Algoritmos y Estructuras de Datos III

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Informe 3

Integrante	LU	Correo electrónico
Jazmín Alvarez Vico	75/15	jazminalvarezvico@gmail.com
Marcelo Pedraza	393/14	marcelopedraza314@gmail.com
Uriel Jonathan Rozenberg	838/12	rozenberguriel@gmail.com
Javier María Cortés Conde Tító	252/15	javiercortescondetito@gmail.com

Reservado para la cdra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

En nuestro problema Brian quiere convertirse en “maestro pokemon” en el menor tiempo posible. Para lograr este objetivo debe ir a todos los “gimnasios” y conquistarlos. Para poder hacerlo, cada gimnasio requiere una cantidad determinada de pociones. Estas pociones pueden obtenerse en las “pokeparadas”. Las pokeparadas sólo pueden visitarse una vez y de cada visita obtenemos tres pociones.

Formalmente, podemos caracterizar nuestras pokeparadas y gimnasios como nodos formando un grafo completo, es decir que existen aristas para unir cualquier par de nodo. nuestras aristas deben tener peso equivalente a la distancia entre dos nodos. Queremos encontrar el camino mínimo que une todos los nodos gimnasios y los nodos pokeparada que sean necesarios para poder conquistar todos los gimnasios.

2. Algoritmo exacto

2.1. Explicación de la solución

Para modelar este problema, creamos dos clases: Nodo y Mochila. La clase Mochila contiene dos enteros “capacidad” el cual corresponde con la capacidad de la entrada (k) y “peso” en el cual se guarda la cantidad de pociones que se tiene en cada momento.
En el Pseudocódigo se llaman a las funciones DameCapacidad y DamePeso, las mismas devuelven estos valores.
La clase Nodo tiene un entero “índice” como identificador de cada nodoy un entero “pociones” en el cual se guardan las pociones que dan las pokeparadas o las que requieren los gimnasios(en este caso el valor es negativo). Además cuenta con un booleano para marcar si ya fue recorrido y dos enteros “x” e “y” para identificar las coordenadas del nodo.
Nuestra solución utiliza la técnica de Backtracking para resolver el problema. Utilizamos variables globales (las cuales estan mencionadas en el pseudocodigo) ya que las necesitamos en todo momento en las funciones de nuestra resolución. De esta forma evitamos copiarlas cada vez, como parámetros de entrada.

Nuestro algoritmo de BT va formando recursivamente todas las soluciones posibles eligiendo gimnasios y pokeparas siempre que esto sea posible. Estas soluciones “parciales” son guardades en una variable que siempre es comparada con una variable de solución “global”. En la misma se guarda siempre el mejor resultado obtenido hasta el momento. El ciclo principal de la función va desde 0 hasta el máximo entre el largo de los vectores de Nodos y pokeparadas. De esta forma vamos probando empezar con todos los nodos posibles. Cada vez que se elige un nodo para continuar se vuelve a llamar a la función BT para seguir construyendo la solución, luego se revierten los cambios para probar las demás soluciones.

En nuestro algoritmo, cortamos una posible solución en el momento que supera la solución global encontrada hasta el momento. De esta forma evitamos seguir ejecutando el algoritmo para una solución que claramente no va a ser la mejor, y así reducimos el tiempo de cómputo.

Actualizamos la solución global, únicamente si logramos recorrer todos los gimnasios. Podemos saber esto ya que guardamos un entero contando los gimnasios recorridos. De esta forma si la solución queda vacia, sabemos que no existe solución para el problema y podemos devolver -1.

Mientras hacemos la lectura de datos, guardamos la cantidad máxima de pociones que requieren los gimnasios y el total de pociones que se requieren para ganar a todos los gimnasios. De esta forma si la capacidad de la mochila es menor al máximo de las pociones que piden los gimnasios o

la cantidad de pociones que se pueden obtener (cantidad de pokeparadas multiplicado por tres) es menor a lo que se necesita para ganar a todos los gimnasios, entonces devolvemos -1 (evitando entrar en el backtracking) ya que el algoritmo no tiene solución.

