Cómputo Evolutivo

Tarea 02 : Problema de la Mochila

Celeste Lorenzo Guerrero : 316162027
Antonio Sebastián Dromundo Escobedo : 419004327
Diego Dozal Magnani : 316032708

March 14, 2023

Esquema de representación de soluciones

`Implementación : src/knapsack.py -> clase Knapsack`

Vamos a considerar un esquema de representación basado en **combinaciones sin repetición**. Consideramos un *item* de la mochila como una tupla `[id, valor, peso]`. Las soluciones se representan por combinaciones de n (la cantidad de *items* generados en base a cada `.txt`) en k con $k <= n$.
Es importante que k en general va a depender de 1 : Nuestra solución inicial generada aleatoriamente y 2 : La solución vecina respecto de la solución actual, las soluciones vecinas pueden tener más elementos que la actual o variar en uno de los elementos. En el apartado de **Descripción de generador de soluciones aleatorias** y **Operador de vecindad** se explica a más detalle.
A partir de este momento vamos a llamar una **solución válida** aquella cuya suma total de pesos es menor o igual a la capacidad total.

- **Espacio de búsqueda** El tamaño de nuestro espacio de búsqueda es

$$C_{(n,k)} = \frac{n!}{k! * (n-k)!}$$

- **Directa o indirecta**
- **Lineal o no lineal**
- **Tipo de mapeo**
  Para esta representación el tipo de mapeo es **muchos a uno** ya que estamos hablando de combinaciones sin repetición por lo que consideramos que las permutaciones `[A,B,C]`, `[B,C,A]`, `[C,B,A]`... son mapeadas a la misma solución.
- **Factibilidad de soluciones**
- **Representación completa**
  En este caso, para cada archivo tenemos los *items* que vamos a considerar y además el ejemplar del problema es **0-1 Knapsack**, por lo que los items no pueden ser divididos. De esta forma, bajo el esquema de representaciones propuesto, si es posible obtener una representación completa.

Ventajas y desventajas del esquema
La ventaja más importante de usar combinaciones cuyo tamaño varía según la ejecución es que provee una **mayor capacidad de exploración** para las soluciones y sus vecindades mientras se puedan seguir agregando items o reemplazando. Sin embargo, al tratarse de combinaciones también disminuye la capacidad de **explotación** y la velocidad de procesamiento es menor al esquema binario.

Descripción de función de evaluación

`Implementación : src/knapsack.py -> Knapsack.get_fitness_sol`

La función de evaluación para una solución S dada bajo el esquema de representación con permutaciones es

$$f(S) = \sum_{i=1}^{n} weight(item_i) - \sum_{j=1}^{k} weight(item_k)$$

con $item_i$ en nuestro conjunto generl de items e $item_k$ en nuestra solución S.De este modo buscamos **minimizar** la pérdida, es decir encontrar una solución S ta que $f(S)$ esté muy cercana a la suma del beneficio total de los items.
La función de evaluación fue determinada en base a lo discutido en el curso el día 23 de Febrero, clase *Tarea 2 - Problema de la mochila* con el profesor.

Descripción de generador de soluciones aleatorias

`Implementación : src/knapsack.py -> Knapsack.generate_random_sol`

La generación de funciones aleatorias simplemente se basa en que cada item de nuestro conjunto total de items tiene una probabilidad de entre 0.5 y 0.75 (a derminar) de ser seleccionado para formar parte de la solución. Las probabilidades tienen ese valor por que inicialmente buscamos que la solución no exceda la capacidad de la *mochila*, por lo que si todos los items tienen una probabilidad de ser elegidos de .5, aproximadamente la solución inicial tendrá la mitad de los items totales.

Función u operador de vecindad

`Implementación : src/knapsack.py -> Knapsack.generate_neighborhood`

Antes de describir el operador de vecindad se define la solución representada bajo el esquema de permutaciones o combinaciones. Si sólo usáramos combinaciones de n en k (un entero fijo) posiblemente estaríamos limitando la capacidad total de nuestra *mochila* con k, nuestras soluciones podrían no acercarse al óptimo global, ya que al limitar las soluciones a un tamaño k podríamos estancarnos en un mínimo local de tamaño k y, por más que intercambiáramos elementos, podríamos tener el caso de no aprovechar la capacidad completa.
Es por esto que la **solución inicial** si tiene un tamaño k inicial que no está fijo, si no que depende de las probabilidades de los mejores items.
Dada una solución $S = [item_i = (id_i, p_i, w_i), ..., item_j = (id_j, p_j, w_j)]$, la vecindad **no necesariamente válida** es $\{S' | S' = S.add(item_k)\}$ con $item_k \in S - U$ siendo U nuestro conjunto total de items e $item_k$ un elemento seleccionado con probabilidad uniforme. **Para tratar con los vecinos que no son soluciones válidas**, la vecindad son aquellas soluciones con la misma cantidad de elementos pero que difieren de un elemento, el cual es elegido de igual forma con probabilidad uniforme. Lo anterior permite que haya una probabilidad de vecinos que estén muy cerca de aprovechar la capacidad máxima y cuyo valor objetivo sea mejor.

