# Proyecto Final

# Algoritmo de búsqueda $A^*$

## 0.1 Abstract

Se denomina algoritmo de búsqueda a cualquier algoritmo que resuelve el problema de búsqueda, es decir, extraer información almacenada en alguna estructura de datos o calculada en el espacio de búsqueda de un dominio de problema. Un tipo de problema que se trabaja con algoritmos de búsqueda es el de búsqueda combinatorial donde generalmente se búsca una subestructura específica de una estructura discreta como un grafo. Cuando se búsca una subestructura con un valor máximo o mínimo de algún parámetro se denomina optimización combinatorial.

Dentro de la búsqued combinatorial, un problema muy común es encontrar el camino más corto entre 2 vértices de un grafo ponderado. En este caso, la subestructura que se trata de extraer es un camino, donde se especifica el vértice inicial y final de antemano y donde el valor que se trata de minimizar es la suma de los pesos asociados a las aristas del camino.

Para este proyecto se resuelve un problema de camino más corto sobre un grafo representado por una matriz, que a su vez representa un tablero con celdas, donde 2 de estas celdas representarán el vértice inicial (cazador) y el vértice final (presa), mientras que las demás celdas especificarán los pesos que conectan a las celdas.

En cuando al algoritmo de búsqueda, se usa el algoritmo $A^*$ el cual combina características del algoritmo de Dijkstra y del algoritmo de búsqueda voraz el mejor primero. Mientras que Dijkstra da prioridad a los vértices más cercanos al vértice inicial, búsqueda voraz el mejor primero da prioridad a vértices que estás más cerca del vértice final. El algoritmo $A^*$ se guiará por la suma de ambos términos, es decir, por la suma del costo $g(x)$ de ir del vétice inicial a un vértice $x$, y el costo $h(x)$ de ir de un vértice $x$ al vértice final, dicha suma se representará por $f(x) = g(x) + h(x)$.

## 0.2 Metodología

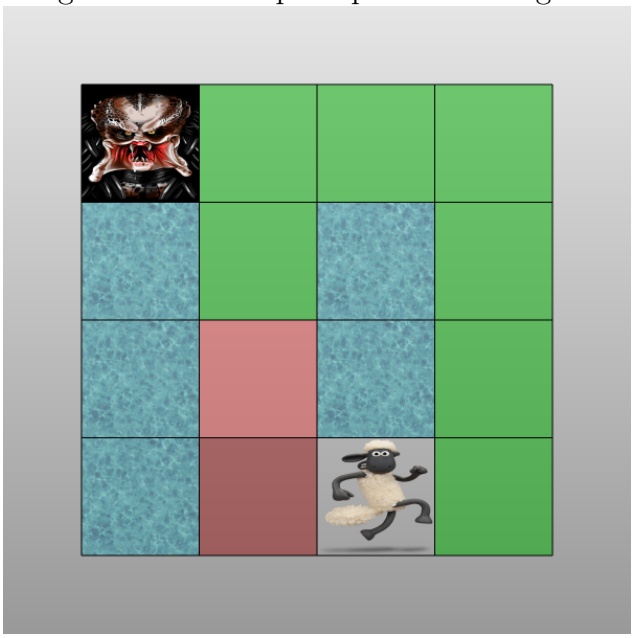
Como ya se comentó, se búscará un camino mínimo sobre una matriz que llamaremos $W$, donde cada elemento $W_{ij}$ la denominaremos celda. Dicha matriz se define por el usuario, previo a la ejecución del programa y contendrá 4 tipos de celdas diferentes:

- $W_{ij} = 0$ que representa celdas que están bloqueadas y por lo tanto no se puede pasar por ellas y no se toman en cuenta a la hora de buscar un camino mínimo. Gráficamente se representan por cuadrados de color azul.
- $W_{ij} = 1$ que representa celdas a las que es posible llegar con un costo de 1. Gráficamente se representan por cuadrados de color verde.

