UNIVERSITAT POLITÈCNICA DE VALÈNCIA

MIARFID
Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital

DSIC
DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

Máster en Inteligencia Artificial, Reconocimiento de Formas e Imagen Digital
Universitat Politècnica de València

Resolución del problema de las cartas mediante técnicas metaheurísticas.

Técnicas de Inteligencia Artificial

Autor: Juan Antonio López Ramírez

Curso 2019-2020

Índice general

CAPÍTULO 1
Introducción

El objetivo de esta memoria es el de describir los resultados obtenidos tras el diseño de dos técnicas metaheurísticas para resolver un determinado problema, escogido entre las propuestas del boletín de prácticas.

Una vez desarrolladas las soluciones metaheurísticas para resolver el problema, se procede a evaluar y contrastar la utilidad de ambos métodos, observando sus parámetros, el número de soluciones generadas, la calidad de la solución, etc.

En este caso, el tema que se va a abordarse es el del problema de las cartas, que consiste en separar, en dos pilas, una baraja de 10 cartas del mismo palo, numeradas del 1 al 10 sin repeticiones. De este modo, en una pila el sumatorio de los números de las cartas ha de acercarse lo máximo posible a 36, mientras que en la otra pila el producto ha de aproximarse a 360.

Para ello, se han aplicado las técnicas de Algoritmos Genéticos y Enfriamiento Simulado. El diseño e implementación de estas técnicas se ha realizado en el lenguaje de programación *Python*.

CAPÍTULO 2
Algoritmo Genético

Un algoritmo genético está basado en la evolución natural y la genética. Al igual que la población evoluciona de forma natural (mejorando su adaptación), un algoritmo genético lo hace a partir de una población de soluciones iniciales, intentando producir nuevas soluciones que sean mejores que las anteriores.

El proceso de evolución es guiado por decisiones probabilísticas basadas en la aptitud de los individuos (marcada por una función de evaluación o *fitness*) hasta obtener un buen individuo, esto es, una buena solución global al problema.

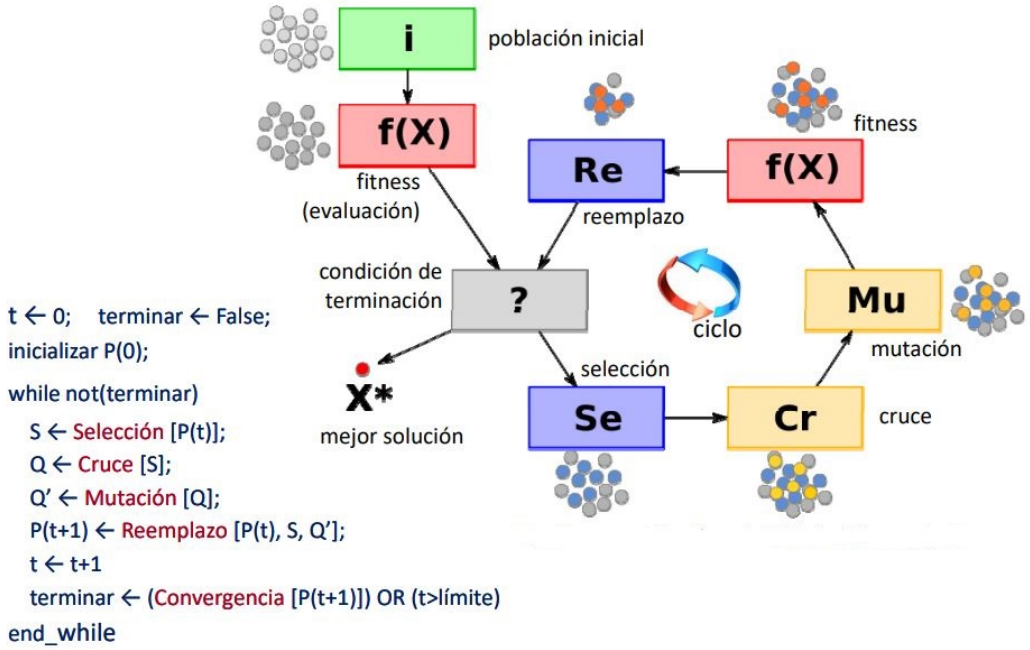


Figura 2.1: Estructura de algoritmo vista en clase de prácticas que se ha seguido para implementar nuestro algoritmo genético.

