Proyecto Longitudinal - Entrega Final

Algoritmos avanzados de búsqueda y optimización

Carol Glass, Sebastián Fripp, Esteban Casalas, Rafael Filardi

26 de Noviembre de 2024

1 Descripción del problema

Se plantea el siguiente problema a abordar sobre resolucion de laberintos:

> Dado un laberinto M, un comienzo s y un destino d, existe un camino que conecta s y d

Nos genera particular interes los A_M (Algoritmos que decidan al problema). Para resolver este problema se puede reformular y tratar como un **problema de búsqueda en grafos**, donde un laberinto es representado por un grafo $G = (V, E)$, en el cual cada nodo $v \in V$ representa una celda del laberinto, y cada arista $e = (u, v) \in E$ representa un posible movimiento entre dos celdas adyacentes u y v. El objetivo es encontrar un camino desde un nodo inicial $s \in V$ (entrada del laberinto) hasta un nodo objetivo $t \in V$ (salida del laberinto). Para esto, se plantean cuatro posibles algoritmos algoritmos de búsqueda que responden ante diferentes estrategias de resolucion: BFS, DFS, Dijkstra y A* (Con varias configuraciones heuristicas).

Dado que cada arista puede tener un costo $c(u, v) \geq 0$, se busca encontrar un camino $P = \{s, v_1, v_2, \ldots, t\}$ tal que la suma de los costos asociados representa la distancia del camino entre el inicio y el destino, es decir:

$$\sum_{i=0}^{n-1} c(v_i, v_{i+1}),$$

donde $c(v_i, v_{i+1})$ es el costo de la arista entre v_i y v_{i+1}, y n es el número de nodos en el camino. Este camino P es **una solución al problema planteado**.

BFS resuelve el camino encontrando realizando una busqueda por niveles, mientras que DFS realiza lo mismo pero en profundidad. Ambos sin dar garantias de optimalidad (camino mas corto). POr otro lado el algoritmo de Dijkstra resuelve este problema buscando el camino de menor coste desde s a t en grafos con pesos no negativos, mientras que el algoritmo A* incorpora una función heurística $h(v)$, que aproxima el costo restante desde un nodo v hasta la meta t. Ambos algoritmos serán evaluados no solo en términos de si encuentran un camino óptimo, sino también en términos de su **complejidad temporal** $O(f(n))$, donde n es el número de nodos explorados, y su eficiencia en escenarios con diferentes restricciones, como la presencia de múltiples soluciones o caminos bloqueados.

Finalmente, se analizará el comportamiento de los algoritmos que resuelven el problema de acuerdo a las distintas estrategias, considerando el largo del camino encontrado, el orden de ejecucion, y nodos visitados.

2 Estrategias de resolucion al problema

1. Podemos encontrar un camino que conduzca a la salida? (BFS/DFS)
2. Podemos lograr que ese camino solucion sea el mas corto posible? (Dijkstra)
3. Podemos encontrar el camino mas corto posible visitando la menor cantidad de nodos posibles? (A* con distindas configuraciones de heuristicas)

3 Algoritmos

En esta sección definiremos los algoritmos utilizados y, para el caso de Dijkstra y A*, brindaremos una demostración de los mismos. Mencionaremos además los algoritmos de Breadth-First Search y Depth-First Search que también resuelven el problema pero con un enfoque diferente.

Estrategia: Encontrar un camino del inicio a la salida

3.1 DFS (Depth First Search)

El algoritmo DFS (Depth First Search) explora un grafo o árbol comenzando desde un nodo inicial y avanzando tan profundamente como sea posible por cada rama antes de retroceder.

Pasos del Algoritmo DFS

1. Marca todos los nodos como no visitados.
2. Selecciona un nodo inicial s desde el cual comenzará la búsqueda.
3. Crea una pila P e inicialízala con el nodo inicial s.
4. Mientras P no esté vacía:
   (a) Extrae el nodo u de la cima de la pila.
   (b) Si u no ha sido visitado:
      i. Marca u como visitado.
      ii. Para cada nodo vecino v de u (en orden preferido):
         - Si v no ha sido visitado, agrégalo a la pila.
5. Repetir hasta que se visite el nodo objetivo t o P esté vacía.

