USC
UNIVERSIDADE DE SANTIAGO DE COMPOSTELA
FACULTADE DE MATEMÁTICAS

Traballo Fin de Grao

Emparellamentos Estables: Algoritmo de Gale-Shapley

Uxío García Andrade

Xullo, 2022

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

GRAO DE MATEMÁTICAS

Traballo Fin de Grao

Emparellamentos Estables: Algoritmo de Gale-Shapley

Uxío García Andrade

Xullo, 2022

UNIVERSIDADE DE SANTIAGO DE COMPOSTELA

Traballo proposto

Área de Coñecemento: Estatística e Investigación Operativa
Título: Emparellamentos Estables: Algoritmo de Gale-Shapley
Breve descrición do contido
O obxecto de estudo deste TFG será o problema de atopar un emparellamento estable entre dous conxuntos de elementos de igual tamaño, onde cada elemento dun conxunto ten unha orde de preferencias sobre os elementos do outro conxunto. Un emparellamento é simplemente unha bixección entre os elementos de ambos conxuntos. Un emparellamento A é estable se non existe ningunha parella (α, β), distinta ás propostas polo emparellamento, coa propiedade de que tanto α como β se prefiren mutuamente fronte ás súas respectivas parellas baixo o emparellamento A. O principal obxectivo deste traballo será introducir o algoritmo de Gale e Shapley, introducido formalmente en 1962, e estudar as súas propiedades matemáticas. O problema do emparellamento estable recibiu unha gran atención da comunidade científica polas súas múltiples aplicacións en distintos ámbitos: asignación de estudantes a colexios, de médicos a hospitais,...
Recomendacións
Non hai ningunha recomendación en particular.
Outras observacións
Non hai ningunha observación.

Índice

Resumo

A primeira parte deste traballo centrarase en introducir e desenvolver as propiedades matemáticas do algoritmo de Gale-Shapley. Este algoritmo pretende dar solución ao problema de atopar emparellamentos estables entre dous conxuntos diferenciados, nos que cada elemento dun conxunto ten unha orde de preferencias sobre o outro conxunto. Este algoritmo, a pesar do seu carácter abstracto e teórico, introducirase a través dun caso particular, no que os emparellamentos son individuais, e que soe estudarse no contexto dos matrimonios entre individuos. A partir de aí, revisarase o caso xeral e compararanse as distintas propiedades que presentan. Para verificar o seu correcto funcionamento, este traballo tamén inclúe un estudo numérico, realizado a partir da implementación do código do algoritmo de Gale-Shapley.

Por outra banda, tamén se incluíu unha das aplicacións de Gale-Shapley mais recentes, as subastas de anuncios en Internet. Despois dunha breve introdución á teoría de subastas, describirase o algoritmo dun mecanismo xeral de subastas e as súas propiedades. De novo, tamén se implementou este algoritmo en código e incluíuse un estudo numérico do mesmo.

Abstract

The first part of this work will focus on introducing and developing the mathematical properties of the Gale-Shapley algorithm. This algorithm aims to provide a solution to the problem of finding stable matches between two distinct sets, where each element of one set has a preference order over the elements of the other set. This algorithm, despite its abstract and theoretical nature, will be introduced through a particular case, in which the matches are one-to-one, and which is usually studied in the context of marriages between people. From there, the general case will be reviewed and the different properties they present will be compared. In order to verify its correct functioning, this work also includes a numerical study, with an implementation in code of the Gale-Shapley algorithm.

On the other hand, one of the most recent Gale-Shapley applications, Internet ad auctions, was also included. After a brief introduction to auction theory, the algorithm of a general auction mechanism and its properties will be described. Again, this algorithm was also implemented in code and a numerical study of the algorithm was performed and included.

Introdución

Ao longo do século XX, a teoría de emparellamentos estables converteuse nunha das áreas de maior interese no eido da economía. Tras os resultados obtidos por Monge e Kantorovich entre finais do século XVIII e principios do XIX, considérase que o traballo que disparou o interese na área foi o producido por Gale e Shapley en 1962, o coñecido como algoritmo de Gale-Shapley, no que se baseará esta memoria.

Inicialmente, o problema ao que pretendían dar solución inicialmente Gale e Shapley era o da admisións de estudantes a universidades, pero a literatura posterior e os traballos relacionados acabarían outorgando maior relevancia ao caso particular de emparellamentos un-a-un, o coñecido como problema do matrimonio estable, no que existen un número igual de homes e mulleres, cada un con preferencias sobre cada membro da outra parte, a partir das cales se debe tratar de conseguir un emparellamento que satisfaga a todos os individuos. En ambos os dous casos, Gale e Shapley demostraron a existencia dun equilibro chamado emparellamento estable, ademais de como é posible obtelo a partir do seu algoritmo. Por outra parte, tamén chegaron a resultados sobre a optimalidade destes resultados para as distintas partes involucradas, ademais de discutir como afecta ao resultado final que algún individuo decida mentir á hora de aportar a súa orde de preferencias. Ademais da ineludible relevancia dos resultados obtidos, foi a claridade, elegancia e simplicidade do artigo o principal motivo da súa difusión entre as principais revistas de economía e matemáticas do momento.

A partir destes resultados que, a pesar de ser aparentemente prácticos e aplicados, son en realidade abstractos, sen chegar a contar cunha aplicación directa. Non foi ata pasado o 1980 que Alvin Roth comezou a implementar esta teoría a situacións reais. En particular, comezou tras a súa contratación por parte dunha institución estadounidense para que resolvese o problema do emparellamento de estudantes de medicina recién graduados con hospitais que buscaban facerse cos seus servizos. Esta non sería a única aplicación que Roth atoparía, senon que, xa entrado o século XXI, recorreu a el para resolver o problema do emparellamento de estudantes e institutos na cidade de New York. Este último tivo un particular éxito, posto que reduciu ata nun 90 % o número de estudantes emparellados con institutos polos que non expresarán ningún tipo de

preferencia.

Todas estas contribucións levaron a que Lloyd Shapley e Alvin Roth fosen galardonados co premio nobel de economía en 2012, recoñecendo así todas as súas aportacións aos emparellamentos estables, un desenvolvendo a teoría abstracta e o outro implementándoa en situacións do mundo real, coadxuvando no progreso da teoría de mercados nos que o diñeiro non é un factor.

A partir de aí, as ramas que aproveitaron o algoritmo de Gale-Shapley e outras contribucións foron moi diversas, sendo unha das aplicacións mais directas a da teoría de subastas. Esta última é unha rama da economía aplicada, na que, como o seu propio nome indica, se estuda o comportamento dos postores en mercados de subastas, centrándose en como as distintas características dunha subasta incentivan certo resultado. Nun primeiro instante, un pode pensar que os emparellamentos estables non están directamente relacionados coas subastas, xa que, como ben se indicou previamente, estes centrábanse en mercados nos que o prezo non influía. Por outra parte, unha subasta non deixa de ser un mercado con dúas partes diferenciadas, na que un ou varios postores emparéllanse cun ou varios obxectos. Así mesmo, existe un amplo número de tipos de subastas, polo que haberá algúns para os que a teoría de emparellamentos estables será mais sinxela de aplicar que para outros.

Deste xeito, ao longo dos capítulos finais desta memoria discutirase un mecanismo xeral de subastas que, apoiándose na teoría de emparellamentos estables e o algoritmo de Gale-Shapley, permitirá producir un algoritmo que produza emparellamentos estables e óptimos para subastas de anuncios en internet. Este mecanismo, cuxa teoría foi desenvolvida inicialmente por Google, converteuse nunha das bases de subastas de ocos en motores de búsqueda, converténdose así nunha das aplicacións con mais repercusión de Gale-Shapley, xa que, dun xeito ou outro, a meirande da poboación foi partícipe deste tipo de subastas ao realizar algún tipo de búsqueda en Google.

Ademais de cubrir o desenvolvemento teórico e matemático dos distintos conceptos mencionados ao longo desta introdución, a memoria tamén incluirá estudos numéricos, implementados en distintas linguaxes de programación, que verificarán o correcto funcionamento dos algoritmos, así como permitirán sacar distintas conclusións acerca dos mesmos.

Capítulo 1

Algoritmos de Emparellamento

En primeiro lugar, comezaranse revisitando algunhas das definicións vistas na asignatura de *Programación Linear e Enteira*, impartida durante o Grao, e que serán precisas para o desenvolvemento da memoria. Este repaso só engloba ás 3 primeiras definicións, polo que o contido restante da memoria requeriu do estudo das distintas fontes presentes na bibliografía.

Definición 1.1. Un grafo G é un par ordenado $G = (N, M)$, sendo N un conxunto de nodos e M un conxunto de arestas que relacionan os nodos.

Definición 1.2. Un grafo dise dirixido se as arestas son pares ordenados, é dicir, $M \in N \times N$

Definición 1.3. Dado un grafo dirixido $G = (N, M)$, dise que é **bipartito** se:

- Existen conxuntos N_1 e N_2 tales que $N_1 \cup N_2 = N$ e $N_1 \cap N_2 = \emptyset$.
- $M \subseteq N_1 \times N_2$.

Definición 1.4. Dado un grafo $G = (N, M)$, un sistema de preferencias para G será unha familia $\{\succ_v\}_{v \in V(G)}$ tal que cada $>_v$ é unha ordenación lineal de $M(v)$. Se $u, u^{'} \in N(v)$ e $u \succ_v u'$, dicimos que v perfire a u antes que $u^{'}$.

Notación 1.5. Usarase a notación $O_{u,n}$ para referirse ao elemento que ocupa a posición n na lista de preferencias de u. Deste xeito, $O_{u,1} = v$ quererá dicir que v é o elemento que u prefire antes que calquera dos demais.

Notación 1.6. Pola contra, a notación $I_{u,v}$ indicará a posición que ocupa o elemento v na orde de preferencias de u. Deste xeito, se $O_{u,1} = v$, teremos que $I_{u,v} = 1$.

Notación 1.7. Para indicar a orde de preferencia dun elemento u con respecto a certos elementos $v_1, v_2, ..., v_n$, usarase a notación $\succ_u := v_1, v_2, ..., v_n$, sendo $O_{u,1} = v_1$ e $O_{u,n} = v_n$.

Definición 1.8 (Emparellamento)**.** Unha **asignación** ou **emparellamento** A é un conxunto de pares ordenados (n, m), onde $n \in N$ e $m \in M$, de tal xeito que cada $n \in N$ aparece como moito nunha parella de A, e cada $m \in M$ aparece como moito nunha parella de A.

Definición 1.9. Unha asignación A entre un conxunto N e M dirase **perfecta** se $|N| = |M| = |A|$.

Definición 1.10. Unha asignación A dirase que é inestable se existen dous elementos $A, B \in N$ que foron asignados a $\alpha, \beta \in M$ respectivamente, a pesar de que $A \succ_\beta B$ e $\beta \succ_A \alpha$, é dicir, β prefire a A antes que a B, e A pefire a β antes que a α.

Unha representación gráfica dunha asignación inestable amósase na figura 1.1.

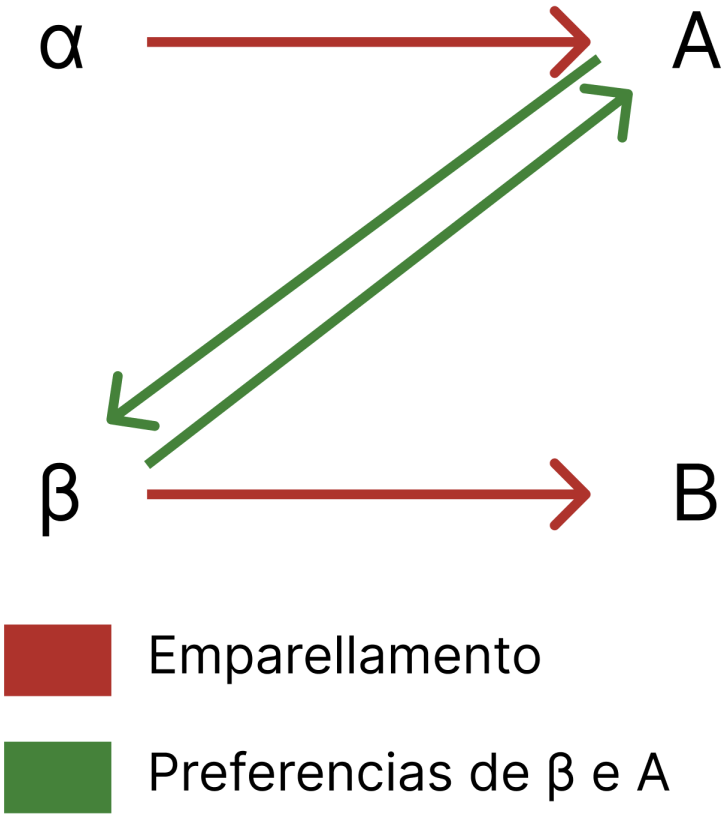

Figura 1.1: Exemplo de asignación inestable.

Definición 1.11. Unha asignación A dirase estable se non é inestable.

Definición 1.12. Estratexia. En teoría de xogos, a estratexia dun xogador é o plan das acción que pode escoller nunha situación na que a saída non depende unicamente das súas accións, senón das accións dos demais xogadores tamén. No referido a emparellamentos estables, a estratexia dun elemento $v \in V(G)$ será a súa orde de preferencia $\{\succ_v\}_v$, polo que, nalgún caso usaranse os dous termos indistintamente.

Definición 1.13. Un algoritmo de emparellamento dirase **a proba de estratexias** ou **non manipulable** se un individuo non ten ningún incentivo para non presentar a súa orde de preferencias reais. De non contar con esta propiedade, un individuo podería conseguir un resultado mais favorable modificando a súa orde de preferencia.

Capítulo 2

Algoritmo de Gale-Shapley

Ao longo deste capítulo describirase a teoría e as propiedades do algoritmo de Gale-Shapley, introducido polos autores homónimos en 1962 [1]. Comenzaremos centrándonos nun caso particular que permite simplificar e introducir o algoritmo dun xeito mais intuitivo, facilitando a posterior descrición do caso xeral.

2.1. Caso Particular: Matrimonio

Antes de introducir o algoritmo que resolve o caso xeral das admisións á universidade, é conveniente estudar un caso particular. Este caso sería o de que o número de estudantes e universidades fose o mesmo, e todas as cuotas fosen unitarias.

Téndese a estudar este caso dentro do seguinte contexto: supoñamos que existe unha comunidade na que temos n homes e n mulleres, onde cada persoa conta cunha lista na que ordea a cada individuo do xénero oposto segundo as súas preferencias. O obxectivo será o de obter unha asignación de xeito que cada persoa poida casar con outra de tal forma que non exista ningunha outra asignación na que existe un casamento que melloraría a satisfacción global.

Para visualizar claramente este problema, introducirase a continuación un exemplo:

Exemplo 2.1. Na figura 2.1 detállase a matriz de preferencias de emparellamentos para tres homes, α, β e γ, e tres mulleres, A, B e C. Nesta matriz, o valor que aparece debaixo da columna de home, será a preferencia que o home que está nesa columna lle asigna á muller á que lle corresponde esa fila. Deste xeito, teriamos, por exemplo, para o home α, a súa orde de preferencias sería $C \succ_\alpha A \succ_\alpha B$.

Coa notación introducida no capítulo exterior, as ordes de preferencias serían as seguintes:

	A		B		C	
	Home	Muller	Home	Muller	Home	Muller
α	2	2	3	1	1	3
β	3	1	1	3	2	2
γ	1	3	2	2	3	1

Figura 2.1: Matriz de preferencias.

Homes:	**Mulleres:**
$\succ_\alpha := C, A, B$	$\succ_A := \beta, \alpha, \gamma$
$\succ_\beta := B, C, A$	$\succ_B := \alpha, \gamma, \beta$
$\succ_\gamma := A, B, C$	$\succ_C := \gamma, \beta, \alpha$

Existen 6 combinacións posibles de matrimonios posibles, dos cales 3 son estables 3 son inestables. Un exemplo de combinación estable sería o caso no que cada home casase coa súa primeira elección, é dicir, a asignación sería a seguinte: $A = (\alpha, C), (\beta, B), (\gamma, A)$. Deste xeito, cada muller obtería o home que está na última posición na súa orde de preferencias, pero seguiría sendo estable, posto que non existe ningunha posibilidade de que os homes estean mais satisfeitos, xa que todos teñen a súa primeira elección. As outras dúas posibilidades de asignacións estables serían asignar a cada muller a súa primeira elección, ou asignar a cada persoa a súa segunda elección (a segunda elección coincide para homes e para mulleres).

Por outra parte, é posible comprobar rapidamente como o resto de asignacións son inestables. Tómese como exemplo a seguinte asignación: $A = (\alpha, C), (\beta, A), (\gamma, B)$. No caso de α e β, temos que $A \succ_\alpha C$ e $C \succ_\beta A$, pero o emparellamento é $(\alpha, C), (\beta, A)$.

Teorema 2.2 (Gale-Shapley)**.** *Sexa $H = h_1, ..., h_n$ un conxunto de n homes, e $M = m_1, ..., m_n$ un conxunto de n mulleres, sempre existe unha asignación estable de casamentos.*

Demostración. A demostración deste teorema realizarase a partir da definición dun procedemento iterativo, cuxo pseudocódigo se pode ver na figura 2.1, coñecido tamén polo nome de algoritmo de aceptación diferida. Neste caso, supoñerase que os homes son os que propoñen matrimonio ás mulleres, pero a inversa sería equivalente e funcionaría do mesmo xeito.

`Iteración 0`:

Antes de comezar a parte iterativa do algoritmo, deberán resolverse as preferencias que non son estrictas. É dicir, se $m_i =_{h_k} m_j$, $1 \leq i, j, k \leq n$, $i \neq j$, será preciso resolver este empate

dalgún xeito. Tipicamente, estes resolveranse ordeando os empates alfabeticamente, pero os tipos de estratexias son moi variados, escolléndose ás veces a resolución de conflitos aleatoria.

`Iteración 1`:

a. Todo home h_i, $i \in 1, ..., n$, propón matrimonio á súa muller preferida, é dicir, a $O_{h_i,1}$.

b. Toda muller m_i, $i \in 1, ..., n$, acepta a proposta do home que mais alto estea na súa orde de preferencia, e rexeita o resto.

`Iteración k`:

a. Todo home h_i, $i \in 1, ..., n$ rexeitado no paso $k-1$ realiza unha nova proposta a $O_{h_i,j}$, $j \in 1, ..., n$, sendo j o menor índice tal que $O_{h_i,j}$ aínda non rexeitase a h_i.

b. Toda muller m_i mantén a súa oferta preferida ata o momento, é dicir, $O_{m_i,j}$, sendo j o menor índice tal que $O_{m_i,j}$ fixo unha proposta a m_i. m_i rexeita o resto de propostas.

`Finalización:` Se nunha iteración non hai ningunha nova proposta, obtense a asignación emparellando a cada muller coa proposta que mantenía.