2.1 Diseño

El diseño general de nuestro algoritmo genético se ha implementado mediante un método en *Python* con cuatro parámetros de entrada, que son:

- *maxIter*. Establece el número de iteraciones máximas que ha de realizar nuestro algoritmo.

- *pobIni*. Establece el número de individuos que forman la población inicial.
- *probCruce*. Determina el porcentaje de la población que será seleccionada para el cruce (en tanto por uno).
- *probMutacion*. Determina el porcentaje de la nueva generación, originada en el cruce, que será seleccionada para la mutación (en tanto por uno).

La estructura de nuestro algoritmo sigue los pasos del visto en las clases de prácticas, como se puede apreciar en la figura 2.1.

2.1.1. Diseño del individuo. Codificación y decodificación

En el problema de las cartas, un posible individuo sería una distribución de las cartas entre las dos pilas. Como restricción se ha añadido que todas las cartas han de estar presentes, independientemente de si están en la primera o en la segunda pila. No es necesario que una pila tenga cartas, pero sí es cierto que esto influye a la hora de realizar el producto o el sumatorio, por lo que es un detalle a tener en cuenta. La primera pila es la que hemos escogido como la que ha de hacer el sumatorio de las cartas, mientras que la segunda se encarga del producto. No es necesario que las cartas estén ordenadas en una pila.

Haciendo uso de las técnicas proporcionadas por el lenguaje *Python*, se ha decidido emplear una tupla con dos listas como método de codificación de un individuo. De esta manera, el individuo que representa que las cartas 1, 3 y 5 se encuentran en la primera pila y que las cartas 2, 4 y 6 están en la segunda, se codificaría de esta forma:

$$individuo = ([1,3,5,7,9],[2,4,6,8,10]) \tag{2.1}$$

2.1.2. Función de evaluación

La función que hemos determinado es una *fitness* **f(x,y)**, donde **x** es la suma de la primera pila de cartas del individuo, e **y** el producto del segundo. Así, nuestra función se define como:

$$\textbf{f(x,y)} = Valor\_absoluto(36 - x) + Valor\_absoluto(360 - y) \tag{2.2}$$

De esta forma, tenemos una función que se pretende minimizar, de modo que la aptitud del individuo será óptima cuando **x** sea 36 e **y** sea 360.

2.1.3. Generación de la población inicial

Para crear la población inicial, el algoritmo toma el valor de *pobIni* que se le pasa como argumento y así sabe cuántos individuos la formarán.

Por una parte, para guardar esta primera población, se hace uso en *Python* de una lista de tuplas donde vamos almacenando las posibles soluciones.

Por otra parte, dentro de cada tupla (es decir, para cada individuo), se recorren los números del 1 al 10 y, para cada uno de ellos, se decide arbitrariamente en cuál de las dos listas de la tupla se guarda, es decir, en qué pila se pone la carta de ese número.

Por último, se comprueba si el individuo generado no se encuentra en la lista de tuplas, en cuyo caso lo añadimos. De esta manera, nos evitamos tener individuos repetidos en nuestra población de partida.

2.1.4. Selección, cruce, mutación y reemplazo

Antes de la selección de individuos que serán sometidos a cruce, primero ordenamos toda la población actual en función de la aptitud de cada uno, de mejor a peor. Posteriormente, seleccionamos, con un porcentaje de *probCruce*, los mejores individuos. Por ejemplo, si hay 100 individuos y *probCruce* es 0.8, se seleccionan los 80 mejores.

Una vez están seleccionados, el cruce que se ha realizado consiste en obtener una solución con la primera fila del padre y la segunda pila de la madre. Debido a que es altamente probable que se repitan números en ambas pilas, se recorre ambas listas eliminando repetidos. De igual forma, si un numero no está en ninguna lista, se añadirá a una de ellas. Esta eliminación y adición se hace de forma proporcional, de manera que no siempre se supriman los repetidos ni se añadan cartas a la misma pila.

