Inteligencia Artificial

Informe Final: Progressive Party Problem

Victor Andres Roberto Gonzalez Rodriguez

18 de julio de 2014

Evaluación

Mejoras 1ra Entrega (10 %):	_____
Código Fuente (10 %):	_____
Representación (15 %):	_____
Descripción del algoritmo (20 %):	_____
Experimentos (10 %):	_____
Resultados (10 %):	_____
Conclusiones (20 %):	_____
Bibliografía (5 %):	_____
Nota Final (100):	_____

Resumen

El *Progressive Party Problem* (PPP), es un complejo problema de optimización combinatorial sujeto a restricciones propuesto en el año 1995. El objetivo del problema consiste en una fiesta de yates, donde los invitados deben ser capaces de recorrer todos los yates anfitriones, cambiandose de yate cada cierto tiempo y cumpliendo ciertas restricciones. Hasta el momento, se han utilizado tres métodos para tratar de resolver el problema: mediante Programación Lineal Entera o Mixta, mediante Programación con Restricciones, y mediante Búsqueda Local (esta última con distintas variaciones). En el presente documento se presenta el estado del arte de este problema, y se complementa con la implementación en C++ del problema, resolviéndolo mediante *Backtracking* con *Graph-based Backjumping* (GBJ), lo cual muestra que para este problema el GBJ es poco útil.

1. Introducción

En el presente documento se analizará en detalle el Problema de la Fiesta Progresiva, o *Progressive Party Problem* (PPP), su historia, el estado del arte, como han evolucionado los métodos resolutivos para este problema, y la completa definición de este problema mediante el método resolutivo *Backtracking* con *Graph-based Backjumping*, junto a lo cual se entrega algunos resultados de experimentos utilizando este método.

Antes de continuar, es necesario detallar qué es el *Progressive Party Problem*. El problema consiste basicamente en lo siguiente: se tiene una fiesta de yates donde cada anfitrión (entiendase el dueño del bote), dispone de su bote de manera que cada asistente a la fiesta pueda pasar por todos los botes, y además, los asistentes van cambiándose de yate cada cierto tiempo (normalmente cada 30 minutos). Todo ésto se debe lograr sin superar la capacidad máxima de pasajeros, y sin visitar el mismo bote más de una vez.

En esencia, en esa dinámica se modela el *Progressive Party Problem*, el cual es un problema combinatorio muy complejo.

En este documento se detallará a cabalidad los detalles del problema, tales como sus variables, las restricciones y el objetivo, de manera que se pueda apreciar de manera estandarizada el problema más allá de las modificaciones que proponen distintos autores. Además, veremos en qué posición se encuentra la ciencia de la computación en la actualidad para resolver éste problema. Finalmente, se presentará un modelo matemático que representará todas sus variables, restricciones y cualquier función que se estime necesaria.

Con este documento el lector podrá quedar completamente interiorizado y actualizado del *Progressive Party Problem*.

Este documento tiene el fin de entregarle al lector una completa produndización sobre el *Progressive Party Problem*, ademas de proponer una implementación en C++ utilizando el método de *Backtracking* con *Graph-based Backjumping* (BT + GBJ).

2. Definición del Problema

El *Progressive Party Problem* se formula de la siguiente manera: considérese una fiesta durante una reunión de yates. Existen n botes y sus tripulaciones. Un número determinado de botes es escogido para ser bote anfitrión: éstos serán los botes donde se realizará la fiesta, donde otras tripulaciones visitarán al bote en intervalos de tiempo $t = 1..T$ de media hora cada uno. El número total de periodos T viene dado. Las *tripulaciones visitantes* se mueven de un bote anfitrión a otro, y no pueden visitar un bote más de una vez. Los botes tienen una cierta capacidad limitada de tripulación a bordo, la cual no debe ser superada: esta es la capacidad de visitas de un bote anfitrión. Además, las *tripulaciones visitantes* no pueden encontrarse más de una vez con la misma *tripulación visitante*.

Lo que se busca es una calendarización que minimice la cantidad de botes anfitriones. A partir de lo recopilado en [1, 4, 10, 3], el problema se define[1] como:

2.1. Parámetros

- T (numero de periodos), el cual define cuantos intervalos de tiempos se manejarán en el problema.
- n (cantidad de botes), el cual dice cuantos botes existen disponibles para realizar la fiesta.[2]
- w_i (tamaño de la tripulación del bote), el cual indica cuantos tripulantes quedan fijos en el bote i.
- p_i (capacidad de tripulación del bote), el cual define la capacidad máxima de personas que puede soportar un bote i sin hundirse.

2.2. Objetivos

- Todas las tripulaciones visitantes deben visitar todos los yates anfitriones.
- Promover que los visitantes tengan la mayor cantidad de interacción social.

[1] En ningun lado existe un modelo estándar de PPP, pero en todas las publicaciones se respeta la formulación hecha por la publicación original [1], la cual es respetada en la definición utilizada en este documento.

[2] Recordar que se busca minimizar la cantidad de botes que se utilizarán finalmente, por lo que es posible que no se utilizen todos los botes disponibles como anfitriones.

- **Minimizar la cantidad de botes anfitriones.**
- Satisfacer las necesidades y restricciones de la fiesta.

