Modelado y Resolución de Problema de Planificación de la Producción con Incertidumbre en Demanda

Informe de Proyecto de Grado presentado al Tribunal Evaluador como requisito de graduación de la carrera Ingeniería en Computación

Tutor
Carlos Testuri

Estudiantes
Gonzalo Alcalde
Nelson Connio

Facultad de Ingeniería –UDELAR

Año 2014

Resumen

Este trabajo trata sobre un problema de planificación de la producción en la industria de elaboración por lotes. Se plantea un problema donde dado un único equipamiento de elaboración de uso exclusivo, un conjunto de productos, una cantidad de períodos y una demanda cierta y conocida de cada producto para cada período, se busca una planificación de producción de cada producto para todos los períodos, tal que minimiza costos de elaboración, envasado, almacenamiento, demanda insatisfecha y restablecimiento de equipamiento, sujetos a restricciones de capacidad, reparto de equipamiento y balance de demanda entre otras. Luego se plantea un problema similar para el caso extendido en que la demanda sea incierta generando una cantidad de escenarios posibles cada uno con una demanda posible para cada producto y cada período, con lo cual se busca una planificación que satisfaga dichas demandas para todos los escenarios. Se modelan algebraicamente e informativamente ambos problemas y se implementan dichos modelos utilizando una herramienta de resolución de problemas de optimización (GLPK) y también utilizando metaheurísticas, a partir del desarrollo de un algoritmo genético a través de la librería Mallba. Dichas implementaciones son luego probadas utilizando un conjunto de pruebas y se exhiben sus resultados. Se concluye que los resultados son diferentes según la implementación, dado que una se enfoca en buscar la solución óptima independientemente del tiempo consumido y la otra se enfoca en buscar una solución factible en un cierto plazo de tiempo.

(Planificación de la Producción, Algoritmos Genéticos, Programación Entera Mixta, Programación Estocástica, Optimización)

Tabla de Contenidos

Capítulo 1: Introducción

1.1: Definición del problema original

El problema trata la planificación de un proceso productivo de alimentos por lotes para atender la demanda durante un horizonte de tiempo discretizado en períodos. El proceso productivo consiste de las etapas de elaboración de productos a partir de los ingredientes, y de envasado de los productos. Las etapas tienen lugar en todos los períodos y cada instancia de una etapa abarca todo el período en que tiene lugar.

A partir de la mezcla de dos ingredientes, se elaboran dos productos. Los productos se envasan, cada uno, en único empaque. Mientras que para la etapa de envasado existe equipamiento independiente para cada producto; para la etapa de elaboración se comparte equipamiento, que debe restablecerse (limpiarse, etc) cada vez que se cambie de producto a elaborar.

En ambas etapas es relevante el uso del equipamiento; en particular la de elaboración, cuyo equipamiento es compartido por ambos productos y debe restablecerse al intercambiarse la producción de estos. En cada período y para cada producto solo existe una única instancia de elaboración y envasado.

Para la elaboración de cada unidad de los productos se utilizan los dos ingredientes de acuerdo a un a formulación establecida por tasas por unidad de los ingredientes.

El uso de equipamiento se mide en tiempo (ej. horas) y se cuenta con tasas de tiempo consumido por unidad de cada producto elaborado y envasado (únicas para todo período). Además se cuenta con el tiempo consumido al restablecerse el equipamiento de elaboración para cada producto (único para todo período). Por otra parte, para cada equipo se dispone del tiempo total de capacidad de operación (único para todo período).

Se deben mantener inventarios mínimos de productos para cada período y se parte de un inventario inicial.

Se cuenta con los costos de elaboración, almacenamiento y envasado de cada producto por período, y el costo de restablecimiento del equipamiento de elaboración por producto.

La demanda es incierta, y se representa para cada período con escenarios discretos independientes con sus correspondientes probabilidades. Esto establece para el horizonte de períodos un árbol de escenarios finales con probabilidades asociadas.

El objetivo es minimizar los costos esperados mientras se atiende la demanda para todos los escenarios y se cumplen las restricciones de disponibilidad de equipamiento.

1.2: Organización del documento

El capítulo 2 explica lo que se ha escrito sobre el tema, además de la definición de unos cuantos términos que fundamentan este trabajo. El capítulo 3 muestra el modelo algebraico usado en este proyecto, de forma matemática, tanto para su variante determinística como para su variante estocástica. El capítulo 4 muestra cómo se implementaron esos modelos en el programa GLPK y en la librería Mallba de algoritmos genéticos, además de en este último caso algunas decisiones adicionales para mejorar los resultados. En el capítulo 5 se presentan pruebas hechas para probar las implementaciones y también sus resultados. Y finalmente en el capítulo 6 planteamos nuestras conclusiones. El resto del documento es un anexo con el código fuente de las implementaciones.

Capítulo 2: Estado del Arte y Marco Teórico

2.1: Estado del Arte

Algunos investigadores han trabajado sobre el problema de optimización de lotes de producción para varios períodos y varios productos con una demanda determinada. Un ejemplo es el trabajo de Belvaux y Wolsey[1] que presenta varios modelos matemáticos para un modelo determinístico: un modelo básico, otro pensado para el uso de sistemas que soportan el método de resolución branch and cut y otro para problemas de multinivel (en donde cada producto necesita de una unidad de los producto predecesores).

El modelo básico de Belvaux y Wolsey considera el problema de producción en lotes de varios productos que pueden ser producidos en varias máquinas en un horizonte de *varios* períodos. Dentro de los datos considerados se tienen:

1. demanda de un producto en un determinado período.
2. tasa de producción de un producto en una determinada máquina.
3. máxima cantidad de un producto que puede ser producido en determinada máquina en cierto período.
4. capacidad total de determinada máquina en cierto período.
5. capacidad requerida por un producto en determinada máquina y en cierto período.

Restaría agregar los costos relevantes en la producción y preparación de las máquinas.
Las variables relevantes son:

1. stock de un producto en cierto período.
2. backlog de un producto en cierto período.
3. cantidad a producir de un producto en determinada máquina y cierto período.
4. variable binaria que indica si cierta máquina está disponible en un determinado período, también llamada variable set-up.
5. variable start-up que indica que determinada máquina fue preparada para determinado producto en un período previo.
6. variable switch-off que indica que cierta máquina fue preparada para un determinado producto en el período posterior.

Dentro de las principales restricciones se tienen:

1. ecuación de balance de flujo: mantiene la relación entre stock, demanda y cantidad a producir.
2. restricciones de corte combinando variables y datos mencionados anteriormente.
3. posibles restricciones para mantener stock de seguridad.

Es importante para definir el modelo considerar el tiempo del intervalo. Si el intervalo es largo, se denomina big bucket model, en caso contrario el modelo se denomina small bucket model. En los modelos big bucket las producciones están limitadas a uno o dos productos por período. La elección del modelo adecuado depende del problema, teniendo en cuenta que producir dos

productos por período puede llevar a incrementar el tamaño del período y esto llevaría a una reducción del número de períodos para que la producción cubra la demanda.
En este tipo de modelos lo importante es encontrar una ajustada formulación para start-up, switch-off y changeovers. En la bibliografía se estudian por separado las posibilidades de establecer un set-up por período o en su defecto dos.
En el modelo Big Bucket with Changeovers tenemos el caso en que varios productos pueden ser programados en cada período contemplando costos por cambios de producto y limpieza. Un ejemplo de este modelo fue aplicado para solucionar el problema de ruteo de un vehículo

Es importante según Belvaux y Wolsey mantener restricciones que relacionen las diferentes capacidades de producción para un producto en un período cuando tenemos un cambio, la mínima cantidad a producir de un producto y la capacidad total. En caso que la producción entre un par de períodos sea mayor que la demanda, estas producciones serán utilizadas en las sucesivas demandas. Si debemos de producir a la máxima capacidad entonces debemos de tener en cuenta las restricciones de set-up y start-up para la máquina.

El modelo de Belvaux y Wolsey fue utilizado para resolver ciertos problemas, los cuales describiremos brevemente:

b4
Es un problema con varios productos, backlogging, tiempos de limpieza y cantidades mínimas y máximas de stock. Un modelo small bucket con dos set-up por período como máximo.
La característica especial del problema es el límite inferior(medido en días) de cada corrida para cada producción de un producto y la completa capacidad de producción en todos los períodos menos el primero y último. Tenemos un pequeño número de productos que son llamados intermedios, ya que son utilizados para generar productos finales en máquinas especiales para manejar estos productos. Se conoce las cantidades necesarias de productos intermedios para generar productos finales.

Chesapeake
Los problemas CHES son un conjunto de cinco problemas que involucran asignación y secuencias de las diferentes operaciones de producción dependiente de costos de set-up.
Los resultados se obtienen en primera instancia mediante el método branch and bound, cuando se encuentra una posible solución entera, se chequea si representa una secuencia de máquina de producción factible, si no lo es, se agregan restricciones para eliminar esa posible solución y se continúa con la búsqueda en el árbol. Es fácilmente implementable en sistemas como MINTO, o con librerías como CPLEX o XPRESS. Una mejora para este enfoque podría ser generar restricciones para eliminar soluciones fraccionarias.

SimpEreng
En Simpson y Erenguc [2] proponen un modelo general para problemas multinivel con ensamblado de productos, incluye familias de productos que consisten en varios productos donde cada familia puede tener un costo fijo o una restricción de recurso asociada.

L1

Este problema involucra restricciones de capacidad, ciertos niveles de stock, backlogging, tamaño mínimo de cada corrida y producción de productos pertenecientes a una familia durante un período y máquina de producción en particular. Se tienen costos por backlogging, mantener stock y tener un stock menor al mínimo de seguridad.

En el trabajo de Brown et al.[3] en la que se estudia la planta de producción Kellogg, la cual se dedica a la producción de cereales para el desayuno y otros alimentos. Desde hace muchos años la empresa utilizaba hojas de cálculo y un software especial para la planeación de materias primas(PMP) y recursos de distribución, pero desde 1987 la empresa entendió que necesitaban coordinar globalmente y sistematizar sus procesos.Luego de un año de desarrollo, se obtuvo un sistema prototipo. La primera versión fue instalada en 1990 y desarrollada paulatinamente por muchos años, esto ayudó a que la transición hacia el sistema actual mucho más centralizado que lo que era inicialmente. Se operan varias plantas en varios países, por lo que es muy importante la coordinación entre todas las plantas para llegar a minimizar costos. Se tiene en cuenta que no todos los productos son producidos en todas las plantas, algunos productos pueden ser envasados en ciertas plantas, y que es necesario coordinar producciones en líneas que son compartidas por varios productos. También se ensamblan productos intermedios y transporte. Algunas decisiones son independientes unas de otras, pero otras son muy dependientes y es lo que lleva a determinar las restricciones. La versión operacional del sistema trabaja sobre producción, empaque, inventarios y distribución con un horizonte de 30 semanas, pero se ejecuta cada domingo en la mañana. La ecuación de balance relaciona stock, producción y demanda. Dentro de las variables determinantes tenemos:

1. cantidad a producir en determinada línea de producción, planta y semana.
2. cantidad a empacar de un producto, en una línea de empaque, en cierta planta y semana.
3. stock de un producto, en una planta y semana.
4. cantidad a enviar de un producto, desde una planta, hacia otra planta, en cierta semana.

Dentro de las principales restricciones aplicadas semanalmente en cada planta tenemos:

1. No se excedan las capacidades de las líneas de producción.
2. No se excedan las capacidades de las líneas de empaque.
3. se envasan todos los productos producidos en un período(balance entre producción y empaque).
4. ecuación de balance entre cantidades de cada producto que ingresan y salen de stock.
5. satisfacer stock de seguridad
6. coordinación entre líneas de procesamiento y líneas de empaque

Las restricciones 1, 2, 4 y 5 son implementadas como de objetivos elásticos que pueden ser violados pero serán penalizados por cada unidad, lo que se trata de representar es la ocurrencia de cuellos de botella o sobretiempos.

Siendo KPS un modelo determinístico maneja aceptablemente la demanda incierta en las primeras dos semanas, pero luego de la tercera o cuarta los resultados ya no son tan aceptables, tomando en cuenta la cantidad de horas que le lleva a KPS llegar a una solución,

los autores se plantearon sino era el momento de cambiar para un modelo multinivel de programación estocástica.

Por último, para la parte de la resolución del problema usando algoritmos genéticos vale mencionar el trabajo de Yoshitomi et al[4] donde se explica cómo utilizar los algoritmos genéticos (GA) en entorno incierto (GAUCE) para resolver diferentes problemas de optimización estocástica. En primer lugar se resolvió el *problema de Asignación estocástica óptima* y *problema de la mochila.* Luego para chequear que el método es óptimo se resolvió el *problema de compresión estocástica de imágenes.*
En GA, la función fitness es aplicada a cada individuo del entorno, en GAUCE, también es aplicada a cada individuo pero para cada generación la función fitness varía ya que también lo hacen las variables estocásticas(según su función de distribución estocástica). Luego, la mejor solución es aquella solución con mayor frecuencia de ocurrencia entre todas las generaciones.

Asignación estocástica óptima
Este problema puede ser resuelto como un simple problema de programación entera determinística, sin embargo facilita la comparación entre las soluciones exactas y el método de algoritmo genético propuesto. La representación común del genotipo es utilizado para evitar la formación de un gen fatal, se utiliza también, un punto de cruzamiento y un punto de mutación. La función objetivo es utilizada como función fitness. El punto de cruce entre dos individuos padres se selecciona aleatoriamente, en ese punto el cromosoma es dividido en dos partes y se cruzan obteniendo dos individuos hijos. El punto de mutación es elegido probabilísticamente. El gen mutante es elegido aleatoriamente. El experimento con una población de tamaño 100, tamaño de generación es de 1500, probabilidad de cruzamiento 0.6 y probabilidad de mutación de 0.05. Con estos parámetros los individuos con más alta frecuencia de solución obtenido por el algoritmo genético son los mismos que los obtenidos con Optimal Assignment Problem aplicando el métodos de branch and bound.
Las pruebas ejecutadas con GAUCE no fueron del todo satisfactorias o realistas, pero en todos los casos la disposición del ranking fue útil para encontrar la solución más adecuada.
Para comparar la eficacia del método de algoritmo genético se compara el mismo contra el valor de la función fitness, en cambio el algoritmo GAUCE es evaluado comparando la solución con más frecuencia contra el valor esperado. GAUCE alcanza el 100% de exactitud en la generación 50, mientras que GA alcanza el 100% de exactitud en la generación 20. Cuando tenemos variables representadas por funciones de distribución estocásticas, el método GA ejecuta un número infinito de casos sin llegar a una solución, por lo que concluyen los autores que no es apropiado para resolver problemas de programación estocástica. Por otro lado, GAUCE si encuentra solución ejecutándolo varias veces y verificando contra el ranking de soluciones.

Problema de la mochila estocástico
En esta prueba los autores utilizaron dos puntos de cruzamiento para un par de individuos padres con cierta probabilidad, los puntos de división son seleccionados aleatoriamente y el cromosoma es dividido en tres partes. Luego,cada parte es combinada con otra de otro padre para obtener un par de individuos hijos. Un punto de mutación es ejecutado con cierta

probabilidad, el gen mutado es seleccionado aleatoriamente. El alelo de dicho gen se cambia por otro. Los datos de esta prueba fueron:
población de 500, tamaño de generación de 1500, probabilidad de cruzamiento 0.6, probabilidad de mutación de 0.1. En estas condiciones los individuos con más alta frecuencia a través de todas las generaciones obtenidas mediante el algoritmo genético coincide con la solución óptima obtenida mediante el método de branch and bound.
Con la técnica GAUCE fueron necesarias 5 ejecuciones para obtener los mismos resultados.

Problema de compresión de imagen estocástica
Cuando una sola imagen es almacenada no es tan importante la compresión de la misma, pero cuando el volumen o cantidad de imágenes se incrementa la compresión se hace muy necesaria, también es muy útil y utilizado en el ensamblado de imágenes. Los pasos para codificar son: Discrete Cosine Transform(8 x 8 pixel), CDT de aquí en más, Quantization y Entropy Coding. Por otro lado, la decodificación es de la siguiente manera: Entropy Coding, Quantization Inverse e Inverse DCT. Debido a que quantization es una aproximación, tanto el codificado como el decodificado no es 100% reversible. El porcentaje de compresión se calcula como volumen codificado / volumen original.
En el problema resuelto por GAUCE los coeficientes de DCT son determinados aleatoriamente. El error en decodificar una imagen y el porcentaje de compresión son calculados para cada individuo.La función fitness utilizada refleja el valor inverso del error entre la imagen original y la decodificada cuando la restricción de compresión es satisfecha, en caso contrario el valor fitness es cero.
El tamaño de población es de 200, tamaño de generación de 100, probabilidad de cruzamiento 0.6 y probabilidad de mutación de 0.1. Se utilizaron 100 imágenes de TV con 160 x 120 pixels, estas imágenes fueron transformadas en imágenes monocromáticas.Las 100 imágenes fueron almacenadas a una tasa de 1 imagen cada 0.1 seg. El promedio y la desviación estándar de los coeficientes DCT para cada bloque de 8x8 pixels son medidos para determinar la distribución normal para cada DCT coeficiente. GAUCE presenta la misma solución que ofrece la solución con más alta frecuencia de ocurrencia. En este caso la solución óptima no fue derivada del método existente, pero dado que GAUCE es aplicado al problema de asignación estocástica de manera exitosa, se considera que GAUCE tiene la habilidad para seleccionar la solución óptima o en el peor caso el óptimo local.

Según los autores, GAUCE tiene un potencial muy importante para encontrar la segunda, tercera, etc según un ranking de soluciones según el valor de la función objetivo. Este ranking es muy útil en casos que la solución óptima no puede aplicarse por algún motivo.

2.2: Marco Teórico

A pesar de que la mayoría de los términos que se utilizan en este informe son de fácil comprensión, hay algunos que merecen una mejor explicación para mejor entendimiento.

Programación Lineal

Es un procedimiento o algoritmo matemático mediante el cual se resuelve un problema formulado a través de un sistema de inecuaciones lineales, optimizando la función objetivo, también lineal. Es decir, consiste en optimizar (minimizar o maximizar) una función lineal, denominada función objetivo, de tal forma que las variables de dicha función estén sujetas a una serie de restricciones que expresamos mediante un sistema de inecuaciones lineales.[5]

Programación entera
Es un caso particular de la programación lineal en donde se requiere que la solución óptima se componga de valores enteros para algunas de las variables. La resolución de este problema se obtiene analizando las posibles alternativas de valores enteros de esas variables en un entorno alrededor de la solución obtenida considerando las variables reales. Muchas veces la solución del programa lineal truncado esta lejos de ser el óptimo entero, por lo que se hace necesario usar algún algoritmo para hallar esta solución de forma exacta. El más famoso es el método Branch and Bound.[6]

Optimización estocástica
En un problema de optimización en donde se tiene que optimizar una función objetivo con sus variables sujetas a restricciones, la optimización estocástica es el caso en el que los parámetros o las restricciones cambian aleatoriamente. Esto hace que surjan diferentes escenarios para resolver y que la solución encontrada tenga que cubrir todos los escenarios posibles. Existen varias metodologías para resolver problemas de optimización estocástica, pero estas aumentan su complejidad según la cantidad de escenarios posibles.

Árbol de escenarios y Principio de no anticipatividad
En los problemas de optimización estocástica, se puede representar la estocasticidad usando un árbol de escenarios. Un árbol como el de la figura 1 representa los diferentes valores de los parámetros y las decisiones anticipativas en el tiempo. En este caso en particular un árbol con tres períodos y 4 escenarios posibles en total (2 posibilidades por período con el primer período como raíz). Cada escenario, cada rama del árbol (1-2-4, 1-2-5, 1-3-6 y 1-3-7) muestra la hipotética situación de cada parámetro del sistema para cada período. Las tramas de los escenarios que coinciden (los nodos 1,2 y 3) representan la misma situación de incertidumbre (el nodo 1 para todos los escenarios, el nodo 2 para los escenarios 1-2-4 y 1-2-5 y el nodo 3 para los escenarios 1-3-6 y 1-3-7), y por tanto deben tomar las mismas decisiones. A esto se llama principio de no-anticipatividad, el cual Wets y Rockafellar [10] explican como

”Si dos escenarios distintos, son idénticos hasta una etapa determinada en el horizonte de tiempo, entonces los valores de las variables de decisión deben ser los mismos hasta esa etapa”.

Este principio garantiza que la solución obtenida del modelo hasta una etapa determinada, no depende de información que aún no ha sido disponible.

Otra forma de representar la información del árbol de escenarios es utilizando una matriz de escenarios como la de la figura 2. El árbol queda convertido en una matriz de 4x3 donde las

tramas que coinciden (las cuales aparecen agrupadas dentro de los rectángulos) aparecen como nodos separados, pero que tienen que representar una misma decisión dentro de sus respectivas agrupaciones. La representación matricial permite la manipulación independiente de los escenarios con respecto a los períodos favoreciendo que la notación algebraica no necesite un número variable de subíndices por escenarios

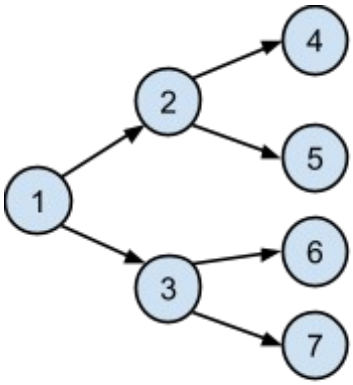

Fig 1. Arbol de escenarios

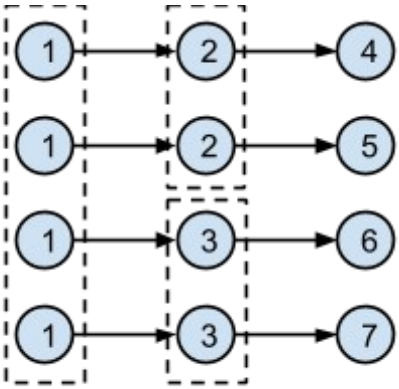

Fig 2. Matriz de escenarios

Metaheurísticas

La resolución de un problema tiene dos objetivos fundamentales que son, encontrar algoritmos con buenos tiempos de ejecución y soluciones usualmente óptimas. Una heurística es un algoritmo que satisface uno de esos objetivos, por ejemplo, normalmente encuentran buenas soluciones, aunque no hay pruebas de que la solución no pueda ser arbitrariamente errónea en algunos casos; o se ejecuta razonablemente rápido, aunque no existe tampoco prueba de que siempre será así. Las heurísticas generalmente son usadas cuando no existe una solución óptima bajo las restricciones dadas (tiempo, espacio, etc.), o cuando no existe del todo.[7]
Una metaheurística es un método heurístico para resolver un tipo de problema computacional general, usando los parámetros dados por el usuario sobre unos procedimientos genéricos y abstractos de una manera que se espera sea eficiente. Normalmente, estos procedimientos son heurísticos. Las metaheurísticas generalmente se aplican a problemas que no tienen un algoritmo o heurística específica que dé una solución satisfactoria, o bien cuando no es posible implementar ese método óptimo.[8]

Algoritmo Genético

Es un caso especial de los llamados algoritmos evolutivos. Los algoritmos evolutivos son métodos de optimización y búsqueda de soluciones basados en los postulados de la evolución biológica. En ellos se mantiene un conjunto de entidades que representan posibles soluciones, las cuales se mezclan, y compiten entre sí, de tal manera que las más aptas son capaces de prevalecer a lo largo del tiempo, evolucionando cada vez hacia mejores soluciones.
En un algoritmo genético existe una población de soluciones, una función de fitness (adaptación), metodologías de selección, cruzamiento y mutación, y consiste en someter a la población de soluciones a acciones aleatorias de cruzamiento y mutación, así como también a

una Selección de acuerdo con algún criterio, en función de su valor de fitness. En cada generación la población que va quedando es la de mayor adaptación. En los Algoritmos Genéticos, las soluciones son representadas como cromosomas que contienen una cantidad de alelos, y cada alelo tiene un valor. En un cruzamiento, se toman dos cromosomas y se intercambian sus alelos en determinado punto (puede ser más de un punto). En una mutación se toma un cromosoma, y uno o varios de sus alelos cambian de valor.[9]

Capítulo 3: Modelado del problema

En este capítulo detallaremos las decisiones que tomamos para obtener la solución de las diferentes secciones.

3.1: Modelo Determinístico

Para el modelo determinístico estaremos asumiendo lo siguiente, de forma de simplificar el diseño:

1) Si bien el planteo original mencionaba sólo dos productos, este modelo contempla más productos.
2) Al inicio del primer período, el inventario inicial es de cero para cada producto.
3) Los costos de los ingredientes al final quedan contemplados dentro de los costos de elaboración.
4) Se descarta el uso de fórmulas de ingredientes dado que no se ha contemplado en la descripción original una cantidad máxima de cada ingrediente y si esa cantidad máxima es igual para cada período o no. Si se hubiese contemplado, solamente sería agregar algunas restricciones adicionales al modelo.
5) Durante un período, el producto elegido para elaborarse será el mismo para el envasado en ese período. Por un lado es como decir que una unidad recién elaborada es inmediatamente envasada. Por otro, es una deducción surgida de que sólo hay una instancia de elaboración y envasado en cada período y que dos productos no pueden compartir la maquinaria de elaboración.

Con estas premisas se pasa a formular el modelo algebraico para el modelo determinístico:

Se definen los conjuntos que representan las entidades del problema.

I = Productos
T = Períodos

Luego se definen los siguientes parámetros:

$Cenv_i^t$ = Costo unitario de envasado del producto *i* en el período *t*.

$Celab_i^t$ = Costo unitario de elaboración del producto *i* en el período *t*.

$Calm_i^t$ = Costo unitario de almacenamiento del producto *i* en el período *t*.

$Cins_i$ = Costo de insuficiencia del producto *i*.

$Cres_i$ = Costo de restablecimiento del equipamiento para el producto *i*.

D_i^t = Demanda del producto *i* en el período *t*.

$Kelab_i$ = Ocupación unitaria de elaboración del producto *i*.
$Kenv_i$ = Ocupación unitaria de envasado del producto *i*.
$KenvM$ = Capacidad de envasado máxima.
$KelabM$ = Capacidad de elaboración máxima.

Y finalmente las variables

X_i^t = Unidades elaboradas del producto *i* en el período *t*. Variable no negativa.
S_i^t = Unidades almacenadas del producto *i* en el período *t*. Variable no negativa.
R_i^t = Unidades atrasadas del producto *i* en el período *t*. Variable no negativa.
Y_i^t = Indica producción del producto *i* en el período *t*. Variable binaria. Y_i^t =1 si X_i^t > 0.
U_i^t = Indica uso del producto *i* en el período *t*. Variable binaria. U_i^t =1 si X_i^t > 0 y X_i^{t-1} = 0.

La formulación del problema queda de la siguiente forma:

$$\min \quad \sum_{i\in I}\sum_{t\in T}\left(\left(Cenv_i^t+Celab_i^t\right)X_i^t+Calm_i^t S_i^t+Cres_i U_i^t+Cins_i R_i^t\right)$$

s.a.

$$\sum_{i\in I} Kenv_i X_i^t \le KenvM\,,\ \forall\, t\in T \qquad (1)$$

$$\sum_{i\in I} Kelab_i X_i^t \le KelabM\,,\ \forall\, t\in T \qquad (2)$$

$$X_i^t+S_i^{t-1}-R_i^{t-1}=D_i^t+S_i^t-R_i^t,\ \forall\, i\in I\,,\ \forall\, t\in T \qquad (3)$$

$$X_i^t \le Y_i^t\sum_{t\in T} D_i^t,\ \forall\, i\in I\,,\ \forall\, t\in T \qquad (4)$$

$$\sum_{i\in I} Y_i^t \le 1,\ \forall\, t\in T \qquad (5)$$

$$U_i^t \ge Y_i^t-Y_i^{t-1},\ \forall\, i\in I\,,\ \forall\, t\in T \qquad (6)$$

$$\sum_{t\in T} X_i^t \ge \sum_{t\in T} D_i^t,\ \forall\, i\in I \qquad (7)$$

$$S_i^0=0,\ \forall\, i\in I \qquad (8)$$

$$Y_i^0=0,\ \forall\, i\in I \qquad (9)$$

$$R_i^0=0,\ \forall\, i\in I \qquad (10)$$

La función objetivo se describe como la minimización de la suma para cada período y cada producto de los costos unitarios de elaboración y envasado por cada unidad producida, más el costo de almacenamiento por cada unidad almacenada, más el costo de restablecimiento por cada vez que se ha cambiado un producto por otro, más el costo de insuficiencia por cada unidad faltante.

A continuación se describen las restricciones al modelo.

1) Las unidades envasadas de un producto en un período son menores que la capacidad máxima de envasado. Esto queda establecido en la ecuación (1).
2) Las unidades elaboradas de un producto en un período son menores que la capacidad máxima de elaboración. Esto queda establecido en la ecuación (2).
3) Para un mismo producto en un período, las unidades producidas más las almacenadas en el período anterior, menos las que quedaron atrasadas en el período anterior es igual a la demanda más las almacenadas en el período, menos las atrasadas en el período. Esto queda establecido en la ecuación (3), también conocida como ecuación de balance de material.
4) Ningún producto *i* se produce en mayor cantidad que la demanda de dicho producto para todos los períodos. Esto queda establecido en la ecuación (4).
5) Sólo un producto se produce por período. Esto queda establecido en la ecuación (5).
6) El cambio de producción del producto *i* en el período *t* se deduce de si el producto *i* se produce en el período *t*, pero no se produjo en el período *t-1.* Esto queda establecido en la ecuación (6).
7) Ningún producto tendrá cantidades atrasadas al final del último período. Esto queda establecido en la ecuación (7).
8) Al inicio, ningún producto tendrá stock almacenado o retrasado, además de que el equipamiento estará sin usar. Esto queda establecido en las ecuaciones (8), (9) y (10).

3.2: Modelo Estocástico

Para el modelo estocástico, la cantidad de posibilidades que pueden surgir al final de cada período es incluida dentro del planteamiento del problema. Si hay *n* posibilidades, se genera un árbol de escenarios *n*-ario completo. La cantidad de escenarios quedará establecida por la cantidad de posibilidades que puedan surgir, elevada a la cantidad de períodos menos uno. Si el modelo tiene 5 períodos y 2 posibilidades por período (demanda alta y demanda baja), habrán 16 escenarios (el primer período es siempre raíz).

$$escenarios = posibilidades^{(periodos-1)}$$

Dado que los escenarios se representan en forma independiente de los períodos, se deben incluir restricciones de no anticipatividad. Para esto, guiado por la idea de la matriz de escenarios, se utiliza una matriz tridimensional. La matriz es una matriz binaria (los únicos valores que pueden tener sus celdas son 0 y 1) que tiene una dimensión escenarios por escenarios por períodos. Dados dos escenarios y un período, la celda correspondiente tiene valor 1 si ambos escenarios son idénticos hasta dicho período. Si no son idénticos, la celda correspondiente tiene valor 0.

Con estas premisas se pasa a formular el modelo algebraico para el modelo estocástico:

Se definen los conjuntos que representan las entidades del problema.