Seguindo este algoritmo, teremos que que todas as mulleres terán, polo menos, unha proposta, xa que, de non ser así, teremos que haberá polo menos 2 homes, h_i e h_j, $j, i \leq n, j \neq i$ que terán realizado unha proposta á mesma muller m_k, $k \leq n$. Non obstante, como non é posible que se produzan empates na orde de preferencia debido á resolución dos mesmos realizada no paso 0, terase que $h_i \succ_{m_k} h_j$ ou $h_j \succ_{m_k} h_i$, polo que m_k rexeitará unha das propostas e o algoritmo procederá a realizar outra iteración.

Por outra parte, vexamos agora como a asignación producida a partir deste algoritmo será estable. Sexa A esta asignación e sexa un emparellamento $(h_i, m_j) \in A$. Supoñamos que existe outra muller m_k, tal que $m_k \succ_{h_i} m_j$. Deste xeito, o home h_i envioulle unha proposta á muller m_k nunha iteración previa a m_j. Como finalmente non foi emparellado con m_k, isto quere dicir que m_k rexeitou a súa proposta por outro home h_l tal que $h_l \succ_{m_k} h_i$, polo que non se pode dar unha inestabilidade no algoritmo. □

```
1 Inicializar propostas para cada muller m a baleiro
2 do
3     m := cada home h envía unha proposta ámuller m que mais alta estea na lista e aínda non lla enviase
          antes
4     if a muller m ten mais dunha proposta then
5         m := a muller m rexeita todas as propostas menos a do home h, que éo que mais alto está na súa
              lista
6     end if
7 while apareza algunha nova proposta
8 devolver a asignación de cada muller co home que enviase a proposta que segue mantendo
```

Código 2.1: Pseudocódigo do algoritmo de aceptación diferida.

Exemplo 2.3. Vexamos agora como se aplicaría Gale-Shapley ao seguinte exemplo: Sexan 4 homes $H = \{\alpha, \beta, \gamma, \delta\}$ e sexan 4 mulleres $M = \{A, B, C, D\}$, coas seguintes ordes de preferencias:

Homes:	**Mulleres:**
$\succ_\alpha \coloneqq A, B, C, D$	$\succ_A \coloneqq \delta, \gamma, \alpha, \beta$
$\succ_\beta \coloneqq A, D, C, B$	$\succ_B \coloneqq \beta, \delta, \alpha, \gamma$
$\succ_\gamma \coloneqq B, A, C, D$	$\succ_C \coloneqq \delta, \alpha, \beta, \gamma$
$\succ_\delta \coloneqq D, B, C, A$	$\succ_D \coloneqq \gamma, \beta, \alpha, \delta$

Unha execución normal do algoritmo sería o seguinte:

Iteración 0:

As ordes de preferencia son todas estrictas, polo que non será preciso resolver empates.

Iteración 1:

a. A recibe propostas de α e β. B recibe proposta de γ. D recibe proposta de δ.

b. A única muller que recibe varias propostas é A, que rexeita β, xa que $\alpha \succ_A \beta$.

Iteración 2:

a. O único home rexeitado no paso 1 foi β, que lle enviará unha proposta a $O_{\beta,2} = D$.

b. D conta con dúas propostas, rexeita δ, xa que $\beta \succ_D \delta$.

Iteración 3:

a. O único home rexeitado no paso 2 foi δ, que lle enviará unha proposta a $O_{\delta,2} = B$.

b. B conta con dúas propostas, rexeita γ, xa que $\delta \succ_B \gamma$.

Iteración 4:

a. O único home rexeitado no paso 3 foi γ, que lle enviará unha proposta a $O_{\gamma,2} = A$.

b. A conta con dúas propostas, rexeita α, xa que $\gamma \succ_A \alpha$.

Iteración 5:

a. O único home rexeitado no paso 4 foi α, que lle enviará unha proposta a $O_{\alpha,2} =$

B.

b. B conta con dúas propostas, rexeita α, xa que $\delta \succ_B \alpha$.

Iteración 6:

a. O único home rexeitado no paso 5 foi α, que lle enviará unha proposta a $O_{\alpha,3} = C$.

b. Non hai ningunha muller con mais dunha proposta, non se produce ningún rexeitamento.

Finalización:

Na iteración 6 non se produciu ningún rexeitamento, polo que non haberá novas propostas e o algoritmo finaliza.

Polo tanto, a asignación final será $S = (\alpha, C), (\beta, D), (\gamma, A), (\delta, D)$. É facilmente comprobable que esta asignación será estable.

2.1.1. Propiedades de Gale-Shapley para o problema do matrimonio

Nesta subsección presentaranse as principais propiedades matemáticas que presenta o algoritmo de Gale-Shapley. Entre elas, as mais desexables e que fixeron que o algoritmo tivese cabida en distintas aplicacións son a de optimalidade da solución obtida e a de que a estratexia sexa non manipulable.

De novo, as propiedades desexadas neste apartado estarán asociadas aos homes, o grupo que realiza as propostas nesta versión, pero tamén serían válidas no caso de que fosen as mulleres as que realizan as propostas. Non obstante, as distintas propiedades non se verificarán para o grupo que recibe as propostas, podendo, por exemplo, dar lugar a situacións nas que unha muller se beneficie de non aportar a súa orde de preferencia real, como veremos posteriormente.

Teorema 2.4. *O algoritmo de Gale-Shapley remata en, como moito $n^2 - 2n + 2$ iteracións, despois de realizarse $n^2 - n + 1$ propostas.*

Demostración. Por unha banda, temos que o algoritmo parará cando nunha iteración non se realice ningunha proposta. Deste xeito, teremos que, no peor caso, un dos homes h_i terá realizado n propostas, sendo emparellado finalmente coa muller que se atopa no último lugar da súa orde de preferencias, $O_{h_i,n}$. Pola contra, o resto dos $n-1$ homes poderán realizar unha proposta, como moito, $n-1$ veces, xa que, senon, a asignación non sería estable. Deste xeito, o número total de propostas virá dado por:

$$(n-1)(n-1) + n = n^2 - 2n + 1 = n^2 - n + 1.$$

Do mesmo xeito, para o número de iteracións, haberá que ter en conta que sempre será preciso 1 iteración inicial, na que realizan n iteracións. Nas iteracións sucesivas, no peor caso, realizarase só 1 proposta, que será no caso no que só 1 home sexa rexeitado. Deste xeito, teremos que o número máximo de iteracións será o mesmo que o de propostas realizadas, restándolle as propostas da iteración inicial. Deste xeito, o número total de iteracións será:

$$(n^2 - n + 1) - (n - 1) = n^2 - 2n + 2.$$

□

Teorema 2.5. ***Optimalidade****. O algoritmo de Gale-Shapley é óptimo para os homes. Isto é, a asignación estable A producida polo algoritmo é tal que non existe outra asignación estable A' na que, dado un home h e dúas mulleres m, m', tal que $(h, m) \in A$ $(h, m') \in A'$ e $m' \succ_h m$. É dicir, non existirá outra asignación estable na que un home estea emparellado a unha muller que prefira antes que á que foi emparellado na asignación producida polo algoritmo de Gale-Shapley.*

Demostración. Supoñamos que un home h é emparellado na asignación producida por Gale-Shapley A cunha muller m, $(h, m) \in A$ que non sexa a mellor parella posible.

Sexa m' a mellor parella posible e sexa.

En GS, os homes realizan as súas propostas en orde decrecente, polo que, nalgunha iteración, existirá un primeiro home h será rexeitado pola primeira muller m' da súa orde de preferencia.

Unha vez que h é rexeitado por m', m' aceptará a proposta doutro home h', polo que teremos que:

$$h' \succ_{m'} h. \tag{2.1}$$

Supoñamos que exista outra asignación A' tal que $(h', m) \in A'$. O home h' aínda non foi rexeitado por ningunha muller no momento no que se produce o rexeitamento de m' a h, xa que se trata do primeiro rexeitamento.

Deste xeito, h' aínda non lle tería enviado a proposta a m' cando lle envía a proposta a m', polo que:

$$m' \succ_{h'} m. \tag{2.2}$$

Polo tanto, teremos a asignación A' tal que $(h', m) \in A'$, pero, por 2.1 e 2.2, esta asignación non será estable, polo que temos unha contradición. □

Pola contra, cabe plantexarse como de óptimo é o emparellamento producido para a parte que acepta as propostas. Como veremos a continuación, o emparellamento non só non será óptimo pola súa parte, senon que será pésimo.

Proposición 2.6. *O emparellamento producido por Gale-Shapley é pésimo (o peor posible dentro dos emparellamentos estables) para as mulleres.*

Demostración. Sexan dous homes h e $h\prime$ e dúas mulleres m e $m\prime$. Supoñamos que m é emparellada con h nunha asignación A, pero h non é a peor parella posible para m. Entón existe unha asignación estable A' na que h está emparellada con $h\prime$, home que lle gusta menos que h, $h \succ_m h'$. Sexa $m\prime$ a parella de h en A'. Como sabemos polo teorema anterior que este emparellamento é óptimo para os homes, temos que $m\prime \succ_h m$. Deste xeito, m e h prefirense entre eles antes que as súas parellas en A', polo que A' non pode ser un emparellamento estable. □

Teorema 2.7. *O algoritmo de Gale-Shapley é* ***a proba de manipulacións de estratexias*** *por parte dos homes, de xeito que un home non pode manipular a súa orde de preferencias de xeito que isto lle beneficie. Do mesmo modo, tamén contará coa propiedade de ser* ***a proba de manipulacións grupais de estratexias*** *por parte dos homes, de xeito que non é posible que un conxunto de homes coordine unha manipulación das súas ordes de preferencias de xeito que estes todos se vexan modificados. Si que será posible, non obstante, que algúns deles acaben cunha parella mellor, mentres que o resto manteñan a mesma parella.*

Demostración. Por unha banda, a demostración da proba da non manipulabilidade de estratexias para homes é trivial a partir da optimalidade da solución. O algoritmo asegura que o resultado será o mellor posible para cada home, polo que cambiar a súa orde de preferencias só poderá levar a un peor emparellamento. Do mesmo xeito, calquera manipulación por parte dun grupo levará a un emparellamento peor para algúns dos elementos do grupo, xa que, debido á característica de optimalidade, a única opción de que o emparellamento resultante fose mellor sería que fose inestable, o que sería unha contradición. □

De novo, plantexámonos a dúbida de se esta propiedade de ser a proba de manipulacións tamén é compartida pola parte que acepta as propostas. Neste caso, a resposta volve a ser negativa.

Proposición 2.8. *O algoritmo de Gale-Shapley non é a proba de manipulacións para as mulleres.*

Demostración. Sexan 3 homes h_1, h_2 e h_3 e 3 mulleres m_1, m_2 e m_3 coas seguintes preferencias:

$$\begin{aligned} \succ_{h_1} &:= m_2, m_1, m_3 & \succ_{m_1} &:= h_2, h_1, h_3 \\ \succ_{h_2} &:= m_2, m_3, m_1 & \succ_{m_2} &:= h_3, h_2, h_1 \\ \succ_{h_3} &:= m_3, m_2, m_1 & \succ_{m_3} &:= h_1, h_2, h_3. \end{aligned}$$

No caso de que todo o mundo aporte as súas preferencias reais, o algoritmo de Gale-Shapley rematará coa asignación $A = \{(h_1, m_1), (h_2, m_2), (h_3, m_3)\}$. Non obstante, se a muller m_2 modifica as

súas preferencias ás seguintes: $\succ_{m_2} := h_3, h_1, h_2$, a asignación pasará a ser $A' = \{(h_1, m_1), (h_3, m_2), (h_2, m_3)\}$, polo que a muller m_2 pasará de ser emparellada co seu segundo home preferido ao seu home mais preferido. □

Neste punto, un pode plantexarse que acontecerá no caso de que o número de homes e mulleres non sexa igual. A extensión do algoritmo de Gale-Shapley ao mesmo problema do matrimonio con distintos números polas dúas partes é natural, funcionando do mesmo xeito, pero producindo un emparellamento no que algún home ou muller permanecerá solteiro ou solteira. En concreto, sexa n o número de homes e k o número de mulleres, no caso de que $n > k$, $n-k$ homes quedarán solteiros, mentres que, se $k > n$, $k - n$ mulleres quedarán solteiras.

Vexamos agora como podemos formalizar este problema. Supoñamos que estamos no caso de que os homes son os que realizan as propostas. Por unha banda, sexan n homes e k mulleres tal que $n > k$. Deste xeito, incluiremos unha serie de mulleres artificiais, $\{m_{k+1}, m_{k+2}, ..., m_n\}$. Estas mulleres artificiais serán engadidas ao final da orde de preferencias para cada home, é dicir:

$$O_{h_i,j} = m_j \ \forall i \in \{1, ..., n\} \ \forall j \in \{k+1, ..., n\}.$$

Deste xeito, a execución de Gale-Shapley será a mesma que a vista previamente, producindo un emparellamento no que algúns dos homes serán emparellados ás mulleres artificiais, que serán os que finalmente quedarán solteiros, sen parella. O caso no que $k > n$ é análogo, no que as mulleres engadirían homes artificiais ás súas ordes de preferencias, sendo aquelas que só poidan aceptar propostas dos homes artificiais as que queden sen parella.

Como vimos previamente, no caso de darse a igualdade de número polas dúas partes, aqueles que realizan as propostas vense beneficiados, pois contarán coa propiedade de que o emparellamento producido por Gale-Shapley será óptimo para eles. Non obstante, no caso de que os números sexan distintos, o lado de maior número verase en desvantaxe. Isto, motivado principalmente pola maior competición para a parte que está en risco de verse sen parella, pode comprobarse de forma rigurosa en [2].

2.2. Extensión: Admisión a universidades

Como ben se indicou previamente, a forma xeral do problema presenta outro parámetro q, chamado cuota, que será o número de propostas que será capaz de aceptar cada un dos elementos que recibe as propostas. Os contextos nos que se soe presentar este problema son variados, sendo os mais populares, ademais dos que trataron D. Gale e L. S. Shapley orixinalmente, as seguintes versións: o emparellamento de pacientes a hospitais, e o de estudantes a universidades, sendo este último no que nos centraremos.

A situación na que se desenvolve o problema a estudar é a seguinte: un conxunto de n estudantes $E = \{e_1, ..., e_n\}$ desexa enviar solicitudes de admisión a unha serie de universides, que cada estudante $e_i \in E$ ordeará segundo a súa preferencia $\succ_{e_i}$; por outra parte, dentro do conxunto de m universidades $C = \{c_1, ..., c_m\}$, cada universidade $c_j \in C$ só poderá admitir un total de q_j estudantes, polo que tamén deberá ordear aos estudantes por unha orde de preferencia $\succ_{c_j}$ para poder facer unha elección. Na versión a estudar, consideraremos que, unha vez ambos os dous grupos presentan as súas preferencias, serán as universidades as que realizarán as propostas aos estudantes.

Como parece indicar a intuición, o algoritmo de Gale-Shapley é facilmente extendible e transladable a este problema. Non obstante, contrario ao que poida parecer, os resultados obtidos na asignación non serán os mesmos [3], pois teremos que algunhas das propiedades non se verificarán neste caso.

Observación 2.9. Nótese que neste problema xa non contamos coa simetría presente no caso particular, posto que, ao contar un dos grupos co parámetro das cuotas e o outro non, non será posible intercambiar os roles dos dous grupos e seguir estando ante o mesmo problema.

Definición 2.10. Para evitar redundancias, retomarase a definición do algoritmo para o caso particular, descrito no teorema 2.2.

A única modificación será nos pasos b) de cada iteración, no que cada universidade $c_j \in C$ aceptará as propostas dos seus estudantes preferidos ata chegar a un número de q_j, rexeitando o resto. De novo, no caso de que non haxa suficientes plazas para todos os estudantes, é dicir, $\sum_{j=1}^{m} q_j < n$, poderase introducir unha universidade artificial c_{m+1}, cunha cuota $q_{m+1} = n - \sum_{j=1}^{m} q_j$, que estará na última posición das ordes de preferencias para todos os estudantes. Deste xeito, os estudantes que, tras a execución do algoritmo acaben finalmente emparellados a esa universidade artificial, serán os que non serán admitidos en ningunha universidade. Para o caso no que a suma das cuotas sexa superior á do número de estudantes poderá tratarse de xeito análogo.

Teorema 2.11. *O algoritmo de Gale-Shapley produce un emparellamento estable para o problema de admisión a universidades.*

Demostración. A demostración da estabilidade do emparellamento producido é análoga á presentada na demostración 2.1. □

2.2.1. Propiedades de Gale-Shapley para o problema das admisións a universidades

Teorema 2.12 (Optimalidade)**.** *O emparellamento producido por Gale-Shapley é óptimo para os estudantes.*

Demostración. De novo, a demostración será análoga á presentada no caso particular 2.1.1. □

Proposición 2.13. *Gale-Shapley é a proba de manipulacións por parte dos estudantes.*

Demostración. Igual que para o caso particular, a optimalidade do emparellamento asegura que calquera manipulación só poderá conducir a un peor emparellamento. □

Por outra parte, a optimalidade para as universidades non pode ser directamente extendida do problema do matrimonio. Para calquera universidade c_j cunha cuota $q_j > 1$, a súa preferencia sobre un emparellamento ou outro non poderá ser directamente derivada da súa orde de preferencias, xa que esta universidade deberá definir unha orde de preferencias de conxuntos de estudantes, e non só de estudantes. Deste xeito, a literatura tamén plantexa unha variación da definición do problema inicial, no que as universidades non presentan a súa orde de preferencias sobre os estudantes, senon sobre os subconxuntos de estudantes. Deste xeito, cada orde de preferencias poderá ter unha lonxitude de ata 2^n elementos, sendo m o número de estudantes. A continuación ilústrase algúns exemplos das posibles manipulacións que unha universidade pode levar a cabo, amosando que non é un algoritmo a proba de manipulacións de estratexias:

Exemplo 2.14 (Incentivos para as universidades)**.** Sexan as universidades $C = \{c_1, c_2\}$, coas cuotas $q_{c_1} = 2, q_{c_2} = 1$, e os estudantes $E = \{e_1, e_2\}$, que teñen as seguintes preferencias:

$$\succ_{e_1} := c_1, c_2.$$
$$\succ_{e_2} := c_2, c_1.$$

Sexan por outra parte as seguintes preferencias por parte das universidades sobre os conxuntos de estudantes, sendo $\succ'_{c_1}$ as preferencias manipuladas de c_1:

$$\succ_{c_1} := \{e_1, e_2\}, \{e_2\}, \{e_1\}.$$
$$\succ'_{c_1} := \{e_2\}, \emptyset.$$
$$\succ_{c_2} := \{e_1\}, \{e_2\}.$$

De usar as preferencias reais, o emparellamento producido será $A = \{(c_1, e_1), (c_2, e_2)\}$, xa que ningunha universidade rexeita a ningún estudante, mentres que, de usar as preferencias manipuladas por c_1, o emparellamento sería $A' = \{(c_1, e_2), (c_2, e_1)\}$, polo que c_1 veríase beneficiado de mentir coas súas preferencias.