Todos estos objetivos serán pulidos y ajustados a un listado de requerimientos que buscan que la fiesta se produzca sin accidentes (respetar limites de capacidad), que se mantenga socialmente activa (evitando que las tripulaciones visitantes se encuentren más de una vez), entre otros requerimientos.

2.3. Restricciones

Las restricciones generales de este problema son bastante simples y directas. Pero más adelante se podrá ver que existen varias representaciones para estas restricciones, aunque todas buscan cubrir los mismos requerimientos.

- Las fiestas solo se pueden realizar en los botes anfitriones.
- La capacidad de un bote no puede ser excedida.
- Las tripulaciones no pueden estar sin hacer nada: o estan visitando botes anfitriones, o bien ellos mismos son anfitriones.
- Las tripulaciones no pueden visitar el mismo bote más de una vez.
- Las tripulaciones visitantes no pueden cruzarse más de una vez.

2.4. Problemas Similares

El *Progressive Party Problem* es considerado un problema particular de *Timetabling* (programación de horarios), por lo que problemas que buscan calendarizar eventos y optimizarlos bajo algun criterio, son considerados problemas similares.

2.5. Variantes

En el articulo [1] y [4] se muestran algunas variantes que buscan reducir el numero de ecuaciones (y por ende el espacio de búsqueda), del problema. Algunas de esos cambios son:

- Se introduce una variable binaria adicional que indica si una tripulación j y j' visitan a un bote i en el mismo intervalo de tiempo determinado.
- Se introduce una variable entera que indica si un bote fue visitado por una tripulación en un instante determinado (variable con penalizaciones).

3. Estado del Arte

El *"Progressive Party Problem"* (PPP) fue postulado por primera vez y resuelto heurísticamente por Peter Hubbard, el cual es miembro del *"Sea Wych Owners Association"* y del Departamento de Matemáticas de la Universidad de Southhapton. Peter Hubbard era el organizador de una reunión de yates, donde debía, entre muchas otras cosas, organizar una fiesta para los participantes de los yates sobre los yates. [8]

P. Hubbard logró dar con la solución al problema heurísticamente (buscando manualmente), pero pensó que sería interesante ver si es que se podría encontrar una mejor solución y más optima. Le sugirió este problema a H. Paul Williams, el cual formuló el problema bajo *Programación Entera Mixta*, mientras que su colega Sally Brailsford trataba de resolver el problema

con software comercial de Programación Entera Mixta. Por otro lado, Barbara Smith usó técnicas de *Programación con Restricciones* (es decir, se formuló como CSP). Estos dos enfoques dieron paso al primer articulo publicado sobre el *Progressive Party Problem.* [1]

La publicación de P. Hubbard y sus colegas en 1995 [1] (la cual fue incluida posteriormente en otra publicación en 1996), se enfocó principalmente en detallar las diferencias empíricas que se obtuvieron al desarrollar la solución del PPP en Programación Entera (PE) y en Programación con Restricciones (PR). El resultado fue desastrozo para la PE, debido a que este tipo de problemas de *Timetabling* eran problemas comunmente resueltos bajo ese paradigma de programación, pero que sin embargo fue bastante alentador para la PR. Esta situación fue lo que inició la interesante "guerra" entre la Programación Entera y la Programación con Restricciones en torno al PPP.

En 1997-1998, un nuevo autor [9, 10], postula que las nuevas técnicas implementadas en la Programación Entera Mixta (MIP) son mucho mejores, por lo que utiliza MIP's para resolver el problema del PPP, lo cual antes había sido desastrozo.

Ya recién en 1999 se utilizó un nuevo acercamiento al PPP, esta vez utilizando *Local Search* (*Tabu Search, Simulated Annealing*), las cuales eran eficientes, pero no siempre óptimas[3].

De ahí en adelante, la literatura sobre el tema decayó, pero distintas implementaciones en distintos programas, métodos y representaciones han tenido publicaciones. [6, 7]

3.1. Métodos Utilizados Para Resolverlo

Para resolver el PPP, se menciona repetidamente en varios de los artículos que se utiliza Programación Entera o Mixta, para los cuales se utiliza el software GAMS[4], o también se menciona el uso de Programación con Restricciones[1]. Pero en los articulos más recientes, se hace uso explícito de métodos de búsqueda local, como *Tabu Search* y *Simulated Annealing.*

3.1.1. Programación Lineal Entera o Mixta

Es un modelo de programación que contiene restricciones y una función objetivo. En el caso de la Programación Entera, todas las variables deben ser enteras, en cambio para la Programación Mixta, las variables pueden tomar valores enteros como tambien binarios.

Este modelo de programación resulta bastante útil en especial para los problemas de tipo *Timetabling*, ya que las restricciones y la función objetivo de este tipo de problemas tienden a ser altamente representables mediante formulas matemáticas sencillas.

Software como GAMS, son altamente utilizados en la resolución de este tipo de problemas.

3.1.2. Programación con Restricciones

La programación con restricciones, es otro modelo de programación donde las variables se representan mediante las restricciones que las relacionan. En este tipo de paradigma, las restricciones de los modelos difieren de los modelos tradicionales de programación, ya que en vez de ejecutarse pasos o procedimientos, la Programación con Restricciones especifica las propiedades de las posibles soluciones. Este enfoque hace que la Programación con Restricciones sea parte del mundo de la Programación Declarativa.

Un software bastante conocido en el mundo académico que utiliza este modelo de programación es Prolog, pero en los estudios del PPP se menciona CHIP.

