Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI5652 - Diseño de Algoritmos II
Trimestre Abril - Julio 2012

INFORME I
Resolviendo el Symmetric Traveling Salesman Problem

Integrante:
Vicente Santacoloma 08-11044

Sartenejas, 18 de mayo de 2012

INTRODUCCIÓN

1. Motivación del proyecto

El presente informe pretende realizar una descripción y análisis de la heurística **iterate-hill-climber** para resolver el problema del **Symmetric Traveling Salesman Problem (STSP).**

Para realizar el análisis de la heurística se empleará diversos resultados experimentales para las corridas de 4 instancias bien conocidas del STSP que son: eil51, kroA100, d198 y rat783. Estos resultados permitirán discutir aspectos como: calidad de las soluciones obtenidas, esfuerzo computacional, robustez y fiabilidad.

Además se prevé discutir otros aspectos acerca de la heurística en cuestión, que permitirá tener un criterio acerca de su utilización.

2. Breve descripción del problema

El **Traveling Salesman Problem (TSP)** consiste en encontrar una ruta, que partiendo desde una posición inicial, se visite a un conjunto de ciudades, y se regrese a dicha posición de inicio, de tal manera que la distancia total recorrida sea mínima y que cada ciudad sea visitada exactamente una vez.

Dependiendo de la naturaleza de la matríz de costos (equivalente a la naturaleza de un grafo G), TSP se divide en dos clases. Si G es simétrico (es decir, G es no dirigido), el TSP es llamado **Symmetric Traveling Salesman Problem (STSP)**, el cual es el caso de estudio para este proyecto. Si G no es simétrico (equivalente a que el grafo G se dirigido), el TSP es llamado **Asymmetric Traveling Salesman Problem (ATSP).**

3. Descripción del contenido del informe

El presente informe consta de:

- **Portada.**
- **Introducción.** Donde se describe brevemente el problema.
- **Algoritmo para el STSP.** Se presenta una descripción de la heurística **iterated-hill-climber**, donde se detalla su funcionamiento, estructuras empleadas y otra información adicional.
- **Detalles de software y hardware empleados.**

- I**nstrucciones de operación.** Descripción detallada de como compilar y correr el software, así como el estado actual de la misma.
- **Resultados experimentales y discusión.** Estudio experimental para caracterizar el rendimiento de la heurística propuesta en base a tres aspectos.
- **Conclusiones y recomendaciones.**
- **Referencias bibliográficas.**

ALGORITMO PARA EL STSP

1. Estructuras de datos utilizados

El **iterated-hill-climber** utiliza las estructuras:

- **distMat:** Matriz que contiene los costos entre cada para de ciudades. Representa el grafo G de la instancia con la cual se ejecutó el algoritmo.
- **nnMat:** Tabla que contiene la lista de los vecinos más cercanos para todas las ciudades.

2. Descripción de los principales algoritmos para resolver el problema

Para la elaboración de la heurística **iterated-hill-climber** se empleó el software **simple LS and ILS algorithms for the TSP** desarrollado por **Thomas Stuetzle [6]**.

A continuación se presenta la descripción de la heurística **iterated-hill-climber** y el algoritmo de **three_opt_first**. Seguidamente se presentará el código en lenguaje C, y luego las referencia a las funciones y variables utilizadas del software mencionado anteriormente **[6].**

iterated-hill-climber

La heurística **iterated-hill-climber**, al igual los métodos de búsqueda local, se vale de la técnica iterativa. Durante cada iteración se elige un tour aleatorio del espacio de búsqueda o de estado. Luego a partir de este tour se tratará de encontrar un tour que sea mínimo local. Esto es, de los vecinos de ese tour se buscará el de menor distancia y se comparará con el actual. En caso de que la distancia del menor tour vecino sea menor que el tour actual, se remplazará este último por el nuevo tour. En caso contrario, el tour actual pasará a ser un mínimo local.

