Resolución del problema de asignación de fechas de exámenes con algoritmos genéticos

Alejandro Bellogín Kouki
alejandro.bellogin@uam.es

6 de febrero de 2008

1. Introducción

El problema de encontrar un calendario óptimo de exámenes para unas fechas determinadas se sabe que es NP-completo debido a su gran cantidad de restricciones[2]. Esto es así ya que, en el caso general, hay que controlar que un profesor no tenga dos exámenes a la vez, ni un alumno, que haya aulas disponibles y que la configuración de los exámenes no sea muy desfavorable para los alumnos (no se quieren exámenes próximos que sean del mismo año, obligatorios o duros).

En este trabajo nos centraremos en esta última condición ya que no se dispone del resto de datos. Para resolverlo utilizaremos distintos tipos de algoritmos genéticos, todos muy cercanos a la formulación original de Goldberg, ya que por la naturaleza de este problema, se presta a resolverlo de esta manera (muchas restricciones y combinaciones posibles). Para terminar mostraremos unos resultados preliminares que se han obtenido con los algoritmos diseñados, aunque la conclusión que se extrae de ellos es que habrá que resolver ciertas colisiones manualmente. Esto es debido a que no se ha dispuesto de mucho tiempo para obtener los mejores parámetros de cada algoritmo, ni para probar con distintas funciones de fitness.

2. Trabajo relacionado

Hemos utilizado la formulación original de algoritmos genéticos vista en la asignatura junto con alguna modificación de la misma, además, hemos implementado un algoritmo memético, que junto con la función de fitness, están basados en lo que se puede ver en [1], aunque función de fitness la hemos adaptado para que no emplee datos sobre el número de alumnos, ya que no disponemos de dichos datos. No obstante, sí utiliza las colisiones entre asignaturas debidas a que pertenecen a cursos consecutivos o al mismo.

3. Propuesta

En este trabajo se han desarrollado tres algoritmos genéticos, con distintas ventajas e inconvenientes cada uno, para intentar resolver el problema de la asignación de exámenes. A continuación hablaremos de ellos y después pasaremos a mencionar algunas resultados que se han obtenido.

Antes de comenzar a explicar los algoritmos diseñados vamos a enseñar qué datos de partida necesitan y cómo se obtienen. Como se verá a continuación, los algoritmos necesitan conocer unos identificadores para cada asignatura y un número que identifique cada fecha disponible; para obtener estos datos se realiza lo siguiente:

- Tenemos almacenada en una base de datos todas las carreras de las que se quiere realizar el calendario de exámenes, de esa manera dependiendo de cuál de ellas en particular se quiere saber su calendario, se generará una lista que incluya los identificadores de las asignaturas que existen en dicha carrera (y para el semestre seleccionado) y que será la que reciba el algoritmo en cuestión.
- Para generar la lista de fechas disponibles de exámenes se toman como parámetros la fecha inicial, la final y una lista de festivos adicionales (fiestas en días laborables, selectividad); a partir de ellos se genera una lista donde se incluyen por cada día de la semana laborable incluído en el intervalo anterior dos índices: uno indicando que el examen se realizará en horario de mañana y otro en horario de tarde. También se consideran los sábados, pero sólo se permite un examen ese día.
- Los algoritmos implementados están preparados para recibir una lista de restricciones extra, de manera que deben intentar desechar a los individuos que no las cumplan. Esta lista de restricciones viene motivada por la titulación doble de Informática y Matemáticas, y se podría utilizar si los exámenes de Matemáticas ya han sido asignados.

Además, se necesita una tabla de colisiones creada a partir de las asignaturas involucradas

Ahora pasaremos a explicar cada uno de los algoritmos implementados, centrándonos en el primero (simple) y, del resto, explicar lo que añaden y las ventajas e inconvenientes encontradas.