Dentro de estos nuevos individuos, la mutación que se ha realizado es la de Intercambio Recíproco. Este consiste en seleccionar dos genes al azar y permutarlos. En nuestro caso, se hace entre un gen del conjunto del sumatorio y otro del producto, es decir, intercambiar una carta de la primera pila por otra de la segunda. El porcentaje de individuos que se selecciona para mutación viene determinado por *probMutacion*, como se ha explicado anteriormente.

Por último, se aplica reemplazo por Estado Estacionario, es decir, se descartan los peores individuos de la población para añadir a la nueva generación (tanto los que han mutado como los que no). Por tanto, el número de descartados es el número de nuevos individuos.

2.2 Implementación

En este apartado se van a comentar los detalles de la implementación de nuestro algoritmo genético. Para ello, se ha realizado un script desde cero en Python gracias a la versatilidad que ofrece este lenguaje de programación.

2.2.1. Esquema general del algoritmo genético

```
def genetico(maxIter, pobIni, probCruce, probMutacion):
    iteracion = 0
    terminar = False
    pueblo = iniciarPopulacho(pobIni)
    mejores = []

    while not terminar:
        seleccionados = seleccion(pueblo,probCruce)
        nuevaGeneracion = cruce(seleccionados)
        nuevaGeneracion = mutacion(nuevaGeneracion,probMutacion)
        pueblo = reemplazo(pueblo,nuevaGeneracion)
        iteracion += 1
        terminar = iteracion == maxIter or convergencia(pueblo)[0]
        preseleccion = []
        for individuo in pueblo:
            validez = aptitud(individuo)
            preseleccion.append((individuo, validez))
        preseleccion = sorted(preseleccion, key = lambda tup:tup[1])
        print("-----------------------------------------------------------")
        print("Iteracion ", iteracion)
        print("-----------------------------------------------------------")
```

```
22          if preseleccion[0] not in mejores:
23              mejores.append(preseleccion[0])
24              mensaje = "El mejor individuo por el momento es "+ str(mejores
                    [-1][0])+", con una aptitud de "+ str(mejores[-1][1])+ ",
                    encontrado en la iteraci n "+ str(iteracion)
25          print(mensaje)
26          print("\n")
27
28      print("-----------------------------------------------------------------")
29      print("El algoritmo ha finalizado")
30      print("-----------------------------------------------------------------")
31      if convergencia(pueblo)[0] == True:
32          res = "La solucion optima es "
33          return res+str(convergencia(pueblo)[1])
34      else:
35          preseleccion = []
36          for individuo in pueblo:
37              validez = aptitud(individuo)
38              preseleccion.append((individuo, validez))
39          preseleccion = sorted(preseleccion, key = lambda tup:tup[1])
40          res = "El mejor individuo es "
41          res1 = ", con un fitness de "
42          return res+str(preseleccion[0][0])+res1+str(aptitud(preseleccion[0][0])
                )
```

Listing 2.1: Esquema del algoritmo genético.

Como podemos observar, el algoritmo realiza el esquema tradicional de un genético, con la modificación de que, a cada iteración, devuelve la mejor solución que lleva hasta ahora y la iteración en la que se encontró dicha solución.

Una vez el algoritmo finaliza, devuelve la solución óptima, si la ha encontrado; o la mejor solución que ha sido capaz de obtener.

De esta forma, la función se ve influenciada por el número máximo de iteraciones, encontrar la solución óptima al problema o encontrar un posible mínimo local sin conseguir mejorar la menor aptitud del conjunto de soluciones halladas.

2.2.2. Método para la función de evaluación

Como hemos mencionado anteriormente, la función de evaluación tomará el valor del sumatorio y sacará su distancia numérica con 36, del mismo modo que procederá con el producto y 360. Finalmente, sumará ambos resultados. El esquema algorítmico es:

```
1  def aptitud(individuo):
2      sumatorio = 0
3      productorio = 1
4
5      for elem in individuo[0]:
6          sumatorio += elem
7      for elem in individuo[1]:
8          productorio *= elem
9
10     sumatorio = abs(36-sumatorio)
11     productorio = abs(360-productorio)
12
13     return sumatorio+productorio
```

Listing 2.2: Función de evaluación.