Este tour que es mínimo local se comparará con el mejor tour encontrado por la heurística hasta el momento, y si es mejor será el nuevo mejor tour encontrado.

El procedimiento de generar un tour aleatorio y buscar su mínimo local para luego compararlo con el mejor tour encontrado por la heurística hasta el momento, se repetirá un número de iteraciones indicado por el que utilice dicha heurística.

El procedimiento de generar vecinos se basa en el 3-opt.

three_opt_first [N]

Busca el mínimo local de un tour dado mediante la generación de vecinos por 3-opt. Este algoritmo realiza varias optimizaciones las cuales las podemos encontrar en el **Stochastic Local Search Foundations and Applications [3].**

Codigo de los algoritmos empleados en lenguaje C.

main.c

```c
#include <stdio.h>
#include <math.h>
#include "TSP-TEST.V0.9/instance.h"
#include "TSP-TEST.V0.9/utilities.h"
#include "TSP-TEST.V0.9/timer.h"
#include "TSP-TEST.V0.9/ls.h"
#include "iterated_hill_climber.h"

int main (int argc, char **argv) {

  int * best;
  int max = 0;
  read_instance(argv[1]);
  distMat = compute_distances();
  nn_ls = MIN (ncities - 1, 40);
  nnMat = compute_NNLists();
  max = ncities*100;
  start_timers();
  best = iterated_hill_climber(max);
  float time = fabs(elapsed_time( VIRTUAL));
  printf("\n\n");
  printf("RESULTS:");
  printf("\n\n");
  printf("Best Tour Found:\n");
  printTour(best);
  printf("Length: %d\n",compute_length(best));
  printf("Total Iteration Number: %d\n",max);
  printf("Total Time: %f seconds\n",time);
  printf("Iteration Best Found: %d\n",iteration_best_found);
  printf("Time Best Found: %f seconds\n", time_best_found);

  return 0;
}
```

iterated_hill_climbing.h

```c
/**
 * Copy one vector to another
 * @param vector1 vector to be copied
 * @param vector2 vector in which to copy
 * @return void
 */
void copy_vector (int * vector1, int * vector2);

/**
 * Find the best tour for the STSP problem by-iterated-hill-climbing heuristic
 * @param max number of iterations
 * @return best tour found by the heuristic
 */
int * iterated_hill_climber(int max);
```

iterated_hill_climbing.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <limits.h>

#include "TSP-TEST.V0.9/instance.h"
#include "TSP-TEST.V0.9/utilities.h"
#include "TSP-TEST.V0.9/timer.h"
#include "TSP-TEST.V0.9/ls.h"
#include "iterated_hill_climber.h"

void copy_vector (int * vector1, int * vector2) {
  int i;
  for(i = 0; i < ncities; i++)
    vector1[i] = vector2[i];
}

int * iterated_hill_climber(int max) {

/* BEGIN */

  int t;
  int * best;
  int * tour;
  int * random;

  seed = (long int) time(NULL);
  tour = malloc((ncities + 1 ) * sizeof(int));

/* Initialize best */
  best = malloc((ncities + 1 ) * sizeof(int));
  random = generate_random_vector();
  copy_vector(best,random);
  free(random);
  best[ncities] = best[0];

  for(t = 0; t < max; t++) {

    dlb = calloc(ncities, sizeof(int));
    random = generate_random_vector();
    copy_vector(tour,random);
    tour[ncities] = tour[0];

    three_opt_first(tour);

    if(compute_length(tour) < compute_length(best)) {
      copy_vector(best,tour);
      best[ncities] = best[0];
      iteration_best_found = t;
      time_best_found = fabs(elapsed_time( VIRTUAL ));
    }
    free(random);
    free(dlb);

  }
  free(tour);
/* END */
  return best;

