áficos3.27Análisis de complejidadessubsection.3.2

Universidad de Buenos Aires

Facultad de Ingeniería

Teoría de Algoritmos I - 75.29 / 95.06

Gale - Shapley

Alumnos:
Fullone, Josefina - (98374)
Luraschi, Sebastian - (97177)
Salvador, Yago - (99725)
Gonzalez, Andres - (95694)

15 de Abril de 2019

Contents

1 Parte 1 - El Club "PICA-PICA"

1.1 Enunciado

El Club "PICA-PICA" organizará su torneo anual de Truco en parejas. En este torneo presentarán una modalidad de armado de los equipos novedoso. En una primera ronda se separan a los participantes en 2 grupos según su "ranking" (por un lado la mitad mejor puntuada y por el otro el resto). Luego entregan a cada persona un listado ordenado alfabéticamente con los inscriptos del grupo al que no pertenece el mismo. Deben indicar su preferencia con un valor entre 1 y 20 según su interés de formar pareja con el mismo.

1. Proponer una variante de Gale-shapley con resolución de empates para el problema. Dar 2 criterios de desempate posibles.
2. Demostrar que su algoritmo es "justo" (matching estable) y que siempre funciona bien.
3. Indicar la complejidad algorítmica de su solución. Explique cómo llega a la misma.
4. En caso de empate, se puede proponer un desempate mediante un tiro de moneda? Desarrolle.
5. Desarrolle un algoritmo que dado un pareo y las preferencias determina si el mismo es matching estable. ¿Qué complejidad algorítmica tiene?
6. Programe ambos algoritmos.
7. ¿Tiene su programa la misma complejidad algorítmica que la teórica? Justifique.

1.2 Variante de Gale-Shapley

El algoritmo implementado presenta como variante la técnica de desempates de prioridades, una vez armadas las estructuras de datos sin empates, se utiliza un Gale-Shapley común y corriente, este consiste en tomar cada jugador de la lista y ver si el mismo tiene o no una pareja. Mientras el jugador no la tenga se recorre la lista de preferencias(previamente ordenada) y se verifica el estado de la potencial pareja, si se encuentra libre se emparejan mutuamente, en caso de que ya posea una pareja previa se revisa a cual prefiere para definir un emparejamiento.

En caso de empate el criterio para resolver el problema y generar el desempate depende del orden en que el programa lea los datos. Supongamos que el programa lee primero "1, Carlos las Viejas" y luego "1, Analia Flor" entonces en el vector de pares se colocaran esos pares así como fueron leídos y luego, haciendo make-heap el programa ordena el vector comparando los índices de preferencia (para esto utiliza el operador $<$). Entonces tenemos $p1$ = Carlos las viejas y $p2$ = Analia Flor. El programa, al hacer el make-heap hace $p1 < p2$ haciendo que Carlos las Viejas suba de posición. Al contrario, si el programa leyera primero "1, Analia Flor" y luego "1, Carlos las Viejas" entonces seria Analia Flor el jugador que subiría de lugar en el heap.

1.3 Algoritmo justo

Para analizar si el algoritmo implementado asegura la estabilidad vamos a suponer que existe una pareja que desea cambiar de compañero. Supongamos que tenemos dos parejas $(j1, p1)$ y $(j2, p2)$, tales que j1 prefiere p2 antes que a p1, y p2 prefiere j1 antes que a j2. En la ejecución del algoritmo que produjo las parejas, la última propuesta de j1 fue, por definición, p1. Sin embargo no sabemos si j1 le propuso a p2 ser pareja en alguna ejecución anterior. Veamos las dos opciones:

Si no lo hizo, entonces p1 debe aparecer más arriba en la lista de preferencias de j1 que en la lista de p2, lo que contradice nuestra suposición de que j1 prefiere p2 antes que a p1.

Si lo hizo, entonces fue rechazado por p2 a favor de algún otro jugador j3, a quien p2 prefería antes que a j1.

Finalmente j2 se establece como pareja final de p2, entonces o j3 es igual a j2 o, p2 prefiere su pareja final j2 antes que j3. En cualquiera de los 2 escenarios llegamos a que nuestra suposición de que p2 prefiere a j1 antes que a j2 se contradice. De aquí se deduce que el conjunto de parejas establecidas esta en una coincidencia estable.

