A01187664_A01411162_T4

October 17, 2021

1 Algoritmos Genéticos

1.0.1 Luis Cossío Ramírez A01187664

1.0.2 Jorge Antonio Ruiz A01411162

Resolver el problema de la mochila con los siguientes datos (de Kreher and Stinson). El peso máximo es 6404180 y el profit de la mejor solución es 13549094. Los parámetros para el algoritmo quedan a discreción del alumno. Realizar la penalización fuera de la función de evaluación (usar decorator para penalización). En los siguientes archivos pueden encontar los pesos y profits.

[ ]:
```
from deap import base, creator, tools, algorithms
import numpy as np
import pandas as pd
import random
import plotly.express as px
import plotly.graph_objects as go
from concurrent.futures import ThreadPoolExecutor
```

1.1 Cargar datos

Los datos se cargan de los archivos proporcionados y se almacenan en un dataframe de pandas. De esta forma se podrá utilizarlos para obtener las sumas de profit y weights de forma ordenada.

[ ]:
```
# Load data
MAX_WEIGHT = 6404180
MAX_PROFIT = 13549094

weights = profits = []

with open('p08_w.txt') as w_file:
  weights = w_file.readlines()

with open('p08_p.txt') as p_file:
  profits = p_file.readlines()

weights = [int(x.strip()) for x in weights]
profits = [int(x.strip()) for x in profits]
```

```
data = pd.DataFrame({'weight': weights, 'profit': profits})
count = data.count()
assert count[0] == count[1]
N = data.count()[0]
print(f'There are {N} items to choose')
print(f'Max weight is {MAX_WEIGHT}')
data.head()
```

```
There are 24 items to choose
Max weight is 6404180
```

[ ]:

	weight	profit
0	382745	825594
1	799601	1677009
2	909247	1676628
3	729069	1523970
4	467902	943972

[ ]:
```
# Define evaluation function.
def func_eval(ind, ):
  """Evaluation function for the individual."""
  assert len(ind) == N
  return data.profit[ind].sum(),

# In case that the sum of the weights is greater than the maximum weight,␣
 ↪profit is zero.
def feasible(ind, ):
  """Feasibility function for the individual. Returns True if feasible False
  otherwise."""
  weight = data.weight[ind].sum()
  return weight <= MAX_WEIGHT

def test_func_eval():
  # Arrange
  expected_sum = data.profit[0] + data.profit[2]
  test_ind = [True, False, True] + [False]*(N-3)
  # Act
  profit = func_eval(test_ind)
  # Assert
  assert profit == expected_sum
test_func_eval()
```

1.2 Definición de parámetros

La representación de cada individuo es una lista de valores booleanos, (ej. [True, False, True, True]) donde cada posición es un elemento y el valor decide si se selecciona para la mochila o no.

Aquí se define que se usará selección de torneo de tamaño 10, cruce de dos puntos, mutación de voltear cada bit con 0.1 de probabilidad, y la funcion de evaluación, que suma a todos los items

que tengan `True` en su correspondiente posición.

[ ]:
```
toolbox = base.Toolbox()

toolbox.register('select', tools.selTournament, tournsize=10)
toolbox.register('mate', tools.cxTwoPoint)
toolbox.register('mutate', tools.mutFlipBit, indpb=0.1)
toolbox.register('evaluate', func_eval)
toolbox.decorate('evaluate', tools.DeltaPenalty(feasible, 0.0))

creator.create('FitnessMax', base.Fitness, weights=(1.0,))
creator.create('Individual', list, fitness=creator.FitnessMax)

toolbox.register('bit', random.choice, seq=[True, False])
toolbox.register('individual', tools.initRepeat, creator.Individual, toolbox.
 ↪bit, n=N)


toolbox.register('population', tools.initRepeat, list, toolbox.individual)

stats = tools.Statistics(key=lambda ind: ind.fitness.values)
stats.register('min', np.min)
stats.register('max', np.max)
stats.register('mean', np.mean)
stats.register('std', np.std)
stats.register('median', np.median)

hof1 = tools.HallOfFame(3)
hof2 = tools.HallOfFame(3)
hof3 = tools.HallOfFame(3)
```

1.3 Experimentacion

Aquí se encapsula cada algoritmo para ser usado concurrentemente varias veces, de modo que se obtenga el resultado de todos los experimentos en una lista.

Se establecen los parámetros, para todos se usa probabilidad de cruce de 0.7 y probabilidad de mutacion de 0.2. Para mu,lambda y mu+lambda se usa lambda de 20, y mu de 5.