2.2.3. Método para la generación de la población inicial

```
def iniciarPopulacho(pobIni):
    pueblo = []
    while len(pueblo)<pobIni:
        tupla = ([], [])
        i = 1
        while i<11:
            aleatorio = randint(0,1)
            if aleatorio==0:
                tupla[0].append(i)
            else:
                tupla[1].append(i)
            i += 1
        if tupla not in pueblo:
            pueblo.append(tupla)
    return pueblo
```

Listing 2.3: Esquema de la generación inicial de la población.

En primer lugar, hay que comentar que la población inicial se ha generado de forma aleatoria entre las posibles operaciones disponibles.

Además, un aspecto interesante del inicio de la población es como se ha escogido el tamaño de la misma. Como veremos más adelante, el tamaño de la población inicial juega un papel fundamental a la hora de encontrar una buena solución para este problema, de modo que se ha decidido añadirlo como parámetro de entrada del algoritmo genético (*pobIni*).

2.2.4. Método de selección

La selección implementada se basa en el método elitista, en el que aquellos individuos con mayor *fitness* son elegidos para el posterior cruce. El esquema algorítmico se puede apreciar a continuación:

```
def seleccion(pueblo, probCruce):
    preseleccion = []
    for individuo in pueblo:
        validez = aptitud(individuo)
        preseleccion.append((individuo, validez))
    preseleccion = sorted(preseleccion, key = lambda tup:tup[1])

    res = []
    i = 0
    while i < probCruce*len(pueblo):
        res.append(preseleccion[i])
        i += 1
    return res
```

Listing 2.4: Selección elitista.

2.2.5. Método de cruce

El Cruce de Permutación es el que selecciona un conjunto de genes consecutivos del padre, se trasladan al hijo y se marcan esos genes en la madre, para posteriormente añadirlos, en orden, al hijo. En nuestro algoritmo se ha hecho una adaptación de este método,

de una forma más simple, en la cuál los genes del padre que pasan al hijo es la primera tupla, mientras que la segunda la hereda de la madre. El esquema algorítmico de esta implementación es:

```
def cruce(seleccionados):
    padre = seleccionados[0][0]
    hijos = []
    cruce1 = ([],[])
    for madre in seleccionados[1:]:
        cruce1 = (padre[0], madre[0][1])
        i = 1
        cont0 = 0
        cont1 = 0
        hijo = ([],[])
        while i<11:
            if (i in cruce1[0] and i in cruce1[1]) or (i not in hijo[0] and i
                not in hijo[1]):
                if cont0<cont1:
                    hijo[0].append(i)
                    cont0+=1
                else:
                    hijo[1].append(i)
                    cont1+=1
            elif i in cruce1[0]:
                hijo[0].append(i)
            elif i in cruce1[1]:
                hijo[1].append(i)
            i+=1
        hijos.append(hijo)
    return hijos
```

Listing 2.5: Cruce empleado en nuestro algoritmo genético.

2.2.6. Método de mutación

Como se ha comentado anteriormente, se ha implementado el mecanismo de mutación por Intercambio Recíproco. Su esquema algorítmico es:

```
def mutacion(nuevaGeneracion, probMutacion):
    i = 0
    while i < probMutacion*len(nuevaGeneracion):
        sel1 = random.choice(nuevaGeneracion[i][0])
        sel2 = random.choice(nuevaGeneracion[i][1])
        nuevaGeneracion[i][0].remove(sel1)
        nuevaGeneracion[i][1].remove(sel2)
        nuevaGeneracion[i][0].append(sel2)
        nuevaGeneracion[i][1].append(sel1)
        i+=1
    return nuevaGeneracion
```

Listing 2.6: Mutación por Intercambio Recíproco.

2.2.7. Método de reemplazo

Como se ha comentado anteriormente, se ha implementado el reemplazo por Estado Estacionario. Su esquema algorítmico es:

```
def reemplazo(pueblo, nuevaGeneracion):
    preseleccion = []
    for individuo in pueblo:
        validez = aptitud(individuo)
        preseleccion.append((individuo, validez))
    preseleccion = sorted(preseleccion, key = lambda tup:tup[1], reverse = True
        )

    i = 0
    while i < len(nuevaGeneracion):
        pueblo.remove(preseleccion[i][0])
        pueblo.append(nuevaGeneracion[i])
        i+=1

    return pueblo
```

Listing 2.7: Reemplazo por Estado Estacionario.

