Victoria González Chacín – 202320852

Zair Montoya – 202321067

Andrea Dávila –

El problema de “Samus y el laberinto de Zebes” es un problema que puede ser planteado de manera efectiva a través de un grafo y algoritmos de búsqueda en este. De esta forma estructuramos nuestra respuesta siguiendo los siguientes pasos:

1. Entender e identificar las características principales de la problemática.
2. Estructurar y ver las restricciones del grafo a usar.
3. Establecer el algoritmo de búsqueda que usaremos.

Para la primera parte tuvimos en cuenta las siguientes consideraciones en el juego de Metroid:

1. Va a existir un orden en donde la plataforma 0, o la plataforma en donde Samus se encuentra al inicio del juego, será la inicial, y la plataforma n-1 siempre será nuestra plataforma final o destino, o en donde se encontrará Raven Beak.
2. Samus puede saltar en cualquier dirección (para atrás o adelante) de una plataforma a la siguiente.
3. En ciertas plataformas hay poderes que le permite a Samus dar un salto de k plataformas, también en cualquier dirección.
4. Samus tiene la capacidad de teletransportarse en cualquier momento (por lo tanto, en cualquier plataforma), a la plataforma que desee (que consume el número de plataformas por las cuales se mueva de energía).
5. En ciertas plataformas hay robots asesinos, que no pueden ser usadas.

De esta forma empezamos a desarrollar nuestra solución. Aun así, es importante aclarar desde un inicio las distintas formas a las cuales llegamos a plantear una misma problemática, viendo como hacer la siguiente aún más eficiente en cuanto a memoria y tiempo de ejecución. La mayor clarificación que tuvimos en cuenta fue la magnitud de plataformas que puede haber, definida como 2<=n<=10^5. Es por esto por lo que planteamos que el peor caso seria un grafo en donde se tienen 100,000 plataformas, no hay energía para teletransportarse, y no hay saltos o poderes disponibles, si no que Samus únicamente puede moverse para adelante o para atrás en plataformas adyacentes.

**Construcción del grafo:**

De esta forma sabemos que los nodos serán las n plataformas existentes, y las aristas serán las conexiones que se dan cuando Samus puede saltar a las plataformas subsiguientes, cuando puede usar poderes para saltar a la plataforma actual + k saltos en cualquier dirección, y cuando quiera teletransportarse de su posición actual a cualquier otra plataforma. El momento definitivo en donde no van a haber conexiones entre plataformas es cuando hay robots asesinos en una de ellas, ya que esto haría automáticamente que Samus pierda el juego. También es importante aclarar como Samus no empieza directamente en la plataforma numero uno, sino que es necesario que el salte a esta como primer movimiento para así empezar el recorrido de n plataformas, es por esto por lo que el primer paso siempre ser C+. Y de esta posición solo podrá saltar para adelante, no podrá teletransportarse, ni se le asignará ningún poder.

Ya con el grafo construido definimos que el problema que se busca solucionar es encontrar el camino más corto (menor número de movimientos) desde la plataforma 0 hasta la plataforma n-1, siempre que tenga suficiente energía para hacerlo.

Aun así, en un primer planteamiento de la solución se notó como únicamente la creación del grafo resulta en un gran uso de tanto tiempo como de memoria durante la ejecución, por lo que siempre resulta en un grafo altamente conexo y completo. Continuamente se analizo que independientemente del algoritmo que se eligiera este se vería obligado a hacer la búsqueda del camino mínimo a través de cada vértice existente, lo cual, en el peor caso, resultaría en una búsqueda masiva y densa. De esta forma propusimos no tener un método que creara un grafo, sino implementar un único método que uniera la búsqueda con la creación del grafo, de esta forma solo se buscaría y se crearan los vértices en caminos efectivos, descartando muchas posibilidades que no llevan directamente a la respuesta que queremos.

**Algoritmo seleccionado:**

Teniendo en cuenta que tenemos aristas de igual costo y que solo nos interesa el minimizar el número de movimientos (no funcionaría DFS, ya que no garantiza el camino más corto), no el peso de las acciones (por lo que no nos serviría Dijsktra), podemos usar el algoritmo de BFS para explorar primero los caminos más cortos, no atascarnos en caminos más largos innecesarios, evitar ciclos y caminos inválidos (considerando que nuestro grafo puede tener muchos caminos posibles).

**Creación de la solución:**

Se sigue el algoritmo estándar de bfs, iniciando con un set de plataformas visitadas y una cola con una tripleta que contiene la plataforma de inicio (0), la energía que tiene Samus y una lista vacía para ir llenándola con los pasos de cada recorrido. De la misma forma se define las plataformas que tienen robots, y un diccionario que contiene los poderes, organizándose por plataforma de inicio, y numero de saltos. Mientras haya elementos por explorar en la cola, usando un while, tomamos el siguiente nodo a explorar (recordando la tripleta que guarda la información de la plataforma actual, energía restante y el camino recorrido hasta ese punto), haciendo un pop de la cola. Después verificamos que, si llegamos a la última plataforma, o en la que esta Raven Beak, devolvemos el camino, ya que significaría que terminamos el recorrido. Por otro lado, si ya estuvimos en esta plataforma con esa energía exacta (posición actual, energía restante), ignoramos este vértice ya que esa combinación exacta ya fue explorada previamente y volver a procesarla no aportara un camino más corto, solo haciendo el algoritmo más redundante, y marcamos el estado como visitado.