Tiempo de ejecucion DFS

$$O(m+n)$$

Visita cada nodo una vez y evalua las aristas conectadas al nodo solo una vez, por lo tanto el orden es la suma del total de nodos y el total de aristas

3.2 BFS (Breadth-First Search)

El algoritmo BFS (Breadth-First Search) es muy similar a DFS, pero en este caso se explora el grafo por niveles. En la implementación del algoritmo, lo único que varía es cómo se ordenan los elementos en la cola: los elementos se extraen del inicio y se agregan al final (cola de tipo FIFO).

Pasos del Algoritmo BFS

1. Marca todos los nodos como no visitados.
2. Selecciona un nodo inicial s desde el cual comenzará la búsqueda.
3. Crea una cola Q e inicialízala con el nodo inicial s.
4. Mientras Q no esté vacía:
   (a) Extrae el nodo u del inicio de la cola.
   (b) Si u no ha sido visitado:
      i. Marca u como visitado.
      ii. Para cada nodo vecino v de u (en orden preferido):
         - Si v no ha sido visitado, agrégalo al final de la cola.
5. Repetir hasta que se visite el nodo objetivo t o Q esté vacía.

Tiempo de ejecucion BFS

$$O(m+n)$$

Visita cada nodo una vez y evalua las aristas conectadas al nodo solo una vez, por lo tanto el orden es la suma del total de nodos y el total de aristas

Estrategia: Encontrar un camino de menor coste del inicio a la salida

3.3 Algoritmo de Dijkstra

El algoritmo de Dijkstra es un algoritmo de búsqueda de caminos mínimos en grafos con pesos no negativos. Dado un grafo $G = (V, E)$ donde V es el conjunto de nodos y E es el conjunto de aristas con pesos $c(u, v) \geq 0$ para cada arista $e_0 = (u, v)$, el algoritmo de Dijkstra encuentra el camino de costo mínimo desde un nodo origen $s \in V$ a un nodo destino $t \in V$.

Pasos del algoritmo

1. Inicializar una función de distancia $d(v)$ para todos los nodos $v \in V$ tal que $d(s) = 0$ (distancia desde el nodo inicial hasta el nodo inicial) y $d(v) = \infty$ para todos los demás nodos.
2. Crear un conjunto Q de nodos no visitados y agregar todos los nodos $v \in V$.
3. Mientras Q no esté vacío:
   (a) Seleccionar el nodo $u \in Q$ con la menor distancia $d(u)$ (en el primer paso, será el nodo s).
   (b) Para cada vecino v de u, si $v \in Q$, actualizar la distancia como sigue:
   $$d(v) = \min(d(v), d(u) + c(u, v))$$
   donde $c(u, v)$ es el costo de la arista entre u y v.
   (c) Eliminar u de Q.
4. Repetir hasta que se visite el nodo objetivo t o Q esté vacío.

El algoritmo garantiza encontrar el camino de costo mínimo entre s y t debido a que siempre selecciona el nodo con la menor distancia acumulada en cada iteración.

La complejidad temporal de Dijkstra es $O(|V|^2)$ para una implementación basada en matrices de adyacencia, o $O(|E| \log |V|)$ cuando se utiliza una cola de prioridad (por ejemplo, un montículo binario).

Demostración

La demostración asegura que las distancias $d(u)$ para cada nodo u seleccionado en cada paso son óptimas.

Paso base: Cuando $u = s$ (primer nodo seleccionado):

- La distancia inicial es $d(s) = 0$, lo cual es correcto porque el costo del camino desde s a sí mismo es cero.
- No hay nodos anteriores a relajar, por lo que $d(s)$ ya es mínima.

