Universidad de Buenos Aires

Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Métodos Numéricos

Tiempos Valientes.

Integrantes	LU	Correo electrónico
Sabogal, Patricio	693/14	pato.sabogal@hotmail.com

Indice

1 Introducción

Este trabajo tiene el objetivo de realizar la implementación de un algoritmo para conseguir la máxima cantidad de agentes confiables dada una encuesta donde los agentes se votan mutuamente dando a conocer si confía de otro agente o no. Se utilizará la técnica *backtracking* para la resolución del problema, dado que el problema se puede modelar bien con un árbol decisiones.

En el informe se propondrá un algoritmo para resolver el problema, mostraremos su correctitud y además implementaremos una mejora del algoritmo utilizando dos podas distintas del árbol de decisión. Cuando hablamos de podas nos referimos métodos para descartar ramas del árbol de decisión que, dependiendo de la efectividad de la poda, disminuirá en mayor o menor medida el tiempo de ejecución del algoritmo de *backtracking*.

Uno de los desafíos que presenta este problema es que a menudo la dimensión del árbol de decisión crece exponencialmente en base a la cantidad de agentes pertenecientes al conjunto y en consecuencia también lo hace el tiempo de computo para encontrar una solución.

2 Desarrollo

Para la implementación de los algoritmos de *backtracking* elegimos como lenguaje C++ por su alto rendimiento en tiempo de ejecución. En particular elegimos usar C++11, que tiene funciones muy útiles para medir tiempos de ejecución y el *parseo* de los datos, como por ejemplo la librería *chronos* de la biblioteca estándar.
Los votos son representados como una matriz de tamaño *#agentes***X***#agentes*, donde la fila i representa los votos del agente i para con los demás agentes (también puede votarse a si mismo). Cada posición de la matriz puede contener los valores *1*, *0* o *-1*, si el voto en la posición es positivo, nulo (no hubo voto), o negativo respectivamente. En la siguiente figura podemos ver como quedaría la matriz para un input de prueba con 4 agente y 5 votos:

Agente	Voto
1	2
1	-4
2	-3
3	1
3	4

	1	2	3	4
1	0	1	0	-1
2	0	0	-1	0
3	1	0	0	1
4	0	0	0	0

Figura 1: Matriz de votos.

Además se tomó la decisión de que si un agente vota incosistentemente con sus votos anterior, es decir que en algún momento voto a otro agente (o a el mismo) positivamente y en otro momento lo vota (o se vota) negativamente, solo importará el voto negativo, ya que este agente no es de confiar.
La ventaja de tener esta matriz es que permite revisar los votos de un agente en tiempo constante, es decir $O(1)$. Generar esta matriz tiene orden de complejidad $O(n^2 + a)$ con n cantidad de agentes y a cantidad de votos.

3 Algoritmo de Backtracking

El programa consta de dos etapas, primero se extraen y *preparan* los datos, y luego se llama al algoritmo de backtracking. El algoritmo de llamado para resolver el problema en sí no es muy interesante, su función es preparar las estructuras que se utilizaran por el propio algoritmo de *backtracking*. A continuación veremos el *pseudo-código* de las dos funciones:

Algorithm 1 Confiables ← función (Votos , Cantidad de Personas)

```
1: Confiables ← {}
2: Recorridos ← {}
3: Restantes ←{1...Cantidad de Personas}
4: return Backtracking(Matriz de votos, Confiables, Recorridos, Restantes)
```

Algorithm 2 Backtracking ← función (Votos , Confiables, Recorridos, Restantes)

```
1: IF Consistente(Votos, Confiables, Recorridos, Restantes) return 0 FI
2: IF tamaño(Restantes) = 0 THEN return tamaño(Confiables) FI
3: ELSE
4: Nuevo ← dameUno(Restantes)
5: Restantes ← Restantes - Nuevo
6: Recorridos ← Recorridos ∪ {Nuevo}
7: return MAX(Backtracking(Votos, confiables, recorridos, restantes),Backtracking(Votos,
   confiables ∪ {Nuevo}, recorridos, restantes))
```

Como podemos ver en el *pseudo-código*, en cada llamada a la función de *backtracking* primero se fija de que el conjunto de confiables sea consistente (según algún criterio), luego verifica si se llegó a un hoja del árbol de recesión y sino elige un nuevo elemento del conjunto de restante y explora los dos caminos del árbol de decisión, es decir que se mete en las ramas de agregar el nuevo elemento y no agregarlo. Una vez dentro de cada rama verificara si el conjunto es consistente o no, de no serlo el recursión detendrá y se retornara 0 ya que el conjunto no es consistente.