Non obstante, neste problema as ordes de preferencias non é o único parámetro que as universidades deben presentar, senon que tamén contan coas cuotas. Deste xeito, un pode preguntarse se lle será posible a unha universidade beneficiarse de mentir á hora de comunicar a súa cuota. Como veremos no exemplo a continuación, a resposta será afirmativa:

Exemplo 2.15 (Incentivos para as universidades)**.** Sexan as universidades $C = \{c_1, c_2\}$, coas cuotas $q_{c_1} = 2, q_{c_2} = 1$, e os estudantes $E = \{e_1, e_2\}$, que teñen as seguintes preferencias:

$$\succ_{e_1} \coloneqq c_1, c_2.$$
$$\succ_{e_2} \coloneqq c_2, c_1.$$

Sexan por outra parte as seguintes preferencias por parte das universidades sobre os conxuntos de estudantes:

$$\succ_{c_1} \coloneqq \{e_1, e_2\}, \{e_2\}, \{e_1\}.$$
$$\succ_{c_2} \coloneqq \{e_1\}, \{e_2\}.$$

De usar as cuotas reais, o emparellamento producido será $A = \{(c_1, e_1), (c_2, e_2)\}$. Por outra parte, no caso de que c_1 modificase a súa cuota a $q_1 = 1$, o emparellamento sería $A' = \{(c_1, e_2), (c_2, e_1)\}$, polo que c_1 veríase beneficiado de reducir a súa cuota.

Capítulo 3

Estudo Numérico Gale-Shapley

Neste capítulo realizarase un estudo numérico no que se verifica a utilidade do algoritmo de Gale-Shapley para o problema do matrimonio. Para iso, produciranse conxuntos de datos para distintos tamaños de problema n, xerando de xeito aleatorio as ordes de preferencia para cada home e cada muller.

3.1. Xeración dos conxuntos de datos

Os tamaños de problema escollidos foron $N = \{10, 20, 50, 100, 200, 500, 1000\}$, xa que se trata dun rango de valores suficientes para sacar as conclusións necesarias. Así mesmo, como o algoritmo de Gale-Shapley ten unha complexidade cadrática $\mathcal{O}(n^2)$, incluso o valor mais grande $n = 1000$ permitirá que a execución dos experimentos non teñan un tempo de execución excesivamente elevado.

Unha vez definidos estes valores, xeraranse as ordes de preferencia para os homes $H = \{h_1, h_2, ..., h_n\}$ e as mulleres $M = \{m_1, m_2, ..., m_n\}$. Describiremos o proceso de xeración das ordes de preferencia para o caso dos homes, xa que o proceso para as mulleres será análogo. O mecanismo está baseado no algoritmo de Fisher-Yates [4], que permite xerar unha permutación aleatoria dun conxunto ordeado, neste caso a orde preferencia.

Deste xeito, sexa un home $h_i, i \in \{1, ..., n\}$, a súa orde de preferencia inicializarase asignando a cada índice da súa orde de preferencia a muller do mesmo índice, é dicir, $O_{h_i,j} = m_j, \forall j \in \{1, ..., n\}$. Deste xeito,

$$\succ_{h_i} := m_1, m_2, ..., m_n.$$

A partir de aí, o proceso consistirá en iterar a través de todos os elementos, xerar un índice aleatorio a través do xerador de números pseudoaleatorios baseado no Mersenne Twister [5], e

intercambiar as posicións dos elementos presentes no índice no que se atopa a iteración e no índice xerado aleatoriamente.

Deste xeito, sexa a función xeradora de números naturais aleatorios nun intervalo $[0, l], l \in \mathbb{N}$:

$$U : \mathbb{N} \longrightarrow \mathbb{N}.$$

E sexa a función que intercambia as posicións na orde de preferencia:

$$f : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N} \times \mathbb{N}$$
$$(n_1, n_2) \longmapsto (n_2, n_1).$$

Polo que o algoritmo procederá do seguinte xeito:

$$\forall j \in n, ..., 1 \quad k \coloneqq U(n), \quad f(O_{h_i,j}, O_{h_i,k}).$$

O algoritmo de Fihser-Yates proporciona unha permutación aleatoria sen sesgo sempre que o xerador de números aleatorios tampouco sexa sesgado, condición que se verifica neste caso.

O código empregado para a xeración dos conxuntos de datos pode revisarse no código A.1.

3.2. Resultados

3.2.1. Satisfacción emparellamentos

O primeiro aspecto a estudar será a satisfacción dos emparellamentos. Para iso, será preciso definir unha función que, dado un emparellamento A, proporcione unha medida de como de contento está cada individuo coa súa respectiva parella. A definición desta función poderá realizarse desde enfoques distintos, pero a que se usará neste estudo, creada explicitamente para este caso de uso, caracterízase pola súa facilidade de interpretación. Sexa s dita función:

$$s : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{R}$$
$$(u, v, n) \longmapsto \frac{(n - I_{u,v} - 1)}{(n - 1)}.$$

Por outra parte, tamén precisaremos algunha liña base coa que comparar os resultados obtidos co algoritmo de Gale-Shapley. As seguintes funcións de emparellamento foron usadas para esta comparación:

- Emparellamento aleatorio: esta función xerará os emparellamentos de xeito aleatorio, facendo uso do algoritmo de Fisher-Yates descrito previamente para producir unha permutación que dea lugar ao emparellamento.

- Emparellamento heurístico: esta función basearase en asignar a cada un dos homes, por orde do seu índice, a muller dispoñible que mais alta estea na súa lista. É dicir, sexa H a función que, para un home dado, obteña a muller coa que será emparellado:

$$H : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \longrightarrow \mathbb{N}$$
$$(h, A) \longmapsto O_{h,i}, \ \text{mín}\, i : \forall (h', m) \in A, m \neq O_{h,i}.$$

Deste xeito, como xa se indicou previamente, realizáronse 1000 iteracións para distintos tamaños de problemas, obtendo os resultados presentes na figura 3.1. Tamén se incluiu outra gráfica con escala logarítmica na figura 3.2, para poder visualizar correctamente os valores de satisfacción para valores de n pequenos. Como era de esperar, o emparellamento aleatorio é o que menos

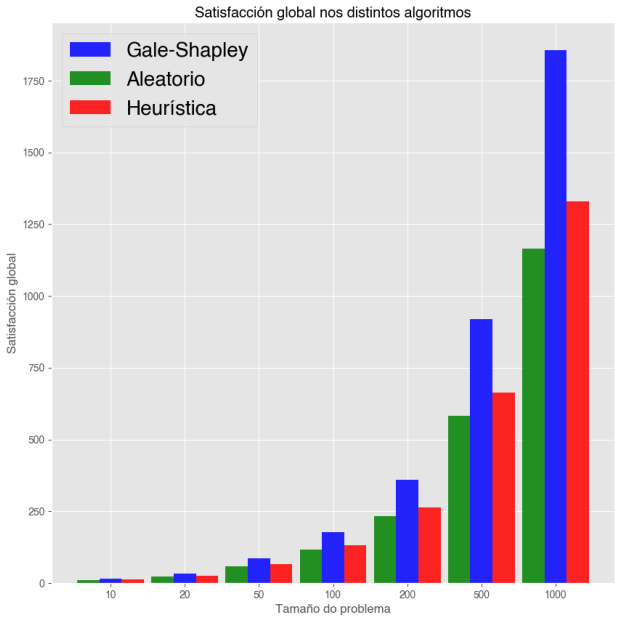

Figura 3.1: Comparación da satisfacción xeral cos emparellamentos nos distintos algoritmos a considerar.

satisfacción xeral produce, posto que en ningún momento se teñen en conta as preferencias para cada individuo. Por outra parte, tamén é posible comprobar o correcto funcionamento do algoritmo Gale-Shapley, posto que supera a calquera dos outros 2 algoritmos para calquera tamaño do problema.

Vexamos agora a comparación da satisfacción para cada unha das partes. En primeiro lugar,

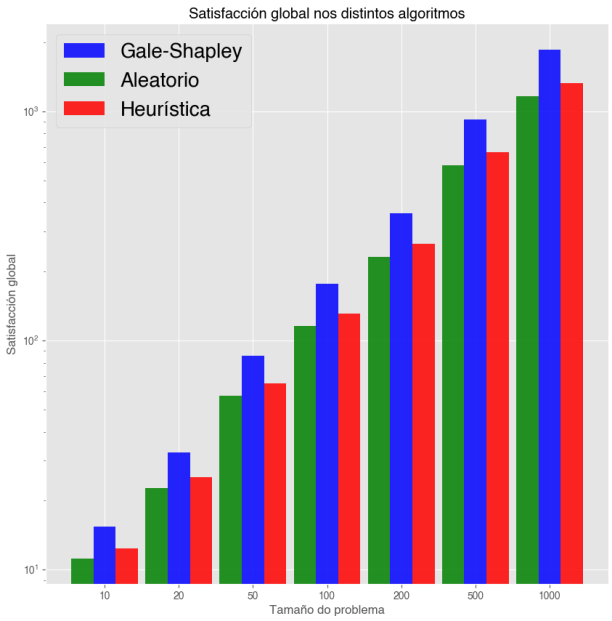

Figura 3.2: Comparación da satisfacción xeral cos emparellamentos nos distintos algoritmos a considerar en escala logarítmica.

comezaremos cos homes que, como xa vimos previamente, son os que contan coa característica de obter un emparellamento óptimo dentro das asignacións estables. Por outra banda, o algoritmo heurístico sabemos que está baseado exclusivamente na satisfacción dos homes, polo que un podería pensar que esta heurística podería chegar a superar a Gale-Shapley nesta métrica. Estes resultados poden verse na figura 3.3.

A continuación, revisaremos a satisfacción das mulleres nos distintos emparellamentos. Como xa se describiu previamente, Gale-Shapley non asegura a optimalidade da asignación para o grupo que acepta as propostas. Do mesmo xeito, a heurística céntrase só nas preferencias dos homes, polo que a satisfacción das mulleres será practicamente a mesma que no caso do algoritmo aleatorio. Estes resultados amósanse na figura 3.4.

Finalmente, comprobouse a optimalidade do emparellamento en Gale-Shapley. Como ben sabemos, esta optimalidade só está presente para os homes, e non as mulleres. Deste xeito, cabe esperar que a satisfacción dos homes sexa superior á das mulleres en Gale-Shapley, como se pode observar na figura 3.5. A satisfacción das mulleres é en torno a un 85 % inferior á dos homes para os distintos tamaños de problema.

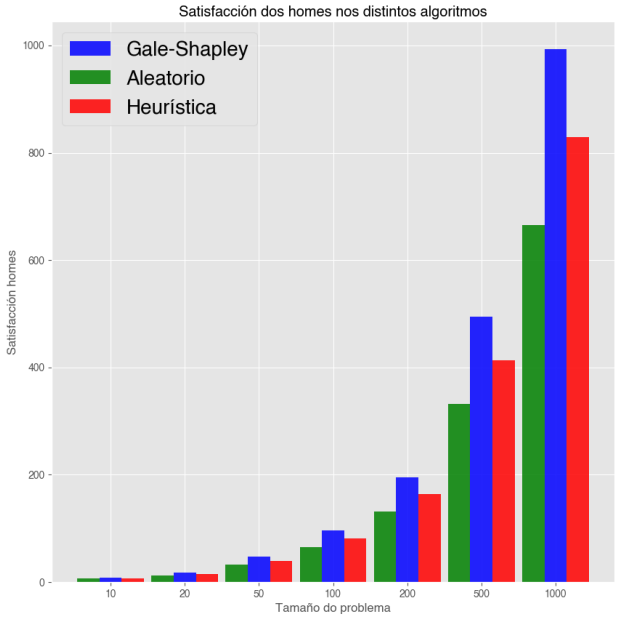

Figura 3.3: Comparación da satisfacción dos homes cos emparellamentos nos distintos algoritmos a considerar.

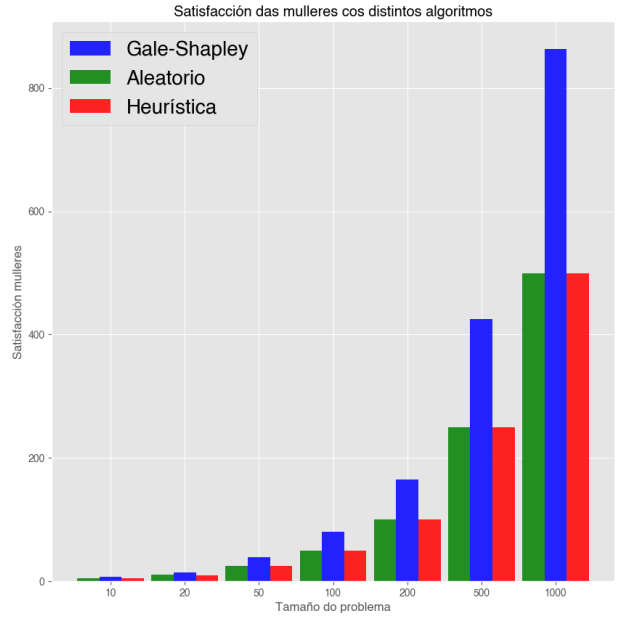

Figura 3.4: Comparación da satisfacción das mulleres cos emparellamentos nos distintos algoritmos a considerar.

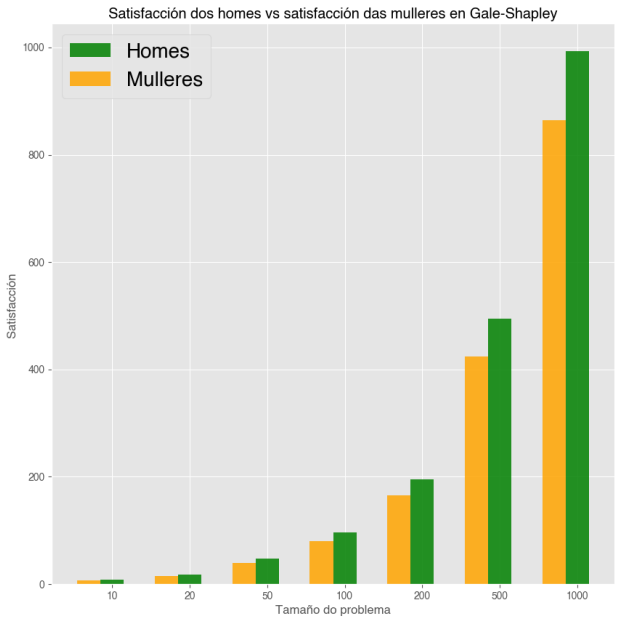

Figura 3.5: Comparación entre a satisfacción dos homes e das mulleres en Gale-Shapley.

3.2.2. Eficiencia de Gale-Shapley

Neste apartado estudarase a eficiencia do algoritmo de Gale-Shapley. En particular, estudarase o número de iteracións realizadas ata a finalización do algoritmo para os distintos n, o que permitirá determinar a súa complexidade computacional.

A figura 3.6 representa con puntos o número de iteracións realizadas nas distintas repeticións do algoritmo en escala logarítmica. Por outra parte, engadiuse a cota superior teórica do número de iteracións $n^2 - 2n + 2$ que non só non se supera en ningunha ocasión, senon que non están cerca. Neste sentido, tamén se incluíu unha regresión lineal que pretende representar a tendencia do número de iteracións a medida que o tamaño do problema aumenta.

O código relacionado coa implementación dos algoritmos está presente no código A.2, mentres que o *script* usado para executar todo o experimento pode observarse no código A.3.

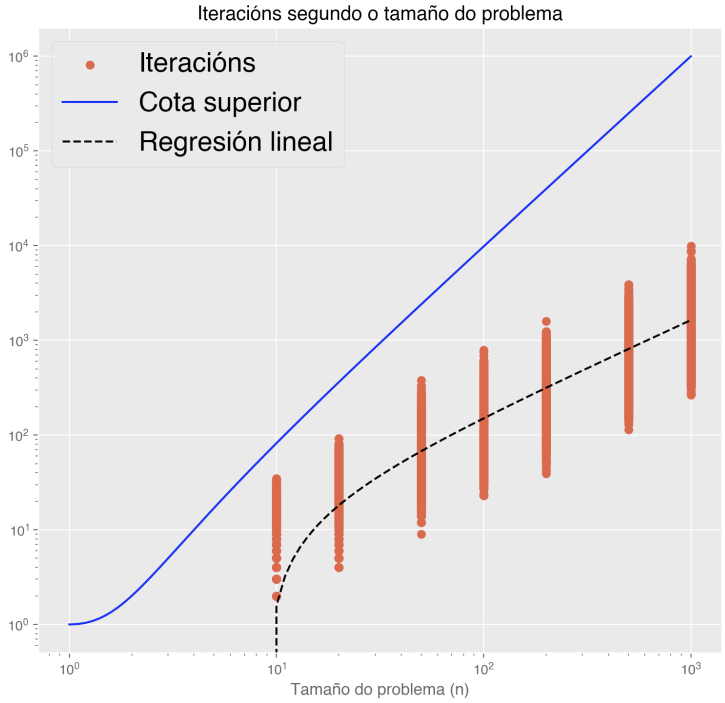

Figura 3.6: Número de iteracións realizadas para distintos tamaños de problema.

Capítulo 4

Mecanismo Xeral de Subasta

Neste capítulo cubrirase unha das aplicacións mais recentes de Gale-Shapley: o desenvolvemento dun mecanismo xeral de subasta para anuncios en motores de búsqueda [6], desenvolvido e posto en práctica por Google.

A publicidade en Internet en xeral, e os anuncios en motores de búsqueda en particular, son casos claros de mercados con dúas partes diferenciadas, xa que temos anunciantes (postores) que compiten por ocos nos que colocar so seus anuncios. Por norma xeral, estes ocos terán valores distintos asignados polos postores, xa que as primeiras posicións terán un maior valor que as que aparezan mais abaixo, debido a que canto mais arriba estea un oco maior será a probabilidade de que un usuario faga click nese anuncio. Un exemplo de subastas nun motor de búsqueda amósase na figura 4.1.

Estes mercados son do mesmo tipo aos descritos previamente, como o caso dos alumnos e as universidades. Polo tanto, non é ningunha sorpresa que algúns dos mecanismos empregados estean basados ou relacionados co algoritmo de Gale-Shapley. Este é o caso do mecanismo descrito neste capítulo, que combinará devandito algoritmo e o mecanismo de asignación de Shapley e Shubik [7], que segue os principios de Gale-Shapley, pero incorpora unha variable que representa o valor que unha das partes obtén ao ser emparellada con outra, ademais de permitir pagos entre individuos.

Antes de nada, procederase a describir un mecanismo típico de subasta de múltiples obxectos con puxas seladas, é dicir, onde os oferentes presentan as súas puxas sen coñecer as puxas do resto de participantes (tradicionalmente facíase por escrito). Isto permitiranos entender mellor a teoría de subastas, ademais de estudar os primeiros intentos de buscar o mellor mecanismo de subastas posibles.

