Resolución de problemas de asignación de recursos mediante Algoritmos Genéticos

Ing. Raúl Oscar Klenzi, Dr. Raymundo Forradellas, Dr. Francisco Ibañez.
Instituto de Informática (LISI/IdeI).
Facultad Ciencias Exactas Físicas y Naturales.
Universidad Nacional de San Juan.
rklenzi, kike, fibanez@iinfo.unsj.edu.ar

RESUMEN

En el presente trabajo se pretende aplicar una alternativa derivada de la Inteligencia Artificial que son los Algoritmos Genéticos. Esta es una excelente metodología de exploración y explotación de grandes espacios de búsquedas. En éste caso concreto se los utilizará como optimizadores, característica que los destaca sobre otras estrategias de optimización.

Se pretende asignar cajeras en cajas de un supermercado en donde se deben cumplir ciertas restricciones que se enumeran a continuación.

1) ***Todas las cajeras deben ser distintas (Alldistint).***
2) ***Una cajera no debe ocupar una caja determinada (notincash).***
3) ***Una cajera debe ocupar una caja determinada (incash).***
4) ***Una cajera no debe ocupar un turno definido (notinturn).***
5) ***Una cajera debe ocupar un turno definido (inturn).***
6) ***Una cajera no debe estar al lado de otra determinada.***

Para la aplicación se utiliza una herramienta disponible en Internet desde la cual se manipulan los algoritmos en sí, en cuanto a estrategias de selección, transposición, mutación y de terminación; a lo que debe adicionarse una correcta elección de la representación y función de aptitud.
[1]

[1] Nota: Este trabajo se desarrolló en el marco del proyecto CICYTCA (UNSJ) *"Resolución de Problemas de Asignación de Recursos Mediante Técnicas de Inteligencia Artificial"*

INTRODUCCIÓN

Existen distintas técnicas para encarar la resolución de problemas de asignación de recursos. El objetivo planteado en este trabajo pretende resolver estos problemas aplicando metodologías de Inteligencia Artificial (IA).

Para ello primeramente daremos una definición aproximada de lo que es IA.

Se dice que un programa de computación es del área de la Inteligencia Artificial si presenta características de aprendizaje, razonamiento, interrelación con el medio que **revela características de inteligencia humanas.**

Así pues, un soft de IA pretende emular características humanas en cuanto a tener capacidades de procesamiento similar al del cerebro humano (Redes Neuronales RN), Administrar todo su conocimiento, experiencia, actuando con sentido común (Sistemas Expertos SE). Aplicar técnicas que mimeticen la teoría Darwiniana de supervivencia del más apto (Algoritmos Genéticos AG, Computación Evolutiva CE), y finalmente comportarse en forma satisfactoria ante información incierta, incompleta, o sin la exactitud suficiente (Lógica Borrosa LB).

Este trabajo centra su estudio en las aplicaciones de AG.

2. ALGORITMOS GENETICOS

Se introduce a los Algoritmos Genéticos, qué son, de dónde vinieron, y en que se parecen y difieren de otros procedimientos de búsqueda, de acuerdo a la publicación de ***Genetic Algorithms in Search and Optimization,*** David Edward Goldberg, Addison-Wesley Pub. Co., 1989 ISBN 0-201-15767-5

2.1 ¿Que son los Algoritmos Genéticos?

Los Algoritmos Genéticos son algoritmos de búsqueda basados en los mecanismos de la selección natural y de la genética natural.

> *Combinan la supervivencia del más adaptado entre estructuras strings (cadenas de caracteres) con un intercambio de información estructurado y a la vez aleatorio dando como resultado un algoritmo de búsqueda con algo de la búsqueda humana instintiva. En cada generación se crea un nuevo conjunto de criaturas artificiales (strings) usando partes de los más adaptados; ocasionalmente se intenta con una nueva parte para mejorar la medición. Si bien son aleatorios, los Algoritmos Genéticos no son puro azar, explotan eficientemente la información histórica para hallar los nuevos puntos de búsqueda esperando un mejor rendimiento.*

El tema central en la investigación en los Algoritmos Genéticos ha sido la robustez, el equilibrio entre eficiencia y eficacia necesarias para la supervivencia en muchos ambientes.

2.2 ¿ En que se diferencian los Algoritmos Genéticos de los métodos tradicionales.?

