Sistemas Inteligentes 2020-1

Parcial 1

- Estudiante: Josue Peña Atencio
- Fecha: Septiembre 18 2020

Problema:

Organizar una secuencia S de longitud N de enteros únicos que se encuentran en el rango [X, Y).

In [226…
```
import numpy as np
import random
from math import exp

X = -100
Y = 101
N = 15
S = random.sample(range(X, Y), N) # random.sample escoge N elementos únicos del rango
```

In [248…
```
S # Este S es exactamente el mismo usado en ambas técnicas. No se modifica en ningún momento.
```

Out[248… `[57, 61, -65, 38, 3, 23, 93, -25, -31, -99, -91, -40, -18, 8, -77]`

1. Enfriamiento Simulado

En el enfriamiento simulado, se tiene una sola lista de variables, y se escoge una de estas al igual que un valor aleatorio para realizar la nueva asignación. En este caso, se tienen que escoger dos variables (indices) de forma aleatoria para hacer el intercambio de sus valores. Un vecino es una configuración o permutación de la secuencia actual que se puede hacer en un paso.

In [159…
```
# Función heuristica
# Calcula la cantidad de conflictos actuales entre los valores de las variables
# state: secuencia de numeros enteros
def h(state):
    n = len(state)
    conflicts = 0
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] > state[j]:
                conflicts += 1
    return conflicts
```

In [160…
```
# Selecciona dos indices aleatoriamente e intercambia los valores de forma aleatoria
# state: secuencia de numeros enteros
def get_new_assignment(state):
    n = len(state)
    ind1 = random.randint(0, n-1)
    ind2 = ind1
    while ind1 == ind2:
        ind2 = random.randint(0, n-1)
    state[ind1], state[ind2] = state[ind2], state[ind1] # Hacemos un swap de las dos posiciones aleatorias
    return state
```

In [240…
```
# Funcion general de enfriamiento simulado
# S_current: secuencia de enteros de entrada
# T: valor inicial de parametro de enfriamiento
# nloops: cantidad de iteraciones que se realizan antes de reducir Ti
# R: valor en el que T decrece
def simulated_annealing(S_current, T, nloops, R):
    found = False
    cc = 0
    while T > 0 and (not found):
        for i in range(nloops):
            cc += 1
            S_next = get_new_assignment(S_current[:])
            # print("{0} {1}\n{2} {3}\n".format(S_current, h(S_current), S_next, h(S_next)))
            if h(S_current) == 0:
                print("Objetivo alcanzado.")
                found = True
                break
            if h(S_next) <= h(S_current):
                S_current = S_next
            else:
                prob = exp((h(S_current)-h(S_next))/T)
                p = random.random()
                # print("T: {:.5f} | prob: {:.5f} | p: {:.5f} | accepts: {}\n".format(T, prob*100, p*100, p < prob))
                if p < prob:
                    S_current = S_next
        T -= R
    print("Cantidad de iteraciones: {}\n".format(cc))
    return cc

# Parametros para visualizar los cambios en las variables y las decisiones tomadas
# simulated_annealing(S,1,50,0.05) # peor
# simulated_annealing(S,0.1,1000,0.05) # mejor
```

Resultados Enfriamiento Simulado

Experimentalmente, se observó que valores iniciales elevados de T (como 10 o 20) retrasan la busqueda, ya que la probabilidad de aceptar nuevas asignaciones que son mucho peores es bastante alta.

Sin embargo, si el valor de T empieza muy pequeño (como 0.1), se necesitan más iteraciones entre cada reducción del parámetro ó un factor de reducción mayor para poder llegar a la respuesta.

Revisando las decisiones que se tomaban durante la ejecución del algoritmo, se observó que de forma general, para este problema, aceptar una asignación que es peor retrasa bastante la busqueda. El algoritmo tiene preferencia por aceptar soluciones "poco" peores, pero en este caso, parece que realizar tal accion crea pasos adicionales que no brindan algun tipo de "atajo" o ventaja evidente más adelante.