[3] Este es un problema común al utilizar estos métodos

[4] `http://www.gams.com/`

3.1.3. *Simulated Annealing*

Simulated Annealing es un procedimiento de búsqueda local para explorar el espacio de soluciones más allá del óptimo local, el cual busca reducir el problema de los algoritmos de búsqueda local, aceptando soluciones de "peor calidad".

Al igual que otros algoritmos de búsqueda local, *Simulated Annealing* necesita una solución inicial, y comienza explorando por el vecindario de todos los optimos locales. A medida que el tiempo pasa, una función de temperatura va disminuyendo, lo cual obliga al algoritmo a explotar más a los óptimos locales en vez de explorar.

Este método necesita definir una representación de los datos, un movimiento para explotar, una función de evaluación que indica que tan buena es una solución, una temperatura inicial y su respectiva función de decaimiento, y el número de iteraciones máximas para el algoritmo.

3.1.4. Metropolis

El algoritmo Metrópolis es una version simplificada del *Simulated Annealing*, donde su princincipal diferencia radica en que en vez de tener una función de decaimento para la temperatura, la temperatura se mantiene constante durante toda la ejecución.

El algoritmo avanza buscando en su vecindario al azar, y luego acepta o no la solución dependiendo de una función probabilística. Es decir, igual que el funcionamiento de *Simulated Annealing*.

3.1.5. *Tabu Search*

Tabu Search es un algoritmo de búsqueda local, que busca explotar y explorar el espacio de búsqueda, de acuerdo a una estructura de memoria adaptativa y flexible.

Al igual que *Simulated Annealing, Tabu Search* necesita una solución inicial para comenzar a explorar, y va almacenando en memoria la lista tabú, para evitar que se produzcan ciclos, y además, puede aceptar soluciones peores a la actual, pero siempre mantiene en memoria a la mejor solución.

Este método necesita definir una representación de los datos, un movimiento para explotar, una función de evaluación que indica que tan buena es una solución, el tamaño de la lista tabú, y el número de iteraciones máximas para el algoritmo.

3.2. Heurísticas y Metaheurísticas

En los distintos articulos revisados, se pueden apreciar distintas técnicas de algoritmos de heurísticas, las cuales buscan ayudar el procesamiento de soluciones para el problema.

En la formulación de Búsqueda Local [3, 9], se utilizan técnicas que buscan optimizar los movimientos realizados para explorar en el espacio de búsqueda. En el caso de la Búsqueda Local se utiliza la heurística *Min-Conflict*[3].

En la heurística *Min-Conflict*, al tener un set de variables (que en el caso de la Búsqueda Local es el vecindario de una solución), se va a tratar de avanzar a la solución que tenga la menor cantidad de violaciones a las restricciones del problema[5], con el fin de priorizar la "reparación" de las soluciones encontradas[5].

Para los artículos donde se utiliza Programación Lineal o Programación con Restricciones, las heurísticas no juegan un papel tan relevante, como lo es para la Búsqueda Local. Esto es debido a que la Programación Lineal trabaja con las variables definidas directamente.

A pesar de esto, se menciona que para la publicación original [1], se pre-procesa el problema mediante el software GAMS, para luego ser resuelto por el software CPLEX 7.0. Esto se hace

[5]Recordar que técnicas como *Tabú-Search* y *Simulated Annealing*, pueden revisar espacios infactibles de soluciones, con el fin de salirse de óptimos locales.

para aplicar las heurísticas utilizadas en GAMS (las cuales se basan en los algoritmos *branch-and-bound*), y así simplificar la tarea que debe realizar CPLEX 7.0 para resolver el problema "reducido". Y para el caso de la Programación con Restricciones, simplemente se aplica un ordenamiento de acuerdo al dominio de las variables[6].

En el artículo [4] se menciona el uso de una heurística específica para el problema, el cual se basa en "etapas de tiempo"(*Time-Stage*), el cual básicamente hace un pre-procesamiento del problema para el primer periodo de tiempo, para luego reutilizar esa información en los siguientes periodos de tiempo, generando asi una solucion inicial.

3.3. Resultados

Considerando los 3 enfoques mencionados en los artículos, se puede ver que claramente el mejor método utilizado es el de Búsqueda Local en cuanto a los tiempos de ejecución. Pero sus representaciones son mucho más elaboradas, refinadas y complicadas, debido a que se introducen restricciones con castigos, lo cual aumenta la eficiencia, pero destruye la semántica original del problema[7].

A partir de lo que se ha recopilado, los artículos [3, 4] han hecho una buena recopilación de las diferencias al ejecutar los distintos métodos. Es por esto, que en este documento utilizaremos la información referenciada en los artículos mencionados, con el fin de simplificar la entrega de los datos.

En la publicación original [1], se menciona que la Programación Lineal fue un total fracaso, y la única vez que sus modelos arrojaron un resultado, al corroborarlo el sistema no pudo lograrlo, ya que después de 189 horas ejecutándose, el programa no logró converger a un resultado satisfactorio. Además, la cantidad de variables y filas de ecuaciones generadas al ejecutar los algoritmos, estaban al borde de la capacidad de los computadores de la época utilizados en el estudio.

Sin embargo, los resultados fueron mas optimistas al utilizar Programación con Restricciones, ya que logró resolver en pocos segundos dos de los problemas que Programación Lineal no pudo resolver. Sin embargo, para instancias mas grandes, el método falló.