1.4 Complejidad algorítmica

Un punto critico de la complejidad total del programa pasa por el propio algoritmo Gale Shapley, parte del código del mismo se presenta a continuación y se analiza pasado el mismo:

matcher.cpp

```
void matcher::gale_shapley(){
    for (auto it = my_tournament.getProposers().begin(); ; ++it){
        if (my_tournament.allMatched()){
            return;
        }
        else if (it == my_tournament.getProposers().end())
            it = my_tournament.getProposers().begin();
        if ((*it)->isFree())
            my_tournament.findPartner((*it));
    }
}
```

tournament.cpp

```
bool tournament::allMatched() {
        for (const auto &each : proposers){
                if (!each->getPartner())
                        return false;
        }
return true;
```

```
void tournament::findPartner(player* aPlayer) {
        for (auto p : aPlayer->getPlayer_preferences()) {
                player* candidate = players[p.second];
                size_t my_pref = candidate->getPriorityOf(aPlayer);
                if (candidate->isFree()) {
                         aPlayer->partnerUp(candidate, p.first, my_pref);
                        return;
                } else {
                        if (candidate->prefers(my_pref)) {
                                candidate->losePartner();
                                aPlayer->partnerUp(candidate, p.first, my_pref);
                                return;
                        }
                }
        }
}
```

Partiendo de lo más granular, se analizará primero las funciones de la clase `tournament` comenzando por `findPartner(player* aPlayer)`. Ambos métodos `partnerUp()` y `losePartner()` para asignar y desasig-

nar parejas respectivamente son O(1). Dado que se almacena el valor de preferencia de la pareja actual, el chequeo de ver si un candidato desea cambiar de pareja, el método `prefers(size_t pref)`, también es O(1), a coste de empeorar la complejidad espacial. Adicionalmente, ver si un candidato esta libre a través del método `isFree()` tambiés es O(1) ya que solo se verifica si su variable `partner` es nula. De esta manera, de la linea 5 en adelante de la función bajo análisis es O(1). En la linea 4, el llamado a `getPriorityOf(player* aPlayer)` implica recorrer, en un peor caso, todo el vector de preferencias, lo que se realiza en O(n). Por otro lado, el acceso a la estructura `players` de la linea 3 es un acceso a un mapa, por lo que es O(log n). Asintoticamente prevalece la operación O(n). De esta forma, debido al ciclo for incial que es también O(n), el método `findPartner(player* aPlayer)` posee una complejidad algorítmica O(n^2).

Continuando con el método `allMatched()`, se recorre el vector con la mitad mejor rankeada de jugadores, por lo que es O(n/2), que asintoticamente equivale a O(n).

Por último, el método general `gale_shapley()`, posee un ciclo inicial en la mitad mejor rankeada de jugadores, luego la verificación de si termino en la línea 3, con complejidad O(n) y, de no ser así, le busca pareja a los jugadores que sea necesario, con complejidad O(n^2). De esta forma, el algoritmo final queda O(n^3).

Se midió el tiempo de ejecución del algoritmo (sin tener en cuenta el armado de las estructuras) utilizando la librería `<chrono>` aumentando el tamaño de la entrada, los resultados se pueden observar en la figura 1. Para generar rapidamente gran cantidad de archivos de entrada (como 5000 jugadores, con 5000 archivos de preferencias) se implementó un scrypt en Matlab acorde a los formatos necesarios que contempla la posibilidad del empate, preferencias entre 1 y 20 y la separación de jugadores en mitades mejores y peores rankeadas. Si bien el armado de estructuras debería haber sido considerado, en términos de complejidad estos no modifican la totalidad del algoritmo. El leer una lista de oferentes a un, vector O(n/2) = O(n) y luego crear un mapa es O(nlogn). A su vez por cada persona en el mapa se crea una lista de preferencias donde cada persona tiene una lista O(n), pero esta lista debe ser ordenada para poder recorrerla luego, es decir O(nlogn). Entonces la precarga de datos termina siendo O($(nlogn)^2$), lo cual no cambia la complejidad total.

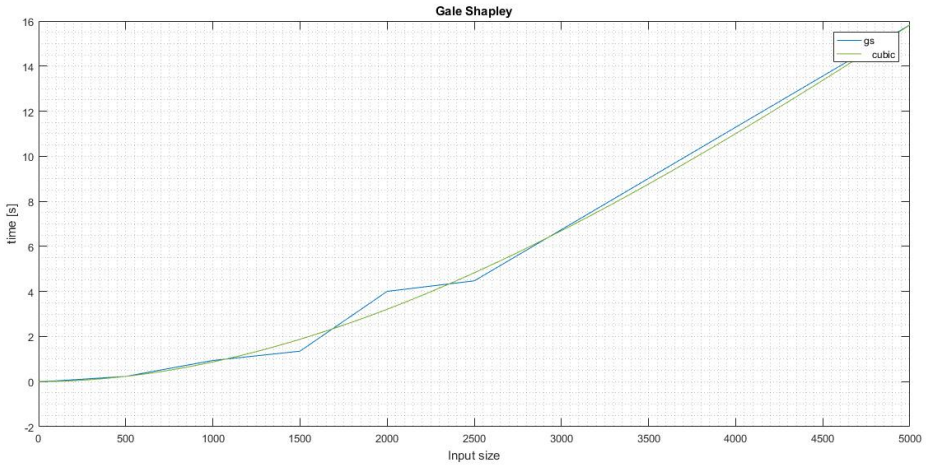

Figure 1: Análisis de complejidad de Gale ' Shapley.

