Practica3G05

December 12, 2019

1 Práctica 3 - Inteligencia Artificial - Parte 2

G05 - Francisco Javier Blázquez Martínez, Boris Carballa Corredoira, Juan Carlos Villanueva Quirós

[1]:
```
# from search import *
import random
import math
```

[2]:
```
class Cromosoma(object):

    def __init__(self, tareas_trabajadores, trabajadores_tareasEnOrden):
        self.tareas_trab = tareas_trabajadores
        self.trab_tareasOrden = trabajadores_tareasEnOrden

    def __str__(self):
        return "Trabajador de la tarea i-ésima: "+str(self.
 ↪tareas_trab)+"\nLista de tareas (en orden) del trabajador i-ésimo:␣
 ↪"+str(self.trab_tareasOrden)
```

[3]:
```
NOT_QUALIFIED = -1

class ProblemaGenetico(object):
    def __init__(self, timetable, restricciones):
        self.working_board = timetable
        self.num_tareas = len(timetable[0])
        self.num_trabajadores = len(timetable)
        self.restricciones = restricciones
        self.restricciones_tarea = [[] for _ in range(self.num_tareas)]
        for restriccion in restricciones:
            self.restricciones_tarea[restriccion[1]].append(restriccion[0])

    def decodifica(self, cromosoma):
        respuesta = "Reparto:\n\n" + str(cromosoma)
        respuesta += "\nMomento de inicio de las tareas: "
        vector_finalizaciones = self.fitness1(cromosoma)[1] #self.
 ↪fitness2(cromosoma)[1]
        for i in range(self.num_tareas):
```

```
            vector_finalizaciones[i] -= self.working_board[cromosoma.
↪tareas_trab[i]][i]
        respuesta += str(vector_finalizaciones)
        return respuesta

    def muta(self, cromosoma, prob):
        # Reemplazo aleatorio: cambiamos el trabajador de una tarea
        if (random.random() < prob):
            tarea_a_cambiar  = random.choice([i for i in range(self.
↪num_tareas)])
            viejo_trabajador = cromosoma.tareas_trab[tarea_a_cambiar]
            nuevo_trabajador = random.choice([i for i in range(self.
↪num_trabajadores) if self.working_board[i][tarea_a_cambiar]!=NOT_QUALIFIED])
            #Cambiamos cromosoma
            cromosoma.tareas_trab[tarea_a_cambiar] = nuevo_trabajador
            cromosoma.trab_tareasOrden[viejo_trabajador].remove(tarea_a_cambiar)
            cromosoma.trab_tareasOrden[nuevo_trabajador].append(tarea_a_cambiar)
        return cromosoma

    def cruza(self, cromosoma1, cromosoma2):

        hijo1 = Cromosoma([0]*self.num_tareas, [[] for _ in range(self.
↪num_trabajadores)])
        hijo2 = Cromosoma([0]*self.num_tareas, [[] for _ in range(self.
↪num_trabajadores)])

        for trabajador in range(self.num_trabajadores):
            for tarea in cromosoma1.trab_tareasOrden[trabajador]:
                if tarea <= (self.num_tareas//2):
                    hijo1.tareas_trab[tarea]=trabajador
                    hijo1.trab_tareasOrden[trabajador].append(tarea)
                else:
                    hijo2.tareas_trab[tarea]=trabajador
                    hijo2.trab_tareasOrden[trabajador].append(tarea)

        for trabajador in range(self.num_trabajadores):
            for tarea in cromosoma2.trab_tareasOrden[trabajador]:
                if tarea <= (self.num_tareas//2):
                    hijo2.tareas_trab[tarea]=trabajador
                    hijo2.trab_tareasOrden[trabajador].append(tarea)
                else:
                    hijo1.tareas_trab[tarea]=trabajador
                    hijo1.trab_tareasOrden[trabajador].append(tarea)

        return [hijo1,hijo2]

    # Hicimos dos bastante diferentes pero válidos ambos
```

```
   def fitness(self, cromosoma):
       return self.fitness1(cromosoma)[0]
       #return self.fitness2(cromosoma)[0]

   ##Fitness: dado un cromosoma, devuelve su fitness, es decir el tiempo que␣
↪se tarda en hacerse todas las tareas
   # Vamos iterando por los trabajadores, estableciendo la hora de inicio de␣
↪su siguiente tarea
   # si la anterior tarea de ese trabajador y las tareas de las restricciones␣
↪ya han finalizado.
   # Si todos los trabajadores están esperando a otro, hay un deadlock y el␣
↪fitness es +infinito.
   def fitness1(self, cromosoma):
       tareas_tiempoFinalizacion = [-1 for _ in range(self.num_tareas)]
       trab_tiempoOcupacion = [0 for _ in range(self.num_trabajadores)]

       tareasCalculadas = 0
       hayCambio = True

       while (hayCambio and tareasCalculadas<self.num_tareas):
           hayCambio = False

           for i in range(self.num_trabajadores):