Los Algoritmos Genéticos son diferentes de los procedimientos de búsqueda tradicionales en cuatro aspectos:

1- Los Ags trabajan con una codificación del espacio parametrico y no con los parametros mismos.

2- Los Ags buscan en una poblacion de puntos y no en un solo punto.

3- Los Ags usan una función objetivo de rentabilidad como información, no derivadas u otro conocimiento auxiliar.

4- Los Ags usan reglas de transición probabilísticas no deterministicas.

Los Algoritmos Geneticos requieren que el conjunto de parametros naturales del problema de optimización sea codificado como un string de longitud finita sacado de algún alfabeto finito.

3. OPTIMIZACIÓN

Las aplicaciones de Software se dividen en dos categorías : Análisis y Síntesis. Las aplicaciones de análisis se representan por el tradicional modelo entrada-salida de datos, procesando los datos de entrada y obteniendo las correspondientes salidas por procedimientos procedurales o heurísticos. Esta es una tarea dificultosa, dado que en la mayoría de los casos no existen fórmulas ni reglas para lograr salidas desde las entradas. Se complica aún más a las restricciones impuestas para que los valores de entrada sean aceptables. Optimización es el proceso de derivar desde las entradas que cumplen ciertas restricciones las salidas deseadas.

3.1 Optimización usando Algoritmos Genéticos

Los Algoritmos genéticos (AG) son técnicas que resuelven problemas de optimización inspirados en la teoría de la evolución y la biogenética. Son una buena alternativa explorando grandes espacios de búsqueda donde existen óptimos globales y locales. Los aspectos básico de los AG son :

1. Representan las posibles soluciones del problema como un string de parámetros (números). Este string es llamado ***cromosoma*** y los parametros ***genes.***
2. Aleatoriamente crean un número (generación) de estos cromosomas.
3. Se calcula la aptitud de cada cromosoma como forma de calificar los mismos en orden a su función de aptitud.
4. Se crea una nueva generación de cromosomas, seleccionando al azar un par de cromosomas (padres) y mezclando sus genes se logran los cromosomas hijos. Este proceso es llamado ***crossover*** y la selección de los padre se hace de acuerdo a la función de aptitud de cada uno.
5. Repitiendo los pasos 3 y 4 para un número dado de ciclos (generaciones).

La aleatoriedad del proceso anterior permite la efectiva exploración de un gran espacio de soluciones. Mientras la selección de soluciones efectivas (cromosomas) y la mezcla de sus genes permite la acumulación de buenas características desde soluciones parcialmente buenas. Como resultado, los AG exploran grandes dominios y convergen a buenas soluciones relativamente rápido. Dando una buena relación entre el tiempo empleado y la calidad de la solución. Típicamente se desea encontrar una buena solución en un corto tiempo y no la solución óptima en tiempo excesivo.

Los dos pasos basicos en la búsqueda de soluciones usando AG son una adecuada representación del problema y un método para encontrar la función de aptitud (función de costo) a evaluar permanentemente. La representación mas simple de un problema es por medio de un string de números. Cada número es representado por un gene que puede estar restringido a una valor máximo y mínimo. La función de costo se define como el costo de una solución dada para un conjunto de valores de genes. En tal representación cada gene representa un parámetro diferente de la solución.

Alternativamente un cromosoma se puede usar para representar una secuencia de tareas que se requiere optimizar. En este caso el número de genes será igual al **número de tareas a ser secuenciado y el valor de cada gene único con un valor que irá** desde uno al número de tareas.

4. EL PROBLEMA

El problema a abordar en este trabajo consta en lograr la asignación de trabajadores (cajeras) a sus correspondientes lugares de trabajo (cajas), cumpliendo determinadas restricciones.

Así por ejemplo, las restricciones existentes, entre otras, son del tipo:

7) Todas las cajeras deben ser distintas (Alldistint).
8) Una cajera no debe ocupar una caja determinada (notincash).
9) Una cajera debe ocupar una caja determinada (incash).
10) Una cajera no debe ocupar un turno definido (notinturn).
11) Una cajera debe ocupar un turno definido (inturn).
12) Una cajera no debe estar al lado de otra determinada.