Paso inductivo: Supongamos que después de n iteraciones, $d(u)$ es la distancia mínima desde s a u para todos los nodos u que ya fueron eliminados del conjunto Q.

Selección del nodo u: El nodo u seleccionado en la iteración actual es el que tiene la menor distancia $d(u)$ en Q.

- Como los nodos ya eliminados de Q tienen distancias mínimas, y u tiene la menor $d(u)$ en Q, no puede haber un camino más corto hacia u que no pase por los nodos ya procesados.

Exploración de las aristas de u: Para cada vecino v de u, si $v \in Q$, se actualiza $d(v)$ como:

$$d(v) = \min(d(v), d(u) + c(u, v))$$

- Por la propiedad de pesos no negativos ($c(u, v) \geq 0$), cualquier camino desde s hacia v que pase por u es al menos tan corto como los caminos previamente calculados.

Eliminación de u: Al eliminar u de Q, su distancia $d(u)$ queda fijada como mínima, ya que todos los posibles caminos más cortos hacia u ya han sido considerados.

Conclusión inductiva: En cada iteración, el nodo u seleccionado tiene su distancia $d(u)$ correctamente calculada como mínima, y los nodos restantes en Q se mantienen con distancias consistentes que serán optimizadas en iteraciones posteriores.

Estrategia: Encontrar encontrar el camino mas corto posible visitando la menor cantidad de nodos posibles?

3.4 Algoritmo A*

El algoritmo A* es una variación del algoritmo de búsqueda de caminos mínimos que utiliza una heurística para guiar la búsqueda. Dado un grafo $G = (V, E)$, el algoritmo A* busca un camino desde un nodo inicial s hasta un nodo objetivo t, minimizando el costo total que combina el costo real desde el inicio hasta el nodo actual y una estimación heurística del costo restante hasta la meta.

Función objetivo: A* utiliza una función de evaluación $f(v)$ que combina el costo acumulado $g(v)$ (el costo del camino desde s hasta v) y una función heurística $h(v)$ (una estimación del costo desde v hasta t):

$$f(v) = g(v) + h(v)$$

donde $h(v)$ es una función que estima el costo de llegar desde v al nodo objetivo t, y debe ser admisible, es decir, $h(v) \leq \text{costo\_mínimo\_real}(v, t)$ para todo v, para asegurar la optimalidad del algoritmo.

Pasos del algoritmo

1. Inicializar:
   - $g(s) = 0$ y $f(s) = h(s)$.
   - Colocar s en una cola de prioridad Q ordenada por $f(v)$.
2. Mientras Q no esté vacía:
   (a) Extraer el nodo u con el menor valor de $f(u)$ de Q.
   (b) Si $u = t$, se ha encontrado el camino óptimo.
   (c) Para cada vecino v de u:
      - Calcular el costo temporal: $g'(v) = g(u) + c(u, v)$.
      - Si $g'(v) < g(v)$, actualizar $g(v) = g'(v)$ y $f(v) = g(v) + h(v)$.
      - Si v no está en Q, agregarlo.

Si la heurística $h(v)$ es admisible (nunca sobreestima el costo real), el algoritmo A* garantiza encontrar el camino de menor costo entre s y t. La complejidad temporal de A* depende de la calidad de la heurística y de la estructura del grafo, pero en general es $O(|E| \log |V|)$ cuando se implementa con una cola de prioridad eficiente.

Demostración de validez de A*

Para la demostración, haremos uso del siguiente lema: Asuma que h es una heurística admisible. Para todo camino óptimo P de s a algún nodo $t \in T$, existe $v \in O$ con $f(v) \leq f^*$.

Demostración: Por el lema anterior, en cualquier arista hay escogencias válidas con $f \leq f^*$ en Q. Si el algoritmo termina de manera no óptima es porque sacó un nodo terminal t^* que tiene mínimo:

$$f[t^*] = g[t^*] + h[t^*], \quad \text{luego } g[t^*] > f^*.$$

Según el lema anterior, existe $v \in O$ con $f[v] < f^*$ en el paso anterior en Q, lo cual es una contradicción.

