ALNS e Simulated Annealing

Riallacciandosi al capitolo 1, si descrive l'algoritmo *adaptive large neighborhood search* in relazione alla *simulated annealing*. Lo si può suddividere in tre parti: inizializzazione (1-5), algoritmo ALNS (6-9) e algoritmo SA (10-21). Nella *simulated annealing* gli elementi più importanti sono la temperatura T, il *cooling rate* e una legge di decremento della temperatura (in questo caso si adotta una legge lineare). L'algoritmo ALNS si preoccupa di **costruire la soluzione**, mentre la SA ha il compito di **valutarla**.

Algoritmo 3 Adaptive large neighborhood search & simulated annealing

input: un set di operatori di distruzione D, un set di operatori di riparazione I, inizializzazione delle costanti P_{init} e cooling rate h
output: X_{best}
1: Genera soluzione iniziale mediante approccio greedy o casualmente
2: Inizializza la probabilità P^t_d per ogni operatore di distruzione d in D e la probabilità P^t_i per ogni operatore di riparazione i in I
3: Sia T la temperatura
4: Sia $X_{current} \leftarrow X_{best} \leftarrow X_{init}$
5: **repeat**
6: Seleziona un operatore di distruzione d^* in D con probabilità P^t_i
7: Sia X^*_{new} la soluzione ottenuta applicando l'operatore d^* a $X_{current}$
8: Seleziona un operatore di riparazione i^* in I con probabilità P^t_i
9: Sia X_{new} la nuova soluzione ottenuta applicando l'operatore i^* a X^*_{new}
10: **if** c (X_{new}) < c ($X_{current}$) **then**
11: $X_{current} \leftarrow X_{new}$
12: Aggiorna le probabilità usando la procedura adattiva di regolazione del peso
13: **else**
14: Sia $\nu \leftarrow e^{-(c(Xnew)-c(Xcurrent))/T}$
15: Genera un numero random ε in [0, 1]
16: **if** ε < ν **then**
17: $X_{current} \leftarrow X_{new}$
18: Aggiorna le probabilità usando la procedura adattiva di regolazione del peso
19: **if** c ($X_{current}$) < c (X_{best}) **then**
20: $X_{best} \leftarrow X_{current}$
21: $T \leftarrow h\ T$
22: **until** T raggiunge valore soglia

Lo pseudocodice appena descritto è relativo ad un problema di minimizzazione. In relazione ai capitoli precedenti, si adatta l'algoritmo al *knapsack problem* che però è un problema di massimizzazione. In questo caso, si vuole massimizzare il valore degli oggetti all'interno dello zaino tenendo conto della capienza dello stesso e quindi si deve considerare nell'algoritmo che si tratti di un problema di massimizzazione.

Algoritmo 3.1 ALNS & SA per il knapsack problem

input: un set di operatori di distruzione D, un set di operatori di riparazione I, inizializzazione delle costanti P_{init} e cooling rate h

output: X_{best}

1: Genera soluzione iniziale casualmente (inserire nello zaino degli oggetti scelti a caso)
2: Inizializza la probabilità P^t_d per ogni operatore di distruzione d in D e la probabilità P^t_i per ogni operatore di riparazione i in I (modalità di rimozione e inserimento nello zaino)
3: Sia T la temperatura
4: Sia $X_{current} \leftarrow X_{best} \leftarrow X_{init}$
5: **repeat**
6: Seleziona un operatore di distruzione d^* in D con probabilità P^t_i
7: Sia X_{new} la soluzione ottenuta applicando l'operatore d^* a $X_{current}$
8: Seleziona un operatore di riparazione i^* in I con probabilità P^t_i
9: Sia X_{new2} la nuova soluzione ottenuta applicando l'operatore i^* a X_{new}
10: **if** c (X_{new2}) > c ($X_{current}$) **then**
11: $X_{current} \leftarrow X_{new2}$
12: Aggiorna le probabilità usando la procedura adattiva di regolazione del peso
13: **else**
14: Sia $\nu \leftarrow e^{-(c(Xcurrent)-c(Xnew2))/T}$
15: Genera un numero random ε in [0, 1]
16: **if** ε < ν **then**
17: $X_{current} \leftarrow X_{new2}$
18: Aggiorna le probabilità usando la procedura adattiva di regolazione del peso
19: **if** c ($X_{current}$) > c (X_{best}) **then**
20: $X_{best} \leftarrow X_{current}$
21: $T \leftarrow h\ T$
22: **until** T scende al di sotto del valore soglia

Segue l'implementazione in Python.