En el artículo [4], se logra aplicar una nueva representación al problema, y esta vez, con computadores más potentes y mejores pre-procesadores, se logró dar con soluciones optimas al cabo de pocos segundos, pero instancias superiores no puedieron ser corroboradas debido a la heurística utilizada.

En la tabla a continuación se puede ver tabulada los resultados al resolver el programa con Programación Entera Mixta (MIP):

En la tabla podemos ver que a la séptima iteración se logró hallar una solución óptima. Pero debido al funcionamiento de la heurística (la cual fue explicada anteriormente), el método no puede seguir comprobando para $t > 7, \forall t \in [1, T]$, lo cual es un inconveniente si se quiere establecer un numero fijo de botes anfitriones.

Para el caso de la Búsqueda Local, el artículo [3] de una buena comparación de todos los métodos utilizados.

En el artículo se compara la Programación Lineal y la Programación con Restricciones usada en [1], incluyendo la segunda formulación postulada para la Programación con Restricciones

[6] Es importante recordar que para este artículo, el algoritmo más eficiente fue el utilizado en Programación con Restricciones

[7] Lo cual no es malo, pero su formulación se aleja mucho de lo original.

etapa de tiempo	nº ecuaciones	nº variables	nº elem. no nulos	var. discretas	tiempo generación	tiempo solución
1	38830	2620	114130	1758	0,73	1,62
...	...	...	...	...	...	...
(7)	245470	77500	322876	1722	3,42	5,50

Cuadro 1: Resultado simplificado usando la heurística *Time-Stage*

propuesta en el mismo artículo. Los resultados podemos observarlos tabulados a continuación:

problema	ILP	CP1	CP2	LS
P_6	fallo	27 min.	pocos seg.	< 1 s.
P_7	fallo	28 min.	pocos seg.	< 1 s.
P_8	fallo	fallo	pocos seg.	1 s.
P_9	fallo	fallo	horas	4 s.
P_10	fallo	fallo	fallo	fallo

Cuadro 2: Resultado de las ejecuciones de los distintos métodos

P_i indica que el problema está instanciado para i períodos de tiempo.

Podemos observar que los resultados son consecuentes con lo presentado en [1], dado que la Programación Entera falla en todas las instancias, mientras que la Programación con Restricciones lo resuelve para instancias pequeñas. Lo interesante de este resultado es que la Búsqueda Local puede encontrar resultados para instancias mucho mas grandes en tiempos muy pequeños comparado con las demás técnicas.

A pesar de esto, el método falla cuando se le instancia con 10 períodos de tiempo, lo cual no concluye si es que el método logró encontrar solución para esa instancia o no.

Podemos afirmar con seguridad, que métodos de Búsqueda Local son considerablemente mejores que métodos que buscan en los espacios de búsqueda completos, como lo hacen la Programación Lineal (aunque se desconocen sus métodos).

4. Modelo Matemático

Existen varias formas de plantear este modelo, pero se busca preservar la simplicidad y semántica que respeten la originalidad del problema.

Es por eso que las representaciones y restricciones aquí definidas, están basadas en la publicación original del *Progressive Party Problem* [1], complementada con la información disponible en [6], y respaldada por la publicación [4].

4.1. Datos y Constantes

- T = intervalos de tiempo definidos para el problema.
- n = cantidad de botes que participarán de la fiesta.

- w_i = tamaño de la tripulación del bote i.
- p_i = cantidad máxima de personas que soporta el bote i.
- $g_i := max\{p_i - w_i, 0\}$ cantidad máxima de invitados que puede recibir el bote anfitrión i.

4.2. Variables

Las variables definidas a continuación son variables que se pueden encontrar a lo largo de todos las publicaciones, y se pueden considerar un modelo matemático base para el PPP.

$$x_{i,j,t} = \begin{cases} 1 & \text{si la tripulación } j \text{ visita el bote } i \text{ en el instante } t, \\ 0 & \text{si no lo hace.} \end{cases} \tag{1}$$

$$h_i = \begin{cases} 1 & \text{si el bote } i \text{ es un bote anfitrión,} \\ 0 & \text{si no lo es.} \end{cases} \tag{2}$$

A continuación se define una variable introducida en el articulo [4], debido a que esta formulación de la variable permite una semántica mucho mas cercana a la propuesta por la publicación original, ademas de evitar lo inconveniente de agregar variables con castigo, como se hace en el artículo donde se utiliza Búsqueda Local [3]. Esto se puede corroborar al mirar que otro autor que utiliza la Búsqueda Local, prefiere la representación sin variables con castigos [9].

$$m_{j,j',t} = \begin{cases} 1 & \text{si la tripulación } j \text{ y } j' \text{ se encuentran en el mismo bote en el instante } t, \\ 0 & \text{si no lo hacen.} \end{cases} \tag{3}$$

4.3. Función Objetivo

La función objetivo de este problema es minimizar la cantidad de botes anfitriones. A pesar de esto, en la literatura se puede apreciar que este valor (la cantidad de botes anfitriones), se define de manera fija como una constante [4, 1, 9, 3], con el fin de reducir la complejidad del problema, y así convertir a la función objetivo en una restricción más.

$$\min z = \sum h_i \quad \forall i \in [1, n], i \neq j \tag{4}$$

4.4. Restricciones

Como se ha mencionado anteriormente, las restricciones acá representadas buscan mostrar la mayor cantidad de semántica fiel al artículo original.