Definición 4.1 (Subasta)**.** Sexan n postores e k obxectos tal que $n > k$. Cada postor conta cun

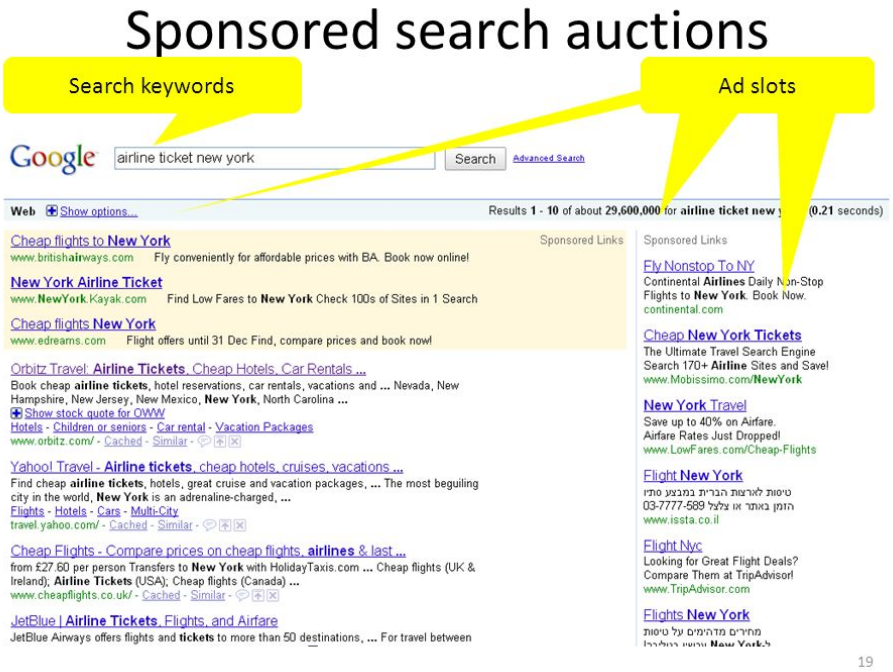

Figura 4.1: Exemplo de ocos de anuncios que son subastados nunha búsqueda no motor de búsqueda Google.

valor $v_{ij} \geq 0 \ \forall i \in \{1, ..., n\}, \ j \in \{1, ..., k\}$ que lle asigna a cada un dos obxectos, aínda que é frecuente atopar subastas nas que os obxectos son iguais, polo que se usará a notación v_i para abreviar. Por outra parte, o vendedor pode definir un prezo de reserva $r_{ij} \geq 0 \ \forall i \in \{1, ..., n\}, \ j \in \{1, ..., k\}$, que será o prezo mínimo polo que lle estará disposto a vender o obxecto j ao postor i. De novo, o máis frecuente será que o prezo de reserva sexa equivalente para todos os postores, polo que se abreviará coa notación r_j para algúns dos exemplos previos, aínda que este non será o caso para o algoritmo que se describirá ao longo do capítulo. Finalmente, cada postor i indicará a súa puxa m_{ij} polo obxecto j, $m_{ij} < v_{ij}$, que tamén pode ser visto como o máximo prezo que está disposto a pagar polo obxecto j.

Unha vez contamos con estes datos de entrada, que son os que as dúas partes poden controlar, poderanse definir outros dous parámetros. Por unha parte, defínese p_{ij} coma o prezo que o postor i paga polo obxecto j. Este dependerá principalmente das puxas realizadas m_{ij}, e, dado un obxecto j, o seu valor será 0 para calquera postor que non gañase a puxa por ese obxecto. Do mesmo xeito, temos a utilidade u_{ij} que un postor i obtén a partir do obxecto j. A utilidade pode representar diversos conceptos, sendo o beneficio obtido a partir do obxecto un dos seus significados máis comúns. Finalmente, a subasta contará cunha asignación A, que será un grupo de pares ordeados (i, j) $i \in \{1, ..., n\}, j \in \{1, ..., k\}$ que representará que o obxecto j foi adxudicado ao postor i.

A partir dos mecanismos de Gale-Shapley e Shapley-Shubik desenvolvéronse varios mecanismos de subasta, entre os que destacan dous: GSP (Segundo prezo de puxa xeral) [8] e VCG (Vickrey–Clarke–Groves) [9]. Estes dous mecanismos e as súas variacións son as bases das subastas en Internet modernas, establecéndose na maioría de motores de búsquedas tras desbancar ao seu predecesor, a subasta baseada no primeiro prezo de puxa. Esta asignaba cada oco ao postor que maior puxa presentase, cobrando un prezo igual á puxa do postor gañador. O problema con este tipo de subastas de primeiro prezo de puxa é que non presentaban equilibrio en estratexias puras (en teoría de xogos, as estratexias puras son aquelas que proporcionan unha definición completa das accións que toma un xogador, como é o caso), é dicir, non existe un perfil de estratexia tal que cada unha delas é mellor que as restantes. Polo tanto, as puxas poden ser axustadas de xeito dinámico dependendo do resto de postores, de posuir esta información previo envío das puxas. Esencialmente, podemos entender esta característica como a de manipulacións de estratexias en Gale-Shapley, introducida no capítulo 2. Na práctica, isto tradúcese en que o gañador i, de saber o resto de puxas (en particular da puxa do segundo postor j que mais ofreceu), podería modificar a súa puxa para que fose da forma $m_i = m_j + \epsilon \ \forall \epsilon > 0$, para poder maximizar a súa utilidade.

GSP xurdiu como primeira alternativa, contando cun procedemento moi semellante ao seu predecesor, pero cada postor coa k puxa mais alta pagará o prezo correspondente á puxa do postor $k+1$. É dicir, o postor que mais ofreceu por un oco non pagará un prezo equivalente á súa puxa, senon que dependerá do ofrecido na segunda puxa mais alta. Non obstante, este tipo de subastas tampouco é a proba de manipulación de estratexias para subastas nas que o número de obxectos é superior a 1, xa que os postores poden modificar as súas puxas para obter uns mellores resultados, como se verá no exemplo a continuación.

Para poder ilustrar o exemplo, introducirase brevemente a definición do mecanismo do GSP, en particular no contexto no que se usa nas subastas de anuncios, que posteriormente se verá que garda semellanzas co mecanismo cuberto neste apartado. Sexan n postores e k ocos. Neste caso todos os ocos consideraranse iguais, polo que cada postor determinará un valor e prezo máximo xeral, e só será no prezo no que se terá en conta a probabilidade de que un usuario faga click nese oco para distinguir os ocos. Cada postor i asigna un valor por click v_i, así como o máximo prezo que está disposto a pagar m_i, que será a súa puxa. Cada postor pagará un prezo por click p_i se gana a puxa, mentres que ese prezo será 0 se a perde. Finalmente, α_j é o número de clicks que un oco recibe por unidade de tempo. A partir destes datos, definirase a utilidade u obtida por un postor i ao que se lle asigna o oco i como $u_i = \alpha_i(v_i - p_i)$.

Exemplo 4.2. Sexan $n = 3$ postores e $k = 2$ ocos. O primeiro oco recibe 100 clicks por hora, mentres que o segundo recibe 50, é dicir, $\alpha_1 = 100, \alpha_2 = 80$. Os valores asignados polos postores son os seguintes: $v_1 = 10, v_2 = 7, v_3 = 2$. Por outra parte, supoñamos que os prezos máximos dispostos a pagar son os mesmos que o valor, é dicir, $m_1 = v_1 = 10, m_2 = v_2 = 7, m_3 = v_3 = 2$,

polo que os prezos pagados por cada oco serán $p_1 = m_2 = 7, p_2 = m_3 = 2$.

Deste xeito, a utilidade obtida polos postores 1 e 2, que gañarían a subasta para os ocos 1 e 2, será calculada do seguinte xeito: $u_1 = 100 \cdot (10 - 7) = 300$, $u_2 = 80 \cdot (7 - 2) = 400$. Pola contra, se o postor 1 observa estes datos e decide cambiar a súa puxa máxima $m_1 = 6$, pasaría a ganar a puxa polo oco 2, pagando un prezo $p_2 = 2$ polo que a súa utilidade veríase incrementada: $u_1 = 80 \cdot (10 - 2) = 640 > 300$.

Por outra parte, a subasta de Vickerey-Clarke-Groves (VCG) xurdiu coa intención de crear un mecanismo de subasta que incentivase aos postores a comunicar as súas verdadeiras preferencias, de xeito que sexa a proba de manipulacións, e evitar os posibles problemas ilustrados no exemplo 4.2.

Este tipo de subasta basease en que, unha vez todos os postores enviaron as súas puxas, o mecanismo central considera todas as posibles combinacións de emparellamentos, quedandose coa que maximiza a suma total das puxas, tendo en conta que só poderá ser usada unha puxa por parte de cada postor. Non obstante, a cantidade que estes postores pagan non é directamente a súa puxa, senon que virá determinado polo dano marxinal causado ao resto de participantes, que será sempre menor ou igual á súa puxa. Este dano marxinal é calculado para cada puxa gañadora a partir da diferenza entre a suma das puxas na mellor combinación de emparellamentos excluindo o participante gañador do obxecto e a suma das puxas do resto de participantes que non gañaron ese obxecto. Vexamos a continuación a definición formal do mecanismo:

Neste caso, introducirase o mecanismo xeral, e non o caso particular das subastas en Internet, xa que, ao contrario que nos casos anteriores, esa simplificación non facilita a súa comprensión. Deste xeito, os distintos parámetros, como o valor, o prezo e a puxa dependen tanto do postor como do oco. Neste caso usaremos a notación proposta en [10]. Sexan n postores $I = \{1, ..., n\}$ e k obxectos $J = \{1, ..., k\}$, e sexa v_{ij} o valor asignado por i a j e m_{ij} o prezo que i está disposto a pagar por j. Sexa A o conxunto de todos os posibles resultados (emparellamentos), e sexa i un postor que gañou un oco. O mecanismo conta cunha autoridade central que tratará de maximizar $\sum_{i \in I} m_{ia_i}$, sendo $a_i \in A$ o obxecto que foi adxudicado a i e m_{ia_i} a puxa correspondente. Denotemos como $\hat{a} \in A$ o mellor resultado posible, isto é, o mellor emparellamento posible para o oco, no caso de que i non estivese na subasta.

Por unha banda, considerarase que cada postor sufrirá un dano $\text{máx}_a \sum_{j \neq i} m_{ja_j}$, isto é, o total das puxas sen ter en conta a puxa do postor que gañou o elemento i. Por outra parte, o mecanismo central compensará a cada postor k devolvendo a cantidade $\sum_{j \neq k} m_{j\hat{a}_k}$, isto é, o total das puxas sen ter en conta o postor i na subasta en xeral. Como o mecanismo pretende atopar o mellor posible resultado, teremos que:

$$\sum_{j \neq i} m_{j\hat{a}_j} \leq \text{máx}_a \sum_{j \neq i} m_{ja_j}.$$

A partir de aí, poderemos definir os prezos e as utilidades derivadas. En primeiro lugar, o prezo pagado por un postor i virá dado por:

$$p_i = \max_a \sum_{j \neq i} m_{ja_j} - \sum_{j \neq i} m_{j\hat{a}_j}. \tag{4.1}$$

Isto é, cada postor i que gaña un oco, deberá pagar a diferenza entre o total das puxas sen ter en conta as súas, menos a suma das puxas no caso de que el non estivese presente na subasta.

Mentres que as utilidades para cada postor i serán da forma:

$$u_i = \begin{cases} v_i(\hat{a}_i - p_i), & \text{se } \sum_{j \neq i} m_{j\hat{a}_j} < \max_a \sum_{j \neq i} m_{ja_j} \\ 0, & \text{se} \sum_{j \neq i} m_{j\hat{a}_j} = \max_a \sum_{j \neq i} m_{ja_j}. \end{cases} \tag{4.2}$$

Exemplo 4.3. Sexan $n = 2$ postores e $k = 2$ obxectos. Sexan as seguintes valoracións:

- $v_{11} = 8 = m_{11}$.
- $v_{12} = 2 = m_{12}$.
- $v_{21} = 5 = m_{21}$.
- $v_{22} = 3 = m_{22}$.

Claramente, aínda que ambos postores prefiren o obxecto 1, a asignación mais favorable será $a = \{(1,1),(2,2)\}$. Por outra, os pagamentos serán da seguinte forma:

- `Postor 1:` $p_1 = \max_a \sum_{j \neq 1} m_{ja_j} - \sum_{j \neq 1} m_{j\hat{a}_j} = 8 - 8 = 0$.
- `Postor 2:` $p_2 = \max_a \sum_{j \neq 2} m_{ja_j} - \sum_{j \neq 2} m_{j\hat{a}_j} = 5 - 2 = 3$.

Proposición 4.4 (Non manipulabilidade)**.** *Indicar as súas preferencias reais é unha estratexia óptima para os postores.*

Demostración. Sexa i un postor que decide representar os seus valores reais $v_{ij} \forall j \in \{1, ..., k\}$ e enviar puxas por ese valor.

O mecanismo recibe estes valores e as puxas de cada un dos outros postores e intenta producir un emparellamento de tal xeito que $\sum_{i \in \{1,...,n\}} m_{ia_i}$ é máximo. Deste xeito, seleccionará o emparellamento $\hat{a}$ tal que

$$v_{i\hat{a}_i} + \sum_{j \neq i} m_{j\hat{a}_j} = \max_a \{v_{ia_i} + \sum_{j \neq i} m_{ja_j}\}. \tag{4.3}$$

Deste xeito, a utilidade para o postor i será:

$$u_{i\hat{a}_i} = v_{i\hat{a}_i} + \sum_{j \neq i} m_{j\hat{a}_i} - \max_a \sum_{j \neq i} m_{ja_j}. \tag{4.4}$$

Mentres que, no caso de que o postor i decidise mentir e enviar outras puxas que non representasen as súas preferencias reais, o mecanismo calculará de novo un emparellamento a', dando lugar á seguinte utilidade para o postor i:

$$u_{ia'_i} = v_{ia'_i} + \sum_{j \neq i} m_{ja'_j} - \max_a \sum_{j \neq i} m_{ja_j}. \tag{4.5}$$

Nótese que neste novo emparellamento é posible que o postor i fose emparellado cun obxecto distinto. Por outra parte, o termo $\max_a \sum_{j \neq i} m_{ja_j}$ mantense constante independentemente da estratexia seguida por i.

Supoñamos agora que a utilidade obtida polo postor i é maior seguindo a segunda estratexia, é dicir, $u_{ia'_i} > u_{i\hat{a}_i}$. Polo tanto, sustituindo 4.4 e 4.5:

$$\begin{aligned} v_{ia'_i} + \sum_{j \neq i} m_{ja'_j} - \max_a \sum_{j \neq i} m_{ja_j} > v_{ia'_i} + \sum_{j \neq i} m_{ja'_j} - \max_a \sum_{j \neq i} m_{ja_j} \Rightarrow \\ v_{ia'_i} + \sum_{j \neq i} m_{ja'_j} > v_{ia'_i} + \sum_{j \neq i} m_{ja'_j}. \end{aligned}$$

Isto é unha contradición coa condición inicial 4.3.

Nótese que no caso de que i non consiga ningún obxecto no emparellamento $\hat{a}$ a súa utilidade será 0. Polo tanto, no caso de intentar cambiar as súas preferencias conseguirá unha utilidade igual ou inferior a 0. □

Deste xeito, podemos revisitar o exemplo 4.2 e verificar como VCG non presenta o mesmo problema que VCG:

Exemplo 4.5. Sexan as condicións descritas en 4.2. Nun primeiro momento, a asignación será a mesma, conseguindo o postor 1 o oco 1, e o postor 2 o oco 2, cada un puxando o mesmo que o valor asignado. Non obstante, os prezos serán calculados do seguinte xeito:

- $p_{11} = 9 - 7 = 2$.
- $p_{22} = 12 - 10 = 2$.

Por outra parte, as utilidades serán calculadas tendo en conta os clicks por hora, α, do mesmo xeito que no exemplo anterior:

- $u_{11} = \alpha_1 \cdot (10 - 2) = 800$.
- $u_{22} = \alpha_2 \cdot (7 - 2) = 400$.

De proceder o postor 1 do mesmo xeito que no exemplo de GSP, o postor 1 pasaría a gañar o oco 2, e o postor 2 o oco 1. Deste xeito, os prezos pasarán a ser os seguintes:

- $p_{12} = 9 - 7 = 2$.
- $p_{21} = 8 - 6 = 2$.

As utilidades, por outra banda, serán:

- $u_{11} = \alpha_2 \cdot (6 - 2) = 320$.
- $u_{22} = \alpha_1 \cdot (7 - 2) = 500$.

Como podemos comprobar, a utilidade do postor 1 viuse reducida, polo que intentar mentir coas súas ordes de preferencia non lle beneficiou do mesmo xeito que no exemplor 4.2.

Nun caso xeral dunha subasta con varios obxectos, non existirá unha relación directa entre unha subasta que use o mecanismo GSP e VCG, xa que, como ben se comprobou previamente, nunha subasta GSP os postores poderán manipular as súas estratexias para conseguir un mellor resultado, mentres que en VCG sempre lles interesará usar as súas preferencias reais. Non obstante, isto non ocorre cando o número de obxectos a subastar é 1, xa que nese caso GSP tamén será non manipulable. Polo tanto, cabe pensar que é posible que as dúas subastas sexan equivalentes nese caso, como comprobaremos no teorema descrito a continuación:

Proposición 4.6. *En subastas nas que o número de obxectos a subastar é 1, os mecanismos GSP e VCG son equivalentes.*

Demostración. Ao tratarse dunha subasta cun único obxecto, só haberá un gañador. En GSP, os resultados para cada postor i poden ser un dos seguintes:

- Gañar o obxecto, que leva a unha utilidade $u_i = v_i - m_j$, sendo j o postor tal que $m_j > m_k$ $\forall k \neq i, k \neq j$.
- Non gañar, polo que a utilidade de i será $u_i = 0$.

Vexamos agora o que acontece en VCG:

- Gañador: $u_i = v_i + \sum_{j \neq i} m_j(\hat{a}_j) - \text{máx}_a \sum_{j \neq i} m_j(a_j)$. Neste caso, o termo $+\sum_{j \neq i} m_j(\hat{a}_j$ é 0, xa que non existe ningún outro postor que gañase un obxecto. Por outra parte, para o termo $\text{máx}_a \sum_{j \neq i} m_j(a_j)$, que representa o valor máximo das puxas no caso de que o gañador i non existise, é directo comprobar que este valor será o da puxa co segundo valor mais alto, é dicir, $\text{máx}_a \sum_{j \neq i} m_j(a_j) = m_j$. Deste xeito, teriamos que a utilidade de i sería equivalente á de GSP:
$$u_i = v_i + \sum_{j \neq i} m_j((\hat{a}_j) - \text{máx}_a \sum_{j \neq i} m_j(a_j) = v_i + m_j.$$
- Perdedor: De novo, a utilidade será $u_i = 0$, xa que, ao non levarse ningún obxecto, os termos $\sum_{j \neq i} m_j((\hat{a}_j)$ e $\text{máx}_a \sum_{j \neq i} m_j(a_j)$ sempre representarán as puxas mais altas, polo que $\sum_{j \neq i} m_j((\hat{a}_j) = \text{máx}_a \sum_{j \neq i} m_j(a_j)$, dando lugar a $u_i = 0$.