I = Productos
S = Escenarios

T = Períodos

Luego se definen los siguientes parámetros:

$Cenv_i^t$ = Costo unitario de envasado del producto *i* en el período *t*.

$Celab_i^t$ = Costo unitario de elaboración del producto *i* en el período *t*.

$Calm_i^t$ = Costo unitario de almacenamiento del producto *i* en el período *t*.

$Cins_i$ = Costo de insuficiencia del producto *i*.

$Cres_i$ = Costo de restablecimiento del equipamiento para el producto *i*.

$D_i^{t,s}$ = Demanda del producto *i* en el período *t* y el escenario *s*.

$Kelab_i$ = Ocupación unitaria de elaboración del producto *i*.

$Kenv_i$ = Ocupación unitaria de envasado del producto *i*.

$Kenv$ = Capacidad de envasado máxima.

$Kelab$ = Capacidad de elaboración máxima.

$Prob_s$ = Probabilidad asociada al escenario *s*.

$NA_{s,sa}^t$ = Matriz auxiliar para modelar la restricción de no anticipatividad. $NA_{s,sa}^t$ = 1 si en el período *t*, los escenarios *s* y *sa* coinciden en la misma trama. $NA_{s,sa}^t$ = 0 en otro caso.

Y finalmente las variables

$X_i^{t,s}$ = Unidades elaboradas del producto *i* en el período *t* y el escenario *s*. Variable no negativa.

$S_i^{t,s}$ = Unidades almacenadas del producto *i* en el período *t* y el escenario *s*. Variable no negativa.

$R_i^{t,s}$ = Unidades atrasadas del producto *i* en el período *t* y el escenario *s*. Variable no negativa.

$Y_i^{t,s}$ = Indica producción del producto *i* en el período *t* y el escenario *s*. Variable binaria.

$Y_i^{t,s}$ =1 si y sólo si $X_i^{t,s}$ > 0.

$U_i^{t,s}$ = Indica uso del producto *i* en el período *t* y el escenario *s*. Variable binaria. $U_i^{t,s}$ =1 si y sólo si $X_i^{t,s}$ > 0 y $X_i^{t-1,s}$ = 0.

La formulación del problema queda de la siguiente forma:

$$\min \quad \sum_{s\in S}\sum_{i\in I}\sum_{t\in T} Prob_s\left(\left(Cenv_i^t + Celab_i^t\right)X_i^{t,s} + Calm_i^t S_i^{t,s} + Cres_i U_i^{t,s} + Cins_i R_i^{t,s}\right)$$

s.a.

$$\sum_{i\in I} Kenv_i X_i^{t,s} \le KenvM \,, \forall\, t\in T \,, \forall\, s\in S \qquad (1)$$

$$\sum_{i\in I} Kelab_i X_i^{t,s} \le KelabM \,, \forall\, t\in T \,, \forall\, s\in S \qquad (2)$$

$$X_i^{t,s} + S_i^{t-1,s} - R_i^{t-1,s} = D_i^{t,s} + S_i^{t,s} - R_i^{t,s}, \forall i \in I, \forall t \in T, \forall s \in S \quad (3)$$

$$X_i^{t,s} \leq Y_i^{t,s} \sum_{t \in T} D_i^{t,s}, \forall i \in I, \forall t \in T, \forall s \in S \quad (4)$$

$$\sum_{i \in I} Y_i^{t,s} \leq 1, \forall t \in T, \forall s \in S \quad (5)$$

$$U_i^{t,s} \geq Y_i^{t,s} - Y_i^{t-1,s}, \forall i \in I, \forall t \in T, \forall s \in S \quad (6)$$

$$\sum_{t \in T} X_i^{t,s} \geq \sum_{t \in T} D_i^{t,s}, \forall i \in I, \forall s \in S \quad (7)$$

$$S_i^{0,s} = 0, \forall i \in I, \forall s \in S \quad (8)$$

$$Y_i^{0,s} = 0, \forall i \in I, \forall s \in S \quad (9)$$

$$R_i^{0,s} = 0, \forall i \in I, \forall s \in S \quad (10)$$

$$\sum_{sp \in S} NA_{s,sp}^{t} X_i^{t,sp} = \sum_{sp \in S} NA_{s,sp}^{t} X_i^{t,s}, \forall i \in I, \forall t \in T, \forall s \in S \quad (11)$$

La función objetivo se describe como la minimización del promedio ponderado (donde el peso es la probabilidad asociada de cada escenario) de la suma para cada escenario, para cada período y cada producto de los costos unitarios de elaboración y envasado por cada unidad producida en ese escenario, más el costo de almacenamiento por cada unidad almacenada en ese escenario, más el costo de restablecimiento por cada vez que se ha cambiado un producto por otro en ese escenario, más el costo de insuficiencia por cada unidad faltante en ese escenario.

A continuación se enumeran las restricciones al modelo.

1) Las unidades envasadas de un producto en un período son menores que la capacidad máxima de envasado. Esto queda establecido en la ecuación (1).
2) Las unidades elaboradas de un producto en un período son menores que la capacidad máxima de elaboración. Esto queda establecido en la ecuación (2).
3) Para un mismo producto en un período y un escenario, las unidades producidas más las almacenadas en el período anterior, menos las que quedaron atrasadas en el período anterior es igual a la demanda en ese escenario más las almacenadas en el período, menos las atrasadas en el período. Esto queda establecido en la ecuación (3), también conocida como ecuación de balance de material.
4) Ningún producto *i* en un escenario se produce en mayor cantidad que la demanda de dicho producto en ese escenario para todos los períodos. Esto queda establecido en la ecuación (4).
5) Sólo un producto se produce por período y escenario. Esto queda establecido en la ecuación (5).
6) El cambio de producción del producto *i* en el período *t* y el escenario *s* se deduce de si en ese escenario *s*, el producto *i* se produce en el período *t*, pero no se produjo en el período *t-1.* Esto queda establecido en la ecuación (6).
7) Ningún producto tendrá cantidades atrasadas al final del último período. Esto queda establecido en la ecuación (7).
8) Al inicio, ningún producto tendrá stock almacenado o retrasado, además de que el equipamiento estará sin usar. Esto queda establecido en las ecuaciones (8), (9) y (10).

9) Restricción de no anticipatividad. Si dos escenarios comparten una trama en cierto período, deben representar una misma decisión. Esto queda establecido en la ecuación (11).

En comparación con el modelo determinístico, el modelo estocástico incorpora la incertidumbre en sus variables y sus parámetros. Un problema con la misma cantidad de productos, la misma cantidad de períodos y los mismos costos pero con una demanda variable, puede producir diferentes planificaciones según la aleatoriedad de la demanda y la probabilidad de que ocurra determinada demanda.

Capítulo 4: Implementación

4.1: Modelo Determinístico - GLPK

GLPK (GNU Linear Programming Kit) [11] es un programa utilizado para la resolución de problemas de programación lineal y programación entera mixta (donde hay variables enteras mezcladas con variables reales) a gran escala. Este fue el programa utilizado para implementar el modelo determinístico.

En la codificación del modelo mediante GLPK se tienen dos archivos: un archivo .mod donde se incluye (escrito en el lenguaje MathProg que es el que se usa en GLPK) la definición de conjuntos, variables y parámetros, la función objetivo y las restricciones. Y un archivo .dat donde se incluye la asignación de valores para cada parámetro y conjunto. Esto tiene la ventaja de que el modelo y los datos a utilizar vienen separados, permitiendo que se utilicen varios archivos .dat para un mismo archivo .mod.

En el archivo .mod establecimos un conjunto de productos PRODUCTOS y un conjunto de períodos PERIODOS. Luego definimos un conjunto de variables que dependen de cada producto y período. Entre estas variables tenemos: X(unidades), S(stock), R(demanda insatisfecha), Y(indica si se produjo) y U(indica cambio de artículo). Todas las variables son enteras mayores o iguales a cero, salvo las dos últimas que son binarias.

Definimos parámetros en función del producto y períodos para: dem (demanda), Cenv(costo de envasado), Celab(costo de elaboración), Calm(costo de almacenamiento). También definimos otros parámetros pero sólo en función del producto, a saber: Cres(costo de restablecimiento), Cins(costo de insuficiencia), Kelab(costo de elaboración) y Kenv(costo de envasado).

Tomamos en nuestro problema como constante KenvM(capacidad máxima de envasado) y KelabM(capacidad máxima de elaboración). Estos parámetros son comunes para todos los productos y resume la máxima cantidad de unidades que pueden envasar y elaborar entre todos los productos y para cada período respectivamente. Y por último dentro del archivo .mod se definen la función objetivo y las restricciones.

En el archivo .dat sólamente se incluyen los valores que tiene cada parámetro como también los elementos de cada conjunto. En el Anexo B se encuentra solamente el archivo .mod.

GLPK se ejecuta en línea de comando y recibe el nombre del archivo .mod, el nombre del archivo .dat y el nombre del archivo en donde se guardará la solución encontrada y el valor óptimo hallado. Al ejecutarse GLPK, este busca la solución utilizando el método branch and bound, y presenta dos salidas. Una en consola que muestra el log de la ejecución indicando si

se encontró una solución óptima y en cuanto tiempo la ha encontrado, y la otra en texto en el archivo indicado como parámetro.

4.2: Modelo Estocástico - GLPK

También se utilizó GLPK para implementar el modelo estocástico en donde la demanda es incierta y se representa con escenarios y sus probabilidades asociadas. En el Anexo B se encuentra el código fuente del archivo .mod respectivo.

Para resolver esta sección, nos apoyamos en lo realizado en la sección anterior, agregándole restricciones de no anticipatividad y modificando la función objetivo. A los conjuntos de PRODUCTOS y PERIODOS, se agrega un nuevo conjunto de escenarios, ESCENARIOS. Mantenemos las variables enteras y positivas: X, S, R. Las variables Y, U, estos últimos son binarias. También mantenemos los parámetros KenvM y KelabM, al igual que : Dem, Cenv, Celab, Calm, Cres, Cins, Prob, Kelab y Kenv.

Para resolver esta sección utilizamos una matriz, definida de la siguiente manera:
param matriz{s in ESCENARIOS, sp in ESCENARIOS, t in PERIODOS};

El objetivo de esta matriz es modelar la restricción de no anticipatividad. Para los escenarios s y sp y el período t, matriz[s,sp,t]=1 si y sólo si en el período t, los escenarios s y sp están dentro de la misma situación de incertidumbre (representan una misma decisión). En los demás casos matriz[s,sp,t]=0.

La función objetivo es:
minimize z:
sum{s in ESCENARIOS}(sum{p in PRODUCTOS,t in PERIODOS:t > 0 } ((Cenv[p,t] + Celab[p,t])*X[p,t,s] + Calm[p,t]*S[p,t,s]+ Cres[p]*U[p,t,s] + Cins[p]*R[p,t,s])*Prob[s]);

Nos basamos en las restricciones generadas en la sección anterior, agregando una restricción de no anticipatividad. Estas restricciones surgen de manera de satisfacer el ”Principio de No Anticipatividad” , formulado por Wets y Rockafellar [10].

Dicha restricción es:
s.t. noanticipatividad_X{p in PRODUCTOS, s in ESCENARIOS, t in PERIODOS:t>0}: sum{sp in ESCENARIOS}na[s,sp,t] * X[p,t,sp] = (sum{sp in ESCENARIOS} na[s,sp,t]) * X[p,t,s];

4.3: Modelo Estocástico - Algoritmos Genéticos

Esta parte consiste en resolver el mismo problema de optimización del punto 4.2 pero utilizando metaheurísticas. Se eligió el enfoque de usar Algoritmos Genéticos. La idea de usar Algoritmos Genéticos fue en parte por la experiencia propia en ese campo, pero también por el paper de Yoshitomi et. al [4], en donde se plantearon el uso de Algoritmos Genéticos para la resolución de optimización estocástica. Y aunque el enfoque en que lo planteaban era diferente al que se

utilizó en este proyecto, se sacó la idea de que el valor de fitness de una solución sea igual al valor de la función objetivo para esa solución, si y sólo si esa solución cumplía con todas las restricciones del problema. En caso de que no fuese así, se elegía un valor tal que lo volviera una solución nada apta. Por ejemplo, si la función objetivo era maximizar, el fitness elegido era 0 para las soluciones que no cumplían con todas las restricciones.

Se eligió, para la implementación la librería Mallba [12], escrita en C++. Mallba es una librería para resolver problemas de optimización combinatoria que provee esqueletos que se instancian con las características de un problema concreto para resolverlo y brinda esqueletos que implementan distintos métodos de resolución (como ser Algoritmos Genéticos, Recocido Simulado, Estrategias de Evolución e híbridos) y las interrelaciones necesarias. Tiene implementado el Solver que ejecuta el algoritmo genético, la clase población en donde se guardan las soluciones de cada generación, los algoritmos de selección y el historial de estadísticas. A cambio, se tienen que implementar allí las clases Problema y Solución, los algoritmos de cruzamiento y mutación y las condiciones de parada. En el Anexo B se encuentra el código fuente de la implementación utilizada.

4.3.1: Enfoque.

Se quiso mantener el mismo enfoque del punto 4.2 de usar una matriz tridimensional para representar las cantidades producidas de un producto para un período en un escenario y agregar restricciones de no anticipatividad. Luego nos dimos cuenta que a la hora de crear la función de fitness, varias de las restricciones podían relajarse. Por un lado,había restricciones que podían contemplarse a la hora de crear las soluciones y por otro lado había restricciones que sólo eran usadas para definir los valores de las variables Y, R, S y U, y estas podían ser definidas a la hora de calcular el fitness. Al final de las 11 restricciones que teníamos, sólo cuatro determinaban si la solución era apta o no:
1) restricciones de capacidades de envasado
2) restricciones de capacidades de elaboración
3) la suma de las cantidades producidas de un producto en un escenario es igual o mayor a la demanda de ese producto en ese escenario
4) en ningún período y ningún escenario, ningún producto produce más que la sumatoria de la demanda sobre todos los períodos de ese producto y escenario.

4.3.2: Implementación

La figura 3 muestra el esquema de clases que hay en la biblioteca Mallba. Las clases marcadas con verde son las clases que se necesitan implementar, mientras que las clases azules son provistas por la biblioteca. En este trabajo sólo se hará mención de las clases Problema y Solución.

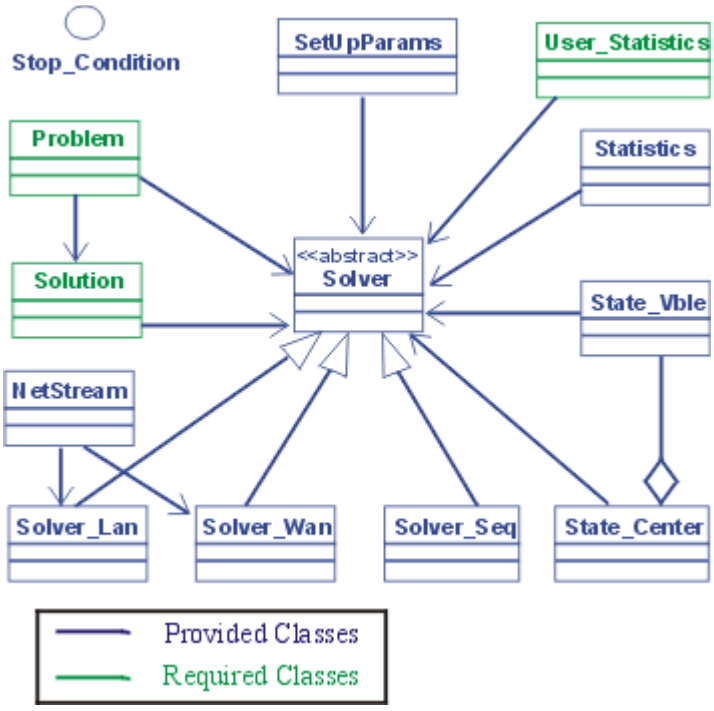

Fig.3 - Esquema de Clases de Mallba

La clase Solución en Mallba representa las soluciones, y tiene sólo dos atributos. El problema al que está asignado y una matriz la cual representa las cantidades producidas (el cromosoma en sí). Esta incluye métodos de construcción (a partir de un problema o a partir de una solución), destrucción, asignación, igualdad y desigualdad, conversión a string y viceversa, entrada y salida por stream y funciones de inicialización y cálculo de fitness. Además de los métodos get y set de la Matriz y el get del Problema.

En la matriz la cantidad de filas es la cantidad de productos y la cantidad de columnas representan períodos por escenarios. Si el problema tiene *n* escenarios, las primeras *n* columnas representan las cantidades producidas para cada escenario en el período 1, las siguientes n columnas representan las cantidades producidas para cada escenario en el período 2, y así sucesivamente. Esta representación de la matriz fue elegida para facilitar la comprensión a la hora de hacer los cruzamientos y la mutación.

La clase Problema en Mallba representa el problema a resolver. Tiene como atributos la cantidad de productos, la cantidad de períodos, la cantidad de posibilidades, la cantidad de escenarios (la cual se deduce a partir de los períodos y las posibilidades), la probabilidad asociada de cada escenario, los topes máximos de capacidades, los costos de elaboración, envasado y almacenamiento por producto y período, las capacidades de elaboración y envasado de cada producto, los costos de demanda insatisfecha y cambio de producto y las demandas de cada producto para cada período y cada escenario. Solamente tiene las funciones get, entrada y salida stream, construcción, destrucción, asignación, igualdad y

desigualdad. Además cuenta con un vector especial. Un vector que contiene las columnas en las cuales se puede realizar el cruzamiento y la mutación.

Cuando se crea una solución, se fija si la columna seleccionada es una columna perteneciente al vector especial. Si la columna pertenece al vector, se elige una fila al azar (un producto) y luego se elige una cantidad entera al azar. Para esa columna y esa fila, esa cantidad será su valor, mientras que para las demás filas de esa columna su valor será cero. Si la columna no pertenece al vector, tomará los valores de la columna anterior (Nótese que la columna 0, que representa el primer escenario y el primer período, siempre pertenece a ese vector). De esta forma, las soluciones creadas siempre respetarán la restricción de no anticipatividad, además de la restricción de que no se produce más de un artículo para un mismo período y un mismo escenario.

A la hora de hacer el cálculo del fitness para esa solución, primero controla si se respetan los topes de capacidades y luego si cumplen con las restricciones de las demandas. Si no se cumple alguna de esas restricciones, se retorna un valor muy alto como penalización. Si cumple con todas, se crean las matrices de demanda insatisfecha, de stock almacenado y de cambio de artículo, en base a las matrices de la demanda y la de cantidad producida, y en base a todo esto se calcula el fitness.

En el procedimiento del cruzamiento, se toman las dos matrices a cruzar, se sortea una columna perteneciente al vector especial y desde esa columna hasta la última columna, se intercambian los valores de una matriz con la otra. Mientras que en el procedimiento de la mutación, se toma la matriz a mutar, se sortea una columna perteneciente al vector especial y se busca la siguiente columna perteneciente a ese vector. Se elige una fila al azar y una cantidad entera al azar y para esa columna y esa fila, esa cantidad será su valor, mientras que para las demás filas de esa columna su valor será cero. Luego ese valor se copia para todas las columnas que hay en medio.

Finalmente, como condición de parada se eligió, que se llegase a la última generación definida o que existiese una solución encontrada (o sea, su fitness no sea un valor muy alto) y dicha solución mejoró menos del 10% con respecto a la mejor solución obtenida anteriormente.

4.3.3: Heurísticas adicionales

Para mejorar los resultados aún más se aplicaron los siguientes criterios, tanto a la hora crear las soluciones como en las mutaciones.

La primera heurística es que cuando se genera la cantidad producida al azar de un producto, colocamos como cantidad tope la sumatoria de la demanda del producto seleccionado para todos los períodos en ese escenario. Así, se controla de que en ningún período y ningún escenario, ningún producto produzca más que la sumatoria de la demanda sobre todos los períodos de ese producto y escenario.

La segunda, si un producto tiene en un período una demanda menor a la cantidad almacenada, podría no producirse ninguna cantidad de ese producto en ese período.

Otra heurística es que si en el último período, el producto seleccionado no tiene cantidades almacenadas, que solamente produzca la cantidad justa para ese último período.

Capítulo 5: Pruebas y Resultados

Todas las pruebas fueron realizadas en una computadora Lenovo Z570, con 8 GB de RAM, procesador Intel Core I7-2670QM de 2.2 Ghz, Sistema Operativo GNU/Linux, la versión 4.45 de GLPK, la versión 3.04 de mpich y la versión 4.7.3 de GCC.

5.1: Modelo Determinístico - GLPK

En las sucesivas pruebas veremos los resultados para la resolución del modelo determinístico, estas pruebas se generaron mediante software GLPK. El archivo .mod con el modelo queda disponible en el Anexo B.

Prueba 1

En esta prueba se consideraron 2 productos y 4 períodos.
El valor del óptimo es 1.083.300, obtenido en menos de un segundo. En la Tabla 5.1 se resumen los resultados de la solución óptima.

Param[Producto]	Período 1	Período 2	Período 3	Período 4
D[I1]	1000	0	0	0
X[I1]	1000	0	0	0
S[I1]	0	0	0	0
R[I1]	0	0	0	0
D[I2]	1000	0	0	0
X[I2]	0	1000	0	0
S[I2]	0	0	0	0
R[I2]	1000	0	0	0

Tabla 5.1 - Esquema de solución de Prueba 1

Esta primera prueba fue para corroborar que funcionaba la implementación del modelo determinístico. GLPK eligió que se produjera primero el producto I1 en el primer período y luego el producto I2. Para el tercer período ya no había más demanda que satisfacer y por eso no se produjo nada más.

Prueba 2

En esta prueba se consideraron 2 productos y 4 períodos, veremos como se cambia el orden de producción. Para lograr esto, nos basta con aumentar el costo de insuficiencia en el producto I2(costo de no producir en cierto período cuando hay demanda) desde 1000 a 1800. En la tabla 5.2 se muestran los siguientes resultados:

Param[Producto]	Período 1	Período 2	Período 3	Período 4
D[I1]	1000	0	0	0
X[I1]	0	1000	0	0
S[I1]	0	0	0	0
R[I1]	1000	0	0	0
D[I2]	1000	0	0	0
X[I2]	1000	0	0	0
S[I2]	0	0	0	0
R[I2]	0	0	0	0

Tabla 5.2 - Esquema de solución de Prueba 2

Con un valor del óptimo mínimo reportado z = 1283300, obtenido en menos de un seg. Efectivamente observamos que se decide producir el producto 2 antes que el producto 1, esto es debido a que el costo de insuficiencia del producto I2 es mayor que el costo de insuficiencia del producto I1.

Prueba 3

Se consideran 3 productos y 4 períodos.
Los datos para la demanda y los diferentes períodos son los que muestra la tabla 5.3:

Demanda	Período 1	Período 2	Período 3	Período 4
D[I1]	1000	0	0	0
D[I2]	1000	0	0	0
D[I3]	1000	0	0	0

Tabla 5.3 - Esquema de demandas de Prueba 3

Dado que Costo de envasado, Costo de Elaboración, Costo de Insuficiencia, Costo de Almacenamiento y las diferentes capacidades son iguales, evidentemente ninguno de estos parámetros influirá en la determinación de las cantidades a producir. La diferencia reside en los valores para Cres como lo muestran las tablas 5.4 y 5.5:

Producto	Cres	Cins	Kelab	Kenv
I1	1800	1000	400	800
I2	1500	1000	400	800
I3	1500	1000	400	800

Tabla 5.4 - Esquema de costos globales de Prueba 3

Parámetros	Periodo 1	Periodo 2	Periodo 3	Periodo 4
Celab[I1]	20	20	20	20
Celab[I2]	20	20	20	20
Celab[I3]	20	20	20	20
Cenv[I1]	20	20	20	20
Cenv[I2]	20	20	20	20
Cenv[I3]	20	20	20	20
Calm[I1]	20	20	20	20
Calm[I2]	20	20	20	20
Calm[I3]	20	20	20	20

Tabla 5.5 - Esquema de costos unitarios de Prueba 3

Según las pruebas, el producto I2 fue el primero en producirse, luego el I1 y finalmente el I3. El valor del óptimo mínimo reportado fue de 3124800 y también tardó menos de un segundo. En la tabla 5.6 se muestra el resultado.

Param[Producto]	Período 1	Período 2	Período 3	Período 4
X[I1]	0	1000	0	0
S[I1]	0	0	0	0
R[I1]	1000	0	0	0
X[I2]	1000	0	0	0
S[I2]	0	0	0	0
R[I2]	0	0	0	0
X[I3]	0	0	1000	0

S[I3]	0	0	0	0
R[I3]	1000	1000	0	0

Tabla 5.6 - Esquema de solucion de Prueba 3

5.2: Modelo Estocástico - GLPK

En las sucesivas pruebas veremos los resultados para la resolución del modelo estocástico, estas pruebas se generaron mediante software GLPK. El archivo .mod con el modelo queda disponible en el Anexo B.

Prueba 1
Este ejemplo está generado para 2 productos, 2 posibilidades (mala, buena) 5 períodos y 16 escenarios, cada uno con probabilidad de 0.0625. Esto representa 1360 inecuaciones de restricciones y 896 variables, de las cuales 352 son binarias. El tiempo de resolución fue de 76.8 seg, 1 min y 17 seg aprox. El valor mínimo objetivo es de 1015.25.

Esta prueba fue para corroborar que funcionaba la implementación del modelo estocástico en GLPK, además de comprobar el aumento del tiempo de ejecución al pasar del modelo determinístico al estocástico. Esto se debe al aumento de la dimensión del problema.

Prueba 2
Este ejemplo está generado para 2 productos, 2 posibilidades (mala, buena) 5 períodos y 16 escenarios, cada uno con probabilidad de 0.0625. Esto representa 1360 inecuaciones de restricciones y 896 variables, de las cuales 352 son binarias. La diferencia con la prueba 1 fueron los costos (aumentaron los costos de elaboración, envasado, almacenamiento, reestablecimiento e insuficiencia). El tiempo de resolución fue de 309.5 seg, 5 min y 10 seg aprox. El valor mínimo objetivo es de 2985.

La prueba fue para corroborar que la modificación de los costos, no sólo influye sobre la elección de las planificaciones óptimas, sino también sobre el tiempo de ejecución, dado que ha tardado más tiempo que en la Prueba 1.

Prueba 3
Este ejemplo está generado para 3 productos, 2 posibilidades (mala, buena) 5 períodos y 16 escenarios, cada uno con probabilidad de 0.0625. Esto representa 1920 inecuaciones de restricciones y 1344 variables, de las cuales 528 son binarias. El tiempo de resolución fue de 106.5 seg, 1 min y 46 seg aprox. El valor mínimo objetivo es de 7136.

La prueba fue para corroborar cómo influiría en el tiempo de ejecución el aumentar la cantidad de productos. Si bien tuvo un menor tiempo de ejecución que la prueba 2, hay que recordar que los costos también influyen.

Prueba 4

Este ejemplo está generado para 2 productos, 3 posibilidades (mala, intermedia, buena) 4 períodos y 27 escenarios, cada uno con probabilidad de 0.037. Esto representa 1890 inecuaciones de restricciones y 1242 variables, de las cuales 486 son binarias. El tiempo de resolución fue de 4 seg. El valor mínimo objetivo es de 877.

La prueba fue para corroborar cómo influiría en el tiempo de ejecución el quitar un período y aumentar las posibilidades (tiene más escenarios que las pruebas anteriores). Una hipótesis de que haya durado tan poco es que reducir la cantidad de períodos influye más que aumentar la cantidad las posibilidades.

Prueba 5
Este ejemplo es similar al de la prueba 2 pero cada escenario tiene probabilidad de 0.01, salvo los escenarios 4,8,12 y 16 que tienen 0.22. El ejemplo está generado para 2 productos, 2 probabilidades(mala, buena) 5 períodos y 16 escenarios. Esto representa 1360 inecuaciones de restricciones y 896 variables, de las cuales 352 son binarias. El tiempo de resolución fue de 75 seg, 1 min y 15 seg aprox. El valor mínimo objetivo es de 2892.

La prueba fue para corroborar cómo influiría en el tiempo de ejecución el cambiar las probabilidades de cada escenario. Durante la experimentación observamos que el tiempo de ejecución disminuye cuando los escenarios se alejan de la equiprobabilidad. También observamos que el valor mínimo objetivo se acerca al de los escenarios con mayor probabilidad.

5.3: Modelo Estocástico - Algoritmos Genéticos

Se utiliza la misma función objetivo del punto 5.2. Todas las pruebas ejecutan 100 corridas independientes de 200 generaciones cada una, tienen poblaciones de 1000 soluciones en cada generación y sus probabilidades de cruzamiento y mutación son del 50%.

También se ejecutarán dichas pruebas en GLPK, con un máximo de 900 segundos (15 minutos) de ejecución. Las soluciones se GA y GLPK se comparan en términos de distancia vectorial absoluta y error relativo del óptimo. La distancia absoluta se mide como la norma euclidiana de la diferencia de las soluciones, y el error relativo del óptimo como el cociente entre la diferencia entre la mejor solución en GA y el óptimo de GLPK, normalizado contra el óptimo de GLPK.

Prueba 1: 2 productos, 5 períodos, 16 escenarios, cada uno con probabilidad de 0.0625.

El algoritmo genético corrió durante 126 segundos y encontró una solución cuyo valor objetivo es de 48797. Por otro lado, la misma prueba en GLPK corrió durante 494 segundos y encontró una solución óptima cuyo valor objetivo es de 47913. La distancia vectorial entre ambas soluciones es de 113,72 y el error relativo del óptimo es de un 1,8%

Esta prueba fue para corroborar que funcionaba la implementación del modelo estocástico en Mallba, además de comprobar no solo que para este y los siguientes casos la ejecución en

GLPK llega a durar muchos minutos, sino cuan próximas al óptimo pueden ser las soluciones encontradas con respecto a la de GLPK.

Prueba 2: 2 productos, 5 períodos, 16 escenarios, cada uno con probabilidad de 0.0625. Con alteraciones en los costos y en la demanda con respecto a la Prueba 1

El algoritmo genético corrió durante 240 segundos y encontró una solución cuyo valor objetivo es de 94192. Por otro lado, la misma prueba en GLPK corrió durante 682 segundos y encontró una solución óptima cuyo valor objetivo es de 92199. La distancia vectorial entre ambas soluciones es de 122,77 y el error relativo del óptimo es de un 2,2%.

Esta prueba fue para comprobar que el tiempo de ejecución en Mallba no depende ni de la dimensión del problema ni de sus parámetros, sino de la cantidad de iteraciones y generaciones.

Prueba 3: 3 productos, 5 períodos, 16 escenarios, cada uno con probabilidad de 0.0625.

El algoritmo genético corrió durante 167 segundos y no pudo encontrar una solución factible. Por otro lado, la misma prueba en GLPK corrió durante 564 segundos y encontró una solución óptima cuyo valor objetivo es de 5652. Se considera este un caso de excepción.

Esta prueba fue para comprobar no sólo la importancia de tener un gran número de generaciones e iteraciones para poder hallar una solución, sino que la implementación en Mallba necesita mejorar.