La primera restricción es que un bote puede ser visitado solo si el bote visitado es un bote anfitrión:

$$x_{i,j,t} \leq h_i \quad , \forall i, j \in [1, n] \quad \forall t \in [1, T] \tag{5}$$

La otra restricción utilizada en el modelo, es la restricción de que no se puede superar la capacidad máxima de pasajeros invitados en un bote. Esta se define como:

$$\sum_{j|j \neq i} w_j x_{i,j,t} \leq g_i \quad , \forall i, j \in [1, n] \quad \forall t \in [1, T] \tag{6}$$

La siguiente restricción permite asegurar que la tripulación de un bote anfitrión no puede abandonar su yate, ya que ellos son los anfitriones. Esto se define de la siguiente manera:

$$h_i + \sum_{i|i \neq j} x_{i,j,t} = 1 \quad , \forall i, j \in [1, n] \quad \forall t \in [1, T] \tag{7}$$

Existe otra restricción relacionada con los movimientos que pueden realizar las tripulaciones. Se requiere que una tripulación invitada no visite un mismo bote más de una vez durante la fiesta, aunque no se requiere que visite todos los botes de las fiesta. Esto se puede leer como:

$$\sum_{t} x_{i,j,t} \leq 1 \quad , \forall i, j \in [1, n], i \neq j \quad \forall t \in [1, T] \tag{8}$$

La siguiente restricciones buscan modelar las interacciones que tienen las tripulaciones visitantes durante la fiesta.

Se mencionó anteriormente, que el modelo que se representa acá es el que se puede ver en [4]. En este modelo se introduce la variable binaria $m_{j,j',t}$ del encuentro de la tripulación j con la tripulación j' en el tiempo t. La restricción que define esta variable indica que si dos tripulaciones se han encontrado en el mismo lugar al mismo tiempo, es porque estuvieron en el mismo bote al mismo tiempo:

$$m_{j,j',t} \geq x_{i,j,t} + x_{i,j',t} - 1 \quad , \forall (j, j', t) | j < j' \quad \forall t \in [1, T] \tag{9}$$

Ahora que la variable $m_{j,j',t}$ se encuentra debidamente definida bajo la restricción antes mencionada, podemos modelar la restricción que nos pide que las tripulaciones visitantes no se puede encontrar más de una vez con otras tripulaciones visitantes durante la fiesta. Esto se puede ver como:

$$\sum_{t} m_{j,j',t} \leq 1 \quad , \forall j < j' \quad \forall t \in [1, T] \tag{10}$$

5. Representación

De acuerdo a la interpretación que se ha realizado del problema, en conjunto con la implementación de un algoritmo de *Backtracking*, el problema se representa en base a la siguientes estructuras de datos:

- ***solucion(i,t):*** matriz de nxT nodos, donde cada nodo representa el barco donde se encuentra la tripulación del barco $i \in [1, n]$ en el periodo $t \in [1, T]$. En el modelo matemático presentado anteriormente, se utiliza una representación binaria de esta estructura de la forma "*solucion(i,j,t)*", la cual tiene un espacio de búsqueda extremadamente grande (2^{nxnxT}), mientras que la representación propuesta tiene un espacio de búsqueda de solo n^{Txn}[8]. Esta representación permite, además, analizar el comportamiento de los botes de manera mucho más gráfica. Por ejemplo, para T=6:

[8]Particularmente, las instancias propuestas tienen valores de $n = 42$ y $T \in [6, 8]$, por lo que el E.B. de la representación original es del orden de 10^{3186}, mientras que la nueva representación es de solo del orden de 10^{477}.

i (bote visitante)	Período 1	Período 2	Período 3	Período 4	Período 5	Período 6
1	bote host	bote host	bote host	bote host	bote host	bote host
...	...	...	...	...	...	...
n	bote host	bote host	bote host	bote host	bote host	bote host

Cuadro 3: Representación de la estrucutra de datos de *solucion(i,t)*.

- ***mismo_bote(i,j,t):*** matriz binaria de $nxnxT$, donde cada nodo indica si un barco i visitó al bote j en el instante t. Esta estructura hace referencia a la restricción expresada en la ecuación (10), se utiliza como una estructura adicional para "facilitar.[el] cómputo de esa restricción. Técnicamente, esta representación es lo mismo que la representación anterior, pero sus usos son distintos, y el algoritmo implementado para resolver el problema, está optimizado para la representación anterior. Gráficamente, es complicado mostrar cómo es esta estructura de datos, dado que es una estructura con 3 dimensiones, y por lo mismo se omitirá su representación gráfica, y se apelará a la capacidad comprensiva del lector.

  Esto fue utilizado así debido a la forma en que funciona *Backtracking*, y para optimizar la ejecución de éste al realizar las comprobaciones de consistencia, las cuales deben evitar ser lentas o computacionalmente costosas.

6. Descripción del algoritmo

Backtracking (BT) es un algoritmo de búsqueda completa, el cual se utiliza para resolver problemas de satisfacción de restricciones (CSP). En el trabajo realizado, el desarrollo realizado es mediante una variante de la misma, llamada *Graph-based Backjumping* (GBJ).