In [241…
```
aux = list()
for i in range(10):
    aux.append(simulated_annealing(S,10,50,0.05))
print("iteraciones promedio:", np.mean(aux))
```

```
Objetivo alcanzado.
Cantidad de iteraciones: 9896

Objetivo alcanzado.
Cantidad de iteraciones: 9495

Objetivo alcanzado.
Cantidad de iteraciones: 9905

Objetivo alcanzado.
Cantidad de iteraciones: 9732

Objetivo alcanzado.
Cantidad de iteraciones: 9949

Objetivo alcanzado.
Cantidad de iteraciones: 9853

Cantidad de iteraciones: 10000

Objetivo alcanzado.
Cantidad de iteraciones: 9854

Objetivo alcanzado.
Cantidad de iteraciones: 9941

Objetivo alcanzado.
Cantidad de iteraciones: 9651

iteraciones promedio: 9827.6
```

In [243…
```
aux = list()
for i in range(10):
    aux.append(simulated_annealing(S,1,50,0.05))
print("iteraciones promedio:", np.mean(aux))
```

```
Objetivo alcanzado.
Cantidad de iteraciones: 338

Cantidad de iteraciones: 1000

Objetivo alcanzado.
Cantidad de iteraciones: 883

Objetivo alcanzado.
Cantidad de iteraciones: 908

Objetivo alcanzado.
Cantidad de iteraciones: 829

Objetivo alcanzado.
Cantidad de iteraciones: 569

Objetivo alcanzado.
Cantidad de iteraciones: 738

Objetivo alcanzado.
Cantidad de iteraciones: 558

Objetivo alcanzado.
Cantidad de iteraciones: 720

Objetivo alcanzado.
Cantidad de iteraciones: 731

iteraciones promedio: 727.4
```

In [244…
```
aux = list()
for i in range(10):
    aux.append(simulated_annealing(S,1,50,0.1))
print("iteraciones promedio:", np.mean(aux))
```

```
Cantidad de iteraciones: 550

Objetivo alcanzado.
Cantidad de iteraciones: 412

Objetivo alcanzado.
Cantidad de iteraciones: 544

Objetivo alcanzado.
Cantidad de iteraciones: 260

Objetivo alcanzado.
Cantidad de iteraciones: 260

Cantidad de iteraciones: 550

Cantidad de iteraciones: 550

Objetivo alcanzado.
Cantidad de iteraciones: 547

Objetivo alcanzado.
Cantidad de iteraciones: 192

Cantidad de iteraciones: 550

iteraciones promedio: 441.5
```

In [245…
```
aux = list()
for i in range(10):
    aux.append(simulated_annealing(S,0.5,50,0.05))
print("iteraciones promedio:", np.mean(aux))
```

```
Objetivo alcanzado.
Cantidad de iteraciones: 355

Objetivo alcanzado.
Cantidad de iteraciones: 378

Cantidad de iteraciones: 550

Cantidad de iteraciones: 550

Objetivo alcanzado.
Cantidad de iteraciones: 322

Objetivo alcanzado.
Cantidad de iteraciones: 319

Objetivo alcanzado.
Cantidad de iteraciones: 238

Objetivo alcanzado.
Cantidad de iteraciones: 427

Cantidad de iteraciones: 550

Objetivo alcanzado.
Cantidad de iteraciones: 486

iteraciones promedio: 417.5
```

In [246…
```
aux = list()
for i in range(10):
    aux.append(simulated_annealing(S,0.1,1000,0.05))
print("iteraciones promedio:", np.mean(aux))
```

```
Objetivo alcanzado.
Cantidad de iteraciones: 577

Objetivo alcanzado.
Cantidad de iteraciones: 219

Objetivo alcanzado.
Cantidad de iteraciones: 414

Objetivo alcanzado.
Cantidad de iteraciones: 396

Objetivo alcanzado.
Cantidad de iteraciones: 250

Objetivo alcanzado.
Cantidad de iteraciones: 407

Objetivo alcanzado.
Cantidad de iteraciones: 321

Objetivo alcanzado.
Cantidad de iteraciones: 595

Objetivo alcanzado.
Cantidad de iteraciones: 837

Objetivo alcanzado.
Cantidad de iteraciones: 565

iteraciones promedio: 458.1
```

In [247…
```
aux = list()
for i in range(10):
    aux.append(simulated_annealing(S,0.1,50,0.005))
print("iteraciones promedio:", np.mean(aux))
```

```
Objetivo alcanzado.
Cantidad de iteraciones: 346

Objetivo alcanzado.
Cantidad de iteraciones: 311

Objetivo alcanzado.
Cantidad de iteraciones: 568

Objetivo alcanzado.
Cantidad de iteraciones: 396

Objetivo alcanzado.
Cantidad de iteraciones: 239

Objetivo alcanzado.
Cantidad de iteraciones: 797

Objetivo alcanzado.
Cantidad de iteraciones: 105

Objetivo alcanzado.
Cantidad de iteraciones: 378

Objetivo alcanzado.
Cantidad de iteraciones: 264

Objetivo alcanzado.
Cantidad de iteraciones: 658

iteraciones promedio: 406.2
```

