# IA 19/29 Borja López Pineda

## Algoritmo Hill Climb (ascenso de colina)

El algoritmo Hill Climb es un algoritmo de búsqueda no exhaustivo, lo que significa que no prueba todas las posibles soluciones. No garantiza la solución óptima. Su funcionamiento es sencillo: parte de una posición aleatoria y se mueve hacia la posición adyacente (vecino) que obtiene le mejor resultado. El objetivo puede ser encontrar el valor máximo o el mínimo, en los siguientes pseudocódigos el objetivo es el valor máximo.

Estructura base:

```
Función HillClimb() : posicion
      iteraciones <- 0
      mejorVecino <- posicionAleatoria()
      Haz
            actual <- mejorVecino
            vecinos <- generarVecinos(actual)
            ParaCada vecino en vecinos haz
                  Si puntuación(vecino) > puntuación(mejorVecino) haz
                        mejorVecino <- vecino
                  FinSi
            FinParaCada
            iteraciones <- iteraciones + 1
      Mientras No (mejorVecino = actual) y iteraciones < MAX_ITERACIONES
      Devolver mejorVecino
FinFunción
```

MAX_ITERACIONES es una constante que define el número máximo de saltos que puede dar el algoritmo antes de forzar su detención. Esto se hace para evitar que se estanque en una meseta. En caso contrario el algoritmo se detendrá cuando encuentre un máximo ya sea relativo o absoluto, detectado porque no hay ningún vecino mejor que la posición actual.

Mejora estocástica:

Esta mejora consiste en no seleccionar al mejor de los vecinos, sino un conjunto de mejores vecinos y, de entre ellos, escoger uno aleatorio. Al introducir cierta aleatoriedad nos acercamos a obtener soluciones que se alejan del máximo local evidente al que estamos condicionados principalmente por la posición desde la que se empieza.

```
Función HillClimb() : posicion
      iteraciones <- 0
      mejorVecino <- posicionAleatoria()
      Haz
            actual <- mejorVecino
            vecinos <- generarVecinos(actual)
            vejoresVecinos <- ConjuntoVacio()
            ParaCada vecino en vecinos haz
                  Si puntuación(vecino) > puntuación(actual) haz
                        mejoresVecinos.agregar(vecino)
                  FinSi
            FinParaCada
            mejorVecino <- ElegirAleatorio(mejoresVecinos)
            iteraciones <- iteraciones + 1
      Mientras No (mejorVecino = actual) y iteraciones < MAX_ITERACIONES
      Devolver mejorVecino
FinFunción
```

En este caso el conjunto de vecinos entre los que elijo aleatoriamente se forma con los que tienen mayor puntuación que la posición actual. Otra forma posible de aplicar esta modificación es quedarse con los que superan la media o un percentil determinado.

Mejora Primera opción:

Es una mejora que se centra en aumentar la eficiencia, reducir el tiempo de ejecución. En lugar de buscar exhaustivamente al mejor vecino, escoge al primero que encuentre que sea mejor que el actual. Puede ser útil cuando el número de vecinos hace intratable al problema. Por ejemplo, cuando se mapean funciones en altas dimensiones.

```
Función HillClimb() : posicion
      mejorVecino <- posicionAleatoria()
      Haz
            actual <- mejorVecino
            Mientras haySiguienteVecino(actual) y actual = mejorVecino haz
                  vecino <- siguienteVecino(actual)
                  Si puntuación(vecino) > puntuación(actual) haz
                        mejorVecino <- vecino
                  FinSi
            FinMientras
      Mientras No (mejorVecino = actual)
      Devolver mejorVecino
FinFunción
```

Por reducir más el número de instrucciones del algoritmo he eliminado la medida de seguridad que limite el número de iteraciones.

En este ejemplo he utilizado las funciones haySiguienteVecino y siguienteVecino

Mejora Reinicio aleatorio:

El principal problema de este algoritmo es que se estanca fácilmente en máximos locales y no garantiza la solución óptima. A fin de encontrar el máximo absoluto, esta mejora repite el proceso de búsqueda cambiando el punto de inicio. En este algoritmo el lugar donde inicia la búsqueda influencia en gran medida la solución que se obtiene, de forma que repetir el proceso permite una mejor solución al quedarnos con la solución más alta de cada intento.

```
Función HillClimb() : posicion
	mejorPosicion <- posicionAleatoria()
	Para i <- 1 hasta N_INTENTOS paso i <- i + 1 haz
		iteraciones <- 0
		mejorVecino <- posicionAleatoria()
		Haz
			actual <- mejorVecino
			Mientras haySiguienteVecino(actual) y actual = mejorVecino Haz
				vecino <- siguienteVecino(actual)
				Si puntuación(vecino) > puntuación(mejorVecino) haz
					mejorVecino <- vecino
				FinSi
			FinMientras
			iteraciones <- iteraciones + 1
		Mientras No (mejorVecino = actual) y iteraciones < MAX_ITERACIONES
		Si puntuación(mejorVecino) > puntuación(mejorPosicionDeTodos) haz
			mejorPosicion <- mejorVecino
		FinSi
	FinPara
	Devolver mejorPosicion
FinFunción
```

N_INTENTOS es una variable que define el número de veces que se repite el proceso de búsqueda.

## Algoritmo Simulating Annealing (Templado simulado):

Se trata de una mejora de Hill Climb inspirada en la cristalización del metal. Cambia la forma de elegir al sucesor para introducir cierto grado de aleatoriedad que disminuye con el tiempo. De la misma forma que el patrón de cristalizado se estabiliza y ralentiza en un metal que se solidifica.

