Actividad No. 3
Solución de problemas mediante Ascensión de Colinas

Adrian González Pardo

4 de octubre de 2020

1. Ventajas y desventajas de la Ascensión de Colinas

Inicialmente el espacio de soluciones de un problema puede ser representado en una grafíca en R^2, R^3, hasta R^n por lo que podemos explorar su espacio mediante una heuristica la cual es generalmente el Ascensio de Colinas el cual proporciona las siguientes ventajas y desventajas:

Ventajas	Desventajas
Permite realizar un número menor de iteraciones con respecto a la exploración del espacio en modalidad de fuerza bruta	El resultado puede quedarse en un minimo local sin que sea el minimo del espacio
Permite moverse en el espacio de diferentes formas, permitiendo que encuentre un minimo	Dependiendo de la implementación del algoritmo puede que que llegue al minimo o no, pero al final arroja una solución
Dependiendo de la implementación puede que tenga un número distinto de iteraciones	En caso de ser un espacio muy grande puede que la implementación, salto o movimiento entre el espacio demore demasiado

2. Pseudocódigo del Algoritmo Steepest Ascent Hill Climbing (Ascenso Más Empinado Escalada de Colinas)(SAHC)

```
/* Caso en el que el nodo tiene 2 vertices */
nodeActual=Set_of_nodes[random]
steps=M
N=0
setOfPaths={}
Path=[]
while(N<steps){

  if(value(nodeIzq(nodeActual))>value(nodeActual) || value(nodeDer(nodeActual))>value(nodeActual))
    {
    /* En esta parte puede eliminarse esta seccion con hacer uso de un
     * nuevo proceso en el cual mute a dos caminos distintos en memoria
     * y reduccion en tiempo */
    if(value(nodeIzq(nodeActual))>value(nodeActual)){
      Path.push(nodeActual)
      nodeActual=nodeIzq(nodeActual)
    }else{
      Path.push(nodeActual)
      nodeActual=nodeDer(nodeActual)
    }
    N++
  }else{
    setOfPaths.add(Path)
    Path=[]
```

```
    nodeActual=Set_of_nodes[random]
    N++
  }
}
show(Path)
```

3. Next-Ascent Hill-Climbing (Próxima Ascenso Escalado)(NAHC)

```
/* Caso en el que el nodo tiene 2 vertices */
nodeActual=Set_of_nodes[random]
steps=M
N=0
setOfPaths={}
Path=[]
while(N<steps){
  if(value(nodeIzq(nodeActual))>value(nodeActual)){
    Path.push(nodeActual)
    nodeActual=nodeIzq(nodeActual)
    N++
  }else if(value(nodeDer(nodeActual))>value(nodeActual)){
    Path.push(nodeActual)
    nodeActual=nodeDer(nodeActual)
    N++
  }else{
    setOfPaths.add(Path)
    Path=[]
    nodeActual=Set_of_nodes[random]
    N++
  }
}
show(Path)
```

4. Random-Mutation Hill-Climbing (Ascenso Por Mutación Aleatoria)(RMHC)

```
/* Caso en el que el nodo tiene 2 vertices */
nodeActual=Set_of_nodes[random]
steps=M
N=0
setOfPaths={}
Path=[]
while(N<steps){
  if(rand()%2==0){
    Path.push(nodeActual)
    nodeActual=nodeIzq(nodeActual)
    N++
  }else{
    Path.push(nodeActual)
    nodeActual=nodeDer(nodeActual)
    N++
  }
}
show(Path)
```

5. Resultados de Forrest y Mitchell

Si bien en sus conclusiones nos podemos dar cuenta en cuestios de los algoritmos heuristicos SAHC y NAHC en la busqueda de soluciones de un espacio bidimensional no encontraron la zona optima, el algoritmo RMHC lo encontro en un número menor de iteraciones con respecto a los algoritmos geneticos que se implementaron en la investigación, esto pasa en relación a que el algoritmo RMHC explora el espacio de soluciones en este caso digamos o aproximemos en grafos es más sencillo tomar la decisión sobre que nodo iterar sin realizar tantas comparaciones.

6. Aplicaciones de Algoritmos de Ascenso de Colinas

Si bien en muchas ocasiones estos algoritmos se ven como una caja negra a la hora de ser utilizados en muchos modulos o toolbox de lenguajes de programación estos estan intimamente aplicados en el cálculo de gradiente descendente de N variables para optimizar un aprendizaje supervisado o no supervisado, donde sobre cada iteración se realiza el calculo de un nuevo movimiento en el espacio de soluciones, por otro lado tambien este tipo de algoritmo esta implementado en algunas aplicaciones en las que se realiza la planificación de rutas de transporte e incluso en sistemas de reconocimiento de objetos 3D, una vez más retomando los terminos del Aprendizaje de Maquina podemos encontrar desde regresión lineal, polinomial y logistica.

