Análisis Comparativo de Diferentes Soluciones Alternativas para el Problema de La Mochila

Alfonso Hugo, Bermúdez Carlos, Stark Natalia
Proyecto UNLPAM-09/F009[1]
Departamento de Informática - Facultad de Ingeniería
Universidad Nacional de La Pampa
Calle 110 esq. 9
(6360) General Pico – La Pampa – Rep. Argentina
E-mail: { alfonsoh, bermudez,nstark }@ing.unlpam.edu.ar
Telef.: (02302)422780/422372, Ext. 6302

Gallard Raúl
Proyecto UNSL-338403[2]
Departamento de Informática
Universidad Nacional de San Luis
Ejército de los Andes 950 - Local 106
(5700) - San Luis -Argentina
E-mail: rgallard@unsl.edu.ar
Telef.: +54 2652 420823
Fax : +54 2652 430224

ABSTRACT

En la actualidad la Computación Evolutiva se usa para buscar soluciones a problemas complejos para los que otras técnicas pueden insumir mucho tiempo y que, por lo general, proveen una única solución óptima. El problema considerado para este análisis, clasificado como NP-duro, es el *zero-one knapsack problem*, comúnmente conocido como el *problema de la mochila*. Este es un problema de planificación en el que, dada una mochila con una cierta capacidad y un conjunto de ítems que poseen un peso y un beneficio asociado para ser seleccionado, se busca encontrar planes de carga adecuado para llenar la misma con el mayor número de ítems y a la vez, tratando de lograr un máximo beneficio.

Esta presentación muestra un trabajo donde se implementaron diversas soluciones propuestas a este problema por diversos autores: Martello y Toth [12], Michalewicz [13], Olsen [16], Liepins [10], Nakano [15] y Orvosh [17]. Las siguientes secciones muestran un análisis comparativo de la performance alcanzada en cada uno de ellos.

PALABRAS CLAVES: *computación evolutiva, zero-one knapsack problem, problema de la mochila, planificación*

---

[1] Grupo de Investigación apoyado por la Universidad Nacional de La Pampa.
[2] Grupo de Investigación apoyado por la Universidad Nacional de San Luis y la ANPCYT (Agencia Nacional para Promover la Ciencia Y la Tecnología).

1. INTRODUCCIÓN

La Computación Evolutiva toma como base la *teoría de la evolución* de Charles Darwin, según la cual las especies van evolucionando para adaptarse al medio que las rodea, de forma que aquellos individuos que mejor se adapten al medio tienen mayor probabilidad de sobrevivir hasta la edad adulta y procrear, consiguiendo que sus genes pasen a la siguiente generación.

La optimización es el proceso de encontrar la mejor solución (u óptimo) de un conjunto de soluciones [8]. En forma general el problema se puede formular de la siguiente manera [18]:

$$x^0 = \underset{x \in \chi}{\arg\operatorname{opt}}\ f(x)$$

en el que la aplicación $f(\bullet)$ asocia a cada elemento x del conjunto χ un vector de números reales $f(x) \in \Re^{\Re}$. Naturalmente, no es necesario que $f(x)$ tenga una forma cerrada o explícita. Los candidatos a solución x son estructuras de objetos (conjuntos, árboles, grafos, etc.) los cuales se pueden caracterizar inequívocamente a través de m atributos en cierto orden. Al dominio del problema se le denota χ y no necesariamente coincide con *el espacio de búsqueda* del algoritmo (usado para encontrar la solución al problema), que se denota X.

Si estudiamos la complejidad de los problemas de optimización se dice que son *polinómicamente programables* cuando existen algoritmos de resolución que dan una respuesta tras realizar un número de operaciones que es, a lo sumo, una función polinómica del tamaño del problema. Este es un requisito mínimo de complejidad, es decir, si un problema no es polinómicamente programable todos los métodos de resolución serán inútiles para tamaños crecientes del problema ya que ninguna computadora sería capaz de afrontarlos en un tiempo razonable. Generalmente en la práctica la mayoría de los problemas de optimización son *no polinómicamente programables.* Por lo anteriormente dicho puede ser imposible encontrar el óptimo global en algunas aplicaciones o no haber una manera de chequear si una solución es un óptimo global. En muchos casos encontrar el óptimo global puede ser inconveniente dado el tiempo requerido. Encontrar una buena solución (un óptimo local) rápidamente puede ser más deseable que encontrar la mejor solución (el óptimo global) lentamente, lo cual ha sido observado al resolver este mismo problema con otras técnicas de optimización tales como el Simulated Annealing y el Tabu Search [1].