Heurísticas

Sin Heurística: Esta heurística implica que $h(v) = 0$ para todos los vértices v. Entonces, el algoritmo A* se transforma literalmente en Dijkstra. Aquí nos planteamos si esta heurística es en realidad admisible, y la respuesta es que sí. Una heurística es admisible cuando $h(v) \leq$ costo_mínimo_real(v, t), y esto se cumple para todos los vértices ya que las distancias son siempre mayores a 0.

Distancia Manhattan: La distancia Manhattan calcula la suma de las diferencias absolutas en las coordenadas x e y entre dos puntos:

$$h(x_1, y_1, x_2, y_2) = |x_2 - x_1| + |y_2 - y_1|$$

Esto representa el camino más corto posible si solo se permite movimiento ortogonal (lo cual coincide con nuestro caso). Como no considera restricciones adicionales (como obstáculos o costos diferenciales en las aristas), el costo calculado por esta heurística nunca será mayor que el costo real, volviéndola una heurística admisible.

Heurística perfecta: Si $\hat{h}(v)$ es una heurística perfecta, entonces coincide exactamente con $h(v)$, lo que implica que proporciona el costo real de un camino desde s hasta el nodo objetivo t. Además, dado que $\hat{h}(v) = h(v)$, esta heurística es admisible. Al ser perfecta, cada expansión realizada será óptima, ya que se considerarán los costos reales mínimos. Esto asegura que el algoritmo alcanzará el nodo objetivo con el menor costo sin necesidad de expandir nodos adicionales.

Heurística "Podar" Laberinto: Consiste en dos pasos:

1. Crear una versión simplificada del laberinto original sobre la cual calcularemos los valores óptimos de distancia hasta el nodo objetivo t para todo nodo v. Con simplificar el laberinto, nos referimos a "quitarle" paredes para que los caminos sean más "directos" hacia el nodo objetivo.

2. Utilizar estas distancias calculadas como valores de la heurística en la ejecución de A*.

Es importante resaltar que, como en un laberinto más sencillo, la solución óptima siempre será menor a la solución óptima del problema complejo, esta heurística no sobreestima el costo verdadero, por lo tanto es admisible.

3.5 Aplicaciones Potenciales de resolver el problema

En principio, puede parecer que resolver un laberinto es un problema trivial y de poca importancia, pero en realidad es algo a lo que se puede sacar mucho provecho si se logra hacer de forma eficiente. Por ejemplo, consideremos el caso del transporte, una de las aplicaciones más conocidas actualmente, donde una persona necesita llegar de un punto A a un punto B, siendo el punto A su casa y el punto B la universidad. Estos algoritmos pueden aportar gran valor al lograr encontrar el camino más corto y eficiente. En este caso, se puede representar el mapa como un grafo, donde los pesos de las aristas representan la demora que podrían agregar al viaje. Por ejemplo, transitar por una calle con mucho tráfico podría afectar considerablemente la duración del trayecto. Implementar este sistema de forma eficiente implica calcular los tiempos de demora en forma dinámica y actualizar los pesos de las aristas para representar el estado en tiempo real del tráfico.

A continuación, se mencionan otros escenarios donde se genera valor al encontrar una ruta óptima:

Salidas de Emergencia

Un caso de importancia vital para las personas es encontrar la salida de emergencia más cercana a su ubicación dentro de un edificio en caso de una emergencia. Un ejemplo que refleja esta aplicación de Dijkstra es el proyecto de KISHWAR, que corresponde al estudio y simulación de una red que representa al edificio del Rectorado de la ESPOL. Este proyecto implementa un sistema de evacuación dinámica, ya que las señaléticas estáticas actuales dentro de los edificios, que indican hacia dónde se encuentra la salida más cercana, no cumplen con las expectativas en cuanto a la cantidad de vidas salvadas. Por el contrario, este proyecto contempla la implementación de una red de sensores y señaléticas basadas en LEDs para redirigir a las personas hacia las salidas de emergencia de manera más eficiente.