2.2. Pseudocódigo

variables Globales

```
1: real MinActual ← ∞
2: real MinGlobal ← 0
3: vint RecorridoGlobal
4: vint RecorridoActual
5: vnod PokeParadas
6: vnod Gimnasios
7: int PokeParadasRecorridas
8: int PocionesNecesarias
9: Mochila Moch
10: entero GimRecorridos
```

BT()

```
1: Si MinActual>MinGlobal
2:    cortar
3: finSi
4: Si GimRecorridos = |Gimnasios|
5:    Si MinActual < Min Gobal
6:       MinGlobal ← MinActual
7:       RecorridoGlobal ← RecorridoAtual
8:    finSi
9: finSi
10: Desde i=0 hasta i<Max(|Gimnasios|,|PokeParadas|)
11:    Si i< |Gimnasios|
12:       gim ← Gimnasios[i]
13:       Si puedoIrGim(gim)
14:          Voy(gim)
15:       finSi
16:    finSi
17:    Si i < |PokeParadas|
18:       pp ← Pokeparadas[i]
19:       PokeParadasRecorridas ← PokeParadasRecorridas + 1
20:       Si puedoIrPP(pp)
21:          Voy(pp)
22:       finSi
23:       PokeParadasRecorridas ← PokeParadasRecorridas - 1
24:    finSi
25: finDesde
26: Si RecorridoGlobal = ∅
27:    Devolver -1
28: Si no
29:    Devolver MinGlobal
30: finSi
```

Voy(Nodo n)

```
1: Marco n
2: Si RecorridoActual ≠ ∅
3:    origen ← ultimo de RecorridoActual
4:    minActual ← minActual+ Distancia(origen, p)
5: finSi
6: Agrego n a RecorridoActual
7: Modifico el peso de moch según las pociones n
8: BT()
9: revierto todas las modificaciones
```

Esta función actualiza los valores antes de hacer el llamado recursivo, y luego revierte las modificaciones locales(RecorridoActual, nodos recorridos, minActual).

puedoIrPP(Nodo n)bool

1: NoConsumoDeMas ← Si al entrar a esta pokeparada se tiene que descartar pociones que no nos dejan ganar a todos los gimnasios asigno True, si no False.
2: res ← moch no está llena y n no está recorrido y NoConsumoDeMas

Esto es una poda; calcula si esta pokeparada genera un estado sin solución trivial: las pokeparadas restantes y las pociones en la mochila no alcanzan para ganar los gimnasios que quedan.

PuedoIrGim(Nodo n)

res ← n no está recorrido y DamePeso(moch) $\geq$ -(DamePociones(n))

Simplemente analiza si lo puede vencer.

2.3. Demostración de Correctitud

Para mostrar la correctitud de nnuestro algoritmo queremos ver tre cosas: que el algoritmo termina, que analiza todos los casos posibles, y que deveulve la solución óptima.

- El algoritmo termina: Podemos ver que la cantidad de nodos en nuestro grafo es finita. Además mientras se está buscando la solución, no se puede recorrer dos veces el mismo nodo, entonces no puede recorrer dos veces la misma instancia ni desarrollarse algún ciclo infinito. Las llamadas recurisivas del algoritmo dependen de la cantidad de nodos del grafo. Puesto, que establecimos que esta cantidad era finita, podemos concluir que la cantidad de llamadas lo será también.
- El algoritmo analiza todos los casos: Como explicamos en la sección 2.1, nuestro algoritmo itera sobre todos los nodos, probando empezar por cada uno de ellos siempre y cuando sea posible. Cada vez que se elige un nodo se vuelve a llamar al backtracking para continuar armando la solución y luego se revierten los cambios (se deselige) para poder probar otras soluciones eligiendo los demás nodos. Este proceso se realiza hasta que no queden más nodos para probar.
- El algoritmo devuelve la solución óptima: El algoritmo cuenta con una variable global que guarda la solución mínima encontrada hasta el momento. Cada vez que se encuentra otra posible solución, estas se comparan acutalizando la solución global de ser necesario. Como probamos que el algoritmo analiza todos los casos, podemos garatizar que devolvemos la solución óptima.

2.4. Complejidad y demostración

Sea t la cantidad total de nodos, es decir t=n+m donde n es la cantidad de gimnasios y m la cantidad de pokeparadas. Proponemos que la complejidad de nuestro algoritmo es t! y lo demostraremos a continuación:

Podemos visualizar a nuestro algoritmo como un árbol de soluciones. Podemos ver que en el peor caso, el camino recorre todos los nodos, tanto gimnasios como pokeparadas. para elegir el primer nodo realizamos un ciclo de t iteraciones. de ahi obtenemos el primer nivel del árbol que tiene t hojas. Ahora por cada una de esas hojas tenemos que elegir un nodo entre los t-1 restantes. asique nuestra complejidad asciende a t(t-1). Así hasta llegar a elegir el ultimo nodo, esto nos costará

$$\prod_{i=0}^{t-1} t - i = t \cdot (t-1) \cdot (t-2) \cdot \quad \cdot 1 = t!$$

Entonces nuestra complejidad resulta $\mathcal{O}$(t!)