En la siguiente figura podemos ver mejor esta idea representada como un árbol de decisiones:

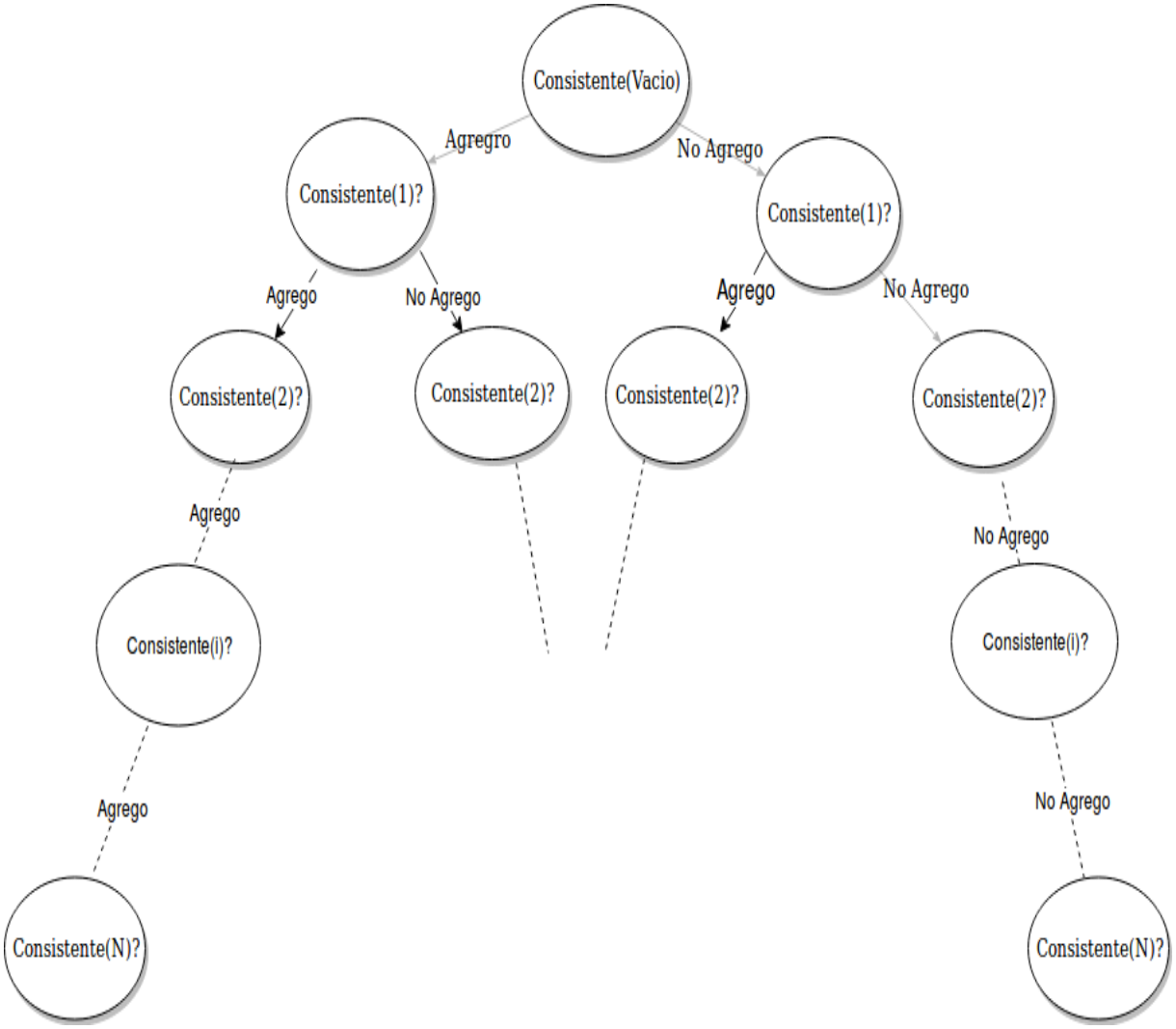

Figura 2: Arbol de decisiones.