3.1. Algoritmo simple

El primer cambio necesario con respecto a la implementación anterior ha sido la realización de un alfabeto no binario, ya que en este caso la codificación iba a utilizar enteros en lugar de 0 o 1 (booleanos) por simplicidad. Usando esta implementación, los alelos deben encargarse de que la negación del mismo dé un elemento distinto al que se tenía anteriormente, eligiendo otro al azar:

```
public IntegerAllele negate(){
    IntegerAllele a = new IntegerAllele();
    while (a.getAllele() == getAllele()){
        a.setAllele(alphabet.nextValue());
    }
    return a;
}
```

Aparte de esta modificación, el algoritmo simple se ha mantenido intacto salvo la inclusión de un método de *sembrado* de poblaciones inicialmente mejores que las aleatorias, asignando a cada asignatura una fecha consecutiva con respecto a la anterior módulo el número de asignaturas. Esta decisión mejoró la calidad de las soluciones en las primeras generaciones, acelerando el proceso de convergencia.

```
@Override
protected void seed() {
    oldPopulation.clear();
    for (int i = 0; i < sizePop; i++) {
        Individual ind = new Individual();
        Chromosome ch = new IntegerChromosome(sizeChromosome);
        for (int j = 0; j < sizeChromosome; j++) {
            Allele a = new IntegerAllele(sizeAllele - 1);
            if (constraints.get(j) != -1) {
                a.setAllele(constraints.get(j));
            } else {
                a.setAllele(j %(sizeAllele - 1));
```

```
                }
                ch.addAllele(j, a);
            }
            ind.setChromosome(ch);
            ind.setFenotype(decode(ind.getChromosome()));
            ind.setFitness(objFunc(ind.getFenotype()));
            ind.setParent1(-1);
            ind.setParent2(-1);
            ind.setXsite(-1);
            oldPopulation.addIndividual(i, ind);
        }
        newPopulation = (Population)oldPopulation.clone();
    }
```

Como se puede ver en el código, se realiza la asignación de fechas consecutivas siempre y cuando no exista una restricción para esa asignatura, en cuyo caso se inicializa la población con el valor de la restricción.

Se probó una modificación del algoritmo más restrictiva: tanto la siembra como las siguientes generaciones no podían contener individuos malos (es decir, que hubiera colisión entre alguna asignatura); esto provocaba que el algoritmo se comportara como una búsqueda ciega al reducir demasiado el espacio de búsqueda (de hecho, el algoritmo obtendría en la primera generación soluciones, mientras que en el resto de generaciones sólo se dedicaría a conseguir que esas soluciones mejoraran).

La creación de la matriz de colisiones también merece ser analizada. Esta matriz necesita para su generación la lista de asignaturas y una matriz que se ha creado previamente con las equivalencias entre las asignaturas (esta matriz no tiene sentido si sólo se piensa calcular el calendario de una titulación, pero cuando se involucra a varias es necesaria, ya que los identificadores de las asignaturas son distintos, pero el examen es el mismo). El siguiente paso para construir la matriz de colisiones es obtener la lista de colisiones para cada asignatura[1], al hacer esto se podría distinguir entre las asignaturas del mismo año y las consecutivas pero como era un requisito que se cumplieran las dos condiciones, se les da la misma puntuación de conflictividad.

```
        int[][] conflicts = new int[subjects.size()][subjects.size()];
        // init
        for (int i = 0; i < subjects.size(); i++) {
            for (int j = 0; j < subjects.size(); j++) {
                conflicts[i][j] = IntegerTimetableProblem.VALUE_NO_CONFLICT;
            }
        }

        // For each subject ...
        for (Subject s : subjects) {
            int id = s.getId();
            // ... we get its conflictive subjects ...
            ArrayList<Integer> conflictiveSubjects = getConflicts(id, s.
                getYear(), s.getDegreeID(), s.getSemester());
            for (Integer integer : conflictiveSubjects) {
```

[1]La variable `mapSubjects` contiene una tabla relacionando los identificadores de la base de datos de las asignaturas que se quieren utilizar y su índice en el array interno de asignaturas (`subjects`). Aquí se utiliza para comprobar si nos interesan las asignaturas que son conflictivas con una determinada asignatura.

```
                // ... and check if it is one of the subjects previously
                    retrieved
                if (mapSubjects.containsKey(integer)) {
                    conflicts[mapSubjects.get(integer)][mapSubjects.get(id)]
                        = IntegerTimetableProblem.VALUE_HIGH_CONFLICT;
                    conflicts[mapSubjects.get(id)][mapSubjects.get(integer)]
                        = IntegerTimetableProblem.VALUE_HIGH_CONFLICT;
                }
            }
            // If a subject is equal to this one, there is not a conflict
            for (int i = 0; i < equals[mapSubjects.get(id)].length; i++) {
                boolean eq = equals[mapSubjects.get(id)][i];
                if (eq){
                    conflicts[mapSubjects.get(id)][i] =
                        IntegerTimetableProblem.VALUE_NO_CONFLICT;
                    conflicts[i][mapSubjects.get(id)] =
                        IntegerTimetableProblem.VALUE_NO_CONFLICT;
                }
            }
        }
```