[ ]:
```
n_experiments = 10
n_gens = 1000

def eaSimple(_):
  pop = toolbox.population(n=20)
  _, log = algorithms.eaSimple(pop, toolbox, cxpb=0.7, mutpb=0.2, ngen=n_gens,␣
 ↪stats=stats, halloffame=hof1, verbose=False)
  log = pd.DataFrame(log)
  return log
```

```
def eaMuPlusLambda(_):
  pop = toolbox.population(n=20)
  _, log = algorithms.eaMuPlusLambda(pop, toolbox, mu=4, lambda_=20, cxpb=0.7,␣
 ↪mutpb=0.2, ngen=n_gens, stats=stats, halloffame=hof2, verbose=False)
  log = pd.DataFrame(log)
  return log

def eaMuCommaLambda(_):
  pop = toolbox.population(n=20)
  _, log = algorithms.eaMuCommaLambda(pop, toolbox, mu=4, lambda_=20, cxpb=0.7,␣
 ↪mutpb=0.2, ngen=n_gens, stats=stats, halloffame=hof3, verbose=False)
  log = pd.DataFrame(log)
  return log

def experiment(algorithm, n_experiments):
  with ThreadPoolExecutor() as executor:
    logs = executor.map(algorithm,range(n_experiments))
  return list(logs)
```

[ ]:
```
def get_summary(logs):
  summary = pd.DataFrame()
  maxs = [log['max'] for log in logs]
  assert len(maxs) == n_experiments
  summary['std'] = np.std(maxs, axis=0)
  summary['avg best'] = np.mean(maxs, axis=0)
  summary['avg+std best'] = summary['avg best'] + summary['std']
  summary['avg-std best'] = summary['avg best'] - summary['std']
  return summary
```

[ ]:
```
def print_hof(hof):
  for ind in hof:
    print(f'profit: {func_eval(ind)[0]}\tweight: {data.weight[ind].sum()}')
```

[ ]:
```
def plot_bests(logs, title):
  summary = get_summary(logs)
  fig = px.line(summary, y=['avg+std best', 'avg-std␣
 ↪best'],line_dash_sequence=['dot'], title=title)
  fig.add_trace(
    go.Scatter(
      x=summary.index, y=summary['avg best'],
      line_color='rgb(0,176,246)',
      name='avg best'))
  fig.show()
```

1.4 Resultados

A continuación se presenta la curva de mejores encontrados por generación. Se muestra como un promedio junto a sus desviaciones estándar de los 10 experimentos que se realizaron de cada algoritmo.

[ ]:
```
logs1 = experiment(eaSimple, n_experiments)
```

[ ]:
```
print_hof(hof1)
plot_bests(logs1, 'E.A. Simple')
```

```
profit: 13443671        weight: 6401856
profit: 13441033        weight: 6398388
profit: 13440957        weight: 6397876
```

[ ]:
```
logs2 = experiment(eaMuPlusLambda, n_experiments)
```

[ ]:
```
print_hof(hof2)
plot_bests(logs2, 'E.A. Mu + Lambda')
```

```
profit: 13487304        weight: 6398363
profit: 13481934        weight: 6380530
profit: 13477999        weight: 6377305
```

[ ]:
```
logs3 = experiment(eaMuCommaLambda, n_experiments)
```

[ ]:
```
plot_bests(logs3, 'E.A. Mu, Lambda')
print_hof(hof3)
```

```
profit: 13467282        weight: 6398105
profit: 13466838        weight: 6403688
profit: 13440957        weight: 6397876
```

[ ]:
```
summary1 = get_summary(logs1)
summary2 = get_summary(logs2)
summary3 = get_summary(logs3)
summary1['Best possible'] = [MAX_PROFIT]*summary1.shape[0]
fig = px.line(summary1, y=['Best possible'],␣
 ↪title='Performance',line_dash_sequence=['dot'],labels={0:'Best possible'})
fig.add_trace(
  go.Scatter(
    x=summary1.index, y=summary1['avg best'],
    line_color='rgb(0,176,246)',
    name='simple'))
fig.add_trace(
  go.Scatter(
    x=summary2.index, y=summary2['avg best'],
    line_color='rgb(176,246,0)',
    name='mu + lambda'))
fig.add_trace(
```

```
  go.Scatter(
    x=summary3.index, y=summary3['avg best'],
    line_color='rgb(176,0,246)',
    name='mu, lambda'))
fig.show()
```

1.5 Conclusiones

El algoritmo Mu + Lambda fue el que tuvo el mejor individuo, logrando un profit máximo de 13,487,304, con peso de 6,398,363

Sin embargo, el algorito que tuvo mejor desempeño, y que convergió más rápidamente (por poco), fue sorpresivamente el simple. Esto es considerando que para todos los algoritmos se usaron los mismos valores de probabilidad de cruce y de mutación.