               tareas_pendientes = [k for k in cromosoma.trab_tareasOrden[i]␣
↪if tareas_tiempoFinalizacion[k]==-1]

               if(not(tareas_pendientes)):
                   continue

               for j in tareas_pendientes:
                   ## Si tiene alguna dependencia por ejecutar paramos y vamos␣
↪al siguiente trabajador
                   if(len([k for k in self.restricciones_tarea[j] if␣
↪tareas_tiempoFinalizacion[k]==-1])>0):
                       break
                   ## Si no la ejecutamos
                   else:
                       tareasCalculadas += 1
                       hayCambio = True

                       ## Aux1 = máximo tiempo de finalización de las␣
↪dependencias
                       aux1 = max([tareas_tiempoFinalizacion[k] for k in self.
↪restricciones_tarea[j]], default=0)
```

```
                        ## Aux2 = cuándo queda libre el trabajador que hace la␣
↪tarea
                        aux2 = trab_tiempoOcupacion[i]
                        aux3 = max(aux1,aux2)

                        tareas_tiempoFinalizacion[j] = aux3 + self.
↪working_board[i][j]
                        trab_tiempoOcupacion[i] = aux3 + self.
↪working_board[i][j]

        if tareasCalculadas<self.num_tareas:
            #print("Hemos llegado a una estado no válido:")
            #print(cromosoma)
            return (math.inf,[])
        else:
            return (max(tareas_tiempoFinalizacion),tareas_tiempoFinalizacion)

    # funcion auxiliar de fitness2
    # calcula el número de restricciones de la tarea pasada y la guarda en␣
↪num_restri_tareas
    def fitness2_num_restricciones(self, tarea, num_restri_tareas,␣
↪restricciones_tareas):
        # -1 => no calculado, -2 => en proceso (sirve como un control de␣
↪repetidos de coste cte)
        if num_restri_tareas[tarea] ==-2: # ya esta repetido
            num_restri_tareas[tarea] = math.inf
            return math.inf
        elif num_restri_tareas[tarea] == -1:
            num_restri_tareas[tarea] = -2
            num = 0
            for restriccion in restricciones_tareas[tarea]:
                num = max(num,1+self.
↪fitness2_num_restricciones(restriccion,num_restri_tareas,restricciones_tareas))
            num_restri_tareas[tarea] = num
            return num
        else:
            return num_restri_tareas[tarea]

    # Calculamos para cada tarea cuantas restricciones tiene, incluyendo como␣
↪restriccion la de ir antes que su
    # anterior por el orden de las tareas del trabajador. Calculamos el máximo␣
↪de la cadena de restricciones que
    # tiene que realizarse para que esté lista para iniciarse. Iteramos por las␣
↪tareas, empezando por las que
    # tienen menos restricciones. Si calculando el máximo anterior pasamos por␣
↪la misma restricción, hay un deadlock
```

```
    # y devolvemos +infinito.
    def fitness2(self, cromosoma):
        tiempoFinTareas = [0 for _ in range(self.num_tareas)]

        restricciones_tareas = [[] for _ in range(self.num_tareas)]
        for restriccion in self.restricciones:
            restricciones_tareas[restriccion[1]].append(restriccion[0])
        for trabajador in range(self.num_trabajadores):
            tarea_anterior = -1
            for tarea in cromosoma.trab_tareasOrden[trabajador]:
                if tarea_anterior != -1:
                    restricciones_tareas[tarea].append(tarea_anterior)
                tarea_anterior = tarea

        #Calculamos el nž de restricciones de cada tarea
        num_restri_tareas = [-1 for _ in range(self.num_tareas)]
        for i in range(self.num_tareas):
            if num_restri_tareas[i] == -1:
                self.
↪fitness2_num_restricciones(i,num_restri_tareas,restricciones_tareas)
            if num_restri_tareas[i] == math.inf: #Hay un deadlock
                return (math.inf,[])

        #Las calculamos en orden
        lista_pares = [(i,num_restri_tareas[i]) for i in range(self.num_tareas)]
        lista_pares.sort(key=lambda tup: tup[0])
        for tupla in lista_pares:
            tarea = tupla[0]
            trabajador = cromosoma.tareas_trab[tarea]
            inicio = 0
            for tarea_ant in restricciones_tareas[tarea]:
                inicio = max(inicio,tiempoFinTareas[tarea_ant])
            tiempoFinTareas[tarea] = inicio + self.
↪working_board[trabajador][tarea]

        return (max(tiempoFinTareas), tiempoFinTareas)
```