Como podemos ver en la figura por cada agente, se generan dos ramas nuevas en el árbol de decisión. Esto es similar a, dado un cojunto, generar su conjunto de partes. De esta idea podemos ver que la complejidad de la llamada recursiva tiene peor caso de orden $O(2^n)$, con n igual a la cantidad de agentes del problema.

La verificación de la consistencia de un conjunto de confiables se divide en 2 etapas:

1. Verificar que el nuevo elemento no vote negativamente a nadie de los agentes pertenecientes al conjunto de confiables y viceversa, es decir, que nadie del conjunto vote al nuevo negativamente.

2. Verificar que el nuevo conjunto no haya votado positivamente a nadie de los elementos ya recorrido, que no pertenezcan al conjunto de confiables, es decir, a los no confiables.

El primer *chequeo* se realiza en orden de $O(n)$ con n igual a la cantidad de agentes del problema, o en otras palabras, se recorre linealmente el conjunto de confiables, verificando que los votos del nuevo con los anterior sean positivos o nulos, donde podrían, en peor caso, estar todos los agentes del problema.
El segundo *chequeo* se realiza en orden de $O(n^3)$, y la justificación de la complejidad se verá mejor explicada con el siguiente *pseudo-código*:

```
1: For i in Recorridos                                          // O(n^3)
2:    For j in Confiables                                       // O(n^2)
3:       IF i notIn(Confiables) & votoPositivo(j,i) Return false FI          // O(n)
4:    End For
5: End For
```

Como pudimos ver, chequear la consistencia del conjunto, por álgebra de complejidad, es de orden $O(n^3)$. Luego tenemos que por cada rama del árbol de decisión tenemos $O(n^3)$, es decir que nuestro algoritmo de backtracking tiene en peor caso orden de $O(2^n n^3)$

3.1 Experimentación

Para la experimentación se generó un set de datos al azar, variando la cantidad de agentes y sus votos. Por cada agente fijo se generaron 5 *sets* de votos y luego se tomó el caso promedio para la disminución de los *outliers*. Veamos ahora como se comporto el algoritmo para ante distintos casos. Para esto se generaron distintos casos de prueba variando la cantidad de agentes y sus votos. Como el tiempo de ejecución crece exponencialmente con la cantidad de agentes, solo se experimentó con hasta 22 agentes ya que el algoritmo se tomaba mucho tiempo para terminar.
Miremos el siguiente gráfico:

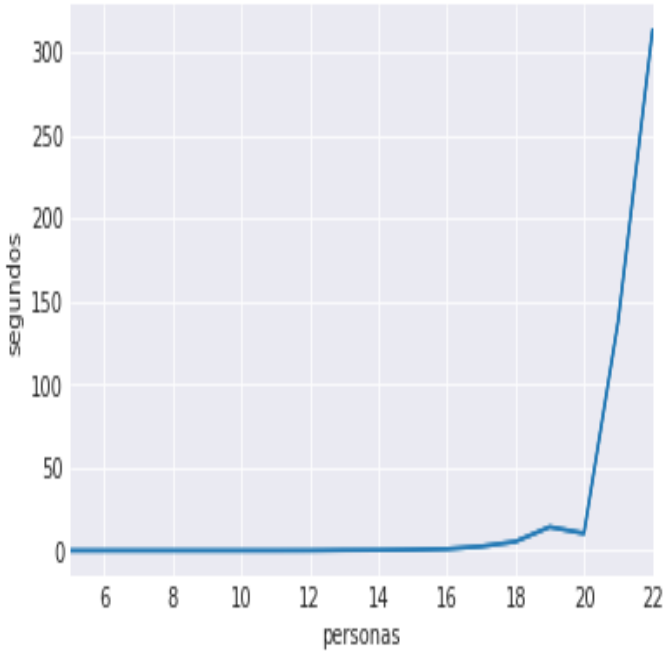

Figura 3: Tiempo de ejecución.