2.2.8. Método auxiliar para la convergencia

El objetivo de este método es el de comprobar si un individuo es o no óptimo, de manera que el algoritmo genético finalizará en caso afirmativo. Su esquema algorítmico es:

```
def convergencia(pueblo):
    for individuo in pueblo:
        if aptitud(individuo)==0:
            optimo = individuo
            return True, optimo
    return False, None
```

Listing 2.8: Convergencia.

2.3 Evaluación

2.3.1. Criterios de evaluación

Según nuestra implementación del algoritmo genético, el objetivo es minimizar la función de evaluación debido a que, para valores más cercanos a 36 y 360 (que es lo que persigue la solución), el valor de *fitness* será cada vez menor.

El tiempo de cómputo de nuestro algoritmo depende de varios factores, como puede ser el argumento *maxIter* o el resto de parámetros que, según su valor, hará que el algoritmo termine antes o después de lo previsto (por la convergencia, como se ve en el siguiente apartado).

2.3.2. Tamaño del problema y convergencia

Debido a que, en el peor de los casos, la talla del problema es factorial de n, para $n = 10$ cartas hay del orden de 3 millones de posibles soluciones.

Para que nuestro algoritmo converja, es necesario comprobar si se ha generado un individuo que satisface el problema, es decir, si su función de evaluación es 0. En caso afirmativo, el algoritmo finalizará devolviendo esa solución, independientemente de la iteración por la que se encuentre ejecutando.

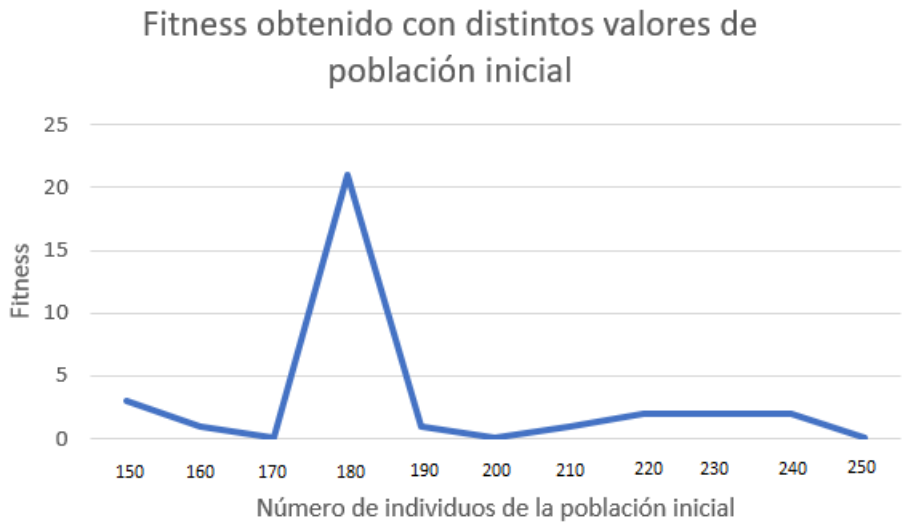


Figura 2.2: Evolución del valor de la función de evaluación al variar el parámetro que establece los individuos de la población inicial.

2.3.3. Parámetros de evaluación

Cada uno de los parámetros que le pasamos al algoritmo genético está relacionado con cada fase del proceso, a excepción del reemplazo. La población depende de *pobIni*, la selección y el cruce de *probCruce*, y la mutación de los nuevos individuos de *probMutacion*. De esta manera, se le da al usuario interacción con la mayor parte posible de la fase evolutiva del algoritmo y permite comprobar, con cada ejecución, cuál es el valor de cada uno que mejor se adapta para encontrar una buena solución al problema.

2.4 Conclusiones

Una vez que ejecutamos de forma continuada nuestro algoritmo, podemos observar que la solución se encuentra, en la mayoría de los casos, en las primeras iteraciones.

Para un valor grande de *pobIni*, la solución que se encuentra suele tener una aptitud menor. Esto puede deberse al hecho de que la talla del problema, a pesar de ser grande, no es de un tamaño desorbitado y es muy posible que, de manera arbitraria, el individuo que se genere en una población inicial sea precisamente el óptimo.

En definitiva, este parámetro *pobIni* es el que decide, en mayor medida, la aptitud de la solución que devuelva el algoritmo, más que el resto de argumentos de entrada.