Por consiguiente, empezamos la búsqueda y “construcción” formal del grafo. Recorremos todos los movimientos o acciones posibles desde la plataforma actual.

De esta forma comenzamos con el proceso de construcción para los movimientos de caminar (C+ y C-):

* Ya que Samus puede caminar para adelante y hacia atrás hacemos dos verificaciones; si la siguiente plataforma (plataforma actual+1) no se sale de los rangos (0<n<n-1) y tampoco tiene robots entonces podemos añadir esa conexión a la cola guardando la siguiente información en la tripleta (posición actual + 1, energía restante, acciones), y lo mismo para la plataforma anterior (posición actual-1) con (posición actual - 1, energía restante, acciones).

Seguimos con los poderes de saltos (S- y S+):

* Empezamos verificando que la posición actual este dentro de las plataformas que tienen poderes, se revisa el salto, y si la posición actual + salto (S+) (el destino) está dentro del rango y no tiene robots, añadimos esa conexión a la cola, y lo mismo para posición actual – salto, añadiendo (posición actual - salto, energía restante, acciones + ["S-"]).

Terminamos con los momentos en donde Samus pueda requerir teletransportarse (Tx):

* Recorremos todas las plataformas a las cuales Samus podría teletransportarse (k), las cuales deben de estar en el rango de la energía disponible, por lo que no puede teletransportarse a lugares en donde la energía no alcanza. También recordamos que desde la plataforma 0 Samus no se puede teletransportar, ni se puede hacerlo a esta. De esta forma también se reduce la cantidad de conexiones posibles, ya que mientras se itera por cada conexión se va reduciendo la energía, consecuentemente reduciendo la cantidad de destinos posibles. Se calcula cual seria la plataforma destino, viendo si se movería para adelante o atrás, y si esta no tiene robots y sigue en el rango de posibilidades, entonces se guarda en la cola, restándole a la energía restante la cantidad de saltos que se tendrían que hacer, con esto queda (destino\_tp, energia\_restante - k, acciones + [accion]).

Si terminamos de explorar todo y no llegamos a la plataforma final, no hay camino valido, y devolvemos “NO SE PUEDE”.

Forma, Círculo

El contenido generado por IA puede ser incorrecto.

**Complejidad:**

* Complejidad Temporal

El algoritmo evita construir un grafo completo por adelantado y, en lugar de eso, genera todas las posibles transiciones “al vuelo” dentro de la propia rutina de BFS. Cada estado se identifica mediante la tupla (posición\_actual, energía\_restante), y como hay n + 1 plataformas (de 0 a n) y energía + 1 posibles niveles de energía, el número total de estados únicos es O(n · energía). A partir de cada uno de estos estados se consideran tres tipos de movimientos:

1. Movimientos básicos C+ y C- (hacia la plataforma siguiente o anterior), siempre que no esté bloqueada por un robot, lo que añade hasta 2 transiciones.
2. Saltos especiales S+ y S-, que aplican el “poder” almacenado en la plataforma actual si existe; este poder puede llevar al estado a otra plataforma con un único salto, añadiendo hasta 2 transiciones más.
3. Teletransportaciones T±k, donde k varía de 1 a energía\_restante: por cada valor de k se generan dos transiciones (hacia adelante y hacia atrás), lo que implica hasta 2·energía\_restante posibilidades en cada estado.

En el peor escenario, el grado de cada nodo —es decir, el recuento de transiciones a evaluar— crece linealmente con la energía disponible, es decir, O(energía). Por tanto, al explorar todos los estados con BFS, la complejidad temporal global asciende a O(n · energía²). Esta expresión refleja la combinación de los O(n · energía) posibles estados y las O(energía) transiciones por estado.

* Complejidad Espacial

La memoria consumida se reparte principalmente en tres estructuras:

* El conjunto visitado, que debe poder almacenar cada uno de los O(n · energía) estados para evitar ciclos y redundancias.
* La cola (deque) de la BFS, que en el peor caso puede contener un número comparable de estados simultáneamente mientras avanza la exploración.
* Las listas de acciones que acompañan a cada estado en la cola, pues guardan la secuencia completa de movimientos desde el inicio hasta dicho estado. Aunque en instancias reales estas rutas suelen ser cortas, en el peor caso podrían alcanzar longitud n, y si se almacenan sin optimización el coste adicional puede llegar a O(n²).

En conjunto, estas estructuras implican un uso de espacio de O(n · energía + n²).