Detalles sobre implementación de recocido simulado

`Implementación : src/knapsack.py -> SimAnnealing`

El flujo del programa completo es el siguiente :
La implementación del algoritmo se ejecuta en el archivo `src/excecutable.py`, el cual realiza una lectura de los *data/*.txt* y a partir de esos archivos genera instancias de ejemplares de Knapsack por medio de la clase `src/Knapsack`, estos ejemplares se pasan como argumento a instancias de `src/simulated_annealing.py : SimAnnealing` y finalmente se ejecutan 10 repeticiones de cada ejemplar en `src/excecutable.py` para obtener los datos.

- **Estrategia de selección**
  Utilizamos mayor descenso y que al obtener la vecindad de tamaño `epsilon` dada una solución S, tomamos el mejor vecino S' tal que $f(S') <= f(S)$, recordemos que buscamos minimizar la función de evaluación [1].

- **Descripción de esquema de enfriamiento**
  Utilizamos el **enfriamiento por decremento lento** $T_k = \frac{T_k}{1+\beta T_k}$. Consideramos $\beta = 0.01$ para **todas las ejecuciones** según las diapositivas *07_RecocidoSimulado.pdf* [2].

Registro de ejeuciones con Recocido Simulado, se anexan tablas generadas con la biblioteca `pandas` :
Es importante mencionar que los parámetros para todos los ejemplares fueron :

Temperatura : 20

Iteraciones : 3000

Tamaño de Vecindad: $\epsilon = 6$

Constante de decremento de temperatura $\beta = 0.01$

En cada tabla el `No.Iteració` se refiere a la repetición, e total para cada ejemplar se corrieron 10 repeticiones cada una con 3000 iteracioness.

Ejemplar /data/ejeL1n10.txt
Maximo beneficio : 412

	No.Iteracion	Mejor	Promedio	Peor
0	0	385	344.347000	261
1	1	395	366.977667	249
2	2	395	371.024333	190
3	3	395	367.841333	266
4	4	395	369.783333	263
5	5	395	364.559333	141
6	6	395	366.197667	187
7	7	395	371.931333	276
8	8	395	371.259333	216
9	9	403	392.029333	323

Ejemplar /data/ejeL10n20.txt
Maximo beneficio : 1086

	No.Iteracion	Mejor	Promedio	Peor
0	0	1063	1038.055667	594
1	1	1063	1038.552333	631
2	2	1063	1038.569667	569
3	3	1063	1038.874333	910
4	4	1063	1038.183333	608
5	5	1063	1039.261000	546
6	6	1063	1038.071333	617
7	7	1063	1038.529333	647
8	8	1063	1037.972333	695
9	9	1063	1038.580667	572

Ejemplar /data/ejeL14n45.txt
Maximo beneficio : 2108

	No.Iteracion	Mejor	Promedio	Peor
0	0	2076	2008.805000	1117
1	1	2076	1997.576000	785
2	2	2075	1995.956333	1057
3	3	2071	2004.371000	1194
4	4	2076	1999.539667	841
5	5	2076	2004.530333	1342
6	6	2076	2003.980667	1181
7	7	2076	2002.018000	1078
8	8	2076	2008.254667	936
9	9	2076	2008.489667	920

Ejemplar /data/ejeknapPI_13_100_1000_18.txt
Maximo beneficio : 101296

	No.Iteracion	Mejor	Promedio	Peor
0	0	69806	67128.199333	52666
1	1	73868	71165.768000	54950
2	2	68184	64960.422000	49548
3	3	63338	60444.572000	45758
4	4	68436	65056.784667	51870
5	5	75472	71909.598000	53716
6	6	68794	66039.596000	47470
7	7	70112	67046.848667	50672
8	8	69978	67042.942000	49944
9	9	79390	76532.228000	58908

Ejemplar /data/ejeknapPI_3_200_1000_14.txt
Maximo beneficio : 123540

	No.Iteracion	Mejor	Promedio	Peor
0	0	76920	74703.915333	58964
1	1	88495	85975.109000	66278
2	2	83684	80773.702333	61621
3	3	84919	82307.647333	63024
4	4	77892	75231.202667	53652
5	5	82378	79655.450000	57493
6	6	85118	82265.820333	64252
7	7	85491	82971.301333	63472
8	8	83898	81009.239333	59661
9	9	88213	85525.596333	68195