Metodi

Sono stati definiti i seguenti metodi:

- **getWeightKnapsack(X):** calcola il peso totale della soluzione X passata in input

```
def getWeightKnapsack(X):
    w = 0
    for t in X:
        w += t[0]
    return w
```

- **getValueKnapsack(X):** calcola il valore totale della soluzione X passata in input

```
def getValueKnapsack(X):
    v = 0
    for t in X:
        v += t[1]
    return v
```

- **checkAvailable(w, c):** passando in input il peso totale w della soluzione attuale e la capacità c dello zaino, verifica se è disponibile un oggetto che al suo inserimento non superi la capacità dello zaino

```
def checkAvailable(w,c):
    global treasures
    for t in treasures:
        if t[0]+w<=c:
            return True
    return False
```

- **getTreasureOk(w, c):** restituisce l'oggetto che può essere inserito nella soluzione corrente con peso totale w, rispettando la capacità c

```
def getTreasureOk(w,c):
    global treasures
    for t in treasures:
        if t[0]+w<=c:
            return t
    return
```

- **initial_solution():** genera la soluzione iniziale

```
def initial_solution():
    global treasures, c
    knapsack = []
    check = [0, 0, 0, 0]
    while(getWeightKnapsack(knapsack) != c and
sum(check)!=len(treasures)):
        value = randint(0, len(treasures)-1)
        if check[value]!=1 and
treasures[value][0]+getWeightKnapsack(knapsack) <= c:
            knapsack.insert(0,treasures[value])
        check[value]=1
    for t in (knapsack): treasures.remove(t)
    return knapsack
```

Il metodo sceglie casualmente con l'ausilio del metodo randint della libreria random, oggetti la cui somma dei loro pesi non supera la capacità dello zaino e li inserisce. Il vettore check tiene conto degli oggetti che ha provato ad inserire all'interno dello zaino in modo da non cercare di reinserirli. Il metodo rimane in esecuzione fino a quando il peso degli oggetti inseriti raggiungerà la capacità o se ha provato ad inserire tutti gli oggetti disponibili da rubare.

- **destroy(op_destr):** applica l'operazione di distruzione op_destr alla soluzione corrente. Il metodo destroy è quindi suddiviso in quattro parti, ovvero pari al numero di operazioni di distruzione che sono stati utilizzati per il problema. si lavora su un vettore copia (l) in quanto la restituzione di Xcurrent comporterebbe problemi di aliasing.

  op_destr = 0

```
def destroy(op_destr):
    global Xcurrent, treasures
    l = Xcurrent.copy()
    match op_destr:
        case 0:
            if(len(Xcurrent)==0): return l
            max = l[0]
            for t in l:
                if t[0] > max[0]:
                    max = t
                elif t[0] == max[0]:
                    if t[1] < max[1]:
                        max = t

            l.remove(max)
            treasures.append(max)
            return l
```

Il primo operatore di distruzione elimina dalla soluzione corrente l'oggetto che ha il peso più grande e, una volta rimosso, lo aggiunge tra gli oggetti disponibili all'inserimento (in treasures).

op_destr = 1

```
case 1:
    if(len(Xcurrent)==0): return l
    min = l[0]
    for t in l:
        if t[1] < min[1]:
            min = t
        elif t[1] == min[1]:
            if t[0] > min[0]:
                min = t

    l.remove(min)
    treasures.append(min)
    return l
```

Il secondo operatore di distruzione elimina dalla soluzione corrente l'oggetto che ha valore minimo e lo aggiunge in treasures (in quanto non sta più nello zaino).