Prueba 4: 2 productos, 4 períodos, 27 escenarios, cada uno con probabilidad de 0.037.

El algoritmo genético corrió durante 143 segundos y encontró una solución cuyo valor objetivo es de 954286. Por otro lado, la misma prueba en GLPK corrió durante 844 segundos y encontró una solución óptima cuyo valor objetivo es de 726200. La distancia vectorial entre ambas soluciones es de 146 y el error relativo del óptimo es de un 31%

Esta prueba fue para comprobar que un aumento en los escenarios influye negativamente en la optimalidad de la solución hallada en la implementación de Mallba.

Prueba 5: 2 productos, 5 períodos, 16 escenarios no equiprobables. Es igual a la prueba 1 pero con las probabilidades alteradas apenas distintas de la equiprobabilidad.

El algoritmo genético corrió durante 131 segundos y encontró una solución cuyo valor objetivo es de 48540. Por otro lado, la misma prueba en GLPK corrió durante 307 segundos y encontró una solución óptima cuyo valor objetivo es de 47903. La distancia vectorial entre ambas soluciones es de 87 y el error relativo del óptimo es 1,3%

Esta prueba y las siguientes fueron para comprobar que tampoco un cambio de las probabilidades influye en el tiempo de ejecución en Mallba. De todas las pruebas, esta tuvo la solución de menor distancia vectorial.

Prueba 6: 2 productos, 5 períodos, 16 escenarios no equiprobables. Es igual a las pruebas 1 y 5 pero con otra alteración de las probabilidades. En esta vez, las probabilidades están más concentradas en ciertos escenarios claves.

El algoritmo genético corrió durante 131 segundos y encontró una solución cuyo valor objetivo es de 49028. Por otro lado, la misma prueba en GLPK corrió durante 339 segundos y encontró una solución óptima cuyo valor objetivo es de 47932. La distancia vectorial entre ambas soluciones es de 127,09 y el error relativo del óptimo es de un 2,3%

Esta prueba fue sólo para ver como empezar a concentrar las probabilidades de manera que GLPK no acabara su ejecución tan rápidamente.

Prueba 7: 2 productos, 4 períodos, 27 escenarios, Es igual a la prueba 4 pero con las probabilidades alteradas y concentradas en ciertos escenarios.

El algoritmo genético corrió durante 145 segundos y encontró una solución cuyo valor objetivo es de 1915860. Por otro lado, la misma prueba en GLPK corrió durante 252 segundos y encontró una solución óptima cuyo valor objetivo es de 726605. La distancia vectorial entre ambas soluciones es de 175 y el error relativo del óptimo es de un 164%

Esta prueba al igual que la prueba 6 fue sólo para ver como empezar a concentrar las probabilidades de manera que GLPK no acabara su ejecución tan rápidamente, pero esta vez en un problema de tres posibilidades.

Prueba 8: 2 productos, 4 períodos, 27 escenarios, Es igual a las pruebas 4 y 7 pero con un escenario con 49% de probabilidad y el resto de forma equiprobable.

El algoritmo genético corrió durante 141 segundos y encontró una solución cuyo valor objetivo es de 823725. Por otro lado, la misma prueba en GLPK corrió durante 302 segundos y encontró una solución óptima cuyo valor objetivo es de 726871. La distancia vectorial entre ambas soluciones es de 123,17 y el error relativo del óptimo es de un 13%

Esta prueba sólo para ver que tan cerca se puede llegar a una distribucion de probabilidades concentrada en un único escenario de manera que GLPK no acabara su ejecución tan rápidamente.

Capítulo 6: Conclusiones y Trabajo Futuro

En este trabajo se ha planteado dos modelos algebraicos para la resolución de un problema de llegar a una planificación que optimice costos de producción para una cantidad de períodos y una cantidad de productos. Se desarrollaron dos modelos, un modelo para el caso de una demanda cierta (modelo determinístico) y otro para una demanda incierta (modelo estocástico). Además se han hecho implementaciones computacionales de dichos modelos, en donde el segundo modelo fue implementado primero utilizando una herramienta de resolución de problemas de optimización lineal y entera y segundo utilizando una metaheurística (algoritmos genéticos). Se han hecho pruebas casi satisfactorias de las implementaciones de dichos modelos.

Sobre el modelo determinístico, es necesario expresar que era esperable que después de codificar el modelo y plantear el conjunto de pruebas para el modelo, las pruebas se hayan ejecutado casi instantáneamente. Al haber un único escenario, la dimensión (cantidad) de las variables es función de productos por períodos, mientras que en el modelo estocástico la dimensión se multiplica por la cantidad de escenarios, la cual es exponencial y dependiente de los períodos.

En cuanto al modelo estocástico, Mallba y GLPK tienen formas distintas de resolver el problema. GLPK primero encuentra la solución óptima relajada a programación lineal y luego busca las soluciones más cercanas al óptimo que sean factibles y de entre ellas elige la óptima mediante el método de branch and bound. Mallba por otro lado parte de un conjunto de soluciones no necesariamente factibles y busca llegar a tener soluciones factibles, y dentro de las factibles se queda con la mejor solución encontrada. La solución encontrada por GLPK siempre es la óptima, pero en ciertos casos el tiempo que toma encontrarla resulta ser tan largo que uno podría conformarse con una solución factible encontrada más rápidamente, aunque no sea óptima.

En cuanto a las pruebas, lo primero que hay que notar es que el tiempo de ejecución en GLPK depende de las dimensiones del problema y sus parámetros, mientras que en Mallba se depende de la cantidad de iteraciones y generaciones. Sin embargo si se aumentan los escenarios, las soluciones que devuelve Mallba están más alejadas y son peores que las de GLPK. Otra cosa que se notó es que el tiempo de ejecución de GLPK se reduce cuando las probabilidades se concentran en un grupo pequeño de escenarios.

Sobre las dificultades que hubo durante la realización del proyecto, no hubo dificultades en cuanto al uso de GLPK, pero sí hubo mucha dificultad para hacer funcionar la librería Mallba. La librería contenía muchos errores que surgían en tiempo de compilación (malas referencias a otras librerías), incluso había errores de codificación que no generaban problemas de compilación pero daban fallas (una mala asignación en el módulo que implementaba las matrices).

Las diferencias entre lo que se planteó y lo que se hizo fueron pocas y estas fueron mencionadas entre los supuestos para los dos modelos algebraicos del capítulo 3. Se plantearon originalmente el uso de ingredientes que al final quedaron incluidos dentro de los costos y capacidades de elaboración de cada producto. Se planteó un inventario inicial y al final se asumió que ese inventario inicial era vacío. Se mencionó en principio que sólo tendríamos dos productos y luego el modelo final fue extendido para más productos.

Como trabajo futuro se proponen las siguientes ideas:

1) Plantear un modelo más complejo donde se consideren más de una máquina de elaboración y envasado, más de una fábrica, la posibilidad de transferir unidades de una fábrica a otra, la existencia de productos compuestos, fórmulas para crear productos compuestos a partir de productos simples, la existencia de un stock mínimo(o de seguridad) de almacenamiento haciendo que sea más costoso satisfacer la demanda con ese stock mínimo, etc.

2) Plantear mejores estrategias en cuanto a la resolución de problemas utilizando metaheurísticas. Por ejemplo, para el caso de utilizar algoritmos genéticos, sugerir mejores criterios a la hora de generar números aleatorios para la creación de soluciones, el cruzamiento y la mutación.

Referencias Bibliográficas

[1] Belvaux, G.; Wolsey, L. A. 2001, “Modelling Practical Lot-Sizing Problems as Mixed-Integer Programs”, *Management Science*, Vol. 47, Nº 7, pag 993-1007

[2] Simpson, N. C.; Erenguc, S. S. 1998, “Production Planning in multiple stage manufacturing environments with joint costs, limited resources and set-up times”, Technical report, Department of Management Science and Systems, University of Buffalo, Buffalo, NY.

[3] Brown, G.; Keegan, J.; Vigus, B.; Wood, K. 2001, “The Kellog Company Optimizes Production, Inventory, and Distribution”, *Interfaces*, Vol. 31, Nº 6, pag 1-15

[4] Yoshitomi, Y.; Ikenoue, H.; Takeba, T.; Tomita, S. 2000, “Genetic Algorithm in Uncertain Environments for solving Stochastic Programming Problem”, *Journal of the Operations Research Society of Japan*. Vol. 43, Nº 2, pag 266-290

[5] http://www.vitutor.com/algebra/pl/a_1.html

[6] http://www.dm.uba.ar/materias/investigacion_operativa/2011/2/capit_5.pdf

[7] http://www.ing.ula.ve/~aguilar/actividad-docente/AYDA/Clase12MiniSem.pdf

[8] http://www.fing.edu.uy/inco/grupos/invop/mh/material/pres4.ppt

[9] http://www.sc.ehu.es/ccwbayes/docencia/mmcc/docs/temageneticos.pdf

[10] Rockafellar, R.T.; Wets, R. J.-B. (1991). “Scenario and policy aggregation in optimization under uncertainty”. *Mathematics of Operations Research*, Vol 16, Nº 1, pag:119–147.

[11] https://www.gnu.org/software/glpk/

[12] http://neo.lcc.uma.es/mallba/easy-mallba/index.html

Apéndice A: Pruebas

Prueba 1 - Modelo Determinístico

2 Productos, 4 Periodos

KelabM- 1000000000, KenvM - 5000000000

Producto	Cres	Cins	Kelab	Kenv
I1	1800	1200	400	800
I2	1500	1000	25	15

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	20	20	20	20
I2	20	20	20	20

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	20	20	20	20
I2	20	20	20	20

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	20	20	20	20
I2	20	20	20	20

Demanda	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	1000	0	0	0
I2	1000	0	0	0

Prueba 2 - Modelo Determinístico

2 Productos, 4 Periodos

KelabM- 1000000000, KenvM - 5000000000

Producto	Cres	Cins	Kelab	Kenv
I1	1800	1200	400	800
I2	1500	1800	25	15

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	20	20	20	20
I2	20	20	20	20

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	20	20	20	20
I2	20	20	20	20

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	20	20	20	20
I2	20	20	20	20

Demanda	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	1000	0	0	0
I2	1000	0	0	0

Prueba 3 - Modelo Determinístico

3 Productos, 4 Periodos

KelabM- 1000000000, KenvM - 5000000000

Producto	Cres	Cins	Kelab	Kenv
I1	1800	1000	400	800
I2	1500	1000	400	800

I3	1500	1000	400	800

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	20	20	20	20
I2	20	20	20	20
I3	20	20	20	20

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	20	20	20	20
I2	20	20	20	20
I3	20	20	20	20

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	20	20	20	20
I2	20	20	20	20
I3	20	20	20	20

Demanda	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	1000	0	0	0
I2	1000	0	0	0
I3	1000	0	0	0

Prueba 1 - Modelo Estocástico - GLPK

2 Productos, 5 Periodos, 2 Posibilidades - los escenarios son 16

KelabM- 100000000, KenvM - 100000000, Prob[s] = 0,0625 para cada escenario s.

Producto	Cres	Cins	Kelab	Kenv
I1	50	50	50	50
I2	30	30	30	30

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	5	5	5	5	5
I2	3	3	3	3	3

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	5	5	5	5	5
I2	3	3	3	3	3

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	5	5	5	5	5
I2	3	3	3	3	3

demanda período 1 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
I2	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6

demanda período 2 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	6	6	6	6	6	6	6	6
I2	6	6	6	6	6	6	6	6	4	4	4	4	4	4	4	4

demanda período 3 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	6	6	6	6	8	8	8	8	6	6	6	6
I2	6	6	6	6	4	4	4	4	6	6	6	6	4	4	4	4

demanda período 4 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	6	6	8	8	6	6	8	8	6	6	8	8	6	6

I2	6	6	4	4	6	6	4	4	6	6	4	4	6	6	4	4

demanda período 5 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	6	8	6	8	6	8	6	8	6	8	6	8	6	8	6
I2	6	4	6	4	6	4	6	4	6	4	6	4	6	4	6	4

Prueba 2 - Modelo Estocástico - GLPK

2 Productos, 5 Periodos, 2 Posibilidades - los escenarios son 16

KelabM- 100000000, KenvM - 100000000, Prob[s] = 0,0625 para cada escenario s.

Producto	Cres	Cins	Kelab	Kenv
I1	400	25	50	50
I2	25	400	30	30

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	20	20	20	20	20
I2	5	5	5	5	5

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	20	20	20	20	20
I2	5	5	5	5	5

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	5	5	5	5	5
I2	20	20	20	20	20

demanda período 1 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8

I2	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6

demanda período 2 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	6	6	6	6	6	6	6	6
I2	6	6	6	6	6	6	6	6	4	4	4	4	4	4	4	4

demanda período 3 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	6	6	6	6	8	8	8	8	6	6	6	6
I2	6	6	6	6	4	4	4	4	6	6	6	6	4	4	4	4

demanda período 4 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	6	6	8	8	6	6	8	8	6	6	8	8	6	6
I2	6	6	4	4	6	6	4	4	6	6	4	4	6	6	4	4

demanda período 5 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	6	8	6	8	6	8	6	8	6	8	6	8	6	8	6
I2	6	4	6	4	6	4	6	4	6	4	6	4	6	4	6	4

Prueba 3 - Modelo Estocástico - GLPK

2 Productos, 5 Periodos, 2 Posibilidades - los escenarios son 16

KelabM- 100000000, KenvM - 100000000, Prob[s] = 0,0625 para cada escenario s.

Producto	Cres	Cins	Kelab	Kenv
I1	50	50	50	50
I2	30	30	30	30
I3	100	100	100	100

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	5	5	5	5	5
I2	3	3	3	3	3
I3	10	10	10	10	10

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	5	5	5	5	5
I2	3	3	3	3	3
I3	10	10	10	10	10

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	10	10	10	10	10
I2	20	20	20	20	20
I3	30	30	30	30	30

demanda período 1 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
I2	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
I3	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20	20

demanda período 2 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	6	6	6	6	6	6	6	6
I2	6	6	6	6	6	6	6	6	4	4	4	4	4	4	4	4
I3	20	20	20	20	20	20	20	20	7	7	7	7	7	7	7	7

demanda período 3 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	6	6	6	6	8	8	8	8	6	6	6	6

I2	6	6	6	6	4	4	4	4	6	6	6	6	4	4	4	4
I3	20	20	20	20	7	7	7	7	20	20	20	20	7	7	7	7

demanda período 4 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	6	6	8	8	6	6	8	8	6	6	8	8	6	6
I2	6	6	4	4	6	6	4	4	6	6	4	4	6	6	4	4
I3	20	20	7	7	20	20	7	7	20	20	7	7	20	20	7	7

demanda período 5 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	6	8	6	8	6	8	6	8	6	8	6	8	6	8	6
I2	6	4	6	4	6	4	6	4	6	4	6	4	6	4	6	4
I3	20	7	20	7	20	7	20	7	20	7	20	7	20	7	20	7

Prueba 4 - Modelo Estocástico - GLPK

2 Productos, 4 Periodos, 3 Posibilidades - los escenarios son 27

KelabM - 100000000, KenvM - 100000000, Prob[s] = 0,037 para cada escenario s.

Producto	Cres	Cins	Kelab	Kenv
I1	50	50	50	50
I2	30	30	30	30

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	5	5	5	5
I2	3	3	3	3

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	5	5	5	5
I2	3	3	3	3

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	5	5	5	5
I2	3	3	3	3

demanda período 1 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	8	8	8	8	8	8	8	8	8	8	8	8	8	8
I2	6	6	6	6	6	6	6	6	6	6	6	6	6	6

demanda período 1 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	8	8	8	8	8	8	8	8	8	8	8	8	8
I2	6	6	6	6	6	6	6	6	6	6	6	6	6

demanda período 2 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	8	8	8	8	8	8	8	8	8	7	7	7	7	7
I2	6	6	6	6	6	6	6	6	6	5	5	5	5	5

demanda período 2 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	7	7	7	7	6	6	6	6	6	6	6	6	6
I2	5	5	5	5	4	4	4	4	4	4	4	4	4

demanda período 3 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	8	8	8	7	7	7	6	6	6	8	8	8	7	7
I2	6	6	6	5	5	5	4	4	4	6	6	6	5	5

demanda período 3 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27

I1	7	6	6	6	8	8	8	7	7	7	6	6	6
I2	5	4	4	4	6	6	6	5	5	5	4	4	4

demanda período 4 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	8	7	6	8	7	6	8	7	6	8	7	6	8	7
I2	6	5	4	6	5	4	6	5	4	6	5	4	6	5

demanda período 4 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	6	8	7	6	8	7	6	8	7	6	8	7	6
I2	4	6	5	4	6	5	4	6	5	4	6	5	4

Prueba 5 - Modelo Estocástico - GLPK

2 Productos, 5 Periodos, 2 Posibilidades - los escenarios son 16

KelabM- 100000000, KenvM - 100000000
Prob[s] = 0,22 para los escenarios 4,8,12 y 16. Prob[s] = 0,01 para los demás escenarios.

Producto	Cres	Cins	Kelab	Kenv
I1	400	25	50	50
I2	25	400	30	30

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	20	20	20	20	20
I2	5	5	5	5	5

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	20	20	20	20	20
I2	5	5	5	5	5

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5

I1	5	5	5	5	5
I2	20	20	20	20	20

demanda período 1 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
I2	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6

demanda período 2 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	6	6	6	6	6	6	6	6
I2	6	6	6	6	6	6	6	6	4	4	4	4	4	4	4	4

demanda período 3 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	6	6	6	6	8	8	8	8	6	6	6	6
I2	6	6	6	6	4	4	4	4	6	6	6	6	4	4	4	4

demanda período 4 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	6	6	8	8	6	6	8	8	6	6	8	8	6	6
I2	6	6	4	4	6	6	4	4	6	6	4	4	6	6	4	4

demanda período 5 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	6	8	6	8	6	8	6	8	6	8	6	8	6	8	6
I2	6	4	6	4	6	4	6	4	6	4	6	4	6	4	6	4

Prueba 1 - Modelo Estocástico - Algoritmos Genéticos

2 Productos, 5 Periodos, 2 Posibilidades - los escenarios son 16

KelabM- 100000000, KenvM - 100000000.

Producto	Cres	Cins	Kelab	Kenv
I1	1000	7000	200	30
I2	1000	7000	15	250

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	16	8	4	2	1
I2	6	2	8	4	10

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	16	32	64	128	256
I2	6	2	8	4	10

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	1	2	4	8	16
I2	1	7	3	9	5

probabilidades

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625

demanda período 1 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
I2	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6

demanda período 2 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	12	12	12	12	12	12	12	12
I2	10	10	10	10	10	10	10	10	5	5	5	5	5	5	5	5

demanda período 3 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	7	7	7	7	5	5	5	5	13	13	13	13	4	4	4	4
I2	8	8	8	8	13	13	13	13	5	5	5	5	20	20	20	20

demanda período 4 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	15	15	2	2	6	6	8	8	3	3	14	14	7	7	10	10
I2	3	3	9	9	10	10	2	2	3	3	5	5	13	13	9	9

demanda período 5 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
I2	2	4	6	8	10	12	14	16	1	3	5	7	9	11	13	15

Prueba 2 - Modelo Estocástico - Algoritmos Genéticos

2 Productos, 5 Periodos, 2 Posibilidades - los escenarios son 16

KelabM- 100000000, KenvM - 100000000, Prob[s] = 0,0625 para cada escenario s.

Producto	Cres	Cins	Kelab	Kenv
I1	10000	7000	200	300
I2	10000	7000	150	250

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	160	80	40	20	50
I2	106	20	80	40	100

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	16	32	64	128	256
I2	60	20	80	40	100

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	0	1000	100	10	1
I2	0	1	10	100	1000

probabilidades

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625

demanda período 1 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
I2	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6

demanda período 2 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	9	9	9	9	9	9	9	9	12	12	12	12	12	12	12	12
I2	6	6	6	6	6	6	6	6	5	5	5	5	5	5	5	5

demanda período 3 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	9	9	9	9	12	12	12	12	9	9	9	9	12	12	12	12
I2	6	6	6	6	5	5	5	5	6	6	6	6	5	5	5	5

demanda período 4 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	9	9	12	12	9	9	12	12	9	9	12	12	9	9	12	12
I2	6	6	5	5	6	6	5	5	6	6	5	5	6	6	5	5

demanda período 5 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

I1	9	12	9	12	9	12	9	12	9	12	9	12	9	12	9	12
I2	6	5	6	5	6	5	6	5	6	5	6	5	6	5	6	5

Prueba 3 - Modelo Estocástico - Algoritmos Genéticos

3 Productos, 5 Periodos, 2 Posibilidades - los escenarios son 16

KelabM- 100000000, KenvM - 100000000, Prob[s] = 0,0625 para cada escenario s.

Producto	Cres	Cins	Kelab	Kenv
I1	50	50	50	50
I2	30	30	30	30
I3	100	100	100	100

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	5	5	5	5	5
I2	30	30	30	30	30
I3	10	10	10	10	10

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	50	50	50	50	50
I2	3	3	3	3	3
I3	10	10	10	10	10

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	10	10	10	10	10
I2	20	20	20	20	20
I3	30	30	30	30	30

probabilidades

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625	0.0625

demanda período 1 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8	8
I2	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6
I3	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7	7

demanda período 2 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	6	6	6	6	6	6	6	6
I2	6	6	6	6	6	6	6	6	4	4	4	4	4	4	4	4
I3	7	7	7	7	7	7	7	7	8	8	8	8	8	8	8	8

demanda período 3 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	6	6	6	6	8	8	8	8	6	6	6	6
I2	6	6	6	6	4	4	4	4	6	6	6	6	4	4	4	4
I3	7	7	7	7	8	8	8	8	7	7	7	7	8	8	8	8

demanda período 4 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	6	6	3	3	6	6	8	8	6	6	8	8	6	6
I2	6	6	2	2	6	6	4	4	5	5	4	4	6	6	8	8
I3	1	1	8	8	7	7	4	4	7	7	8	8	7	7	8	8

demanda período 5 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	1	6	8	4	8	6	7	6	8	10	8	6	13	6	8	6
I2	6	2	6	4	5	4	6	8	6	4	11	4	6	14	6	4

I3	7	8	3	8	7	6	7	8	9	8	7	12	7	8	15	8

Prueba 4 - Modelo Estocástico - Algoritmos Genéticos

2 Productos, 4 Periodos, 3 Posibilidades - los escenarios son 27

KelabM- 100000, KenvM - 100000, Prob[s] = 0,037 para cada escenario s.

Producto	Cres	Cins	Kelab	Kenv
I1	4	500000	40	60
I2	7000000	900	30	70

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	20	40	80	16
I2	15	20	40	80

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	10	30	40	80
I2	20	45	80	16

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	0	1000	100	10
I2	0	1	10	100

probabilidades de los escenarios del 1 al 14

1	2	3	4	5	6	7	8	9	10	11	12	13	14
0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037

probabilidades de los escenarios del 15 al 27

15	16	17	18	19	20	21	22	23	24	25	26	27
0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037	0.037

demanda período 1 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	8	8	8	8	8	8	8	8	8	8	8	8	8	8
I2	9	9	9	9	9	9	9	9	9	9	9	9	9	9

demanda período 1 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	8	8	8	8	8	8	8	8	8	8	8	8	8
I2	9	9	9	9	9	9	9	9	9	9	9	9	9

demanda período 2 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	9	9	9	9	9	9	9	9	9	6	6	6	6	6
I2	3	3	3	3	3	3	3	3	3	6	6	6	6	6

demanda período 2 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	6	6	6	6	3	3	3	3	3	3	3	3	3
I2	6	6	6	6	9	9	9	9	9	9	9	9	9

demanda período 3 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	11	11	11	22	22	22	33	33	33	44	44	44	55	55
I2	32	32	32	16	16	16	8	8	8	4	4	4	20	20

demanda período 3 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	55	66	66	66	77	77	77	88	88	88	99	99	99
I2	20	4	4	4	8	8	8	16	16	16	32	32	32

demanda período 4 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14

I1	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I2	27	26	25	24	23	22	21	20	19	18	17	16	15	14

demanda período 4 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	15	16	17	18	19	20	21	22	23	24	25	26	27
I2	13	12	11	10	9	8	7	6	5	4	3	2	1

Prueba 5 - Modelo Estocástico - Algoritmos Genéticos

2 Productos, 5 Periodos, 2 Posibilidades - los escenarios son 16

KelabM- 100000000, KenvM - 100000000.

Producto	Cres	Cins	Kelab	Kenv
I1	1000	7000	200	30
I2	1000	7000	15	250

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	16	8	4	2	1
I2	6	2	8	4	10

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	16	32	64	128	256
I2	6	2	8	4	10

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	1	2	4	8	16
I2	1	7	3	9	5

probabilidades

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

0.0606	0.0606	0.0606	0.0656	0.0656	0.0606	0.0606	0.0656	0.0656	0.0606	0.0607	0.0607	0.0607	0.0607	0.0656	0.0656

demanda período 1 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
I2	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6

demanda período 2 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	12	12	12	12	12	12	12	12
I2	10	10	10	10	10	10	10	10	5	5	5	5	5	5	5	5

demanda período 3 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	7	7	7	7	5	5	5	5	13	13	13	13	4	4	4	4
I2	8	8	8	8	13	13	13	13	5	5	5	5	20	20	20	20

demanda período 4 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	15	15	2	2	6	6	8	8	3	3	14	14	7	7	10	10
I2	3	3	9	9	10	10	2	2	3	3	5	5	13	13	9	9

demanda período 5 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
I2	2	4	6	8	10	12	14	16	1	3	5	7	9	11	13	15

Prueba 6 - Modelo Estocástico - Algoritmos Genéticos

2 Productos, 5 Periodos, 2 Posibilidades - los escenarios son 16

KelabM- 100000000, KenvM - 100000000.

Producto	Cres	Cins	Kelab	Kenv
I1	1000	7000	200	30
I2	1000	7000	15	250

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	16	8	4	2	1
I2	6	2	8	4	10

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	16	32	64	128	256
I2	6	2	8	4	10

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4	Periodo 5
I1	1	2	4	8	16
I2	1	7	3	9	5

probabilidades

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0.0925	0.0425	0.0425	0.0725	0.0725	0.0425	0.0425	0.0925	0.0925	0.0425	0.0425	0.0725	0.0725	0.0425	0.0425	0.0925

demanda período 1 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9	9
I2	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6	6

demanda período 2 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	8	8	8	8	8	8	8	8	12	12	12	12	12	12	12	12
I2	10	10	10	10	10	10	10	10	5	5	5	5	5	5	5	5

demanda período 3 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	7	7	7	7	5	5	5	5	13	13	13	13	4	4	4	4
I2	8	8	8	8	13	13	13	13	5	5	5	5	20	20	20	20

demanda período 4 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	15	15	2	2	6	6	8	8	3	3	14	14	7	7	10	10
I2	3	3	9	9	10	10	2	2	3	3	5	5	13	13	9	9

demanda período 5 según escenarios

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
I1	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
I2	2	4	6	8	10	12	14	16	1	3	5	7	9	11	13	15

Prueba 7 - Modelo Estocástico - Algoritmos Genéticos

2 Productos, 4 Periodos, 3 Posibilidades - los escenarios son 27

KelabM- 100000, KenvM - 100000, Prob[s] = 0,037 para cada escenario s.

Producto	Cres	Cins	Kelab	Kenv
I1	4	500000	40	60
I2	7000000	900	30	70

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	20	40	80	16
I2	15	20	40	80

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	10	30	40	80

I2	20	45	80	16

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	0	1000	100	10
I2	0	1	10	100

probabilidades de los escenarios del 1 al 14

1	2	3	4	5	6	7	8	9	10	11	12	13	14
0.100	0.005	0.006	0.125	0.005	0.006	0.100	0.005	0.006	0.075	0.005	0.006	0.100	0.006

probabilidades de los escenarios del 15 al 27

15	16	17	18	19	20	21	22	23	24	25	26	27
0.006	0.125	0.005	0.006	0.100	0.005	0.006	0.075	0.005	0.006	0.100	0.005	0.006

demanda período 1 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	8	8	8	8	8	8	8	8	8	8	8	8	8	8
I2	9	9	9	9	9	9	9	9	9	9	9	9	9	9

demanda período 1 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	8	8	8	8	8	8	8	8	8	8	8	8	8
I2	9	9	9	9	9	9	9	9	9	9	9	9	9

demanda período 2 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	9	9	9	9	9	9	9	9	9	6	6	6	6	6
I2	3	3	3	3	3	3	3	3	3	6	6	6	6	6

demanda período 2 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	6	6	6	6	3	3	3	3	3	3	3	3	3

I2	6	6	6	6	9	9	9	9	9	9	9	9	9

demanda período 3 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	11	11	11	22	22	22	33	33	33	44	44	44	55	55
I2	32	32	32	16	16	16	8	8	8	4	4	4	20	20

demanda período 3 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	55	66	66	66	77	77	77	88	88	88	99	99	99
I2	20	4	4	4	8	8	8	16	16	16	32	32	32

demanda período 4 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I2	27	26	25	24	23	22	21	20	19	18	17	16	15	14

demanda período 4 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	15	16	17	18	19	20	21	22	23	24	25	26	27
I2	13	12	11	10	9	8	7	6	5	4	3	2	1

Prueba 8 - Modelo Estocástico - Algoritmos Genéticos

2 Productos, 4 Periodos, 3 Posibilidades - los escenarios son 27

KelabM- 100000, KenvM - 100000, Prob[s] = 0,037 para cada escenario s.