2.5. Podas y Estrategias

Para nuestro algoritmo implementamos tres podas y las nombramos con letras. Mantendremos esta notación más adelante en la sección de experimentación.

- Poda A: Esta poda se encuentra en la sección 2.2 en el alogoritmo BT en las lineas 1 a 3. Basicamente, mientras se esta buscando una posible solución, nos permite cortar una rama en el momento inmediato que supera la solución mínima encontrada hasta el momento.
- Poda B Esta poda se encuentra en la sección 2.2 en el algoritmo puedoIrPP linea 2. Esta poda permite no recorrer pokeparadas una vez que la mochila esta llena, ya que recorrerlas no traería ningún beneficio ya que recorreríamos más distancia sin ningún beneficio, porque no podemos obtener pociones.
- Poda C Esta poda se encuentra en la sección 2.2 en el algoritmo puedoIrPP linea 1. A diferencia de la poda B esta poda sirve para los casos en los que en la mochila quedan 1 o 2 lugares libres. Entonces Si al desperdiciar esas pociones, ya no se puede ganarle a los gimnasios que quedan por recorrer evita ir a pokeparadas.

Solo encontramos una estrategia: Al iterar, recorremos primero las pokeparadas y después los gimnasios ya que consideramos que por lo general, es más probable que la solución se obtenga de empezar por una pokeparada.

2.6. Experimentación y análisis de resultados

3. Heurística constructiva golosa

El metodo de heurística golosa consiste en elegir siempre la mejor opción posible para continuar y formar la solución total. En este caso el criterio para elegir una mejor opción sería la cercanía. Bus

3.1. Explicación de la solución

Vamos a dividir la solución en dos partes: por un lado, está la elección del primer nodo a visitar; por el otro, la estrategia para visitar todos los gimnasios.

- Elegir el primero: como se explica en el pseudocódigo, primero buscamos empezar en gimnasios triviales; estos son gimnasios de fuerza 0, que no requieren ninguna poción. Si no encuentra ninguno, busca gimnasios “fáciles”; que sólo requieran una pokeparada previa para vencerlos. En caso de existir alguno, empezamos en la pokeparada mas cercana (la estrategia es luego dentro del algoritmo dirigirse a este gimnasio fácil). Si nuevamente estamos sin suerte, empezamos desde una pokeparada cualquiera. En esta implementación seleccionaremos al primero de la lista.
- Una vez elegido el primero, siempre mira lo más cercano: primero analiza si puede ganar el gimnasio más cercano, de ser imposible, se dirige a la pokeparada más cercana. Una vez efectivizado el movimiento, itera hasta que no haya gimnasios para vencer, o no queden pokeparadas para recargar.

3.2. Pseudocódigo

goloso()

```
1: distanciaRecorrida ← 0
2: Recorrido ← ∅
3: ElegirPrimero
4: Mientras Gimnasios ≠ ∅
5:    Si puedo ganarle al gimnasio más cercano
6:       proxLugar=GimMasCercano
7:    Si No quedan pokeparadas
8:       distanciaRecorrida ← -1
9:       Recorrido ← ∅
10:      Cortar mientras
11:   Si No
12:      proxLugar=PokeParadaMasCercana
13:   finSi
14:   moverse(proxLugar)
15: finMientras
16: Devolver distanciaRecorrida y Recorrido
```

Esta es la función principal. Si puede vencer al gimnasio más cercano, lo hace. Si no, intenta ir a la pokeparada más cercana. Si ya no existen pokeparadas, devuelve -1.

moverse(Nodo n)

```
1: Modifico la mochila según las pociones de n
2: Agregar atrás de Recorrido a n
3: distanciaRecorrida + dist(n, posiciónActual)
4: Actualizar posición actual
5: Si n es gimnasio
6:    sacar a n de Gimnasios
7: Si No
8:    sacar a n de pokeParadas
9: finSi
```

Dado un nodo n este algoritmo actualiza las variables posición actual, distanciaRecorrida y Recorrido, y depende del tipo de nodo, lo elimina de la lista correspondiente.

Este algoritmo es una manera de elegir el nodo inicial. Si existe un gimnasio trivial(necesita 0 pociones) lo elijo como nodo inicial, si no, busco un gimnasio que sólo necesite una visita a una pokeparada (3 pociones o menos) y elijo la pokeparada como nodo incial. Si no existe ninguno de estos, elijo una pokeparada al azar como nodo inicial.