}
```

Partes del software **simple LS and ILS algorithms for the TSP [6]** utilizado en el código mostrado anteriormente:

Variables:

- distMat
- nn_ls
- nnMat
- ncities
- iteration_best_found
- time_best_found
- seed
- dlb

Funciones:

- read_instance
- compute_distances
- MIN
- compute_NNLists
- start_timers
- fabs
- elapsed_time
- compute_length
- generate_random_vector
- three_opt_first

3. Parámetros que utiliza el algoritmo

El algoritmo **iterated-hill-climber** requiere además de las estructuras de datos especificadas anteriormente, los parámetros:

- **ncities:** Número de ciudades de la instancia a evaluar.
- **max:** Número de iteraciones a realizar. Este parámetro, fue considerado en función del número de ciudades de la instancia multiplicado por 100. La razón de esto, es que mientras mayor sea la cantidad de ciudades más iteraciones se va a necesitar para obtener un resultado aceptable.

4. Información adicional

Se decidió utilizar únicamente la función **three_opt_first** en lugar de las otras, o en combinación con las otras, tal como **two_opt_best (**también contenida en el software **[6])**, puesto que el tiempo computacional que requiere es mucho mayor, y no ofrecen ningún beneficio importante. Es más aprovechable utilizar ese tiempo computacional adicional, en aumentar el número de iteraciones del algoritmo, ya que así se podrá explorar más elementos (tours) del espacio de estado.

El diseño de este algoritmo esta basado en el libro **How to Solve It: Modern Heuristics [4]**, en combinación con el software **simple LS and ILS algorithms for the TSP [6]**

DETALLES DE SOFTWARE Y HARDWARE EMPLEADO

Hardware Empleado

Procesador: Intel(R) Core(TM) 2 Duo CPU P8600 @ 2.40GHz
Memoria RAM: 4GiB

Software Empleado

Sistema Operativo: Debian GNU/Linux Wheezy.
Kernel: Linux version 3.1.0-1-amd64
Lenguaje de Programación: C ANSI
Compilador: gcc 4.6.3
Software: **simple LS and ILS algorithms for the TSP [6]**

Estado Actual

100% Funcional

INSTRUCCIONES DE OPERACION

Para instalar el software primero debemos descomprimir el archivo en formato .tar.gz mediante:

$ tar -xvf STSP.tar.gz

Para compilar el código en plataforma linux, deberemos posicionarnos en la carpeta STSP y luego ejecutar el comando:

$ make

Luego se producirá el ejecutable ***stsp***. Para ejecutarlo escribimos:

$ ./stsp <tsplibfile>

Donde ***<tsplibfile>*** es una instancia del STSP las cuales se encuentra en el subdirectorio ***ALL_tsp***

Por ejemplo:

$ ./stsp ALL_tsp/eil51.tsp

RESULTADOS EXPERIMENTALES Y DISCUSIÓN

TABLA DE RESULTADOS 1

A	eil51.tsp	kroA100.tsp	d198.tsp	rat783.tsp
B	51	100	198	783
C	406	21282	15780.9	8966.2
D	0%	0%	0,005703422%	1,819214%
E	0	0	0,875595	10,36876
F	406	21282	15780	8948
G	10	10	4	0

Reseña:

- **A.** Nombre de la instancia.
- **B.** Número de ciudades.
- **C.** Distancia promedio de 10 corridas de la heurística.
- **D.** Porcentaje de desviación de la distancia promedio de la heurística, con respecto a la solución óptima.
- **E.** Desviación estándar del valor promedio de la heurística.
- **F.** Distancia de la mejor solución obtenida en las 10 corridas de la heurística.
- **G.** Número de ocurrencias de la mejor solución en las 10 corridas de la heurística.

TABLA DE RESULTADOS 2

A	eil51.tsp	kroA100.tsp	d198.tsp	rat783.tsp
B	406	21282	15780	8806
C	406	21282	15780	8948
D	0%	0%	0%	1,614808%
E	5100	10000	19800	78300
F	22,3	47,1	8109,2	36852,3
G	7	0	242	13786
H	0,137254%	0,471%	40,95556%	52.93448%
I	0,459228	2,960584	14,73412	303,2057
J	0,0068	0,0144005	6,026776	142,8785
K	0	0	0,184011	52,40328
L	1,480745%	0,486407%	40,90354%	47,12263%

Reseña:

- **A.** Nombre de la instancia.
- **B.** Distancia de la solución óptima.
- **C.** Distancia de la mejor solución obtenida en las 10 corridas de la heurística.