2. Algoritmo genético

In [185…
```
# Función de selección
# Elige dos individuos aleatoriamente que van a ser cruzados
# generacion: lista de individuos que componen la generacion actual
# Retorna las posiciones de dos individuos en la generacion
def seleccion(generacion):
    tGen = len(generacion)
    ind1 = random.randint(1, tGen-1) # Por qué aquí estaba tGen-2 y no tGen-1?
    ind2 = ind1
    while ind1 == ind2:
        ind2 = random.randint(1,tGen-1)
    return (generacion[ind1], generacion[ind2])

#gen = [[1,2, 3],[4, 5, 6],[7, 8, 9],[10, 11, 12],[13, 14, 15],[16, 17, 18],[19, 20, 21],[22, 23, 24]]
#a,b = seleccion(gen)
#print(a)
#print(b)
```

In [184…
```
# Funcion de descarte de los individuos menos aptos
# generacion: lista de individuos que componen la generacion actual
# Retorna la generacion despues de eliminar la mitad menos apta
def descarte(generacion):
    tGen = len(generacion)
    return (generacion[:tGen//2])

#descarte(gen)
```

In [183…
```
# Funcion de cruce
# Precondicion: ambos individuos tienen la misma longitud
# ind1 e ind2 son individuos de la generacion actual
# Retorna dos nuevos individuos obtenidos a partir de ind1 e ind2 por ORDER-BASED CROSSOVER:
# http://t-news.cn/Floc2018/FLoC2018-pages/proceedings_paper_281.pdf (por lo general se crea un solo hijo, pero yo lo adapté para dos)
def cruce(ind1,ind2):
    n = len(ind1)

    # seleccionar n//2 indices únicos de forma aleatoria en el padre 2
    p2_inds = random.sample(range(0, n), n//2)
    p2_inds.sort()

    # guardar valores de los indices del padre 2 en orden
    vals_p2 = []
    for i in p2_inds:
        vals_p2.append(ind2[i])
    aux = set(vals_p2)

    # obtener indices correspondientes en el padre 1 a cada elemento en los indices anteriores
    p1_inds = []
    for i in range(n):
        if ind1[i] in aux:
            p1_inds.append(i)

    # guardar valores de los indices del padre 1 en orden
    vals_p1 = []
    for i in p1_inds:
        vals_p1.append(ind1[i])

    new1 = ind1
    new2 = ind2

    # se hace el crossover de los elementos en los ordenes correspondientes
    for i in range(len(p1_inds)):
        new1[p1_inds[i]] = vals_p2[i]
        new2[p2_inds[i]] = vals_p1[i]

    return (new1, new2)

# cruce([1, 2, 3, 4, 5, 6, 7, 8],[8, 2, 7, 5, 4, 6, 3, 1])
```

In [182…
```
# Funcion de mutacion
# ind es un individuo de la generacion actual
# prob es un valor entre 0 y 1 que corresponde a la probabilidad de mutacion
# Retorna un individuo que puede ser identico al que entró o puede tener un intercambio de valores en dos posiciones aleatorias
def mutacion(ind, prob):
    p = random.randint(1,100)
    if p < prob*100:
        tInd = len(ind)
        ind1 = random.randint(1, tInd-1)
        ind2 = ind1
        while ind1 == ind2:
            ind2 = random.randint(1, tInd-1)
        ind[ind1], ind[ind2] = ind[ind2], ind[ind1] # Hacemos un swap de las dos posiciones aleatorias
    return (ind)

# mutacion([1, 2, 3, 4, 5, 6, 7, 8],0.9)
```

In [181…
```
# Funcion newInd
# S: secuencia de enteros de entrada
# Genera un nuevo individuo aleatorio
# Retorna el individuo construido
# El individuo corresponde a una permutacion aleatoria de S
def newInd(S):
    return list(np.random.permutation(S))

# newInd(S)
```

In [180…
```
# Funcion primeraGen
# S: secuencia de enteros de entrada
# nIndGen: numero de individuos por generacion
# Retorna la primera generacion poblada con el numero de individuos requeridos
def primeraGen(S, nIndGen):
    generacion = []
    while len(generacion) < nIndGen:
```

```python
        generacion.append(newInd(S))
    return generacion

# primeraGen(S, 3)
```