1.5 Desempate mediante tiro de moneda

En caso de empate entre dos parejas es posibles realizar el desempate tirando una moneda y asignado cada jugador a cada cara de la misma. El algoritmo de Gale Shapley es un algoritmos justo que asegura la estabilidad de las parejas, por eso, cuando se llega al caso de empate es porque no hay otra mejor pareja disponible para el jugador en cuestión. Llegado el punto en donde es lo mismo asignar una pareja o la otra cualquier criterio de desempate es valido para definir la pareja final.

1.6 Matching stable

Para determinar si un matcheo es estable, se desarrollo la función mostrada a continuación:

matcher.cpp

```
bool matcher::is_stable(){
    size_t i = 0;
    player * player_aux;

    for(auto each : my_tournament.getProposers()){//O(n)
        i = each->getPartnerPref();//O(n)
        for(auto it = each->getPlayer_preferences().begin(); i; i--){//O(n)
            player_aux = my_tournament.getPlayer(it->second);
            if(player_aux->getPriorityOf(each) <
                    player_aux->getPartnerPref())//O(n)
                return false;
        }
    }
    return true;
}//O(n)
```

Al igual que la solución encontrada al problema original y utilizando las mismas estructuras, se llega a un algoritmo de complejidad O(n^3). Esto se debe a, los 2 ciclos for anidados y la operación `getPriorityOf(player* Player)` dentro del último ciclo, cuya complejidad mencionada previamente es O(n). Multiplicado por la influencia de ambos ciclos se llega a O(n^3).

1.7 Algoritmos implementados

El algoritmo implementado para resolver el problema se muestra en la sección 3.1

Yo volaria esta subseccion al joraca

1.8 Complejidad teórica vs algorítmica

Debido a una elección deficiente de las estructuras, no se logro alcanzar la complejidad teórica óptima. Según lo estudiado, el algoritmo puede aplicarse con una complejidad O(n^2), donde el núcleo de la solución del problema puede reducirse a O(n logn). No es el caso del armado de las estructuras necesarias, motivo por le cual el algoritmo total se define del orden de O(n^2).

En nuestro caso el no elegir de manera adecuada las estructuras utilizadas provocó que el núcleo del algoritmo ya se encuentre por encima de esta complejidad, como se analizó en la sección 1.4.

2 Parte 2 - Funciones matemáticas / estadísticas

2.1 Enunciado

Dados un conjunto de n números, las siguientes funciones matemáticas / estadísticas: Máximo, Media, Moda, Mediana, Desviación estándar, Permutaciones del conjunto, Variaciones del conjunto tomados de r elementos (r≪n) y Variaciones con repetición del conjunto de r elementos (r≪n).

Y los diferentes escenarios propuestos:

1. Los valores están cargados en un vector
2. Los valores están cargados en una lista
3. Los valores están ordenados en un vector de mayor a menor
4. Los valores están precargados en un estructura sugerida por el grupo.

Resuelva:

1. Proponga algoritmos para cada una de las resoluciones
2. Analice la complejidad algorítmica (tiempo y espacio) de cada caso teniendo en cuenta el mejor, peor y caso promedio.
3. Compare entre cada función su complejidad gráficamente.
4. Programe los algoritmos

