UNIVERSIDAD DE GRANADA

Práctica 3.b

Enfriamiento simulado, búsqueda local reiterada y evolución diferencial para el APC.

Mario Román García

3 de junio de 2018

- DNI: 77145669N
- Email: mromang08@correo.ugr.es

- Grupo 1 (Lunes de 17:30 a 19:30)
- Algoritmos: ES, ILS, DE/rand/1, DE/current-to-best/1
- Metaheurísticas, 2017-2018.

Índice

1. Aplicación de algoritmos

1.1. Esquema de representación de soluciones

Como hemos tratado anteriormente, una solución del problema viene dada por un vector de pesos. Nuestra representación específica de una solución vendrá dada por una lista de valores reales.

$$\{w_i\}_{0 \leq i \leq n} = (w_1, w_2, \ldots, w_n)$$

Donde n será el número de características que tenga nuestro conjunto de datos y w_i. En nuestra implementación específica, los pesos vendrán dados por un vector contiguo en memoria de valores en coma flotante de precisión doble en 64 bits; para facilitar su tratamiento, en ciertos algoritmos se traducirán a listas enlazadas, pero su interpretación matemática será siempre constante y se describe explícitamente en la función objetivo.

1.2. Clasificador 1-NN, función objetivo

Se define la **distancia con pesos** $\{w_i\}$ entre dos vectores t y s como

$$\text{dist}(w, t, s) = \sum_{i=0}^{n} w_i (t_i - s_i)^2.$$

Y nuestro clasificador kNN para unos pesos $\{w_i\}$ consiste en devolver la clase del punto que minimiza la distancia. Es decir, es una implementación de un clasificador 1NN, que para cada instancia devuelve la clase de su vecino más cercano. En el caso de la función objetivo, al no tener un conjunto de test separado, usamos la técnica de *leave-one-out*.

Algorithm 1 Función objetivo (w : Pesos, T : Training)

1: $\text{Obj}(w, T) = \alpha \cdot \text{precision}(w, T) + (1 - \alpha)\text{simplicity}(w)$
2: $\text{TasaRed}(w) = \text{length}[x < 0.2 \mid x \in w]/\text{length } w$
3: $\text{TasaClas}(w, T) = \sum_{t \in T}(\text{knn}(w, T - t, t) == s.Clase)/\text{length } s$
4: $\text{knn}(w, T, t) = (\text{minimizador}_{t' \in T}(\text{dist}^2(\text{trunca}(w), t', t))).Clase$
5: $\text{trunca}(w) = \{0 \text{ si } w_i < 0.2; \quad w_i \text{ en otro caso } \mid w_i \in w\}$

En el código original esta función objetivo aparecerá implementada dos veces: una vez para el puntuador de los algoritmos y otra vez para la función objetivo. Esta duplicación tiene como ventaja que separa completamente las partes de evaluación del código de los algoritmos, reduciendo la posibilidad de error. Además,

- la implementación para evaluación es corta y es más fácil de verificar que está escrita correctamente;
- mientras que la implementación de la función objetivo está fuertemente optimizada, usando paralelismo, pero en caso de que tuviera cualquier error, eso no se vería reflejado en las puntuaciones de los algoritmos.

Es importante notar que la formulación original del problema no asume que todas las variables sean reales, y en el caso en el que son discretas asume una distancia de Hamming que simplemente indica si las dos características son exactamente iguales.

$$\text{dist}(a,b) = \delta_{ab} = \begin{cases} 1 & \text{si } a = b \\ 0 & \text{si } a \neq b \end{cases}$$

En los conjuntos de datos que trataremos en este análisis, tenemos de hecho que todas las características vienen dadas por reales, y nuestra implementación, aunque sería fácilmente extensible, no tratará el otro caso explícitamente.

1.3. Generación de soluciones aleatorias

La generación de una **solución aleatoria inicial** se realiza llamando repetidas veces a las librerías del lenguaje [Yor16]. Estas proporcionan funciones que nos permiten clacular listas potencialmente infinitas de números reales distribuidos uniformemente en el intervalo $[0, 1]$. Además, usamos replicamos el proceso aleatorio con mónadas para poder generar varios individuos cuidando que sean muestras independientes bajo el mismo generador aleatorio.