Para terminar, explicaremos la función de fitness utilizada. Su objetivo es puntuar alto a los individuos peores, siguiendo la siguiente fórmula:

$$\sum_{i=1}^{E-1}\sum_{j=i+1}^{E}\left(\sum_{p=1}^{P} t_{ip}t_{j(p+1)}c_{ij}d_{p(p+1)} + t_{ip}t_{j(p-1)c_{ij}d_{p(p-1)}}\right) + At_{i(P+1)}$$

$t_{ip} = 1$	Si la asignatura i está asignada al periodo p, 0 en otro caso
c_{ij}	Es el valor previamente calculado de la colisión entre las asignaturas i y j
d_{pq}	En función de la distancia entre dos periodos se le asigna un valor a esta variable

Si comparamos esta fórmula con el código:

```
protected double decode(Chromosome chromosome) {
    int value = 0;

    for (int i = 0; i < events.size() - 1; i++) {
        int periodI = ((IntegerChromosome) chromosome).getAllele(i).
            getAllele();
        for (int j = i + 1; j < events.size(); j++) {
            int factor = conflicts[i][j];
            int periodJ = ((IntegerChromosome) chromosome).getAllele(j).
                getAllele();
            if ((periodI < periods.size()) && (periodJ < periods.size()))
                {
                        // period P+1 = unscheduled (there may be two events
                            unscheduled)
                int difference = Math.abs(periods.get(periodI) - periods.
                    get(periodJ));
                if (difference == 0) {
```

```
                    value += factor * Utils.VALUE_SAME_HOUR;
                // this is suppose to happen only with the same individual
                } else if (difference < Utils.DISTANCE_NEXT_DAY) {
                    // same day ==> 3
                    value += factor * Utils.VALUE_SAME_DAY;
                } else if (difference < (Utils.DISTANCE_SAME_DAY + 2 *
                    Utils.DISTANCE_NEXT_DAY)) {
                    // next day ==> 1
                    value += factor * Utils.VALUE_NEXT_DAY;
                } else {
                    // otherwise ==> 0
                    value += factor * 0;
                }
            }
        }
        for (int j = 0; j < equals[i].length; j++) {
            boolean bs = equals[i][j];
            int periodJ = ((IntegerChromosome) chromosome).getAllele(j).
                getAllele();
            if (bs && periodI != periodJ) {
                value += IntegerTimetableProblem.VALUE_NOT_EQUAL;
            }
        }

        int constraintI = constraints.get(i);
        if ((constraintI != -1) && (constraintI != periodI)) {
            value += IntegerTimetableProblem.
                VALUE_NOT_SATISFYING_CONSTRAINT;
        }
        // Heavy penalty if it is not scheduled
        if (periodI == periods.size()) {
            value += IntegerTimetableProblem.VALUE_NOT_SCHEDULED;
        }
    }
    int i = events.size() - 1;
    if ((constraints.get(i) != -1) && (constraints.get(i) != ((
        IntegerChromosome) chromosome).getAllele(i).getAllele()))) {
        value += IntegerTimetableProblem.VALUE_NOT_SATISFYING_CONSTRAINT;
    }
    // Heavy penalty if it is not scheduled
    if (((IntegerChromosome) chromosome).getAllele(i).getAllele() ==
        periods.size()) {
        value += IntegerTimetableProblem.VALUE_NOT_SCHEDULED;
    }
    // the biggest value, the worst solution
    return value;
}
```

podemos ver que para nosotros A vale `IntegerTimetableProblem.VALUE_NOT_SCHEDULED`, y d_{pq} es uno de los siguientes valores: `Utils.VALUE_SAME_HOUR, Utils.VALUE_SAME_DAY, Utils.VALUE_NEXT_DAY,`