- $W_{ij} = 2$ que representa celdas con obstáculos y a las que es posible llegar con un costo de 2. Gráficamente se representan por cuadrados de color rojo claro.
- $W_{ij} = 2.5$ que representa celdas con obstáculos y a las que es posible llegar con un costo de 2.5. Gráficamente se representan por cuadrados de color rojo oscuro.

Una vez que se definen los valores de la matriz y se inicia el programa, se escogen 2 celdas $S_0$ y $S_N$ al azar que no tengan valor cero. Dichas celdas representarán, el vértice inicial con $S_0$ y el vértice final $S_N$ del camino mínimo que se búscará. Gráficamente $S_0$ está representado por la imagen del personaje predador, y $S_N$ por la imagen de la oveja.

Figure 1: Matriz que representará el grafo



Para este proyecto se considera que las celdas están conectadas a sus 8 vecinos, es decir, los vecinos de la celda $W_{ij}$ son los elementos: $W_{(i-1)(j-1)}$, $W_{(i)(j-1)}$, $W_{(i+1)(j-1)}$, $W_{(i-1)(j)}$, $W_{(i+1)(j)}$, $W_{(i-1)(j+1)}$, $W_{(i)(j+1)}$, $W_{(i+1)(j+1)}$, siempre y cuando estas celdas estén dentro de la matriz y no tengan un valor de 0.

En este caso se búscará un camino que minimice $f(S_N)$, es decir, que el costo de llegar del vértice inicial $S_0$ al vértice final $S_N$ sea mínimo. Se define $h(c)$ como la distancia diagonal, esto es, $h(c) = max(|c.x - S_N.x|, |c.y - S_N.y|)$ donde $c.x$, $S_N.x$ son las coordenadas $x$ de las celdas $c$ y $S_N$ respectivamente, mientras que $c.y$, $S_N.y$ son las coordenadas $y$ de las celdas $c$ y $S_N$ respectivamente.

Para explorar los vértices se usa una cola de prioridad, lo cual permitirá obtener el vértice con menor costo $f(x)$ en tiempo constante. Para mantener el estado de cada vértice, se usa

una matriz *world* donde llevará registro de los costos y la celda padre por donde se llegó al nodo actual y que es camino mínimo.

También, debido a que se tiene un grafo no dirigido, se requiere saber cuando un nodo ya se visitó para no volverlo a agregar a la cola de prioridad, para esto, se usa una matriz llamada *closed* con valores booleanos que se inicializará en $false$ ya que al inicio ningún nodo se ha visitado. Cada que se saca un vértice o celda de la cola de prioridad se marca como visitado, cambiando el valor a $true$ dentro de la matriz *closed*.

Para poder interactuar en tiempo real con el tablero, cambiando la posición de la celda $S_N$ mientras se recalcula el camino más corto entre $S_0$ y $S_N$, se programó una interfaz gráfica donde se muestra en una ventana el tablero con todas las celdas de la matriz $W$, el vértice inicial $S_0$, el vértice final $S_N$ así como el camino mínimo encontrado, marcado por etiquetas con números del 1 al $m$, donde el vértice $S_0$ tiene la etiqueta 1 y el vértice $S_N$ tiene la etiqueta $m$.

La forma de interactuar con el tablero, es mediante el teclado, donde las teclas de dirección arriba, abajo, izquierda y derecha representan los respectivos movimientos de la celda $S_N$ representada por la oveja, para moverse de manera diagonal se usan las teclas $q$, $w$, $a$ y $s$, que representan los movimientos diagonal a la izquierda-arriba, derecha-arriba, izquierda-abajo y derecha-abajo, respectivamente.

En cuanto a la programación orientada a objetos, se definieron 4 clases: Cell, ASearch, Plot y Program, las cuales se describen a continuación:

- Clase Cell: Contiene información del estado de cada celda o vértice, como lo es, las coordenadas en la matriz $W$, las coordenadas del padre de la celda, los costos $f$, $g$ y $h$.
- Clase ASearch: Clase que implementa el algoritmo $A^*$, haciendo uso de las estructuras de datos mencionadas, las cuales son miembros de la clase. También contiene método para extraer la ruta mínima encontrada.
- Clase Plot: Se encarga de mostrar la ventana con la visualización del tablero así como de la ruta mínima encontrada.
- Clase Program: Cumple una función de intermediario entre la clase ASearch y Plot ya que contiene un método llamado $run()$ el cual se encarga de leer el teclado en espera de que el usuario mueva la celda $S_N$ para así recalcular la ruta mínima llamando al algoritmo $A^*$ y repintando la ruta mínima en el tablero.