Teniendo esto en cuenta, hemos tomado distintos valores para *pobIni* y hemos ejecutado el algoritmo genético manteniendo constantes el resto de parámetros. Para *maxIter* = 100, *probCruce* = 0.9 y *probMutacion* = 0.5, el rango de valores que hemos asignado a *pobIni* es 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250.

Los resultados obtenidos pueden observarse en la figura 2.2. A la vista de los resultados se observa que hay un mínimo con *pobIni* = 170 y con *pobIni* = 250. Además, ambas soluciones son óptimas, pues su aptitud es 0. En cuanto a tiempo de cómputo, se ha estimado que esas soluciones se han alcanzado, respectivamente, en 0.068 y 0.11 segundos.

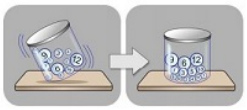
CAPÍTULO 3
Enfriamiento simulado

La idea básica del enfriamiento simulado es la de realizar una exploración más amplia del conjunto de soluciones al principio, con lo que se disminuye la probabilidad de caer en mínimos locales.

En el enfriamiento simulado hay que aceptar, con una determinada probabilidad, estados sucesores en los que la función de evaluación empeore, con el objetivo de poder salir de óptimos locales. Esta probabilidad disminuye conforme avanza la búsqueda.

De esta forma, la búsqueda de soluciones se concentra más en las etapas finales.

S_{ACTUAL} := $S_{INICIAL}$; S_{MEJOR} := S_{ACTUAL}; T := $T_{INICIAL}$; I=0;



Mientras *"Existan sucesores de* S_{ACTUAL}*"* y *"criterio_terminación=falso"* **hacer**

I=I+1;

S_{NUEVO} ← Operador-aleatorio (S_{ACTUAL}) *;movimiento aleatorio, no el mejor!*

Δf ← f(S_{NUEVO}) - f(S_{ACTUAL})

si Δf >0 entonces *{mejora la solución: se acepta el movimiento}*

S_{ACTUAL} ← S_{NUEVO}

Si S_{NUEVO} mejor que S_{MEJOR} entonces S_{MEJOR} := S_{NUEVO}

si no

S_{ACTUAL} ← S_{NUEVO} con probabilidad $e^{\Delta f/T}$

T = α (I, T) *; Actualizar T acorde planificación del enfriamiento, por ejemplo, T=T/(1+kT), siendo 0<k<1*

finMientras

Devolver S_{MEJOR}

Figura 3.1: Estructura de algoritmo vista en clase de prácticas que se ha seguido para implementar nuestro algoritmo de Enfriamiento Simulado.

3.1 Diseño

En enfriamiento simulado, primero se define una función f, que en nuestro caso es la misma función de evaluación que habíamos definido en el algoritmo genético.

En cada iteración de búsqueda local, en vez de escoger el mejor sucesor, se elige un sucesor aleatorio. Si el sucesor mejora la situación, esto es, si $\triangle f \geq 0$, entonces se acepta.

Si no mejora la situación ($\triangle f < 0$), el sucesor se acepta con una probabilidad menor a 1. Esta probabilidad, típicamente, es:

$$probabilidad = e^{\triangle f/T} \tag{3.1}$$

Esta probabilidad se decrementa con el empeoramiento de la evaluación o con la temperatura (T), que va bajando conforme avanza la búsqueda.

Una función de actualización de T que suele usarse y que vamos a emplear en nuestro algoritmo es:

$$T = T/(1 + kT) \tag{3.2}$$

El diseño general de nuestro algoritmo de enfriamiento simulado se ha implementado mediante un método en *Python* con dos parámetros de entrada, que son:

- *t_inicial*. Es la temperatura al principio de la ejecución del algoritmo.
- *k*. Es la constante utilizada en la actualización de la temperatura.

La estructura de nuestro algoritmo sigue los pasos del visto en las clases de prácticas, como se puede apreciar en la figura 3.1.

3.2 Implementación

En este apartado se van a comentar los detalles de la implementación de nuestro algoritmo de enfriamiento simulado. Para ello, se ha realizado un script en Python de forma similar a como lo hicimos anteriormente para el algoritmo genético.

