Práctica 2.

Optimización Local de Soluciones

Metaheurísticas – Grado de Ingeniería Informática

Universidad de Córdoba

2018 / 2019

El objetivo de esta práctica es el de iniciar al alumnado en la programación y análisis de metaheurísticas basadas en trayectorias y particularmente, en el caso de métodos de búsqueda local. Para ello, se presentará un guión de actividades a realizar para generar y analizar diversos métodos de búsqueda local en el problema de la mochila múltiple cuadrática.

Se aporta un código esqueleto, de las mencionadas búsquedas locales, que el alumnado deberá completar y depurar, y una serie de tareas a realizar.

1. Código esqueleto

Los ficheros proporcionados en la página de la asignatura definen parcialmente las siguientes clases, además de las clases de la práctica anterior con algunas modificaciones:

- `MQKPEvaluator`: Evaluador utilizado en la práctica anterior. En la nueva versión, se añaden las siguientes funciones y variables miembro:
  - `_numEvaluations`: Variable donde se contabiliza el número de soluciones que se evalúan a través de `computeFitness` o `computeDeltaFitness`.
  - `computeDeltaFitness`: Función que calcula la diferencia en fitness si a la solución que se le pasa como argumento se le aplicase una operación de una asignación de un objeto a una mochila.
  - `resetNumEvaluations`: Función que resetea la variable interna que contabiliza el número de evaluaciones.
  - `getNumEvaluations`: Función que devuelve el número de veces que se ha evaluado alguna solución.
- `MQKPInstance`: Clase instancia utilizada en la práctica anterior. En la nueva versión, se añaden los siguientes métodos:
  - `randomPermutation`: Función que genera una permutación de los enteros de 0 a (*size* - 1). No tiene mucha relación con las instancias del problema, pero provee código que será interesante en diferentes puntos de las diferentes metaheurísticas.
  - `getDeltaSumProfits`: Función que calcula la diferencia en la suma de los

beneficios si a la solución se le aplicase la asignación del objeto *indexObject* a la mochila *indexKnapsack*.
    - `getDeltaMaxCapacityViolation`: Función que calcula la diferencia en la máxima violación de alguna de las capacidades de las mochilas si a la solución se le aplicase la asignación del objeto indexObject a la mochila indexKnapsack.

- `MQKPSolution`: Clase solución utilizada en la práctica anterior. En la nueva versión, se añaden los siguientes métodos:
    - `getFitness`: función que devuelve el fitness de la solución.
    - `setFitness`: función que asigna el fitness de la solución.
    - `copy`: función que copia la información de otra solución.

- `MQKPChangeOperation`: Es una clase abstracta para representar cualquier operación de modificación sobre una solución. Declara la siguiente función:
    - `apply`: Función que aplica el cambio que define el propio objeto sobre la solución que recibe como argumento.

- `MQKPObjectAssignmentOperation`: Es subclase de la clase anterior y codifica una operación de asignación de un objeto a una mochila, pudiendo ser ésta la mochila 0, es decir, sacarlo de la mochila en la que se encuentre. Define las siguientes variables miembro y métodos:
    - `_indexObj`: Entero que indica el objeto que se va a cambiar de mochila.
    - `_indexKnapsack`: Entero que indica la mochila donde se insertará el objeto.
    - `_deltaFitness`: Double que indica la diferencia en el fitness de la solución una vez que se aplica dicho cambio, siempre que la solución no haya cambiado cuando se creó esta operación.
    - `apply`: Aporta la definición correspondiente a la función heredada de la superclase.
    - `setValues`: Función que asigna los valores la operación.

- `MQKPNeighExplorer`: clase abstracta que define las operaciones de cualquier operador que explora la vecindad de una solución dada. Declara la siguiente función:
    - `findOperation`: Función que busca una operación que aplicada a la solución devuelva otra solución vecina. Se utilizará para buscar una solución vecina que la mejore, o la mejor de las soluciones vecinas. Devuelve verdadero si ha encontrado una operación válida, que mejora la solución actual. Falso en otro caso.

- `MQKPSimpleFirstImprovementNO`: subclase de la anterior, que se encarga de explorar el vecindario de una solución dada, devolviendo la primera operación de asignación de un objeto a una mochila que encuentre que mejore la calidad de la solución recibida. En caso de que no exista ninguna operación que mejore la calidad de la solución recibida, entonces devolverá `false`. Define la siguiente función heredada.
    - `findOperation`: Definición de la función heredada. Importante, el objeto *operation*

que recibe debe ser un objeto de la clase `MQKPObjectAssignmentOperation`.

- `MQKPSimpleBestImprovementNO`: subclase de `MQKPNeighExplorer`, que se encarga de explorar el vecindario de una solución dada, devolviendo `true` y la operación de asignación de un objeto que mejora en mayor medida la calidad de la solución recibida. En caso de que no exista ninguna operación que mejore la calidad de la solución recibida, entonces devolverá `false` y la operación que produzca el menor detrimento de ésta. Define la siguiente función heredada.
  - `findOperation`: Definición de la función heredada. Importante, el objeto *operation* que recibe debe ser un objeto de la clase `MQKPObjectAssignmentOperation`.
- `MQKPLocalSearch`: clase que recibiendo una solución al problema, la optimiza localmente e iterativamente hasta alcanzar un óptimo local. Define las siguientes variables miembro y métodos:
  - `_results`: vector de doubles donde almacena la calidad de la última solución aceptada.
  - `optimise`: Función que optimiza una solución aplicado repetidamente la exploración de su vecindario hasta alcanzar un óptimo local.
  - `getResults`: Función que devuelve el vector con los resultados de las soluciones aceptadas, en cada paso, por la búsqueda local.

2. Tareas a realizar

Se debe (todas las acciones sobre código tienen el comentario *TODO* en código, que indica que hay tareas por hacer):

1. Completa el código de `MQKPSolution::copy`.
2. Completa la definición de `MQKPObjectAssignmentOperation`.
3. Completa el código de `MQKPObjectAssignmentOperation::apply` y el de `MQKPObjectAssignmentOperation::setvalues`.
4. Completa la función `MQKPInstance::randomPermutation` para que genere una permutación de los enteros de 0 a (*size*-1) en el vector *perm*. Esta función no contiene código propio de las instancias de los problemas, pero será útil en diversos puntos donde las metaheurísticas deban tomar decisiones. Para ello, rellena *perm* con la permutación identidad, y después, recórrelo intercambiando cada elemento con otro escogido de forma aleatoria.
5. Completa la función `MQKPInstance::getDeltaSumProfits` para que devuelva la diferencia en la suma de beneficios si el objeto `indexObject` se asignase a la mochila `indexKnapsack`. Para ello, simplemente se debe:
   a. Si el objeto estaba en una mochila distinto de 0, entonces restar, a 0, el beneficio del objeto y el conjunto de dicho objeto con cualquier otro que esté en la misma mochila

origen; y

   b. Si la nueva mochila es superior a 0, sumar el beneficio del objeto y el conjunto de dicho objeto y cualquier otro que esté en la mochila de destino.

6. Completa la función `MQKPInstance::getDeltaMaxCapacityViolation` para que devuelva la diferencia en la máxima violación en la solución actual, si el objeto `indexObject` se asignase a la mochila `indexKnapsack`. Para ello, lo más fácil es cambiar la asignación del objeto en `solution`, invocar a `MQKPInstance::getMaxCapacityViolation` y almacenar el resultado en una variable auxiliar, deshacer el cambio anterior, volver a invocar al método `MQKPInstance:getMaxCapacityViolation`, almacenar el resultado en una segunda variable auxiliar y devolver la diferencia. Aunque esta operación se puede optimizar bastante, no lo vamos a hacer.

7. Completa la función `MQKPEvaluator::computeDeltaFitness` para que devuelva la diferencia de los fitness de la solución actual y si el objeto `indexObj` estuviera en `indexKnapsack`. Para ello:
   a. Almacenar la máxima violación actual en `actualMaxViolation`
   b. Almacenar la diferencia en la máxima violación en `deltaMaxCapacityViolation`
   c. Sumar ambas cantidades y almacenarlas en `newMaxViolation`
   d. Almacenar la suma de beneficios actual en `actualSumProfits`
   e. Almacenar la diferencia en la suma de beneficios en `deltaSumProfits`
   f. Sumar ambas cantidades y almacenarlas en `newSumProfits`
   g. Si `actualMaxViolation` y `newMaxViolation` son positivas, devolver el negativo de `deltaMaxCapacityViolation`
   h. Si ambos valores son iguales a 0, devolver `deltaSumProfits`
   i. Si sólo `actualMaxViolation` es positivo, devolver la suma de `newSumProfits` más el negativo de `deltaMaxCapacityViolation`
   j. Si sólo `newMaxViolation` es positivo, devolver la suma del negativo de `actualSumProfits` más el negativo de `newMaxViolation`.

8. Completa la función `MQKPSimpleFirstImprovementNO::findOperation`. Se debe generar una permutación de los índices de los objetos. Después, recorrer esa permutación probando poner el objeto en cualquier mochila y evaluando el `deltaFitness`. Tan pronto como se obtenga un `deltaFitness` positivo, se debe inicializar la operación y devolver `true`. Si no se encontrase un `deltaFitness` positivo en todas las posibles asignaciones, entonces se devuelve `false`.

9. Completar la función `MQKPSimpleBestImprovementNO::findOperation`. Se debe generar una permutación de los índices de los objetos. Después, recorrer esa permutación probando poner el objeto en cualquier mochila y evaluando el `deltaFitness`. Ir almacenando el cambio que produce el mayor `deltaFitness` para devolverlo una vez se hayan probado todos los posibles cambios. Si el mejor `deltaFitness` resulta ser negativo, entonces devolver `false`, en otro caso, devolver

`true`.

10. Completa la función `MQKPLocalSearch::optimise`. Debe explorar el vecindario de la solución provista con el explorador indicado y, mientras se consiga mejorar la solución, aplicar la operación encontrada y repetir.
11. Implementar un programa que, recibiendo una lista de pares fichero instancia del problema y número de mochilas, realice la siguiente experimentación en cada uno de ellos (el programa ya está parcialmente implementado en main.cpp; para completarlo, revisa los comentarios `TODO`). Para compilarlo, es necesario utilizar la opción `-std=c++0x`:
    a. Realice optimizaciones con los dos métodos de exploración de vecindarios a partir de soluciones aleatorias durante 5 segundos (máximo 100.000 evaluaciones aproximadamente y 5 optimizaciones locales). Si todo es correcto, toda la experimentación lleva menos de 80 segundos.
    b. Devuelva en pantalla valores en cuatro columnas para cada experimento, la primera con la calidad de la última solución generada por la búsqueda local con el explorador de primera mejora y la segunda con la mejor calidad generada hasta el momento por dicha búsqueda local, la tercera con la calidad de la última solución generada por la búsqueda local con el explorador de mejor mejora y la cuarta con la mejor generada hasta el momento por dicha búsqueda local.
12. Ejecuta el programa en las instancias provistas con 3 y 5 mochilas, guardando los resultados.
13. Dibuja dicha información en gráficas de convergencia para cada instancia del problema y número de mochilas.
14. Crea un documento **PDF** describiendo la experimentación realizada, el código incluido en el orden indicado en esta sección (puntos 1-11; para main.cpp sólo lo añadido, para el resto, las funciones correspondientes), y analizando los resultados obtenidos.
15. Desarrolla los primeros pasos para la optimización local de soluciones del problema seleccionado, diferente del MQKP.
16. Incluye una descripción del paso anterior en el documento a entregar (A entregar en la práctica 3).
17. Recuerda incluir la descripción de la representación y evaluación de soluciones (práctica 1) del problema seleccionado por el grupo en esta memoria.
18. Sube el documento a moodle.
19. En la evaluación del las prácticas de otros compañeros, deberéis proveer una pequeña retroalimentación de los errores que encontréis. Para dicha calificación, podéis guiaros de la siguiente rúbrica.

	Elemento a considerar	Posibilidades			
		Peor ----------> Mejor			
Penalizaciones	¿Es un documento PDF?	*No - se pone un 0*		*Sí - se corrige*	
	¿Tiene faltas de ortografía o gramaticales?	Con faltas de ortografía - *La nota máxima será 6*		Con alguna falta - *Se penaliza un poco*	Sin faltas y bien estructurado y escrito
	¿El documento contiene lo que se pide en el apartado 14 y en el orden correcto?	No es completo - *La nota máxima que se puede obtener sería un 8*		Es completo y desordenado - *se quitan 2 puntos*	Completo y ordenado
A evaluar	¿El código del apartado 1 es correcto y legible?	Poco legible e incorrecto	Legible pero incorrecto	Correcto pero poco legible	Correcto y legible
	¿Es correcto y legible el código del apartado 2?	Poco legible e incorrecto	Legible pero incorrecto	Correcto pero poco legible	Correcto y legible
	¿Es correcto y legible el código del apartado 3?	Poco legible e incorrecto	Legible pero incorrecto	Correcto pero poco legible	Correcto y legible
	¿Es correcto y legible el código del apartado 4?	Poco legible e incorrecto	Legible pero incorrecto	Correcto pero poco legible	Correcto y legible
	¿Es correcto y legible el código del apartado 5?	Poco legible e incorrecto	Legible pero incorrecto	Correcto pero poco legible	Correcto y legible
	¿Es correcto y legible el código del apartado 6?	Poco legible e incorrecto	Legible pero incorrecto	Correcto pero poco legible	Correcto y legible
	¿Es correcto y legible el código del apartado 7?	Poco legible e incorrecto	Legible pero incorrecto	Correcto pero poco legible	Correcto y legible
	¿Es correcto y legible el código del apartado 8?	Poco legible e incorrecto	Legible pero incorrecto	Correcto pero poco legible	Correcto y legible
	¿Es correcto y legible el código del apartado 9?	Poco legible e incorrecto	Legible pero incorrecto	Correcto pero poco legible	Correcto y legible
	¿Es correcto y legible el código del apartado 10?	Poco legible e incorrecto	Legible pero incorrecto	Correcto pero poco legible	Correcto y legible
	¿Es correcto y legible el código del apartado 11?	Poco legible e incorrecto	Legible pero incorrecto	Correcto pero poco legible	Correcto y legible
	¿Son correctos los resultados presentados y el análisis de éstos?	Son incorrectos	Hay algo raro	Son correctos	Son correctos e interesantes
	¿Son correctos los pasos realizados en el apartado 12 de la Práctica 1?	Yo diría que no	Demasiado genérico y no entiendo si son correctos	Parece que sí, pero deja alguna duda	Está muy claro y concreto y yo diría que sí es correcto

	¿Son correctos los pasos realizados para el apartado 16? (A entregar en la práctica 3)	Yo diría que no	Demasiado genérico y no entiendo si son correctos	Parece que sí, pero deja alguna duda	Está muy claro y concreto y yo diría que sí es correcto
General	Supón que eres el jefe de una empresa para la cual trabajan quienes han realizado el informe que estás evaluando	El informe es tan malo que los despedirías y buscarías nuevos empleados	El informe es lo suficientemente malo como para bajarles el sueldo	El informe es correcto. Les pedí X y han hecho X	El informe es tan bueno que considero que es bueno para mi empresa que estén contentos en su trabajo, y les voy a subir el sueldo.