[4]:
```
#Ejemplo función de fitness
cromosoma = Cromosoma([0,0,1],[[0,1],[2]])
grid = [[10,20,-1],[-1,-1,30]]
problemaAux = ProblemaGenetico(grid, [(1,2)])
problemaAux.fitness(cromosoma)
```

[4]: 60

[5]:
```
## Elegimos de entre los trabajadores cualificados para una determinada tarea␣
↪uno al azar.
def poblacion_inicial(problema_genetico, size):
```

```
    nueva_poblacion = []
    for i in range(size):
        genotipo = Cromosoma([0]*problema_genetico.num_tareas, [[] for _ in␣
↪range(problema_genetico.num_trabajadores)])

        ## Si queremos que de verdad sea aleatorio tenemos que hacer shuffle de␣
↪cada lista de genotipo!
        for j in range(problema_genetico.num_tareas):
            worker = random.choice([i for i in range(problema_genetico.
↪num_trabajadores) if problema_genetico.working_board[i][j]!=NOT_QUALIFIED])
            genotipo.tareas_trab[j] = worker
            genotipo.trab_tareasOrden[worker].append(j)

        nueva_poblacion.append(genotipo)
    return nueva_poblacion
```

[7]:
```
#Ejemplo función poblacion_inicial
grid = [[1,1,1,1],[2,2,2,2],[NOT_QUALIFIED,1,NOT_QUALIFIED,1]]
problema = ProblemaGenetico(grid, [])
poblacion_ini = poblacion_inicial(problema, 2)
print(poblacion_ini[0])
print(poblacion_ini[1])
```

```
Trabajador de la tarea i-ésima: [1, 1, 1, 1]
Lista de tareas (en orden) del trabajador i-ésimo: [[], [0, 1, 2, 3], []]
Trabajador de la tarea i-ésima: [1, 0, 0, 0]
Lista de tareas (en orden) del trabajador i-ésimo: [[1, 2, 3], [0], []]
```

[8]:
```
## La primera mitad de las tareas las realiza el trabajador que decida el␣
↪padre,
## la segunda mitad quien diga la madre

def cruza_padres(problema_genetico,padres):
    nueva_poblacion = []
    for i in range(0,len(padres),2):
        nueva_poblacion += problema_genetico.cruza(padres[i],padres[i+1])
    return nueva_poblacion
```