Algorithm 2 Solución inicial (t : Training)

1: solInicial(t) = tomaLos (nAttr(t)) primerosDe aleatorioUniforme$(0.0, 1.0)$
2: PoblInicial$(n = 30, t)$ = replica n solInicial

1.4. Descripción del algoritmo de búsqueda local

El algoritmo de búsqueda local que usaremos en el algoritmo de enfriamiento simulado y en la búsqueda local iterada proviene de la implementación previa de la búsqueda local que desarrollamos en la primera práctica.

Usamos como operador de generación de vecino la mutación de una solución que creamos en la primera práctica, que modifica una componente aleatoria tomando un ε entre $(0, \sigma)$, donde hemos determinado $\sigma = 0.3$.

Algorithm 3 Generación de vecino (w : Pesos, σ : Varianza)

1: $\varepsilon \leftarrow$ realAleatorioEntre$(0, \sigma)$
2: indx $\leftarrow$ enteroAleatorioEntre$(0,$ nAtributos$(a) - 1)$
3: mutacion(ε, i, w) = truncaEntre0y1 ajustaIndice indx $(+\varepsilon)$ w

El método de exploración del entorno consulta si la solución creada es mejor que la anterior y, en el caso de que lo sea, cambia la solución por ella. Nótese que en el caso del Enfriamiento Simulado, habrá que considerar además la condición de la temperatura que permite cambiar la solución por una peor. Normalmente, esta evaluación del vecino habrá además que añadirla al contador de evaluaciones de la función objetivo.

Algorithm 4 Paso de búsqueda local (w : solución)

1: $w' \leftarrow$ generaVecino(w, σ)
2: $w \leftarrow$ maximizador$_{v \in \{w, w'\}}$(evalua(v))
3: evals $\leftarrow$ evals $+ 1$

2. Pseudocódigo de los algoritmos

2.1. Enfriamiento simulado

Consideramos un esquema de enfriamiento de Cauchy. La temperatura inicial queda determinada por la siguiente fórmula.

$$T_0 = \frac{\mu C(S_0)}{-\log(\phi)}$$

En esta fórmula, $C(S_0)$ es el coste de la solución inicial que hemos generado aleatoriamente. Hemos tomado como valores constantes ambos parámetros, siendo tanto $\mu = 0.3$ como $\phi = 0.3$. Además de esta temperatura inicial, debemos fijar la temperatura final y lo haremos como $T_f = 0.001$; excepto en el caso en el que la temperatura inicial fuera menor que ella, donde fijamos por ejemplo $T_f = 0.01T_0$ para prevenir este caso.

El esquema de enfriamiento quedará entonces, con un parámetro β, determinado por la siguiente fórmula.

$$T_{k+1} = \frac{T_k}{1 + \beta T_k}$$

Donde el parámetro β dependerá de la temperatura inicial T_0, de la temperatura final T_f, y de un parámetro M que representa el número de enfriamientos que se realizarán y que se calcula explícitamente como $M = 15000/\text{maxVecinos}$, redondeado. A su vez, el número máximo de vecinos que se consideran en cualquier paso del enfriamiento simulado queda determinado por maxVecinos = 10nAtributos. El parámetro β viene explícitamente dado por la siguiente fórmula.

$$\beta = \frac{T_0 - T_f}{MT_0T_f}$$

Nótese que se produce el enfriamiento al superar el número de maxVecinos explorados o al superar el número máximo de saltos, maxExitos = 0.1maxVecinos. Como observamos tras la experimentación que se generan demasiados vecinos entre cada enfriamiento, hemos decidido probar con un número menor de éxitos antes de enfriar, dado como 0.01maxVecinos. El código en sí consta de varias funciones, cada una de ellas ejecutando un paso más interno. El más interno de todos es el paso de búsqueda, que funciona usando la búsqueda local para generar un vecino nuevo y aceptar dependiendo de las condiciones de temperatura. La condición para escoger en un paso determinado un vecino peor viene dada por la siguiente fórmula, donde T es la temperatura y la k es una constante que nuestro caso tomamos como 1; nos queda

$$\text{random} \leq \exp\left(\frac{-\Delta f}{kT}\right).$$

La implementación sigue el siguiente pseudocódigo, donde jump decide si se saltará a la siguiente solución.