## 0.3 Resultados

A contiuación se muestran ejempos de la ejecución del programa con diferentes matrices o tableros, así como diferentes pocisiones iniciales $S_0$ y posiciones finales $S_N$.

Figure 2: Ejemplo 1, posiciones $S_0$ y $S_N$ aleatorias

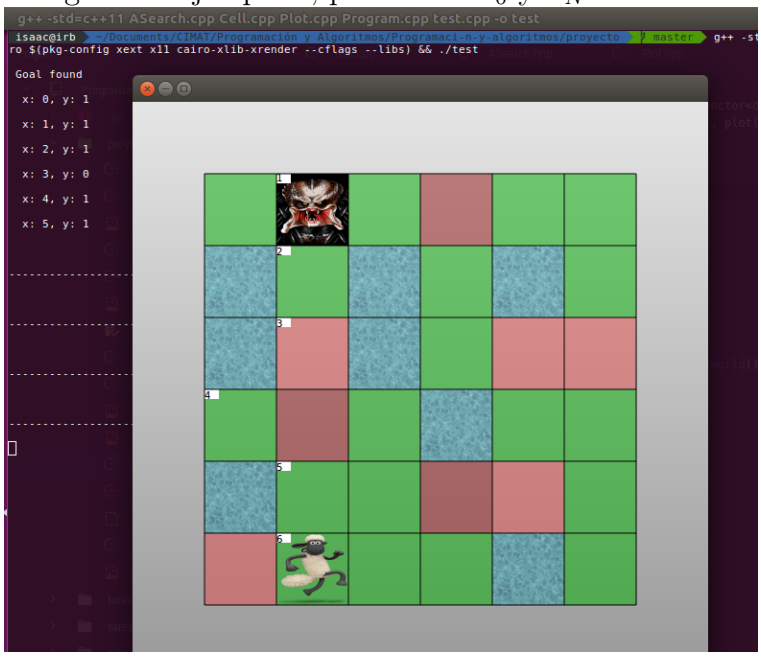



Figure 3: Ejemplo 1, se mueve la oveja una celda a la derecha

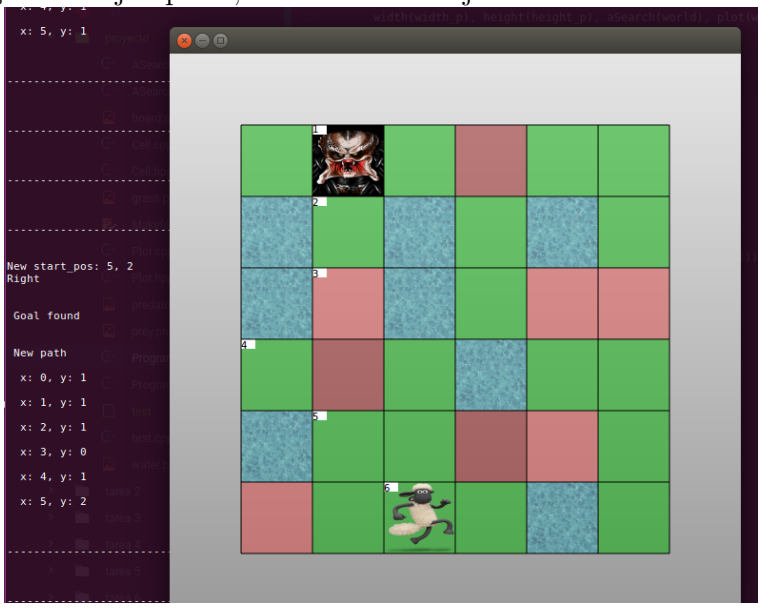
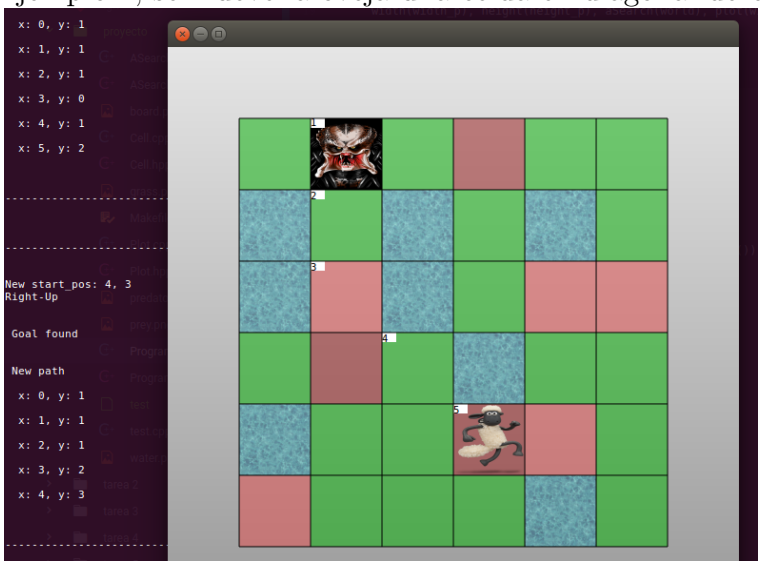