BT con GBJ [2], es un algoritmo de búsqueda completa del tipo "*look back*", pero que en vez de hacer saltos hacia atrás de manera cronológica, lo hace de manera "inteligente". Estos saltos los hace identificando la variable más arriba en el arbol de BT, que está generando conflicto con la actual instanciación; luego se hace un *back jump* a esa variable culpable.

La gran diferencia al juntar GBJ con BT respecto al BT puro, es que GBJ busca evitar el *trashing*, el cual consta de estar constantemente buscando en sectores que se sabe que serán infactibles. Pero GBJ falla cuando tenemos un grafo de restricciones fuertemente conexo, ya que al estar gran parte de las variables con conflicto, el *back jump* siempre será a la variable anterior, por lo cual el BT con GBJ se transforma en BT puro practicamente. Esto se puede ver corroborado en el estudio realizado en [2].

En la formulación del problema se exige que el objetivo del problema se enfoque en "maximizar la cantidad de bloques de duración de la fiesta comenzando desde $T = 6$". Pero este objetivo está cubierto bajo la restricción "Cada tripulación invitada debe siempre tener un anfitrión asociado", dado que la representación elegida para el problema asegurará que siempre se maximizará la cantidad de bloques de duración de la fiesta.

Inicio → Leer Archivos → Algoritmo BT+GBJ → Salvar/Mostrar Información → Fin

Figura 1: Esquema general del programa.

De acuerdo a lo planteado, se plantea el esquema general del programa en la figura 1. La lectura de archivos se hace en base a los datos de las instancias proporcionadas por Camila

Diaz (*input* del algoritmo), gracias a lo cual podemos definir las capacidades, tripulaciones y los anfitriones entre todos los participantes del problema. Esta información se obtiene desde 2 archivos distintos:

- **ppp.cap**: contiene los datos de todos los botes que participarán en el problema. Por cada linea se tienen 2 números; cada linea representa un barco y los numeros representan la capacidad y la tripulación a bordo respectivamente.
- **ppp_x.hst**: archivo donde se definen que barcos serán anfitriones, y cuales serán vistantes. Cada linea del archivo representa un barco, y cada linea contiene un "1" o "0" representando si es anfitrión o no respectivamente. Pero existen distintas instancias, y en este caso, tenemos hasta 6 instancias (ppp_1.hst, ppp_2.hst, etc...).

Dado que se debe definir qué instancia utilizar, y cuantos períodos se utilizarán, el programa logra distinguir si es que los parametros son enviados por linea de comandos, y si no lo están, se le pide explicitamente al usuario ingresar manualmente la instancia y los períodos.

Una vez que se hace define la instancia y los períodos, se procede a guardar los datos en 2 variables. Una matriz de $nx2$, y un vector de largo n (donde n es la cantidad de barcos en el problema):

```
vector< vector<int> > botes;
vector<int> instancia;
[...]

int main(int argc, char **argv)
{
    [...]
    botes = utils.get_boats();
    instancia = utils.get_instance(n_instancia);
    [...]
```

Estas variables son utilizadas para comprobar las distintas restricciones dentro del problema, tales como identificar si un barco es anfitrión o no.

Una vez que esto queda definido, pasamos a ejecutar el algoritmo de BT con GBJ. Para ello, se implementó BT con el método iterativo por sobre el recursivo. La razón de esto es que debido a la gran cantidad de variables que se tienen en juego, las ramificaciones del algoritmo recursivo genera un gran consumo de memoria RAM (en ciertas ocaciones por sobre los 4 GB), y debido a que el computador de desarrollo tiene solo 4 GB, se optó por seguir la implementación iterativa del BT, debido a que el consumo de memoria es mínimo, debido a que el *stack* de variables se maneja manualmente y de manera lineal. De esta manera se evitan efectos indeseados de rendimiento al agotarse la RAM.

```
Array2D solucion(n,T,-1); // matriz de nxT inicializado con valores -1
Array3D mismo_bote(n,n,T,0); // matriz de nxnxT inicializado con valores 0
stack<node> auxiliar;
```

Las primeras dos variables definidas aquí son las variables más importantes, ya que definirán las soluciones del problema y servirán como columna vertebral para el backtracking, en especial la variable "solucion", ya que "mismo_bote" es una variable auxiliar.

La variable "auxiliar", es el stack que se utilizará para hacer *backtracking*. En ella se almacenarán los nodos del árbol del *backtracking*.

Para explicar la parte central del algoritmo que viene a continuación, se utilizará seudocódigo, para facilitar la comprensión:

- **i**: variable que representa la tripulación del bote que va a salir donde un anfitrión.
- **j**: variable que representa un instante de tiempo.
- **b**: variable que representa el anfitrión que recibirá a "i".

```
mientras (i < n) y (j < T):
    instanciamos solucion(i,j) = b

    si estamos en una hoja:
        si la instanciación es consistente:
            se encontró una solución

            si es el último valor del dominio:
                hacemos backtrack

                si no se puede hacer backtrack:
                    finalizar
            si no:
                b = b + 1

        si no es consistente:
            si es el último valor del dominio:
                hacemos backtrack

                si no se puede hacer backtrack:
                    finalizar
            si no:
                b = b + 1

    si estamos en un nodo intermedio:
       si la instanciación es consistente:
           avanzamos a la siguiente variable, y recordamos el paso en el stack

       si no es consistente:
           si es el último valor del dominio:
                hacemos backtrack

                si no se puede hacer backtrack:
                    finalizar
            si no:
                b = b + 1
```

Durante toda la ejecución del BT con GBJ, se muestran por pantalla datos releventes, como las iteraciones, conflictos, soluciones, etc. Y cada vez que se encuentra una solución, esta se escribe a un archivo el cual simplemente se va sobre-escribiendo. El archivo tendrá n lineas que representan cada uno de los botes, y cada linea tendrá T columnas, con las calendarizaciones de la fiesta.