Como podemos ver en el gráfico de arriba, el tiempo de ejecución después de cierto punto se va a la crece considerablemente. Notemos que el crecimiento de los agentes siempre fue el mismo y que para el final la diferencia entre los dos últimos tamaño es incomparable con la diferencia entre los anteriores. Esto esta dado claramente por la naturaleza exponencial del algoritmo.

En el siguiente gráfico veremos los resultados anteriores, pero representados en escala logarítmica ya que para los primero valores, no podemos apreciar bien la naturaleza del algoritmo:

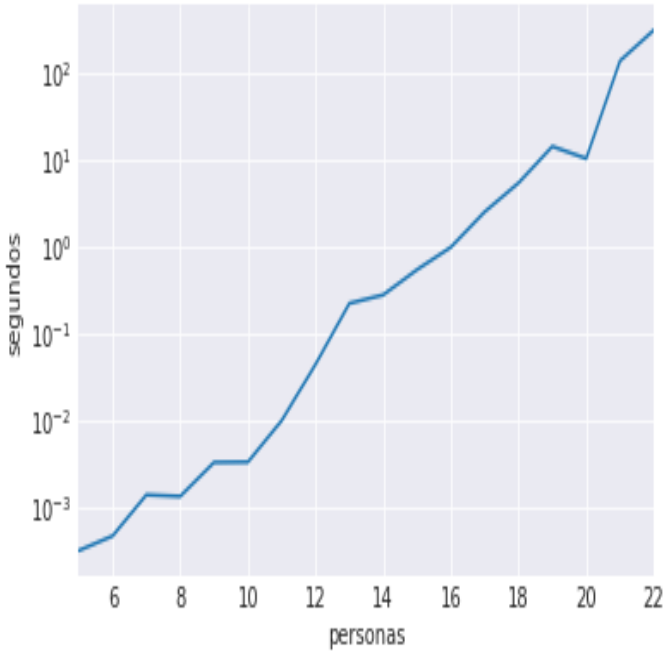

Figura 4: Tiempos de ejecución representados con escala logarítmicas.

Notemos que al mostrar logaritmicamente el crecimiento exponencial del algorítmico, en el gráfico queda una tendencia lineal.

4 Podas

Ya realizada la implementación del algoritmo veremos ahora como podemos mejorar los tiempos de ejecución implementando dos podas distintas.

4.1 Poda a futuro

Primero se decidió mejorar la función de consistencia, teniendo en cuenta que, al agregar un nuevo elemento este podría ser inconsistente con lo votos de los agentes del conjunto de confiables si tuviéramos en cuenta los votos a los elementos de conjunto de agentes restantes a recorrer. Con esta idea en mente agregamos apilamos el algoritmo de consistencia para *chequear* las inconsistencias que se generarían a futuro cuando agregásemos estos agentes problemáticos.
En el siguiente *pseudo-código* planteamos la idea anterior:

```
1: For i in Confiables                    // O(n^2)
2:   For j in Restantes                   // O(n)
3:     IF (votoPositivo(nuevo,j) & votoNegativo(i,j)) || (votoPositivo(i,j) & votoNegativo(nuevo,j)) // O(1)
4:       Return false
5:     textbfFI
6:   End For
7: End For
```

Notemos que la complejidad de la consistencia no cambia, ya que la nueva verificación aporta $O(n^2)$ y por álgebra de limites la complejidad de la función sigue siendo $O(n^3)$

Veamos ahora como se comporta el algoritmo con la nueva poda en comparación con el algoritmo base con el mismo set de datos de los gráficos anteriores, con en el siguiente gráfico:

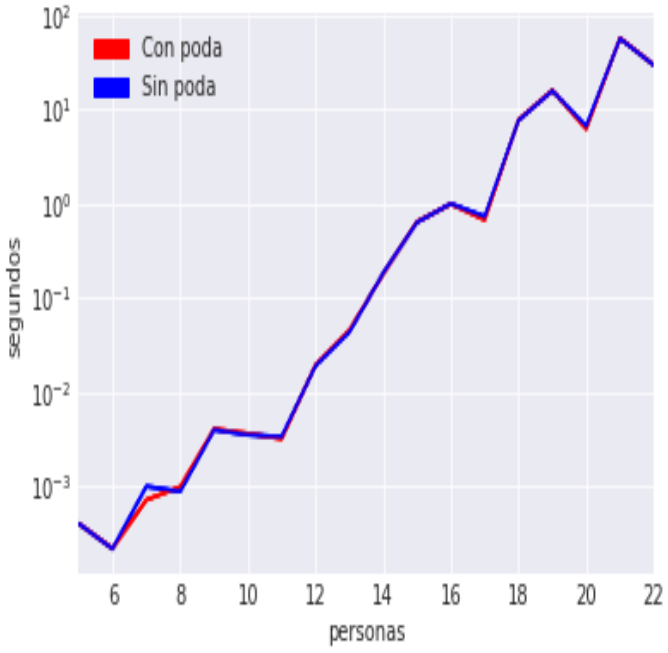

Figura 5: Tiempos de ejecución representados con escala logarítmicas.

Es claro que, dado el gráfico anterior, los tiempos de ejecución no cambiaron prácticamente nada. Para algunos casos podemos ver que la poda funcionó mejor, pero igualmente la diferencia es despreciable. Esto se debe dar en consecuencia a la nuevas operaciones que se realizan en la función de consistencia. El nuevo algoritmo disminuye en llamadas recursivas pero aumenta la cantidad de operaciones por *chequeo* de consistencia y termina dando prácticamente el mismo tiempo de ejecución.

4.2 Poda de optimización

Veamos ahora el segundo método para recortar ramas de nuestro árbol de decisión. La idea principal de esta nueva poda reducir los caminos de búsqueda si ya no es posible que superen a una solución ya que calculada. Imaginemos que tenemos un árbol de decisión dado por 5 agentes y además ya hemos encontrado un posible solución donde todos los agente pertenecen al conjunto; No tendría sentido seguir recorriendo mas ramas porque ninguna podría superar el tamaño de solución ya calculada.
Esta es básicamente la idea de la nueva poda y con esto en mente implementamos la mejora. Para implementarla agregamos una variable compartida llamada **Max** donde todas las llamadas recursivas tienen acceso a esta variable y una condición nueva al algoritmo de backtracking, donde antes se verificaba que el conjunto sea consistente ahora además se fija que la cantidad de personas restantes mas el tamaño de la solución siendo calculada supere a la mayor de las soluciones ya calculadas. Luego, cada vez que se llega a una nueva solución, esta debe superar en tamaño a la mayor de las soluciones anteriores, por lo tanto la nueva solución pasa a ser la mayor de las soluciones.

Veamos como quedaría el *pseudo-código* de nuestro nuevo algoritmo de backtracking:

Algorithm 3 Backtracking ← función (Votos , Confiables, Recorridos, Restantes, Max)

1: **IF** Max > **tamaño**(Confiables) + **tamaño**(Restantes) return 0 **FI**
2: **IF Consistente**(Votos, Confiables, Recorridos, Restantes) return 0 **FI**
3: **IF tamaño**(Restantes) = 0 **THEN**
4: Max ← **tamaño**(Confiables)
5: return **tamaño**(Confiables) **FI**
6: **ELSE**
7: Nuevo ← **dameUno**(Restantes)
8: Restantes ← Restantes **-** Nuevo
9: Recorridos ← Recorridos ∪ **{Nuevo}**
10: return **MAX**(**Backtracking**(Votos, confiables, recorridos, restantes),**Backtracking**(Votos, confiables ∪ **{Nuevo}**, recorridos, restantes))

Notemos que solo aporta muy pocas operación de tiempo constante, teniendo en cuenta que la estructura de datos utilizada para representar los conjuntos devuelve el tamaño en $O(1)$. Veamos

como se comporta este nuevo algoritmo contra el algoritmo base, contrastado con el *set* de datos que venimos usando desde el principio. Utilizaremos el gráfico sin el escala logarítmica para que pueda apreciarse la gran diferencia en tiempo de ejecución de los dos algoritmo.