3 Algoritmos implementados

3.1 Algoritmos propuestos

Si bien los algoritmos pedidos deben tratarse en estructuras distintas, la forma en que se llevaron a cabo fueron muy similar para todos los casos, variando, por ejemplo en el caso de la lista, el tiempo en acceder a los elementos dado que su acceso es secuencial y no aleatorio. Para la obtención del máximo se optó por una búsqueda lineal. Si la estructura estuviera ordenada o fuera un heap, esa búsqueda se reduce a tomar el primer elemento simplemente. Análogamente tomamos recorridos lineales sumando elemento a elemento y realizando los respectivos cálculos en el caso de la media y la desviación. En el caso de la moda y la mediana, se comienza ordenando el arreglo, lista o heap usando la función del standard de c++ sort, esto se realiza siempre y cuando la estructura no esté previamente ordenado, para luego iterar sobre sus elementos y obtener lo que cada algoritmo pide. Luego, para los algoritmos de variaciones con repetición y permutaciones, en primer lugar debemos ordenar el arreglo para poder utilizar la función del std next permutation, que se encarga de hacer los respectivos swaps en la estructura para conseguir las permutaciones. La diferencia entre ambos algoritmos reside en que permutaciones se ejecuta únicamente para el tamaño del arreglo, mientras que la segunda ejecuta permutaciones para todo n valor menor al tamaño del arreglo. Por último, para variaciones se propone conseguir los conjuntos de r elementos usando recursividad. La idea es dejar un prefijo constante y generar los conjuntos que tienen a ese prefijo, luego de imprimir todos, reemplazar el último elemento con alguno de los restantes.

3.2 Análisis de complejidades

Tomando en cuenta lo expuesto en la sección anterior, se puede expresar ahora las complejidades de cada algoritmo. Se toma T(n) como el tiempo de la función. En el caso de la obtención del máximo es más que evidente que el peor caso sera lineal, es decir, $T(n) = O(n)$. En el mejor de los casos, o sea, cuando el acceso es inmediato al máximo sea porque se usa un heap o porque la estructura está ordenada, $T(n) = O(1)$. El caso promedio es, o bien el peor caso, o el mejor caso dado que no se puede saltear el recorrer todo el arreglo, a menos que se encuentre ordenado. Espacialmente su complejidad es $O(1)$ dado que no requiere de otras estructuras para su funcionamiento, hay pocas variables internas y no hay llamados a otras funciones que puedan modificar esto. Al obtener la media y la desviación de la estructura sucede lo mismo que en el caso del máximo, se recorre todo el arreglo para obtener la media. Coincide así el peor caso, no así el mejor que en este caso es idéntico al peor, entonces la función se ejecuta siempre en tiempo $O(n)$. Espacialmente es $O(1)$ por los mismos motivos. Luego, las funciones moda y mediana son dependientes de la función sort, siempre y cuando no esté ordenado el arreglo. Esta, según el standard de C++, es $O(nlogn)$ en todo caso. Si bien dentro de ambas funciones se itera a lo largo del arreglo luego de ordenar, esta no se encuentra anidada,

por lo que la complejidad total del algoritmo es T(n) = O(n) + O(nlogn) que es equivalente a T(n) = O(nlogn). Entonces, el peor caso, mejor y promedio son equivalentes si el arreglo no se encuentra ordenado. Por otro lado, si estuviera ordenado, no haría falta ejecutar el ordenamiento, quedando solo el recorrido del mismo. La moda en este último caso es para cualquier estructura O(n), mientras que la mediana depende de la estructura utilizada. En el caso del heap, como está implementado a través del standard de c++, este funciona como un arreglo, es decir que tenemos un acceso secuencial a la mediana, quedando su complejidad en O(1). En cambio, para la lista su acceso es secuencial, por lo tanto habrá que iterar por sus elementos hasta llegar a la mediana, quedando una complejidad de O(n). Sus complejidades espaciales son O(1) para la mediana y O(n) como peor caso en la moda donde todos los elementos se repiten igual cantidad de veces a excepción de uno y O(1) para el caso en el que un elemento es el más repetido. Con las funciones de variación de conjuntos, variación de conjuntos repetidos y permutaciones se aprecian situaciones similares. En las dos últimas se va iterando por cada permutación, donde obtener cada permutación es O(n!) y se copian elemento a elemento en el vector de salida. La diferencia principal entre ellas es que las variaciones con repetición itera O(n*n) veces dado que después de copiar toda la permutación, copia a su vez cada set restando de a un elemento. Por lo tanto, permutaciones corre en O(n*n!) en todo caso y variaciones con repetición en O(n*n*n!). Por último, para las variaciones, su complejidad es O(n!). Al tratarse de una función recursiva, hay que imponer un caso de corte, este es cuando el índice es igual al tamaño del vector de entrada. Luego, como se itera por todos los elementos hasta el último y dentro de esta iteración se ejecuta la llamada recursiva aumentando el índice, se logran iteraciones anidadas del estilo O(n*(n-1)...(1)) = O(n!). Espacialmente, el mejor caso sería tener 1 elemento donde la salida será O(1), con más elementos, la salida aumenta y al utilizar una estructura donde guardar los sets dentro de las funciones, su complejidad espacial será O(n!) para las permutaciones dado que son todas las posibles y $O(2^n)$ para las variaciones de conjunto.

3.3 Gráficos