Deste xeito, teremos que tanto o gañador como os perdedores obterán as mesmas utilidades en GSP e VCG para o caso de $k = 1$, polo que ambos mecanismos serán equivalentes. □

Unha vez introducidas as subastas e os principais mecanismos nos que se inspira o algoritmo descrito neste capítulo, será posible proceder á introdución do mesmo. Nun primeiro momento, será preciso definir claramente os conceptos empregados, xa que, aínda que a meirande parte deles son comúns ao resto de subastas, isto axudará a fixar a notación a usar, así como aclarar o significado dalgúns deles.

4.1. Introdución ao mecanismo xeral de subasta

Comezaremos este apartado precisando as definicións que usaremos ao longo desta sección. Algunhas delas, como a do concepto de subasta ou de emparellamento estable, xa foron introducidas previamente, pero repetiranse brevemente, xa que nesta aplicación presentan algunhas diferenzas que precisan ser observadas. Ademais, deste xeito fixaremos a notación que será empregada no mecanismo xeral de subasta.

Definición 4.7 (Subasta)**.** Sexa un conxunto de postores $I = \{1, 2, ..., n\}$ e un conxunto de ocos $J = \{1, 2, ..., k\}$. Dado un postor i e un oco j, defínese un *valor* $v_{ij} \geq 0$ que o postor i lle asigna

ao oco j, un *prezo máximo* $m_{ij} \leq v_{ij}$ que o postor i estaría disposto a pagar polo oco j e un *prezo de reserva* $p_{ij} \geq 0$ que o vendedor lle asigna ao oco j, que sería o prezo mínimo polo que estaría disposto a vender ese oco. Unha subasta será unha tripla (v, m, r), onde v, m, er son as matrices de dimensión $n \times k$ con entradas v_{ij}, m_{ij} e r_{ij} respectivamente.

Definición 4.8. Sexa $u = (u_1, u_2, ..., u_n)$ un vector de utilidades non negativo, $p = (p_1, p_2, ..., p_k)$ un vector de prezos non negativos, e $\mu \in I \times J$ unha asignación de postor-oco tal e como introduciramos no capítulo 1. Referirémonos neste capítulo a **asignación** a unha tripla (u, p, μ).

Definición 4.9. (Asignación factible). Unha *asignación* (u, p, μ) dise factible para unha subasta (v, m, r) se, para todo $(i, j) \in \mu$,

$$p_j \in [r_{ij}, m_{ij}]. \tag{4.6}$$

$$u_i + p_j = v_{ij}. \tag{4.7}$$

mentres que, para cada postor non asignado i, a súa utilidade é $u_i = 0$ e para cada oco non asignado j ten un prezo $p_j = 0$.

Definición 4.10. (Asignación estable). Unha *asignación* (u, p, μ) dise estable para unha subasta (v, m, r) se, para cada $(i, j) \in I \times J$, polo menos unha das seguintes desigualdades é satisfeita:

$$u_i + p_j \geq v_{ij}. \tag{4.8}$$

$$p_j \geq m_{ij}. \tag{4.9}$$

$$u_i + r_{ij} \geq v_{ij}. \tag{4.10}$$

Unha interpretación xeométrica das condicións de estabilidade dun emparellamento amósase na figura 4.2.

Definición 4.11. Unha parella $(i, j) \in I \times J$ dise bloqueante se non verifica algunha das desigualdades necesarias para que unha asignación sexa estable.

Definición 4.12. (Optimalidade de postor). Unha asignación estable e factible (u^*, p^*, μ^*) dise óptima desde o punto de vista do postor se, para cada asignación estable e factible (u, p, μ) e todo postor $i \in I$ temos $u_i^* \geq u_i$.

4.2. Algoritmo

Ata o de agora, todas as definicións introducidas nesta sección refírense ao plantexamento do problema a estudar. Estas son independentes do algoritmo escollido para a resolución do mesmo.

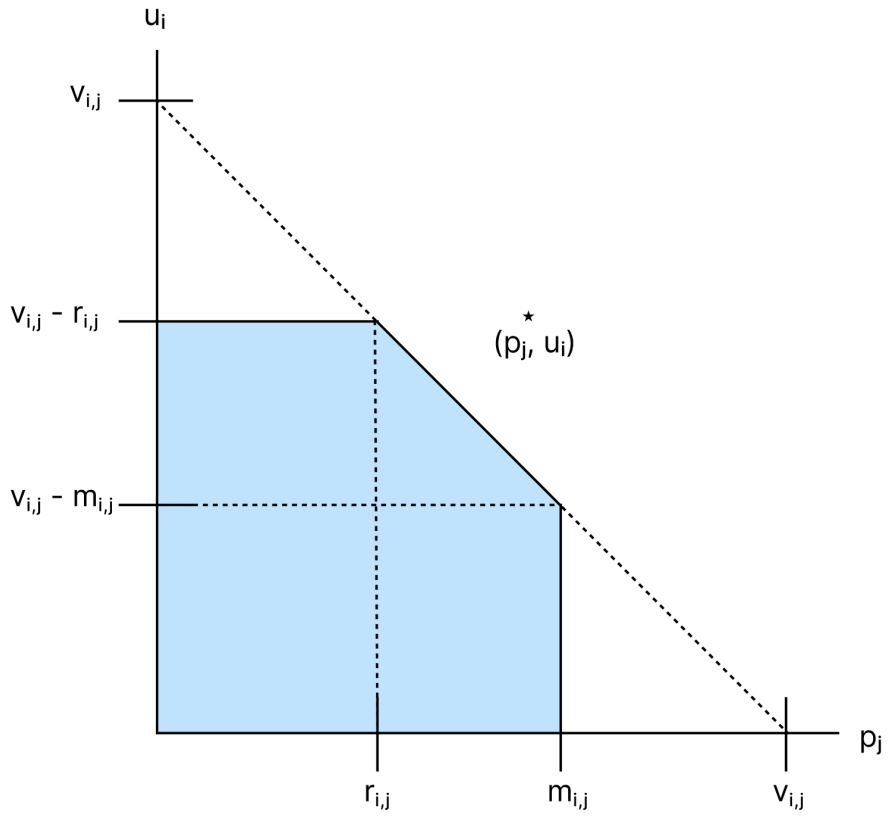

Figura 4.2: Un emparellamento será estable sempre que un postor $i \in I$ e un oco $j \in J$ teñan respectivamente unha utilidade u_i e un prezo p_j tal que as coordenadas (p_j, u_i) son tales que o punto está fóra da área de cor azul.

A continuación presentaranse as definicións dos conceptos asociados ao procedemento iterativo que se presentará posteriormente.

Definición 4.13. (Grafo de actualización). Dada unha subasta (v, m, r) o grafo de actualización para unha asignación (u, p, μ) é un multigrafo bipartito direccionado, no que $\mu \in I \times J \cup \{j_0\}$, sendo j_0 o oco simulado. Este grafo conta con 5 tipos de aristas: para cada postor i e para cada oco $j \in J$ temos que:

- Existe unha *aresta cara adiante* dende i ata j se $p_j \in [r_{ij}, m_{ij}]$.
  O seu peso asociado será $u_i + p_j - v_{ij}$. Como se verá posteriormente, este tipo de aresta irase creando a medida que os prezos se vaian actualizando segundo os emparellamentos.

- Existe unha *aresta cara atrás* dende j ata i se $(i, j) \in \mu$. O seu peso asociado será $v_{ij} - u_i - p_j$.
  O grafo de actualización non conta con ningunha destas arestas nun primeiro momento, senon que se irán engadindo segundo se engadan emparellamentos en cada iteración.

- Existe unha *aresta de prezo de reserva* dende i ata j se $u_i + r_{ij} > v_{ij}$ e $m_{ij} > r_{ij}$.
  O seu peso asociado será de $u_i + r_{ij} - v_{ij}$. Este tipo de aresta correspóndese aos emparellamentos nos que o postor está interesado no oco e a súa utilidade é suficiente para asegurarse de que o emparellamento non será bloqueante.

- Existe unha *aresta de prezo de máximo* dende i ata j se $u_i + m_{ij} > v_{ij}$ e $m_{ij} > r_{ij}$.
  O seu peso asociado será de $u_i + m_{ij} - v_{ij}$. Este tipo de aresta correspóndese aos emparellamentos nos que o postor está interesado no oco e a utilidade asociada á asignación é suficiente para exceder a diferenza entre prezo máximo e o seu valor. No caso de seguir unha estratexia de optimizar ganancias, estas arestas estarán presentes sempre que o postor teña unha utilidade positiva.

- Existe unha *aresta terminal* dende i ata j_0 se $u_i > 0$.
  O seu peso asociado será de u_i. Todo postor con utilidade positiva contará con este tipo de aresta.

Definición 4.14. (Camiño alternado). Un camiño alternado P no grafo de actualización é unha n-tupla $(i_0, j_1, i_1, j_2, ..., i_l, j_{l+1})$. Comeza cun postor non asignado i_0 cunha utilidade $u_{i_0} > 0$, e segue cunha secuencia de arestas cara adiante e arestas cara atrás, de forma que os elementos da tupla sexan, alternativamente, postores e ocos. Finalmente, o último oco debe ser engadido a partir dunha aresta de prezo de reserva, de prezo máximo ou terminal. Todos os vértices deben de ser distintos, exceptuando o último, que pode aparecer mais dunha vez.

Definición 4.15. O peso $w(P)$ dun camiño alternado defínese como a suma dos pesos das súas arestas.

Definición 4.16. Sexa i_0 un postor e sexa y un vértice calquera dun grafo de actualización G. Defínese a distancia $d(i_0, y)$ como o número de vértices percorridos em G dende i_0 ata y usando exclusivamente arestas cara adiante e cara atrás. Se non é posible alcanzar o vértice y dende i_0, a distancia será ∞.

Definición 4.17 (Posición xeral)**.** Unha subasta (v, m, r) esta en posición xeral se, para cada postor i, non existen 2 camiños alternados no grafo de actualización que empecen no postor i, sigan unha serie de arestas cara adiante e cara atrás e rematen nunha aresta de prezo de reserva, prezo máximo ou terminal e teñan o mesmo peso.

Nota: Usarase a notación $G^{(t)}$ en distintos elementos da subasta para facer referencia á iteración t.

Inicialización. Sexa unha subasta (v, m, r) de n postores e k ocos. O algoritmo comeza cunha asignación baleira (u^0, p^0, μ^0), onde cada un dos elementos é inicializado do seguinte xeito:

- $u_i^{(0)} = B \ \forall i \in I$, sendo $B > max\{v_{ij} \mid (i,j) \in I \times J\}$.
- $p_j^{(0)} = 0 \ \ \forall j \in J$.
- $\mu^{(0)} = \emptyset$.

`Iteración` t`:`

Sexa (u^t, p^t, μ^t) unha asignación e G^t o grafo de actualización correspondente á iteración t do algoritmo Asignación Estable. A iteración t do algoritmo consistirá nos seguintes pasos.

1. Buscar o camiño alternado de custe mínimo P de G^t. No caso de non existir ningún camiño alternado, o algoritmo detense e devolve a asignación $(u^{(t)}, p^{(t)}, \mu^{(t)})$. Sexa $w^{(t)}(P)$ o peso asociado a P e sexa

$$P \ = \ (i_0, j_1, i_1, j_2, i_2, ..., j_l, i_l, j_{l+1}) \text{ para un } l \geq 0.$$

2. Calculase o vector de distancias $d^{(t)} \in \mathbb{R}^{n+k}$ tal como se indica en 4.16. É posible usar o algoritmo de Dijkstra para calcular este vector de distancias, xa que, por definición, todas as arestas do grafo de actualización serán non negativas. Do mesmo xeito, os vértices que non están emparellados só poden ser alcanzados desde i_0, o vértice desde o que se están a calcular as distancias, polo que só haberá, como moito, $2k$ vértices alcanzables en cada iteración. Por iso, como Dijkstra ten un custe computacional igual ao cadrado dos vértices alcanzables, sabemos que a complexidade deste paso será $O(k^2)$.

3. Calculase o novo vector de utilidades:

$$u_i^{(t+1)} \ = \ u_i^{(t)} \ - \text{máx}(w^{(t)}(P) \ - \ d(i_0, i), 0) \ \forall i \in I. \tag{4.11}$$

4. Calculase o novo vector de prezos:

$$p_j^{(t+1)} \;=\; p_j^{(t)} \;-\; \text{máx}(w^{(t)}(P) \;-\; d(i_0, j), 0) \;\; \forall j \in J. \tag{4.12}$$

No caso de que a última aresta de P sexa do tipo prezo de reserva, o prezo do último oco j_{l+1} virá dado por $p_{l+1}^{(t+1)} = \text{máx}(p^{(t+1)}, r_{i_l j_{l+1}})$.

5. Finalmente, actualízase a asignación $\mu^{(t)}$ ao longo de P para obter a nova asignación $\mu^{(t+1)}$. Distinguimos 3 casos dependendo da natureza do tipo de aresta final:

- **Aresta terminal:** se P acaba nunha aresta terminal, temos que $j_{l+1} = j_0$, é dicir, é o oco simulado. Desta forma, emparéllase cada postor i_x co oco $j_{x+1} \;\; \forall x \in \{0, 1, ..., l-1\}$. O postor i_l queda desemparellado.

- **Aresta de prezo máximo:** considéranse dous subcasos.

  Caso 1 $j_{l+1} = j_l$ O caso é equivalente ao da aresta terminal.

  Caso 2 $j_{l+1} \neq j_l$ A asignación mantense igual, $\mu^{(t)} = \mu^{(t+1)}$.

- **Aresta de prezo de reserva:** considéranse tres subcasos.

  Caso 1 $\nexists\, i \;:\; (i, j_{l+1}) \in \mu^{(t)}$.

  $\forall x \in \{0, ..., l\}$, emparellar o postor i_x co oco j_{x+1}.

  Caso 2 $\exists\, i \;:\; (i, j_{l+1}) \in \mu^{(t)}$ e $r_{i_l j_{l+1}} \leq p_{j_{l+1}}^{(t+1)}$

  Neste caso, a asignación mantense igual $\mu^{(t)} = \mu^{(t+1)}$.

  Caso 3 $\exists\, i_{l+1} \;:\; (i_{l+1}, j_{l+1}) \in \mu^{(t)}$ e $r_{i_l j_l + 1} > p_{j_{l+1}}^{(t+1)}$.

  Se P é un camiño tal que P non visita j_{l+1} dúas veces, eliminarase o emparellamento, de xeito que $(i_{l+1}, j_{l+1}) \notin \mu^{(t+1)}$. A continuación, emparéllanse o resto de postores e ocos de P do mesmo xeito que o visto para o caso de arestas termianis.

  Se P visita j_{l+1} dúas veces, teremos que $j_{l+1} = j_d$ para algún $d \in \{1, ..., l-1\}$. Deste xeito, os elementos finais de P forman un ciclo con polo menos 2 postores e 2 ocos. Polo tanto, simplemente se procederá a emparellar os elementos de P a partir do índice d, é dicir, procedese a emparellar i_x con j_{x+1} para $x \in \{d, d+1, ..., l\}$.

Unha vez o introducido, veranse algunhas das propiedades coas que conta este algoritmo. En primeiro lugar, demostraranse unha serie de invariantes que permitirán derivar as propiedades que o algoritmo descrito presenta.

4.2.1. Invariantes

- **(A1)** A asignación $(u^{(t)}, p^{(t)}, \mu^{(t)})$ xerada na iteración t é estable para a subasta (v, m, r).
- **(A2)** $\forall (i,j) \in \mu^{(t)}$, $u_i^{(t)}$ e $p_j^{(t)}$ verifican as condicións de asignación factible.
- **(A3)** $p_j = 0 \quad \forall j \in J \ : \nexists i \in I \ (i,j) \in \mu$.
- **(B1)** $m_{ij} \geq 0$ e $u_i^{(t)} + m_{ij} = v_{ij} \Rightarrow (i,j) \notin \mu^{(t)}$.
- **(B2)** $m_{ij} \geq 0$ e $u_i^{(t)} + r_{ij} = v_{ij} \Rightarrow (i,j) \in \mu^{(t)}$ ou $p_j^{(t)} \geq r_{ij}$.

Demostracións dos invariantes. Todos os invariantes poden ser demostrados por indución, como se verá a continuación. Comenzaranse demostrando os invariantes (B1) e (B2), xa que se usarán nas demostracións de (A1), (A2) e (A3).

Demostración (B1).

- Caso base, $t = 0$. Trivial, non existe ningún emparellamento.
- Caso $t+1$. Sexa i un postor interesado no oco j tal que non están emparellados, $(i,j) \notin \mu^{(t)}$. Temos que $m_{ij} \geq 0$ e $u_i^{(t+1)} + m_{ij} = v_{ij}$. No algoritmo só é posible emparellar un oco e un postor se estes están presentes nun camiño alternado de xeito consecutivo. Á súa vez, para que isto ocorra será preciso que exista unha aresta que una os dous vértices. Vexamos que, nas condicións deste caso, non existe aresta de ningún dos tipos:

  Aresta terminal. j non é o oco simulado, polo que non pode existir unha aresta deste tipo.

  Aresta de prezo máximo. Unha das condicións necesarias para este tipo de arestas é $u_i^{(t+1)} + m_{ij} > v_{ij}$. Como temos que $u_i^{(t+1)} + m_{ij} = v_{ij}$, non se pode dar esta aresta.

  Aresta de prezo de reserva. $m_{ij} > r_{ij}$ e $u_i^{(t+1)} + m_{ij} = v_{ij} \Rightarrow u_i^{(t+1)} + r_{ij} < v_{ij}$, polo que non se pode dar esta aresta.

  Aresta cara atrás. Trivial, xa que $(i,j) \notin \mu$.

  Aresta cara adiante. Veremos que, no caso de darse esta aresta, tería peso negativo. Para darse esta aresta, ten que darse que $p_j \in [r_{ij}, m_{ij})$. En particular, $p_j < m_{ij}$. Pola condición $u_i^{(t+1)} + m_{ij} = v_{ij}$, temos que $u_i^{(t+1)} - v_{ij} = -m_{ij}$. Polo tanto, o seu peso asociado sería $u_i + p_j - v_{ij} = p_j - m_{ij} < 0$, xa que $0 \leq p_j < m_{ij}$.

  Deste xeito, non existe ningunha aresta entre i e j e $(i,j) \notin \mu^{(t+1)}$. □

Demostración (B2).