3.2.1. Esquema general

El esquema algorítmico para el enfriamiento simulado puede resumirse de esta forma: En el caso de obtener una solución que mejore el *fitness*, se acepta. En caso de que la solución empeore la aptitud, se acepta con una determinada probabilidad, que depende de una cierta temperatura. Inicialmente para valores altos de la temperatura la mayoría de sucesores serán aceptados. A medida que la temperatura vaya decreciendo, disminuirá la probabilidad de aceptar un sucesor.

El esquema algorítmico general de nuestro enfriamiento simulado es:

```
1  def enfriamiento_simulado(t_inicial, k):
2      s_actual = iniciar_individuo()
3      sucesores = suc(s_actual)
4      s_mejor = s_actual
5      t = t_inicial
6      i = 0
7      while len(sucesores)!=0 and not convergencia(i, s_actual):
8          i=i+1
9          print("------------------------------------------------------------")
10         print("Iteracion ", i)
11         print("------------------------------------------------------------")
12         s_nuevo = seleccionar_sucesor(sucesores)
13         incremento = aptitud(s_actual)-aptitud(s_nuevo)
14         if incremento>0:
```

```
            s_actual = s_nuevo
            sucesores = suc(s_actual)
            if aptitud(s_nuevo)<aptitud(s_mejor):
                s_mejor = s_nuevo
        else:
            if random.random() < math.e**(incremento/t):
                s_actual = s_nuevo
                sucesores = suc(s_actual)
                t = actualizarT(i, k, t)
        print("La solucion actual es ", s_actual, ", con un fitness de ",
            aptitud(s_actual))
    return "\nLa solucion es "+str(s_mejor)+", con un fitness de "+str(aptitud(
        s_mejor))
```

Listing 3.1: Esquema enfriamiento simulado

3.2.2. Generación y selección de sucesores

Por un lado, para la generación de sucesores, se ha modificado el método de generación de la población inicial visto en el algoritmo genético, de manera que ahora el número de sucesores viene determinado por el tamaño de la solución actual. En concreto, por el tamaño de la primera componente de la tupla, que representa la primera pila de cartas.

Su esquema algorítmico es:

```
def suc(s_actual):
    sucesores = []
    while len(sucesores)<2*len(s_actual[0]):
        tupla = ([], [])
        i = 1
        while i<11:
            aleatorio = randint(0,1)
            if aleatorio==0:
                tupla[0].append(i)
            else:
                tupla[1].append(i)
            i += 1
        sucesores.append(tupla)
    return sucesores
```

Listing 3.2: Generación de sucesores.

Por otro lado, para escoger un sucesor, se realiza una selección arbitraria del conjunto de sucesores. El esquema algorítmico que se ha implementado es:

```
def suc(s_actual):
    sucesores = []
    while len(sucesores)<2*len(s_actual[0]):
        tupla = ([], [])
        i = 1
        while i<11:
            aleatorio = randint(0,1)
            if aleatorio==0:
                tupla[0].append(i)
            else:
                tupla[1].append(i)
            i += 1
        sucesores.append(tupla)
    return sucesores
```

Listing 3.3: Selección arbitraria de un sucesor.

3.2.3. Actualizar temperatura

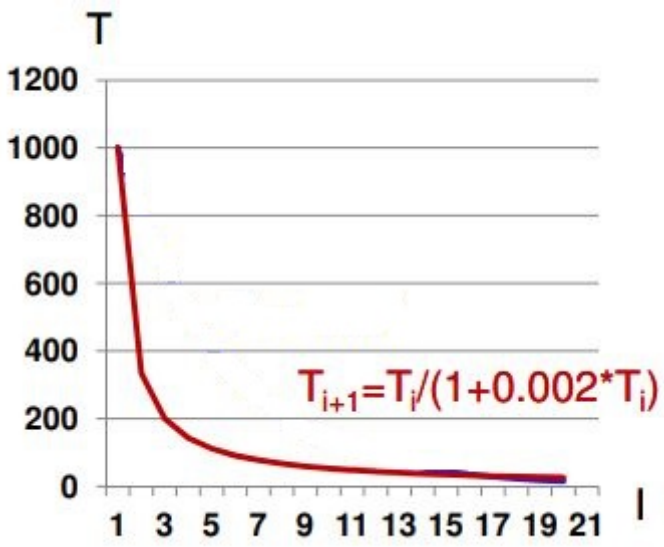


Figura 3.2: Evolución de la temperatura con la fórmula propuesta.

