Travelling Umpire Problem

Autor: Benjamín Aguilera
Ayudante: Pablo Estobar

Representación

Se usan dos clases para representar el problema y su solución.

Se usan vectores de vectores para representar las diferentes matrices presentes en el problema.

```
class TUP {

  int nUmpires;
  int nTeams;
  int nRounds;
  vector<vector<int>> distMatrix;
  vector<vector<int>> oppMatrix;
}
```

```
class Solution {
  TUP* problem;
  int distance;
  int d1;
  int d2;
  vector<vector<int>> visitsMatrix;
}
```

Restricciones

Debido a problemas de tiempo y complicaciones en el desarrollo, sólo se implementó la revisión de dos de las tres restricciones que conforman el problema, siendo estas:

- Un árbitro no debe visitar la misma sede más de una vez en $N - d_1$ rondas consecutivas.
- Un árbitro no debe dirigir el partido del mismo equipo más de una vez en $\lfloor N/2 \rfloor - d_2$ rondas consecutivas.

Se corrigió un error respecto de la entrega 2, la cuál provocaba que las infracciones a ambas restricciones no se contabilizaran correctamente.

A pesar de que no se revise la tercera restricción (cada juez visita la sede de cada equipo por lo menos una vez), se priorizan soluciones que cumplen esta restricción durante la generación de la solución inicial.

Solución Inicial

Se reemplazó el algoritmo aleatorio por uno de tipo greedy.

Este algoritmo representa el problema como un grafo bipartito, con los partidos por un lado y los árbitros por el otro, para cada ronda.

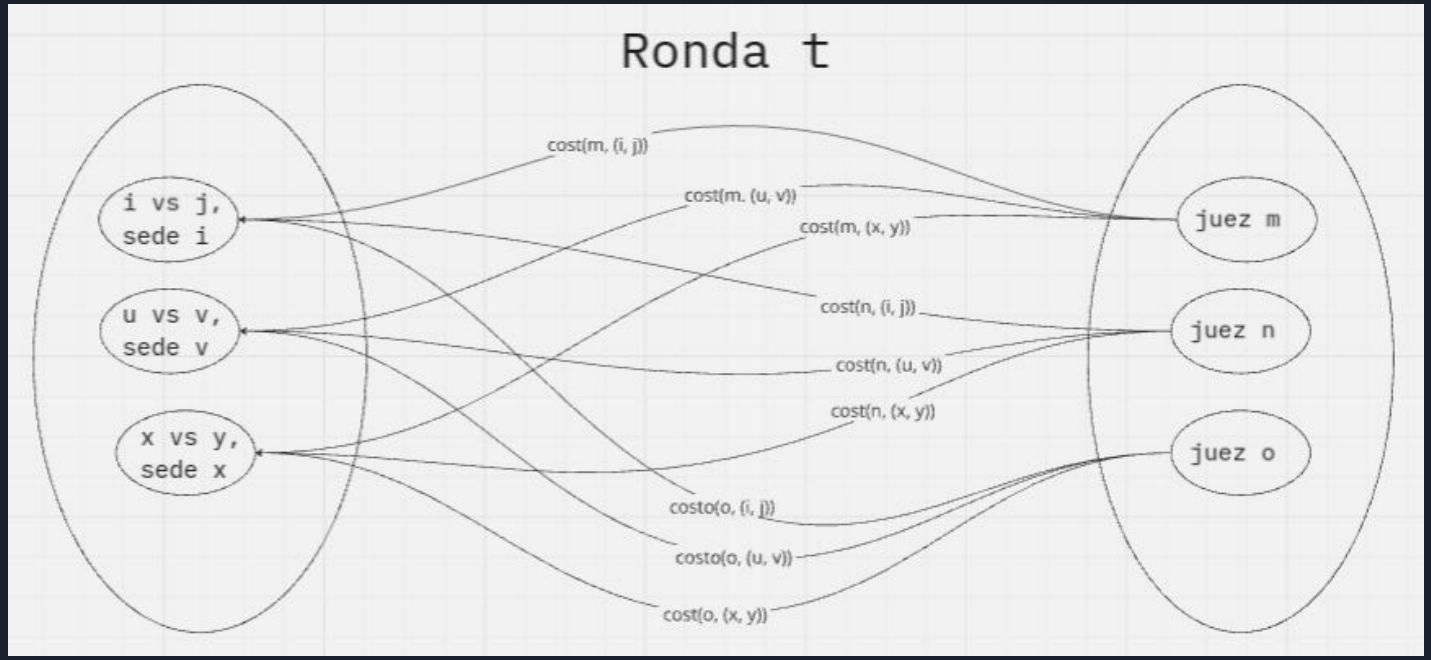
En cada ronda, para cada juez y cada partido, se calcula el costo parcial usando una función de costo.