- Caso base, $t = 0$. Trivial, non se dan as condicións en ningún debido ao vector de utilidades $u_i^{(0)} \forall i \in I$.

- Caso $t + 1$. Se a parella de postor e oco xa fose emparellada nalgunha iteración anterior xa se verificaría, posto que $(i, j) \in \mu^{(t)} \Rightarrow (i, j) \in \mu^{(t+1)}$. Se $(i, j) \notin \mu^{(t)}$, sería preciso distinguir 2 casos:

  Caso $u_i^{(t)} + r_{i,j} = v_{i,j}$. Por hipótese de indución, como $(i, j) \notin \mu^{(t)}$, entón temos que se verifica $p_j^{(t)} \geq r_{i,j}$. Como $p_j^{(t+1)} \geq p_j^{(t)}$, as condicións do lema estarían satisfeitas.

  Caso $u_i^{(t)} + r_{i,j} \neq v_{i,j}$. Como $u_i^{(t+1)} \leq u_i^{(t)}$, temos que a desigualdade sería $u_i^{(t)} + r_{i,j} \geq v_{i,j}$. Do mesmo xeito, combinando coa actualización da utilidade, teriamos

  $$u_i^{(t)} - w^{(t)}(P) + d^{(t)}(i_0, i) + r_{ij} = v_{ij},$$

  polo que, combinado coa desigualdade anterior, conclúese que $w^{(t)}(P) > d^{(t)}(i_0, i)$, isto é, existirá algunha aresta dende i ata algún oco, e esta non será a única presente no camiño, posto que non é igual ao peso. Así mesmo, nas condicións dadas, existiría unha aresta de prezo de reserva dende i ata j. De ser esta a aresta final do camiño alternado, teriamos que (i, j) serían emparellados, seguindo o procedemento de actualización da asignación para camiños alternados de prezo de reserva.

□

Demostración (A1).

- Caso base, $t = 0$. Como $u_i^{(0)} = B \ \forall i \in I$ e $p_j^{(0)} = 0 \ \forall j \in J$. Deste xeito, teremos que $u_i + p_j = B + 0 \geq v_{i,j}$, polo que se verifica (3).

- Caso $t + 1$. Considéranse 3 casos distintos para cada emparellamento (i, j):

  Caso 1: $p_j^{(t)} \in [r_{i,j}, m_{i,j}]$. $(u^{(t)}, p^{(t)}, \mu^{(t)})$ é estable pola hipótese de indución, polo que $u_i^{(t)} + p_j^{(t)} \geq v_{i,j}$. Neste caso preséntanse 2 posibilidades:
  Se $d^{(t)}(i_o, i) \geq w^{(t)}(P)$, a utilidade non se actualizará debido á fórmula do paso 3 do algoritmo, polo que $u_i^{(t+1)}) = u_i^{(t)}$ e $p_j^{(t+1)} \geq p_j^{(t)}$, polo que se verifica 4.8.
  Por outra parte, se $d^{(t)}(i_o, i) < w^{(t)}(P)$, a utilidade e o prezo actualizaranse segundo as fórmulas descritas no algoritmo:

  $$u_i^{(t+1)} = u_i^{(t)} - (w^{(t)}(P) - d^{(t)}(i_o, i)).$$
  $$p_j^{(t+1)} = p_j^{(t)} - (w^{(t)}(P) - d^{(t)}(i_o, j)).$$

  Ademais, como existe unha aresta cara adiante dende i ata j, temos que

  $$d^{(t)}(i_o, j) \leq d^{(t)}(i_o, i) + (u_i^{(t)} + p_j^{(t)}) - v_{i,j}.$$

Polo tanto, susituindo en 4.8

$$u_i^{(t+1)} + p_j^{(t+1)} \geq u_i^{(t)} - w^{(t)}(P) + d^{(t)}(i_o, i) + p_j^{(t)} + w^{(t)}(P) - d^{(t)}(i_o, i) - u_i^{(t)} - p_j^{(t)} + v_{ij}$$
$$= v_{ij} \geq v_{ij}.$$

- Caso 2: $p_j^{(t)} \geq m_{ij}$. A condición 4.9 verifícase trivialmente.
- Caso 3: $p_j^{(t)} < r_{ij}$ e i está interesado en j. $u_i^{(t)}$ é estable pola hipótese de indución, polo que $u_i^{(t)}$ verifica 4.10. Distínguense 2 opcións aquí:
  Se $d^{(t)}(i_o, i) \geq w^{(t)}(P)$, a utilidade non se actualizará debido á fórmula do paso 3 do algoritmo, polo que $u_i^{(t+1)} = u_i^{(t)}$ e $u_i^{(t+1)}$ satisfai 4.10. Se $d^{(t)}(i_o, i) \geq w^{(t)}(P)$, entón $u_i^{(t+1)} = u_i^{(t)} - (w^{(t)}(P) - d^{(t)}(i_o, i))$, mentres que, grazas á aresta de prezo de reserva de i a j, temos que

  $$w^{(t)}(P) \leq d^{(t)}(i_o, i) + (u_i^{(t)} + p_j^{(t)}) - v_{ij},$$

  polo que, sustituindo, en (5), vemos que $u_i^{(t)}$) tamén a satisfai. A presenza da aresta de prezo de reserva pódese obter grazas ás invariantes (B1) e (B2).

□

Demostración (A2).

- Caso base, $t = 0$. Trivial, a inicialización faise de tal xeito que $p_j^{(0)} = 0 \ \forall j \in J$.
- Caso $t+1$. Tal e como se pode ver na actualización dos prezos (paso 4 do algoritmo), temos que $p^{(t+1)} \geq p^{(t)}$. Sexa emparellamento $(i, j) \in G^{(t)}$, no que existe unha aresta cara atrás dende j ata i. Pola hipótese de indución, $(u^{(t)}, p^{(t)}, \mu^{(t)})$ satisfai (A2), polo que, debido á definición de aresta cara atrás, esta terá peso 0, e $d^{(t)}(i_o, i) = d^{(t)}(i_o, j)$.

□

(Demostración A3).

- Caso base, $t = 0$. Trivial, $p_j^{(0)} = 0 \ \forall j \in J$.
- Caso $t + 1$:
  Como sabemos polo paso 4 do algoritmo, $p^{(t+1)} \geq p^{(t)}$. Os ocos emparellados en $\mu^{(t)}$ tamén estarán emparellados en $\mu^{(t+1)}$. Do mesmo xeito, como o emparellamento é realizado a través dun camiño alternado, no que, salvo a última aresta, o resto son cara adiante ou cara atrás, como moito emparellarase 1 oco novo, que non estaba emparellado antes.
  Vexamos que o resto dos ocos non se poden alcanzar dende o postor inicial i_0 dun camiño

alternado P en $G^{(t)}$. Para calquera oco j non emparellado en $t+1$, por hipótese de indución, $p^{(t)} = 0$. Do mesmo xeito, $\forall i \in I, r_{i,j} > 0$ debido á suposición de posición xeral, polo que non existe aresta cara adiante en j e $p_j = 0$.

□

Lema 4.18. *A asignación $(u^{(t)}, p^{(t)}, \mu^{(t)})$ obtida polo algoritmo Asignación Estable é factible e estable.*

Demostración. O invariante (A1) asegura a estabilidade da asignación.
As condicións 4.6 e 4.7 da factibilidade veñen dadas do invariante (A2). Como o algoritmo remata cando non existen mais camiños alternados, a utilidade será $u_i^{(t)} = 0$ para todo postor non emparellado, xa que, de non ser así, existiría un camiño alternado debido á aresta terminal dende i ata j_0. □

Lema 4.19. *O algoritmo Asignación Estable remata en, como moito, $n(2k+1)$ iteracións.*

Demostración. Na iteración inicial, o grafo de actualización $G^{(0)}$ ten, como moito, nk arestas de reserva, nk arestas de prezo máximo e n arestas de prezo terminal. Veremos como, en cada iteración, o número de arestas vese reducida nunha unidade.

Sexa unha iteración t do algoritmo Asignación estable. Vexamos que, dado un camiño alternado $P = (i_0, j_1, i_1, ..., j_l, i_l, j_{l+1})$, a aresta $(i, j) = (i_l, i_{l+1})$ non aparecerá no grafo de actualización $G^{(t+1)}$. Separamos agora en casos:

Caso 1: Se (i, j) é unha aresta terminal, entón $w^{(t)}(P) = d^{(t)}(i_0, i) + u_i^{(t)}$, polo que $u_i^{(t+1)} = u_i^{(t+1)} - (w^{(t)}(P) - d^{(t)}(i_0, i)) = 0$.

Caso 2: Se (i, j) é unha aresta de prezo máximo, entón $w^{(t)}(P) = d^{(t)}(i_0, i) + (u_i^{(t)} + m_{ij} - v_{ij})$, polo que $u_i^{(t+1)} + m_{ij} = u_i^{(t)} - (w^{(t)}(P) = d^{(t)}(i_0, i)) + m_{ij} = v_{ij}$.

Caso 3: Se (i, j) é unha aresta de prezo máximo, entón $w^{(t)}(P) = d^{(t)}(i_0, i) + (u_i^{(t)} + r_{ij} - v_{ij})$, polo que $u_i^{(t+1)} + r_{ij} = u_i^{(t)} - (w^{(t)}(P) = d^{(t)}(i_0, i)) + r_{ij} = v_{ij}$.

Como as utilidades non aumentan e os prezos non se ven reducidos en ningunha parte do algoritmo, a aresta (i, j) non volverá aparecer en ningunha iteración $\hat{t} > t$.

Deste xeito, teremos que o algoritmo rematará, como moito, en $nk + nk + n = n(2k + 1)$ iteracións. □

Lema 4.20. *Sexa (v, m, r) unha subasta en posición xeral, e sexa (u', p', μ') unha asignación factible e estable. Para calquera iteración t do algoritmo Asignación Estable, teremos que $u'_i \leq u_i^{(t)}$ $\forall i \in I$ e $p'_j \geq p_j^{(t)}$ $\forall j \in J$.*

Demostración. Esta demostración, de caracter técnico, non se incluirá debido á súa extensión, pero poderá consultarse en [6]. □

Teorema 4.21. *Se unha subasta (v, m, r) está en posición xeral, terá un emparellamento estable óptimo para os postores único. Este emparellamento pode ser atopado cunha complexidade computacional de $\mathcal{O}(nk^3)$.*

Demostración. Sexa unha subasta (v, m, r) en posición xeral. O algoritmo Asignación Estable producirá como resultado un emparellamento (u', p', μ'), que é estable e factible polo lema 4.18. Aplicando o lema 4.20 ao emparellamento anterior despois da última iteración do algoritmo implica que u', v', μ' é preferido a calquera asignación estable por calquera postor, polo que será óptimo para os postores.

Por outra parte, temos que o número máximo de iteracións será $n(2k + 1)$ polo lema 4.19. Por outra parte, grazas a que os pesos do grafo de actualización son non negativos, é posible usar Dijkstra para calcular as distancias, que ten unha complexidade computacional de $\mathcal{O}(k^2)$, sendo k o número de vértices. Deste xeito, como o Dijkstra se usa en cada unha das iteracións, e para o cálculo da complexidade computacional elimínanse as constantes, teremos que esta será $\mathcal{O}(nk^3)$. □

Finalmente, revisarase a manipulabilidade de estratexias por parte dos postores. En particular, probarase un resultado en torno á manipulación de estratexias, tanto para postores individuais como grupos, de xeito similar á que xa se viu previamente para Gale-Shapley.

Lema 4.22 (Lema de Hwang)**.** *Sexa (u, p, μ) unha asignación factible para unha subasta (v, m, r) en posición xeral, e sexa (u^*, p^*, μ^*) a asignación óptima para os postores para esa subasta. Definimos*

$$\hat{I} = \{i \in I \mid u_i > u_i^*\}.$$

Se $\hat{I} \neq \emptyset$, entón existirá un par bloqueante $(i, j) \in (I \setminus \hat{I}) \times J$.

Demostración. De novo, a demostración deste lema será fundamentalmente técnica, separando en casos e empregando técnicas semellantes ás doutras demostracións xa detalladas, polo que non se incluirá nesta memoria. Esta pode consultarse en [6]. □

Teorema 4.23. *Non é posible que un postor ou unha agrupación de postores manipulen as súas puxas de xeito que cada postor na agrupación se beneficie de xeito estricto da manipulación.*

Demostración. Sexa unha agrupación de postores $\hat{I}$ que pretenden beneficiarse de mentir coas súas preferencias. Sexa a tripla (v, m, r) a subasta que representa as preferencias reais para todos os postores, e sexa a tripla $(\hat{v}, \hat{m}, r)$ a subasta que representa as puxas modificadas polo grupo $\hat{I}$.

Nótese que os prezos de reserva son os mesmos, xa que non son controlados polos postores, e $\hat{v}_i = v_i$, $\hat{m}_i = m_i \ \forall i \notin \hat{I}$.

Sexa (u, p, μ) a asignación óptima para os postores na subasta $(\hat{v}, \hat{m}, r)$, que será factible, xa que as restricións de factibilidade son as mesmas para as dúas subastas no caso de que $i \in I \setminus \hat{I}$, mentres que, para $i \in \hat{I}$, teremos que verificar que $p_j \leq m_{ij}$ para todo $(i, j) \in \mu$. Isto é directo, xa que, de non darse, $(\hat{v}, \hat{m}, r)$ non sería óptimo para os postores, xa que sempre haberá outro emparellamento no que non se excedan os prezos máximos e sexa óptimo para os postores.

Como (u, p, μ) é factible, é posible usar o lema 4.22. Deste xeito, temos que existe un par $(i, j) \in (I \setminus \hat{I}) \times J$ que sexa bloqueante para a subastas (v, m, r). □

Capítulo 5

Estudo numérico mecanismo xeral de subasta

Do mesmo xeito que no capítulo 3, ao longo deste capítulo resumirase o estudo numérico realizado a partir do mecanismo xeral de subasta descrito no capítulo anterior. Para iso, desenvolveuse o código na linguaxe de programación Rust presente no Apéndice B, recorrendo á axuda dalgún script en Python para a xeración dos conxuntos de datos.

5.1. Exemplo inicial:

Antes de nada, para poder comprender correctamente o funcionamento do algoritmo, explicarase paso a paso o seguinte exemplo:

Exemplo 5.1. Sexa un conxunto de 3 postores $I = \{i_1, i_2, i_3\}$ que compiten por 2 ocos $J = \{j_1, j_2\}$ de anuncios nunha entrada do motor de búsqueda Google. Estes ocos son das mesmas características, coa única vantaxe de que os primeiros ocos sairán de primeiros na propia páxina, o que leva a unha maior probabilidade de click por parte do usuario.

Deste xeito, cada un dos postores valorará mellor un oco canto menor sexa o seu índice, o que da lugar aos seguintes valores:

- `Postor 1:` $v_{11} = 8.0$ $v_{12} = 6.0$.
- `Postor 2:` $v_{21} = 9.0$ $v_{22} = 7.0$.
- `Postor 3:` $v_{31} = 7.0$ $v_{32} = 6.0$.

Así mesmo, para evitar que este exemplo sexa un caso particular, as súas respectivas puxas máximas non serán os seus valores reais, senon que serán valores inferiores:

- `Postor 1:` $m_{11} = 7.0$ $m_{12} = 5.0$.
- `Postor 2:` $m_{21} = 9.0$ $m_{22} = 4.0$.
- `Postor 3:` $m_{31} = 6.0$ $m_{32} = 5.0$.

Finalmente, os prezos de reserva definidos para cada un dos ocos serán iguais para todos os postores, é dicir:

- `Oco 1:` $r_{11} = 2.0 = r_{21} = r_{31}$.
- `Oco 2:` $r_{12} = 0.0 = r_{22} = r_{32}$.

A partir de aquí, denotaremos por (v, m, r) as matrices de tamaño $n \times k$ que teñen como respectivas entradas v_{ij}, m_{ij} e r_{ij}. Deste xeito, a nosa subasta será a tripla (v, m, r). A partir de aquí, xa poderemos inicializar a tripla (u, p, μ), que será coñecida como o emparellamento.

En primeiro lugar, definimos o valor B co que inicializaremos o vector de utilidades:

$$B > \text{máx}\{v_{i,j} \mid (i,j) \in I \times J\} \Rightarrow B = 10.0$$

Deste xeito, inicializamos o emparellamento:

- $u_i^{(0)} = 10.0 \ \forall i \in I$.
- $p_j^{(0)} = 0 \ \ \forall j \in J$.
- $\mu^{(0)} = \emptyset$.

A partir de aquí, crearemos o grafo de actualización $G^{(0)}$ usando a seguinte notación para cada elemento da matriz: sexa $g_{ij}^{(0)}$, representarase como un vector de 5 elementos para indicar, respectivamente: o primeiro valor será maior que 0 se existe unha aresta cara adiante dende i ata j; o segundo elemento será maior que 0 se existe unha aresta cara atrás dende j ata i; o terceiro elemento será maior que 0 se existe unha aresta de prezo de reserva dende i ata j; o cuarto elemento será maior que 0 se existe unha aresta de prezo de reserva dende i ata j; mentres que o quinto elemento será maior que 0 se existe unha aresta terminal dende i ata j_0. Nótese que a matriz $G^{(t)}$ será de dimensións $n \times k + 1$, sendo a última columna a que represente o oco simulado.

Deste xeito, esta matriz terá os seguintes valores na iteración 1:

$$G^{(1)} = \begin{pmatrix} (0,0,4.0,9.0,0.0) & (4,0,4.0,9.0,0.0) & (0,0,0,0,10.0) \\ (0,0,3.0,10.0,0.0) & (3.0,0,5.0,7.0,0.0) & (0,0,0,0,10.0) \\ (0,0,5.0,9.0,0.0) & (6.0,0,4.0,9.0,0.0) & (0,0,0,0,10.0) \end{pmatrix}.$$

Unha representación gráfica deste grafo de actualización pode verse na figura 5.1.