Algorithm 5 Paso de búsqueda en enfriamiento simulado (w : solución)

1: $w' \leftarrow$ busquedaLocal(w, σ)
2: diff $= w.fitness - w'.fitness$
3: $jump \leftarrow$ (diff < 0 || random $<$ exp$(-$diff$/T)$)
4: update(evals $+ 1$)
5: update(localEvals $+ 1$)
6: update(if jump then exitos $+ 1$)

El nivel superior es un paso de enfriamiento, este realiza pasos de búsqueda pero además controla el enfriamiento tras un determinado número de evaluaciones o tras un número determinado de éxitos. El paso de enfriamiento aprovecha además para actualizar la mejor solución encontrada hasta el momento

Algorithm 6 Paso de enfriamiento (w : solución)

1: Ejecuta pasoDeBusqueda hastaQue (localevals $>$ maxVecinos || exitos $>$maxExitos)
2: $\beta \leftarrow (T_0 - T_f)/(mT_0T_f)$
3: $T_{new} = T/(1 + \beta T)$
4: if newBest $>$best then update(best)

Finalmente, el nivel más externo implementa el algoritmo haciendo que itere hasta superar el número máximo de evaluaciones permitidas.

Algorithm 7 Enfriamiento simulado

1: $w \leftarrow$ solucionAleatoria
2: mientras evals $<$ 15000 aplica pasoEnfriamiento
3: return best

2.2. Búsqueda local iterada

La implementación de la búsqueda local iterada aprovecha la implementación anterior de una búsqueda local y necesita poco código sobre él. El añadido más importante es un operador que provoque una mutación fuerte que nos permita seguir iterando la búsqueda local y la necesidad de guardar a cada paso la mejor solución encontrada hasta el momento para permitirnos volver a ella a la hora de devolverla incluso si accedemos a una solución peor durante la búsqueda.

Nuestro operador de mutación fuerte queda implementado como sigue. Toma un valor $t = 0.1n$ dando el número de componentes a mutar, luego elige aleatoriamente las componentes y mapea las mutaciones sobre ellas, cada una aleatoria y usando en este caso $\sigma = 0.4$.

Algorithm 8 Mutación fuerte (w : solución)

1: t $\leftarrow$ nAtribs/10
2: indices $\leftarrow$ randoms$(0, \text{nAtribs} - 1)$
3: mapea (mutaComponente ($\sigma = 0.4$)) en indices de w

La búsqueda local que implementamos en este caso es concretamente la siguiente. En el bucle incrementamos el número de evaluaciones (nótese que evaluamos la bondad del vecino conforme lo generamos en el paso de búsqueda) y elegimos el mejor entre la solución actual y él.

Algorithm 9 Búsqueda local (w : solución)

1: vecino $\leftarrow$ pasoBusqueda
2: evals $\leftarrow$ evals + 1)
3: $w \leftarrow$ máx(w, vecino)

El paso de comparación ocurre al final de cada búsqueda local. Compara esta solución con la mejor hasta el momento y aplica una mutación ferte a la mejor de ellas.

Algorithm 10 Paso comparación (w : solución)

1: best $\leftarrow$ máx(best, w)
2: w $\leftarrow$ mutacionFuerte(w)
3: $w \leftarrow$ máx(w, vecino)

Finalmente, la implementación del algoritmo final es la encargada de decidir que se aplicarán 1000 pasos de búsqueda local antes de aplicar una comparación y una mutación fuerte. Usamos la variable nbusq para actualizarla cada 1000 pasos de búsqueda local y empezar una nueva búsqueda.