Se van eligiendo los arcos de menor a mayor costo hasta que cada juez esté asignado a un partido.

El proceso se repite en cada ronda.

Esta función no asegura soluciones factibles.

Solución Inicial



Función de costo para Greedy

$$costo(u, (i, j)) = distancia(k, i) - incentivo(u, i)$$
$$+penalizacion \cdot infracciones(u, i, t)$$

Para computar el costo de cada arco entre jueces y partidos, se usa esta función, donde:

- u es el juez actual
- (i, j) es un partido entre i y j
- distancia(k, i) es la distancia entre la sede k actual y una sede i candidata
- incentivo (u, i) toma un valor positivo si el juez u nunca ha visitado la sede del equipo i
- penalización es un valor fijo muy grande que se agrega al cometer alguna infracción a alguna restricción
- infracción(u, i, t) es el número de infracciones que se cometen al asignar el juez u a la sede i en la ronda t

Este algoritmo está fuertemente inspirado en el algoritmo presentado por Michael Trick en su publicación del año 2012 [1]

Simulated Annealing

Simulated annealing es un algoritmo de optimización probabilístico usado para encontrar soluciones aproximadas a problemas complejos de optimización.

Este algoritmo comienza con una alta temperatura, que permite la exploración del espacio de búsqueda al permitir soluciones de peor calidad que la actual, lo cuál permite escapar de óptimos locales. Mientras la temperatura va disminuyendo, el algoritmo se vuelve más selectivo, intensificando la búsqueda en la solución actual y aceptando sólo soluciones que mejoren la actual.

Parámetros del algoritmo

El comportamiento de Simulated annealing puede ser ajustado mediante la modificación de los siguientes parámetros:

- alpha (α): Tasa de reducción de la temperatura.
- it: Número de iteraciones por cifra de temperatura.
- T_0 : Temperatura inicial
- T□ᵢ□: Temperatura mínima

Función de Evaluación

La función de evaluación se calcula de la siguiente manera:

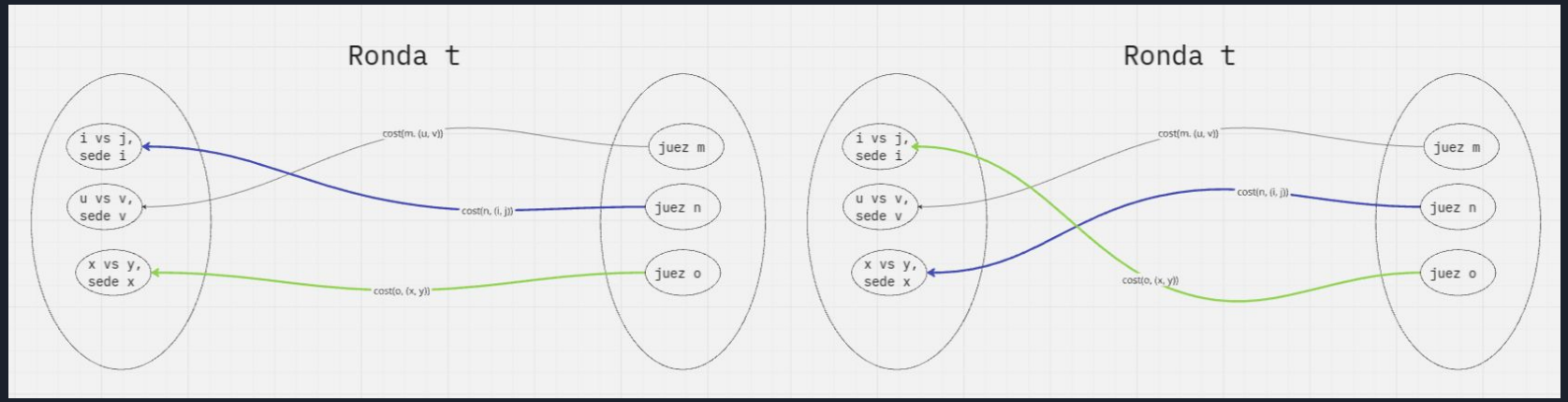
$$costo(u, (i, j)) = distancia(k, i)$$
$$+penalizacion \cdot infracciones(u, i, t)$$