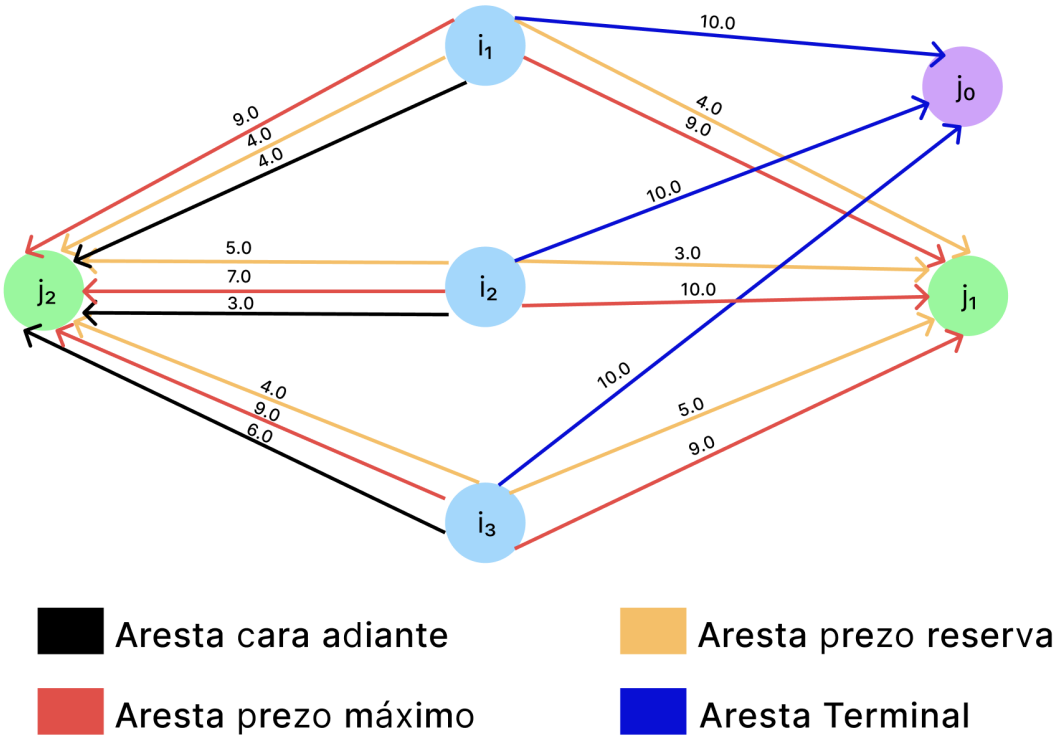

Figura 5.1: Grafo de actualización do exemplo na iteración inicial.

Polo tanto, comezamos a execución do algoritmo:

`Iteración 1:`

1. Claramente, ao non contar con ningunha aresta cara atrás, o camiño alternado de menor peso será $P = (i_2, j_1)$. O seu peso total será o valor da aresta de prezo de reserva, $w^{(1)}(P) = 3.0$.

2. Calcúlanse as distancias dende i_2 a calquera vértice:

   - Postores: $d^{(1)}(i_2, i_1) = \infty$ $d^{(1)}(i_2, i_2) = 0$ $d^{(1)}(i_2, i_3) = \infty$.
   - Ocos: $d^{(1)}(i_2, j_1) = \infty$ $d^{(1)}(i_2, j_2) = 3.0$.

3. Actualízase o vector de utilidades:

$$u^{(2)} = (10.0, 7.0, 10.0).$$

4. Actualízase o vector de prezos:

$$p^{(2)} = (2.0, 0.0).$$

5. Actualízanse os emparellamentos. Trátase do caso de prezo de reserva. Como o oco j_1 non está emparellado en $\mu^{(1)}$, simplemente emparellamos alternativamente todos os elementos presentes en P.

$$\mu^{(2)} = \{(i_2, j_1\}).$$

A continuación, calcúlase de novo o grafo de actualización:

$$G^{(2)} = \begin{pmatrix} (4.0, 0, 4.0, 9.0, 0.0) & (4, 0, 4.0, 9.0, 0.0) & (0, 0, 0, 0, 10.0) \\ (0, 0.0, 0, 7.0, 0.0) & (0.0, 0, 0.0, 4.0, 0.0) & (0, 0, 0, 0, 7.0) \\ (5.0, 0, 5.0, 9.0, 0.0) & (6.0, 0, 4.0, 9.0, 0.0) & (0, 0, 0, 0, 10.0) \end{pmatrix}.$$

E comezamos coa segunda iteración:

`Iteración 2:`

1. $P = (i_1, j_2)$, $w(P) = 4.0$.
2. Distancias ao vértice i_1:

   - Postores: $d^{(2)}(i_1, i_1) = 0$ $d^{(2)}(i_1, i_2) = \infty$ $d^{(2)}(i_1, i_3) = \infty$.
   - Ocos: $d^{(2)}(i_1, j_1) = 4.0$ $d^{(2)}(i_1, j_2) = 4.0$.

3. $u^{(3)} = (6.0, 7.0, 10.0)$.
4. $p^{(3)} = (2.0, 0.0)$.
5. Trátase do caso de prezo de reserva. Como o oco j_2 non está emparellado en $\mu^{(2)}$, simplemente emparellamos alternativamente todos os elementos presentes en P.

$$\mu^{(3)} = \{(i_2, j_1), (i_1, j_2)\}.$$

Novo grafo de actualización:

$$G^{(3)} = \begin{pmatrix} (0.0, 0, 0.0, 4.0, 0.0) & (0.0, 0, 0.0, 5.0, 0.0) & (0, 0, 0, 0, 6.0) \\ (0, 0.0, 0, 7.0, 0.0) & (0.0, 0, 0.0, 4.0, 0.0) & (0, 0, 0, 0, 7.0) \\ (5.0, 0, 5.0, 9.0, 0.0) & (6.0, 0, 4.0, 9.0, 0.0) & (0, 0, 0, 0, 10.0) \end{pmatrix}.$$

`Iteración 3:`

1. $P = (i_3, j_2)$, $w(P) = 4.0$.
2. Distancias ao vértice i_3:

   - Postores: $d^{(3)}(i_3, i_1) = \infty$ $d^{(3)}(i_3, i_2) = \infty$ $d^{(2)}(i_1, i_3) = 0$.

- Ocos: $d^{(3)}(i_3, j_1) = 5.0$ $d^{(3)}(i_3, j_2) = 6.0$.

3. $u^{(4)} = (6.0, 7.0, 6.0)$.
4. $p^{(4)} = (2.0, 0.0)$.
5. Trátase do caso de prezo de reserva. Como o oco j_2 si que está emparellado en $\mu^{(3)}$, temos que mirar o prezo de reserva $r_{3,2}$. Como $r_{3,2} \not> p_2$, o emparellamento mantense igual.

$$\mu^{(4)} = \{(i_2, j_1), (i_1, j_2)\}.$$

Novo grafo de actualización:

$$G^{(4)} = \begin{pmatrix} (0.0, 0, 0.0, 4.0, 0.0) & (0.0, 0, 0.0, 5.0, 0.0) & (0, 0, 0, 0, 6.0) \\ (0, 0.0, 0, 7.0, 0.0) & (0.0, 0, 0.0, 4.0, 0.0) & (0, 0, 0, 0, 7.0) \\ (1.0, 0, 1.0, 5.0, 0.0) & (0.0, 0, 0.0, 5.0, 0.0) & (0, 0, 0, 0, 6.0) \end{pmatrix}.$$

`Iteración 4:`

1. $P = (i_3, j_1)$, $w(P) = 1.0$.
2. Distancias ao vértice i_3:

- Postores: $d^{(4)}(i_3, i_1) = \infty$ $d^{(4)}(i_1, i_2) = \infty$ $d^{(2)}(i_1, i_3) = 0$.
- Ocos: $d^{(4)}(i_1, j_1) = 1.0$ $d^{(4)}(i_1, j_2) = \infty$.

3. $u^{(5)} = (6.0, 7.0, 5.0)$.
4. $p^{(5)} = (2.0, 0.0)$.
5. Trátase do caso de prezo de reserva. Como o oco j_1 si que está emparellado en $\mu^{(4)}$, temos que mirar o prezo de reserva $r_{3,1}$. Como $r_{3,1} \not> p_1$, o emparellamento mantense igual.

$$\mu^{(5)} = \{(i_2, j_1), (i_1, j_2)\}.$$

Novo grafo de actualización:

$$G^{(5)} = \begin{pmatrix} (0.0, 0, 0.0, 4.0, 0.0) & (0.0, 0, 0.0, 5.0, 0.0) & (0, 0, 0, 0, 6.0) \\ (0, 0.0, 0, 7.0, 0.0) & (0.0, 0, 0.0, 4.0, 0.0) & (0, 0, 0, 0, 7.0) \\ (0.0, 0, 0.0, 4.0, 0.0) & (0.0, 0, 0.0, 4.0, 0.0) & (0, 0, 0, 0, 5.0) \end{pmatrix}.$$

`Iteración 5:`

1. $P = (i_3, j_1)$, $w(P) = 4.0$.
2. Distancias ao vértice i_3:

- Postores: $d^{(5)}(i_1, i_1) = \infty$ $d^{(5)}(i_1, i_2) = \infty$ $d^{(5)}(i_1, i_3) = 0$.
- Ocos: $d^{(5)}(i_1, j_1) = \infty$ $d^{(5)}(i_1, j_2) = \infty$.

3. $u^{(6)} = (6.0, 7.0, 1.0)$.
4. $p^{(6)} = (2.0, 0.0)$.
5. Trátase do caso de prezo de reserva. Como o oco j_1 si que está emparellado en $\mu^{(5)}$, temos que mirar o prezo de reserva $r_{3,1}$. Como $r_{3,1} \not> p_1$, o emparellamento mantense igual.

$$\mu^{(6)} = \{(i_2, j_1), (i_1, j_2)\}.$$

$$G^{(6)} = \begin{pmatrix} (0.0, 0, 0.0, 4.0, 0.0) & (0.0, 0, 0.0, 5.0, 0.0) & (0, 0, 0, 0, 6.0) \\ (0, 0.0, 0, 7.0, 0.0) & (0.0, 0, 0.0, 4.0, 0.0) & (0, 0, 0, 0, 7.0) \\ (0.0, 0, 0.0, 0.0, 0.0) & (0.0, 0, 0.0, 0.0, 0.0) & (0, 0, 0, 0, 1.0) \end{pmatrix}.$$

`Iteración 6:`

1. $P = (i_3, j_0)$, $w(P) = 1.0$.
2. Distancias ao vértice i_3:

   - Postores: $d^{(6)}(i_1, i_1) = \infty$ $d^{(6)}(i_1, i_2) = \infty$ $d^{(6)}(i_1, i_3) = 0$.
   - Ocos: $d^{(6)}(i_1, j_1) = \infty$ $d^{(6)}(i_1, j_2) = \infty$.

3. $u^{(7)} = (6.0, 7.0, 0.0)$.
4. $p^{(7)} = (2.0, 0.0)$.
5. Trátase do caso de aresta terminal. O último oco presente en P non se emparella. ao tratarse P dun camiño con lonxitude 2, o emparellamento mantense igual.

$$\mu^{(7)} = \{(i_2, j_1), (i_1, j_2)\}.$$

Novo grafo de actualización:

$$G^{(7)} = \begin{pmatrix} (0.0, 0, 0.0, 4.0, 0.0) & (0.0, 0, 0.0, 5.0, 0.0) & (0, 0, 0, 0, 6.0) \\ (0, 0.0, 0, 7.0, 0.0) & (0.0, 0, 0.0, 4.0, 0.0) & (0, 0, 0, 0, 7.0) \\ (0.0, 0, 0.0, 0.0, 0.0) & (0.0, 0, 0.0, 0.0, 0.0) & (0, 0, 0, 0, 0.0) \end{pmatrix}.$$

`Iteración 7:`

1. Non existe ningún camiño alterando P, polo que o algoritmo detén a súa execución.

A asignación final (u, p, μ) é:

- **Utilidades:** $u = (6.0, 7.0, 0.0)$.
- **Prezos:** $p = (2.0, 0.0)$.
- **Emparellamento:** $\mu = \{(i_2, j_1), (i_1, j_2)\}$.

Unha vez comprendido o funcionamento do algoritmo, procederase a detallar o experimento que puxo a proba o algoritmo, permitindo derivar distintas conclusións sobre as súas características e a súa eficiencia.

5.2. Xeración de datos

Ao longo desta sección, haberá detalles de implementación nos que non se profundizará debido a que xa foron cubertos no Capítulo 3. A pesar de que os conxuntos de datos contan con características distintas, a base da xeración dos datos será común.

Neste caso, os tamaños de problema escollidos foron $N = \{3, 5, 8, 10, 20, 50, 100\}$. O principal motivo de usar tamaños mais pequenos que no estudo de Gale-Shapley é a complexidade computacional do mesmo. Deste xeito, de usar tamaños da mesma orde, os tempos de execución comezarían a dispararse. Por outra parte, nun primeiro momento usarase o caso particular de que o número de postores e o número de ocos é o mesmo, $n = k$, principalmente por unha cuestión de simplicidade, pero posteriormente estes parámetros iranse variando. Do mesmo xeito, o número de iteracións realizadas neste caso tamén será unha orde inferior, repetindo cada execución para cada tamaño 100 veces. O *script* que foi empregado para este experimento está presente no código B.3.

Finalmente, para producir os valores iniciais para cada postor e oco usarase unha distribución uniforme para xerar estes números. Deste xeito, o valor, prezo máximo e prezo de reserva serán xerados con devandita distribución seguindo as seguintes restricións:

$$0.0 \leq r_{ij} < m_{ij} \leq v_{ij} \leq 10.0 \ \forall i, j \in \{1, ..., n\}.$$

A intuición detrás destas restricións será forzar que cada un dos postores estea interesado en todos os ocos, por iso o prezo que puxa é superior ao de reserva, ademais de poder optar por distintas estratexias, por iso a súa puxa pode ser inferior ao valor real que o postor lle asigna ao oco.

O código do *script* implementado en Python empregado para a xeración dos conxuntos de datos amósase no código B.1.

5.3. Resultados

A implementación do algoritmo do mecanismo xeral de subastas empregado para os experimentos pode consultarse no código B.2.

5.3.1. Calidade dos emparellamentos

En primeiro lugar, antes de proceder a revisar os prezos e utilidades obtidas, verificarase que o algoritmo funcionou como era esperado. Ao tratarse dunhas condicións nas que calquera dos

postores conta cunha puxa superior ao prezo de reserva de calquera dos ocos, é de esperar que todos os postores sexan emparellados con un oco.

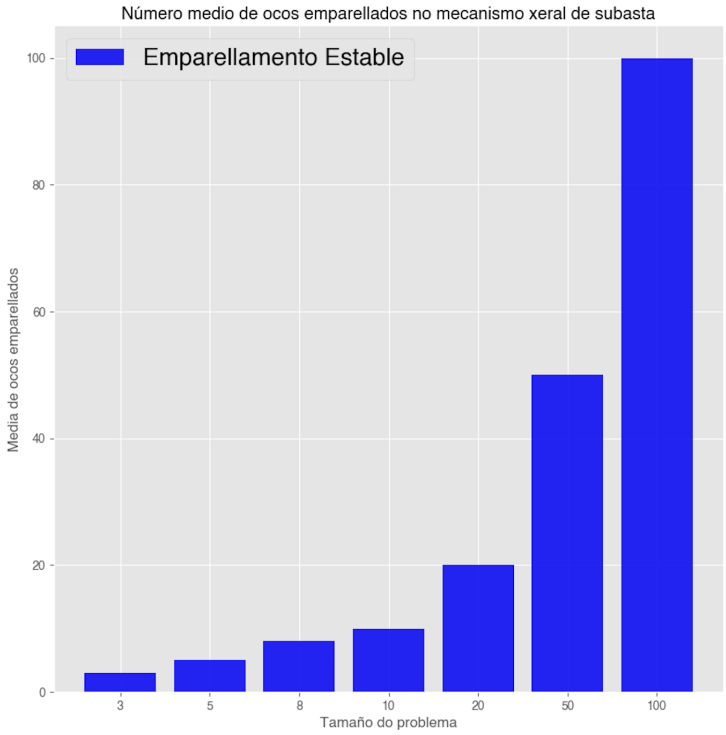

Figura 5.2: Media de emparellamentos para distintos tamaños de problema.

Como se pode observar na figura 5.2, na que se representa a media do número de emparellamentos producidos no experimento para cada un dos tamaños de problema. Aí podemos observar como este valor é exactamente igual ao tamaño do problema, polo que podemos verificar a suposición inicial.

A partir de aí, estudaremos a distribución das utilidades e dos prezos finais. Ao tratarse de dous parámetros xerados a partir dunha distribución uniforme e cunhas cotas restrictivas, un poderá pensar que non deberían estar moi espallados.

Por unha banda, na figura 5.3 pódese observar un diagrama de caixa que amosa a distribución das medias dos prezos. Debido ás restricións nos prezos de reserva, considerablemente inferiores ás puxas e valores, estes son relativamente pequenos. Do mesmo xeito, a medida que se aumenta o tamaño do problema, a diferenza entre a media de prezos é cada vez mais pequena, polo que a maioría de postores pagarán un prezo semellante polo oco. Deste xeito, o prezo de cada oco xa non virá tan influenciado pola diferenza entre os valores que os postores lle asignaron inicialmente.

Por outra parte, a figura 5.4 amosa un diagrama de caixa coa distribución das medias das

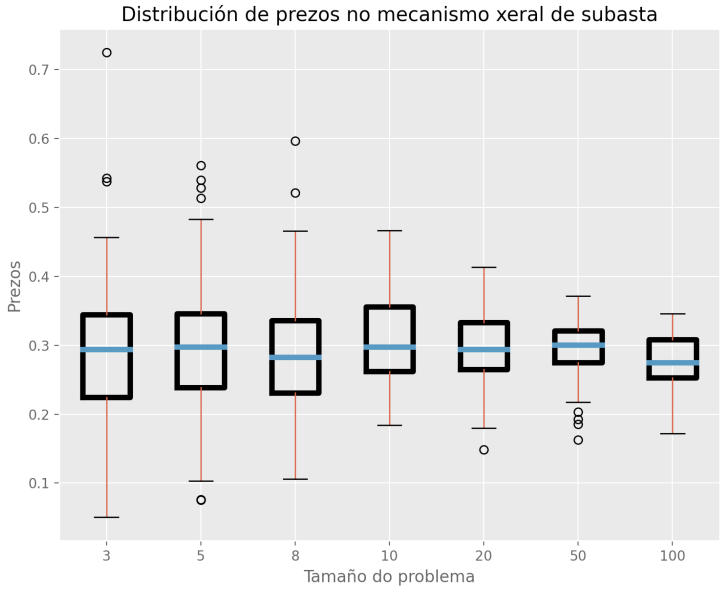

Figura 5.3: Distribución de prezos para distintos tamaños de problema.

utilidades. De novo, pode observarse como estas utilidades son elevadas, preto do máximo imposto inicialmente, sendo mais uniformes a medida que se achega a tamaños de problema mais elevados. Isto é consecuencia directa da distribución dos prezos, xa que, ao ser mais ben baixos, a utilidade tenderá a non baixar demasiado en cada iteración.