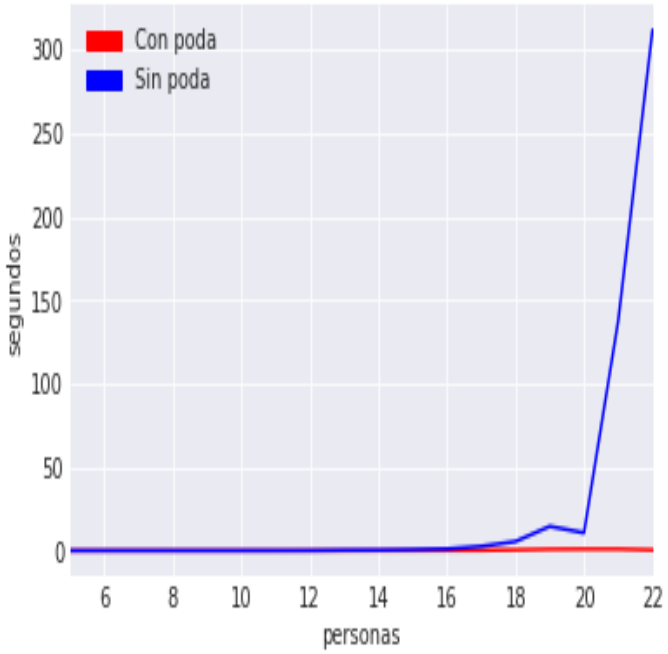

Figura 6: Tiempos de ejecución.

Es clara la diferencia abismal entre los dos algoritmos. Aunque en pero caso siguen teniendo la misma complejidad, este nuevo algoritmo, en caso promedio es altamente superior al algoritmo base. Podemos apreciar que mucha parte del tiempo de ejecución del algoritmo base se pierde recorriendo ramas que ya podrían ser descartas porque no aportarían una mejor solución que la previamente calculada.
Veamos como se comportan representados con escala logarítmica:

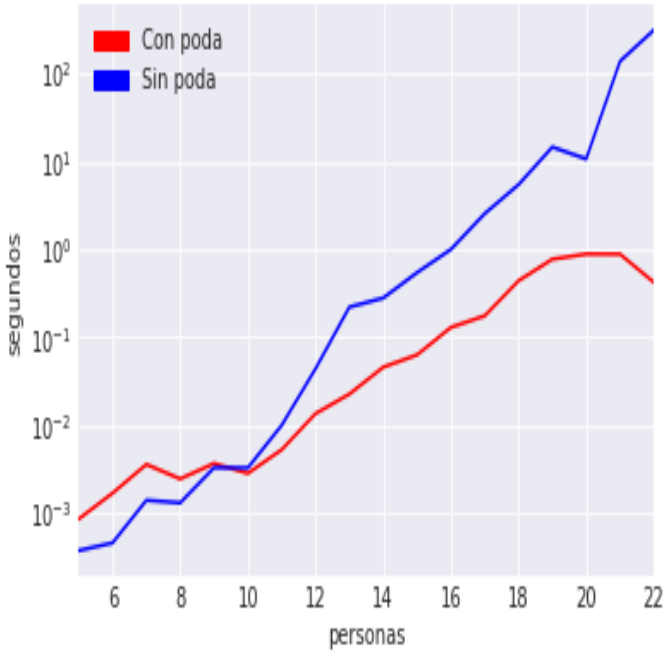

Figura 7: Tiempos de ejecución en escala logarítmica.

La misma diferencia se puede apreciar en este gráfico, podemos ver que en valores chicos los dos algoritmos se comportan relativamente parecido, pero a medida que crece la cantidad de agentes ya empieza a notarse la gran diferencia entre los dos.

5 Conclusiones

Con los resultados obtenidos en este trabajo concluiremos en dos cosas; En primer lugar diremos que la "poda a futuro" demostró ser un mal método para cortar las ramas del árbol de decisión, y en segundo que la poda de optimización por tamaño de una solución obtenida resultó ser una gran mejora para nuestro algoritmo de *backtracking*.
Además, aunque la cantidad de votos de los agentes no influía en el peor caso del algoritmo, es claro que los distintos tipos de combinaciones de los votos generaban que el algoritmo mejorado con las podas, terminase mucho mas rápido que el algoritmo base.