El sucesor se elige de forma aleatoria. En caso de ser mejor se toma directamente. En caso de ser peor, se toma con una probabilidad, si no se elige se vuelve a tomar otro sucesor de forma aleatoria.

La probabilidad de coger un sucesor peor que el actual depende de la diferencia en la puntuación (como de relativamente malo sea) y de una función temperatura. La función temperatura decrece linealmente (aunque existen variaciones) con el tiempo. La probabilidad se calcula con la siguiente formula:

$$P(dif, temp) = e^{dif/temp}$$

De esta forma para una diferencia constante el descenso de la probabilidad se desacelera hasta estabilizarse y la probabilidad ya está en el rango [0-1] porque dif siempre es negativo ya que se calcula como puntuación(sucesor) - puntuación(actual)

```
Función SimulatingAnnealing() : posicion
      temperatura <- TEMPERATURA_INICIAL
      mejorVecino <- posicionAleatoria()
      Haz
            actual <- mejorVecino
            Mientras haySiguienteVecino(actual) y actual = mejorVecino haz
                  vecino <- siguienteVecino(actual)
                  Si puntuación(vecino) > puntuación(actual) haz
                        mejorVecino <- actual
                  SiNo
                        Si escojo(actual, vecino, temperatura)
                              mejorVecino <- actual
                        FinSi
                  FinSi
            FinParaCada
            temperatura <- temperatura – DECREMENTO_TEMPERATURA
      Mientras No (mejorVecino = actual)
      Devolver mejorVecino
FinFunción
```

```
Función escojo(actual, vecino, temperatura) : booleano
      diferencia <- puntuación(vecino) – puntuación(actual)
      aleatorio <- numeroAleatorio()/MAXIMO_ALEATORIO
      probabilidad <- exp(diferencia/temperatura)
      Devolver aleatorio < probabilidad
FinFunción
```

TEMPERATURA_INICIAL y DECREMENTO_TEMPERATURA son constantes que deben ser ajustadas hasta obtener buenos resultados. Dependerá del rango de valores que pueda tomar la función puntuacion.

La función escojo necesita generar un numero decimal aleatorio entre 0 y 1, como numeroAleatorio() en códigos anteriores devolvía números enteros he introducido la constante MAXIMO_ALEATORIO.

## Genetic Algorithm (Algoritmo genético)

Este algoritmo genera individuos de forma aleatoria, combina los mejores para generar nuevos individuos y repite el proceso. La configuración de un individuo se almacena en sus genes, los genes están formados por cromosomas individuales. Combinar dos individuos significa dividir por la misma altura los genes de los individuos y generar las dos nuevas posibles combinaciones.

La emparejarían se realiza de forma aleatorio y el número de individuos no cambia. Lo ideal es que las generaciones converjan, pero podrían hacerlo sobre un máximo local. Para solucionar ese problema, se introduce cierta aleatoriedad en la selección de los mejores individuos. A parte, se pueden producir mutaciones, en las que un cromosoma cambia de forma aleatoria.

Estructura base:

```
Función reproducir(padreA, padreB) : individuo, individuo
      p <- numeroAleatorio() modulo N_CROMOSOMAS
      parteA1 <- padreA.genes.subcadena(0, p)
      parteA2 <- padreA.genes.subcadena(p+1, N_ CROMOSOMAS-1)
      parteB1 <- padreB.genes.subcadena(0, p)
      parteB2 <- padreB.genes.subcadena(p+1, N_ CROMOSOMAS-1)
      hijo1 <- concatenar(parteA1, parteB2)
      hijo2 <- concatenar(parteA2, parteB1)
      Devolver hijo1, hijo2
FinFunción
```

N_CROMOSOMAS es el número de elementos en los genes de un individuos. El atributo genes de un individuo devuelve una cadena con sus cromosomas. subcadena de cromosomas divide la cadena (en este ejemplo comienza desde el 0). concatenar permite crear nuevos individuos a partir de dos fragmentos de cadenas.

```
Función GeneticAlgorithm() : individuo
      población <- generarPoblacionAleatoria()
      iteraciones <- 0
      Mientras iteraciones < MAX_ITERACIONES haz
            puntuaciones <- puntuación(población)
            población <- ordenar(población, puntuaciones)
            población <- cribar(contar(población) / 2)
            población <- aleatorizar(población)
            nuevaPoblación <- poblacionVacia()
            Para i <- 0 hasta i < contar(población) paso i <- i +2 haz
                  nuevaPoblación.agregar(reproducir(población[i],
      población[i+1]))
            FinPara
            Población <- nuevaPoblación
            iteraciones <- iteraciones + 1
      FinMientras
      puntuaciones <- puntuación(poblacion)
      población <- ordenar(población, puntuaciones)
      Devolver población[0]
FinFunción
```

generarPoblacionAleatoria siempre devuelve un número par de individuos.

En puntuaciones se guarda la puntación de cada individuo en el mismo orden que se almacenan en población. ordenar devuelve el primer conjunto ordenado por los valores del segundo conjunto.

## Resultados.

Se han intentado resolver dos problemas.

El primero trata de hallar el máximo de una función con dos parámetros. En esta función todos los máximos locales tienen la misma altura. En este caso el algoritmo HillClimb es el que mejor resuelve el problema.

El segundo es encontrar el pixel más blanco en una imagen en escala de grises con ruido perlin. Aquí el algoritmo HillClimb obtiene soluciones mucho peores que el resto, aunque su mejora Reinicio Aleatorio logra aumentar un poco el resultado. El algoritmo Simulated Annealing y el genético, especialmente este último, obtienen resultados muy parecidos a una busqueda completa. La pega es que requieren ajustar sus parámetros correctamente para el problema en cuestión.

Se adjuntan imágenes generadas a partir de los puntos que comprueba cada algoritmo.