Robótica

En robótica, la búsqueda avanzada es crucial para la navegación y toma de decisiones en entornos dinámicos. Por un lado, esto puede observarse en la colaboración entre robots, donde se aplican algoritmos como *Swarms* o algoritmos de búsqueda multiagente. Estos algoritmos coordinan la acción de múltiples robots para alcanzar objetivos conjuntos, como la búsqueda y rescate o la automatización industrial. Por otro lado, también se emplean algoritmos basados en aprendizaje por refuerzo y búsqueda evolutiva, que permiten que los robots ajusten su comportamiento y rutas en función de cambios en su entorno, adaptándose continuamente.

Pathfinding

Una de las aplicaciones más comunes es el *pathfinding* en videojuegos, una parte crucial que aporta gran valor a la jugabilidad y la experiencia del usuario. Dependiendo del videojuego, esta tarea puede escalar en complejidad, ya que el terreno podría no ser tan trivial como un laberinto bidimensional. En casos más simples, esta técnica se conoce como *graph traversal*.

Estas son algunas de las aplicaciones más directas, pero no las únicas. En estos ejemplos, nos centramos en la idea de buscar "caminos". Sin embargo, no debemos olvidar que los algoritmos de Dijkstra y A* son algoritmos de optimización y no están limitados a este enfoque. Por ende, existen otros contextos en los que estos algoritmos pueden utilizarse, siempre y cuando el problema pueda traducirse a una representación en la que estos algoritmos tengan sentido.

4 Síntesis

El documento presentado aborda la resolución de problemas de búsqueda y optimización en laberintos mediante algoritmos avanzados, específicamente Dijkstra y A* (con diferentes heuristicas, con el fin de reducir la cantidad de nodos visitados). El problema se modela como un grafo en el que se busca el camino óptimo entre dos nodos (entrada y salida). Ambos algoritmos se analizan en términos de complejidad, eficiencia y adaptabilidad a diversas configuraciones.

Se presentan también otros métodos complementarios, como DFS y BFS, y se destacan las aplicaciones prácticas de estos algoritmos, que van desde el diseño de rutas de emergencia hasta su uso en robótica y videojuegos.

Un punto ya planteado al inicio, pero reforzado tras el análisis de la resolución de laberintos, es que este problema resulta altamente escalable y aplicable. Lo que en un principio parece algo simple, como simplemente dibujar un camino que conecte un punto de partida con una salida, se convierte en un problema de optimización y exploración con importantes aplicaciones en problemas reales de negocio.

Referencias

- Wikipedia. (2024). *Maze-solving algorithm*. Recuperado de https://en.wikipedia.org/wiki/Maze-solving_algorithm
- Tosato, M. (2024, enero 9). *Exploring the depths: Solving mazes with A search algorithm*. Medium. Recuperado de https://matteo-tosato7.medium.com/exploring-the-depths-solving-mazes-with-a-search-algorithm-c15253104899
- Kidscodecs. (2018). *Maze solving algorithms*. Recuperado de https://kidscodecs.com/maze-solving-algorithms/
- SCIRP. (n.d.). *Paper information*. Recuperado de https://www.scirp.org/journal/paperinformation?paperid=70460
- Velasco, M. (2024). *El Algoritmo A**. Recuperado de https://mauricio-velasco.github.io/algobo2024/notes/A_Star.pdf
- Velasco, M. (2024). *Heaps and Dijkstra Notes*. Recuperado de https://mauricio-velasco.github.io/algobo2024/notes/Heaps_and_Dijkstra.pdf
- Heuristics in Game Programming. Stanford University. Recuperado de https://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
- Game programming resources. Stanford University. Recuperado de https://theory.stanford.edu/~amitp/GameProgramming/