0. Además, como una cuestión técnica, el periodo $P + 1$ está codificado como el último de la lista de periodos disponibles (es decir, la lista de periodos contiene uno más de lo que debería), y por lo tanto, para saber si una asignatura está asignada al periodo $P + 1$ (es decir, que no ha encontrado un periodo válido aún) hay que realizar lo siguiente:

```
((IntegerChromosome) chromosome).getAllele(i).getAllele() == periods.size()
```

También se puede ver que la función de fitness penaliza las soluciones que no cumplen con las restricciones iniciales. Por último, los valores que se van a maximizar son los transformados por la función $f(x) = \frac{1}{x+1}$ a partir de los valores del fitness de cada individuo.

3.2. Algoritmo simple mejorado

Este algoritmo es un intento de mejorar el simple utilizando ideas que se emplearon para realizar el memético.

Básicamente el cambio realizado con respecto al algoritmo simple es que en cada generación sólo se intentan asignar un número determinado de asignaturas. El resultado es que es mucho más rápido pero pierde las ventajas de los algoritmos genéticos, ya que sólo se ejecuta un número pequeño de generaciones (exactamente: número total de asignaturas / asignaturas cada vez). Además, no se intentan mejorar las asignaciones ya realizadas.

Estos inconvenientes se solventan en la implementación memética.

3.3. Algoritmo memético

Basándonos en el algoritmo descrito en [1] hemos implementado un algoritmo memético, esto quiere decir que tenemos un algoritmo genético y a la vez utiliza descenso de gradiente. El mayor cambio ha sido la fase inicial, en donde se eligen las asignaturas a asignar en ese momento y se aplica el algoritmo genético simple explicado para obtener una población óptima para esas asignaturas, en las iteraciones siguientes se intentan mejorar las asignaturas ya asignadas, por lo que la ganancia en resultados es muy grande; como es lógico, los tiempos de ejecución aumentan mucho, incluso para pocas generaciones, ya que se multiplican las generaciones por el número de veces que se ejecuta el algoritmo genético, lo cual viene dado por el número máximo de asignaturas a asignar en cada iteración.

El código de esta fase inicial se muestra a continuación (`run` es el método para comenzar a utilizar el algoritmo memético):

```
int e = 0;
do {
    e += this.examsToSchedule;
    int n = 0;
    if (e > events.size()) {
        n = examsToSchedule - (events.size() - e);
        e = events.size();
    } else {
        n = examsToSchedule;
    }
    ArrayList<Integer> examsUnscheduled = getMaxNextEvents();
    SortedMap<Integer, Integer> map = new TreeMap<Integer, Integer>();
    for (Integer integer : examsUnscheduled) {
        int d = degree(integer);
        map.put(integer, d);
    }
```

```
        ArrayList<Integer> values = new ArrayList<Integer>(map.values());
        Collections.sort(values);
        ArrayList<Integer> valuesToPick = new ArrayList<Integer>();
        for (int i = values.size() - 1,  j = 0; i >= 0 && j < n; j++, i--) {
            // pick up the n subjects with largest degree
            valuesToPick.add(values.get(i));
        }

        // memetic alg over those n events, keeping intact the rest
        ArrayList<Integer> eventsToPick = new ArrayList<Integer>();
        for (int j = 0; j < valuesToPick.size(); j++) {
            for (int i = 0; i < events.size(); i++) {
                if (map.get(i) == valuesToPick.get(j)) {
                    if (!eventsToPick.contains(i) && isMutable(i)) {
                        eventsToPick.add(i);
                        mutableEvents.set(i, false);
                        break;
                    }
                }
            }
        }

        run(eventsToPick);
        Individual ind = newPopulation.getIndividual(bestIndividual);
        for (Integer integer : eventsToPick) {
            int eventFixed = (Integer) ind.getChromosome().getAllele(integer).
                getAllele();
            constraints.set(integer, eventFixed);
        }
    } while (e != events.size());
```