In [179…

```python
# Funcion fitness
# ind: es un individuo de la generacion actual
# Retorna un valor numerico que representa la aptitud del individuo
# Calcula la cantidad de anti-inversiones en el individuo
def fitness(ind):
    n = len(ind)
    ordered = 0
    for i in range(n):
        for j in range(i + 1, n):
            if ind[i] < ind[j]:
                ordered += 1
    return ordered

# fitness([4,3,2,1])
```

In [194…

```python
# Funcion general
# nIndGen: numero de individuos por generacion
# nGen: numero de generaciones que realizara el algoritmo
# pMut: probabilidad de mutacion
def genetico(S,nIndGen,nGen,pMut):
    goal_fitness = fitness(sorted(S))
    print("S inicial:",S,goal_fitness-fitness(S))
    generacion = primeraGen(S,nIndGen)
    while nGen > 0:
        generacion.sort(key = fitness, reverse = True)
        current_fitness = fitness(generacion[0])
        print(generacion[0], goal_fitness-current_fitness)
        if goal_fitness-current_fitness == 0: # Se encuentra la respuesta
            print("\nObjetivo alcanzado.")
            break
        generacion = descarte(generacion)
        children = []
        while len(children) + len(generacion) < nIndGen:
            parent1, parent2 = seleccion(generacion)
            child1, child2 = cruce(parent1,parent2)
            child1 = mutacion(child1, pMut)
            child2 = mutacion(child2, pMut)
            children.append(child1)
            children.append(child2)
        generacion = generacion + children
        nGen = nGen - 1
```

Resultados Algoritmo Genético

La probabilidad de mutación es de los parametros más influyentes. Si esta es baja (eg 0.0005), todas las generaciones se mantienen con un nivel de aptitud alto. Si es más alta (eg 0.1), entre más generaciones hay, los individuos se vuelven cada vez menos aptos (los individuos se van volviendo cada vez más desordenados)

Se observó también que la cantidad de generaciones no es un parametro que mejore la aptitud de las generaciones a través del tiempo (para este problema). No importa cuantas más generaciones se añadan, si hay pocos individuos por generacion y una probabilidad de mutacion más o menos alta, nunca se llega a una solución.

El parametro de cantidad de generaciones se deja en 10 para la mayoria de las pruebas. Lo que más influye en el desempeño es la cantidad de individuos por generacion y la probabilidad de mutación.

In [206…

```python
genetico(S,100000,10,0.0005)
```

```
S inicial: [-96, -63, -53, 48, -77, 50, -55, -54, 3, -17] 14
[-96, -77, -55, -54, -63, -53, -17, 3, 48, 50] 2
[-96, -77, -55, -54, -63, -53, -17, 3, 48, 50] 2
[-96, -77, -63, -55, -54, -53, -17, 3, 50, 48] 1
[-96, -77, -63, -55, -54, -53, -17, 3, 48, 50] 0

Objetivo alcanzado.
```

In [ ]:

```python
genetico(S,1000,10000,0.05)
```

In [209…

```python
genetico(S,30000,10,0.0001)
```

```
S inicial: [-96, -63, -53, 48, -77, 50, -55, -54, 3, -17] 14
[-96, -63, -77, -55, -17, -54, -53, 48, 3, 50] 4
[-77, -96, -63, -55, -54, -17, -53, 3, 48, 50] 2
[-96, -77, -63, -53, -55, -17, -54, 3, 48, 50] 3
[-96, -77, -63, -55, -54, -53, -17, 3, 50, 48] 1
[-77, -96, -63, -55, -54, -53, -17, 3, 48, 50] 1
[-96, -77, -55, -63, -17, -54, -53, 3, 48, 50] 3
[-96, -77, -63, -55, -17, -53, 48, -54, 3, 50] 5
[-77, -55, -96, -17, -63, -53, -54, 3, 48, 50] 7
[-63, -96, -55, -77, -17, -53, 3, -54, 48, 50] 7
[-53, -96, -63, -77, -17, -55, 3, -54, 50, 48] 10
```

In [211…

```python
genetico(S,30000,20,0.1)
```

```
S inicial: [-96, -63, -53, 48, -77, 50, -55, -54, 3, -17] 14
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, -53, 3, 48, 50] 1
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
```

```
[-96, -77, -63, -55, -54, -17, 3, -53, 48, 50] 2
```

