 Resultados Experimentales

A continuación presentaremos los resultados que se obtuvieron al ejecutar los 12 benchmarks.

5.1 Tabla comparativa

La siguiente tabla resume los tiempos de planificación inicial (`initTimeMs`) por escenario y por cola. Cada celda muestra la media $\pm$ desvío estándar sobre $N = 10$ repeticiones independientes.

Cuadro 1: Tiempo de planificación inicial (ms) por benchmark y cola de prioridad. Valores reportados como media $\pm$ desvío estándar (N repeticiones por celda).

Benchmark	BinaryHeap	FibonacciHeap	BucketQueue
empty_small	0.108 ± 0.057	0.147 ± 0.090	0.120 ± 0.064
empty_medium	9.411 ± 0.990	14.078 ± 1.045	16.202 ± 1.108
empty_large	307.941 ± 30.770	444.782 ± 58.996	1414.986 ± 273.709
wall_with_gap	0.348 ± 0.078	0.406 ± 0.091	0.316 ± 0.053
simple_maze	0.001 ± 0.000	0.001 ± 0.000	0.003 ± 0.000
random_20pct	2.807 ± 0.289	3.385 ± 0.271	2.972 ± 0.348
random_30pct	1.653 ± 0.254	1.889 ± 0.275	1.653 ± 0.281
dynamic_wall_appears	0.705 ± 0.088	0.924 ± 0.173	0.774 ± 0.094
dynamic_wall_disappears	0.254 ± 0.034	0.305 ± 0.023	0.268 ± 0.032
dynamic_multiple	1.530 ± 0.271	1.973 ± 0.312	1.715 ± 0.192
large_obstacles	12.046 ± 1.673	15.859 ± 1.504	15.211 ± 1.606
large_dynamic	17.619 ± 3.799	24.087 ± 5.721	24.707 ± 4.780

Se observa que *Binary Heap* obtiene, en general, los mejores tiempos en escenarios medianos y grandes, mientras que *Bucket Queue* puede resultar competitiva en instancias pequeñas, pero pierde eficiencia cuando crece el rango efectivo de claves (por ejemplo en *empty_large*).

5.2 Análisis gráfico

La Figura 1 muestra la comparación de tiempos iniciales por benchmark (escala logarítmica en eje y), con barras de error correspondientes al desvío estándar. Esto permite visualizar simultáneamente instancias pequeñas y grandes, así como la variabilidad relativa entre repeticiones.

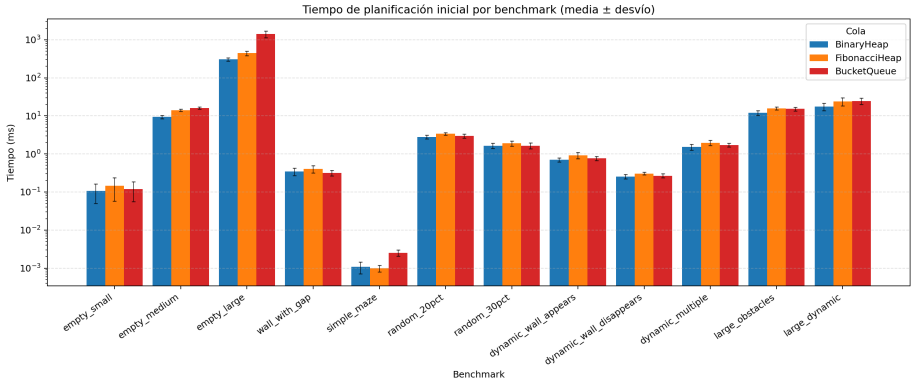

Figura 1: Tiempo de planificación inicial por benchmark y cola de prioridad (media ± desvío estándar, escala logarítmica).

La Figura 2 presenta el tiempo total de ejecución (incluyendo replanificaciones), donde se mantiene la tendencia general observada en `initTimeMs`. Adicionalmente, se presenta la tabla completa numérica correspondiente para respaldar los datos observados estadísticamente y favorecer la reproducibilidad.

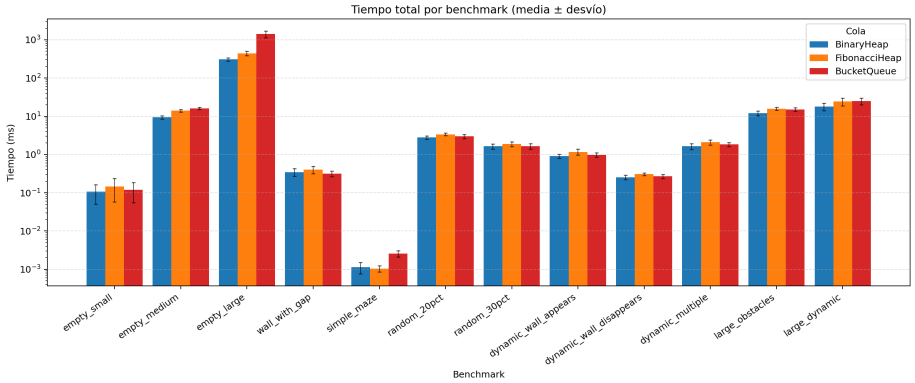

Figura 2: Tiempo total por benchmark y cola de prioridad (media ± desvío estándar, incluye replanificación).

Cuadro 2: Tiempo total de ejecución (ms) por benchmark y cola de prioridad. Valores reportados como media ± desvío estándar (N repeticiones por celda).

Benchmark	BinaryHeap	FibonacciHeap	BucketQueue
dynamic_multiple	1.645 ± 0.290	2.108 ± 0.325	1.838 ± 0.208
dynamic_wall_appears	0.912 ± 0.114	1.169 ± 0.205	0.978 ± 0.127
dynamic_wall_disappears	0.255 ± 0.034	0.306 ± 0.023	0.269 ± 0.033
empty_large	307.942 ± 30.772	444.783 ± 58.996	1414.986 ± 273.709
empty_medium	9.411 ± 0.990	14.078 ± 1.045	16.202 ± 1.108
empty_small	0.108 ± 0.057	0.147 ± 0.090	0.120 ± 0.064
large_dynamic	17.851 ± 3.844	24.349 ± 5.771	25.017 ± 4.826
large_obstacles	12.046 ± 1.673	15.859 ± 1.504	15.212 ± 1.606
random_20pct	2.807 ± 0.289	3.385 ± 0.272	2.973 ± 0.348
random_30pct	1.653 ± 0.254	1.890 ± 0.275	1.653 ± 0.281
simple_maze	0.001 ± 0.001	0.001 ± 0.001	0.003 ± 0.001
wall_with_gap	0.348 ± 0.078	0.406 ± 0.091	0.316 ± 0.053

5.3 Alcance y limitaciones de la medición

Las mediciones de tiempo reportadas corresponden al promedio de $N = 10$ ejecuciones por cada combinación *benchmark–cola*, acompañadas de desvío estándar e intervalos de confianza del 95 %. En la mayoría de los escenarios la dispersión es pequeña respecto de la media, lo que respalda la comparación relativa entre colas, en instancias muy rápidas (p. ej., grillas pequeñas) el ruido del sistema puede representar una fracción mayor del tiempo medido.

Persisten limitaciones metodológicas importantes. En primer lugar, los escenarios aleatorios usan semillas fijas, la variabilidad reportada refleja ruido temporal del hardware y del SO, pero no distintas topologías de obstáculos. En segundo lugar, todos los benchmarks son sintéticos, los resultados deben interpretarse como un **punto de partida** para la comparación interna entre colas, no como validación definitiva en mapas reales con patrones de claves y densidades distintas. Finalmente, la evaluación empírica se centró de forma exclusiva en los tiempos de ejecución. Debido a que no se realizaron mediciones de consumo de memoria ni un profiling detallado, la evaluación del trade-off entre tiempo y memoria escapa al alcance de los resultados presentados y se propone como una etapa siguiente.

6 Discusión

Los resultados muestran que en promedio global *Binary Heap* obtiene menor tiempo que *Fibonacci Heap* y *Bucket Queue*, tanto en planificación inicial como en tiempo total. En términos relativos, *Binary Heap* mejora aproximadamente un 14.6 % respecto de *Fibonacci Heap* y un 64.4 % respecto de *Bucket Queue* en `initTimeMs`.

6.1 Brecha entre teoría y práctica

Teóricamente *Fibonacci Heap* ofrece mejores cotas amortizadas para algunas operaciones (por ejemplo, *decrease-key*). Pero observando los resultados, bajo esta implementación esa ventaja no se traduce en mejor rendimiento final. Esto hace pensar que el costo constante de mantener una estructura más compleja domina sobre la mejora asintótica en muchos escenarios del benchmark.

La hipótesis principal que ofrecemos para explicar la eficiencia del *Binary Heap* es su alta localidad espacial, teóricamente, al estar basado en un arreglo contiguo, favorece sustancialmente el uso de la caché del procesador. En cambio, *Fibonacci Heap* depende de nodos dinámicos y punteros, lo que incrementa los saltos de memoria y reduce dicha localidad. Debido a que nuestro marco experimental no incluyó profiling detallado ni métricas empíricas de consumo de memoria y fallos de caché, esta explicación se mantiene como un concepto teórico, dejando pendiente la resolución del trade-off entre tiempo y memoria para futuros experimentos.

Por otro lado el caso de *Bucket Queue*, el rendimiento depende fuertemente de la distribución efectiva de claves. Cuando la parametrización de buckets no coincide bien con el rango de prioridades observado, aparecen escaneos de buckets vacíos que incrementan el tiempo total, especialmente en instancias grandes.

6.2 Profundidad técnica y patrón de operaciones

En cuanto al patrón de operaciones de D* Lite, este algoritmo fundamenta su eficiencia en llamadas masivas a la función `UpdateVertex`, la cual internamente realiza inserciones, actualizaciones (`DecreaseKey`/`IncreaseKey`) o eliminaciones en la cola de prioridad. En escenarios con re-planificación dinámica como los implementados (aparición o desaparición de obstáculos), `DecreaseKey` y `PopMin` dominan el perfil de ejecución.

Teóricamente, el *Fibonacci Heap* realiza `DecreaseKey` en $O(1)$ amortizado, lo que motivaría su uso frente a la cota $O(\log n)$ del *Binary Heap*. Sin embargo, para que el *Fibonacci Heap* logre ser superior en la práctica, la proporción entre la cantidad de `DecreaseKey` y `PopMin` debe ser extremadamente alta, ya que la operación `PopMin` en la variante Fibonacci dispara una onerosa rutina de consolidación de árboles estructurales. En nuestras grillas sintéticas (4-conexas, costos unitarios), la conectividad es baja y las ramas de propagación son acotadas, por lo que el algoritmo extrae nodos (`PopMin`) frecuentemente, obligando al *Fibonacci Heap* a reestructurarse constantemente antes de capitalizar la reducción de costos en la clave. Adicionalmente, el costo constante de crear y destruir nodos enlazados impacta negativamente.

En contraste, el *Binary Heap* maneja sus operaciones de ajuste (*sift-up*/*sift-down*) mediante intercambios simples de índices de arreglos. Aunque asintóticamente es $O(\log n)$, la escasa altura del heap binario y la constante de operación ínfima hacen que en las replanificaciones, las operaciones de propagación terminen siendo sustancialmente más rápidas.

En el caso de *Bucket Queue*, pese a que maneja `Insert` y `DecreaseKey` en $O(1)$, la operación de extracción del mínimo puede sufrir demoras por escanear *buckets* vacíos. D* Lite puede introducir saltos amplios en las claves debido a la heurística y los cambios por obstáculos, provocando que los ítems queden dispersos, esto evidencia por

qué la estructura pierde competitividad a medida que el mapa escala o los cambios son disruptivos.

6.3 Implicaciones prácticas

Según nuestros benchmarks, para lograr una mejor eficiencia al implementar D* Lite en aplicaciones reales, los resultados sugieren utilizar *Binary Heap* como opción por defecto, en particular para grillas medianas y grandes y para entornos con cambios dinámicos frecuentes. *Bucket Queue* puede ser útil en contextos acotados y bien parametrizados, mientras que *Fibonacci Heap* podría justificarse solo cuando el patrón de operaciones y la escala hagan visible su ventaja amortizada.

Estos resultados deben interpretarse junto con la limitación de que se calcularon como promedios sin un análisis de varianza completo.

7 Conclusiones

En este trabajo se analizó el impacto de la elección de la cola de prioridad en el rendimiento del algoritmo D* Lite para planificación de rutas en grafos dinámicos. Se compararon tres implementaciones: *Binary Heap*, *Fibonacci Heap* y *Bucket Queue* mediante 12 benchmarks sintéticos controlados que cubren escenarios estáticos, aleatorios y dinámicos.