[9]:
```
#Ejemplo función cruza_padres
nueva_poblacion = cruza_padres(problema, poblacion_ini)
print("Padres:\n")
print(poblacion_ini[0])
print(poblacion_ini[1])
print("\nHijos\n")
print(nueva_poblacion[0])
print(nueva_poblacion[1])
```

```
Padres:
```

```
Trabajador de la tarea i-ésima: [1, 1, 1, 1]
Lista de tareas (en orden) del trabajador i-ésimo: [[], [0, 1, 2, 3], []]
Trabajador de la tarea i-ésima: [1, 0, 0, 0]
Lista de tareas (en orden) del trabajador i-ésimo: [[1, 2, 3], [0], []]

Hijos

Trabajador de la tarea i-ésima: [1, 1, 1, 0]
Lista de tareas (en orden) del trabajador i-ésimo: [[3], [0, 1, 2], []]
Trabajador de la tarea i-ésima: [1, 0, 0, 1]
Lista de tareas (en orden) del trabajador i-ésimo: [[1, 2], [3, 0], []]
```

[10]:
```python
## Muta todos los individuos con una cierta probabilidad

def muta_individuos(problema_genetico, poblacion, prob):
    nueva_poblacion = []
    for individuo in poblacion:
        nueva_poblacion.append(problema_genetico.muta(individuo,prob))
    return nueva_poblacion
```

[11]:
```python
#Ejemplo función muta_individuos
print("Antes\n")
print(poblacion_ini[0])
print("\nDespués\n")
mutado = muta_individuos(problema,[poblacion_ini[0]],0.5)
print(mutado[0])
```

```
Antes

Trabajador de la tarea i-ésima: [1, 1, 1, 1]
Lista de tareas (en orden) del trabajador i-ésimo: [[], [0, 1, 2, 3], []]

Después

Trabajador de la tarea i-ésima: [1, 2, 1, 1]
Lista de tareas (en orden) del trabajador i-ésimo: [[], [0, 2, 3], [1]]
```

[12]:
```python
## Elige n elementos por torneo de k candidatos (grupos de torneo aleatorios)

def seleccion_por_torneo(problema_genetico, poblacion, n, k):
    seleccionados = []
    for i in range(n):
        participantes = random.sample(poblacion,k)
        seleccionado = min(participantes, key=problema_genetico.fitness)
        seleccionados.append(seleccionado)
    return seleccionados
```

[13]:
```
#Ejemplo función seleccion_por_torneo
tam_poblacion = 5
poblacion_ini = poblacion_inicial(problema, tam_poblacion)
print("Población de tamaño: "+str(tam_poblacion)+"\n")
for i in range(tam_poblacion):
    print(poblacion_ini[i])

tam_seleccionados = 3
mejor_entre_k = 3
print("\nSeleccionamos "+str(mejor_entre_k)+" aleatoriamente y cogemos el␣
 ↪mejor, así "+str(tam_seleccionados)+" veces\n")
seleccionados =␣
 ↪seleccion_por_torneo(problema,poblacion_ini,tam_seleccionados,mejor_entre_k)
for i in range(tam_seleccionados):
    print(seleccionados[i])
```

```
Población de tamaño: 5

Trabajador de la tarea i-ésima: [0, 1, 1, 0]
Lista de tareas (en orden) del trabajador i-ésimo: [[0, 3], [1, 2], []]
Trabajador de la tarea i-ésima: [0, 2, 0, 0]
Lista de tareas (en orden) del trabajador i-ésimo: [[0, 2, 3], [], [1]]
Trabajador de la tarea i-ésima: [0, 1, 1, 0]
Lista de tareas (en orden) del trabajador i-ésimo: [[0, 3], [1, 2], []]
Trabajador de la tarea i-ésima: [1, 1, 0, 2]
Lista de tareas (en orden) del trabajador i-ésimo: [[2], [0, 1], [3]]
Trabajador de la tarea i-ésima: [1, 2, 1, 1]
Lista de tareas (en orden) del trabajador i-ésimo: [[], [0, 2, 3], [1]]

Seleccionamos 3 aleatoriamente y cogemos el mejor, así 3 veces

Trabajador de la tarea i-ésima: [0, 1, 1, 0]
Lista de tareas (en orden) del trabajador i-ésimo: [[0, 3], [1, 2], []]
Trabajador de la tarea i-ésima: [1, 1, 0, 2]
Lista de tareas (en orden) del trabajador i-ésimo: [[2], [0, 1], [3]]
Trabajador de la tarea i-ésima: [0, 2, 0, 0]
Lista de tareas (en orden) del trabajador i-ésimo: [[0, 2, 3], [], [1]]
```