op_destr = 2

```
case 2:
    if(len(Xcurrent)==0): return l
    max = l[0]
    for t in l:
        if t[0]/t[1] > max[0]/max[1]:
            max = t
        elif t[0]/t[1] == max[0]/max[1]:
            if t[0] > max[0]:
                max = t

    l.remove(max)
    treasures.append(max)
    return l
```

Il terzo operatore di distruzione elimina l'oggetto il cui rapporto tra il suo peso e il suo valore è massimo. Rimuovendolo dallo zaino, viene inserito tra gli oggetti disponibili per l'inserimento.

op_destr = 3

```
case 3:
    if (len(Xcurrent)==0): return l
    if len(l)-1 >= 2: n = randint(2, len(l)-1)
    else: n = 1
    todelete=random.sample(range(0,len(l)),n)
    for i in sorted(todelete, reverse = True):
        treasures.append(l[i])
    for fd in sorted(todelete, reverse = True):
        del l[fd]
    return l
```

Il quarto operatore di distruzione è molto utile per ritornare ad una soluzione più remota, in modo da evitare di rimanere bloccati inserendo e rimuovendo un oggetto alla volta (è probabile che sia sempre lo stesso) soprattutto se i dati sono pochi. Quindi, questo operatore di distruzione elimina dalla soluzione corrente un numero casuale degli oggetti (da 2 a tutti gli oggetti presenti) e li inserisce tra gli oggetti disponibili per un eventuale inserimento successivo.
Per evitare errori se si richiede di eseguire un operatore inesistente, il metodo restituisce la copia della soluzione corrente.

```
case _:
    return l
```

- **repair(op_rep, X):** applica l'operazione di riparazione op_rep alla soluzione X

Anche in questo caso, il numero di operatori di riparazione sono quattro.

op_rep = 0

```
def repair(op_rep, X):
    global treasures, c
    l = X.copy()
    match op_rep:
        case 0:
            if len(treasures)==0 or
checkAvailable(getWeightKnapsack(l),c) == False: return l
            min = getTreasureOk(getWeightKnapsack(l),c)
            for t in treasures:
                if t[0] < min[0] and getWeightKnapsack(l)+t[0] <= c :
                    min = t
                elif t[0] == min[0] and getWeightKnapsack(l)+t[0] <= c:
                    if t[1] > min[1]:
                        min = t

            l.append(min)
            treasures.remove(min)
            return l
```

Il primo operatore di riparazione inserisce nella soluzione X (passata in *input* al metodo) l'oggetto che il peso più piccolo tra quelli disponibili all'inserimento. Ci si avvale del metodo checkAvailable() per verificare se esiste un oggetto che rispetti la capienza dello zaino una volta inserito e in caso affermativo lo si sceglie come oggetto con peso minimo candidato utilizzando il metodo getTreasureOk(). Viene aggiunto l'oggetto di peso minimo nella soluzione X e lo si rimuove dagli oggetti disponibili.

op_rep = 1

```
case 1: # valore max

    if len(treasures)==0 or checkAvailable(getWeightKnapsack(l),c) ==
False: return l
    max = getTreasureOk(getWeightKnapsack(l),c)
    for t in treasures:
        if t[1] > max[1] and getWeightKnapsack(l)+t[0] <= c :
            max = t
        elif t[1]==max[1] and getWeightKnapsack(l)+t[0] <= c :
            if t[0] < max[0]:
                max = t

    l.append(max)
    treasures.remove(max)
    return l
```

Il secondo operatore di riparazione inserisce nella soluzione X l'oggetto con il valore massimo tra quelli disponibili, facendo i dovuti controlli come con l'operatore precedente.

op_rep = 2

```
case 2:

    if len(treasures)==0 or checkAvailable(getWeightKnapsack(l),c) ==
False: return l
    min = getTreasureOk(getWeightKnapsack(l),c)
    for t in treasures:
        if t[0]/t[1] < min[0]/min[1] and getWeightKnapsack(l)+t[0] <=
c:
            min = t
        elif t[0]/t[1] == min[0]/min[1] and getWeightKnapsack(l)+t[0]
<= c:
            if t[0] < min[0]:
                min = t

    l.append(min)
    treasures.remove(min)
    return l
```

Il terzo operatore di riparazione inserisce nella soluzione X l'oggetto che ha il rapporto peso/valore minimo.

op_rep = 3

```
case 3:

    if len(treasures)==0 or checkAvailable(getWeightKnapsack(l),c) ==
False: return l
    if len(treasures)-1 >= 2: n = randint(2, len(treasures)-1)
    else: n = 1
    for i in range(n):
        if(checkAvailable(getWeightKnapsack(l),c) == False):
            return l
        else:

            shuffle(treasures)
            t = getTreasureOk(getWeightKnapsack(l),c)
            l.append(t)
            treasures.remove(t)
    return l
```

Il quarto e ultimo operatore di riparazione ha la stessa utilità del quarto operatore di distruzione: permette di aggiungere da 2 a più oggetti che rispettino la capienza dello zaino in modo casuale.

```
case _:
    return l
```

Inizializzazione

```
allTreasures = [[1,20], [4, 100], [3, 30], [7, 500]]
treasures = [[1,20], [4, 100], [3, 30], [7, 500]]
P_d = [0.25,0.25,0.25,0.25] #probabilità per ogni operatore di
distruzione
P_i = [0.25,0.25,0.25,0.25] #probabilità per ogni operatore di
riparazione
T = 100 #temperatura
h = 0.05 #cooling rate
c = 8 #capacità dello zaino
inc = 0.05
flag = True
D = [0, 1, 2, 3]
I = [0, 1, 2, 3]
Xinit = []
Xcurrent = []
Xbest = []
```

Nell'inizializzazione:

- **allTreasures:** è un vettore in cui sono presenti tutti glgli oggetti utilizzati nel problema;
- **treasures:** è un vettore contenente gli oggetti che non sono nello zaino;
- **P_d:** è il vettore delle probabilità di ogni operatore di distruzione;
- **P_i:** è il vettore delle probabilità di ogni operatore di riparazione;
- **T:** è la temperatura;
- **h:** è il *cooling rate*;
- **c:** è la capacità dello zaino;
- **inc:** è l'incremento della probabilità dell'i-esimo operatore che ha portato all'aggiornamento della soluzione corrente, in quanto migliore e decremento delle restanti probabilità;
- **flag:** è il booleano utilizzato per incrementare/decrementare le probabilità solo se non supera/scende sopra/sotto 1/0;
- **D:** è il vettore degli operatori di distruzione;
- **I:** è il vettore degli operatori di riparazione;
- **Xinit:** è la soluzione iniziale;
- **Xcurrent:** è la soluzione corrente;
- **Xbest:** è la soluzione ottima.

Segue l'associazione del codice alla restante parte dell'Algoritmo 3.1.

6: Seleziona un operatore di distruzione d^* in D con probabilità P^t_d

7: Sia X_{new} la soluzione ottenuta applicando l'operatore d^* a $X_{current}$

```
d = random.choices(
        D,
        P_d,
        k=1
)[0]

Xnew = destroy(d)
```

8: Seleziona un operatore di riparazione i^* in I con probabilità P^t_i

9: Sia X_{new2} la nuova soluzione ottenuta applicando l'operatore i^* a X_{new}

```
i = random.choices(
    I,
    P_i,
    k=1
)[0]

Xnew2 = repair(i,Xnew)
```

10: **if** c (X_{new2}) > c ($X_{current}$) **then**

11: $X_{current} \leftarrow X_{new2}$

12: Aggiorna le probabilità usando la procedura adattiva di regolazione del peso

```
if (getValueKnapsack(Xnew2) > getValueKnapsack(Xcurrent)):
    Xcurrent = Xnew2.copy()

    for pd in range(len(P_d)):
        if pd != d and P_d[pd]-inc < 0 or P_d[d]+inc > 1 :
            flag = False

    for pi in range(len(P_i)):
        if pi != i and P_i[pi]-inc < 0 or P_i[i]+inc > 1:
            flag = False

    #if flag == False: inc = 0.01

    if flag:
        P_d[d]+=inc
        P_d[d] = round(P_d[d],1)
        for pd in range(len(P_d)):
            if pd != d:
                P_d[pd]-=inc
                P_d[pd]=round(P_d[pd],1)

        P_i[i]+=inc
        P_i[i] = round(P_i[i],1)
        for pi in range(len(P_i)):
            if pi != i:
                P_i[pi]-=inc
                P_i[pi]=round(P_i[pi],1)
```