Figure 4: Ejemplo 1, se mueve la oveja una celda en diagonal derecha-arriba




Como se puede observar, cada que se mueve a la oveja (celda $S_N$), se recalcula la ruta mínima y se vuelven a etiquetar las celdas que pertenecen a dicha ruta.

Figure 5: Ejemplo 2, Posición inicial dentro de obstáculo cóncavo, posición final de lado derecho

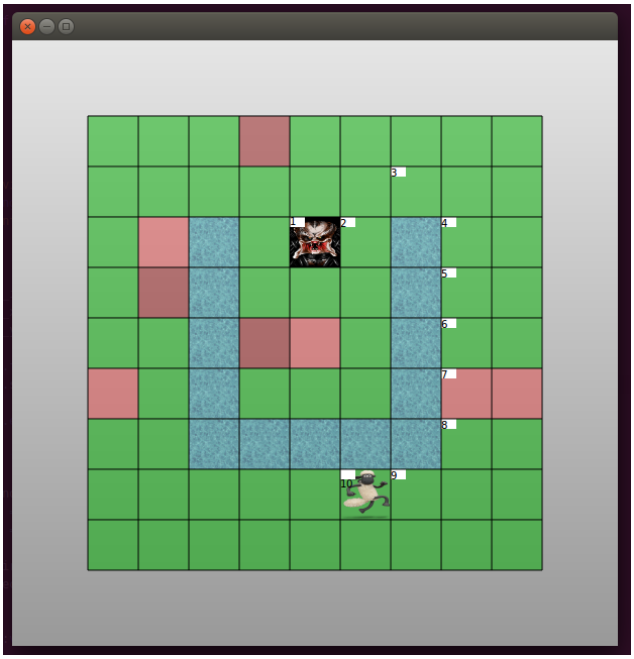
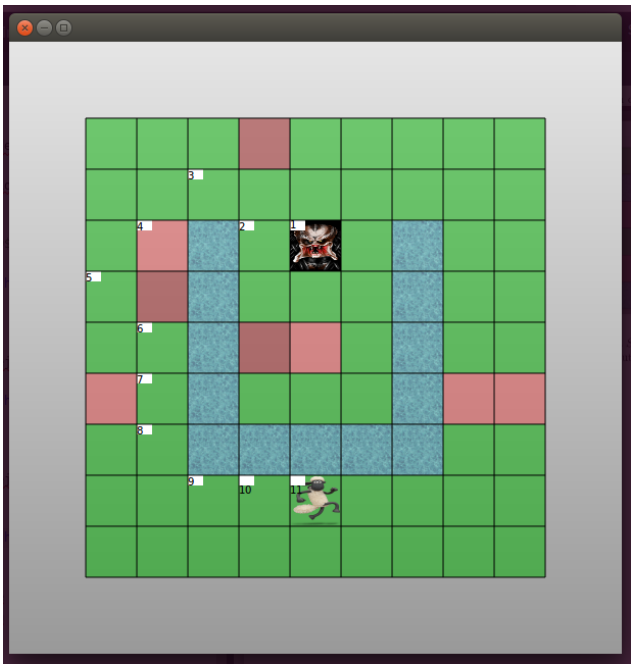