En función del problema real de un supermercado, el mismo puede constar de k2 cajeras, k1 cajas, y t turnos, y haber una cajera especializada en el trato de tarjetas de créditos, mas no sabe el tratamiento que se le da al cobro con tickets canasta o restaurante, a su vez no puede venir en el turno de la tarde. Esto involucra tener en cuenta, a la hora de elevar una solución, restricciones del tipo 1), 2), 3) y 4).

Mientras el número de cajeras, cajas, y turnos sea reducido puede llegarse a realizar una asignación en forma manual, pero en la medida que estas variables empiezan a crecer, el problema se hace manualmente intratable. Aún para buenos equipos de computación en la medida que aumentan las restricciones se llega a explosiones combinatorias.

5. HERRAMIENTA UTILIZADA

Se ha definido ya el uso de AG como metodología de búsqueda de una solución, para ello se hará uso de una herramienta del Instituto de Tecnología de Massachusset MIT, disponible en INTERNET y que no es más que una colección de bibliotecas en C++ que permiten generar y correr distintos tipos de Algoritmos Genéticos.

La herramienta aludida recibe el nombre de GALIB2.42 y absolutamente todas sus características se encuentran en el correspondiente archivo HTML y en el presente informe solo destacaremos algunas de ellas.

Hay tres cosas que se deben analizar al intentar resolver un problema mediante Algoritmos Genéticos.

1. Definir una representación.
2. Definir el operador genético.
3. Definir la función objetivo.

GALIB ayuda con los dos primeros ítems proveyendo ejemplos desde los cuales se pueden construir la representación y operadores particulares. En muchos casos se pueden usar las representaciones y operadores existentes con pequeñas modificaciones.

La función objetivo la desarrolla completamente el usuario de GALIB. Una vez definidas la representación, operadores y una medida del objetivo, se puede aplicar cualquier Algoritmo Genético para encontrar la mejor solución a un problema dado.

Cuando se usa un Algoritmo Genético para resolver un problema de optimización, se debe estar en condiciones de representar un solución a su problema en una única estructura de datos. El Algoritmo Genético (AG) creará una población de soluciones basada en una muestra de la estructura de datos provista. El AG opera sobre la población y evoluciona hacia la mejor solución. En GALIB la librería contiene cuatro tipos de Genomas:

- GAListGenome.
- GATreeGenome.
- GAArrayGenome.
- GABinaryStringGenome.

Clases que se derivan de la clase base GAGenome y una estructura de datos como la indicada por sus nombres como ejemplo: GAListGenome se deriva de la clase GAList también como de la clase GAGenome. Usa una estructura de datos que trabaja con la definición del problema a resolver.
Hay diferentes tipos de Algoritmos Genéticos. Galib incluye cuatro tipos básicos:

- Simple, ***GASimpleGA ga(genome);***
- Estado-Permanente, ***GASteadyStateGA ga(genome);***
- Incremental, ***GAIncrementalGA ga(genome);***
- Islas de poblaciones, ***GADemeGA ga(genome);***

Estos algoritmos difieren en la forma en que crean nuevos individuos y reemplazan los viejos durante el curso de la evolución.

GALIB provee dos mecanismos primarios para extender las capacidades de construir objetos. Sobre todo (y desde el punto de vista del C++) se pueden derivar sus propias clases y definir nuevas funciones miembros. Solamente se necesitan hacer ajustes menores al comportamiento de la clase Galib, en la mayoría de los casos se define una sola función y se le dice a la clase Galib que la use en lugar de las definidas por default.

Los AG cuando están correctamente implementados, son capaces de realizar tanto la tarea de exploración (búsqueda a lo ancho) como de explotación (búsqueda local) del

espacio de búsqueda. El comportamiento que tendrá dependerá como trabajen los operadores, y de la forma del espacio de búsqueda.