- **D.** Porcentaje de desviación de la mejor solución de la heurística con respecto a la solución óptima.
- **E.** Número total de iteraciones.
- **F.** Número promedio de iteraciones para encontrar la mejor solución.
- **G.** Mejor número de iteraciones para encontrar la mejor solución.
- **H.** Porcentaje de iteraciones promedio para encontrar la mejor solución con respecto al número total de iteraciones.
- **I.** Tiempo promedio de las 10 corridas de la heurística, en segundos.
- **J.** Tiempo promedio de las 10 corridas de la heurística para encontrar la mejor solución, en segundos.
- **K.** Mejor tiempo para encontrar la mejor solución.
- **L.** Porcentaje de tiempo promedio para encontrar la mejor solución con respecto al tiempo total promedio.

ANÁLISIS DE RESULTADOS

Tomando en cuenta los tres aspectos:

1. Calidad de las soluciones que son obtenidas.

Tal como se puede ver en las tablas de resultados, para 3 de las 4 instancias se logró la distancia mínima óptima. Además en las dos primeras se logró dicha distancia óptima en cada una de las corridas.

Para la última instancia sin bien no se logró la distancia óptima, podemos decir que esta fue bastante aceptable.

Ahora, para evaluar la calidad de las soluciones, es decir, ver que tan buenas son en comparación con las óptimas, deberemos considerar otros mecanismos puesto que para un problema no conocido, no podremos saber su solución óptima.

Uno de estos mecanismos es tomar en cuenta para un número máximo de iteraciones, a partir de ¿cuál esta logró obtener su mejor solución? Para la última instancia, se puede apreciar que en promedio a la iteración 36852 se logró la mejor solución, lo que representa aproximadamente el 53% de las iteraciones totales. Esto es, que en casi la mitad del número de iteraciones, la heurística no logró conseguir un mejor resultado. Por lo tanto podremos decir que la solución obtenida, puede ser buena, pudiendo en algunos casos llegar al óptimo tal como se aprecia en las demás instancias, en particular, en la primera y segunda, donde se necesitó menos del 1% para lograr la mejor solución.

Ahora bien, si por ejemplo nuestro algoritmo para un total de 1000 iteraciones encontró la mejor solución en la iteración 900, no podemos dar ninguna idea de la calidad de la solución, a menos que la cantidad de ciudades para la instancia del problema fuera muy pequeña.

Finalmente, cabe señalar que estos mecanismos para evaluar la calidad de una solución, solo deben ser usados para tener un indicio y nunca como un reflejo de la realidad, puesto que nuestra heurística no considera todo el espacio de soluciones; solo un pequeño subconjunto. Alternativamente en lugar de solo considerar el número de iteraciones, se podría hacer en base al tiempo total de ejecución, comparándolo con el tiempo en el que se encontró la mejor solución.

2. Esfuerzo computacional

Para la última instancia el tiempo computacional requerido para el total de iteraciones fue relativamente pequeño, tomando para cada caso, su número de ciudades. Además si se observa el tiempo con el cual se logró obtener la mejor solución, podemos decir que este es mucho menor, que el que se necesitó a lo largo de todas las iteraciones.

Cabe destacar, que a medida que aumenta la cantidad de ciudades, mayor es el tiempo que se va a necesitar, para aumentar la posibilidad de conseguir mejores soluciones. Sin embargo para un rango de instancia que tenga una cantidad de pequeña a mediana de ciudades, el tiempo requerido para obtener buenos resultados es aceptable. Gracias a la simplicidad del algoritmo, este es bastante rápido en términos computacionales.

3. Robustez y fiabilidad