Ejemplar /data/eje1n1000.txt
Maximo beneficio : 998946925614

	No.Iteracion	Mejor	Promedio	Peor
0	0	936154087739	9.105996e+11	544491993497
1	1	940410900622	9.109729e+11	497892431045
2	2	945551089020	9.172351e+11	500109463096
3	3	951061372008	9.258634e+11	536677433879
4	4	939109307887	9.101974e+11	511873806743
5	5	937091588402	9.074451e+11	557385619717
6	6	926736555947	8.964334e+11	492728712411
7	7	940464963876	9.151645e+11	513589025924
8	8	939895558111	9.180414e+11	553214932197
9	9	933054150058	9.028622e+11	498506430085

Pesudocódigo

Para el esquema representativo del **0-1 Knapsack Problem** creamos una clase `src/knapsack.py` con los siguientes métodos :

Algorithm 1: `Generate Random Solution`

Data: *Items*
Result: *Solution*
Solution ←lista vacía
for $i \leftarrow 1$ *to* $|Items|$ **do**
  lanzar una moneda *[r]*nd-choice* **if** *"sol"* **then**
    Solution.$add(item_i \in Items)$
  end
end
Returns : Solution

Algorithm 2: `Fitness`

Data: $Solution, Items$
Result: $Fitness$
$max\_benefit \leftarrow 0$
for $i \leftarrow 1$ *to* $|Items|$ **do**
  $max\_benefit \mathrel{+}= Items[i].benefit$
end
$sol\_benefit \leftarrow 0$
for $j \leftarrow 1$ *to* $|Solution|$ **do**
  $sol\_benefit \mathrel{+}= Solution[j].benefit$
end
Returns : $max\_benefit$ - $sol\_benefits$

Algorithm 3: `Get Neighbor`

Data: $Solution, Items, Capacity$
Result: $Neighbor$
$diff$ ←Items - Solution
$new\_item \leftarrow$ `RandomItemFrom(`$diff$`)`
if $Solution.weight < Capacity$ **then**
  $Neighbor$ =$Solution.append(new\_item)$
  Returns : $Neighbor$
end
else
  $Neighbor = Solution.remove($`RandomItemFrom(`$Solution$`)`$)$
  $Neighbor = Solution.append(new\_item)$
  Returns : $Neighbor$
end

Algorithm 4: `Generate Neighborhood`

Data: $Solution, Items, Capacity, \varepsilon$
Result: $Neighborhood$

```
Neighborhood ← []
for i in ε do
    Neighborhood.append(GetNeighbor(Solution, Items, Capacity))
end
Returns : Neighborhood
```

Para el **Recocido Simulado** usamos la clase de `knapsack.py` en una nueva clase `src/simulated`$_a nnealing.py.Cuandonosrefiramo$

Algorithm 5: `Further Decline`

Data: $Knapsack, Solution$
Result: $Best\_Neighbor$

```
Neighborhoo ←Knapsack.GenerateNeighborhood(Solution, Knapsack.Items, Knapsack.Capacity, ε)
for i in |Neighborhoo| do
    if Neighborhoo[i].HasTheBestFitness then
        Returns : Neighborhoos[i]
    end
end
```

Algorithm 6: `Execute Simulated Annealing`

Data: $Knapsack$, $Max\_Iterations$, $CoolingBeta$, $Temperature$
Result: $Best\_Solution$

```
current_solution ←Knapsack.GenerateRandomSolution
for current_iteration ¡ Max_Iterations do
    candidate_solution = Knapsack.Further Decline(current_solution)
    if candidate_solution.Fitness ¡= current_solution.Fitness then
        current_solution ←candidate_ solution.Fitness
    end
    else
        if randomNumber(0,1) < accepting_ Proba then
            current_solution ←candidate_ solution.Fitness
        end
    end
    Temperature ←Temperature∗CoolingBeta
end
Returns: current_solution
```

References

[1] M. en C. Oscar Hernández Constantino. Cómputo evolutivo: Metaheurísticas de trayectoria, 2023. Accessed February 9, 2023. ://drive.google.com/file/d/1v_QJSyTIjFTj97RBx2zmdR-itV8ofQwz/view.

[2] M. en C. Oscar Hernández Constantino. Cómputo evolutivo: Recocido simulado, 2023. Accessed February 7, 2023. https://drive.google.com/file/d/1vgUr5fHPQY_x_PKy9Mr_s5lDVS3egZd-/view.