Figure 6: Ejemplo 2, Posición inicial dentro de obstáculo cóncavo, posición final de lado izquierdo



Como se puede observar, en el caso de obstáculos cóncavos, se ignoran los caminos que entran en dichos obstáculos, a diferencia del algoritmo de búsqueda voraz el mejor primero, que primero exploraría las celdas dentro del obstáculo cóncavo ya que, considerando sólamente la distancia o costo para llegar al nodo final, dichas celdan parecen serr más promisorias. En cambio, si también se consideran celdas con un costo bajo hacia el nodo inicial, como en el algoritmo $A^*$, se evitarán dichos problemas.

## 0.4 Conclusión

El hecho de que el algoritmo $A^*$ tome en cuenta un costo compuesto por la suma de $g(x)$ y $h(x)$, hace que haya un compromiso entre exporar nodos con costo bajo respecto al nodo inicial y que se exploten regiones que estén cerca respecto al nodo final. El poder controlar hasta cierto grado este compromiso exploración-explotación puede ser muy importante ya que hay problemas en los que de antemano sabemos ciertos criterios que pueden ayudar a guiar la búsqueda de forma que se exploten y se favorezcan ciertas regiones, en vez de tratar de explorar demasiadas regiones antes de llegar a la solución.

Esto hace que $A^*$ sea un algoritmo bastante flexible. Si quitáramos el costo o heurística $h$ obtendriamos el comportamiento del algoritmo de *Dijkstra*, el cual da prioridad a la exploración sobre la explotación ya que no tiene alguna indicación sobre que región explotar más que la que le indica la función $g(x)$, lo cual tiene la ventaja de que garantiza siemmpre

dar el camino más corto, pero de manera más lenta. En el otro extremo, si se sobreestima la heurística, dando prioridad a $h(x)$, se obtendrá un comportamiento como el de búsqueda voraz el mejor primero, ya que en la suma $f(x) = g(x) + h(x)$, el término $h(x)$ es el que predomina, lo cual tiene la ventaja de que se encontrará una solución de manera muy rápida, aunque no se garantice que sea la más corta

En el mejor de los casos, si se tiene una heurística $h(x)$ que siempre da el valor exacto del costo de ir de un vértice cualquiera $x$ al nodo final $S_N$, el algoritmo $A^*$ será muy rápido y garantizará encontrar el camino o ruta más corta, sin explorar nodos que no forman parte del camino más corto. El problema está en encontrar dicha heurística.

Entonces, de manera general, mientras más pequeño sea el valor de $h(x)$ respecto al costo real de ir de $x$ a $S_N$, más exploración realizará el algoritmo y más lento será, pero de igua forma mayor seguridad dará de asegurar que se encuentra uno de los caminos más cortos. Al contrario, mientras más grande sea $h(x)$, más rápido será el algoritmo y menos exploración hará (y por lo tanto mayor explotación de ciertas regiones) pero a la vez menos seguros podremos estar de que se de con el camino más corto.

Finalmente, también habría que considerar el hacer uso de versiones modificadas de $A^*$ como lo es beam search, en el cual se pone un límite sobre el número de vértices que se pueden considerar como candidatos al mismo tiempo, ignorando aquellos con mayor costo. Otra opción podría ser $IDA^*$ donde se impone un límite sobre el valor de $f(x)$ que se considerará, ignorando aqullos vértices con un valor de $f(x)$ mayor a cierto umbral y ejecutando el algoritmo repetidas veces aumentando el umbral hasta que ya no se obtiene una mejor solución.