7.1 Hallazgos principales

1. **Predominancia de Binary Heap:** En promedio global, *Binary Heap* obtiene 35.77 ms de tiempo de planificación inicial, mejorando significativamente respecto de *Fibonacci Heap* (41.87 ms, +14.6 %) y *Bucket Queue* (100.39 ms, +64.4 %).

2. **Brecha teoría-práctica:** Aunque *Fibonacci Heap* posee mejores cotas amortizadas teóricas, este beneficio no se reflejó en el rendimiento práctico bajo los benchmarks utilizados, debido al mayor costo constante de mantener la estructura.

3. **Hipótesis sobre la localidad de memoria:** Se teoriza que el rendimiento superior de *Binary Heap* se atribuye a su mayor localidad espacial (basado en arreglo) frente a estructuras con punteros. Validar el trade-off exacto entre tiempo y consumo de memoria o fallos de caché representa una oportunidad fundamental para investigaciones futuras.

4. **Escalabilidad:** La brecha de rendimiento se amplifica en instancias grandes (por ejemplo, en *empty_large*, *Binary Heap* es 3x más rápido que *Bucket Queue*).

7.2 Transferencia de conocimiento

Los resultados son directamente aplicables a sistemas de robótica, navegación autónoma, planificación de movimiento, entre otras aplicaciones, donde el algoritmo D* Lite

es ampliamente utilizado. La elección de la cola de prioridad en la implementación del algoritmo puede terminar afectando significativamente el rendimiento en tiempo real, por lo cual este análisis experimental aporta evidencia sólida para guiar decisiones de diseño.

8 Trabajo Futuro

Este trabajo se puede seguir expandiendo y mejorando, por ejemplo haciendo:

- **Variabilidad estructural:** Complementar las repeticiones con distintas semillas de generación aleatoria por corrida, para separar la variabilidad temporal del efecto de distintas topologías de obstáculos.
- **Otras estructuras de datos para la cola de prioridad:** Explorar *Radix Heap*, *Pairing Heap* y *Skew Heap*, que pueden ofrecer perfiles de desempeño distintos según el patrón de operaciones.
- **Variantes del algoritmo:** Estudiar otras optimizaciones de D* Lite (por ej., D* Lite con Theta*) y cómo impactan y se ven afectadas por la elección de la cola de prioridad.
- **Benchmarks del mundo real:** Incluir mapas más realistas o datasets estándar utilizados ampliamente en robótica y *path planning* (como por ejemplo los provistos por escenarios de videojuegos o repositorios estandarizados), lo cual puede exhibir patrones de claves y distribuciones distintos a los escenarios puramente sintéticos evaluados.
- **Profiling detallado:** Medir no solo tiempo de ejecución, sino también accesos a memoria, fallos de caché, y predicción de saltos, para cuantificar los mecanismos que explican las diferencias observadas.

Estas extensiones u otras ayudarán y tendrán un gran valor en la comunidad de algoritmos de rutas dinámicas.

Nota de redacción: Durante la preparación de este trabajo se utilizó Grammarly con el fin de mejorar el lenguaje y la legibilidad del manuscrito. (Aclaramos esto para asegurar la integridad académica, debido a que dicha herramienta emplea inteligencia artificial).

Bibliografía

Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, *1*(1), 269-271. https://doi.org/10.1007/BF01386390

Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, *4*(2), 100-107. https://doi.org/10.1109/TSSC.1968.300136

Fredman, M. L., & Tarjan, R. E. (1987). Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, *34*(3), 596-615. https://doi.org/10.1145/28869.28874

Stentz, A. (1994). Optimal and efficient path planning for partially-known environments. *Proceedings of the IEEE International Conference on Robotics and Automation*, 3310-3317. https://doi.org/10.1109/ROBOT.1994.351061

Koenig, S., & Likhachev, M. (2002). D* Lite. *Proceedings of the AAAI Conference on Artificial Intelligence*, *18*(1), 476-483.

Zhang, L., Wang, Y., & Li, H. (2024). A Hybrid Path Planning Approach for Mobile Robots Based on Improved D* Lite and DWA. *Journal of Robotics*, *2024*, 1-15.