3.3. Complejidad

Vamos a demostrar que la complejidad de esta heurística es $\mathcal{O}((n+m)^2)$ donde n es la cantidad de gimnasios y m la cantidad de pokeparadas. Por un lado, vamos a analizar las funciones auxiliares. La función *moverse* usa todas operaciones en $\mathcal{O}(1)$ y no contiene ciclos. Entonces su complejidad es $\mathcal{O}(1)$.

Luego, ElegirPrimero recorre los gimnasios en busca de uno trivial ($\mathcal{O}(n)$), luego recorre nuevamente los gimnasios en busca de uno "fácil", y por cada uno que encuentre busca la pokeparada más cercana ($\mathcal{O}(n * m)$), y por último elige una pokeparada cualquiera ($\mathcal{O}(1)$). En resumen, esta función auxiliar tiene complejidad $\mathcal{O}(n * m)$
Viendo el algoritmo principal vemos que cada iteración de la función toma dos posibilidades: o se mueve a un gimnasio o a una pokeparada. En el peor caso el algoritmo va a hacer $\mathcal{O}(n+m)$ iteraciones. Cada iteración primero calcula el gimnasio más cercano, y luego la pokeparada más cercana. Todas las demas operaciones son $\mathcal{O}(1)$. En conclusión, cada iteración tiene complejidad ($\mathcal{O}(n+m)$). Finalmente vemos que el algoritmo tiene complejidad $\mathcal{O}((n+m)^2)$.

3.4. Experimentación y análisis de resultados

4. Heurística de búsqueda local

4.1. Explicación de la solución

Este algoritmo ordena los gimnasios por fuerza de menor a mayor, y luego "rellena" con las pokeparadas necesarias cada espacio entre gimnasios, de manera que en cada gimnasio se tienen las pociones necesarias para ser derrotado. Luego, define su vecindad de soluciones como todas las soluciones que tengan un intercambio de pokeparadas. Supongamos que se tiene la solución

$P_1, P_2, G_1, P_3, P_4, P_5, G_2$.

Un vecino de esta solución es
$P_4, P_2, G_1, P_3, P_1, P_5, G_2$.

Intercambiando P_1 con P_4

Una vez calculados todos los vecinos posibles, se queda con el mínimo entre el conjunto de vecinos y la solución original.

4.2. Pseudocódigo

4.3. Complejidad

4.4. Experimentación y análisis de resultados

5. Metaheurística

Como metaheurística decidimos implementar un algoritmo "GRASP" La técnica de este tipo de algoritmos consiste en combinar sucecivamente un algoritmo golozo randomizado para explorar nuevos espacios de soluciones y luego aplicar una busqueda local para mejorar cada solución. Al mismo tiempo se van comparando las soluciones y siempre se va eligiendo la mejor posible hasta que el algoritmo finaliza bajo algún criterio.

5.1. Explicación de la solución

Utilizamos como criterio de terminación el tamaño del conjunto de "entradas validas". Este conjunto esta formado por Todos los gimnasios que pidan cero pociones para ser vencidos y todas las pokeparadas. Aplicamos el algoritmo golozo comenzando de cada uno de los nodos de este conjunto.

El algoritmo golozo randomizado es una variación de nuestra heuristica goloza presentada en la sección (). Para explorar el nuevo espacio de soluciones seleccionamos como candidatos al veinte porciento de los nodos por recorrer más cercanos a la posición actual. Luego le asignamos una probabilidad a cada uno de ellos. La misma utiliza el siguiente criterio:

- Si el nodo es gimnasio y tengo suficientes pociones en mi mochila para ir, +30
- Si el nodo es pokeparada y entran las 3 pociones en la mochila, +25
- Si el nodo es pokeparada y entran 2 pociones en la mochila, +15
- Si el nodo es pokeparada y entra sólo 1 pocion en la mochila, +10
- Se le suma a cada nodo bajo criterio de cercanía 10*#MasCercanos - 10* i donde i representa la posición en ese orden.

Guardamos el valor de la sumatoria de los valores asignados a cada nodo y creamos un número aleatorio entre 1 y este valor. El nodo elegido será el nodo con valor mayor más proximo al número aleatorio. Estas acciones se realizan en la función que llamamos en el pseudocódigo "ElegirProximo".

Finalmente aplicamos la función de busqueda local de la sección ().

5.2. Pseudocódigo

GRASP(vnod entradasValidas)

```
1: solu = ∞
2: Desde i=0 hasta |entradasValidas|
3:    GolozoRand(entradasValidas[i])
4:    Blocas(RecorridoGlobal, MinGlobal)
5:    Si MinGlobal<solu
6:       solu=MinGlobal
7:       RecorridoSolu=RecorridoGlobal
8:    finSi
9: fin Desde
```

5.3. Complejidad

5.4. Experimentación y análisis de resultados