Un problema de optimización clásico es el problema de la mochila que consiste en encontrar un plan de carga adecuado para llenar la misma con el mayor número de ítems, tratando de lograr un máximo beneficio. Para ello se dispone de un conjunto de n ítems, los cuales tienen un peso y un beneficio asociado, se debe seleccionar un subconjunto de dichos ítems de forma que la suma de los beneficios sea máxima y la suma total de sus pesos no supere la *capacidad* de la mochila.

2. PROBLEMA DE LA MOCHILA

A continuación se presenta el modelo del problema:

Dado un conjunto de pesos W[i], beneficios P[i] y capacidad C, encontrar un vector (en nuestro caso será *binario*) x=<x[1],..,x[n]> tal que,

$$\text{Max } P(x) = \Sigma_{i=1..n}\, x[i]*P[i]$$

tal que:

$$\Sigma_{i=1..n}\, x[i]*W[i] <= C$$

La generación de los pesos y beneficios puede ser aleatoria, pero según Martello [12], la dificultad del problema se ve afectada por la correlación que pueda existir entre los beneficios y pesos. En este caso se tomaron datos no correlacionados, donde:

W[i]:= uniformemente aleatorio ([1..v]).

P[i]:= uniformemente aleatorio ([1..*v*]).

El problema de la mochila presenta una característica interesante y es la existencia de una restricción ($\Sigma_{i=1..n}$ x[i]*W[i]<= C). Durante la construcción de los individuos que formarán las distintas poblaciones, se puede dar el caso de que dicha restricción sea violada, por lo que se plantea la necesidad de implementar algún mecanismo, dentro del Algoritmo Evolutivo, que evite o disminuya el impacto que pueden tener los individuos ilegales dentro de la población.

Dentro de este trabajo hemos enfocado el problema usando distintos tipos de soluciones alternativas tales como: las *funciones de penalidad*, los *algoritmos de reparación* y los *algoritmos decodificadores*.

2.1. FUNCIONES DE PENALIDAD

Los algoritmos que implementan *funciones de penalidad* se basan en la evaluación de todos los individuos como si no existiesen restricciones (*fitness*=$\Sigma_{i=1..n}$ *x[i]*W[i]*), para posteriormente aplicar una penalidad que disminuya el *fitness* de aquellos individuos que violen las restricciones. La primera consideración que tenemos al implementar este tipo de algoritmos es el valor de penalidad que se aplica a los individuos ilegales.

Con la aplicación de penalidades muy altas existe el peligro de que individuos legales, aunque no óptimos, prevalezcan sobre los demás dentro de la población, haciendo que ésta converja prematuramente. Esto se debe a que, a menudo, la obtención de individuos legales óptimos requiere la producción de ilegales, pero las penalidades son tan altas que imposibilitan su reproducción.

Con la aplicación de penalidades moderadas o bajas el peligro que se presenta es inverso (esto es, puede suceder que individuos ilegales prevalezcan sobre los demás y hagan converger prematuramente la población). Esto se debe a que la penalidad que se aplica no es lo suficientemente alta y tiene como consecuencia que los individuos ilegales estén mejor clasificados en la evaluación que los legales.

Con respecto a los tipos de funciones de penalidad, existe una gran variedad: dinámicas, estáticas, en función del grado de violación, etc.. En el estudio realizado se han utilizado funciones de penalidad estáticas, pudiéndose diferenciar dos tipos de funciones:

Eval(x)=F(x)-Pen(x) y
*Eval(x)=F(x)*Pen(x).*

En el primer tipo de penalidad, se han implementado tres tipos de funciones *Pen(x)*: lineal, cuadrático y logarítmico, aplicándose esta función sólo a los individuos inválidos (si el individuo es válido *Pen(x)*=0).