Como anteriormente se mencionó un ciclo de un AG involucra tres operaciones, selección, transposición y mutación, GALIB posee distintas alternativas para cada uno de estos operadores:

5.1 Selección

- *GARouletteWheelSelector().*
- *GARankSelector().*
- ***GATournamentSelector().***
- *GADSSelector().*
- *GASRSSelector().*
- *GAUniformSelector().*

5.2 Transposición

- *ga.crossover(GAStringPartialMatchCrossover).*
- *ga.crossover(GAStringTwoPointCrossover).*
- *ga.crossover(GAStringGenome::UniformCrossover).*
- *ga.crossover(GAStringGenome::EvenOddCrossover).*
- *ga.crossover(GAStringGenome::OnePointCrossover).*
- *ga.crossover(GAStringGenome::OrderCrossover).*
- *ga.crossover(GAStringGenome::CycleCrossover).*

5.3 Mutación

- *genome.mutator(GAStringSwapMutator).*
- *genome.mutator(GAStringFlipMutator).*
- *genome.mutator(GAStringGaussianMutator).*

Se sugiere que para más información sobre las características del soft, remitirse a: mbwall@mit.edu.

5.4 Definiendo una Representación

Para el presente problema se opto por la representación ***GAStringGenome*** que es una especialización de la estructura ***GAArrayGenome***.
Así, consideremos un ejemplo práctico respecto de la forma que tendría esta representación.

Ejemplo:

Supongamos el caso de 7 posibles cajeras, a distribuirse en 5 cajas; aquí el string (genome-cromosoma) tendría una longitud equivalente al número de cajas y los valores, que cada posición del string podría llegar a tomar (allele), sería el nombre de la cajera.

Cromosoma, genoma, o string de longitud 5 = cantidad de cajas

Gen1	Gen2	Gen3	Gen4	Gen5

La posición del Gen1 se corresponde con la caja1; La del gen2 con la caja 2 y así sucesivamente.
El valor que cada Gen puede tomar es el correspondiente al nombre de cada cajera, en este caso:
Nombre cajera 1 = A, Nombre cajera 2 = B, ..., Nombre cajera 7 = G.
Una vez instanciado un Gen recibe el nombre de allele. De esta manera posibles string serían:

A	B	C	D	E

F	B	A	C	G

La cantidad de strings que pueden llegar a generarse, y que conforman el espacio de búsqueda, es igual a:

ESPACIO DE BUSQUEDA = ***(NUMERO DE CAJERAS)*** $^{\textit{NUMERO DE CAJAS}}$
que para nuestro ejemplo

ESPACIO DE BUSQUEDA = $7^5 = 16807$.

Esto significa que la exploración para encontrar la o las respuestas correctas se debe hacer sobre 16.807 strings distintos, por turnos, entre los que se encuentran algunos como:

C	C	C	C	C

A	E	A	A	E

Y que deberán ser eliminados por nuestro algoritmo atendiendo a las restricciones que se estén considerando, de hecho la restricción de que todas las cajeras sean distintas eliminaría los dos últimos strings presentados.

5.5 Definiendo un Operador Genético

En este caso y tras varias alternativas de prueba y error, teniendo ya la función de aptitud definida (se hará en el paso siguiente), se eligió un Algoritmo Genético del tipo de Estado-Permanente de la biblioteca de GALIB.

GA with overlapping populations (steady-state) *Este algoritmo genético usa solapamiento de poblaciones en cantidades especificadas por el usuario. El algoritmo crea una población de individuos por clonación de cromosomas. En cada generación el algoritmo crea una población temporal de individuos, agrega estos a la población previa, entonces elimina los peores individuos restableciendo, a la población, su tamaño original.*

Particularmente se tiene como operadores interno de selección, transposición y mutación respectivamente a:

GARankSelector():
El seleccionador escoge el mejor miembro de la población cada vez.

ga.crossover(GAStringGenome::UniformCrossover).
[illegible] entre strings de acuerdo a una dist[illegible] uniforme de posiciones a modo de ejemplo si tenemos dos strings

PADRES

A	B	C	C	C

B	***D***	***D***	***G***	***E***

HIJOS

A	***D***	C	***G***	C

B	B	***D***	C	***E***

genome.mutator(GAStringFlipMutator).
Aquí se cambia gen un al azar, reemplazandoló por otro gen del alfabeto código

Elegidos ya los operadores, resta aún definir el valor de los parámetros con que cada uno se llevará a cabo, en nuestro caso,

```
params.set(gaNnReplacement,pob );        //porcentaje de la poblacion a reemplazar
params.set(gaNpopulationSize, pob1);     // tamaño de la población
params.set(gaNpCrossover, 0.9);  //probabilidad de crossover
params.set(gaNpMutation, 0.012); //probabilidad de mutación
params.set(gaNnGenerations, gener1);     // número de generaciones.
```

El tamaño de la población pob1, se trato de que responda a un crecimiento de tipo logarítmico, en función del crecimiento de las cajas y cajeras.