Es importante también explicar que es lo que hace la función de consistencia. Básicamente revisa si se cumplen todas las restricciones, y si no se cumple alguna, acusa a la variable culpable de fallar.[9]

Las restricciones del problema se separaron en 5 funciones, para así tener un control claro de lo que se está tratando y qué se está tratando:

- **restriccion_1**: función que se se encarga de verificar que solo los botes anfitriones pueden recibir visitas, además de asegurarse que los tripulantes del bote anfitrión no salgan de su bote.
- **restriccion_2**: función que se encarga de comprobar que las capacidades de tripulación de un bote para un instante determinado, no se vean sobrepasadas.
- **restriccion_3**: función que se encarga de verificar que todos los invitados tengan un anfitrión al cual asistir.
- **restriccion_4**: función que se preocupa de evitar que una tripulación invitada vuelva más de una vez a un mismo bote anfitrión.
- **restriccion_5**: función que verifica que cualquier pareja de tripulantes esté a lo más una vez junta sobre un anfitrión cualquiera.

7. Experimentos

El código fue ejecutado en un computador con las siguientes características:

- Procesador: Intel Core 2 Duo E7500 3.2GHz
- Memoria RAM: 4 GB
- Disco Duro: Partición exclusiva de 80 GB
- Sistema Operativo: Ubuntu 14.04 x64
- IDE: Qt Creator

El primer experimento consistió en comparar una implementación de BT con GBJ y una con BT puro, para determinar cual era más eficiente en términos de tiempo neto. Debido a la falta de experiencia, no se registraron los datos de iteraciones, chequeos de restricciones, backtracks ni nada de eso[10], y solo se logró verificar que debido a la fuerte conexión entre las variables debido a las restricciones, el BT con GBJ se comporta de manera casi idéntica, pero como BT con GBJ tiene más funciones preocupadas de hacer saltos inteligentes, este último terminaba siendo más lento que el BT puro.

Otro experimento realizado consistió en generar instancias más pequeñas que las proporcionadas por Camila Diaz. Estas instancias fueron de solo 7 a 15 botes, con distintas configuraciones de botes anfitriones. Esto se realizó con el fin de definir de manera rápida y efectiva, los comportamientos que tenía el programa para distintas configuraciones de anfitriones.

Finalmente, en las etapas inciales del desarrollo del programa final[11], se ejecutó el algoritmo para poder verificar que la solución de referencia entregada por Camila Diaz estaba siendo generada por el algoritmo.

[9] Esto fue cambiado más adelante en un intento de mejorar el rendimiento del programa.

[10] Repetir el experimento estaba fuera de las opciones debido al alto tiempo que se demoraron los programas en encontrar soluciones

[11] al cual lamentablemente olvidé añadir un contador de iteraciones, y que además generó unas soluciones erroneas debido a una mala programación de la restricción 5

8. Resultados

Para el primer experimento, donde se comparó la implementación de BT puro versus BT con GBJ, pero no se registraron las iteraciones ni nigún indicador similar, debido a la falta de experiencia por parte del programador, pero si se logró comparar la velocidad en tiempo real entre las dos implementaciones.

Para el caso del BT puro, el problema se terminó de ejecutar 14 minutos antes que el de BT con GBJ, aunque se ignora la cantidad de iteraciones que hizo cada uno. Aún así se puede concluir que debido a que el grafo de restricciones del *Progressive Party Problem*, es altamente conexo, el BT con GBJ tiende a comportarse como BT puro, y como la implementación de BT con GBJ tiene que hacer comprobaciones extras para determinar a cual variable hacer el salto inteligente, se asume que esos calculos extras impactaron negativamente en el problema.

De todas maneras, se adjunta el código de las dos implementaciones.

Para los otros experimentos, se utilizaron instancias semi-triviales, para verificar diferencias de comportamiento para el algoritmo. A continuación se presenta una tabla con algunos de las instancias y los resultados obtenidos:

nº botes	anfitriones	iteraciones	nodos consist.	nodos no consist.	*Backtracks*	nº sols.
7	1-6	10183	1992	8191	1272	720 (*)
7	2-7	217253	-	-	-	fallo
7	1-4	711	88	623	88	0
10	1-4	978	88	890	88	0
42	1-12	-	-	-	-	11 (**)

Cuadro 4: Instancias de prueba

(*) Esto es un caso "trivial", donde se espera que la cantidad de soluciones sean exactamente $6! = 720$, que corresponden a la cantidad de permutaciones posibles entre los anfitriones disponibles.

(**) Esta version del algoritmo se ejecutó con un error en la restricción 5, la cual pudo haber generado soluciones adicionales.

Gracias a estos experimentos y a otros similares, se descubrió que el mínimo de anfitriones en una instancia debe ser de al menos la cantidad de periodos de tiempo, es decir, $\sum_{i=0}^{n} h_i \geq T$, lo cual es interesante, considerando que en ningun momento se supone ese dato, ni se menciona en ninguna publicación.