[14]:
```
def nueva_generacion(problema_genetico, k, poblacion, n_padres, n_directos,␣
 ↪prob_mutar):
    ## Realizamos la selección por torneo
    padres2 = seleccion_por_torneo(problema_genetico, poblacion, n_directos, k)
    padres1 = seleccion_por_torneo(problema_genetico, poblacion, n_padres ,  k)
    ## Realizamos los cruces, la siguiente generación son los cruces de los␣
 ↪padres y
    ## k elementos seleccionados por torneo de los no padres
    cruces =  cruza_padres(problema_genetico,padres1)
```

```
    generacion = padres2+cruces
    ## Aplicamos mutaciones
    resultado_mutaciones = muta_individuos(problema_genetico, generacion,␣
 ↪prob_mutar)
    return resultado_mutaciones
```

[15]:
```
# Sus argumentos son:
# problema_genetico: una instancia de la clase ProblemaGenetico con la␣
 ↪representación adecuada del problema de optimización
# que se quiere resolver.
# k: número de participantes en los torneos de selección.
# nGen: número de generaciones (que se usa como condición de terminación)
# size: número de individuos en cada generación
# prop_cruce: proporción del total de la población que serán padres.
# prob_mutación: probabilidad de realizar una mutación de un gen.

def algoritmo_genetico(problema_genetico,k,ngen,size,prop_cruces,prob_mutar):
    ## Generamos una población inicial
    poblacion= poblacion_inicial(problema_genetico,size)

    ## Tomamos el número de padres (siempre par) y no padres en base a la␣
 ↪proporción dada
    n_padres=round(size*prop_cruces)
    n_padres= int(n_padres if n_padres%2==0 else n_padres-1)
    n_directos = size-n_padres

    ## Hacemos avanzar ngen generaciones nuestra población
    for _ in range(ngen):
        poblacion=nueva_generacion(problema_genetico,k,poblacion,n_padres,␣
 ↪n_directos,prob_mutar)

    ## Nos quedamos con el mejor individuo en base a nuestra función de fitness
    ## devolvemos su fenotipo (decodificación) y valor (función fitness)
    mejor_cr= min(poblacion, key=problema_genetico.fitness)
    mejor=problema_genetico.decodifica(mejor_cr)
    return (mejor,problema_genetico.fitness(mejor_cr))
```

[16]:
```
grid = [[10,20,-1],[-1,-1,30]]
problemaAux = ProblemaGenetico(grid, [(1,2)])

#algoritmo_genetico(problema_genetico,k,ngen,size,prop_cruces,prob_mutar):
solucion = algoritmo_genetico(problemaAux,3,20,10,0.7,0.1)
print(str(solucion[0])+"\n")
print("Mejor tiempo: "+ str(solucion[1])+"\n")
```

```
Reparto:

Trabajador de la tarea i-ésima: [0, 0, 1]
```

```
Lista de tareas (en orden) del trabajador i-ésimo: [[1, 0], [2]]
Momento de inicio de las tareas: [20, 0, 20]

Mejor tiempo: 50
```