13: **else**
14: Sia $\nu \leftarrow e^{-(c(X_{current})-c(X_{new2}))/T}$
15: Genera un numero random ε in [0, 1]
16: **if** $\varepsilon < \nu$ **then**
17: $X_{current} \leftarrow X_{new2}$
18: Aggiorna le probabilità usando la procedura adattiva di regolazione del peso

```
else:
    n = pow(math.e, -(getValueKnapsack(Xcurrent) -
getValueKnapsack(Xnew2))  / T)
    e = random.random()
    if e < n:
        Xcurrent=Xnew2.copy()

        for pd in range(len(P_d)):
            if pd != d and P_d[pd]-inc < 0 or P_d[d]+inc > 1:
                flag = False

        for pi in range(len(P_i)):
            if pi != i and P_i[pi]-inc < 0 or P_i[i]+inc > 1:
                flag = False

        if flag:
            P_d[d]+=inc
            P_d[d] = round(P_d[d],1)
            for pd in range(len(P_d)):
                if pd != d:
                    P_d[pd]-=inc
                    P_d[pd] = round(P_d[pd],1)

            P_i[i]+=inc
            P_i[i] = round(P_i[i],1)
            for pi in range(len(P_i)):
                if pi != i:
                    P_i[pi]-=inc
                    P_i[pi] = round(P_i[pi],1)
    else:
        treasures = allTreasures.copy()
        for t in Xcurrent: treasures.remove(t)
```

19: **if** $c(X_{current}) > c(X_{best})$ **then**
20: $X_{best} \leftarrow X_{current}$
21: $T \leftarrow h\,T$

```
if (getValueKnapsack(Xcurrent) > getValueKnapsack(Xbest)):
    Xbest=Xcurrent.copy()

T = h * T
```

Tutta quest'ultima parte dell'Algoritmo (i.e. linee 6-21) è racchiuso nel ciclo **repeat-until** con condizione T > soglia.

A titolo illustrativo, segue qualche output del programma.

Output 1:

Xinitt:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[3, 30], [1, 20], [4, 100]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

…

Output 2:

Xinit:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

Xbest:[[1, 20], [7, 500]]

…

Output 3:

Xinit:[[7, 500]]

Xbest:[[7, 500]]

Xbest:[[7, 500]]

Xbest:[[7, 500]]

Xbest:[[7, 500]]

Xbest:[[7, 500]]

Xbest:[[7, 500]]

Xbest:[[7, 500]]

Xbest:[[7, 500], [1, 20]]

Xbest:[[7, 500], [1, 20]]

Xbest:[[7, 500], [1, 20]]

Xbest:[[7, 500], [1, 20]]

Xbest:[[7, 500], [1, 20]]

…

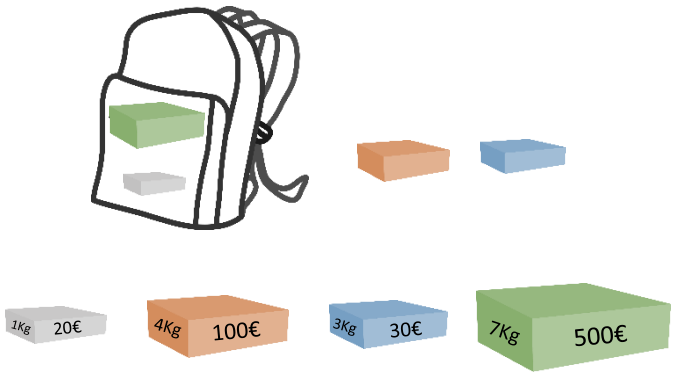

Figura 3.1 - Rappresentazione della soluzione ottima restituita dall'Algoritmo 3.1

Come si può notare dalla Figura 3.1, l'algoritmo comprende come la soluzione migliore sia [[7,500],[1,20]].