Esa información podría ser valiosa para buscar soluciones para instancias generalizadas, donde T también es una incógnita, y no como en este caso, donde T viene dado.

La otra información rescatable, es que al poner los anfitriones agrupados hacia al final de la lista (como se muestra en la fila 2 del cuadro), se generan muchisimas más iteraciones que en el caso homólogo de la fila 1. Esto se produce debido a que el BT intenta asignar como host siempre a la primera variable, generandose muchos más conflictos, lo cual al parecer, agotó el stack del BT y emite un *Segmentation Fault*.

Finalmente, para la ejecución del experimento para verificar que la solución de referencia era generada por la aplicación, después de largas horas, se identificó un error en una de las restricciones, y que la linea que muestra los resultados, estaba comentada, por lo que se perdió información valiosa y el tiempo de ejecución se extendió muchisimo más debido a la cantidad de soluciones adicionales que se tuvieron que comprobar (cerca de 17 soluciones generadas). Pe-

ro finalmente, esto fue comprobado, y la solucioón de referencia estaba contenida dentro de las soluciones generadas por el algoritmo.

9. Conclusiones

- El *Progressive Party Problem* describe un problema del tipo *Time Tabling* que fue planteado casi fortuitamente por P. M. Hubbard, donde los resultados fueron desconcertantes inicialmente, debido a que los métodos para resolver ese tipo de problemas no arrojó los resultados esperados. De esto último se desprende que la Programación Lineal no es el método más óptimo para resolver el *Progressive Party Problem*.
- Entre los métodos para resolver el *Progressive Party Problem* que se vieron en este documento, como por Programación Lineal, Programación con Restricciones y Búsqueda Local, se logró comprobar que el enfoque de Búsqueda Local es el más eficiente en general para las distintas instancias del problema.
- Las variantes encontradas para el problema, no son de mayor incumbencia, ya que los objetivos no se alejan de los originales, pero sus representaciones sí. Esto último se ve reflejado negativamente en la semántica de las restricciones para los distintos métodos que usan los autores. Y es más, en una representación utilizada en la Programación Lineal, elimina las restricciones cargadas de una semántica alejada de la original.
- El modelo matemático utilizado para representar el problema es el más sencillo y el más simple en términos semánticos, lo cual facilita la lectura.
- Aplicar el método de *Backtracking* con *Graph-based Backjumping*, al *Progressive Party Problem*, entrega soluciones muy satisfactorias, las cuales se resuelven en tiempos considerablemente aceptables (en el rango de horas).
- Se demostró que el *Progressive Party Problem*, es un problema fuertemente conexo a nivel del grafo de restricciones, lo cual lo hace inmune a la mejora propuesta por el *Graph-based backjumping*.
- Finalmente, se puede concluir que el *Progressive Party Problem* es un problema combinatorio altamente complejo, el cual tiene una inclinación a resolverse más facilmente mediante técnicas incompletas por sobre las completas. Su representación sencilla es engañadora, ya que su complejidad tiende a ser alta, pero distintas heurísticas ayudan a facilitar el proceso de encontrar soluciones. Es por esto último que adaptar distintas heurísticas a este problema es algo necesario, ya que el único límite parece ser el ingenio humano.

10. Bibliografía

Referencias

[1] B.M.Smith, S.C.Brailsford, P.M.Hubbard, and H.P.Williams. The progressive party problem: Integer linear programming and constraint programming compared. *Constraints*, 1:119–138, 1995.

[2] Peter Clark and Rob Holte. The progressive party problem. `http://www.cs.utexas.edu/users/pclark/papers/gbj.pdf`, 1992. (Ultima visita el 17/07/2014).

[3] Philippe Galinier and Jin-Kao Hao. Solving the progressive party problem by local search. In Stefan Voß, Silvano Martello, IbrahimH. Osman, and Catherine Roucairol, editors, *Meta-Heuristics*, pages 419–432. Springer US, 1999.

[4] Erwin Kalvelagen. On solving the 'progressive party problem as a mip. http://amsterdamoptimization.com/pdf/ppp.pdf, 2002. (Ultima visita el 26/05/2014).

[5] Steven Minton, Mark D. Johnston, Andrew B. Philips, , and Philip Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Eighth National Conference on Artificial Intelligence (AAAI-90)*, 1990.

[6] Helmut Simonis. Customizing search (progressive party problem). http://4c.ucc.ie/~hsimonis/ELearning/party/handout.pdf, 2009. (Ultima visita el 17/07/2014).

[7] Helmut Simonis. Progress on the progressive party problem. In Willem-Jan Hoeve and JohnN. Hooker, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, volume 5547 of *Lecture Notes in Computer Science*, pages 328–329. Springer Berlin Heidelberg, 2009.

[8] Joachim Paul Walser. The progressive party problem. http://www.ps.uni-saarland.de/~walser/ppp/ppp.html, 1997. (Ultima visita el 26/05/2014).

[9] Joachim Paul Walser. Solving linear pseudo-boolean constraint problems with local search. In *FOURTEENTH NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, 1997.

[10] Joachim Paul Walser. *Domain-Independent Local Search for Linear Integer Optimization*. PhD thesis, Universität des Saarlandes, 1998.