Los rangos que se han planteados para cajas y cajeras van desde 5 a 100 y 7 a 120 respectivamente, y la población de strings nunca excede los 500 individuos.

Esto significa que se hace evolucionar como máximo una población que ronda los 500 individuos intentando encontrar soluciones válidas en un espacio de búsqueda que como máximo es: $\mathbf{120^{100}=8,281797452201*10^{208}}$

La cantidad de individuos reemplazados en cada generación se determina por la variable *pob* y en nuestro caso se encontraron buenos resultados haciendo que la cantidad de individuos reemplazados coincida con la peor mitad de la población.

La cantidad de generaciones que se pretende evolucionar cada población responde también a una expresión logarítmica función de la cantidad de cajeras **k2**. La cantidad máxima de generaciones para 120 cajeras y 100 cajas es: 360.

Un aspecto importante en la definición del algoritmo utilizado es la forma en que terminará cada ciclo del AG. Entre otras posibilidades que brinda la herramienta estan:

GABoolean GAGeneticAlgorithm::TerminateUponGeneration(GAGeneticAlgorithm &)
En ésta alternativo una vez alcanzado el número de generaciones que se quiere hacer evolucionar a la población automáticamente se detiene con la esperanza que se haya encontrado la solución.

GABoolean GAGeneticAlgorithm::TerminateUponConvergence(GAGeneticAlgorithm &)
Este caso difiere del anterior en que la evolución se detiene cuando los strings han convergido a un cierto valor de aptitud. Dicho de otra manera si el valor de aptitud del mejor strings no ha cambiado durante un cierto número de generaciones se asume que el algoritmo a convergido.

Es ésta última alternativa la utilizada para concluir con la búsqueda en la población principal

5.6 Definiendo la Función Objetivo

Una medida de la función Objetivo la da el número de restricciones que no se violan, es justamente este valor el que se intentara maximizar. De modo que la función objetivo es maximizar el número de restricciones que no se violan.

El problema considerado tiene una particular ventaja como es que la cantidad de restricciones que deben satisfacerse se conoce en función del número de cajas, cajeras y turnos.

Así cuando se plantea que se debe satisfacer la restricción de que todas las cajeras sean distintas, la cantidad que se debe alcanzar para k1 cajas es:

Cantidad de restricciones del tipo Alldistint, por cada turno = (k1(k1-1))/2*

De modo que si tenemos 5 cajas la restricción de todas distintas deberá dar 5*4/2=10
Si hay 100 cajas, las restricciones todas distintas a satisfacer serán: 100*99/2=4950.

La restricción del tipo que una cajera no ocupe una determinada caja para cualquier turno es igual a la cantidad de cajeras que no deben estar en esas cajas. Entonces si se tiene que tres cajeras no pueden ocupar cajas determinadas, el objetivo será tres, o sea se cuenta que tal caja no este ocupada por aquella cajera.

La restricción del tipo que una cajera ocupe una determinada caja tiene similares características que la restricción anterior.

Finalmente la restricción que una cajera no ocupe un turno determinado es igual al número de restricciones de este tipo planteada por la cantidad de cajas. Esto por cada turno.

Una alternativa para hacer evolucionar al algoritmo en cuanto a la función objetivo podría ser que la población inicialmente generada tienda a cumplir con la suma de las anterior restricciones, esto es:

k1(k-1)/2 + cant. Restricc. del tipo no encaja + cant. de Restricc. del tipo en caja + cant. de Restricc. del tipo no en turno*k1*

De modo que si se solicita que la cajera E no ocupe el primer turno, que la A, B y C ocupen las tres primeras cajas, y que la cajera D no ocupe la quinta caja, a lo que hay que adicionarle el hecho que todas sean distintas, totalizamos una cantidad de 19 restricciones que deben ser satisfechas. Una solución propuesta es:

A	*B*	*C*	*D*	*F*

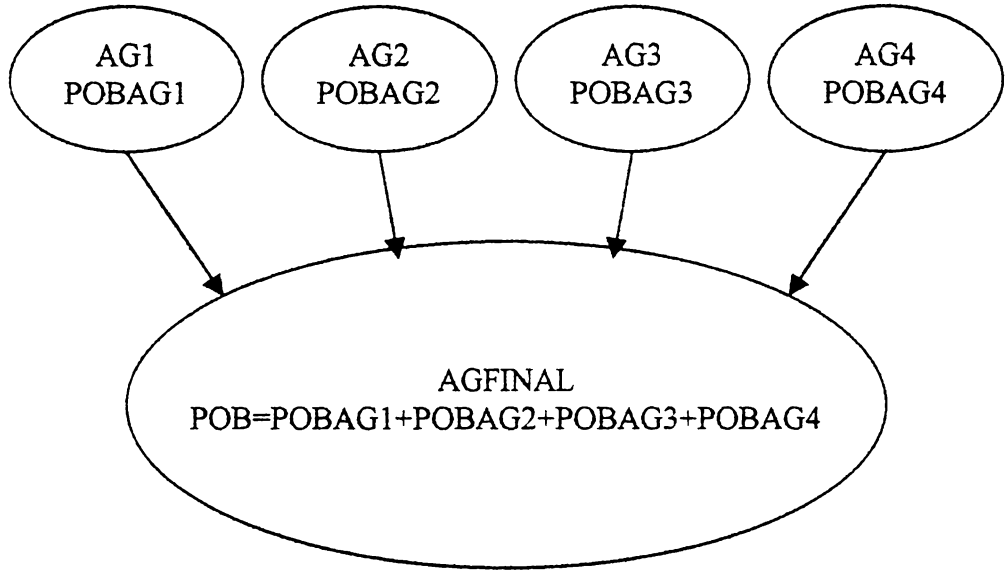
A	B	C	D	G

En la medida que el espacio solución se reduce con respecto al espacio de búsqueda (aumenta el número de restricciones) resulta cada vez más difícil satisfacerlas, aún conociendo la cantidad de restricciones que deben ser satisfechas.

5.7 Características de la población inicial.

La forma de generar inicialmente una población al azar y hacerla evolucionar hacia la solución es una alternativa válida pero que es más lenta que la que se propone seguidamente.

Para garantizar que en la población principal tenga strings que satisfagan los distintos tipos de restricciones se corren inicialmente tantos AG como tipos de restricciones existan. De esta manera si existen cuatro tipos de restricciones se correrán cuatro AG que generaran 1/4 de la población total a evolucionar respectivamente.
En este caso la POBAG1 tiene solamente strings de cajeras distintas, POBAG2 strings de cajeras que no deben estar en una caja para cualquier turno, POBAG3 strings de cajeras que para cualquier turno ocupan una caja determinada, y finalmente POBAG4 strings que solo tienen cajeras que no deben estar en un turno dado.

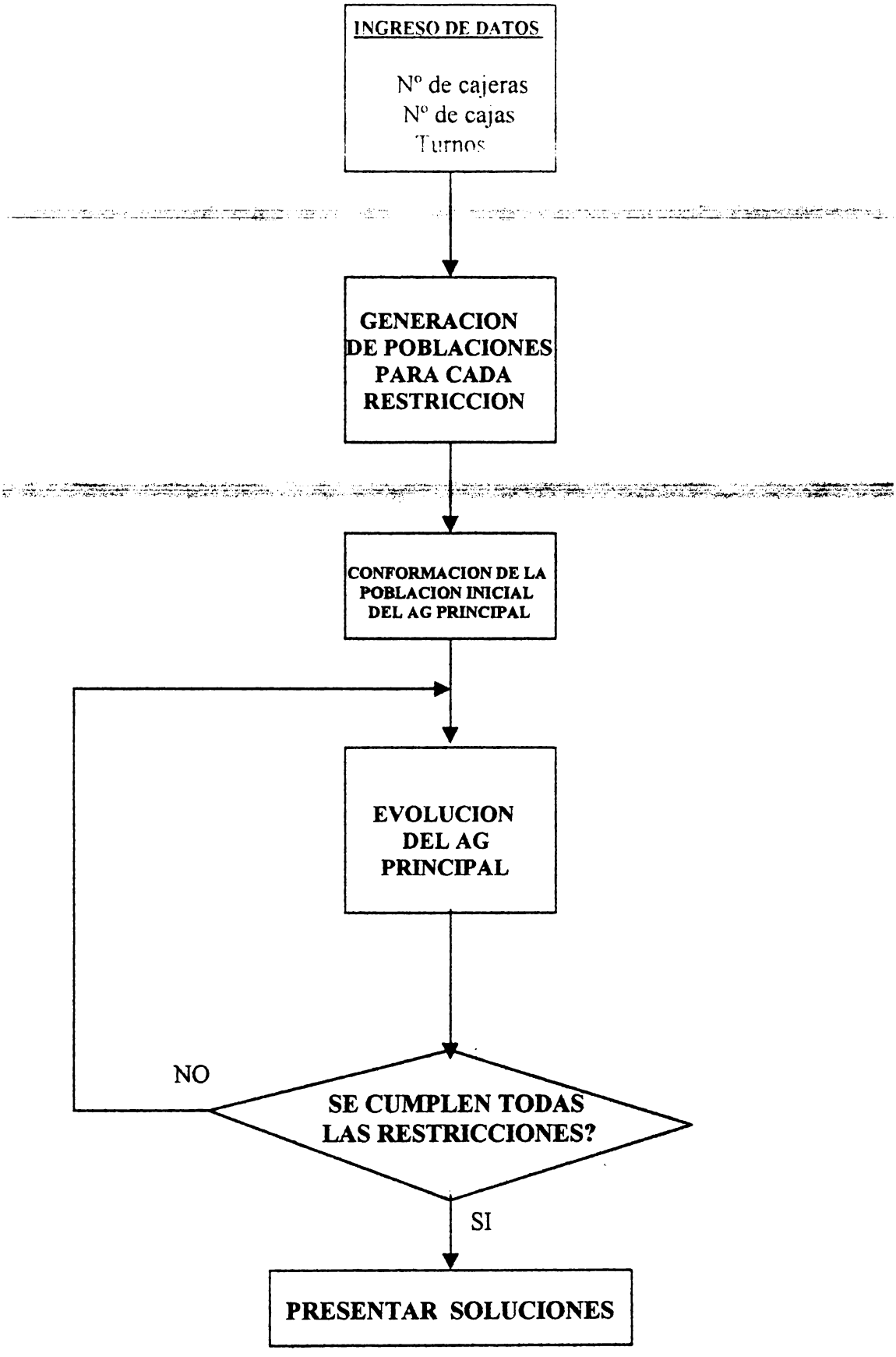


A partir de allí el AGFINAL evoluciona hasta encontrar la solución deseada, con los parámetros especificados con anterioridad y teniendo una función objetivo que se presenta a continuación. En ella se reflejan los distintos pesos que las diferentes restricciones llevan pues hay restricciones más difícil de satisfacer que otras, aquellas con mayor peso son las más difícil de satisfacer.

score= ((alldistint**2.12**) + notincash + (incash**10**) + (notinturn**2.12**));

Los pesos que cada restricción tiene, se han determinado en forma empírica.

6. DIAGRAMA DEL ESQUEMA PROPUESTO EN LA IMPLEMENTACIÓN.



De acuerdo al diagrama anterior el AG principal puede llegar a evolucionar varias veces hasta tanto encuentre una solución que cumpla con todas las restricciones. En función de los parámetros que el Ag posee esto rara vez ocurre puesto que la mayoría de las veces una sola corrida del algoritmo permite encontrar la solución buscada.

7. SOLUCIONES ENCONTRADAS

A continuación se presenta una tabla en la que aparecen distintas soluciones a **problemas planteados y resueltos mediante técnicas de AG y con la función objetivo expresada con anterioridad.**

N° CAJERAS	N° CAJAS	TURNOS	POBLACION EN STRINGS	RESTRICIONES	TIEMPO DE CALCULO (SEG)
7	5	3	48	68	0.90
20	11	3	100	387	1,10
40	31	3	100	2175	7,81
120	100	3	100	17286	107,56

SOLUCIONES AG

A modo comparativo se presentan dos ejemplos tratados mediante CLP-Ilog y se puede visualizar la diferencia de tiempos logrados con esta alternativa respecto de los AG.

N° CAJERAS	N° CAJAS	TURNOS	RESTRICIONES	TIEMPO DE CALCULO (SEG)
7	5	3	44	0.055
120	100	3	23755	2,69

Según publicación del VII RPIC Reunión de Trabajo en Procesamiento de la Información y Control pag 29[Rueda, Arias, Forradellas, Ibañez]. Setiembre 1997 San Juan Argentina.

SOLUCIONES CLP-Ilog

Las comparaciones fueron desarrolladas en un máquina Pentium y aplicaciones win32 .

8. CONCLUSIONES

Ventajas Teóricas de la Propuesta

- Se obtienen respuestas cercanas al óptimo con solo algunas bibliotecas implementadas en C++.
- Los tiempos de demora en encontrar una solución se pueden seguir reduciendo, ajustando determinados parámetros que hacen al funcionamiento de los AG.

Desventajas teóricas de la Propuesta

- Dada la característica pseudoaleatoria de la misma no siempre entrega la misma solución para determinado grupo de restricciones ya que depende de la dirección en la que se movió la búsqueda. Esto se aleja un tanto de la realidad en la que cajeras de un supermercado ocupan las inmediaciones de un mismo lugar.

- En el trato de CLP-Ilog la respuesta inicial para un mismo grupo de restricciones es siempre la misma.

Desventajas de la Implementación

- La implementación programada no reconoce cuando un conjunto de restricciones no puede ser satisfechas. Por lo que intentaría encontrar una respuesta indefinidamente.

- No permite generar absolutamente todas las respuestas posibles para un conjunto de restricciones dadas. En caso de obtenerlas el tiempo en alcanzarlo puede resultar excesivo.

- El tiempo que demanda la búsqueda resulta excesivo en la medida que aumenta el número de restricciones.

COMPARACION SOBRE:	ALGORITMO PLANTEADO	CLP-Ilog
COMPLEJIDAD DEL SOFTWARE.	BIBLIOTECAS EN C++	BIBLIOTECAS EN C++ MAS HERRAMIENTA Ilog
CARACTERISTICA DE LA BUSQUEDA	PSEUDOALEATORIA	DETERMINISTICA
A IGUALDAD DE CONDICIONES INICIALES.	NO ENCUENTRA SIEMPRE LA MISMA SOLUCION.	ENCUENTRA LA MISMA SOLUCION.
ANTE POCOS CAMBIOS EN LAS RESTRICCIONES	OBTIENE GENERALMENTE SOLUCIONES ALEJADAS DE LAS ANTERIORES.	OBTIENE SOLUCIONES CERCANAS A LAS ANTERIORES
TIEMPOS DE COMPUTOS PARA ENCONTRAR SOLUCIONES.	EN ESTA IMPLEMENTACION $T_{CLP\text{-}Ilog}*30$	$T_{CLP\text{-}Ilog}$
SI NO EXISTEN SOLUCIONES PARA UN CONJUNTO DE RESTRICCIONES DADAS.	ESTA IMPLEMENTACION NO RESPONDE	Ilog PERMITE GENERAR UNA RESPUESTA CUANDO EL SISTEMA NO TIENE SOLUCIONES
ENCUENTRA TODAS LAS SOLUCIONES?	ESTA IMPLEMENTACION NO, LOGRARLO PUEDE INSUMIR MUCHO TIEMPO DE COMPUTO	SI, CON ESCASO ESFUERZO DE CALCULO ADICIONAL

9. BIBIOGRAFÍA CONSIDERADA

- *Genetic Algorithms in Search and Optimization.* David Edward Goldberg, Addison-Wesley Pub. Co., 1989 ISBN 0-201-15767-5.
- *Un criterio para seleccionar operadores genéticos para resolver CSP. Trabajo presentado en el CACIC '98.(paper)*
- *Heuristic Genetic Algorithms for Constrained Problems A.E. Eiben, P-E. Raué, Zs. Ruttkay. Artificial Intelligence Group. Departament of Mathematics and Computer Science Vrije Universiteit Amsterdam.(paper)*
- ***Solving Constrain Satisfaction Problems Using Genetic Algorithms. A.E. Eiben, P-E. Raué, Zs. Ruttkay. Artificial Intelligence Group. Departament of Mathematics and Computer Science Vrije Universiteit Amsterdam.(paper)***
- ***A Genetic Algorithm for Resource_Constrained Scheduling*** Tésis Doctoral de Matthew Bartschi Wall, Massachusetts Institute of Tecnology