Producto	Cres	Cins	Kelab	Kenv
I1	4	500000	40	60
I2	7000000	900	30	70

Celab	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	20	40	80	16
I2	15	20	40	80

Cenv	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	10	30	40	80
I2	20	45	80	16

Calm	Periodo 1	Periodo 2	Periodo 3	Periodo 4
I1	0	1000	100	10
I2	0	1	10	100

probabilidades de los escenarios del 1 al 14

1	2	3	4	5	6	7	8	9	10	11	12	13	14
0.01	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.49

probabilidades de los escenarios del 15 al 27

15	16	17	18	19	20	21	22	23	24	25	26	27
0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02	0.02

demanda período 1 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	8	8	8	8	8	8	8	8	8	8	8	8	8	8
I2	9	9	9	9	9	9	9	9	9	9	9	9	9	9

demanda período 1 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	8	8	8	8	8	8	8	8	8	8	8	8	8
I2	9	9	9	9	9	9	9	9	9	9	9	9	9

demanda período 2 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14

I1	9	9	9	9	9	9	9	9	9	6	6	6	6	6
I2	3	3	3	3	3	3	3	3	3	6	6	6	6	6

demanda período 2 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	6	6	6	6	3	3	3	3	3	3	3	3	3
I2	6	6	6	6	9	9	9	9	9	9	9	9	9

demanda período 3 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	11	11	11	22	22	22	33	33	33	44	44	44	55	55
I2	32	32	32	16	16	16	8	8	8	4	4	4	20	20

demanda período 3 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	55	66	66	66	77	77	77	88	88	88	99	99	99
I2	20	4	4	4	8	8	8	16	16	16	32	32	32

demanda período 4 según escenarios del 1 al 14

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I1	1	2	3	4	5	6	7	8	9	10	11	12	13	14
I2	27	26	25	24	23	22	21	20	19	18	17	16	15	14

demanda período 4 según escenarios del 15 al 27

	15	16	17	18	19	20	21	22	23	24	25	26	27
I1	15	16	17	18	19	20	21	22	23	24	25	26	27
I2	13	12	11	10	9	8	7	6	5	4	3	2	1

Apéndice B: Código

Modelo Determinístico - GLPK

```
set PRODUCTOS;
set PERIODOS;


/*Declaracion de Variables*/

var X{i in PRODUCTOS,t in PERIODOS} >= 0; /* cantidad a producir del producto i en el período t*/
var S{i in PRODUCTOS,t in PERIODOS} >= 0; /* cantidad en stock del producto i en el período t*/
var R{i in PRODUCTOS,t in PERIODOS} >= 0; /* demanda insatisfecha del producto i en el período t*/
var Y{i in PRODUCTOS,t in PERIODOS},binary; /* indica producción del producto i en el período t*/
var U{i in PRODUCTOS,t in PERIODOS},binary; /* indica cambio del producto i en el período t*/


/* defino datos */

param dem{i in PRODUCTOS,t in PERIODOS}; /* demanda del producto i en el período t */
param Cenv{i in PRODUCTOS, t in PERIODOS}; /* Costo de Envasado del producto i en el período t */
param Celab{i in PRODUCTOS, t in PERIODOS}; /* Costo de Elaboración del producto i en el período t */
param Calm{i in PRODUCTOS, t in PERIODOS}; /* Costo de Almacenamiento del producto i en el
período t */
param Cres{i in PRODUCTOS}; /* Costo de Restablecimiento del producto i */
param Cins{i in PRODUCTOS}; /* Costo de insuficiencia del producto i */
param Kelab{i in PRODUCTOS}; /* Capacidad de Elaboración del producto i */
param Kenv{i in PRODUCTOS}; /* Capacidad de Envasado del producto i */

param KenvM; /* Capacidad Máxima de Envasado */
param KelabM; /* Capacidad Máxima de Elaboración */


/*Definicion de la funcion*/

minimize z:
(sum{i in PRODUCTOS,t in PERIODOS:t > 0} ((Cenv[i,t] + Celab[i,t])*X[i,t])) +
(sum{i in PRODUCTOS,t in PERIODOS:t > 0} (Calm[i,t]*S[i,t])) +
(sum{i in PRODUCTOS,t in PERIODOS:t > 0} (Cres[i]*U[i,t])) +
(sum{i in PRODUCTOS,t in PERIODOS:t > 0} (Cins[i]*R[i,t]))
;
```

```
/*Restricciones*/


1) s.t. capacidadEnvasado{t in PERIODOS: t > 0}:sum{i in PRODUCTOS} (Kenv[i]*X[i,t]) <= KenvM;
/* no se envasa mas de la capacidad máxima de envasado */
2) s.t. capacidadElaborado{t in PERIODOS: t > 0}:sum{i in PRODUCTOS} (Kelab[i]*X[i,t]) <= KelabM;
/* no se elabora mas de la capacidad máxima de elaboración */
3) s.t. demanda{i in PRODUCTOS,t in PERIODOS : t > 0}:X[i,t]+S[i,t-1]-R[i,t-1]=dem[i,t]+S[i,t]-R[i,t];
/* ecuacion de balance entre las variables */
4) s.t. productoProducido1{i in PRODUCTOS,t in PERIODOS: t > 0}:X[i,t] <= Y[i,t]* (sum{x in PERIODOS:
x > 0} (dem[i,x]));
/* la producción del producto i en el período t es menor que la demanda para el producto i de la
sumatoria hasta el final */
5) s.t. productoProducido3{t in PERIODOS: t > 0}:sum{i in PRODUCTOS} Y[i,t] <= 1;
/* solo se puede producir un producto en el períodos t */
6) s.t. cambioArticulo1{i in PRODUCTOS,t in PERIODOS : t > 0}:U[i,t]>=Y[i,t]-Y[i,t-1];
/* indica que hubo producción del producto i en el período t*/
7) s.t. stockInicialVacio{i in PRODUCTOS}: S[i,0]=0;
/* stock inicial es cero */
8) s.t. prodInicialVacio{i in PRODUCTOS}: Y[i,0]=0;
/* se setea en cero al inicio */
9) s.t. sinDemandaInsatisfechaInicial{i in PRODUCTOS}: R[i,0]=0;
/* demanda insatisfecha inicial es cero */
10) s.t. totalDemanda{i in PRODUCTOS}:sum{t in PERIODOS: t > 0} (X[i,t]) >= sum{t in PERIODOS: t >
0} (dem[i,t]);
/* se satisface la demanda de todos los periodos */

end;
```

Modelo Estocástico - GLPK

```
set PRODUCTOS;
set PERIODOS;
set ESCENARIOS;

/*Declaracion de Variables*/

var X{p in PRODUCTOS,t in PERIODOS, s in ESCENARIOS} >= 0; /* cantidad a producir del producto p,
en el periodo t, en el escenario s */
var S{p in PRODUCTOS,t in PERIODOS, s in ESCENARIOS} >= 0; /* cantidad en stock del producto p,
en el periodo t, en el escenario s */
var R{p in PRODUCTOS,t in PERIODOS, s in ESCENARIOS} >= 0; /* demanda insatisfecha del producto
p, en el periodo t, en el escenario s*/
var Y{p in PRODUCTOS,t in PERIODOS, s in ESCENARIOS},binary; /* indica producción del producto p,
en el periodo t, en el escenario s */
var U{p in PRODUCTOS,t in PERIODOS, s in ESCENARIOS},binary; /* indica cambio del producto p, en
el periodo t, en el escenario s */
```

```
/* defino datos */

param dem{p in PRODUCTOS, s in ESCENARIOS,t in PERIODOS}; /* demanda del producto p, en el
periodo t, en el escenario s */
param Cenv{p in PRODUCTOS, t in PERIODOS}; /* costo de envasado del producto p, en el periodo t */
param Celab{p in PRODUCTOS, t in PERIODOS}; /* Costo de Elaboración del producto p en el período t
*/
param Calm{p in PRODUCTOS, t in PERIODOS}; /* Costo de Almacenamiento del producto p en el
período t */
param Cres{p in PRODUCTOS}; /* Costo de Restablecimiento del producto p */
param Cins{p in PRODUCTOS}; /* Costo de insuficiencia del producto p */
param Kelab{p in PRODUCTOS}; /* Capacidad de Elaboración del producto p */
param Kenv{p in PRODUCTOS}; /* Capacidad de Envasado del producto p */
param matriz{s in ESCENARIOS, sp in ESCENARIOS, t in PERIODOS}; /* */
param prob{s in ESCENARIOS}; /* Probabilidades de los escenarios s */

param KenvM; /* Capacidad Máxima de Envasado */
param KelabM; /* Capacidad Máxima de Elaboración */

/*Definicion de la funcion*/

minimize z:
sum{s in ESCENARIOS}(sum{p in PRODUCTOS,t in PERIODOS:t > 0 } ((Cenv[p,t] + Celab[p,t])*X[p,t,s]
+ Calm[p,t]*S[p,t,s]+ Cres[p]*U[p,t,s] + Cins[p]*R[p,t,s])*prob[s]);

/*Restricciones generales*/

1) s.t. capacidadEnvasado{s in ESCENARIOS, t in PERIODOS: t > 0}:sum{p in PRODUCTOS}
(Kenv[p]*X[p,t,s]) <= KenvM;
/* no se supera la capacidad de envasado máxima */
2) s.t. capacidadElaborado{s in ESCENARIOS, t in PERIODOS: t > 0}:sum{p in PRODUCTOS}
(Kelab[p]*X[p,t,s]) <= KelabM;
/* no se supera la capacidad de elaboración máxima */
3) s.t. productoProducido{ s in ESCENARIOS,t in PERIODOS: t > 0}:sum{p in PRODUCTOS} Y[p,t,s] <=
1;
/* solo se puede producir un producto en el período t y escenario s */
4) s.t. stockInicialVacio{p in PRODUCTOS, s in ESCENARIOS}: S[p,0,s]=0;
/* stock cero para todos los productos, escenarios y periodo inicial */
5) s.t. prodInicialVacio{p in PRODUCTOS, s in ESCENARIOS}: Y[p,0,s]=0;
/* la variable es cero en todos los productos, escenarios y periodo inicial */
6) s.t. insInicialVacio{p in PRODUCTOS, s in ESCENARIOS}: R[p,0,s]=0;
/* demanda insatisfecha incial cero */
7) s.t. topeProducido{p in PRODUCTOS,s in ESCENARIOS,t in PERIODOS: t > 0}:X[p,t,s] <=
Y[p,t,s]*( (sum{x in PERIODOS: x > 0} (dem[p,s,x])));
/* no se puede producir más de la demanda pendiente */
8) s.t. totalDemanda{p in PRODUCTOS, s in ESCENARIOS}:sum {t in PERIODOS: t > 0} (X[p,t,s]) >=
sum{t in PERIODOS: t > 0} (dem[p,s,t]);
```

```
/* lo producido es mayoy a igual que la sumatoria de la demanda de todos los períodos */
9) s.t. demanda{p in PRODUCTOS, s in ESCENARIOS, t in PERIODOS: t > 0}:X[p,t,s]+S[p,t-1,s]-R[p,t-
1,s]=dem[p,s,t]+S[p,t,s]-R[p,t,s];
/* ecuación de balance entre variables y parámetros */
10) s.t. cambioArticulo{p in PRODUCTOS,s in ESCENARIOS, t in PERIODOS : t > 0 }:U[p,t,s] >= Y[p,t,s]-
Y[p,t-1,s];
/* indica que hubo producción del producto p en el período t y escenario s */
11) s.t. noanticipatividad_X{p in PRODUCTOS, s in ESCENARIOS, t in PERIODOS:t>0}: sum{sp in
ESCENARIOS}matriz[s,sp,t] * X[p,t,sp] = (sum{sp in ESCENARIOS} matriz[s,sp,t] ) * X[p,t,s];
/* restricciones de no anticipatividad */
```

Modelo Estocástico - Algoritmos Genéticos

newGAstructures.hh

```
#ifndef INC_newGA_mallba_hh
#define INC_newGA_mallba_hh

#include <iostream>
#include <fstream>
#include <math.h>
#include <values.h>
#include <vector>
#include <string>
#include <Mallba/Rlist.h>
#include <Mallba/Rarray.h>
#include <Mallba/Matrix.hh>
#include <Mallba/Messages.h>
#include <Mallba/mallba.hh>
#include <Mallba/States.hh>
#include <Mallba/random.hh>
#include <Mallba/time.hh>
#include <Mallba/netstream.hh>
#include <assert.h>

using namespace std;

#ifndef _INDIVIDUAL_
#define _INDIVIDUAL_

struct individual // index of a individual in the population and its fitness
{
        int    index;
        double fitness;
```

```
        double sel_parameter;
        bool   change;
};

#endif
#endif
```

newGA.hh

```
#ifndef INC_newGA
#define INC_newGA
#include "newGAstructures.hh"


skeleton newGA
{
// Si se definen más de 5 nuevos operadores por parte del usuario, se debe cambiar esta
constante.
#define MAX_OP_USER 5
// Si se algún operador tiene más de 5 parámetros se debe modificar esta variable
#define MAX_PROB_PER_OP 5

  provides class SetUpParams;
  provides class Statistics;
  provides class Population;
  provides class Inter_Operator;
  provides class Migration;
  provides class Selection;
  provides class Selection_Tournament;
  provides class Selection_Roulette_Wheel;
  provides class Selection_Rank;
  provides class Selection_Best;
  provides class Selection_Worst;
  provides class Operator_Pool;
  provides class Solver;
  provides class Solver_Seq;
  provides class Solver_Lan;
  provides class Solver_Wan;
  provides class StopCondition;

  requires class Problem;
  requires class Solution;
  requires class UserStatistics;
  requires class Intra_Operator;
```

```
  requires class Crossover;
  requires class Mutation;
  requires class StopCondition_1;
  requires bool terminateQ (const Problem& pbm, const Solver& solver, const SetUpParams&
setup);

// Problem ----------------------------------------------------------------------

  requires class Problem
  {
        private:
              int productos;
              int periodos;
              int posibilidades;
              int escenarios;
              int KelabM;
              int KenvM;
              vector<int> Cres;
              vector<int> Cins;
              vector<int> Kelab;
              vector<int> Kenv;
              vector<int> cortes;
              vector<float> probabilidades;
              Matrix<int> Celab;
              Matrix<int> Cenv;
              Matrix<int> Calm;
              vector<Matrix<int> > demanda;
        public:
              Problem();
              ~Problem();

              friend ostream& operator<< (ostream& os, const Problem& pbm);
              friend istream& operator>> (istream& is, Problem& pbm);

              Problem& operator=  (const Problem& pbm);
              bool operator== (const Problem& pbm) const;
              bool operator!= (const Problem& pbm) const;

              Direction direction () const;

              int getProductos() const;
              int getPeriodos() const;
              int getPosibilidades() const;
              int getEscenarios() const;
```

```
                int getKelabM() const;
                int getKenvM() const;
                vector<int> getCres() const;
                vector<int> getCins() const;
                vector<int> getKelab() const;
                vector<int> getKenv() const;
                vector<int> getCortes() const;
                vector<float> getProbabilidades() const;
                Matrix<int> getCelab() const;
                Matrix<int> getCenv() const;
                Matrix<int> getCalm() const;
                vector<Matrix<int> > getDemanda() const;
                int demandaTope(int prod, int esc) const;
 };

//Solution ----------------------------------------------------------------------

 requires class Solution
 {
        public:
                Solution (const Problem& pbm);
                Solution (const Solution& sol);
                ~Solution();

                friend ostream& operator<< (ostream& os, const Solution& sol);
                friend istream& operator>> (istream& is, Solution& sol);
                friend NetStream& operator << (NetStream& ns, const Solution& sol);
                friend NetStream& operator >> (NetStream& ns, Solution& sol);

                const Problem& pbm() const;

                Solution& operator=  (const Solution& sol);
                bool operator== (const Solution& sol) const;
                bool operator!= (const Solution& sol) const;

                char *to_String() const;
                void to_Solution(char *_cadena_);

                void initialize();
                double fitness ();
                unsigned int size() const;

                Matrix<int> getMatrizX() const;
                void setMatrizX(Matrix<int> x);
```

```
        private:
                const Problem& _pbm;
                Matrix<int> matrizX;
 };

// UserStatistics ------------------------------------------------------------------------

  requires class UserStatistics
  {
        private:
                struct user_stat
                {
                        unsigned int trial;
                        unsigned long nb_evaluation_best_found_trial;
                        unsigned long nb_iteration_best_found_trial;
                        double best_cost_trial;
                        double worst_cost_trial;
                        float time_best_found_trial;
                        float time_spent_trial;
                };

                Rlist<struct user_stat> result_trials;

        public:
                UserStatistics ();
                ~UserStatistics();

                friend ostream& operator<< (ostream& os, const UserStatistics& usertats);

        UserStatistics& operator= (const UserStatistics& userstats);
                void update(const Solver& solver);
                void clear();
 };

// Intra_Operator ( clase abstracta ) ----------------------------------------------------------

  requires class Intra_Operator
  {
        protected:
                unsigned int _number_operator;
                float *probability;

        public:
```

```
                Intra_Operator(const unsigned int _number_op);
                virtual ~Intra_Operator();

                static Intra_Operator *create(const unsigned int _number_op);
                friend ostream& operator<< (ostream& os, const Intra_Operator& intra);

                virtual void execute(Rarray<Solution*>& sols) const=0;
                virtual void setup(char line[MAX_BUFFER]) = 0;
                unsigned int number_operator() const;

                virtual void RefreshState(const StateCenter& _sc) const=0;
                virtual void UpdateFromState(const StateCenter& _sc)=0;
 };

// Crossover  ------------------------------------------------------------------------------

 requires class Crossover: public Intra_Operator
 {
        public:
                Crossover();
                virtual ~Crossover();

                friend ostream& operator << (ostream& os, const Crossover&  cross);

                void cross(Solution &sol1,Solution &sol2) const;
                virtual void execute(Rarray<Solution*>& sols) const;
                virtual void setup(char line[MAX_BUFFER]);

                virtual void RefreshState(const StateCenter& _sc) const;
                virtual void UpdateFromState(const StateCenter& _sc);
 };

// Mutation ---------------------------------------------------------------------------------

 requires class Mutation: public Intra_Operator
 {
        public:
                Mutation();
                virtual ~Mutation();

                friend ostream& operator<< (ostream& os, const Mutation&  mutation);

                void mutate(Solution& sol) const;
                // applies mutation over all solutions in array sols
```

```
                virtual void execute(Rarray<Solution*>& sols) const;
                virtual void setup(char line[MAX_BUFFER]);

                virtual void RefreshState(const StateCenter& _sc) const;
                virtual void UpdateFromState(const StateCenter& _sc);

 };

// StopCondition ------------------------------------------------------------------------
 provides class StopCondition
 {
        public:
                StopCondition();
                virtual bool EvaluateCondition(const Problem& pbm,const Solver& solver,const
SetUpParams& setup)=0;
                ~StopCondition();
 };

// StopCondition_1 {subclase------------------------------------------------------------------------

 requires class StopCondition_1 : public StopCondition
 {
        public:
                StopCondition_1();
                virtual bool EvaluateCondition(const Problem& pbm,const Solver& solver,const
SetUpParams& setup);
                ~StopCondition_1();
 };
// SetUpParams ------------------------------------------------------------------------

 provides class SetUpParams
 {
        private:
                unsigned int    _independent_runs;
                unsigned long   _nb_evolution_steps;
                unsigned int    _population_size;               // number of individuals
                unsigned int    _population_additional_size;  // size of offspring in each generation
                bool            _combine;                       // combines parents and offsprings
to select new parents ?
                bool            _display_state;

                unsigned long    _refresh_global_state;
                bool            _synchronized;
                unsigned int    _check_asynchronous;
```

```
        // selection of parents and offsprings
        mutable unsigned int _select_parents;
        mutable unsigned int _select_offsprings;
        mutable unsigned int _parameter_select_parents;
        mutable unsigned int _parameter_select_offsprings;

        Rlist<unsigned int> _intra_operators;
        Rlist<unsigned int> _inter_operators;

        Operator_Pool& _pool;

public:
        SetUpParams (Operator_Pool& pool);
        Operator_Pool& pool() const;

        friend ostream& operator<< (ostream& os, const SetUpParams& setup);
        friend istream& operator>> (istream& is, SetUpParams& setup);

        const unsigned int    independent_runs() const;
        const unsigned long    nb_evolution_steps() const;
        const unsigned int  population_size() const;
        const unsigned int  population_additional_size() const;
        const bool combine() const;
        const bool display_state() const;
        const unsigned long refresh_global_state() const;
        const bool synchronized() const;
        const unsigned int check_asynchronous() const;

        void independent_runs(const unsigned int val);
        void nb_evolution_steps(const unsigned long val);
        void population_size(const unsigned int val);
        void population_additional_size(const unsigned int val);
        void combine(const bool val);
        void display_state(const bool val);
        void refresh_global_state(const unsigned long val);
        void synchronized(const bool val);
        void check_asynchronous(const unsigned int val);

      // gets the i-th operator of inter-population
      const unsigned int  inter_operator_index(const unsigned int index) const;
        const unsigned int  inter_operators_size() const;

      // gets the i-th operator of intra-population
```

```
        const unsigned int intra_operator_index(const unsigned int index) const;
        const unsigned int intra_operators_size() const;

        const unsigned int select_parents() const;
        const unsigned int select_offsprings() const;
        const unsigned int parameter_select_parents() const;
        const unsigned int parameter_select_offsprings() const;

        void select_parents(const unsigned int val);
        void select_offsprings(const unsigned int val);
        void parameter_select_parents(const unsigned int val);
        void parameter_select_offsprings(const unsigned int val);

        void RefreshState(const StateCenter& _sc) const;
        void UpdateFromState(const StateCenter& _sc) const;

        ~SetUpParams();
};

// Statistics ------------------------------------------------------------------------

provides class Statistics
{
    private:
        struct stat
        {
            unsigned int trial;
            unsigned long nb_generation;
            unsigned long nb_evaluation;
            double best_cost;
            double global_best_cost;
            double average_cost;
            double standard_deviation;
        };

        Rlist<struct stat> stats_data;

    public:
        Statistics();
        ~Statistics();

        friend ostream& operator<< (ostream& os, const Statistics& stats);

        Statistics& operator= (const Statistics& stats);
```

```
                void update(const Solver& solver);
                void clear();
 };

// Population ---------------------------------------------------------------------

  provides class Population
  {
        private:
                Rarray<Solution*> _parents;     // individuals in population
                Rarray<Solution*> _offsprings;  // offsprings of current population
                Rarray<Solution*> _new_parents; // individuals of previous population
                Rarray<struct individual> _fitness_values;
                Rarray<struct individual> _fitness_aux;
                const SetUpParams& _setup;
                unsigned int _upper_cost,_lower_cost; // lower and upper fitness of individuals in
population
                unsigned long _evaluations;
                double _average_cost;

        public:
                inline void Evaluate(Solution *sol, struct individual &_f);

                Population(const Problem& pbm,const SetUpParams& setup); // crea un array de
objetos population;
                ~Population();

                friend ostream& operator<< (ostream& os, const Population& population);
                friend istream& operator>> (istream& is, Population& population);
                Population& operator= (const Population& pop);
                const SetUpParams& setup() const;
                void initialize();

                // Generate a new pool of individuals in population
                void evolution();

                // interchange solutions between island
                void interchange(const unsigned long current_generation, NetStream& channel);

                // creates a array with fitness of all individuals in population and its position in the
population
        void evaluate_parents();
```

```
                // creates a array with fitness of all individuals and offsprings in population and its
position in the population
                void evaluate_offsprings();

                // selects parents to creates offsprings
                void select_parents();

                // selects individuals for the new population
                void select_offsprings();

                const Rarray<Solution*>& parents() const;
                const Rarray<Solution*>& offsprings() const;
                Rarray<struct individual>& fitness_values();

                unsigned int upper_cost() const;
                unsigned int lower_cost() const;
                unsigned int evaluations() const;
                Solution& solution(const unsigned int index) const;
                double fitness(const unsigned int index) const;

                double best_cost() const;
                double worst_cost() const;
                Solution& best_solution() const;
                Solution& worst_solution() const;
                double average_cost() const;
                double standard_deviation() const;
 };

// Inter_Operator ( abstract )-------------------------------------------------------

  provides class Inter_Operator
  {
        protected:
                unsigned int migration_rate;
                unsigned int migration_size;
                unsigned int migration_selection_1;
                unsigned int migration_selection_2;
                unsigned int migration_selection_conf_1;
                unsigned int migration_selection_conf_2;

                unsigned int _number_operator;
                const Direction direction;

        public:
```

```
                Inter_Operator(const unsigned int _number_op, const Direction dir);
                virtual ~Inter_Operator();

                friend ostream& operator<< (ostream& os, const Inter_Operator& inter);

                virtual void execute(Population& pop,const unsigned long
current_generation,NetStream& _netstream,const bool synchronized,const unsigned int
check_asyncrhonous) const=0;
                virtual void setup(char line[MAX_BUFFER]);
                unsigned int number_operator() const;

                virtual void RefreshState(const StateCenter& _sc) const;
                virtual void UpdateFromState(const StateCenter& _sc);
 };

// Migration: public Inter_Operator ------------------------------------------------------

 provides class Migration: public Inter_Operator
 {
        public:
                Migration(const Direction dir);
                virtual ~Migration();

                friend ostream& operator<< (ostream& os, const Migration& migration);

                virtual void execute(Population& pop,const unsigned long
current_generation,NetStream& _netstream,const bool synchronized,const unsigned int
check_asyncrhonous) const;
 };

// Selection ( Makes a random selection ) ---------------------------------------

 provides class Selection
 {
        protected:
                unsigned int _number_selection;
                const Direction direction;

        public:

                Selection(const Direction dir);
                Selection(const unsigned int _number_sel, const Direction dir);
                virtual ~Selection();
```

```
                friend ostream& operator<< (ostream& os, const Selection& sel);

        virtual void prepare(Rarray<struct individual>& fitness_values,const bool remplace); //
const;
                virtual struct individual select_one(const Rarray<Solution*>& to_select_1,const
Rarray<Solution*>& to_select_2,const Rarray<struct individual>& fitness_values,const unsigned
int dummy,const bool remplace) const;
                unsigned int number_selection() const;
  };

// Selection_Tournament  ---------------------------------------------------------------------------

  provides class Selection_Tournament: public Selection
  {
        public:
                Selection_Tournament(const Direction dir);
                virtual ~Selection_Tournament();

                friend ostream& operator<< (ostream& os, const Selection_Tournament& sel);

                virtual struct individual select_one(const Rarray<Solution*>& to_select_1,const
Rarray<Solution*>& to_select_2,const Rarray<struct individual>& fitness_values,const unsigned
int tourment_size,const bool remplace) const;
  };

// Selection_Roulette_Wheel  -----------------------------------------------------------------------

  provides class Selection_Roulette_Wheel: public Selection
  {
        public:
                Selection_Roulette_Wheel(const Direction);
                virtual ~Selection_Roulette_Wheel();

                friend ostream& operator<< (ostream& os, const Selection_Roulette_Wheel&
sel);

        virtual void prepare(Rarray<struct individual>& fitness_values,const bool remplace); //
const;
                virtual struct individual select_one(const Rarray<Solution*>& to_select_1,const
Rarray<Solution*>& to_select_2,const Rarray<struct individual>& fitness_values,const unsigned
int dummy,const bool remplace) const;
  };

// Selection_Rank  ---------------------------------------------------------------------------
```

```
  provides class Selection_Rank: public Selection
  {
        public:
              Selection_Rank(const Direction dir);
              Selection_Rank(const unsigned int _number_sel, const Direction dir);
              virtual ~Selection_Rank();

              friend ostream& operator<< (ostream& os, const Selection_Rank& sel);

            virtual void prepare(Rarray<struct individual>& fitness_values,const bool remplace); //
const;
              virtual void reset();

              virtual struct individual select_one(const Rarray<Solution*>& to_select_1,const
Rarray<Solution*>& to_select_2,const Rarray<struct individual>& fitness_values,const unsigned
int portion,const bool remplace) const;
  };

// Selection_Best  ----------------------------------------------------------------------------

  provides class Selection_Best: public Selection_Rank
  {
        private:
              mutable unsigned int selection_best_position;

        public:
              Selection_Best(const Direction);
              virtual ~Selection_Best();

              friend ostream& operator<< (ostream& os, const Selection_Best& sel);

              virtual void reset();
        virtual struct individual select_one(const Rarray<Solution*>& to_select_1,const
Rarray<Solution*>& to_select_2,const Rarray<struct individual>& fitness_values,const unsigned
int position,const bool remplace) const;
  };

// Selection_Worst  ---------------------------------------------------------------------------

  provides class Selection_Worst: public Selection_Rank
  {
        private:
              mutable unsigned int selection_worst_position;
```

```
        public:
                Selection_Worst(const Direction);
                virtual ~Selection_Worst();

                friend ostream& operator<< (ostream& os, const Selection_Worst& sel);

                virtual void reset();
                virtual struct individual select_one(const Rarray<Solution*>& to_select_1,const
Rarray<Solution*>& to_select_2,const Rarray<struct individual>& fitness_values,const unsigned
int position,const bool remplace) const;
  };

// Operator_Pool ---------------------------------------------------------------

 // pool with all operators and selections that can be chosen in the setup file
  provides class Operator_Pool
  {
        private:
                mutable Rlist<Intra_Operator>          _intra_operators;
                Rlist<Selection>         _selectors;
                Rlist<Inter_Operator>            _inter_operators;

        public:
                Operator_Pool(const Problem& pbm);
                ~Operator_Pool();

                Intra_Operator& intra_operator(const unsigned int index) const;
                Rlist<Intra_Operator>& intra_operators() const;
                Selection& selector(const unsigned int index) const;
                const Rlist<Selection>& selectors() const;
                Inter_Operator& inter_operator(const unsigned int index) const;
                const Rlist<Inter_Operator>& inter_operators() const;

  };

// Solver  ---------------------------------------------------------------------

  provides class Solver
  {
        protected:
                const Problem&     problem;
                const SetUpParams& params;
                UserStatistics _userstat;
```

```
Statistics     _stat;
Population current_population;
StateCenter _sc;

double              best_cost;
double              worst_cost;
Solution   best_solution;
double     average_cost;
double     standard_deviation;
float      total_time_spent;
float      time_spent_in_trial;
float       start_trial;
float       start_global;

bool        _end_trial;

State_Vble  _current_trial;
State_Vble  _current_iteration;
State_Vble  _current_evaluations;

State_Vble  _current_best_solution;
State_Vble  _current_best_cost;
State_Vble  _current_worst_cost;
State_Vble  _current_average_cost;
State_Vble  _current_standard_deviation;
State_Vble  _current_time_spent;

State_Vble  _best_solution_trial;
State_Vble  _best_cost_trial;
State_Vble  _worst_cost_trial;
State_Vble  _iteration_best_found_in_trial;
State_Vble  _evaluations_best_found_in_trial;
State_Vble  _time_best_found_trial;
State_Vble  _time_spent_trial;

State_Vble  _trial_best_found;
State_Vble  _iteration_best_found;
State_Vble  _evaluations_best_found;
State_Vble  _global_best_solution;
State_Vble  _global_best_cost;
State_Vble  _global_worst_cost;
State_Vble  _time_best_found;
```

```
        State_Vble _crossover_probability; // probability of applying the operator over
population
        State_Vble _mutation_probability; // probability of applying the operator over
population
        State_Vble _user_op_probability[MAX_OP_USER]; // probabilities of user
operators
        State_Vble _migration_rate;
        State_Vble _migration_size;
        State_Vble _migration_selection_1;
        State_Vble _migration_selection_2;
        State_Vble _migration_selection_conf_1;
        State_Vble _migration_selection_conf_2;
        State_Vble _select_parents;
        State_Vble _select_offsprings;
        State_Vble _parameter_select_parents;
        State_Vble _parameter_select_offsprings;

        State_Vble  _display_state;


    public:
        Solver (const Problem& pbm, const SetUpParams& setup);
        virtual ~Solver ();

        virtual int pid() const;
        bool end_trial() const;
        void end_trial(bool et);

        // Execution methods ----------------------------------------------------------

        // Full execution
        virtual void run () =0;
        virtual void run (const unsigned long int nb_generations) =0;
        virtual void run (const Population& pop,const unsigned long int nb_generations)
=0;

        //Partial execution
        virtual void StartUp()=0;
        virtual void StartUp(const Population& pop)=0;

        virtual void DoStep()=0;

        // Statistics handling --------------------------------------------------------
```

```
        Statistics& statistics();
        UserStatistics& userstatistics ();
        Population& population();
        const SetUpParams& setup() const;
        const Problem& pbm() const;

        // State handling ------------------------------------------------------------------------

        void RefreshState();
        void RefreshCfgState();
        void UpdateFromState();
        void UpdateFromCfgState();
        StateCenter* GetState();

        unsigned int current_trial() const;
        unsigned long current_iteration() const;
        unsigned long current_evaluations() const;
        Solution current_best_solution() const;
        double current_best_cost() const;
        double current_worst_cost() const;
        double current_average_cost() const;
        double current_standard_deviation() const;
        float  current_time_spent() const;
        Solution  best_solution_trial() const;
        double best_cost_trial() const;
        double worst_cost_trial() const;
        unsigned int iteration_best_found_in_trial() const;
        unsigned int evaluations_best_found_in_trial() const;
        float time_best_found_trial() const;
        float time_spent_trial() const;
        unsigned int trial_best_found() const;
        unsigned int iteration_best_found() const;
        unsigned int evaluations_best_found() const;
        Solution global_best_solution() const;
        double global_best_cost() const;
        double global_worst_cost() const;
        float time_best_found() const;
        int display_state() const;

        float *crossover_probability() const;
        float *mutation_probability() const;
        float *user_op_probability(const int index) const;
unsigned int migration_rate() const;
unsigned int migration_size() const;
```

```
unsigned int migration_selection_1() const;
unsigned int migration_selection_2() const;
unsigned int migration_selection_conf_1() const;
unsigned int migration_selection_conf_2() const;
        unsigned int select_parents() const;
        unsigned int select_offprings() const;
        unsigned int parameter_select_parents() const;
        unsigned int parameter_select_offsprings() const;

        void current_trial(const unsigned int value);
        void current_iteration(const unsigned long value);
        void current_evaluations(const unsigned long value);
        void current_best_solution(const Solution& sol);
        void current_best_cost(const double value);
        void current_worst_cost(const double value);
        void current_average_cost(const double value);
        void current_standard_deviation(const double value);
        void current_time_spent(const float value);
        void best_solution_trial(const Solution& sol);
        void best_cost_trial(const double value);
        void worst_cost_trial(const double value);
        void iteration_best_found_in_trial(const unsigned int value);
        void evaluations_best_found_in_trial(const unsigned int value);
        void time_best_found_trial(const float value);
        void time_spent_trial(const float value);
        void trial_best_found(const unsigned int value);
        void iteration_best_found(const unsigned int  value);
        void evaluations_best_found(const unsigned int  value);
        void global_best_solution(const Solution& sol);
        void global_best_cost(const double value);
        void global_worst_cost(const double value);
        void time_best_found(const float value);
        void display_state(const int value);

        void crossover_probability(const float *probability);
        void mutation_probability(const float *probability);
        void user_op_probability(const int index,const float *probability);
        void migration_rate(const unsigned int rate);
        void migration_size(const unsigned int size);
        void migration_selection_1(const unsigned int seleciton_1);
        void migration_selection_2(const unsigned int selection_2);
        void migration_selection_conf_1(const unsigned int selection_conf_1);
        void migration_selection_conf_2(const unsigned int selection_conf_2);
        void select_parents(const unsigned int selection);
```

```
                void select_offsprings(const unsigned int selection);
                void parameter_select_parents(const unsigned int value);
                void parameter_select_offsprings(const unsigned int value);

                void show_state() const;
                void KeepHistory(const Solution& best_sol,const double best_cost,const double
worst_cost,const float time_spent_trial,const float total_time_spent);
 };

 provides class Solver_Seq: public Solver
 {
        public:
                Solver_Seq ( const Problem& pbm, const SetUpParams& setup);
                virtual ~Solver_Seq ();

                // Execution methods ---------------------------------------------------------------

                // Full execution
                virtual void run ();
                virtual void run (const unsigned long int nb_generations);
                virtual void run (const Population& pop,const unsigned long int nb_generations);

                //Partial execution
                virtual void StartUp();
                virtual void StartUp(const Population& pop);

                virtual void DoStep();
 };

 provides class Solver_Lan: public Solver
 {
        private:
                NetStream _netstream;
                int mypid;

                int receive_local_state();

                unsigned int _current_trial;
                unsigned long _current_iteration;
                unsigned long _current_evaluations;
                double _best_cost_trial;
                Solution _best_solution_trial;
                double _worst_cost_trial;
                float _time_best_found_in_trial;
```

```
                unsigned long _iteration_best_found_in_trial;
                unsigned long _evaluations_best_found_in_trial;

                // Termination phase //
                bool final_phase;
                int acum_evaluations;
                int acum_iterations;

        public:
                Solver_Lan (const Problem& pbm, const SetUpParams& setup,int argc,char
**argv);
                virtual ~Solver_Lan ();
                virtual int pid() const;
                NetStream& netstream();

                // Execution methods ---------------------------------------------------------------

                // Full execution
                virtual void run ();
                virtual void run (const unsigned long int nb_generations);
                virtual void run (const Population& pop,const unsigned long int nb_generations);

                //Partial execution
                virtual void StartUp();
                virtual void StartUp(const Population& pop);

                virtual void DoStep();

                //Communication
                void send_local_state_to(int _mypid);
                void check_for_refresh_global_state();
                void reset();
 };

 provides class Solver_Wan: public Solver
 {
        private:
                NetStream _netstream;
                int mypid;

                int receive_local_state();

                unsigned int _current_trial;
                unsigned long _current_iteration;
```

```
                unsigned long _current_evaluations;
                double _best_cost_trial;
                Solution _best_solution_trial;
                double _worst_cost_trial;
                float _time_best_found_in_trial;
                unsigned long _iteration_best_found_in_trial;
                unsigned long _evaluations_best_found_in_trial;

                // Termination phase //
                bool final_phase;
                int acum_evaluations;
                int acum_iterations;

        public:
                Solver_Wan (const Problem& pbm, const SetUpParams& setup,int argc,char
**argv);
                virtual ~Solver_Wan ();
                virtual int pid() const;
                NetStream& netstream();

                // Execution methods ---------------------------------------------------------------

                // Full execution
                virtual void run ();
                virtual void run (const unsigned long int nb_generations);
                virtual void run (const Population& pop,const unsigned long int nb_generations);

                //Partial execution
                virtual void StartUp();
                virtual void StartUp(const Population& pop);

                virtual void DoStep();

                //Communication
                void send_local_state_to(int _mypid);
                void check_for_refresh_global_state();
                void reset();
  };

}

#endif
```

newGA.req.cc

```
#ifndef INC_REQ_newGA
#define INC_REQ_newGA

#include <math.h>
#include <string>
#include <stdio.h>
#include <stdlib.h>
#include <sstream>
#include "newGA.hh"

#define MAX_FITNESS 1000000000000

skeleton newGA
{

	// Problem ----------------------------------------------------------

	Problem::Problem ()
	{

	}

	ostream& operator<< (ostream& os, const Problem& pbm)
	{
		int i,j,k,largo;
		os << "Cantidad de productos: " << pbm.productos << endl;
		os << "Cantidad de períodos: " << pbm.periodos << endl;
		os << "Cantidad de posibilidades: " << pbm.posibilidades << endl;
		os << "Cantidad de escenarios: " << pbm.escenarios << endl;
		os << "Puntos de corte: ";
		largo=(pbm.cortes).size();
		for (i=0;i < largo;i++){
			os << pbm.cortes[i] << ",";
		}
		os << endl;
		os << "Probabilidades de cada escenario: ";
		os << endl;
		for (i=0;i < pbm.escenarios;i++){
			os << "Escenario " << i << ": " << pbm.probabilidades[i] << endl;
		}

		os << "Capacidades de elaboración: " << endl;
```

```
for (i=0;i < pbm.productos;i++){
        os << "Prod " << i << ": " << pbm.Kelab[i] << ",";
}
os << endl;
os << "Capacidad de elaboración Maxima: " << pbm.KelabM << endl;
os << "Capacidades de envasado: " << endl;
for (i=0;i < pbm.productos;i++){
        os << "Prod " << i << ": " << pbm.Kenv[i] << ",";
}
os << endl;
os << "Capacidad de envasado Maxima: " << pbm.KenvM << endl;
os << "Costo de cambio de artículo: " << endl;
for (i=0;i < pbm.productos;i++){
        os << "Prod " << i << ": " << pbm.Cres[i] << ",";
}
os << endl;
os << "Costo de demanda insuficiente: " << endl;
for (i=0;i < pbm.productos;i++){
        os << "Prod " << i << ": " << pbm.Cins[i] << ",";
}
os << endl;
os << "Costo de elaboración: " << endl;
for (j=0;j < pbm.periodos;j++){
        os << "Período " << j << ": ";
        for (i=0;i < pbm.productos;i++){
                os << "Prod " << i << ": " << pbm.Celab(i,j) << ",";
        }
        os << endl;
}
os << "Costo de envasado: " << endl;
for (j=0;j < pbm.periodos;j++){
        os << "Período " << j << ": ";
        for (i=0;i < pbm.productos;i++){
                os << "Prod " << i << ": " << pbm.Cenv(i,j) << ",";
        }
        os << endl;
}
os << "Costo de almacenamiento: " << endl;
for (j=0;j < pbm.periodos;j++){
        os << "Período " << j << ": ";
        for (i=0;i < pbm.productos;i++){
                os << "Prod " << i << ": " << pbm.Calm(i,j) << ",";
        }
        os << endl;
```

```
	}
	for (j=0;j < pbm.periodos;j++){
		os << "Demanda para el período " << j << ":" << endl;
		for (k=0;k < pbm.escenarios;k++){
			os << "Escenario " << k << ": ";
			for (i=0;i < pbm.productos;i++){
				os << "Prod " << i << ": " << pbm.demanda[i](k,j) << ",";
			}
			os << endl;
		}
	}
	return os;
}

istream& operator>> (istream& is, Problem& pbm)
{
	int i,j,k;
	is >> pbm.productos;
	is.ignore(256,'\n');
	is >> pbm.periodos;
	is.ignore(256,'\n');
	is >> pbm.posibilidades;
	is.ignore(256,'\n');
	pbm.escenarios = pow(pbm.posibilidades,pbm.periodos-1);

	int largo=0;
	int maxvalores=pbm.escenarios*pbm.periodos;

	for (i=0;i < pbm.periodos;i++){
		largo+=pow(pbm.posibilidades,i);
	}
	pbm.cortes.resize(largo);
	j=pbm.escenarios*pbm.posibilidades;
	k=0;
	for (i=0;i < maxvalores;i++){
		if (i % pbm.escenarios==0)
			j/=pbm.posibilidades;
		if (i % j==0){
			pbm.cortes[k]=i;
			k++;
		}
	}
	is >> pbm.KelabM;
	is.ignore(256,'\n');
```

```
is >> pbm.KenvM;
is.ignore(256,'\n');
is.ignore(256,'\n');
pbm.probabilidades.resize(pbm.escenarios);
for (i=0;i < pbm.escenarios;i++){
        is >> pbm.probabilidades[i];
}
is.ignore(256,'\n');
is.ignore(256,'\n');
pbm.Cres.resize(pbm.productos);
for (i=0;i < pbm.productos;i++){
        is >> pbm.Cres[i];
}
is.ignore(256,'\n');
is.ignore(256,'\n');
pbm.Cins.resize(pbm.productos);
for (i=0;i < pbm.productos;i++){
        is >> pbm.Cins[i];
}
is.ignore(256,'\n');
is.ignore(256,'\n');
pbm.Kelab.resize(pbm.productos);
for (i=0;i < pbm.productos;i++){
        is >> pbm.Kelab[i];
}

is.ignore(256,'\n');
is.ignore(256,'\n');
pbm.Kenv.resize(pbm.productos);
for (i=0;i < pbm.productos;i++){
        is >> pbm.Kenv[i];
}
is.ignore(256,'\n');
is.ignore(256,'\n');
pbm.Celab = Matrix<int> (pbm.productos,pbm.periodos,0,1,-1);
for (i=0;i < pbm.productos;i++){
        for (j=0;j < pbm.periodos;j++){
                is >> pbm.Celab(i,j);
        }
}
is.ignore(256,'\n');
is.ignore(256,'\n');
pbm.Cenv = Matrix<int> (pbm.productos,pbm.periodos,0,1,-1);
for (i=0;i < pbm.productos;i++){
```

```
                for (j=0;j < pbm.periodos;j++){
                        is >> pbm.Cenv(i,j);
                }
        }
        is.ignore(256,'\n');
        is.ignore(256,'\n');
        pbm.Calm = Matrix<int> (pbm.productos,pbm.periodos,0,1,-1);
        for (i=0;i < pbm.productos;i++){
                for (j=0;j < pbm.periodos;j++){
                        is >> pbm.Calm(i,j);
                }
        }
        is.ignore(256,'\n');
        is.ignore(256,'\n');
        pbm.demanda.resize(pbm.productos);
        for (i=0;i < pbm.productos;i++){
                pbm.demanda[i] = Matrix<int>(pbm.escenarios,pbm.periodos,0,1,-1);
        }
        for (k=0;k < pbm.periodos;k++){
                for (i=0;i < pbm.productos;i++){
                        for (j=0;j < pbm.escenarios;j++){
                                is >> pbm.demanda[i](j,k);
                        }
                }
                is.ignore(256,'\n');
                is.ignore(256,'\n');
        }

        return is;
}

Problem& Problem::operator=  (const Problem& pbm)
{
        int i,j,k;
        productos=pbm.productos;
        periodos=pbm.periodos;
        posibilidades=pbm.posibilidades;
        escenarios=pbm.escenarios;
        KelabM=pbm.KelabM;
        KenvM=pbm.KenvM;
        for (i=0;i < productos;i++){
                Cres[i]=pbm.Cres[i];
                Cins[i]=pbm.Cins[i];
                Kelab[i]=pbm.Kelab[i];
```

```
                Kenv[i]=pbm.Kenv[i];
                cortes[i]=pbm.cortes[i];
                probabilidades[i]=pbm.probabilidades[i];
        }
        for (i=0;i < productos;i++){
                for (j=0;j < periodos;j++){
                        Celab(i,j)=pbm.Celab(i,j);
                        Cenv(i,j)=pbm.Cenv(i,j);
                        Calm(i,j)=pbm.Calm(i,j);
                }
        }
        for (i=0;i < productos;i++){
                for (j=0;j < escenarios;j++){
                        for (k=0;k < periodos;k++){
                                demanda[i](j,k)=pbm.demanda[i](j,k);
                        }
                }
        }
        return *this;
}

bool Problem::operator== (const Problem& pbm) const
{
        int i,j,k;
        if (productos!=pbm.productos) return false;
        if (periodos!=pbm.periodos) return false;
        if (posibilidades!=pbm.posibilidades) return false;
        if (escenarios!=pbm.escenarios) return false;
        if (KelabM!=pbm.KelabM) return false;
        if (KenvM!=pbm.KenvM) return false;
        bool esta = true;
        for (i=0;(i < productos) && esta;i++){
                esta = esta && (Cres[i]==pbm.Cres[i]);
                esta = esta && (Cins[i]==pbm.Cins[i]);
                esta = esta && (Kelab[i]==pbm.Kelab[i]);
                esta = esta && (Kenv[i]==pbm.Kenv[i]);
                esta = esta && (cortes[i]==pbm.cortes[i]);
                esta = esta && (probabilidades[i]==pbm.probabilidades[i]);
        }
        if (!esta) return false;
        for (i=0;(i < productos) && esta;i++){
                for (j=0;(j < periodos) && esta;j++){
                        esta = esta && (Celab(i,j)==pbm.Celab(i,j));
                        esta = esta && (Cenv(i,j)==pbm.Cenv(i,j));
```

```
                        esta = esta && (Calm(i,j)==pbm.Calm(i,j));
                }
        }
        if (!esta) return false;
        for (i=0;i < productos;i++){
                for (j=0;j < escenarios;j++){
                        for (k=0;k < periodos;k++){
                                if (demanda[i](j,k)==pbm.demanda[i](j,k)) return false;
                        }
                }
        }
        return esta;
}

bool Problem::operator!= (const Problem& pbm) const
{
        return !(*this == pbm);
}

Direction Problem::direction() const
{
        //return maximize;
        return minimize;
}

Problem::~Problem()
{
        Cres.clear();
        Cins.clear();
        Kelab.clear();
        Kenv.clear();
        cortes.clear();
        demanda.clear();
        probabilidades.clear();
}

int Problem::getProductos() const{
        return productos;
}

int Problem::getPeriodos() const{
        return periodos;
}
```

```
int Problem::getPosibilidades() const{
	return posibilidades;
}

int Problem::getEscenarios() const{
	return escenarios;
}

int Problem::getKelabM() const{
	return KelabM;
}

int Problem::getKenvM() const{
	return KenvM;
}

vector<int> Problem::getCres() const{
	return Cres;
}

vector<int> Problem::getCins() const{
	return Cins;
}

vector<int> Problem::getKelab() const{
	return Kelab;
}

vector<int> Problem::getKenv() const{
	return Kenv;
}

vector<int> Problem::getCortes() const{
	return cortes;
}

vector<float> Problem::getProbabilidades() const{
	return probabilidades;
}

Matrix<int> Problem::getCelab() const{
	return Celab;
}
```

```
Matrix<int> Problem::getCenv() const{
	return Cenv;
}

Matrix<int> Problem::getCalm() const{
	return Calm;
}

vector<Matrix<int> > Problem::getDemanda() const{
	return demanda;
}

int Problem::demandaTope(int prod, int esc) const{
	int demmax=0;
	int demmin=999999999;
	int dem=0;
	for (int k=0;k<periodos;k++){
		dem+=demanda[prod](esc,k);
	}
	if (dem > demmax)
		demmax=dem;
	if (dem < demmin)
		demmin=dem;
	return demmax;
}

// Solution ---------------------------------------------------------

Solution::Solution (const Problem& pbm):_pbm(pbm)
{
	int largo=pbm.getPeriodos() * pbm.getEscenarios();
	int prod=pbm.getProductos();
	matrizX = Matrix<int>(prod, largo,0,1,-1);
	const vector<int> cortes = pbm.getCortes();
	int i,j,k,l,cant,dem,per,esc,demprom;
	bool esta;
	vector<int> auxprod,auxcant;
	vector<Matrix<int> > demanda = _pbm.getDemanda();
	for (j=0;j<largo;j++){
		esta = false;
		k=0;
		while (!esta && k < cortes.size()){
			esta = cortes[k] == j;
			k++;
```

```
                }
                if (esta){ //si es un punto de corte, se genera un valor para el alelo

                        auxprod.resize(0);
                        auxcant.resize(0);
                        esc=j%pbm.getEscenarios();
                        for (i=0;i<prod;i++){//se buscan los productos con demanda para
satisfacer en este período
                                per=j/pbm.getEscenarios();
                                dem = demanda[i](esc,per);
                                while (per > 0){
                                        per--;
                                        dem += demanda[i](esc,per);
                                }
                                per=j;
                                while (per > pbm.getEscenarios()){
                                        per -= pbm.getEscenarios();
                                        dem -= matrizX(i,per);
                                }
                                if (dem > 0){
                                        auxprod.push_back(i);
                                        auxcant.push_back(dem);
                                }
                        }
                        if (auxprod.size() > 0){//si existen productos con demanda
insatisfecha se elige uno
                                l = rand_int(0,auxprod.size()-1); //sortear lugar
                                k = auxprod[l]; //elegir producto

                                if (j/pbm.getEscenarios() == pbm.getPeriodos() - 1) //si es
el último período, se produce la cantidad justa
                                        cant = auxcant[l];
                                else{
                                        demprom = pbm.demandaTope(k,esc);
                                        cant = rand_int(1,demprom); //sortear cantidad
                                }
                                for (i=0;i<prod;i++){ //sólo el producto elegido tendrá el
valor sorteado, los otros productos tendrán x=0. Así se controla que no se produce más de un
producto por período.
                                        if (i==k){
                                                matrizX(i,j)=cant;
                                        }else{
                                                matrizX(i,j)=0;
                                        }
```

```
                                        }
                                }else{
                                        for (i=0;i<prod;i++){ //si todos los productos tienen
demanda satisfecha, no se produce nada
                                                matrizX(i,j)=0;
                                        }
                                }
                        }else{//si no es un punto de corte, se copia el último valor. Así se
mantiene la no anticipatividad.
                                for (i=0;i<prod;i++){
                                        matrizX(i,j)=matrizX(i,j-1);
                                }
                        }
                }
        }

        const Problem& Solution::pbm() const
        {
                return _pbm;
        }

        Solution::Solution(const Solution& sol):_pbm(sol.pbm())
        {
                int per = _pbm.getPeriodos();
                int esc = _pbm.getEscenarios();
                int largo = per*esc;
                int prod = _pbm.getProductos();
                Matrix<int> x =sol.getMatrizX();
                for (int i=0;i<prod;i++){
                        for (int j=0;j<largo;j++){
                                matrizX(i,j)=x(i,j);
                        }
                }
        }

        istream& operator>> (istream& is, Solution& sol)
        {
                return is;
        }

        ostream& operator<< (ostream& os, const Solution& sol)
        {
                Problem _pbm = sol.pbm();
                int per = _pbm.getPeriodos();
```

```
	int esc = _pbm.getEscenarios();
	int prod = _pbm.getProductos();
	Matrix<int> matrizX = sol.getMatrizX();
	for (int i=0;i<prod;i++){
		os << "Producto " << i << endl;
		for (int j=0;j<esc;j++){
			os << "Escenario " << j << ": ";
			for (int k=0;k<per;k++){
				os << matrizX(i,j+k*esc) << ',';
			}
			os << endl;
		}
	}
	return os;
}

NetStream& operator << (NetStream& ns, const Solution& sol)
{
	return ns;
}

NetStream& operator >> (NetStream& ns, Solution& sol)
{
	return ns;
}

Solution& Solution::operator= (const Solution &sol)
{
	int per = _pbm.getPeriodos();
	int esc = _pbm.getEscenarios();
	int largo = per*esc;
	int prod = _pbm.getProductos();
	Matrix<int> x =sol.getMatrizX();
	for (int i=0;i<prod;i++){
		for (int j=0;j<largo;j++){
			matrizX(i,j)=x(i,j);
		}
	}
	return *this;
}

bool Solution::operator== (const Solution& sol) const
{
	if (sol.pbm() != _pbm) return false;
```

```
	return true;
}

bool Solution::operator!= (const Solution& sol) const
{
	return !(*this == sol);
}

void Solution::initialize()
{

}

double Solution::fitness ()
{
	bool error = false;
	int KelabM = _pbm.getKelabM();
	int KenvM = _pbm.getKenvM();
	vector<int> Kelab = _pbm.getKelab();
	vector<int> Kenv = _pbm.getKenv();
	vector<int> Cres = _pbm.getCres();
	vector<int> Cins = _pbm.getCins();
	vector<float> probabilidades = _pbm.getProbabilidades();
	Matrix<int> Celab = _pbm.getCelab();
	Matrix<int> Cenv =_pbm.getCenv();
	Matrix<int> Calm =_pbm.getCalm();
	int per = _pbm.getPeriodos();
	int esc = _pbm.getEscenarios();
	int largo = per*esc;
	int prod = _pbm.getProductos();
	int elab,env,dem,sum,i,j,k,determinante;
	vector<Matrix<int> > demanda = _pbm.getDemanda();

	double fitness = 0.0;

	//control de capacidades
	for (j=0;j<largo && !error;j++){
		elab=0;
		env=0;
		for (i=0;i<prod;i++){
			elab += matrizX(i,j)*Kelab[i];
			env += matrizX(i,j)*Kenv[i];
		}
		error = ((elab > KelabM) or (env > KenvM));
```

```
		}
		if (error) return MAX_FITNESS;

		//controlar que la suma de las X supere la demanda total, pero que ninguna X supere el total entero.
		for(i=0;i<prod && !error;i++){
			for (j=0;j<esc && !error;j++){
				dem = 0;
				for (k=0;k<per;k++){
					dem += demanda[i](j,k);
				}
				k=j;
				sum=0;
				while (k < esc*per && !error){
					sum += matrizX(i,k);
					error = matrizX(i,k) > dem;
					k += esc;
				}
				if (sum < dem){
					error = true;
				}
			}
		}
		if (error) return MAX_FITNESS;

		//Armado de las matrices S, R y U
		vector<Matrix<int> > MatrizS(prod);
		vector<Matrix<int> > MatrizR(prod);
		vector<Matrix<int> > MatrizU(prod);
		for (i=0;i<prod;i++){
			MatrizS[i] = Matrix<int>(esc,per,0,1,-1);
			MatrizR[i] = Matrix<int>(esc,per,0,1,-1);
			MatrizU[i] = Matrix<int>(esc,per,0,1,-1);
			for (j=0;j<esc;j++){
				if (matrizX(i,j) > 0){
					MatrizU[i](j,0) = 1;
				}else{
					MatrizU[i](j,0) = 0;
				}
				determinante = matrizX(i,j) - demanda[i](j,0);
				if (determinante > 0){
					MatrizS[i](j,0) = determinante;
					MatrizR[i](j,0) = 0;
				}else{
```

```
                                        MatrizR[i](j,0) = -determinante;
                                        MatrizS[i](j,0) = 0;
                                }
                                for (k=1;k<per;k++){
                                        if ((matrizX(i,j+k*esc) > 0) && (MatrizU[i](j,k-1) == 0)){
                                                MatrizU[i](j,k) = 1;
                                        }else{
                                                MatrizU[i](j,k) = 0;
                                        }
                                        determinante = matrizX(i,j+k*esc) + MatrizS[i](j,k-1) -
MatrizR[i](j,k-1) - demanda[i](j,k);
                                        if (determinante > 0){
                                                MatrizS[i](j,k)=determinante;
                                                MatrizR[i](j,k)=0;
                                        }else{
                                                MatrizR[i](j,k)=-determinante;
                                                MatrizS[i](j,k)=0;
                                        }
                                }
                        }
                }
                //calculo del fitness

                for (j=0;j<esc;j++){
                        for (k=0;k<per;k++){
                                for (i=0;i<prod;i++){
                                        fitness += (Cenv(i,k) + Celab(i,k))*matrizX(i,j+k*esc) *
probabilidades[j];
                                        fitness += Calm(i,k)*MatrizS[i](j,k) * probabilidades[j];
                                        fitness += Cres[i]*MatrizU[i](j,k) * probabilidades[j];
                                        fitness += Cins[i]*MatrizR[i](j,k) * probabilidades[j];

                                }
                        }
                }
                MatrizS.clear();
                MatrizU.clear();
                MatrizR.clear();
                return fitness;
        }

        char *Solution::to_String() const
        {
                int per = _pbm.getPeriodos();
```

```
	int esc = _pbm.getEscenarios();
	int largo = per*esc;
	int prod = _pbm.getProductos();
	ostringstream oss;
	for (int i=0;i<prod;i++){
		for (int j=0;j<largo;j++){
			oss << matrizX(i,j) << ' ';
		}
	}
	const string& tmp = oss.str();
	const char* res = tmp.c_str();
	char* string=new char[strlen(res)-1];
	strncpy(string,res,strlen(res)-1);
	return string;
}

void Solution::to_Solution(char *_string_)
{
	int per = _pbm.getPeriodos();
	int esc = _pbm.getEscenarios();
	int largo = per*esc;
	int prod = _pbm.getProductos();
	int d;
	istringstream iss(_string_);
	for (int i=0;i<prod;i++){
		for (int j=0;j<largo;j++){
			iss >> matrizX(i,j);
		}
	}
}

unsigned int Solution::size() const
{
	return strlen(to_String());
}

Solution::~Solution()
{

}

Matrix<int> Solution::getMatrizX() const{
	return matrizX;
}
```

```
void Solution::setMatrizX(Matrix<int> x){
	int per = _pbm.getPeriodos();
	int esc = _pbm.getEscenarios();
	int largo = per*esc;
	int prod = _pbm.getProductos();
	for (int i=0;i<prod;i++){
		for (int j=0;j<largo;j++){
			matrizX(i,j)=x(i,j);
		}
	}
}


// UserStatistics -----------------------------------------------------

UserStatistics::UserStatistics ()
{}

ostream& operator<< (ostream& os, const UserStatistics& userstat)
{
	os << "\n----------------------------------------------------------------" << endl;
	os << "                  STATISTICS OF TRIALS                          " << endl;
	os << "----------------------------------------------------------------" << endl;

	for (int i=0;i< userstat.result_trials.size();i++)
	{
		os << endl
		   << userstat.result_trials[i].trial
		   << "\t" << userstat.result_trials[i].best_cost_trial
		   << "\t\t" << userstat.result_trials[i].worst_cost_trial
		   << "\t\t\t" << userstat.result_trials[i].nb_evaluation_best_found_trial
		   << "\t\t\t" << userstat.result_trials[i].nb_iteration_best_found_trial
		   << "\t\t\t" << userstat.result_trials[i].time_best_found_trial
		   << "\t\t" << userstat.result_trials[i].time_spent_trial;
	}
	os << endl << "----------------------------------------------------------------" << endl;
	return os;
}

UserStatistics& UserStatistics::operator= (const UserStatistics& userstats)
{
	result_trials=userstats.result_trials;
	return (*this);
```

```
	}

	void UserStatistics::update(const Solver& solver)
	{
		if( (solver.pid()!=0) || (solver.end_trial()!=true)
		  || ((solver.current_iteration()!=solver.setup().nb_evolution_steps())
		      && !terminateQ(solver.pbm(),solver,solver.setup())))
			return;

		struct user_stat *new_stat;

		if ((new_stat=(struct user_stat *)malloc(sizeof(struct user_stat)))==NULL)
			show_message(7);
		new_stat->trial                                 = solver.current_trial();
		new_stat->nb_iteration_best_found_trial      =
solver.iteration_best_found_in_trial();
		new_stat->nb_evaluation_best_found_trial =
solver.evaluations_best_found_in_trial();
		new_stat->worst_cost_trial                              = solver.worst_cost_trial();
		new_stat->best_cost_trial                       = solver.best_cost_trial();
		new_stat->time_best_found_trial                         =
solver.time_best_found_trial();
		new_stat->time_spent_trial                              = solver.time_spent_trial();

		result_trials.append(*new_stat);
	}

	void UserStatistics::clear()
	{
		result_trials.remove();
	}

	UserStatistics::~UserStatistics()
	{
		result_trials.remove();
	}

// Intra_operator  ------------------------------------------------------------

	Intra_Operator::Intra_Operator(const unsigned int
_number_op):_number_operator(_number_op),probability(NULL)
	{}

	unsigned int Intra_Operator::number_operator() const
```

```
	{
		return _number_operator;
	}

	Intra_Operator *Intra_Operator::create(const unsigned int _number_op)
	{
		switch (_number_op)
		{
			case 0: return new Crossover;break;
			case 1: return new Mutation();break;
		}
	}

	ostream& operator<< (ostream& os, const Intra_Operator& intra)
	{
		switch (intra.number_operator())
		{
			case 0: os << (Crossover&)intra;break;
			case 1: os << (Mutation&)intra;break;
		}
		return os;
	}

	Intra_Operator::~Intra_Operator()
	{}

//  Crossover_PMX1:Intra_operator ------------------------------------------------------------

	Crossover::Crossover():Intra_Operator(0)
	{
		probability = new float[1];
	}

	void Crossover::cross(Solution& sol1,Solution& sol2) const // dadas dos soluciones de la
poblacion, las cruza
	{
		const Problem& _pbm = sol1.pbm();
		int per = _pbm.getPeriodos();
		int esc = _pbm.getEscenarios();
		int largo = per*esc;
		int prod = _pbm.getProductos();
		int i,j,k,temp;
		vector<int> cortes = _pbm.getCortes();
		Matrix<int> mat1 = sol1.getMatrizX();
```

```
                Matrix<int> mat2 = sol2.getMatrizX();
                k = rand_int(0,cortes.size()-1);
                for (i=0;i<prod;i++){
                        for (j=cortes[k];j<largo;j++){
                                temp = mat1(i,j);
                                mat1(i,j)=mat2(i,j);
                                mat2(i,j)=temp;
                        }
                }
                sol1.setMatrizX(mat1);
                sol2.setMatrizX(mat2);
        }

        void Crossover::execute(Rarray<Solution*>& sols) const
        {
                for (int i=0;i+1<sols.size();i=i+2)
                        if (rand01()<=probability[0]) cross(*sols[i],*sols[i+1]);
        }

        ostream& operator<< (ostream& os, const Crossover&  cross)
        {
                 os << "Crossover." << " Probability: "
               << cross.probability[0]
                    << endl;
                 return os;
        }

        void Crossover::RefreshState(const StateCenter& _sc) const
        {
                 _sc.set_contents_state_variable("_crossover_probability",(char
*)probability,1,sizeof(float));
        }

        void Crossover::UpdateFromState(const StateCenter& _sc)
        {
                  unsigned long nbytes,length;
                 _sc.get_contents_state_variable("_crossover_probability",(char
*)probability,nbytes,length);
        }

        void Crossover::setup(char line[MAX_BUFFER])
        {
                int op;
                sscanf(line," %d %f ",&op,&probability[0]);
```

```
        assert(probability[0]>=0);
}

Crossover::~Crossover()
{
        delete [] probability;
}

//  Mutation: Sub_operator ------------------------------------------------------

Mutation::Mutation():Intra_Operator(1)
{
        probability = new float[1];
}

void Mutation::mutate(Solution& sol) const
{
        const Problem& _pbm = sol.pbm();
        int per = _pbm.getPeriodos();
        int esc = _pbm.getEscenarios();
        int largo = per*esc;
        int prod = _pbm.getProductos();
        int i,j,k,l,temp,limizq,limder,cant,dem,per2;
        vector<int> cortes = _pbm.getCortes();
        Matrix<int> mat = sol.getMatrizX();
        vector<int> auxprod,auxcant;
        vector<Matrix<int> > demanda = _pbm.getDemanda();

        l = rand_int(0,cortes.size()-1);
        limizq = cortes[l];
        if (l == cortes.size()-1){
                limder = limizq + 1;
        }else{
                limder = cortes[l+1];
        }

        auxprod.resize(0);
        auxcant.resize(0);
        int esc2=limizq%esc;
        for (i=0;i<prod;i++){//se buscan los productos con demanda para satisfacer en
este período
                per2=limizq/esc;
                dem = demanda[i](esc2,per2);
                while (per2 > 0){
```

```
                        per2--;
                        dem += demanda[i](esc2,per2);
                }
                per2=limizq;
                while (per2 > esc){
                        per2 -= esc;
                        dem -= mat(i,per2);
                }
                if (dem > 0){
                        auxprod.push_back(i);
                        auxcant.push_back(dem);
                }
        }

        if (auxprod.size() > 0){//si existen productos con demanda insatisfecha se elige
uno
                l = rand_int(0,auxprod.size()-1); //sortear lugar
                k = auxprod[l]; //elegir producto

                if (limizq/esc == per - 1)//si es el último período, se produce la cantidad
justa
                        cant = auxcant[l];
                else{
                        int demprom = _pbm.demandaTope(k,esc2);
                        cant = rand_int(1,demprom); //sortear cantidad
                }
                for (i=0;i<prod;i++){ //sólo el producto elegido tendrá el valor sorteado, los
otros productos tendrán x=0. Así se controla que no se produce más de un producto por
período.
                        if (i==k){
                                mat(i,limizq)=cant;
                        }else{
                                mat(i,limizq)=0;
                        }
                }
        }else{
                for (i=0;i<prod;i++){ //si todos los productos tienen demanda satisfecha,
no se produce nada
                        mat(i,limizq)=0;
                }
        }

        for (i=0;i<prod;i++){//entre los limites, se copia el último valor. Así se mantiene la
no anticipatividad.
```

```
                        for (j=limizq+1;j<limder;j++){
                                mat(i,j)=mat(i,j-1);
                        }
                }
                sol.setMatrizX(mat);
        }

        void Mutation::execute(Rarray<Solution*>& sols) const
        {
                for (int i=0;i<sols.size();i++)
                        if(rand01() <= probability[0])   mutate(*sols[i]);
        }

        ostream& operator<< (ostream& os, const Mutation&  mutation)
        {
                os << "Mutation." << " Probability: "
                  << mutation.probability[0]
                  << endl;
                return os;
        }

        void Mutation::setup(char line[MAX_BUFFER])
        {
                int op;
                sscanf(line," %d %f",&op,&probability[0]);
                assert(probability[0]>=0);
        }

        void Mutation::RefreshState(const StateCenter& _sc) const
        {
                _sc.set_contents_state_variable("_mutation_probability",(char
*)probability,1,sizeof(probability));
        }

        void Mutation::UpdateFromState(const StateCenter& _sc)
        {
                unsigned long nbytes,length;
                _sc.get_contents_state_variable("_mutation_probability",(char
*)probability,nbytes,length);
        }

        Mutation::~Mutation()
        {
                delete [] probability;
```

```
	}

// StopCondition_1 ------------------------------------------------------------------------

	StopCondition_1::StopCondition_1():StopCondition()
	{}

	bool StopCondition_1::EvaluateCondition(const Problem& pbm,const Solver&
solver,const SetUpParams& setup)
	{
		unsigned long generacion = solver.current_iteration();
		double best_cost = solver.global_best_cost();
		double worst_cost = solver.current_worst_cost();
		if( generacion > setup.nb_evolution_steps()
			|| ((worst_cost <= best_cost*1.10) && best_cost < MAX_FITNESS)){
			return true;
		}
		return false;
	}

	StopCondition_1::~StopCondition_1()
	{}

	//-------------------------------------------------------------------
	// Specific methods ---------------------------------------------------
	//-------------------------------------------------------------------

	bool terminateQ (const Problem& pbm, const Solver& solver,
			 const SetUpParams& setup)
	{
		StopCondition_1 stop;
		return stop.EvaluateCondition(pbm,solver,setup);
	}
}
#endif
```

newGA.pro.cc

```
#include "newGA.hh"

skeleton newGA
{

// StopCondition ------------------------------------------------------------------------
```

```
StopCondition::StopCondition()
{}

StopCondition::~StopCondition()
{}

// SetUpParams ----------------------------------------------------

SetUpParams::SetUpParams (Operator_Pool& pool)
: _independent_runs(0),
  _nb_evolution_steps(0),
  _population_size(0),
  _population_additional_size(0),
  _select_parents(0),
  _select_offsprings(0),
  _parameter_select_parents(0),
  _parameter_select_offsprings(0), // Parameter of selection is fixed to 0
  _inter_operators(),
  _intra_operators(),
  _combine(1),
  _refresh_global_state(1),
  _synchronized(0),
  _check_asynchronous(1),
  _display_state(0),
  _pool(pool)
{}

Operator_Pool& SetUpParams::pool() const
{
        return _pool;
}

istream& operator>> (istream& is, SetUpParams& setup)
{
        char buffer[MAX_BUFFER]; // current line in the setup file
        char command[50];
        long op;
        int parameter;
        short int nb_section=0;
        short int nb_io = 0;
        short int nb_selection = 0;
        short int nb_param=0;
        short int nb_LAN_param=0;
```

```
while (is.getline(buffer,MAX_BUFFER,'\n'))
{
	sscanf(buffer," %s ",command);
	if (!(strcmp(command,"General"))) nb_section=0;
	if (!(strcmp(command,"Selections"))) nb_section=1;
	if (!(strcmp(command,"Intra-Operators"))) nb_section=2;
	if (!(strcmp(command,"Inter-Operators"))) nb_section=3;
	if (!(strcmp(command,"LAN-configuration"))) nb_section=4;

	op=-1;
	sscanf(buffer," %ld%*s ",&op);
	if (op<0) continue;
	switch (nb_section)
	{
		case 0: switch (nb_param)
			{
				case 0: setup.independent_runs(op); break;
				case 1: setup.nb_evolution_steps(op); break;
				case 2: setup.population_size(op); break;
				case 3: setup.population_additional_size(op);
break;
				case 4: setup.combine(op); break;
				case 5: setup.display_state(op); break;
			}
			nb_param++;
			break;
		case 1: op=-1; // creates the chosen selection method
			parameter=0;
			sscanf(buffer," %d %d",&op,&parameter);
			if (nb_selection>=2) break;
			assert(parameter>=0);
			if (nb_selection==0)
			{
				setup.select_parents(op);
				setup.parameter_select_parents(parameter);
			}
			else
			{
				setup.select_offsprings(op);
				setup.parameter_select_offsprings(parameter);
			}
			nb_selection++;
			break;
```

```
                                case 2:
setup.pool().intra_operators().append(Intra_Operator::create(op));
                                        setup.pool().intra_operator(nb_io).setup(buffer);
                                        setup._intra_operators.append(new unsigned int(nb_io));
                                        nb_io++;
                                        break;
                                case 3: setup._inter_operators.append(new unsigned int(op));
                                        setup.pool().inter_operator(op).setup(buffer);
                                        break;
                                case 4: if (nb_LAN_param>=3) break;
                                        switch (nb_LAN_param)
                                        {
                                                case 0: setup.refresh_global_state(op); break;
                                                case 1: setup.synchronized(op); break;
                                                case 2: assert(op>0);
                                                        setup.check_asynchronous(op);
break;
                                        }
                                        nb_LAN_param++;
                                        break;
                        }
                }

                return is;
        }

        ostream& operator<< (ostream& os, const SetUpParams& setup)
        {
                os << "CONFIGURATION -----------------------------------------" << endl << endl;
                os << "\t" << "Independent runs : " << setup.independent_runs()    << endl
                  << "\t" << "Evolution steps: "   << setup.nb_evolution_steps() << endl
                  << "\t" << "Size of Population: " <<  setup.population_size() << endl
          << "\t" << "Size of Additional population: " << setup.population_additional_size() <<
endl;
                if (setup.combine())
                        os << "\t" <<"With combination between parents and offsprings" << endl;
                else
                        os <<  "\t" <<"Without combination between parents and offsprings" <<
endl;

                os << "\t" << "Display State: " << setup.display_state() << endl << endl
                  << "\t" << "Selections:" << endl
                  << "\t" << "----------" << endl << endl
```

```
            << "\t" << "Selection parents    -> " << setup.pool().selector(setup.select_parents()) <<
endl
                    << "\t" << "Parameter of selection: " << setup.parameter_select_parents() <<
endl
            << "\t" << "Selection offsprings -> " << setup.pool().selector(setup.select_offsprings())
<< endl
                    << "\t" << "Parameter of selection: " << setup.parameter_select_offsprings() <<
endl  << endl
          << "\t" << "Intra_Operators: " << endl
                    << "\t" << "-----------" << endl << endl;

                for (int i=0;i<setup.intra_operators_size();i++)
                        os << "\t" << (setup.pool().intra_operator(setup.intra_operator_index(i)))
<< endl;

                os << endl << "\t" << "Inter_Operators: " << endl
                  << "\t" << "-----------" << endl << endl;

                for (int i=0;i<setup.inter_operators_size();i++)
                        os << "\t"  << "Operator: " <<
setup.pool().inter_operator(setup.inter_operator_index(i)) << endl;

                os << endl << "\t" << "LAN configuration:" << endl
                  << "\t" << "----------------------" << endl << endl
                  << "\t" << "Refresh global state in number of generations: " <<
setup.refresh_global_state() << endl;

                if (setup.synchronized())
                        os << "\t" << "Running in synchronous mode" << endl;
                else
                        os << "\t" << "Running in asynchronous mode" << endl;
                os << "\t" << "Interval for checking asynchronous receptions: " <<
setup.check_asynchronous() << endl << endl;

                os << endl << endl << "END CONFIGURATION
-----------------------------------------" << endl << endl;
        return os;
          }

          const unsigned int    SetUpParams::independent_runs() const
          {
                  return _independent_runs;
          }
```

```
const unsigned long    SetUpParams::nb_evolution_steps() const
{
        return _nb_evolution_steps;
}

const unsigned int SetUpParams::population_size() const
{
        return _population_size;
}

const unsigned int SetUpParams::population_additional_size() const
{
        return _population_additional_size;
}

const bool SetUpParams::combine() const
{
        return _combine;
}

const unsigned long SetUpParams::refresh_global_state() const
{
        return _refresh_global_state;
}

const bool SetUpParams::synchronized() const
{
        return _synchronized;
}

const unsigned int SetUpParams::check_asynchronous() const
{
        return _check_asynchronous;
}

const bool  SetUpParams::display_state() const
{
        return _display_state;
}

void SetUpParams::independent_runs(const unsigned int val)
{
        _independent_runs=val;
}
```

```
void SetUpParams::nb_evolution_steps(const unsigned long val)
{
	_nb_evolution_steps=val;
}

void SetUpParams::population_size(const unsigned int val)
{
	_population_size=val;
}

void SetUpParams::population_additional_size(const unsigned int val)
{
	_population_additional_size=val;
}

void SetUpParams::combine(const bool val)
{
	_combine=val;
}

void SetUpParams::display_state(const bool val)
{
	_display_state=val;
}

void SetUpParams::refresh_global_state(const unsigned long val)
{
	_refresh_global_state=val;
}

void SetUpParams::synchronized(const bool val)
{
	_synchronized=val;
}

void SetUpParams::check_asynchronous(const unsigned int val)
{
	_check_asynchronous=val;
}

const unsigned int SetUpParams::select_parents() const
{
	return _select_parents;
```

```
}

const unsigned int SetUpParams::select_offsprings() const
{
	return _select_offsprings;
}

const unsigned  int SetUpParams::parameter_select_parents() const
{
	return _parameter_select_parents;
}

const unsigned int SetUpParams::parameter_select_offsprings() const
{
	return _parameter_select_offsprings;
}

void SetUpParams::select_parents(const unsigned int val)
{
	_select_parents=val;
}

void SetUpParams::select_offsprings(const unsigned int val)
{
	_select_offsprings=val;
}

void SetUpParams::parameter_select_parents(const unsigned int val)
{
	_parameter_select_parents=val;
}

void SetUpParams::parameter_select_offsprings(const unsigned int val)
{
	_parameter_select_offsprings=val;
}

const unsigned int SetUpParams::intra_operator_index(const unsigned int index) const
{
	return _intra_operators[index];
}

const unsigned int SetUpParams::intra_operators_size() const
{
```

```
		return _intra_operators.size();
	}

	const unsigned int SetUpParams::inter_operator_index(const unsigned int index) const
	{
		return _inter_operators[index];
	}

	const unsigned int SetUpParams::inter_operators_size() const
	{
		return _inter_operators.size();
	}

	void SetUpParams::RefreshState(const StateCenter& _sc) const
	{
		 _sc.set_contents_state_variable("_select_parents",(char
*)&_select_parents,1,sizeof(_select_parents));
		 _sc.set_contents_state_variable("_parameter_select_parents",(char
*)&_parameter_select_parents,1,sizeof(_parameter_select_parents));
		 _sc.set_contents_state_variable("_select_offsprings",(char
*)&_select_offsprings,1,sizeof(_select_offsprings));
		 _sc.set_contents_state_variable("_parameter_select_offsprings",(char
*)&_parameter_select_offsprings,1,sizeof(_parameter_select_offsprings));
		 _sc.set_contents_state_variable("_display_state",(char
*)&_display_state,1,sizeof(bool));
	}

	void SetUpParams::UpdateFromState(const StateCenter& _sc) const
	{
		unsigned long nbytes,length;
		_sc.get_contents_state_variable("_select_parents",(char
*)&_select_parents,nbytes,length);
		_sc.get_contents_state_variable("_parameter_select_parents",(char
*)&_parameter_select_parents,nbytes,length);
		_sc.get_contents_state_variable("_select_offsprings",(char
*)&_select_offsprings,nbytes,length);
		_sc.get_contents_state_variable("_parameter_select_offsprings",(char
*)&_parameter_select_offsprings,nbytes,length);
		_sc.get_contents_state_variable("_display_state",(char
*)&_display_state,nbytes,length);
	}

	SetUpParams::~SetUpParams()
	{}
```

```
// Statistics ----------------------------------------------------

	Statistics::Statistics()
	{}

	ostream& operator<< (ostream& os, const Statistics& stats)
	{
		os << "\n-----------------------------------------------------------------" << endl;
		os << "                  STATISTICS OF CURRENT TRIAL                    " << endl;
		os << "-----------------------------------------------------------------" << endl;
		for (int i=0;i< stats.stats_data.size();i++)
		{
			os << endl
			   << " Trial:     " << stats.stats_data[i].trial
			   << " Generation: " << stats.stats_data[i].nb_generation
			   << " Evaluation: " << stats.stats_data[i].nb_evaluation
			   << " Current best cost: " << stats.stats_data[i].best_cost
			   << " Global best cost: " << stats.stats_data[i].global_best_cost
			   << " Avg: " << stats.stats_data[i].average_cost
			   << " Std. Dev.: " << stats.stats_data[i].standard_deviation;
		}

		os << endl << "-----------------------------------------------------------------" << endl;
		return os;
	}

	Statistics& Statistics::operator= (const Statistics& stats)
	{
		stats_data = stats.stats_data;
		return *this;
	}

	void Statistics::update(const Solver& solver)
	{
		struct stat *new_stat=(struct stat *)malloc(sizeof(struct stat));

		new_stat->trial=solver.current_trial();
		new_stat->nb_generation=solver.current_iteration();
		new_stat->nb_evaluation=solver.current_evaluations();
		new_stat->average_cost=solver.current_average_cost();
		new_stat->standard_deviation=solver.current_standard_deviation();
		new_stat->best_cost=solver.current_best_cost();
		new_stat->global_best_cost=solver.global_best_cost();
```

```
            stats_data.append(*new_stat);
        }

        void Statistics::clear()
        {
            stats_data.remove();
        }

    Statistics::~Statistics()
        {}

// Population ----------------------------------------------------

        Population::Population(const Problem& pbm,const SetUpParams& setup)
        :_parents(setup.population_size()),
         _fitness_values(setup.population_size()),
         _new_parents(setup.population_size()),
         _offsprings(setup.population_additional_size()),
         _setup(setup),
         _evaluations(0)
        {
            for (int i=0;i<_parents.size();i++)
            {
                _parents[i]=new Solution(pbm);
                _new_parents[i]=new Solution(pbm);
                _fitness_values[i].index = i;
                _fitness_values[i].change = true;
            }
            for (int i=0;i<_offsprings.size();i++)
                _offsprings[i]=new Solution(pbm);

        }

        void Population::Evaluate(Solution* sols,struct individual &_f)
        {
            if(_f.change)
            {
                _f.change  = false;
                _f.fitness = sols->fitness();
                _evaluations++;
            }
        }
```

```
Population& Population::operator= (const Population& pop)
{
	for (int i=0;i<_parents.size();i++)
	{
		*_parents[i]=*((pop.parents())[i]);
		_fitness_values[i] = pop._fitness_values[i];
		_evaluations = pop._evaluations;
	}
	return (*this);
}

istream& operator>> (istream& is, Population& population)
{
	return is;
}

ostream& operator<< (ostream& os, const Population& population)
{
	os << "-------------------------------------------------------------" << endl;
	os << "                        PRESENT POPULATION                   " << endl << endl;
	for (int i=0;i<population._parents.size();i++){
		os << "Individual nº " << i << endl << *population._parents[i] << endl;
		os << "Fitness " << population._fitness_values[i].fitness << endl;
	}
	os << endl << "-------------------------------------------------------------" << endl;
	return os;
}

const SetUpParams& Population::setup() const
{
	return _setup;
}

void Population::initialize()
{
	for (int i=0;i<_parents.size();i++)
	{
		_parents[i]->initialize();
		_fitness_values[i].index = i;
		_fitness_values[i].change = true;
	}
	evaluate_parents();
}
```

```
	void Population::evaluate_parents()
	{
		double upper_fitness=(infinity() * (-1));
		double lower_fitness=(infinity());
		double current_fitness;
		double cost=0.0;

		for (int i=0;i<_fitness_values.size();i++)
		{
			Evaluate(_parents[_fitness_values[i].index],_fitness_values[i]);
			current_fitness = _fitness_values[i].fitness;

			if (current_fitness > upper_fitness )
			{
				_upper_cost=i;
				upper_fitness = current_fitness;
			}

			if (current_fitness < lower_fitness )
			{
				_lower_cost=i;
				lower_fitness = current_fitness;
			}

			cost += current_fitness;
		}

		_average_cost =  cost / _fitness_values.size();
	}

	void Population::evaluate_offsprings()
	{
		int i=0;
		if (_setup.combine())  // new individuals selected between current individuals and
offsprings
		{
			_fitness_aux=Rarray<struct individual>(_parents.size() +
_offsprings.size());
		   for (i=0;i<_parents.size();i++)
			{
				Evaluate(_parents[_fitness_values[i].index],_fitness_values[i]);
				_fitness_aux[i] = _fitness_values[i];
			}
```

```
                    for (int j=i;(j-i)<_offsprings.size();j++)
                    {
                            _fitness_aux[j].index=j;
                            _fitness_aux[j].change=true;
                            Evaluate(_offsprings[j-i],_fitness_aux[j]);
                    }
            }
            else    // new individuals selected only between offsprings
            {
                    _fitness_aux=Rarray<struct individual>(_offsprings.size());
                    for (i=0;i<_offsprings.size();i++)
                    {
                            _fitness_aux[i].index=i;
                            _fitness_aux[i].change=true;
                            Evaluate(_offsprings[i],_fitness_aux[i]);
                    }
            }
    }

    void Population::evolution()
    {
            select_parents();  // selects individuals to apply operators

            // apply selected operators
            for (int i=0;i<_setup.intra_operators_size();i++)

    _setup.pool().intra_operator(_setup.intra_operator_index(i)).execute(_offsprings);

            evaluate_offsprings();
            select_offsprings(); // selects new individuals
            evaluate_parents(); // calculates fitness of new individuals
    }

    void Population::interchange(const unsigned long current_generation, NetStream&
channel)
    {
            // apply selected operators
            for (int i=0;i<_setup.inter_operators_size();i++)


_setup.pool().inter_operator(_setup.inter_operator_index(i)).execute((*this),current_generation,c
hannel,_setup.synchronized(),_setup.check_asynchronous());
    }
```

```
	void Population::select_parents()
	{
		_setup.pool().selector(_setup.select_parents()).prepare(_fitness_values,false);
		struct individual ind;
		for (int i=0;i<_offsprings.size();i++)
		{
			ind =
_setup.pool().selector(_setup.select_parents()).select_one(_parents,_offsprings,_fitness_values
,_setup.parameter_select_parents(),false);
			*_offsprings[i] = *_parents[ind.index];
		}
	}

	void Population::select_offsprings()
	{
		_setup.pool().selector(_setup.select_offsprings()).prepare(_fitness_aux,false);
		const int ps = _parents.size();
		Rarray<struct individual> aux(ps);

		for (int i=0;i<ps;i++)
		{
			if (_setup.combine())
			{
				aux[i] =
_setup.pool().selector(_setup.select_offsprings()).select_one(_parents,_offsprings,_fitness_aux,
_setup.parameter_select_offsprings(),false);
				if(aux[i].index < ps)
				{
					*_new_parents[i] = *_parents[aux[i].index];
					aux[i].index = i;
				}
				else
				{
					*_new_parents[i] = *_offsprings[aux[i].index-ps];
					aux[i].index = i;
				}
			}
			else
			{

aux[i]=_setup.pool().selector(_setup.select_offsprings()).select_one(_offsprings,_offsprings,_fitn
ess_aux,_setup.parameter_select_offsprings(),false);
				*_parents[i] = *_offsprings[aux[i].index];
```

```
                        aux[i].index = i;
                    }
                }

                if (_setup.combine()) // interchanges current and new parents in the population
                {
                    Solution *interchange;
                    for (int i=0;i<ps;i++)
                    {
                        interchange=_parents[i];
                        _parents[i]=_new_parents[i]; // interchanges pointers to solutions (
NO solutions !! )
                        _new_parents[i]=interchange;
                    }
                }
                for (int i=0;i<ps;i++)
                {
                    _fitness_values[i] = aux[i];
                }
        }

        const Rarray<Solution*>& Population::parents() const
        {
                return _parents;
        }

        const Rarray<Solution*>& Population::offsprings() const
        {
                return _offsprings;
        }

        Rarray<struct individual>& Population::fitness_values()
        {
                return _fitness_values;
        }

        unsigned int Population::upper_cost() const
        {
                return _upper_cost;
        }

        unsigned int Population::lower_cost() const
        {
                return _lower_cost;
```

```
}

unsigned int Population::evaluations() const
{
	return _evaluations;
}

double Population::best_cost() const
{
	if ((*_parents[0]).pbm().direction() == minimize)
		return _fitness_values[_lower_cost].fitness;
	else
		return _fitness_values[_upper_cost].fitness;
}

double Population::worst_cost() const
{
	if ((*_parents[0]).pbm().direction() == minimize)
		return _fitness_values[_upper_cost].fitness;
	else
		return _fitness_values[_lower_cost].fitness;
}

Solution& Population::best_solution() const
{
	if ((*_parents[0]).pbm().direction() == minimize){
		return *_parents[_fitness_values[_lower_cost].index];
	}else
		return *_parents[_fitness_values[_upper_cost].index];
}

Solution& Population::worst_solution() const
{
	if ((*_parents[0]).pbm().direction() == minimize)
		return *_parents[_fitness_values[_upper_cost].index];
	else
		return *_parents[_fitness_values[_lower_cost].index];
}

Solution& Population::solution(const unsigned int index) const
{
	return *_parents[index];
}
```

```
double Population::fitness(const unsigned int index) const
{
	return _fitness_values[index].fitness;
}

double Population::average_cost() const
{
	return _average_cost;
}

double Population::standard_deviation() const
{
	double standard=0.0;
	for (int i=0;i<_fitness_values.size();i++)
		standard += pow ((_fitness_values[i].fitness - _average_cost),2);
	standard=sqrt(standard / (_fitness_values.size()-1));
	return standard;
}

Population::~Population()
{
	for (int i=0;i<_parents.size();i++)
		delete(_parents[i]);
	for (int i=0;i<_offsprings.size();i++)
		delete(_offsprings[i]);
	for (int j=0;j<_new_parents.size();j++)
		delete(_new_parents[j]);
}


// Inter_operator ------------------------------------------------------------

Inter_Operator::Inter_Operator(const unsigned int _number_op,const Direction dir):
_number_operator(_number_op),
direction(dir),
migration_rate(1),
migration_size(1),
migration_selection_1(0),
migration_selection_2(0),
migration_selection_conf_1(0),
migration_selection_conf_2(0)
{}

unsigned int Inter_Operator::number_operator() const
```

```
	{
		return _number_operator;
	}

	void Inter_Operator::setup(char line[MAX_BUFFER])
	{
		int op;
		int new_migration_rate=1;
		int new_migration_size=1;
		int new_migration_selection_1=0;
		int new_migration_selection_conf_1=0;
		int new_migration_selection_2=0;
		int new_migration_selection_conf_2=0;

		sscanf(line," %d %d %d %d %d %d %d
",&op,&new_migration_rate,&new_migration_size,&new_migration_selection_1,&new_migration
_selection_conf_1,&new_migration_selection_2,&new_migration_selection_conf_2);

		assert(new_migration_rate>0);
		assert(new_migration_size>0);
		assert(new_migration_selection_1>=0);
		assert(new_migration_selection_conf_1>=0);
		assert(new_migration_selection_2>=0);
		assert(new_migration_selection_conf_2>=0);

		migration_rate=new_migration_rate;
		migration_size=new_migration_size;
		migration_selection_1=new_migration_selection_1;
		migration_selection_conf_1=new_migration_selection_conf_1;
		migration_selection_2=new_migration_selection_2;
		migration_selection_conf_2=new_migration_selection_conf_2;
	}

	void Inter_Operator::RefreshState(const StateCenter& _sc) const
	{
		_sc.set_contents_state_variable("_migration_rate",(char
*)&migration_rate,1,sizeof(migration_rate));
		_sc.set_contents_state_variable("_migration_size",(char
*)&migration_size,1,sizeof(migration_size));
		_sc.set_contents_state_variable("_migration_selection_1",(char
*)&migration_selection_1,1,sizeof(migration_selection_1));
		_sc.set_contents_state_variable("_migration_selection_2",(char
*)&migration_selection_2,1,sizeof(migration_selection_2));
```

```
		_sc.set_contents_state_variable("_migration_selection_conf_1",(char
*)&migration_selection_conf_1,1,sizeof(migration_selection_conf_1));
		_sc.set_contents_state_variable("_migration_selection_conf_2",(char
*)&migration_selection_conf_2,1,sizeof(migration_selection_conf_2));
	}

	void Inter_Operator::UpdateFromState(const StateCenter& _sc)
	{
		unsigned long nitems,length;
		_sc.get_contents_state_variable("_migration_rate",(char
*)&migration_rate,nitems,length);
		_sc.get_contents_state_variable("_migration_size",(char
*)&migration_size,nitems,length);
		_sc.get_contents_state_variable("_migration_selection_1",(char
*)&migration_selection_1,nitems,length);
		_sc.get_contents_state_variable("_migration_selection_2",(char
*)&migration_selection_2,nitems,length);
		_sc.get_contents_state_variable("_migration_selection_conf_1",(char
*)&migration_selection_conf_1,nitems,length);
		_sc.get_contents_state_variable("_migration_selection_conf_2",(char
*)&migration_selection_conf_2,nitems,length);
	}

	ostream& operator<< (ostream& os, const Inter_Operator& inter)
	{
		switch (inter.number_operator())
		{
			case 0: os << (Migration&)inter;break;
		}
		return os;
	}

	Inter_Operator::~Inter_Operator()
	{}

// Migration ----------------------------------------------------------

	Migration::Migration(const Direction dir):Inter_Operator(0,dir)
	{}

	void Migration::execute(Population& pop,const unsigned long
current_generation,NetStream& _netstream,const bool synchronized,const unsigned int
check_asynchronous) const
	{
```

```
		Solution* solution_to_send;
		Solution* solution_received;
		Solution* solution_to_remplace;
		bool need_to_revaluate=false;
		int mypid;

		int nb_proc=_netstream.pnumber();  // Get the number of processes running

		mypid=_netstream.my_pid();

		int to   = (mypid + 1) % nb_proc;                    // Source (from) and Target (to) of
processes
		int from = (nb_proc + mypid - 1) % nb_proc;

		// process number 0 is only to store the global state
		if (to==0) to=1;
		if (from==0) from=nb_proc - 1;

		_netstream << set_target(to)  << set_source(from)
				<< get_target(&to) << get_source(&from);

		if ( (current_generation % migration_rate) == 0
		  && (current_generation!=pop.setup().nb_evolution_steps())) // in this generation
this operator have to be applied
		{

	pop.setup().pool().selector(migration_selection_1).prepare(pop.fitness_values(),false);

			_netstream << pack_begin;
			for (int i=0;i<migration_size;i++)
			{
				// select individual to send
				solution_to_send = pop.parents()
[pop.setup().pool().selector(migration_selection_1).select_one(


pop.parents(),pop.offsprings(),pop.fitness_values(),migration_selection_conf_1,false).index];

				_netstream << *solution_to_send;
			}
			_netstream << pack_end;

			if (synchronized)    // synchronous mode: blocked until data are received
```

```
                        {

        pop.setup().pool().selector(migration_selection_2).prepare(pop.fitness_values(),true);
                                _netstream << wait(packed);
                                _netstream << pack_begin;
                                for (int i=0;i<migration_size;i++)
                                {
                    // select individual to be remplaced
                                        struct individual ind;
                                        ind =
pop.setup().pool().selector(migration_selection_2).select_one(

        pop.parents(),pop.offsprings(),pop.fitness_values(),migration_selection_conf_2,true);
                                        solution_to_remplace = pop.parents()[ind.index];
                                        solution_received=new Solution(solution_to_remplace-
>pbm());
                                        _netstream >> *solution_received;

                                        // remplace policy
                                        if ((solution_received->fitness()<=solution_to_remplace-
>fitness() && direction==minimize)
                                         || (solution_received->fitness()>=solution_to_remplace-
>fitness() && direction==maximize))
                                        {
                                                need_to_revaluate=true;
                                                for(int j = 0; j < pop.parents().size(); j++)
                                                {
                                                        if(pop.fitness_values()[j].index == ind.index)
                                                        {
                                                                pop.fitness_values()[j].change =
true;
                                                                *pop.parents()[ind.index] =
*solution_received;
                                                        }
                                                }
                                        }
                                        delete(solution_received);

                                }
                                _netstream << pack_end;

                        }
                } // end if
```

```
                if (!synchronized && ((current_generation % check_asynchronous) ==0))
                { // asynchronous mode: if there are not data, continue;
                  // but, if there are data, i have to receive it
                        int pending=false;
                        _netstream._probe(packed,pending);
                        if (pending)
                        {

        pop.setup().pool().selector(migration_selection_2).prepare(pop.fitness_values(),true);

                                        _netstream << pack_begin;
                                        for (int i=0;i<migration_size;i++)
                                        {
                                                pending=false;
                                                _netstream._probe(regular,pending);
                                                if (!pending) break;

                                                // select individual to be remplaced
                                                struct individual ind;
                                                ind =
pop.setup().pool().selector(migration_selection_2).select_one(

        pop.parents(),pop.offsprings(),pop.fitness_values(),migration_selection_conf_2,true);
                                                solution_to_remplace = pop.parents()[ind.index];
                                                solution_received=new Solution(solution_to_remplace-
>pbm());
                                                _netstream >> *solution_received;

                                                // remplace policy
                                                if ((solution_received->fitness()<=solution_to_remplace-
>fitness() && direction==minimize)
                                                 || (solution_received->fitness()>=solution_to_remplace-
>fitness() && direction==maximize))
                                                {
                                                        need_to_revaluate=true;

                                                        for(int j = 0; j < pop.parents().size(); j++)
                                                        {
                                                                if(pop.fitness_values()[j].index == ind.index)
                                                                {
                                                                        pop.fitness_values()[j].change =
true;
                                                                        *pop.parents()[ind.index] =
*solution_received;
```

```
                                                }
                                        }
                                }
                                delete(solution_received);
                        } // end for
                        _netstream << pack_begin;
                } // end if
        }

        if (need_to_revaluate) pop.evaluate_parents();
    }

    ostream& operator<< (ostream& os, const Migration& migration)
    {
        os << "Migration."
           << endl << "\t" <<  " Rate: " << migration.migration_rate
           << endl << "\t" <<  " Size: " << migration.migration_size
           << endl << "\t" <<  " Selection 1: " << migration.migration_selection_1
           << endl << "\t" <<  " Selection 1 Parameter: " <<
migration.migration_selection_conf_1
           << endl << "\t" <<  " Selection 2: " << migration.migration_selection_2
           << endl << "\t" <<  " Selection 2 Parameter: " <<
migration.migration_selection_conf_2;
        return os;
    }

    Migration::~Migration()
    {}

// Selection ----------------------------------------------------------

    Selection::Selection(const Direction dir):_number_selection(0),direction(dir)
    {}

    Selection::Selection(const unsigned int _number_sel, const Direction
dir):_number_selection(_number_sel),direction(dir)
    {}

  void Selection::prepare(Rarray<struct individual>& fitness_values,const bool remplace)
    {}

  struct individual Selection::select_one(const Rarray<Solution*>& to_select_1,const
Rarray<Solution*>& to_select_2,const Rarray<struct individual>& fitness_values,const unsigned
int dummy,const bool remplace) const
```

```
{          // select a random individual
        return fitness_values[rand_int(0,fitness_values.size()-1)];
}

unsigned int Selection::number_selection() const
{
        return _number_selection;
}

ostream& operator<< (ostream& os, const Selection& sel)
{
        switch (sel.number_selection())
        {
                case 0: os << "Random Selection"; break;
                case 1: os << (Selection_Tournament&)sel; break;
                case 2: os << (Selection_Roulette_Wheel&)sel; break;
                case 3: os << (Selection_Rank&)sel; break;
                case 4: os << (Selection_Best&)sel; break;
                case 5: os << (Selection_Worst&)sel; break;
        }
        return os;
}

Selection::~Selection()
{}

// Selection_Tournament----------------------------------------------------

        Selection_Tournament::Selection_Tournament(const Direction dir):Selection(1,dir)
        {}

        struct individual Selection_Tournament::select_one(const Rarray<Solution*>&
to_select_1,const Rarray<Solution*>& to_select_2,const Rarray<struct individual>&
fitness_values,const unsigned int tournament_size, const bool remplace) const
        {
                unsigned int best_sol=0;
                double best_fitness=((-1) * direction * infinity());
                unsigned int index;

                if (remplace) best_fitness = -1 * best_fitness;

                unsigned int new_tournament_size=tournament_size;
                if (tournament_size==0) new_tournament_size=1;
```

```
		for (int i=0;i<new_tournament_size;i++)
		{
			index=rand_int(0,fitness_values.size()-1);

			switch (direction)
			{
				case (minimize): if (((!remplace) &&
(fitness_values[index].fitness<best_fitness))
                                                                || ((remplace) &&
(fitness_values[index].fitness>best_fitness)))
                                                                {
                                                                        best_sol = index;
                                                                        best_fitness =
fitness_values[index].fitness;
                                                                }
                                                                break;
                                case (maximize): if (((!remplace) &&
(fitness_values[index].fitness>best_fitness))
                                                                || ((remplace) &&
(fitness_values[index].fitness<best_fitness)))
                                                                {
                                                                        best_sol = index;
                                                                        best_fitness =
fitness_values[index].fitness;
                                                                }
                                                                break;
			}
		}

		return fitness_values[best_sol];
	}

	ostream& operator<< (ostream& os, const Selection_Tournament& sel)
	{
		os << "Tournament Selection.";
		return os;
	}

	Selection_Tournament::~Selection_Tournament()
	{}

// Selection_Roulette_Wheel ---------------------------------------------------
```

```
	Selection_Roulette_Wheel::Selection_Roulette_Wheel(const Direction
dir):Selection(2,dir)
	{}

	void Selection_Roulette_Wheel::prepare(Rarray<struct individual>& fitness_values,const
bool remplace)
	{
		double overall_fitness=0.0;

		// inverts fitness values to select less fitness individuals with a high probability
		if ((direction==maximize && (remplace)) || ((direction==minimize) && (!
(remplace))))
		{
			// fitness assigned if the fitness value is 0 in this case
			double value_if_zero=MAXDOUBLE;
			unsigned int nb_zeros=0;

			for (int i=0;i<fitness_values.size();i++)
			{
				if (fitness_values[i].fitness!=0)
					value_if_zero-=fitness_values[i].fitness;
				else
					nb_zeros++;
			}

			value_if_zero=value_if_zero/nb_zeros;

			// Warning !! if fitness is 0 (1/0 ?)
			for (int i=0;i<fitness_values.size();i++)
			{
				if (fitness_values[i].fitness!=0)
					fitness_values[i].sel_parameter = (1 /
fitness_values[i].fitness );
				else
					fitness_values[i].sel_parameter = value_if_zero;
				overall_fitness+= fitness_values[i].sel_parameter;
			}
		}
		else
		{
			for (int i=0;i<fitness_values.size();i++)
			{
				fitness_values[i].sel_parameter = fitness_values[i].fitness;
				overall_fitness+= fitness_values[i].sel_parameter;
```

```
                        }

                }

                if (overall_fitness>MAXDOUBLE) overall_fitness=MAXDOUBLE;

                // calculate relative fitness
                double previous=0.0;
                for (int i=0;i<fitness_values.size();i++)
                {
                        fitness_values[i].sel_parameter =       (fitness_values[i].sel_parameter /
overall_fitness) + previous;
                        previous = fitness_values[i].sel_parameter;
                }
        }

        struct individual Selection_Roulette_Wheel::select_one(const Rarray<Solution*>&
to_select_1,const Rarray<Solution*>& to_select_2,const Rarray<struct individual>&
fitness_values,const unsigned int dummy, const bool remplace) const
        {
                double random_selected=rand01();
                int i=0;

                while (random_selected > fitness_values[i].sel_parameter )
                        i++;

                return fitness_values[i];
        }

        ostream& operator<< (ostream& os, const Selection_Roulette_Wheel& sel)
        {
                os << "Roulette Wheel Selection.";
                return os;
        }

        Selection_Roulette_Wheel::~Selection_Roulette_Wheel()
        {}

// Selection_Rank ------------------------------------------------

        int lessF(const struct individual &i1,const  struct individual &i2)
        {
                return i1.fitness < i2.fitness;
        }
```

```
	int greaterF(const struct individual &i1,const  struct individual &i2)
	{
		return i1.fitness > i2.fitness;
	}

	Selection_Rank::Selection_Rank(const Direction dir):Selection(3,dir)
	{}

	Selection_Rank::Selection_Rank(const unsigned int _number_sel, const Direction
dir):Selection(_number_sel,dir)
	{}

	void Selection_Rank::reset()
	{}

   void Selection_Rank::prepare(Rarray<struct individual>& fitness_values,const bool remplace)
	{
      reset();

		// sort individuals
		if (((direction==maximize) && (!(remplace))) || ((direction==minimize) &&
(remplace)))
			fitness_values.sort(greaterF);
		else
			fitness_values.sort(lessF);
	}

	struct individual Selection_Rank::select_one(const Rarray<Solution*>&
to_select_1,const Rarray<Solution*>& to_select_2,const Rarray<struct individual>&
fitness_values,const unsigned int portion,const bool remplace) const
	{

		unsigned int new_portion=portion;
		if (portion==0 || portion>100) new_portion=100;

		return fitness_values[rand_int(0,(( fitness_values.size() * new_portion )/ 100)-1)];
	}

	ostream& operator<< (ostream& os, const Selection_Rank& sel)
	{
		os << "Rank-Ordered Selection.";
		return os;
	}
```

```
	Selection_Rank::~Selection_Rank()
	{}

// Selection_Best -----------------------------------------------

	Selection_Best::Selection_Best(const Direction
dir):Selection_Rank(4,dir),selection_best_position(0)
	{}

	struct individual Selection_Best::select_one(const Rarray<Solution*>& to_select_1,const
Rarray<Solution*>& to_select_2,const Rarray<struct individual>& fitness_values,const unsigned
int position,const bool remplace) const
	{
		int position_to_return=position-1;
		if (position_to_return<0)
		{
			position_to_return = selection_best_position;
			selection_best_position++;
		}

		position_to_return=(int)position_to_return % fitness_values.size();

		return fitness_values[position_to_return];
	}

	void Selection_Best::reset()
	{
		selection_best_position=0;
	}

	ostream& operator<< (ostream& os, const Selection_Best& sel)
	{
		os << "Selection of best ordered individuals.";
		return os;
	}

	Selection_Best::~Selection_Best()
	{}

// Selection_Worst ----------------------------------------------

	Selection_Worst::Selection_Worst(const Direction
dir):Selection_Rank(5,dir),selection_worst_position(0)
```

```
        {}

    struct individual  Selection_Worst::select_one(const Rarray<Solution*>& to_select_1,const
Rarray<Solution*>& to_select_2,const Rarray<struct individual>& fitness_values,const unsigned
int position,const bool remplace) const
        {
                int position_to_return=position-1;
                if (position_to_return<0)
                {
                        position_to_return = selection_worst_position;
                        selection_worst_position++;
                }

                position_to_return=(int)position_to_return % fitness_values.size();

                int index=(fitness_values.size()-1) - position_to_return;
                return fitness_values[index];
        }

        void Selection_Worst::reset()
        {
                selection_worst_position=0;
        }

        ostream& operator<< (ostream& os, const Selection_Worst& sel)
        {
                os << "Selection of worst ordered individuals.";
                return os;
        }

        Selection_Worst::~Selection_Worst()
        {}

// Operator_Pool ---------------------------------------------------------------

        Operator_Pool::Operator_Pool(const Problem& pbm)
        {
      // introduces all operators and selections in lists

                // Index to be chosen in setup file
                //-----------------------------------
                // The Intra_Operators are introduced dimanicly in setup

                _selectors.append(new Selection(pbm.direction()));                      // 0
```

```
        _selectors.append(new Selection_Tournament(pbm.direction()));       // 1
        _selectors.append(new Selection_Roulette_Wheel(pbm.direction()));   // 2
        _selectors.append(new Selection_Rank(pbm.direction()));             // 3
        _selectors.append(new Selection_Best(pbm.direction()));             // 4
        _selectors.append(new Selection_Worst(pbm.direction()));            // 5

        _inter_operators.append(new Migration(pbm.direction()));            // 0
    }

    Intra_Operator& Operator_Pool::intra_operator(const unsigned int index) const
    {
        assert(index < _intra_operators.size());
        return _intra_operators[index];
    }

    Rlist<Intra_Operator>& Operator_Pool::intra_operators() const
    {
        return _intra_operators;
    }

    Selection& Operator_Pool::selector(const unsigned int index) const
    {
        assert(index < _selectors.size());
        return _selectors[index];
    }

    const Rlist<Selection>&  Operator_Pool::selectors() const
    {
        return _selectors;
    }

    Inter_Operator& Operator_Pool::inter_operator(const unsigned int index) const
    {
        assert(index < _inter_operators.size());
        return _inter_operators[index];
    }

const Rlist<Inter_Operator>& Operator_Pool::inter_operators() const
    {
        return _inter_operators;
    }

    Operator_Pool::~Operator_Pool()
    {}
```

```
// Solver (superclasse)-----------------------------------------------

        Solver::Solver (const Problem& pbm, const SetUpParams& setup)
        : problem(pbm),
          params(setup),
          _stat(),
          _userstat(),
          _sc(),
          current_population(pbm,setup),
          best_cost((-1) * pbm.direction() * infinity()),
          worst_cost((-1) * best_cost),
          best_solution(problem),
          average_cost(0.0),
          standard_deviation(0.0),
          time_spent_in_trial(0.0),
          total_time_spent(0.0),
          start_trial(0.0),
          start_global(0.0),
          _current_trial("_current_trial",_sc),
          _current_iteration("_current_iteration",_sc),
          _current_evaluations("_current_evaluations",_sc),
          _current_best_solution("_current_best_solution",_sc),
          _current_best_cost("_current_best_cost",_sc),
          _current_worst_cost("_current_worst_cost",_sc),
          _current_average_cost("_current_average_cost",_sc),
          _current_standard_deviation("_current_standard_deviation",_sc),
          _current_time_spent("_current_time_spent",_sc),
          _best_solution_trial("_best_sol_trial",_sc),
          _best_cost_trial("_best_cost_trial",_sc),
          _worst_cost_trial("_worst_cost_trial",_sc),
          _iteration_best_found_in_trial("_iteration_best_found_in_trial",_sc),
          _evaluations_best_found_in_trial("_evaluations_best_found_in_trial",_sc),
          _time_best_found_trial("_time_best_found_trial",_sc),
          _time_spent_trial("_time_spent_trial",_sc),
          _trial_best_found("_trial_best_found",_sc),
          _iteration_best_found("_iteration_best_found",_sc),
          _evaluations_best_found("_evaluations_best_found",_sc),
          _global_best_solution("_global_best_solution",_sc),
          _global_best_cost("_global_best_cost",_sc),
          _global_worst_cost("_global_worst_cost",_sc),
          _time_best_found("_time_best_found",_sc),
          _crossover_probability("_crossover_probability",_sc),
          _mutation_probability("_mutation_probability",_sc),
```

```
  _migration_rate("_migration_rate",_sc),
  _migration_size("_migration_size",_sc),
  _migration_selection_1("_migration_selection_1",_sc),
  _migration_selection_2("_migration_selection_2",_sc),
  _migration_selection_conf_1("_migration_selection_conf_1",_sc),
  _migration_selection_conf_2("_migration_selection_conf_2",_sc),
  _select_parents("_select_parents",_sc),
  _select_offsprings("_select_offsprings",_sc),
  _parameter_select_parents("_parameter_select_parents",_sc),
  _parameter_select_offsprings("_parameter_select_offsprings",_sc),
  _display_state("_display_state",_sc)
{
        current_trial(0);
        current_iteration(0);
        current_evaluations(0);
        current_best_solution(best_solution);
        current_best_cost(best_cost);
        current_worst_cost(worst_cost);
        current_average_cost(average_cost);
        current_standard_deviation(standard_deviation);
        current_time_spent(total_time_spent);
        best_solution_trial(best_solution);
        best_cost_trial(best_cost);
        worst_cost_trial(worst_cost);
        iteration_best_found_in_trial(0);
        evaluations_best_found_in_trial(0);
        time_best_found_trial(time_spent_in_trial);
        time_spent_trial(time_spent_in_trial);
        trial_best_found(0);
        iteration_best_found(0);
        evaluations_best_found(0);
        global_best_solution(best_solution);
        global_best_cost(best_cost);
        global_worst_cost(worst_cost);
        time_best_found(total_time_spent);
        float prob[MAX_PROB_PER_OP] = {0.0};
        crossover_probability(prob);
        mutation_probability(prob);
        char aux[] = "_user_op_probability";
        char nombre[30];
        for(int i = 0; i < MAX_OP_USER; i++)
        {
                sprintf(nombre,"%s%d",aux,i);
                _user_op_probability[i].set_name((char *)nombre);
```

```
                _sc.add(_user_op_probability[i]);
                user_op_probability(i,prob);
        }
        migration_rate(0);
        migration_size(0);
        migration_selection_1(0);
        migration_selection_2(0);
        migration_selection_conf_1(0);
        migration_selection_conf_2(0);
        select_parents(0);
        select_offsprings(0);
        parameter_select_parents(0);
        parameter_select_offsprings(0);
        display_state(setup.display_state());
}

int Solver::pid() const
{
        return 0;
}

bool Solver::end_trial() const
{
        return _end_trial;
}

void Solver::end_trial(bool et)
{
        _end_trial = et;
}

unsigned int Solver::current_trial() const
{
        unsigned int value=0;
        unsigned long nitems,length;
        _current_trial.get_contents((char *)&value, nitems, length);
        return value;
}

unsigned long Solver::current_iteration() const
{
        unsigned long value=0;
        unsigned long nitems,length;
        _current_iteration.get_contents((char *)&value, nitems, length);
```

```
        return value;
    }

    unsigned long Solver::current_evaluations() const
    {
        unsigned long value=0;
        unsigned long nitems,length;
        _current_evaluations.get_contents((char *)&value, nitems, length);
        return value;
    }

    Solution Solver::current_best_solution() const
    {
        Solution sol(problem);
        unsigned long nitems,length;
        char data_stored[_current_best_solution.get_nitems() +
_current_best_solution.get_length()];
        _current_best_solution.get_contents(data_stored, nitems, length);
        sol.to_Solution(data_stored);
        return sol;
    }

    double Solver::current_best_cost() const
    {
        double value=0.0;
        unsigned long nitems,length;
        _current_best_cost.get_contents((char *)&value, nitems, length);
        return value;
    }

    double Solver::current_worst_cost() const
    {
        double value=0.0;
        unsigned long nitems,length;
        _current_worst_cost.get_contents((char *)&value, nitems, length);
        return value;
    }

    double Solver::current_average_cost() const
    {
        double value=0.0;
        unsigned long nitems,length;
        _current_average_cost.get_contents((char *)&value, nitems, length);
        return value;
```

```
}

double Solver::current_standard_deviation() const
{
        double value=0.0;
        unsigned long nitems,length;
        _current_standard_deviation.get_contents((char *)&value, nitems, length);
        return value;
}

float  Solver::current_time_spent() const
{
        float value=0.0;
        unsigned long nitems,length;
        _current_time_spent.get_contents((char *)&value, nitems, length);
        return value;
}

Solution  Solver::best_solution_trial() const
{
        Solution sol(problem);
        char data_stored[_best_solution_trial.get_nitems() +
_best_solution_trial.get_length()];
        unsigned long nitems,length;
        _best_solution_trial.get_contents(data_stored, nitems, length);
        sol.to_Solution(data_stored);
        return sol;
}

double Solver::best_cost_trial() const
{
        double value=0.0;
        unsigned long nitems,length;
        _best_cost_trial.get_contents((char *)&value, nitems, length);
        return value;
}

double Solver::worst_cost_trial() const
{
        double value=0.0;
        unsigned long nitems,length;
        _worst_cost_trial.get_contents((char *)&value, nitems, length);
        return value;
}
```

```
unsigned int Solver::iteration_best_found_in_trial() const
{
	unsigned int value=0;
	unsigned long nitems,length;
	_iteration_best_found_in_trial.get_contents((char *)&value, nitems, length);
	return value;
}

unsigned int Solver::evaluations_best_found_in_trial() const
{
	unsigned int value=0;
	unsigned long nitems,length;
	_evaluations_best_found_in_trial.get_contents((char *)&value, nitems, length);
	return value;
}

float Solver::time_best_found_trial() const
{
	float value=0.0;
	unsigned long nitems,length;
	_time_best_found_trial.get_contents((char *)&value, nitems, length);
	return value;
}

float Solver::time_spent_trial() const
{
	float value=0.0;
	unsigned long nitems,length;
	_time_spent_trial.get_contents((char *)&value, nitems, length);
	return value;
}

unsigned int Solver::trial_best_found() const
{
	unsigned int value=0;
	unsigned long nitems,length;
	_trial_best_found.get_contents((char *)&value, nitems, length);
	return value;
}

unsigned int Solver::iteration_best_found() const
{
	unsigned int value=0;
```

```
		unsigned long nitems,length;
		_iteration_best_found.get_contents((char *)&value, nitems, length);
		return value;
	}

	unsigned int Solver::evaluations_best_found() const
	{
		unsigned int value=0;
		unsigned long nitems,length;
		_evaluations_best_found.get_contents((char *)&value, nitems, length);
		return value;
	}

	Solution Solver::global_best_solution() const
	{
		Solution sol(problem);
		char data_stored[_global_best_solution.get_nitems() +
_global_best_solution.get_length()];
		unsigned long nitems,length;
		_global_best_solution.get_contents(data_stored, nitems, length);
		sol.to_Solution(data_stored);
		return sol;
	}

	double Solver::global_best_cost() const
	{
		double value=0.0;
		unsigned long nitems,length;
		_global_best_cost.get_contents((char *)&value, nitems, length);
		return value;
	}

	double Solver::global_worst_cost() const
	{
		double value=0.0;
		unsigned long nitems,length;
		_global_worst_cost.get_contents((char *)&value, nitems, length);
		return value;
	}

	float Solver::time_best_found() const
	{
		float value=0.0;
		unsigned long nitems,length;
```

```
		_time_best_found.get_contents((char *)&value, nitems, length);
		return value;
	}

	int Solver::display_state() const
	{
		int value=0;
		unsigned long nitems,length;
		_display_state.get_contents((char *)&value, nitems, length);
		return value;
	}

	float *Solver::crossover_probability() const
	{
		float *current_probability = new float[MAX_PROB_PER_OP];
		unsigned long nitems,length;
		_sc.get_contents_state_variable("_crossover_probability",(char
*)&current_probability,nitems,length);
		return current_probability;
	}

	float *Solver::mutation_probability() const
	{
		float *current_probability = new float[MAX_PROB_PER_OP];
		unsigned long nitems,length;
		_sc.get_contents_state_variable("_mutation_probability",(char
*)&current_probability,nitems,length);
		return current_probability;
	}

	float *Solver::user_op_probability(const int index) const
	{
		float *current_probability = new float[MAX_PROB_PER_OP];
		unsigned long nitems,length;
		char aux[30] = "_user_op_probability";
		sprintf(aux,"%s%d",aux,index);

		_sc.get_contents_state_variable(aux,(char *)&current_probability,nitems,length);
		return current_probability;
	}

	unsigned int Solver::migration_rate() const
	{
		unsigned int rate=0;
```

```
		unsigned long nitems,length;
		_sc.get_contents_state_variable("_migration_rate",(char *)&rate,nitems,length);
		return rate;
	}

	unsigned int Solver::migration_size() const
	{
		unsigned int size=0;
		unsigned long nitems,length;
		_sc.get_contents_state_variable("_migration_size",(char *)&size,nitems,length);
		return size;
	}

	unsigned int Solver::migration_selection_1() const
	{
		unsigned int selection_1=0;
		unsigned long nitems,length;
		_sc.get_contents_state_variable("_migration_selection_1",(char
*)&selection_1,nitems,length);
		return selection_1;
	}

	unsigned int Solver::migration_selection_2() const
	{
		unsigned int selection_2=0;
		unsigned long nitems,length;
		_sc.get_contents_state_variable("_migration_selection_2",(char
*)&selection_2,nitems,length);
		return selection_2;
	}

	unsigned int Solver::migration_selection_conf_1() const
	{
		unsigned int selection_conf_1=0;
		unsigned long nitems,length;
		_sc.get_contents_state_variable("_migration_selection_conf_1",(char
*)&selection_conf_1,nitems,length);
		return selection_conf_1;
	}

	unsigned int Solver::migration_selection_conf_2() const
	{
		unsigned int selection_conf_2=0;
		unsigned long nitems,length;
```

```
		_sc.get_contents_state_variable("_migration_selection_conf_2",(char
*)&selection_conf_2,nitems,length);
		return selection_conf_2;
	}

	unsigned int Solver::select_parents() const
	{
		unsigned int select_parents=0;
		unsigned long nitems,length;
		_sc.get_contents_state_variable("_select_parents",(char
*)&select_parents,nitems,length);
		return select_parents;
	}

	unsigned int Solver::select_offprings() const
	{
		unsigned int select_offsprings=0;
		unsigned long nitems,length;
		_sc.get_contents_state_variable("_select_offsprings",(char
*)&select_offsprings,nitems,length);
		return select_offsprings;
	}

	unsigned int Solver::parameter_select_parents() const
	{
		unsigned int parameter_select_parents;
		unsigned long nitems,length;
		_sc.get_contents_state_variable("_parameter_select_parents",(char
*)&parameter_select_parents,nitems,length);
		return parameter_select_parents;
	}

	unsigned int Solver::parameter_select_offsprings() const
	{
		unsigned int parameter_select_offsprings;
		unsigned long nitems,length;
		_sc.get_contents_state_variable("_parameter_select_offsprings",(char
*)&parameter_select_offsprings,nitems,length);
		return parameter_select_offsprings;
	}

	void Solver::current_trial(const unsigned int value)
	{
		_current_trial.set_contents((char *)&value,1,sizeof(int));
```

```
}

void Solver::current_iteration(const unsigned long value)
{
	_current_iteration.set_contents((char *)&value,1,sizeof(long));
}

void Solver::current_evaluations(const unsigned long value)
{
	_current_evaluations.set_contents((char *)&value,1,sizeof(long));
}

void Solver::current_best_solution(const Solution& sol)
{
	//cout << "solucion" << endl << sol << endl;
_current_best_solution.set_contents(sol.to_String(),1,sol.size());
}

void Solver::current_best_cost(const double value)
{
	_current_best_cost.set_contents((char *)&value,1,sizeof(double));
}

void Solver::current_worst_cost(const double value)
{
	_current_worst_cost.set_contents((char *)&value,1,sizeof(double));
}

void Solver::current_average_cost(const double value)
{
	_current_average_cost.set_contents((char *)&value,1,sizeof(double));
}

void Solver::current_standard_deviation(const double value)
{
	_current_standard_deviation.set_contents((char *)&value,1,sizeof(double));
}

void Solver::current_time_spent(const float value)
{
	_current_time_spent.set_contents((char *)&value,1,sizeof(float));
}

void Solver::best_solution_trial(const Solution& sol)
```

```
{
	_best_solution_trial.set_contents(sol.to_String(),1,sol.size());
}

void Solver::best_cost_trial(const double value)
{
	_best_cost_trial.set_contents((char *)&value,1,sizeof(double));
}

void Solver::worst_cost_trial(const double value)
{
	_worst_cost_trial.set_contents((char *)&value,1,sizeof(double));
}

void Solver::iteration_best_found_in_trial(const unsigned int value)
{
	_iteration_best_found_in_trial.set_contents((char *)&value,1,sizeof(int));
}

void Solver::evaluations_best_found_in_trial(const unsigned int value)
{
	_evaluations_best_found_in_trial.set_contents((char *)&value,1,sizeof(int));
}

void Solver::time_best_found_trial(const float value)
{
	_time_best_found_trial.set_contents((char *)&value,1,sizeof(float));
}

void Solver::time_spent_trial(const float value)
{
	_time_spent_trial.set_contents((char *)&value,1,sizeof(float));
}

void Solver::trial_best_found(const unsigned int value)
{
	_trial_best_found.set_contents((char *)&value,1,sizeof(int));
}

void Solver::iteration_best_found(const unsigned int  value)
{
	_iteration_best_found.set_contents((char *)&value,1,sizeof(int));
}
```

```
	void Solver::evaluations_best_found(const unsigned int  value)
	{
		_evaluations_best_found.set_contents((char *)&value,1,sizeof(int));
	}

	void Solver::global_best_solution(const Solution& sol)
	{
		//cout << sol.to_String() << endl;
		_global_best_solution.set_contents(sol.to_String(),1,sol.size());
	}

	void Solver::global_best_cost(const double value)
	{
		_global_best_cost.set_contents((char *)&value,1,sizeof(double));
	}

	void Solver::global_worst_cost(const double value)
	{
		_global_worst_cost.set_contents((char *)&value,1,sizeof(double));
	}

	void Solver::time_best_found(const float value)
	{
		_time_best_found.set_contents((char *)&value,1,sizeof(float));
	}

	void Solver::display_state(const int value)
	{
		_display_state.set_contents((char *)&value,1,sizeof(float));
	}

	void Solver::crossover_probability(const float *new_probability)
	{
		_sc.set_contents_state_variable("_crossover_probability",(char
*)new_probability,MAX_PROB_PER_OP,sizeof(float));
	}

	void Solver::mutation_probability(const float *new_probability)
	{
		_sc.set_contents_state_variable("_mutation_probability",(char
*)new_probability,MAX_PROB_PER_OP,sizeof(float));
	}

	void Solver::user_op_probability(const int index, const float *new_probability)
```

```
	{
		char aux[30] = "_user_op_probability";
		char aux2[2] = " ";
		sprintf(aux2,"%d",index);
		strcat(aux,aux2);
		//sprintf(aux,"%s%d",aux,index);
		_sc.set_contents_state_variable(aux,(char
*)new_probability,MAX_PROB_PER_OP,sizeof(float));
	}

	void Solver::migration_rate(const unsigned int rate)
	{
		_sc.set_contents_state_variable("_migration_rate",(char *)&rate,1,sizeof(int));
	}

	void Solver::migration_size(const unsigned int size)
	{
		_sc.set_contents_state_variable("_migration_size",(char *)&size,1,sizeof(int));
	}

	void Solver::migration_selection_1(const unsigned int selection_1)
	{
		_sc.set_contents_state_variable("_migration_selection_1",(char
*)&selection_1,1,sizeof(int));
	}

	void Solver::migration_selection_2(const unsigned int selection_2)
	{
		_sc.set_contents_state_variable("_migration_selection_2",(char
*)&selection_2,1,sizeof(int));
	}

	void Solver::migration_selection_conf_1(const unsigned int selection_conf_1)
	{
		_sc.set_contents_state_variable("_migration_selection_conf_1",(char
*)&selection_conf_1,1,sizeof(int));
	}

	void Solver::migration_selection_conf_2(const unsigned int selection_conf_2)
	{
		_sc.set_contents_state_variable("_migration_selection_conf_2",(char
*)&selection_conf_2,1,sizeof(int));
	}
```

```
	void Solver::select_parents(const unsigned int selection)
	{
		_sc.set_contents_state_variable("_select_parents",(char
*)&selection,1,sizeof(int));
	}

	void Solver::select_offsprings(const unsigned int selection)
	{
		_sc.set_contents_state_variable("_select_offsprings",(char
*)&selection,1,sizeof(int));
	}

	void Solver::parameter_select_parents(const unsigned int value)
	{
		_sc.set_contents_state_variable("_parameter_select_parents",(char
*)&value,1,sizeof(int));
	}

	void Solver::parameter_select_offsprings(const unsigned int value)
	{
		_sc.set_contents_state_variable("_parameter_select_offsprings",(char
*)&value,1,sizeof(int));
	}

	Statistics& Solver::statistics()
	{
		return _stat;
	}

	UserStatistics& Solver::userstatistics()
	{
		return _userstat;
	}

	Population& Solver::population()
	{
		return current_population;
	}

	const SetUpParams& Solver::setup() const
	{
		return params;
	}
```

```
const Problem& Solver::pbm() const
{
	return problem;
}

void Solver::KeepHistory(const Solution& best_sol,const double best_cost,const double
worst_cost,const float time_spent_in_trial,const float total_time_spent)
{
	bool betterG=false;
	bool worseG=false;
	bool betterT=false;
	bool worseT=false;

   switch (problem.direction())
	{
		case minimize: betterG = (best_cost < global_best_cost() || (best_cost ==
global_best_cost() && time_spent_in_trial < time_best_found()));
				worseG  = (worst_cost > global_worst_cost());
				betterT = (best_cost < best_cost_trial() || (best_cost ==
best_cost_trial() && time_spent_in_trial < time_best_found_trial()));
				worseT  = (worst_cost > worst_cost_trial());
				break;
		case maximize: betterG = (best_cost > global_best_cost() || (best_cost ==
global_best_cost() && time_spent_in_trial < time_best_found()));
				worseG  = (worst_cost < global_worst_cost());
				betterT = (best_cost > best_cost_trial() || (best_cost ==
best_cost_trial() && time_spent_in_trial < time_best_found_trial()));
				worseT  = (worst_cost < worst_cost_trial());
				break;
	}

	if (betterT)
	{
		best_solution_trial(best_sol);
		best_cost_trial(best_cost);
		time_best_found_trial(time_spent_in_trial);
		iteration_best_found_in_trial(current_iteration());
		evaluations_best_found_in_trial(current_evaluations());
		if (betterG)
		{
			//cout << best_cost << " mucho menos que " <<
global_best_cost() << endl;
			//cout << best_sol << endl;
			//cout << " antes de sustituir " << global_best_solution() << endl;
```

```
                                global_best_solution(best_sol);
                                //cout << best_cost << " ahora es " << global_best_solution() <<
endl;
                                global_best_cost(best_cost);
                                time_best_found(time_spent_in_trial);
                                trial_best_found(current_trial());
                                iteration_best_found(current_iteration());
                                evaluations_best_found(current_evaluations());
                        }
                }

                if (worseT)
                {
                        worst_cost_trial(worst_cost);
                        if (worseG)
                                global_worst_cost(worst_cost);
                }
        }

        StateCenter *Solver::GetState()
        {
                return &_sc;
        }

        void Solver::RefreshState()
        {
           current_best_solution(best_solution);
                current_best_cost(best_cost);
                current_worst_cost(worst_cost);
                current_average_cost(average_cost);
                current_standard_deviation(standard_deviation);
                current_time_spent(total_time_spent);
                time_spent_trial(time_spent_in_trial);

        KeepHistory(best_solution,best_cost,worst_cost,time_spent_in_trial,total_time_spent);
        }

        void Solver::RefreshCfgState()
        {
                for (int i=0;i<params.pool().intra_operators().size();i++)
                        params.pool().intra_operator(i).RefreshState(_sc);
                for (int i=0;i<params.pool().inter_operators().size();i++)
                        params.pool().inter_operator(i).RefreshState(_sc);
                params.RefreshState(_sc);
```

```
	}

	void Solver::UpdateFromState()
	{
		best_solution=current_best_solution();
		best_cost=current_best_cost();
		worst_cost=current_worst_cost();
		average_cost=current_average_cost();
		standard_deviation=current_standard_deviation();
		total_time_spent=current_time_spent();
		time_spent_in_trial=time_spent_trial();

	KeepHistory(best_solution,best_cost,worst_cost,time_spent_in_trial,total_time_spent);
	}

	void Solver::UpdateFromCfgState()
	{
		for (int i=0;i<params.pool().intra_operators().size();i++)
			params.pool().intra_operator(i).UpdateFromState(_sc);
		for (int i=0;i<params.pool().inter_operators().size();i++)
			params.pool().inter_operator(i).UpdateFromState(_sc);
		params.UpdateFromState(_sc);
	}

	void Solver::show_state() const
	{
		cout << endl << " Current State ----------------------------------------" << endl;
/*		cout << endl << "Selection parents    -> " << select_parents();
		cout << endl << "Parameter of selection: " << parameter_select_parents();
		cout << endl << "Selection offsprings -> " << select_offprings();
		cout << endl << "Parameter of selection: " << parameter_select_offsprings() <<
endl;
		cout << endl << "Crossover_probability: " << crossover_probability();
		cout << endl << "Mutation_probability: " << mutation_probability();
		cout << endl << "User_Operator_probability: " << user_op_probability(0);
		cout << endl << "Migration_rate: " << migration_rate();
		cout << endl << "Migration_size: " << migration_size();
		cout << endl << "Migration_selection_1: " << migration_selection_1();
		cout << endl << "Migration_selection_conf_1: " << migration_selection_conf_1();
		cout << endl << "Migration_selection_2: " << migration_selection_2();
		cout << endl << "Migration_selection_conf_2: " << migration_selection_conf_2()
<< endl;
*/		cout << endl << "Current trial: " << current_trial();
		cout << endl << "Current iteration: " << current_iteration();
```

```
		cout << endl << "Current evaluations: " << current_evaluations();
		cout << endl << "Current best cost: " << current_best_cost();
		cout << endl << "Current worst cost: " << current_worst_cost();
		cout << endl << "Current Average cost: " << current_average_cost();
		cout << endl << "Current Standard Deviation: " << current_standard_deviation();
		cout << endl << endl <<  "Trial: ";
		cout << endl << "Best cost trial: " << best_cost_trial();
		cout << endl << "Worst cost trial: " << worst_cost_trial();
		cout << endl << "Iteration best found in trial: " << iteration_best_found_in_trial();
		cout << endl << "Evaluations best found in trial: " <<
evaluations_best_found_in_trial();
		cout << endl << "Time best found trial: " << time_best_found_trial();
		cout << endl << "Time spent in trial: " << time_spent_trial();
		cout << endl << endl << "Global: ";
		cout << endl << "Global best cost: " << global_best_cost();
		cout << endl << "Global worst cost: " << global_worst_cost();
		cout << endl << "Trial best found: " << trial_best_found();
		cout << endl << "Iteration best found: " << iteration_best_found();
		cout << endl << "Evaluations best found: " << evaluations_best_found();
		cout << endl << "Time best found: " << time_best_found();
		cout << endl << endl << "Best Solution: " << endl << global_best_solution();
		cout << "Best Solution Fitness: " << global_best_solution().fitness() << endl;
		cout << "Current time spent (so far): " << current_time_spent() << endl;
		cout << "Current time spent in seconds (so far): " <<
current_time_spent()/1000000 << endl;
	}

	Solver::~Solver()
	{
		_sc.removeAll();
      }

// Solver sequencial ------------------------------------------------

	Solver_Seq::Solver_Seq (const Problem& pbm, const SetUpParams& setup)
	: Solver(pbm,setup)
	{
		random_seed(time(0));
		_end_trial=true;
	}

	Solver_Seq::~Solver_Seq ()
	{}
```

```
void Solver_Seq::StartUp()
{
	Population pop(problem,params);
	pop.initialize();
	StartUp(pop);
}

void Solver_Seq::StartUp(const Population& pop)
{
	start_trial=_used_time();
	start_global=total_time_spent;

	current_trial(current_trial()+1);
	current_iteration(0);
	current_evaluations(pop.evaluations());

	// initialize state variables in the current trial

	Solution initial_solution(problem);

	time_spent_in_trial=0.0;
	best_cost_trial((-1) * problem.direction() * infinity());
	worst_cost_trial((-1) * best_cost_trial());
	best_solution_trial(initial_solution);
	time_best_found_trial(0.0);

	current_population=pop;
	current_population.evaluate_parents();

	// gets current interesting values in the current population

	best_cost=current_population.best_cost();
	best_solution=current_population.best_solution();
	worst_cost=current_population.worst_cost();
	average_cost=current_population.average_cost();
	standard_deviation=current_population.standard_deviation();

	// refresh state with these values
	RefreshState();
	RefreshCfgState();

	_stat.update(*this);
	_userstat.update(*this);
```

```
	if (display_state())
		show_state();
}

void Solver_Seq::DoStep()
{
	current_iteration(current_iteration()+1);
	current_population.evolution();
	current_evaluations(current_population.evaluations());
	// gets current interesting values in the current population

	best_cost=current_population.best_cost();
	best_solution=current_population.best_solution();
	//cout << "Mejor sol 1" << best_solution << endl;
	worst_cost=current_population.worst_cost();
	average_cost=current_population.average_cost();
	standard_deviation=current_population.standard_deviation();

	time_spent_in_trial = _used_time(start_trial);
	total_time_spent    = start_global + time_spent_in_trial;

	// refresh state with these values
	RefreshState();
	RefreshCfgState();
	//cout << "Mejor sol 2" << best_solution << endl;
	if( (current_iteration() % params.refresh_global_state()) == 0)
		UpdateFromCfgState();

	//cout << "Mejor sol 3" << best_solution << endl;
	_stat.update(*this);
	_userstat.update(*this);

	if (display_state())
		show_state();
}

void Solver_Seq::run ()
{
	while (current_trial() < params.independent_runs()){
		run(params.nb_evolution_steps());
	}
}

void Solver_Seq::run (const unsigned long int nb_generations)
```

```
	{
		StartUp();
		while ((current_iteration() < nb_generations) && !
(terminateQ(problem,*this,params))){
			DoStep();
		}
	}

	void Solver_Seq::run (const Population& pop,const unsigned long int nb_generations)
	{
		StartUp(pop);
		while ((current_iteration() < nb_generations) && !
(terminateQ(problem,*this,params)))
			DoStep();

	}

	// Solver LAN ----------------------------------------------------------

	Solver_Lan::Solver_Lan (const Problem& pbm, const SetUpParams& setup,int argc,char
**argv):
		  _best_solution_trial(pbm), Solver(pbm,setup),_netstream(),
		 // Termination phase //
			final_phase(false),acum_evaluations(0),acum_iterations(0)

	{

		NetStream::init(argc,argv);
		mypid=_netstream.my_pid();
		random_seed(time(0) + (mypid+1));
		//  random_seed(time(0) + (mypid+1));
	}

	Solver_Lan::~Solver_Lan ()
	{
		NetStream::finalize();
	}

	int Solver_Lan::pid() const
	{
		return mypid;
	}

	NetStream& Solver_Lan::netstream()
```

```
{
        return _netstream;
}

void Solver_Lan::StartUp()
{
        Population pop(problem,params);
        pop.initialize();
        StartUp(pop);
}

void Solver_Lan::StartUp(const Population& pop)
{
        _netstream << barrier;
        // Termination phase //
        final_phase = false;
        acum_evaluations = 0;
        acum_iterations  = 0;

        start_trial=_used_time();
        start_global=total_time_spent;

        _end_trial=false;

        current_trial(current_trial()+1);
        current_iteration(0);
        current_evaluations(pop.evaluations());

        // initialize state variables in the current trial

        Solution initial_solution(problem);

        time_spent_in_trial=0.0;
        best_cost_trial((-1) * problem.direction() * infinity());
        worst_cost_trial((-1) * best_cost_trial());
        best_solution_trial(initial_solution);
        iteration_best_found_in_trial(0);
        time_best_found_trial(0.0);
        time_spent_trial(0.0);

        if (mypid!=0)
        {
current_population=pop;
                current_population.evaluate_parents();
```

```
                // gets current interesting values in the current population

                best_cost=current_population.best_cost();
                best_solution=current_population.best_solution();
                worst_cost=current_population.worst_cost();
                average_cost=current_population.average_cost();
                standard_deviation=current_population.standard_deviation();

                // refresh state with these values
                RefreshState();
                RefreshCfgState();

                send_local_state_to(mypid);

                _stat.update(*this);
                _userstat.update(*this);

        }
}

void Solver_Lan::DoStep()
{
        current_iteration(current_iteration()+1);
        current_population.evolution();
        current_evaluations(current_population.evaluations());

        // Termination phase //
        _netstream << set_source(0);
        int pending;
        _netstream._probe(regular, pending);
        if(pending)
                final_phase = true;
        ////////////////////////

        current_population.interchange(current_iteration(),_netstream);

        // gets current interesting values in the current population

        best_cost=current_population.best_cost();
        best_solution=current_population.best_solution();
        worst_cost=current_population.worst_cost();
        average_cost=current_population.average_cost();
        standard_deviation=current_population.standard_deviation();
```

```
        time_spent_in_trial = _used_time(start_trial);
        total_time_spent    = start_global + time_spent_in_trial;

        // refresh state with these values
        RefreshState();
        RefreshCfgState();

         // in this iteration i have to send data about my local state to the global state
        if ((int)current_iteration() % params.refresh_global_state() ==0)
        {
                send_local_state_to(mypid);
                UpdateFromCfgState();
        }

        _stat.update(*this);
        _userstat.update(*this);

        // if (display_state()) show_state();
}


void Solver_Lan::send_local_state_to(int _mypid)
{
        _netstream << set_target(0);
        _netstream << pack_begin
                        << _mypid
                        << current_trial()
                        << current_iteration()
                        << current_evaluations()
                        << best_cost_trial()
                        << best_solution_trial()
                        << iteration_best_found_in_trial()
                        << evaluations_best_found_in_trial()
                        << time_best_found_trial()
                        << worst_cost_trial()
                        << current_best_cost()
                        << current_best_solution()
                        << current_worst_cost()
                        << current_average_cost()
                        << current_standard_deviation()
                        << pack_end;
}
```

```
int Solver_Lan::receive_local_state()
{
	int r_pid=0;

	_netstream._wait(packed);

	_netstream << pack_begin
		>> r_pid
		>> _current_trial
		>> _current_iteration
		>> _current_evaluations
		>> _best_cost_trial
		>> _best_solution_trial
		>> _iteration_best_found_in_trial
		>> _evaluations_best_found_in_trial
		>> _time_best_found_in_trial
		>> _worst_cost_trial
		>> best_cost
		>> best_solution
		>> worst_cost
		>> average_cost
		>> standard_deviation
		<< pack_end;
	return r_pid;
}

void Solver_Lan::check_for_refresh_global_state() // Executed in process with pid 0
{
	unsigned int nb_finalized_processes=0;
	int received_pid;
	int nb_proc=_netstream.pnumber();

	_netstream << set_source(MPI_ANY_SOURCE);

	while (!_end_trial)
	{
		received_pid=0;
		received_pid=receive_local_state();

		current_trial(_current_trial);

		// refresh the global state with received data ( a local state )
		current_iteration(_iteration_best_found_in_trial);
		current_evaluations(_evaluations_best_found_in_trial);
```

```
KeepHistory(_best_solution_trial,_best_cost_trial,_worst_cost_trial,_time_best_found_in_trial,st
art_global + _time_best_found_in_trial);
                        // the process that has send data has finished the current trial
                        if (received_pid==-1)
                        {
                                // Termination phase //
                                if(!final_phase && terminateQ(problem,*this,params))
                                {
                                        acum_iterations = params.nb_evolution_steps() *
nb_finalized_processes;
                acum_evaluations = acum_iterations* params.population_additional_size() +

nb_finalized_processes*params.population_size();
                                        for(int i = 1; i < _netstream.pnumber(); i++)
                                        {
                                                _netstream << set_target(i);
                                                _netstream << 1;
                                        }
                                        final_phase = true;
                                }
                                nb_finalized_processes++;
                                acum_iterations += _iteration_best_found_in_trial;
                                acum_evaluations += _evaluations_best_found_in_trial;
                        }
                        if (nb_finalized_processes==nb_proc-1)
                                _end_trial=true;

                        current_iteration(_current_iteration);
                        current_evaluations(_current_evaluations);

                        time_spent_in_trial = _used_time(start_trial);
                        total_time_spent    = start_global + time_spent_in_trial;
                        RefreshState();
                        RefreshCfgState();

                        _stat.update(*this);
                        //_userstat.update(*this);

                        // display current global state
                        if (display_state()) show_state();
                } // end while
```

```
		// Actualizacion de las estadisticas // Termination phase //
		iteration_best_found_in_trial(acum_iterations/(_netstream.pnumber()-1));
		evaluations_best_found_in_trial(acum_evaluations/(_netstream.pnumber()-1));

		bool betterG=false;
		double best_cost = best_cost_trial();
	switch (problem.direction())
		{
			case minimize: betterG = (best_cost < global_best_cost() || (best_cost ==
global_best_cost() && time_best_found_trial() <= time_best_found()));

					break;
			case maximize: betterG = (best_cost > global_best_cost() || (best_cost ==
global_best_cost() && time_best_found_trial() <= time_best_found()));
					break;
		}

		if (betterG)
		{
			iteration_best_found(iteration_best_found_in_trial());
			evaluations_best_found(evaluations_best_found_in_trial());
		}

		RefreshState();
		RefreshCfgState();

		_stat.update(*this);
		_userstat.update(*this);

		// display the global state at the end of the current trial
		if (display_state()) show_state();
	}

	void Solver_Lan::run ()
	{
		while (current_trial() < params.independent_runs())
			run(params.nb_evolution_steps());
	}

	void Solver_Lan::run (const unsigned long int nb_generations)
	{
		StartUp();
		if (mypid!=0)
		{
```

```
                while (!final_phase && (current_iteration() < nb_generations) && !
(terminateQ(problem,*this,params)))
                        DoStep();
                send_local_state_to(-1);
        }
        else
        {
                check_for_refresh_global_state();
        }

        _netstream << barrier;
        reset();
}

void Solver_Lan::run (const Population& pop,const unsigned long int nb_generations)
{
        StartUp(pop);
        if (mypid!=0)
        {
                while (!final_phase && (current_iteration() < nb_generations) && !
(terminateQ(problem,*this,params)))
                        DoStep();
                send_local_state_to(-1);
        }
        else
        {
                check_for_refresh_global_state();
        }

        _netstream << barrier;
        reset();
}

void Solver_Lan::reset()
{
   Solution left_solution(problem);
        int i;
        _netstream << set_source(MPI_ANY_SOURCE);
        if (mypid!=0)
        {
                int pendingr = false;
                int pendingp = false;
                do
                {
```

```
                                pendingr = false;
                                pendingp = false;
                                _netstream._probe(regular,pendingr);
                                _netstream._probe(packed,pendingp);
                                if (pendingr) _netstream >> i;
                                if (pendingp) _netstream << pack_begin >> left_solution <<
pack_end;
                        } while (pendingr || pendingp);
                }
                _netstream << barrier;
        }

        // Solver WAN ------------------------------------------------------

        Solver_Wan::Solver_Wan (const Problem& pbm, const SetUpParams& setup,int
argc,char **argv):
                    _best_solution_trial(pbm), Solver(pbm,setup),_netstream(),
         // Termination phase //
         final_phase(false),acum_evaluations(0),acum_iterations(0)
                {

                NetStream::init(argc,argv);
                mypid=_netstream.my_pid();
                random_seed(time(0) + (mypid+1));
                //  random_seed(time(0) + (mypid+1));
        }

        Solver_Wan::~Solver_Wan ()
        {
                NetStream::finalize();
        }

        int Solver_Wan::pid() const
        {
                return mypid;
        }

        NetStream& Solver_Wan::netstream()
        {
                return _netstream;
        }

        void Solver_Wan::StartUp()
        {
```

```
        Population pop(problem,params);
        pop.initialize();
        StartUp(pop);
}

void Solver_Wan::StartUp(const Population& pop)
{
        _netstream << barrier;

    // Termination phase //
final_phase = false;
acum_evaluations = 0;
acum_iterations = 0;

        start_trial=_used_time();
        start_global=total_time_spent;

        _end_trial=false;

        current_trial(current_trial()+1);
        current_iteration(0);
        current_evaluations(pop.evaluations());

        // initialize state variables in the current trial

        Solution initial_solution(problem);

        time_spent_in_trial=0.0;
        best_cost_trial((-1) * problem.direction() * infinity());
        worst_cost_trial((-1) * best_cost_trial());
        best_solution_trial(initial_solution);
        iteration_best_found_in_trial(0);
        time_best_found_trial(0.0);
        time_spent_trial(0.0);

        if (mypid!=0)
        {
current_population=pop;
                current_population.evaluate_parents();

                // gets current interesting values in the current population

                best_cost=current_population.best_cost();
                best_solution=current_population.best_solution();
```

```
                worst_cost=current_population.worst_cost();
                average_cost=current_population.average_cost();
                standard_deviation=current_population.standard_deviation();

                // refresh state with these values
                RefreshState();
                RefreshCfgState();

                send_local_state_to(mypid);

                _stat.update(*this);
                _userstat.update(*this);

        }
    }

    void Solver_Wan::DoStep()
    {
        current_iteration(current_iteration()+1);
        current_population.evolution();
        current_evaluations(current_population.evaluations());
// Termination phase //
_netstream << set_source(0);
int pending;
_netstream._probe(regular, pending);
if(pending)
   final_phase = true;
//////////////////////

        current_population.interchange(current_iteration(),_netstream);

        // gets current interesting values in the current population

        best_cost=current_population.best_cost();
        best_solution=current_population.best_solution();
        worst_cost=current_population.worst_cost();
        average_cost=current_population.average_cost();
        standard_deviation=current_population.standard_deviation();

        time_spent_in_trial = _used_time(start_trial);
        total_time_spent    = start_global + time_spent_in_trial;

        // refresh state with these values
        RefreshState();
```

```
        RefreshCfgState();

         // in this iteration i have to send data about my local state to the global state
        if ((int)current_iteration() % params.refresh_global_state() ==0)
        {
                send_local_state_to(mypid);
                UpdateFromCfgState();
        }

        _stat.update(*this);
        _userstat.update(*this);

        // if (display_state()) show_state();
}


void Solver_Wan::send_local_state_to(int _mypid)
{
        _netstream << set_target(0);
        _netstream << pack_begin
                        << _mypid
                        << current_trial()
                        << current_iteration()
                        << current_evaluations()
                        << best_cost_trial()
                        << best_solution_trial()
                        << iteration_best_found_in_trial()
                        << evaluations_best_found_in_trial()
                        << time_best_found_trial()
                        << worst_cost_trial()
                        << current_best_cost()
                        << current_best_solution()
                        << current_worst_cost()
                        << current_average_cost()
                        << current_standard_deviation()
                        << pack_end;
}

int Solver_Wan::receive_local_state()
{
        int r_pid=0;

        _netstream._wait(packed);
```

```
		_netstream << pack_begin
				>> r_pid
				>> _current_trial
				>> _current_iteration
				>> _current_evaluations
				>> _best_cost_trial
				>> _best_solution_trial
				>> _iteration_best_found_in_trial
				>> _evaluations_best_found_in_trial
				>> _time_best_found_in_trial
				>> _worst_cost_trial
				>> best_cost
				>> best_solution
				>> worst_cost
				>> average_cost
				>> standard_deviation
				<< pack_end;
		return r_pid;
	}

	void Solver_Wan::check_for_refresh_global_state() // Executed in process with pid 0
	{
		unsigned int nb_finalized_processes=0;
		int received_pid;
		int nb_proc=_netstream.pnumber();

		_netstream << set_source(MPI_ANY_SOURCE);

		while (!_end_trial)
		{
			received_pid=0;
	received_pid=receive_local_state();

	current_trial(_current_trial);

	// refresh the global state with received data ( a local state )
	current_iteration(_iteration_best_found_in_trial);
	current_evaluations(_evaluations_best_found_in_trial);

KeepHistory(_best_solution_trial,_best_cost_trial,_worst_cost_trial,_time_best_found_in_trial,st
art_global + _time_best_found_in_trial);
	// the process that has send data has finished the current trial
	if (received_pid==-1)
	{
```

```
            // Termination phase //
            if(!final_phase && terminateQ(problem,*this,params))
            {
                  acum_iterations = params.nb_evolution_steps() * nb_finalized_processes;
                  acum_evaluations = acum_iterations* params.population_additional_size() +

nb_finalized_processes*params.population_size();
                  for(int i = 1; i < _netstream.pnumber(); i++)
                  {
                        _netstream << set_target(i);
                                                                  _netstream  << 1;
                   }
                   final_phase = true;
             }
             nb_finalized_processes++;
             acum_iterations += _iteration_best_found_in_trial;
             acum_evaluations += _evaluations_best_found_in_trial;
          }
          if (nb_finalized_processes==nb_proc-1)
                _end_trial=true;

          current_iteration(_current_iteration);
          current_evaluations(_current_evaluations);

          time_spent_in_trial = _used_time(start_trial);
          total_time_spent    = start_global + time_spent_in_trial;
          RefreshState();
          RefreshCfgState();

          _stat.update(*this);
          //_userstat.update(*this);

          // display current global state
          if (display_state()) show_state();
       } // end while

       // Update Stats // Termination phase //
       iteration_best_found_in_trial(acum_iterations/(_netstream.pnumber()-1));
       evaluations_best_found_in_trial(acum_evaluations/(_netstream.pnumber()-1));

       bool betterG=false;
       double best_cost = best_cost_trial();
       switch (problem.direction())
       {
```

```
        case minimize: betterG = (best_cost < global_best_cost() || (best_cost ==
global_best_cost() && time_best_found_trial() <= time_best_found()));
                break;
        case maximize: betterG = (best_cost > global_best_cost() || (best_cost ==
global_best_cost() && time_best_found_trial() <= time_best_found()));
                break;
    }

    if (betterG)
            {
        iteration_best_found(iteration_best_found_in_trial());
        evaluations_best_found(evaluations_best_found_in_trial());
            }

    RefreshState();
    RefreshCfgState();

    _stat.update(*this);
    _userstat.update(*this);

    // display the global state at the end of the current trial
    if (display_state()) show_state();
      }

      void Solver_Wan::run ()
      {
            while (current_trial() < params.independent_runs())
                  run(params.nb_evolution_steps());
      }

      void Solver_Wan::run (const unsigned long int nb_generations)
      {
            StartUp();
            if (mypid!=0)
            {
                  while (!final_phase && (current_iteration() < nb_generations) && !
(terminateQ(problem,*this,params)))
                        DoStep();
                  send_local_state_to(-1);
            }
            else
            {
                  check_for_refresh_global_state();
            }
```

```
        _netstream << barrier;
        reset();
    }

    void Solver_Wan::run (const Population& pop,const unsigned long int nb_generations)
    {
        StartUp(pop);
        if (mypid!=0)
        {
            while (!final_phase && (current_iteration() < nb_generations) && !
(terminateQ(problem,*this,params)))
                DoStep();
            send_local_state_to(-1);
        }
        else
        {
            check_for_refresh_global_state();
        }

        _netstream << barrier;
        reset();
    }

    void Solver_Wan::reset()
    {
       Solution left_solution(problem);
        int i;
        _netstream << set_source(MPI_ANY_SOURCE);
        if (mypid!=0)
        {
         int pendingr = false;
         int pendingp = false;
         do
         {
            pendingr = false;
            pendingp = false;
            _netstream._probe(regular,pendingr);
            _netstream._probe(packed,pendingp);
            if (pendingr) _netstream >> i ;
            if (pendingp) _netstream << pack_begin >> left_solution << pack_end;
          } while (pendingr || pendingp);

        }
```

```
                _netstream << barrier;
        }

}
```

MainSeq.cc

```
#include "newGA.hh"

int main (int argc, char** argv)
{
        using skeleton newGA;

        system("clear");

        if(argc < 4)
                show_message(1);

        ifstream f1(argv[1]);
        if (!f1) show_message(11);

        ifstream f2(argv[2]);
        if (!f2) show_message(12);

        Problem pbm;
        f2 >> pbm;

        Operator_Pool pool(pbm);
        SetUpParams cfg(pool);
        f1 >> cfg;
        Solver_Seq solver(pbm,cfg);
        solver.run();
        if (solver.pid()==0)
        {
                solver.show_state();
                cout << "Solution: " << solver.global_best_solution()
                    << " Fitness: " << solver.global_best_solution().fitness() << endl;
                cout << "\n\n :( ---------------------- THE END ---------------- :) "  << endl;

                ofstream fexit(argv[3]);
                if(!fexit) show_message(13);
                fexit << solver.userstatistics();

        }
```

```
        return(0);
}
```