Para conseguir mejorar los elementos ya asignados, se ejecuta el método siguiente cada vez que se genera un nuevo individuo.

```
private void hillClimbing(IntegerChromosome ch) {
    IntegerChromosome p = (IntegerChromosome) ch.clone();
    //Collections.shuffle(p);
    boolean exit = true;
    do {
        exit = true;
        for (int i = 0; i < p.size(); i++) {
            int period = p.getAllele(i).getAllele();
            ArrayList<Integer> ev = getEvents(period, ch);
            for (int k = 0; k < ev.size(); k++) {
                int e = ev.get(k);
                if (!isMutable(e) && (!nextEvents.contains(e))) {
                    break;
                }
                Double d = Double.POSITIVE_INFINITY;
                int q = -1;
```

```
                for (int j = 0; j < p.size(); j++) {
                    if (i != j) {
                        double delta = hillClimbingOperator(e, i, j, ch);
                        if (delta < d) {
                            d = delta;
                            q = j;
                        }
                    }
                }
                if ((d != Double.POSITIVE_INFINITY) && (d > 0)) {
                    // Change event e to period q
                    ch.getAllele(e).setAllele(q);
                    exit = false;
                } else {
                    exit = true;
                }
            }
        }
    } while (!exit);
}
```

Este método utiliza el operador codificado como sigue:

```
private double hillClimbingOperator(int indexEvent, int indexPeriodP, int
    indexPeriodQ, IntegerChromosome ch) {
    double delta = 0.0;
    for (int i = 0; i < events.size(); i++) {
        int isSchP = isScheduled(indexEvent, indexPeriodP, ch);
        int isSchQ = isScheduled(indexEvent, indexPeriodQ, ch);
        int isSchNextP = isScheduled(i, indexPeriodP + 1, ch);
        int isSchPrevP = isScheduled(i, indexPeriodP - 1, ch);
        int isSchNextQ = isScheduled(i, indexPeriodQ + 1, ch);
        int isSchPrevQ = isScheduled(i, indexPeriodQ - 1, ch);
        int cij = conflicts[indexEvent][i];
        delta += (isSchP * isSchNextP * cij * distance(indexPeriodP,
            indexPeriodP + 1));
        delta += (isSchP * isSchPrevP * cij * distance(indexPeriodP,
            indexPeriodP - 1));
        delta -= (isSchQ * isSchNextQ * cij * distance(indexPeriodQ,
            indexPeriodQ + 1));
        delta -= (isSchQ * isSchPrevQ * cij * distance(indexPeriodQ,
            indexPeriodQ - 1));
    }
    return delta;
}
```

o visto de otra manera:

$$\Delta(i,p,q) = \sum_{j=1}^{E}(t_{ip}t_{j(p+1)}c_{ij}d_{p(p+1)} + t_{ip}t_{j(p-1)}c_{ij}d_{p(p-1)}) -$$

$$- \quad (t_{iq}t_{j(q+1)}c_{ij}d_{q(q+1)} + t_{iq}t_{j(q-1)}c_{ij}d_{q(q-1)})$$

4. Examen y evaluación

No se ha tenido todo el tiempo deseado para realizar pruebas con los distintos algoritmos para obtener sus parámetros óptimos (probabilidad de cruce y de mutación, tamaño de la población, número de generaciones máxima y número de asignaturas a evaluar en cada iteración -en algoritmo memético, sobre todo). No obstante, se muestran a continuación los mejores resultados obtenidos con cada algoritmo (con distintos tamaño de población y generaciones):

Algoritmo	Probabilidad de cruce	Probabilidad de mutación	Fitness mejor individuo
Simple	0,8	0,005	410
Simple modificado	0,8	0,005	220
Memético	0,6	0,007	190

5. Trabajo futuro y conclusiones

Se han aplicado dos tipos de algoritmos genéticos (memético y simple) a la resolución del problema de encontrar un calendario de exámenes. Los resultados encontrados son satisfactorios en tiempo pero no en rendimiento, ya que precisan de modificaciones manuales. Una posible continuación de este trabajo sería buscar funciones de fitness mejores y parametrizar los algoritmos con pruebas más exhaustivas.

Referencias

[1] Edmund K. Burke and J. P. Newall. A multistage evolutionary algorithm for the timetable problem. *IEEE Transactions on Evolutionary Computation*, 3(1):63–74, 1999.

[2] Tim B. Cooper and Jeffrey H. Kingston. The complexity of timetable construction problems. In *Proceedings of the First International Conference on the Practice and Theory of Automated Timetabling (ICPTAT '95)*, pages 511–522, 1995.