Algorithm 11 Búsqueda local iterada (w : solución)

1: $w \leftarrow$ solucionAleatoria
2: mientras (nbusq < 15) aplica
3: si (evals < 1000) pasoBusqueda,
4: si no, pasoComparacion, nbusq $\leftarrow +1$, evals $= 0$

2.3. Evolución diferencial

El algoritmo de evolución diferencial se basará principalmente en una función de cruce que será la que varíe entre las dos versiones del algoritmo, dándonos en caso DE/Rand/1 y el caso DE/current-to-best/1. Empezamos describiendo la primera de ellas, que escoge tres índices para los padres de una nueva solución de forma aleatoria y, también de forma aleatoria, elige en cada componente si quedarse con al del hijo o con la nueva componente del padre. Tomamos como constantes para el algoritmo los dos valores $CR = F = 0.5$, que son los sugeridos en el guion.

Algorithm 12 Cruce DE/Rand/1 (w : Solución, pd1,pd2,pd3 : Padre)

1: mapea cruce w pd1 pd2 pd3
2: cruce(i,p1,p2,p3) = si random <CR entonces $p1 + F(p2 - p3)$ siNo i
3: trunca

En el caso de DE/Current-to-best/1, el cambio estará en la función de cruce, que ahora pasará a involucrar también a la mejor solución hasta el momento. La elección entre una y otra componente será la componente particular que distinga entre ambas versiones.

Algorithm 13 Cruce DE/Current/1 (w : Solución, pd,pd2 : Padre, bs : Individuo)

1: mapea cruce w pd1 pd2 bs
2: cruce(i,p1,p2,b) = si random <CR entonces $i + F(b - i) + F(p1 - p2)$ siNo i
3: trunca

Nótese como en ambos casos se ha usado la recombinación binomial para decidir si quedarse con el resultado del cruce o con el peso anterior. En el siguiente código implementaremos el paso de evolución diferencial, que debe generar padres aleatoriamente, cruzarlos siguiendo los cruces anteriores, y terminar haciendo un reemplazamiento uno-a-uno, en el que cada individuo pueda ser sustituido por su descendiente si este fuera mejor.

Algorithm 14 Paso diferencial (popl : Población, best : Individuo)

1: padres ← reordena popl aleatoriamente
2: descendencia ← mapea cruce sobre (individuos, padres)
3: nuevapopl ← mapea max (individuos, descendencia)
4: evals ← +1

Finalmente, el algoritmo aplica este paso evolutivo hasta que agota el número de evaluaciones posibles de la función objetivo. Nótese que podemos usar la generación de una población aleatoria que ya propusimos en l práctica de algoritmos genéticos.

Algorithm 15 Evolución diferencial (popl : Población, best : Individuo)

1: popl ← poblacionAleatoria
2: mientras evals <15000
3: aplica pasoDiferencial

3. Procedimiento considerado, manual de usuario

Al igual que en la segunda práctica se usa **Haskell** [P+03] y paralelismo con [Les08], especialmente para la función objetivo. El proceso de validación y generación de los resultados se hace reproducible con [Smi92] y se encuentra en el archivo `makefile`, en el que se declaran las semillas de aleatoriedad (`$SEEDn`) que son las que se envían a los

distintos algoritmos. Volvemos a elegir semillas `0,1,2,3,4`, pero pueden ser cambiadas para su ejecución en el propio archivo `makefile`.

Además de los ejecutables de validación `bin/fivefold` y `bin/scorer`, así como los algoritmos de la primera y segunda prácticas, presentamos los ejecutables de los nuevos algoritmos:

- `bin/SimulatedAnnealing`, implementación del enfriamiento simulado con los parámetros fijados en el guion;
- `bin/ILS`, implementación de la búsqueda local reiterada con los parámetros y la mutación fijados en el guion;
- `bin/DERand`, implementación de la primera variante de la evolución diferencial, que utiliza la fórmula de cruce DE/Rand/1;
- `bin/DECurrent`, implementación de la segunda variante de la evolución diferencial, que utiliza la fórmula de cruce DE/current-to-best/1.

Todas las implementaciones reciben como argumento de línea de comandos una semilla aleatoria y leen por la entrada estándar un conjunto de entrenamiento; acabarán devolviendo una solución por salida estándar.

Para el resto de detalles de ejecución nos referimos a la primera práctica.

4. Experimentos y análisis de los resultados

4.1. Enfriamiento simulado

El enfriamiento simulado es un algoritmo basado en trayectorias simples con una mejora crucial respecto a la búsqueda local que implementamos en prácticas anteriores al incorporar la posibilidad de empeorar la solución local bajo ciertas condiciones probabilísticas determinadas por el esquema de enfriamiento. Esperamos que esta técnica permita aumentar la diversidad de las soluciones que explora, aunque en última instancia, cuando la temperatura sea baja, acabe convergiendo a una de ellas de manera similar a como lo haría la búsqueda local.

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.641	0.847	0.744	48.145	0.950	0.864	0.907	16.544	0.833	0.773	0.803	52.122
Partición 2	0.688	0.847	0.767	46.980	0.700	0.909	0.805	16.653	0.778	0.750	0.764	52.707
Partición 3	0.703	0.861	0.782	46.789	0.900	0.909	0.905	16.093	0.741	0.818	0.779	49.616
Partición 4	0.734	0.861	0.798	46.964	0.675	0.909	0.792	17.462	0.815	0.841	0.828	48.050
Partición 5	0.719	0.861	0.790	47.091	0.857	0.909	0.883	16.247	0.667	0.727	0.697	53.195
Media	0.697	0.855	0.776	47.194	0.816	0.900	0.858	16.600	0.767	0.782	0.774	51.138

Cuadro 1: Enfriamiento Simulado en el problema del APC

Es importante hacer notar que durante los primeros experimentos en el conjunto de datos más pequeño, `parkinsons`, obtuvimos resultados consistentemente peores al

aplicar el valor de maxVecinos sugerido inicialmente. Para evitarlo, tomamos la recomendación de cambiarlo por un valor menor que acelerara el enfriamiento, obteniendo mejores resultados tras el cambio.

Notamos que el algoritmo funciona mejor precisamente en los conjuntos de menos dimensionalidad, donde puede haber sido más fácil explorar con operadores de mutación que cambian un número proporcional de componentes del vector solución; en cualquier caso y como podrá observarse en la comparación final, esta es una tendencia de todos los algoritmos que hemos observado hasta el momento que simplemente se acentúa en este caso.

Es llamativo además que, salvo en el primer conjunto de datos, es el primer algoritmo que consistentemente incrementa normalmente la tasa de clasificación y la tasa de reducción a la par en los distintos ejemplos. Entendemos que el ser una trayectoria simple bajo el esquema de enfriamiento, que en las últimas fases será similar a una búsqueda local sobre un entorno reducido, hace que acabe acercándose normalmente a un óptimo local donde ambas cantidades deben haber sido optimizadas igualmente al haber tomado $\alpha = 0.5$.

4.2. Búsqueda local iterativa

La búsqueda local iterativa es un algoritmo basado en trayectorias múltiples. En este caso no es preocupante, como era en los anteriores, la posibilidad de que una búsqueda local quede atascada al principio en un óptimo local; pero sigue siendo posible que las mutaciones que aplicamos no sean lo suficientemente fuertes como para evitar alejar lo suficiente a nuestra solución. Además, las búsquedas locales se pueden ver interrumpidas cada cierto número de pasos para aplicar una mutación fuerte que las aleje del óptimo local que podrían estar persiguiendo sin haber convergido aún. Un estudio de diversidad y convergencia similar al que realizamos en la práctica anterior con los algoritmos genéticos podría ser útil para determinar si este es el caso.

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.656	0.861	0.759	47.940	0.750	0.909	0.830	16.082	0.833	0.864	0.848	50.119
Partición 2	0.797	0.861	0.829	47.144	0.800	0.909	0.855	15.695	0.759	0.909	0.834	49.586
Partición 3	0.797	0.875	0.836	46.165	0.675	0.909	0.792	15.444	0.741	0.864	0.802	49.371
Partición 4	0.734	0.875	0.805	49.511	0.675	0.909	0.792	15.733	0.685	0.864	0.774	48.586
Partición 5	0.719	0.847	0.783	47.364	0.771	0.909	0.840	15.839	0.725	0.864	0.795	50.410
Media	0.741	0.864	0.802	47.625	0.734	0.909	0.822	15.759	0.749	0.873	0.811	49.614

Cuadro 2: Búsqueda local iterativa en el problema del APC

Notamos finalmente que debe existir alguna zona del espacio de búsqueda en el conjunto de datos `parkinsons` cercana a varios óptimos locales donde el algoritmo se queda alrededor del valor `0.909` en la tasa de reducción. Descartamos que sea un error concreto del algoritmo porque es un comportamiento que se ha repetido ya en otros algoritmos con código independiente al suyo.

4.3. Evolución diferencial

Los dos últimos algoritmos son ejemplos de búsqueda evolutiva y más similares a los algoritmos genéticos anteriores. Volvemos en este caso a tener una población en la que explotaremos la bondad de algunas soluciones, escogidas aleatoriamente o ordenadamente, para generar nuevas soluciones que aprovechen componentes suyas.

Al aplicar la primera vertiente del algoritmo, DE/Rand/1, obtenemos unos resultados sorprendentemente buenos y además mucho menos variables entre las distintas ejecuciones y conjuntos de datos que los obtenidos en los algoritmos anteriores. La búsqueda evolutiva nos está permitiendo usar la diversidad que introducían los algoritmos genéticos de forma más controlada que ellos al usar tres elementos de la población para añadir un componente de explotación a la búsqueda. Estos algoritmos se han alejado de la bioinspiración que tenían los originales, pero nos permiten introducir muchas variantes (cambiando el número de padres, probando nuevos operadores de cruce o modelos de recombinación) que además nos pueden servir para adaptarlos al problema.

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.719	0.917	0.818	52.742	0.775	0.909	0.842	16.829	0.907	0.932	0.920	48.356
Partición 2	0.797	0.931	0.864	52.227	0.725	0.909	0.817	15.002	0.667	0.909	0.788	47.529
Partición 3	0.734	0.903	0.819	53.598	0.725	0.909	0.817	15.076	0.704	0.932	0.818	47.835
Partición 4	0.734	0.931	0.832	53.605	0.675	0.909	0.792	15.788	0.796	0.932	0.864	47.827
Partición 5	0.688	0.931	0.809	53.331	0.829	0.909	0.869	15.987	0.725	0.909	0.817	48.056
Media	0.734	0.923	0.828	53.101	0.746	0.909	0.827	15.736	0.760	0.923	0.841	47.921

Cuadro 3: Evolución diferencial DE/Rand/1 en el problema del APC

La segunda vertiente del algoritmo utiliza explícitamente a cada paso el mejor individuo de la población. Debemos tener en cuenta que DE/Current-to-best/1 está usando más información de la que tenemos disponible en la propia población y haciendo tender a las soluciones hacia la mejor de ellas. Esto puede estar haciendo que la convergencia de la población en su conjunto sea más rápida y que la población sea mucho menos diversa. Quizá esto explique los resultados que estamos obteniendo en este segundo caso, que aunque son ligeramente peores que en el caso anterior, consiguen en casos concretos mejorar nuestras mejores soluciones hasta el momento.

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
Partición 1	0.672	0.667	0.669	58.493	0.950	0.864	0.907	15.162	0.852	0.750	0.801	51.180
Partición 2	0.828	0.708	0.768	57.101	0.825	0.864	0.844	15.325	0.722	0.750	0.736	48.993
Partición 3	0.734	0.708	0.721	56.072	0.900	0.818	0.859	14.919	0.648	0.705	0.676	49.411
Partición 4	0.703	0.764	0.734	56.960	0.750	0.818	0.784	15.113	0.796	0.727	0.762	50.090
Partición 5	0.750	0.708	0.729	56.434	0.800	0.773	0.786	16.683	0.843	0.773	0.808	47.524
Media	0.737	0.711	0.724	57.012	0.845	0.827	0.836	15.440	0.772	0.741	0.757	49.440

Cuadro 4: Evolución diferencial DE/Rand/1 en el problema del APC

En los dos casos, la información específica del problema que estamos usando es la estructura de espacio vectorial que poseen las soluciones para aplicar los operadores de

cruce y la asunción implícita de continuidad para poder trabajar sabiendo que podemos tomar componentes de las soluciones mejores y que probablemente mejoren nuestra solución. Asumimos en la forma que construimos las soluciones que las distintas componentes son independientes para poder mezclar componentes de varios individuos en uno solo durante el cruce.

Una vía interesante sería la de probar distintos valores para las constantes CR y F del algoritmo, que han funcionado muy bien hasta el momento pero cuyo valor podría ser crucial para la ejecución.

4.4. Resultados globales y comparación

Comparamos ahora los resultados con todos los resultados obtenidos por los algoritmos de referencia en las prácticas anteriores. Nos parece aquí interesante considerar primero la comparación entre la búsqueda local y el enfriamiento simulado, siendo ambos algoritmos de trayectorias simples; nótese que obtienen resultados aproximadamente iguales en los conjuntos de datos más grandes, a pesar de que el enfriamiento simulado suele aprovechar mejor la tasa de reducción, y hay una mejora del enfriamiento simulado en el conjunto `parkinsons`. Podríamos atribuir este comportamiento al hecho de que hay en el segundo un componente que nos permite explorar en ocasiones posibles mejoras de la tasa de reducción, a pesar de que en ocasiones lo desvíe del óptimo local.

Ambos obtienen peores resultados que la búsqueda local iterada, donde podemos comprobar que en el caso de nuestros problemas, usar varias trayectorias es más interesante que una sola. Si atendemos al estudio de la convergencia que realizamos en la práctica anterior para algoritmos genéticos, pudimos observar que la convergencia prematura era un problema común, y los algoritmos de trayectorias múltiples pueden estar ayudando a solventarlo. La búsqueda local iterada es, como era esperable, consistentemente mejor que la búsqeuda local, lo que atribuimos a este comportamiento de trayectorias múltiples.

Los mejores algoritmos en cualquier caso son los basados en búsqueda evolutiva, que usan de nuevo poblaciones de soluciones y que parecen no sufrir este problema de convergencia prematura que tenían los demás. Son también algoritmos en los que hay implícita mucha información del problema, tanto en la estructura de espacio vectorial, como en el cruce que modifica vectores enteros y en la independencia entre componentes del vector solución.

También resaltamos que el único algoritmo de referencia que está a la altura de una comparación de este tipo es la búsqueda local. Los algoritmos greedy iniciales están muy lejos de resultados obtenidos al aplicar ideas más complejas.

Finalmente presentamos la tabla completa de resultados de todos los algoritmos implementados. Sobre ella podemos notar por ejemplo un fenómeno esperable del hecho de que el criterio de parada sea el número de evaluaciones de la función objetivo y que sea constante: los tiempos son aproximadamente iguales entre algoritmos, rondando el minuto en el conjunto de datos más grande. Las pequeñas variaciones pueden ser atribuidas al tiempo que tardan los algoritmos en organizar la población o aplicar cruces y mutaciones, además de detalles de la carga del procesador al paralelizar.

Cuadro 5: Resultados globales en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
1NN	0.816	0.000	0.408	0.032	0.783	0.000	0.391	0.008	0.774	0.000	0.387	0.016
RELIEF	0.819	0.014	0.416	0.118	0.794	0.000	0.397	0.024	0.767	0.000	0.383	0.055
BL	0.628	0.969	0.799	15.354	0.391	0.973	0.682	1.192	0.622	0.954	0.788	9.458
ES	0.697	0.855	0.776	47.194	0.816	0.900	0.858	16.600	0.767	0.782	0.774	51.138
ILS	0.741	0.864	0.802	47.625	0.734	0.909	0.822	15.759	0.749	0.873	0.811	49.614
DERand	0.734	0.923	0.828	53.101	0.746	0.909	0.827	15.736	0.760	0.923	0.841	47.921
DECurrent	0.737	0.711	0.724	57.012	0.845	0.827	0.836	15.440	0.772	0.741	0.757	49.440

Cuadro 6: Resultados globales en el problema del APC

	Ozone				Parkinsons				Spectf			
	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)	clas	red	Agr.	T(s)
1NN	0.816	0.000	0.408	0.032	0.783	0.000	0.391	0.008	0.774	0.000	0.387	0.016
RELIEF	0.819	0.014	0.416	0.118	0.794	0.000	0.397	0.024	0.767	0.000	0.383	0.055
BL	0.628	0.969	0.799	15.354	0.391	0.973	0.682	1.192	0.622	0.954	0.788	9.458
BL2	0.738	0.800	0.769	13.177	0.655	0.909	0.782	0.898	0.768	0.855	0.811	7.488
AGE-CA	0.747	0.703	0.725	57.608	0.744	0.764	0.754	16.566	0.802	0.668	0.735	54.469
AGE-BLX	0.744	0.831	0.787	53.886	0.786	0.882	0.834	17.787	0.768	0.791	0.779	36.351
AGG-CA	0.738	0.645	0.691	32.838	0.785	0.382	0.583	9.434	0.761	0.627	0.694	30.513
AGG-BLX	0.763	0.642	0.702	59.889	0.745	0.536	0.641	16.059	0.794	0.605	0.699	53.961
AM-(10,1.0)	0.769	0.817	0.793	57.391	0.809	0.882	0.845	16.333	0.734	0.791	0.763	57.729
AM-(10,0.1)	0.775	0.767	0.771	60.870	0.779	0.864	0.821	19.237	0.738	0.805	0.771	53.216
AM-(10,0.1mej)	0.713	0.786	0.749	59.689	0.798	0.882	0.840	18.029	0.753	0.759	0.756	54.846
AM-div	0.775	0.811	0.793	63.017	0.729	0.891	0.810	19.188	0.760	0.827	0.794	59.235
ES	0.697	0.855	0.776	47.194	0.816	0.900	0.858	16.600	0.767	0.782	0.774	51.138
ILS	0.741	0.864	0.802	47.625	0.734	0.909	0.822	15.759	0.749	0.873	0.811	49.614
DERand	0.734	0.923	0.828	53.101	0.746	0.909	0.827	15.736	0.760	0.923	0.841	47.921
DECurrent	0.737	0.711	0.724	57.012	0.845	0.827	0.836	15.440	0.772	0.741	0.757	49.440

El algoritmo que ha dado los mejores resultados una vez implementado es la evolución diferencial DE/Rand/1. Es importante hacer notar que ha sido además uno de los menos complejos en su implementación y conceptualmente una vez proporcionado el pseudocódigo de los guiones de prácticas. Su elemento más complejo y probablemente más significativo a la hora de obtener buenos resultados es el operador de cruce, con el que podrían probarse más variaciones además de las dos estudiadas. Es comparable con los genéticos, que han obtenido también buenos resultados, pero que no usaban los operadores de cruce que proporciona la evolución diferencial y que aprovechan mejor la estructura del espacio.

Referencias

[Les08] Roman Leshchinsky. Data.vector. *Hackage*, 2008. https://hackage.haskell.org/package/vector-0.12.0.1/docs/Data-Vector.html.

[P+03] Simon Peyton-Jones et al. The Haskell 98 language and libraries: The revised report. *Journal of Functional Programming*, 13(1):0–255, Jan 2003. http://www.haskell.org/definition/.

[Smi92] Paul Smith. Gnu make. *GNU Operating system*, 1992. https://www.gnu.org/software/make/.

[Yor16] Brent Yorgey. Control.monad.random.class. *Hackage*, 2016. https://hackage.haskell.org/package/MonadRandom-0.5.1/docs/Control-Monad-Random-Class.html.