Básicamente Michalewicz propone calcular el fitness con la siguiente función[13]:

$$Eval(x)= \Sigma_{i=1..n} x[i]*P[i]-Pen(x)$$

Donde:

Pen(x)=0 si el individuo es factible, y

Pen(x) se define de la siguiente manera dependiendo del tipo de penalidad que se desea aplicar para todos los otros casos.

- ✓ Penalidad Lineal: $Pen(x) = \rho*(\Sigma_{i=1..n} x[i]*W[i]-C)$
- ✓ Penalidad cuadrática: $Pen(x)= (\rho*(\Sigma_{i=1..n} x[i]*W[i]-C))^2$
- ✓ Penalidad logarítmica: $Pen(x)=\log_2 (\rho*(\Sigma_{i=1..n} x[i]*W[i]-C))$

Con $\rho=\max_{i=1..n}\{P[i]/W[i]\}$ y $C=0.5* \Sigma_{i=1..n} W[i]$

En el segundo caso, la función de penalidad se aplica sobre todos los individuos, puesto que no sólo penaliza a aquellos individuos que violan las restricciones, sino también a aquellos individuos que hacen un mal uso de la *mochila* ($\Sigma_{i=1..n}$ x[i]*W[i]<<C).

Olsen propone calcular el fitness de la siguiente forma [16]:

$$Eval(x)=(\Sigma_{i=1..n}\ x[i]*P[i]) * Pen(x)$$

Para todos los elementos (factibles o no):

$$Pen(x) = 1 - \frac{\left|\sum_{i=1}^{n} W[i]*x[i]-C\right|}{\delta}$$

$$\delta = min\ \{C, \left|\sum_{i=1}^{n} W[i]*x[i]-C\right|\}$$

En este trabajo, y a modo comparativo, también se ha introducido otro tipo de penalidad, *death penalty*. Esta penalidad consiste en eliminar aquellos individuos no factibles. Esta técnica resultó exitosa en los Estrategias Evolutivas [2] para problemas de optimización numérica, aunque tiene en contra el costo computacional que supone la evaluación de los individuos en aquellos problemas en que la generación de individuos ilegales es muy pequeña, además del hecho de que las soluciones no factibles no contribuyen con el *pool* genético de la población [13].

Si el individuo es factible:	$Eval(x)=\Sigma_{i=1..n}\ x[i]*P[i]$
Si el individuo no es factible:	Se genera aleatoriamente un individuo factible que lo reemplaza en la población.

2.2. ALGORITMO REPARADOR

Es otro enfoque que puede utilizarse. Este tipo de algoritmo consiste en transformar los individuos inválidos en individuos válidos. El problema de este tipo de algoritmos es que resultan computacionalmente intensivos, sobre todo en el caso de que el número de individuos no factibles dentro de la población sea elevado [13].

Para estos algoritmos se usa una cadena binaria de longitud *n* para representar una solución *x* del problema: El iésimo ítem se selecciona para la mochila si y solo sí X[i]=1.

La evaluación del fitness de *x* se determina como: Eval (x) = $\Sigma_{i=1..n}$ *x'*[i].P[i], donde *x'* es una versión reparada de *x*.

Los algoritmos de reparación se basan en la siguiente implementación:

```
Procedure repair(x)
Begin
    Mochila=false
    X'=X
    If Σ_{i=1..n} X'[i]*W[i]>C then
        Mochila=true
    While (Mochila) do
        i= select item_mochila
```

```
            X'[i]=0
            If Σ_{i=1..n} X'[i]*W[i]>C then
                X'[i]=0
        End
    End
```

Los algoritmos difieren en el procedimiento *select:*

$A_r[1]$ (reparación aleatoria). El procedimiento *select* selecciona aleatoriamente un elemento de la mochila.

$A_r[2]$ (reparación voraz). Aquí todos los ítems de la mochila se ordenan en orden decreciente del cociente beneficio/peso. El procedimiento *select* selecciona para eliminar el último ítem.

2.3. ALGORITMO DECODIFICADOR

Los decodificadores ofrecen un enfoque interesante en técnicas evolutivas. En estas técnicas, un cromosoma "da instrucciones al decodificador" de cómo construir una solución factible. Por ejemplo, una secuencia de ítems para este problema puede interpretarse como "tomar un ítem de la secuencia, si el mismo entra en la mochila". Tal interpretación conduce siempre a una solución factible.

Se utilizaron tres técnicas para generar la lista de ítem: decodificación aleatoria, decodificación voraz según orden decreciente del cociente p_i / w_i (beneficio-peso) y decodificación voraz según el parámetro p_i (beneficio).

Particularmente se trabajó con un string binario, el cual determina la secuencia de ítems a incorporar en la mochila, o sea que cada 1 en la iésima posición indica que el iésimo ítem de la lista es incorporado a la mochila siempre y cuando no supere la capacidad máxima fijada para la misma.

Los algoritmos de decodificación se basan en la siguiente implementación:

```
Procedure decode(x)
Begin
    Construir una lista L de ítems
    i=1
    SumaPesos = 0
    SumaBenef = 0
    While (i <= n) do
        j = x [i]
        remover el j-ésmo ítem de la lista L
        If SumaPesos + W[i] <= C then
        Begin
            SumaPesos = SumaPesos + W[i]
            SumaBenef = SumaBenef  + P[j]
        End
        i = i+1
    End
End
```

Los algoritmos difieren en el procedimiento *Construir:*

- ✓ A_d [1] (decodificación aleatoria). El procedimiento *Construir* crea una lista aleatoria L de ítems de manera tal que el orden de los ítems se corresponde al orden (aleatorio) del archivo de entrada.

- ✓ A_d [2] (decodificación voraz). El procedimiento *Construir* crea una lista aleatoria L de ítems en orden decreciente de p_i / w_i (cociente beneficio-peso). La decodificación del vector binario se hace en base a esta secuencia ordenada.

- ✓ A_d [3] (variante de A_d [2]). El procedimiento *Construir* crea una lista aleatoria L de ítems en orden decreciente de p_i (beneficio).

3. DESCRIPCIÓN DEL EXPERIMENTO

Para el experimento, se ha optado por la codificación binaria (un 1 en la posición *i* significa que el ítem *i* es seleccionado para la mochila). Los experimentos han tomado siempre los mismos 100 ítems, cuyos valores de w_i (peso) y p_i (beneficio) fueron generados de forma aleatoria (*no correlacionada*), mientras que la capacidad de la mochila elegida fue del tipo *capacidad promedio* ($C=0.5* \Sigma_{i=1..n} W[i]=600.5$). Para cada experimento se han realizado 19 ejecuciones, con una población inicial aleatoria con tamaño fijo de 100 individuos, usando selección proporcional, elitismo, crossover de un punto (probabilidad 0.65) y mutación bit a bit (para cada experimento se ha realizado una ejecución con probabilidad 0.05 y otra con probabilidad 0.001), mientras que el número máximo de generaciones es de 1000. En todos los casos se ha tomado como criterio de terminación la estabilidad del fitness medio.

Para todos los casos en donde se aplicó penalidad fue considerado $\rho=25$.

Respecto a la implementación de la función de penalidad definida por Olsen, además de la versión original, se implementaron dos variantes de la misma. Con estas variantes se intenta analizar el comportamiento del algoritmo si se disminuye la penalidad de los individuos no factibles con un valor constante, sin alterar la penalidad de los individuos factibles.

Las variantes son:

1. si el individuo era no factible, se obtenía la función Pen'(x)=Pen(x)-0.005 y se obtenía el *fitness* como $Eval(x)=(\Sigma_{i=1..n} x[i]*P[i]) * Pen'(x)$.
2. si el individuo era no factible, se obtenía la Pen'(x)=Pen(x)-0.01 y se obtenía el *fitness* como $Eval(x)=(\Sigma_{i=1..n} x[i]*P[i]) * Pen'(x)$.

En cuanto a la implementación de los algoritmos reparadores, un aspecto importante a considerar es en qué porcentaje los individuos reparados reemplazarán a los individuos originales de la población. Luego de analizar las propuestas de Liepins y Orvosh, y con el fin de corroborar sus resultados, fueron implementadas tres versiones:

- ✓ Nunca se substituyen los individuos originales por los reparados [10] [11].
- ✓ Se substituye solo el 5% de los individuos originales por los reparados [17].
- ✓ Se substituye el 100% de los individuos originales por los reparados [15].

Respecto a los algoritmos decodificadores, se implementaron las tres variantes propuestas.

En la tabla 1 se describen todos los experimentos realizados y se indica la nomenclatura que se usa para la identificación de cada uno de ellos:

Descripción		P_m=0.05	P_m=0.001
Death Penalty		dp5	dp1
Penalidad	Lineal	pli5	pli1
	Cuadrática	pcu5	pcu1
	Logarítmica	plo5	plo1
Penalidad de Olsen	Versión Original	poa5	poa1
	Indiv. No Factibles Pen'(x)=Pen(x)-0.005	pob5	pob1
	Indiv. No Factibles Pen'(x)=Pen(x)-0.01	poc5	poc1
Algoritmo de Reempl. Aleatoria	Ningún reemplazo	raa5	raa1
	5% de individuos reparados reemplazan a orig.	rab5	rab1
	100% de individuos reparados reemplaza a orig.	rac5	rac1
Algoritmo de Reempl. Voraz	Ningún reemplazo	rva5	rva1
	5% de individuos reparados reemplazan a orig.	rvb5	rvb1
	100% de individuos reparados reemplaza a orig.	rvc5	rvc1
Algoritmo de Decodific.	Lista aleatoria de ítems	dea5	dea1
	Lista de ítems en orden decreciente de p_i / w_i	deb5	deb1
	Lista de ítems en orden decreciente de p_i	dec5	dec1

Tabla 1: Nomenclatura de los experimentos de acuerdo a la probabilidad de mutacion utilizada.

MEDIDAS DE LA PERFORMANCE

Como una indicación de la performance de los algoritmos se tomaron las siguientes variables relevantes:

Ebest=((*opt_val*-mejor valor)/*opt_val*)*100
Es el porcentaje de error del mejor individuo encontrado comparado con el mejor valor conocido o estimado (*opt_val*). Esto da una medida de cuánto se alejan los mejores individuos encontrados de *opt_val.*

Epop=((*opt_val*- fitness promedio poblacional)/*opt_val*)*100
Es el porcentaje de error de la media de la población comparada con el mejor valor conocido o estimado (*opt_val*). Esto da una medida de cuan lejos está el fitness medio del *opt_val*.

Individuos inválidos
Cuenta el número de individuos ilegales que han sido dados como solución óptima en la ejecución de un experimento.

4. RESULTADOS

Los resultados de los experimentos han sido divididos en dos partes, la primera se refiere a todos los experimentos en los que la probabilidad de mutación utilizada ha sido 0.05, mientras que la segunda parte se refiere a experimentos con una probabilidad de mutación del 0.001.

4.1. EXPERIMENTOS CON PROBABILIDAD DE MUTACIÓN 0.05

Analizando los resultados resumidos en la Tabla 2 podemos observar que el valor de *mejor fitness* de individuo válido fue encontrado usando el algoritmo de reparación aleatoria con la

variante de remplazo del 100% de los individuos (rac5), si bien se obtienen mejores *fitness* para algunos otros experimentos la totalidad, o más de la mitad, de esos individuos corresponden a soluciones inválidas (plo5, poa5, pob5, rva5, rvb5). Los valores más bajos fueron obtenidos en las tres variantes de la implementación del algoritmo decodificador (dea5,deb5,dec5), siendo la diferencia muy significativa con respecto a los restantes experimentos.

Para evaluar el *Ebest*, la primera consideración a tener en cuenta es que para el cálculo del *Ebest,* el parámetro *opt_val* tomado es el *mejor fitness* encontrado en un individuo legal (1178). De ahí surgen los valores negativos que se observan en el caso de la implementación de *penalidad logarítmica* y las variantes de *reparación voraz* en las que no se reemplaza la totalidad de las reparaciones realizadas en la población original.

Si sólo se consideran los experimentos realizados en donde la cantidad de individuos inválidos es nula, se observa claramente que el error promedio más grande se registró con el algoritmo deb5 y el menor con el algoritmo rac5 y que el *Ebest* promedio (sin considerar los resultados obtenidos usando algoritmos decodificadores) está entre el 6.63% y el 12.52%.

Al observar el error poblacional *Epop* se nota que, en casi la totalidad de los experimentos realizados exceptuando aquellos que usan algoritmos decodificadores, la magnitud del error promedio está en el orden del 20,48% y el 42,81%, lo cual nos indica que la población mantiene diversidad genética.

Si bien se observan valores de *Epop* promedio inferiores al 32% (instancias plo5, rva5 y rvb5), ellos corresponden a los experimentos en que la totalidad de soluciones obtenidos superan la capacidad de la mochila. En los tres experimentos realizados con algoritmos decodificadores, el *Epop* promedio duplicó el valor de error obtenido en los otros experimentos.

	Fitness		Cant. Indiv. Inválidos	Ebest Promedio	Epop Promedio
	Mejor	Promedio			
dp5	1118.00	673.673684	0	7.4479492	42.8120811
pli5	1121.00	711.845789	0	7.9885622	39.5716647
pcu5	1126.00	692.620526	0	7.3630596	41.2036905
plo5	1434.47	936.694698	19	-19.1015887	20.4843211
poa5	1138.05	784.296776	14	6.1209901	33.4213263
pob5	1115.46	782.370050	12	6.4739523	33.5848854
poc5	1112.21	772.815653	2	8.5425789	34.3959548
raa5	1131.00	799.449474	10	7.4166741	32.1350192
rab5	1125.00	796.786842	6	6.7062818	32.3610491
rac5	1150.00	774.264737	0	6.6347958	34.2729425
rva5	1235.00	815.637368	19	-3.1409168	30.7608346
rvb5	1256.00	810.177368	19	-2.6941292	31.2243321
rvc5	1040.00	747.063158	0	12.5234563	36.5820749
dea5	728.00	366.763684	0	42.1856849	68.8655616
deb5	713.00	359.854211	0	45.3578769	69.4521044
dec5	751.00	369.654737	0	41.7612367	68.6201412

Tabla 2: Resultados obtenidos en los experimentos realizados con P_m=0.05

4.2. EXPERIMENTOS CON PROBABILIDAD DE MUTACIÓN 0.001

Analizando los resultados resumidos en la Tabla 3 podemos observar un comportamiento de los algoritmos similar al evaluado en la Tabla 2. Sólo resta destacar que con esta probabilidad de

mutación se logra una mejor calidad de resultados, lo cual se puede observar en los mayores valores de *mejor fitness* y *fitness promedio*. Consecuentemente, los valores de *Ebest* y *Epop* disminuyen en un orden del 60% respecto a los valores observados en los experimentos con p_m=0.05.

	Fitness		Cant. Indiv. Inválidos	Ebest Promedio	Epop Promedio
	Mejor	Promedio			
dp1	1166.00	1046.224737	0	2.4037173	11.1864
pli1	1172.00	1063.135263	0	1.9345903	9.7508
pcu1	1163.00	1049.784211	0	2.4349924	10.8842
plo1	1471.13	1401.063537	19	-24.8044962	-18.9358
poa1	1176.02	1093.251977	16	0.9427218	7.1942
pob1	1170.02	1084.051656	3	1.3492985	7.9752
poc1	1170.02	1075.584993	0	1.7648110	8.6940
raa1	1178.00	1106.240526	3	0.5852918	6.0916
rab1	1177.00	1088.052105	5	0.9605933	7.6356
rac1	1171.00	1055.929474	0	1.8005540	10.3625
rva1	1281.00	1219.262632	19	-8.6408721	-3.5028
rvb1	1249.00	1074.902632	19	-4.4455366	8.7519
rvc1	1042.00	992.021053	0	11.5539273	15.7877
dea1	858.00	545.322105	0	30.2609240	53.7078
deb1	842.00	454.400000	0	37.9814136	61.4261
dec1	863.00	540.198947	0	30.1626307	54.1427

Tabla 3: Resultados obtenidos en los experimentos realizados con P_m=0.001

Observando el Gráfico 1 podemos apreciar que, independientemente de la probabilidad de mutación utilizada, los resultados obtenidos para los distintos experimentos realizados son similares, lo cual se ve por la similitud de las curvas descriptas. Sin embargo, notamos que cuando se utiliza P_m=0.001 se logra una mejor calidad de resultados, en este caso dado por un mejor aprovechamiento de la capacidad máxima de la mochila.

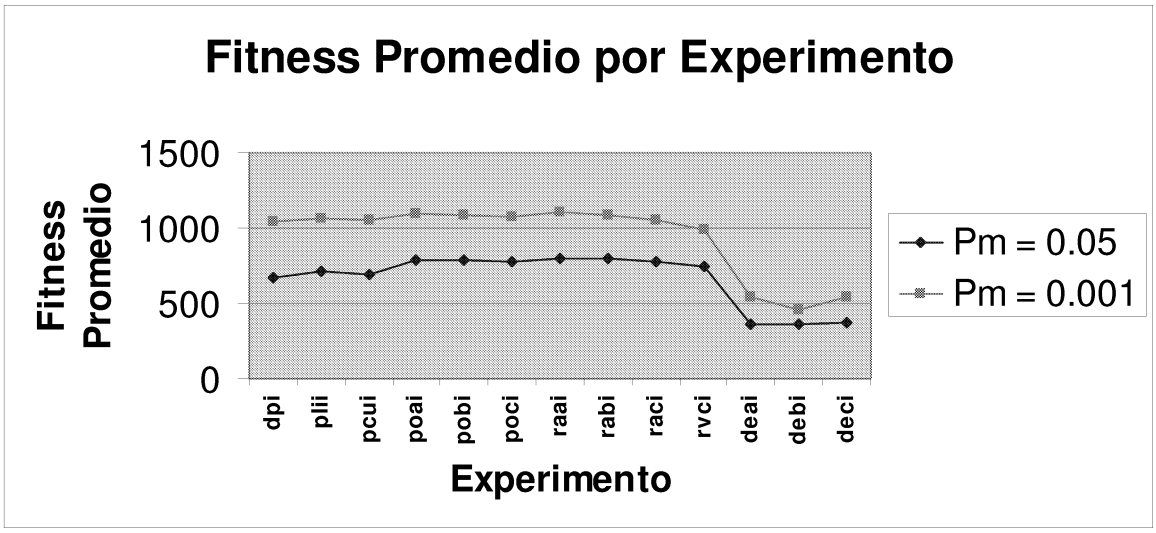

Grafico 1: Fitness Promedio por Experimento para cada una de las probabilidades de mutación evaluadas (sólo se consideran los experimentos cuyas soluciones son válidas).

5. CONCLUSIONES

Si bien no podemos ser concluyentes, a la luz de los resultados alcanzados e independientemente de cual de los dos valores de probabilidad de mutación se use, podríamos indicar que:

✓ el uso de algoritmos decodificadores para este problema no aporta buenos resultados y que mejores resultados pueden ser obtenidos utilizando algoritmos que penalicen a los individuos inválidos o bien los reparen.

✓ la función de penalidad logarítmica no logra penalizar adecuadamente a los individuos inválidos para evitar su recombinación, lo cual surge al analizar la cantidad de individuos inválidos que se obtienen como mejores soluciones al utilizar esta función de penalidad.

✓ con respecto a la penalidad de Olsen, las mejores soluciones se encontraron cuando la penalidad de los individuos inválidos era decrementada en un 0.01, principalmente a la hora de analizar la cantidad de individuos inválidos y la similitud de fitness con las otras variantes.

✓ al implementar algoritmos de reparación aleatoria, tal como lo indica Michalewicz, hemos observado que no es necesario realizar la reparación del 100% de los individuos inválidos para obtener un buen porcentaje de soluciones alternativas que sean de buena calidad. Con sólo reparar el 5% se obtienen buenos resultados y a un costo computacional mucho menor.

✓ Al implementar algoritmos de reparación voraz detectamos que es necesario realizar el 100% de las reparaciones detectadas para alcanzar buenas soluciones.

Independientemente de la implementación considerada, podemos afirmar que los Algoritmos Evolutivos deben ser considerados como una buena alternativa a la hora de buscar un plan de carga adecuado para una mochila, ya que se logra a través de su uso, una carga adecuada y un *pool* de soluciones posibles. En nuestro análisis se ha observado una mejor *performance* al utilizar probabilidad de mutación igual a 0.001.

6. RECONOCIMIENTOS

Nosotros reconocemos la cooperación de los integrantes del grupo del proyecto por aportar nuevas ideas y críticas constructivas. También a la Universidad Nacional de La Pampa y la Universidad Nacional de San Luis de las cuales recibimos un continuo apoyo.

7. BIBLIOGRAFÍA

[1] Alfonso H., Bertone E., Minetti G., Gallard R. *Comparación de Tres Métodos De Búsqueda Incompletos para El Problema de La Mochila*. Anales WICC 2000 (Workshop de Investigadores en Ciencias de la Computación), Universidad Nacional de La Plata, Argentina, 2000.

[2] Bäck , T., *Evolutionary Algorithms in Theory and Practice,* Oxford University Press, New York, NY, 1996

[3] Belew, R. and L. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms,* Morgan Kaufmann Publishers, San Mateo, CA, 1991.

[4] Davis, L., editor, *Handbook of Genetic Algorithms*, Van Nostrand Reinhold, New York, 1991.

[5] Fogel, D., editor, *Proceedings of the First IEEE Conference on Evolutionary Computation*, IEEE Press, Orlando, FL, 1994.

[6] Gen, M. and Cheng, R., *Genetic Algorithms & En*gineering Design, Wiley-Interscience, New York, 1997.

[7] Goldberg, D., *Genetic Algorithms in Search, Optimization and Machine Learning,* Addison-Wesley, Reading, MA, 1989.

[8] Ilog, Inc., *Optimization Technology White Paper: A comparative sutdy of optimization techniques*, www.ilog.com

[9] Liepins, G.E. y Hilliard, M.R., *Greedy Genetics*, in Proceedings Second International Conference on Genetic Algorithms and their Applications, pp. 90-99, 1987.

[10] Liepins, G., M. Hilliard, J. Richardson, and M. Pallmer, *Genetic algorithm application to set covering and traveling salesman problems*, in Brown, editor, OR/AI: *The Integration of Problem Solving Strategies*, 1990.

[11] Liepins, G. and W. Potter, *A genetic algorithm approach to multiple fault diagnosis,* in Davis [4], pp 237-250.

[12] Martello, S. and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley & Sons, Chichester, Uk, 1990.

[13] Michalewicz, Z., *Genetic Algorithm + Data Structure = Evolution Programs*, 3ª ed., Springer-Verlag, New York, 1996.

[14] Michalewicz, Z., Dasgupta, D., Le Riche, R., Schonauer, M., *Evolutionary algorithms for constrained engineering problems.*

[15] Nakano, R. and T. Yamada, *Conventional genetic algorithms for job-shop problems*, in Belew and Booker [3], pp 477-479.

[16] Olsen, A., *Penalty Functions and the Knapsack Problem*, in Fogel [5], pp. 554-558.

[17] Orvosh, D. and L. Davis, *Using a genetic algorithm to optimize problems with feasibility constraint*, in Fogel [5], pp. 548-552.

[18] Pérez Serrada, A., *Una Introducción a la Computación Evolutiva.* Marzo 1996.