In [218…
```
genetico(S,30000,20,0.00001)
```

```
S inicial: [-96, -63, -53, 48, -77, 50, -55, -54, 3, -17] 14
[-96, -63, -54, -77, -53, -55, -17, 3, 48, 50] 4
[-77, -63, -55, -96, -54, -53, -17, 3, 48, 50] 3
[-96, -77, -63, -55, -54, -17, -53, 3, 48, 50] 1
[-96, -77, -63, -55, -54, -53, -17, 3, 50, 48] 1
[-96, -77, -63, -55, -54, -17, -53, 3, 50, 48] 2
[-96, -77, -63, -54, -55, -53, -17, 3, 48, 50] 1
[-96, -77, -63, -55, -54, -53, -17, 3, 50, 48] 1
[-96, -77, -55, -63, 3, -54, -17, -53, 48, 50] 5
[-96, -77, -63, 48, -54, -55, -53, -17, 50, 3] 7
[-96, 48, -77, -63, -54, -55, -17, -53, 3, 50] 9
[-96, -77, -55, -17, -63, 3, 50, -54, -53, 48] 9
[-77, -96, -63, -55, 3, -54, 48, 50, -53, -17] 8
[-96, -77, -54, 3, -63, -53, 48, -17, 50, -55] 11
[3, -63, -54, -77, -96, -55, -53, -17, 48, 50] 13
[-63, -77, -55, -54, -96, -53, 50, 48, -17, 3] 10
[-55, 48, -63, -96, -77, -54, -17, 3, 50, -53] 15
[-96, -55, 48, -77, -53, 3, -63, 50, -54, -17] 15
[-96, -17, -55, -53, -54, 3, 50, -63, -77, 48] 18
[50, -63, -96, -77, -55, -53, -54, 3, -17, 48] 13
[-96, -17, -53, -55, 50, -63, -54, -77, 3, 48] 18
```

In [219…
```
genetico(S,50000,10,0.00001)
```

```
S inicial: [-96, -63, -53, 48, -77, 50, -55, -54, 3, -17] 14
[-96, -77, -63, -54, -55, -17, 3, -53, 48, 50] 3
[-96, -63, -77, -54, -55, -53, -17, 3, 48, 50] 2
[-96, -77, -63, -55, -54, -53, 3, -17, 48, 50] 1
[-96, -77, -63, -55, -54, 3, -53, -17, 48, 50] 2
[-96, -77, -63, -55, -54, -53, -17, 3, 48, 50] 0

Objetivo alcanzado.
```

3. Comparación de técnicas

Enfriamiento Simulado

Ventajas:

- Es fácil de implementar
- Es eficiente en memoria (solo se emplean 2 conjuntos de variables)
- Se puede adaptar para cualquier tipo de problema con facilidad

Desventajas:

- No es claro qué valor de T (y del resto de parametros) es el más indicado para resolver el problema
- No es tan fácil identificar si una aceptación de una asignación erronea es beneficiosa o no en algún momento

Algoritmo Genético

Ventajas:

- Es un método más exhaustivo. Puede funcionar mejor para problemas más complejos (con más restricciones) o con mucho ruido que no se pueden resolver de forma analítica fácilmente
- Es muy modular y esto abre la posibilidad de hacer el procesamiento de forma paralela (procesar sub-conjuntos de cada generacion de forma paralela y comparar los mejores individuos)
- Como se emplean varios conjuntos de variables en lugar de uno sólo, se reduce mucho la posibilidad de llegar a una solución local sub-optima

Desventajas:

- Puede llegar a usar mucha memoria dependiendo de la representación computacional de los cromosomas (nIndGen*nGen)
- Requiere de muchos individuos (y consecuentemente, cantidad de iteraciones y memoria) para poder llegar a una solución óptima
- Es un poco complejo y extenso en cuanto a implementación
- La estrategia de cruce o crossover varía mucho con respecto al problema que se esté resolviendo (es necesario implementar una función de cruce para cada tipo de problema diferente)

Conclusión

Se recomienda usar la técnica de enfriamiento simulado. Según la experiencia de resolver el taller, ésta es más practica, simple y eficiente para problemas como el que se está tratando de resolver en este taller. El algoritmo genético parece ser más adecuado para problemas mucho más complejos, de mayor escala y variabilidad en cuanto al espacio de solución (muchas soluciones locales sub-optimas).

In [ ]: