Búsqueda Tabú

Introducción

El algoritmo de búsqueda tabú va generando posibles soluciones a partir de una solución inicial, va buscando optimizar dicha solución observando los posibles cambios y guardando los resultados obtenidos en una lista “tabú”.

Sobre el nodo elegido se sacan todos los posibles movimientos, posibles colores de vecinos adyacentes, y se eliminan los movimientos tabú.

Representación

La solución será un vector de tantas posiciones como nodos tenga el grafo. Cada color será un número, el número máximo de colores posibles será el número de nodos del grafo, pues en el peor de los casos, todos están conectados con todos y harían falta tal número de colores.

Para optimizar el tiempo de cómputo, este vector vendrá emparejado con el número objetivo, que primeramente dará el número de conflictos que tiene dicho vector. Más tarde nos indicará (en negativo) el número de colores que no se han utilizado.

Pseudocódigo

A continuación describiremos los principales métodos utilizados para la realización. El algoritmo principal en el que le indicamos el número de iteraciones que queremos que realice.

`busquedaTabu (iter){`	

`solucionActual = (objetivo (inicioAleatorio (length *grafoProblema*)))`
`Iteraciones = 1`
`Salida = 0`
	Inicializamos el problema

`while (salida ==0 && iter != iteraciones){`
 	
`(let* ((vecinos '())`
`(listaTabu '())`
`(aspirantes '())`
`(conjuntoAspirantes '())`
`(mejorAspirante '())`
`(viejaSolucion '())`
`)`
	Inicializa la iteración a partir de la solución actual

`vecinos = (vecindario solucionActual)`
`aspirantes = (aspirantesFuncion solucionActual listaTabu)`
`conjuntoAspirantes = (dameConjunto aspirantes listaTabu vecinos)`
`mejorAspirante =(mejor conjuntoAspirantes)`
`viejaSolucion = solucionActual`
	Identificamos vecinos, aspirantes y candidatos totales… y elegimos el mejor.

`If(length(vecinos) == 0{`
`If(mejorAspirante[2]< solucionActual[2])`
`solucionActual = mejorAspirante`
`}else{`
`Salida = 1`
`}`
 	
`(push viejaSolucion listaTabu)`
 	
`Iteraciones++`
`)`
	Si el mejor aspirante mejora, actualizamos.

Salas Cantalejo, Alberto
Bellés Camacho, Víctor Juan

) solucionActual)	Devolvemos la solución

Vecindario devuelve los vecinos cuyo valor de la función objetivo es mejor que el valor de la función objetivo de la solución actual.

vecindario (actual){	
movimientosPosibles '() vecinos '()	

for i from 0 to (length(*grafoProblema*)-1) do{
(push i movimientosPosibles)
}
movimientosPosibles = (reverse movimientosPosibles)
	Genera todos los movimientos posibles

for i from 0 to (length (actual[1])-1) do{
 	
movimientos = movimientosPosibles.delete(actual[1][i])
for j from 0 to (length(movimientos)-1) do{
nuevaSolucion = (movimiento actual[1] i
movimientos[j])
 	
if (nuevaSolucion[2]) < actual[2])
(push nuevaSolucion vecinos)
}
}
vecinos
}
}
	Va descartando movimientos hasta encontrar alguno que mejore.
	En caso de encontrarlo lo lista para su posterior comprobación.

El resto de métodos están descritos en el código.

Ejemplos

En el código aportamos el problema de coloración de mapas con el mapa de Andalucía. Inicializando el vector a 1's. Durante la ejecución por cada iteración veremos la evolución de este vector y como va decrementando el número de conflictos.

También tenemos un grafo cíclico de 6 nodos y otro con el típico problema de K5.

Otros problemas

Para poder reutilizar el mismo algoritmo para otros problemas que no sean de coloración habría que adaptar el objetivo y el generador de movimientos que nos cualifica cuan buena es cada solución y las elige. El algoritmo principal no se vería afectado porque utiliza el segundo valor de la solución. También debería adaptarse la representación de los individuos.

Resultados y comparación

Problema	Tiempo (milis)	Fin-iteraciones	Iteraciones
C6	670,8043	15	3
	842,4054		3
	3369,6215		4
K5	37,8003	15	2
	31,2002		2
	46,8003		3
Andalucía	2245,1284	15	4
	1825,2116		2
	6396,041		5

El problema dado se apoya en la aleatoriedad del principio para ir perfeccionando los conflictos que haya, por lo que intentará que los nodos válidos sean guardados, mantenido mediante reglas tabú y así agilizar el procesamiento.

Ninguno de los ejemplos ha tenido que llegar al límite de 15 iteraciones que pusimos. Y dependiendo de la aleatoriedad del principio es capaz de solventar el problema en menos iteraciones.

Cabe destacar cierta disparidad, pues en primeras ejecuciones de un mismo algoritmo corre muchísimo mas rápido que en sucesivas (defecto que viene dado por el entorno). Además, de como lo escrito sobre el otro algoritmo, cuanto el tiempo es mayor, la probabilidad de ser interrumpido también lo es y crece el tiempo por intercambio del procesador.

Dificultades y errores

- La dificultad de optimizar el grafo una vez obtenido encontrada una solución sin conflictos.
- El bajo nivel de desarrollo de funciones del entorno. Lisp no esta tan fuertemente apoyado por la comunidad y solo tenemos funciones muy básicas.
- El que añadir a la lista Tabú, además de los casos de solución, y el como añadirlos

Bibliografía y referencias

- Heuristic methods for graph coloring problems, A. Lim, Y. Zhu, Q. Lou, B. Rodrigues, apartado 3.4 - http://www.cs.us.es/cursos/ia1/trabajos/HeuristicMethods.pdf
- Conceptos, algoritmos y aplicación al problema de las N-reinas, capitulo 3 busqueda tabú, Alicia Cirila Riojas Cañari - http://sisbib.unmsm.edu.pe/bibvirtualdata/monografias/basic/riojas_ca/cap3.pdf
- Artificial intelligence: a modern approach, Stuart Russel y Peter Norvig, páginas 154 y 222.

Salas Cantalejo, Alberto
Bellés Camacho, Víctor Juan

- Implementación de Búsqueda Tabú en la Solución del Problema de Asignación Cuadrática, Dagoberto Ramón Quevedo Orozco y Roger Z. Ríos Mercado - http://yalma.fime.uanl.mx/~roger/work/Papers/article/article-inge-2010.pdf
- Universidad de Granada, algorítmica tema 4 - http://sci2s.ugr.es/docencia/algoritmica/Tema04-BusquedaTabu-10-11.pdf