donde:

- u es el juez actual
- (i, j) es un partido entre i y j
- distancia(k, i) es la distancia entre la sede k actual y una sede i candidata
- penalización es un valor fijo muy grande que se agrega al cometer alguna infracción a alguna restricción
- infracción(u, i, t) es el número de infracciones que se cometen al asignar el juez u a la sede i en la ronda t

Movimiento

Se eligió un movimiento de swap entre dos asignaciones en la misma ronda



Cambios con respecto a la última entrega

Se arregló un bug que provocaba que las infracciones a las restricciones no se contaran correctamente.

Se agregó una función solutionWriter, que escribe las soluciones en un archivo txt con el formato que se especificaba en la presentación del proyecto.

Experimentos

Variación de la cantidad de iteraciones entre 2500, 5000 y 10000 para cada instancia.

Cada instancia con cada número de iteraciones se ejecutará 5 veces para obtener datos más representativos.

Se eligen las instancias umps4, umps8, umps6 y umps10 para este experimento.

Los parámetros del problema serán los siguientes:

- Temp inicial: 2000
- Temp mínima: 500
- α: 0,95
- $d_1 = d_2 = 0$

Resultados

Instancia	N° de iteraciones	Mejor solución	Tiempo de ejecución (promedio) [s]	Infracciones de equipo	Infracciones de sede
ums4	2500	5176 (OPT!)	5	0	0
	5000	5176 (OPT!)	9	0	0
	10000	5176 (OPT!)	19.25	0	0

Resultados

Instancia	N° de iteraciones	Mejor solución	Tiempo de ejecución (promedio) [s]	Infracciones de equipo	Infracciones de sede
ums6	2500	14077 (OPT!)	5	0	0
	5000	14077 (OPT!)	11	0	0
	10000	14077 (OPT!)	22	0	0

Resultados

Instancia	N° de iteraciones	Mejor solución	Tiempo de ejecución (promedio) [s]	Infracciones de equipo	Infracciones de sede
ump8	2500	34856	7	1	1
	5000	34240	14	1	0
	10000	34187	28	1	0

Resultados

Instancia	N° de iteraciones	Mejor solución	Tiempo de ejecución (promedio) [s]	Infracciones de equipo	Infracciones de sede
ump10	2500	54245	9	0	0
	5000	53211	18	0	0
	10000	56588	39	0	0

Resultados

Se comparan los resultados obtenidos con las mejores soluciones publicadas en el sitio web TUP Benchmark[2].

Se observa que para las instancias umps4 y umps8, se logra llegar a la solución óptima, mientras que las instancias umps8 y umps10 no llegan a este resultado, pero, sin embargo, se acercan bastante a los resultados esperados.

Conclusiones

Los resultados obtenidos en este experimento son satisfactorios. Teniendo en cuenta que el algoritmo no asegura soluciones factibles, este encuentra este tipo de soluciones con éxito para instancias más simples y se acerca bastante para instancias más complejas.

Esto da a entender que la implementación requiere de unos pequeños ajustes para alcanzar el nivel de rendimiento presentado en solvers de similar naturaleza.

Referencias

- [1] Trick, M.A., Yildiz, H., & Yunes, T.H. (2012). Scheduling Major League Baseball Umpires and the Traveling Umpire Problem. Interfaces, 42, 232-244.
- [2] Túlio A. M. Toffolo, Tony Wauters, and Michael Trick. An automated benchmark website for the traveling umpire problem, June 2015, http://gent.cs.kuleuven.be/tup