Como se ha comentado anteriormente,la actualización de la temperatura se ha implementado mediante el método:

$$T = T/(1 + kT) \tag{3.3}$$

Este método permite obtener una variación de temperatura como la que podemos apreciar en la gráfica de la figura 3.2, suponiendo valores de k entre 0 y 1.

3.2.4. Método auxiliar para la convergencia

A diferencia del algoritmo genético, en enfriamiento simulado la convergencia se alcanza cuando se ha realizado un número máximo de iteraciones, además de comprobar si el individuo que representa la solución actual es apto. Con que una de ambas condiciones se cumpla, el algoritmo converge. Su esquema algorítmico es:

```
1 def convergencia(i, s_actual):
2     if i>5000 or aptitud(s_actual)==0:
3         return True
4     else:
5         return False
```

Listing 3.4: Convergencia.

3.3 Evaluación

3.3.1. Criterios de evaluación

Según nuestra implementación del enfriamiento simulado, el objetivo sigue siendo minimizar la función de evaluación, debido a que se trata del mismo problema que en el algoritmo genético.

El tiempo de cómputo de nuestro algoritmo depende del método que hemos implementado para generar sucesores, debido a que el bucle principal recorre esa lista de sucesores. En nuestro caso, generamos 2*n sucesores, donde n es el tamaño de la primera componente de la tupla de la solución actual, es decir, el número de cartas de la primera pila.

3.3.2. Tamaño del problema y convergencia

La talla del problema, como se ha comentado anteriormente, es factorial de n, con n = número de cartas. Para 10 cartas hay del orden de 3 millones de posibles soluciones.

Para que nuestro algoritmo converja se ha de cumplir una de dos condiciones:

- O bien que se realicen un número máximo de iteraciones, que en nuestro caso las hemos establecido a 5000.
- O bien que la solución actual sea óptima, es decir, en nuestro problema, que su aptitud sea 0.

3.3.3. Parámetros de evaluación

Por otra parte, en la evaluación del enfriamiento simulado se ha realizado una comparativa con los resultados que nos proporcionan distintos valores de los parámetros del algoritmo. Estos valores son:

- Para temperatura, 1, 10, 100, 1000, 10000.
- Para k, 0.001, 0.01, 0.1, 1, 10.

Los resultados obtenidos pueden observarse en la figura 3.3. A la vista de los resultados se observa que hay un mínimo con T = 10000 y k = 1. Además, esta solución es óptima, pues su aptitud es 0. En cuanto a tiempo de cómputo, se ha estimado que esa solución se ha alcanzado en 0.029 segundos.

3.4 Conclusiones

A la vista de los resultados anteriores, se podría decir que, a mayor temperatura inicial, mejor función de evaluación da la solución obtenida. Esto se debe a que hay una mayor dispersión de los datos, por lo que hay más posibilidades de alcanzar un óptimo local mejor a cada iteración.

Figura 3.3: Evolución del valor de la función de evaluación al variar la temperatura y k.

CAPÍTULO 4
Conclusiones finales

Observando lo expuesto anteriormente podemos extraer varias conclusiones.

En primer lugar, se han implementado dos programas en Python para solucionar el problema de las cartas mediante las técnicas metaheurísticas de enfriamiento simulado y algoritmos genéticos. Durante el desarrollo y evaluación de las soluciones se ha observado que la implementación de los algoritmos es sencilla mientras que el ajuste de los parámetros es bastante tedioso y, a la hora de encontrar una mejor solución, es el paso definitivo.

Por otra parte, se ha observado que en nuestro caso particular los algoritmos genéticos han sido capaces de obtener mejores resultados, en muchos de los casos, soluciones óptimas (sobre todo con un valor alto del parámetro de población inicial). Además, un aspecto interesante a destacar es la velocidad con la que ambos algoritmos son capaces de obtener una solución factible con un buen resultado en la función objetivo.

Por último, recalcar que este trabajo ha permitido obtener una idea más amplia de las técnicas metaheurísticas evolutivas e iterativas, además de como aplicarlas a posibles problemas de la vida real.