La heurística **iterated-hill-climber** permite obtener buenos resultados para problemas que no tengan una cantidad excesiva de ciudades, sin tener que modificar la cantidad de iteraciones a realizar. Tal como se mencionó antes, la cantidad de iteraciones, se asignó en función de la cantidad de ciudades multiplicadas por 100. Sin embargo si el problema es grande, muy probablemente necesitaremos más iteraciones, para quizás obtener mejor resultados. Claro está que no podemos simplemente colocarle una cantidad excesiva de iteraciones, porque nuestro algoritmo nunca terminaría.

Un aspecto importante que vale la pena señalar es que el obtener buenas soluciones está determinado por la escogencia aleatoria de un tour. En la tabla 2, se puede apreciar que en la iteración 0 para la segunda instancia, se logró obtener el mejor tour o uno de los tour vecino al mejor, que resultó ser o permitió conseguir el óptimo. Esto digamos, pudo ser un golpe de suerte. Para otras corridas podríamos no lograr nunca esto, y estar seleccionando siempre los peores tour. Lo que si podemos concluir es que a medida que crece el tamaño del problema, menor es la probabilidad de ir conseguir aleatoriamente mejores tours. Esto sería equivalente a ganarse la lotería. Para lo cual si queremos tener más chance de ganarla deberemos comprar la mayoría o todos los boletos, que se traduce en ejecutar la heurística por un tiempo muy largo, que en la práctica es inviable, pese a la ventaja en rapidez comparado con una búsqueda exhaustiva.

CONCLUSIONES Y RECOMENDACIONES

En base al análisis y discusión de resultados, realizado a lo largo del informe, podemos en primer lugar concluir que esta heurística representa una buena opción para resolver el problema STSP, que como sabemos, podemos determinar de manera sencilla cómo calcular su solución óptima en la teoría, pero no en la práctica.

Podemos decir que las ventajas de usar la heurística iterated-hill-climber para resolver el problema STSP, es que es sumamente sencillo de implementar; no requiere realizar ajustes para la configuración de los parámetros en instancias de problemas de tamaño pequeño o mediano; y además en dichas instancias permite obtener resultados relativamente buenos.

Entre las desventajas podemos decir, que este algoritmo en principio solo garantiza obtener soluciones que son óptimo local; no da ninguna información segura de que tanto se desvía el óptimo local del global (como se dijo anteriormente solo se puede tener una pequeña noción, que no necesariamente es reflejo de la realidad); el óptimo que se obtenga depende de la elección aleatoria del tour, y no en base a un criterio; y finalmente, no es posible proporcionar una cota superior del tiempo computacional, especialmente para problemas muy grandes.

Evaluando las ventajas y desventajas se puede señalar que esta heurística no es la mejor para resolver el problema STSP, pese a los buenos resultados obtenidos para instancias: eil51.tsp, kroA100.tsp, d198.tsp y rat783.tsp. Será más recomendable utilizar heurísticas, que utilicen algún criterio y no únicamente el azar, para generar un tour a evaluar.

REFERENCIAS BIBLIOGRÁFICAS

1. Barr, R., Golden, B., Kelly, J., Resende, M., and Stewart, W. Designing and reporting on computational experiments with heuristic methods. Journal of Heuristics 1, 1 (1995).
2. Gutin, G., and Punnen, A. The traveling salesman problem and its variations, vol. 12. Kluwer Academic Pub, 2002.
3. Hoos, H., and Stutzle, T. Stochastic local search: Foundations and applications. Morgan Kaufmann, 2005.
4. Michalewicz, Z., and Fogel, D. How to solve it: modern heuristics. Springer-Verlag New York Inc, 2000.
5. Reinelt, G. Tsplib. http://comopt.ifi.uniheidelberg.de/software/TSPLIB95/, 2012.
6. Thomas Stuetzle. TSP-TEST, Version 0.9. Available from http://www.sls-book.net, 2004.