7. Aplicación de RMHC en distintos problemas

7.1. Knapsack Problem

```
knapsack_max_peso=S
objects={(p,b)_0,(p,b)_1,...,(p,b)_n}
# Conjunto de datos donde
# => p es el peso del objeto
# => b es el beneficio del objeto
sol_i=[].push(objects[random%objects.lenght])
lim_iterator=M
iterator=0
while iterator<lim_iterator || get_best_solution(sol_i) do
  # get_best_solution es una funcion en la cual aproxima a un valor
  #  numerico R el cual tiene un rango para decir que la solucion
  #  es la mejor que puede dar el programa
  obj=objects[random%objects.lenght]
  if obj not in sol_i
    # Verifica si el objeto que fue iterado esta o no esta en la lista
    #  de objetos de la mochila para verificar si pueden ser agregados
    if obj.p+sol_i.sumP()<=knapsack_max_peso
      # Verifica si la suma de los pesos del objeto y del contenido
      #  de la mochila no rebasa el peso maximo  de la mochila para
      #  ser agregado
      sol_i.push(obj)
    end
  end
  iterator=iterator+1
end
show(sol_i)
# Muestra la solucion obtenida por el programa el cual busca maximizar
# la utilidad/beneficio de la mochila
```

7.2. Travel Salesman Problem

```
cities={(i,[j,c])_0,(i,[j,c])_1,...,(i,[j,c])_n}
# Conjunto de ciudades donde
# => i es el indice del grafo (ciudad)
#     donde entre parentesis son
# => => j es el indice del siguiente grafo (ciudad)
# => => c es el costo de trasladarse de i a j
i_city=cities[random%cities.lenght]
path=[i_city]
lim_iterator=M
iterator=0
all_cities=cities.lenght-1
c_cities=0
use_index=[]
stuck_city=false
while (c_cities<all_cities || iterator<lim_iterator) && !stuck_city do
  vecino=i_city.get_vecinos()
  index=random%vecino.lenght
  if index not in use_index && use_index.lenght<vecino.lenght-1
    # Verifica si el indice no es repetitivo con respecto al arreglo
    #  y verifica si la ciudad a viajar no rebasa el limite, en cualquier caso
    #  esto genera que el viajero se quedo atascado ya viajo a todas
    #  las ciudades j
```

```
    use_index.push(index)
    city=vecino[use_index.last].city
    if city not in path
      # Verifica si la ciudad no esta en el camino del viajero
      i_city=city
      path.push(city)
      c_cities=c_cities+1
      use_index=[]
    end
  else
    stuck_city=true
  end
  iterator=iterator+1
end
showPath(path)
# Muestra el camino recorrido del viajero
# y a su vez muestra el costo total en el cual se desea sea poco
```

7.3. Obtencion de minimos de la función f(x)

Función:

$$f(x) = \sum_{i=1}^{D} x_i^2 \quad con \quad x_i \in [-10, 10]$$

```
t_dim=D>=1
espace=[]
p=[0,0,...,0]
# p es el punto de t_dim dimensiones el cual contendra los puntos
#  al menos en 0
espace.push(p)
lim_iterator=M
iterator=0
num_minimos=1
while iterator<lim_iterator || get_num_esperado_minimos(num_minimos) do
  # get_num_esperado_minimos es una funcion la cual establece un
  #  numero determinado de minimos ya que la funcion en D dimensiones
  #  es muy dificil de explorar
  xj_select=random%t_dim
  for i=0;i<t_dim;i++
    # Genera cualquier combinacion de puntos de [-10,10] para cada xi
    p[i]=random%10
    p[i]=(random%2==0)?(p[i]):(-p[i])
  end
  # Como bien sabemos es una funcion cuadratica en cada valor
  #  xi independiente por lo tanto solo es necesario modificar
  #  xj ya que ahi es donde existe el valor minimo de esa variables
  p[xj_select]=0
  if p not in espace
    # Verifica si el punto de D dimensiones no existe en el espacio
    #  de soluciones propuesto para evitar redudancia de datos
    espace.push(p)
    num_minimos=num_minimos+1
  end
  iterator=iterator+1

end
show_points(espace)
```