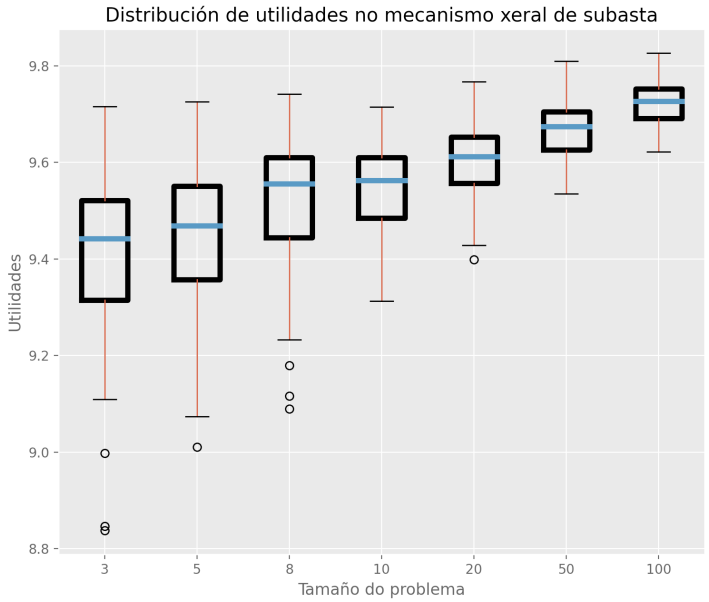

Figura 5.4: Distribución das utilidades para distintos tamaños de problema.

A continuación, realizouse outro estudo fixando o número de postores a 30, e variando o

parámetro k, o número de ocos. Completáronse probas $\forall k \in \{1, ..., 30\}$, de novo realizando 100 iteracións para cada un dos valores. O *script* empregado neste caso está presente no código B.4.

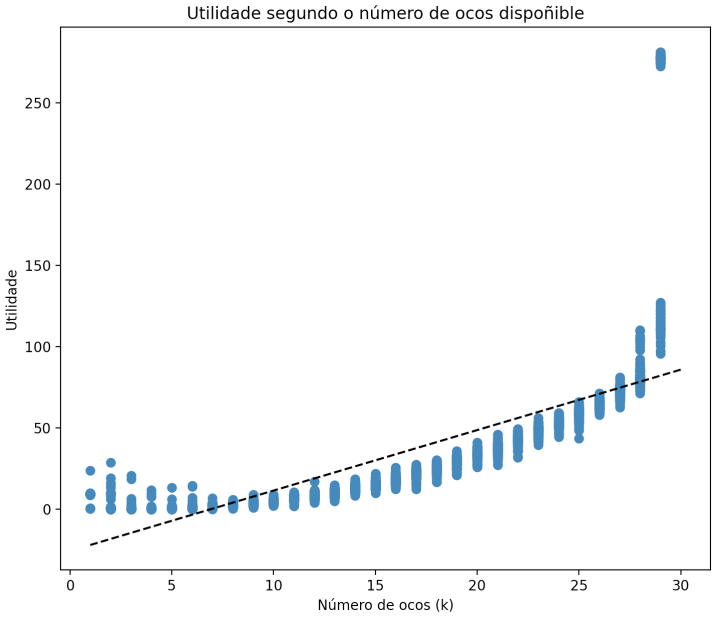

Figura 5.5: Utilidade total segundo o número de ocos dispoñibles.

A pesar de que si que se pode observar na regresión lineal representada na figura 5.5 a tendencia crecente da utilidade total segundo o número de ocos dispoñibles, esta non é tan evidente para valores baixos. Para eses casos de números de ocos baixos, a dispersión dos valores da utilidade leva a contar con valores de utilidade total mais altos que os que se atopan preto do número de 10 ocos. Será a partir de aí cando o número de ocos sexa o suficientemente elevado para contrarrestar devandita dispersión.

5.3.2. Eficiencia do algoritmo

De xeito similar ao estudado en Gale-Shapley, neste caso tamén nos centramos en comprobar o número de iteracións que o algoritmo precisa para a súa finalización, o que permitirá determinar a súa complexidade computacional.

A figura 5.6 representa con puntos as iteracións realizadas polo algoritmo nas distintas repeticións do experimento en escala logarítmica. A liña azul representa a cota superior, a curva $n(2k+1)$, que, ao tratarse do caso particular de $n = k$, esta curva será da forma $2n^2 + n$. Finalmente, a liña de cor negro representa a curva obtida a partir dunha regresión lineal calculada a partir do número de iteracións.

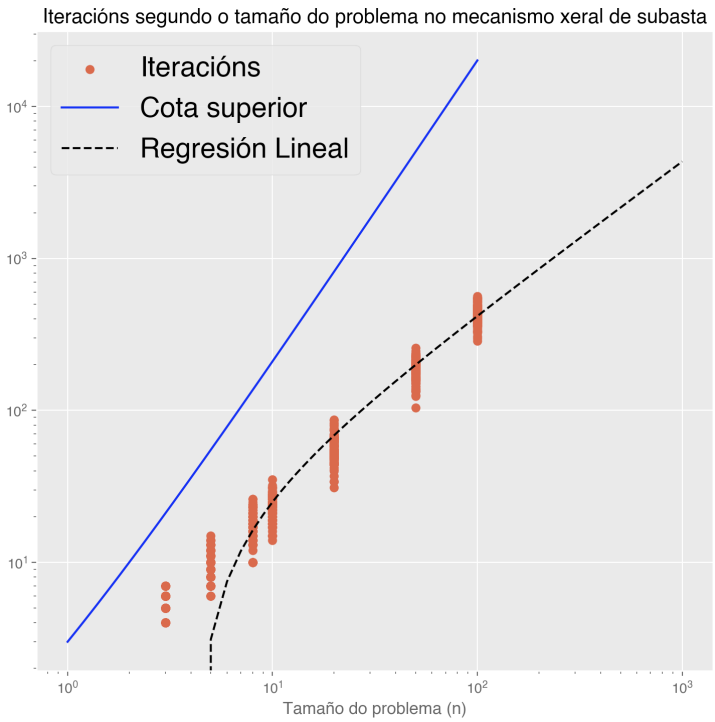

Figura 5.6: Número de iteracións segundo o tamaño do problema.

Como se pode observar, en ningún momento o número de iteracións se acerca ao número máximo de iteracións, situándose a meirande parte dos puntos mais cerca da función identidade que da cota máxima. Isto é debido a que, para alcanzar esta cota, sería preciso contar cunha situación moi particular, na que cada un dos postores presente unha aresta de cada tipo a cada un dos ocos, e en cada iteración só podería eliminarse unha aresta.

Apéndice A

Código do estudo numérico de Gale-Shapley

A.1. Xeración dos conxuntos de datos

```
import random
import sys

if __name__ == '__main__':

    n = int(sys.argv[1])
    n_list = [i for i in range(n)]

    f = open('dataset.txt', 'w')

    f.write(str(n) + '\n')

    for _ in range(n):
        random.shuffle(n_list)
        for i in range(n-1):
            f.write(str(n_list[i]) + ' ')
        f.write(str(n_list[-1]) + '\n')

    for _ in range(n):
        random.shuffle(n_list)
        for i in range(n-1):
            f.write(str(n_list[i]) + ' ')
        f.write(str(n_list[-1]) + '\n')

    f.close()
```

Código A.1: Código usado para a xeración dos conxuntos de datos.

A.2. Código dos algoritmos

```
import sys
import random

def stable_match(n, men_preferences, women_preferences):
    men_index = [0 for _ in range(n)]
    free_men = set([i for i in range(n)])
    women_proposals = [[] for _ in range(n)]
    active_proposals = True
    n_iter = 0
    while len(free_men) > 0:
        n_iter += 1
        for m in free_men:
            curr_choice = men_preferences[m][men_index[m]]
            men_index[m] += 1
            women_proposals[curr_choice].append(m)
        for w in range(n):
            p = len(women_proposals[w])
            if p > 1:
                curr_w_pref = women_preferences[w]
                first_index = curr_w_pref.index(women_proposals[w][0])
                for i in range(1,p):
                    curr_index = curr_w_pref.index(women_proposals[w][i])
                    if curr_index < first_index:
                        women_proposals[w][0], women_proposals[w][i] = women_proposals[w][i],
                            women_proposals[w][0]
                        free_men.add(women_proposals[w][i])
                        first_index = curr_index
                women_proposals[w] = [women_proposals[w][0]]

            if p > 0 and women_proposals[w][0] in free_men:
                free_men.remove(women_proposals[w][0])

    matches = [(x[0],i) for (i,x) in enumerate(women_proposals)]

    return (matches, n_iter)

def get_satisfaction(n, preferences, match):
    total = 0

    for i in range(n):
        index = preferences[i].index(match[i])
        individual_satisfaction = (n - index - 1)/(n-1)
        total += individual_satisfaction
    return total

def read_dataset(path):
    men_pref = []
    wom_pref = []
    f = open(path, 'r')
    n = int(f.readline())
    for _ in range(n):
        line = f.readline()
```

```
52          arr = [int(x) for x in line.split(" ")]
53          men_pref.append(arr)
54      for _ in range(n):
55          line = f.readline()
56          arr = [int(x) for x in line.split(" ")]
57          wom_pref.append(arr)
58
59      return (n, men_pref, wom_pref)
60
61  def random_allocation(n):
62      n_range = [i for i in range(n)]
63      random.shuffle(n_range)
64      matches = [(x, i) for i,x in enumerate(n_range)]
65      return matches
66
67  def heuristic_allocation(n, m_pref):
68      matched_w = set()
69      men_match = [-1 for _ in range(n)]
70      for i in range(n):
71          curr_prefs = m_pref[i]
72          for j in range(n):
73              if curr_prefs[j] not in matched_w:
74                  men_match[i] = curr_prefs[j]
75                  matched_w.add(curr_prefs[j])
76                  break
77
78      matches = [(i,x) for i,x in enumerate(men_match)]
79      return matches
80
81  def get_men_and_women_matches(matches):
82      women_matches = [i for (i,_) in matches]
83      men_matches = [-1 for _ in range(n)]
84
85      for (i,j) in matches:
86          men_matches[i] = j
87
88      return (men_matches, women_matches)
89
90  def write_results(f, algo, n, m_matches, w_matches, m_pref, w_pref, n_iter):
91      women_sat = get_satisfaction(n, w_pref, women_matches)
92      men_sat = get_satisfaction(n, m_pref, men_matches)
93      overall_sat = women_sat + men_sat
94      f.write(algo + ',' + str(n) + ',' + str(overall_sat) + ',' + str(men_sat) + ',' + str(women_sat) +
            ',' + str(n_iter) + '\n')
95
96
97
98  if __name__ == '__main__':
99      path = sys.argv[1]
100
101     n, m_pref, w_pref = read_dataset(path)
102
103     out_path = sys.argv[2]
104
```

```
105     f = open(out_path, 'a')
106
107     gs_matches, n_iter = stable_match(n, m_pref, w_pref)
108     men_matches, women_matches = get_men_and_women_matches(gs_matches)
109     write_results(f, 'gs', n, men_matches, women_matches, m_pref, w_pref, n_iter)
110
111     random_matches = random_allocation(n)
112     men_matches, women_matches = get_men_and_women_matches(random_matches)
113     write_results(f, 'rand', n, men_matches, women_matches, m_pref, w_pref, -1)
114
115     heur_matches = heuristic_allocation(n, m_pref)
116     men_matches, women_matches = get_men_and_women_matches(heur_matches)
117     write_results(f, 'heur', n, men_matches, women_matches, m_pref, w_pref, -1)
118
119     f.close()
```

Código A.2: Código dos algoritmos usados no experimento.

A.3. Código para a realización do experimento

```
1  n=(10 20 50 100 200 500 1000)
2  len=${#n[@]}
3  iter=1000
4
5  for((j=0; j < $iter; j++))
6  do
7      for((i=0; i < $len; i++))
8      do
9          echo "Execucion: ${j} - n: ${n[$i]}"
10         python3 generate_dataset.py ${n[$i]}
11         python3 gs.py dataset.txt results.csv
12     done
13 done
```

Código A.3: Script empregado para o experimento.

Apéndice B

Código do estudo numérico do mecanismo xeral de subasta

B.1. Código para a xeración dos datos

```
import random
import sys


if __name__ == '__main__':

    n = int(sys.argv[1])
    k = int(sys.argv[2])
    n_list = [i for i in range(n)]

    f = open('rust-dataset.txt', 'w')

    f.write(str(n) + '\n')
    f.write(str(k) + '\n')

    for _ in range(n):
        for _ in range(k-1):
            val = random.uniform(9.0,10.0)
            f.write(str(val) + ' ')
        val = random.uniform(9.0,10.0)
        f.write(str(val) + '\n')
        for _ in range(k-1):
            val = random.uniform(7.0,9.0)
            f.write(str(val) + ' ')
        val = random.uniform(7.0,9.0)
        f.write(str(val) + '\n')
        for _ in range(k-1):
            val = random.uniform(0.0,0.5)
            f.write(str(val) + ' ')
        val = random.uniform(0.0,0.5)
```

```
        f.write(str(val) + '\n')

    f.close()
```

Código B.1: Código usado para a xeración dos conxuntos de datos para as subastas.

```

use ndarray::{Array, Array1, Array2, Axis, arr1, concatenate};
use std::io::Write;
use std::vec::Vec;
use std::collections::{HashSet, VecDeque};
use std::fs;

#[derive(Debug)]
struct Matching {
    utilities: Array1<f64>,
    prices:   Array1<f64>,
    pairs: Array1<(usize, usize)>
}

#[derive(Debug)]
struct Node {
    forward_edge: Option<f64>,
    reserve_price_edge: Option<f64>,
    maximum_edge: Option<f64>,
    backward_edge: Option<f64>,
}

impl Default for Node{
    fn default() -> Self {
        Node {
            forward_edge: None,
            reserve_price_edge: None,
            maximum_edge: None,
            backward_edge: None,
        }
    }
}

#[derive(Debug)]
struct TerminalNode {
    terminal_edge: Option<f64>,
}

#[derive(Debug)]
struct UpdateGraph {
    graph: Vec<Vec<Node>>,
    terminal_arr: Vec<TerminalNode>
}
#[derive(Debug)]
enum EdgeType {
    MAXIMUM,
    RESERVE,
    TERMINAL
}
```

```
50
51 #[derive(Debug)]
52 struct AlternatingPath {
53     path: Vec<usize>,
54     cost: f64,
55     final_edge: EdgeType
56 }
57
58 #[derive(Debug)]
59 struct Auction {
60     values: Array2<f64>,
61     max_prices: Array2<f64>,
62     reserve_prices: Array2<f64>,
63     matching: Matching,
64     graph : UpdateGraph
65 }
66
67 #[derive(Debug)]
68 struct Distances {
69     slots_distances: Array1<f64>,
70     bidders_distances: Array1<f64>
71 }
72
73 fn vec_to_arr2(vec: Vec<Vec<f64>>, n: usize, k: usize) -> Array2<f64> {
74     let mut arr = Array2::<f64>::default((n, k));
75     for (i, mut row) in arr.axis_iter_mut(Axis(0)).enumerate() {
76         for (j, col) in row.iter_mut().enumerate() {
77             *col = vec[i][j];
78         }
79     }
80     return arr;
81 }
82
83 impl Auction {
84
85
86     pub fn read_file(file: String) -> Option<(Array2<f64>, Array2<f64>, Array2<f64>)> {
87         let content = std::fs::read_to_string(file).expect("Error when trying to read file");
88         let mut content_split = content.split("\n");
89         let n = content_split.next().unwrap().parse::<usize>().unwrap();
90         let k = content_split.next().unwrap().parse::<usize>().unwrap();
91
92         let mut values = Vec::with_capacity(n);
93         let mut max_prices = Vec::with_capacity(n);
94         let mut reserve_prices = Vec::with_capacity(n);
95
96         for _ in 0..n {
97             let mut line = content_split.next()?.split(" ");
98             let mut val_row = Vec::with_capacity(k);
99             for _ in 0..k{
100                 let val = line.next().unwrap().parse::<f64>().unwrap();
101                 val_row.push(val);
102             }
103             values.push(val_row);
```

```
104             let mut max_line = content_split.next()?.split(" ");
105             let mut max_row = Vec::with_capacity(k);
106             for _ in 0..k{
107                 let max_val = max_line.next().unwrap().parse::<f64>().unwrap();
108                 max_row.push(max_val);
109             }
110             max_prices.push(max_row);
111             let mut reserve_line = content_split.next()?.split(" ");
112             let mut reserve_row = Vec::with_capacity(k);
113             for _ in 0..k{
114                 let reserve_val = reserve_line.next().unwrap().parse::<f64>().unwrap();
115                 reserve_row.push(reserve_val);
116             }
117             reserve_prices.push(reserve_row);
118         }
119         let val_arr = vec_to_arr2(values, n, k);
120         let max_arr = vec_to_arr2(max_prices, n, k);
121         let reserve_arr = vec_to_arr2(reserve_prices, n, k);
122         return Some((val_arr, max_arr, reserve_arr));
123     }
124 
125     pub fn new(values: Array2<f64>, max_prices: Array2<f64>, reserve_prices: Array2<f64>) -> Self {
126 
127         const B: f64 = 10.0;
128 
129         let n_bidders = values.shape()[0];
130         let n_slots = values.shape()[1];
131 
132         for i in 0..n_bidders{
133             for j in 0..n_slots{
134                 if values[[i, j]] < max_prices[[i, j]] {
135                     println!("Maximum prices bidder {} assigned to slot {} is higher than its value, its
                             max_price will be set to its value {}", i, j, values[[i, j]])
136                 }
137             }
138         }
139 
140         let matching = Matching{
141             utilities: Array::from_elem(n_bidders,B),
142             prices:   Array::zeros(n_slots),
143             pairs:    arr1(&[])
144         };
145 
146         let update_graph = Auction::update_graph_values(&matching, &values, &max_prices,
                 &reserve_prices);
147 
148         return Self {
149             values,
150             max_prices,
151             reserve_prices,
152             matching,
153             graph: update_graph
154         }
155     }
```

```
156
157
158     pub fn update_graph_values(matching: &Matching,values: &Array2<f64>, max_prices: &Array2<f64>,
            reserve_prices: &Array2<f64>) -> UpdateGraph{
159
160         let n_bidders = values.shape()[0];
161         let n_slots = values.shape()[1];
162         let mut terminals: Vec<TerminalNode> = Vec::new();
163         for i in 0..n_bidders {
164             if matching.utilities[i] > 0.0 {
165                 terminals.push(TerminalNode{terminal_edge : Some(matching.utilities[i])});
166             }
167         }
168
169         let mut graph = Vec::new();
170
171         for _ in 0..n_bidders {
172             let mut row: Vec<Node> = Vec::with_capacity(n_slots);
173             for _ in 0..n_slots {
174                 row.push(Node::default())
175             }
176             graph.push(row);
177         }
178         for i in 0..n_bidders {
179             for j in 0..n_slots {
180
181
182                 if matching.prices[j] >= reserve_prices[[i,j]] && matching.prices[j] < max_prices[[i,
                        j]]{
183                     let forward_edge_value = matching.utilities[i] + matching.prices[j] - values[[i, j]];
184                     graph[i][j].forward_edge = Some(forward_edge_value);
185                 }
186                 for (bidder, slot) in &matching.pairs{
187                     if *bidder == i && *slot == j {
188                         let backward_edge_value = values[[i, j]] - matching.utilities[i] -
                                matching.prices[j];
189                         graph[i][j].backward_edge = Some(backward_edge_value);
190                         break;
191                     }
192                 }
193                 if matching.utilities[i] + reserve_prices[[i, j]] > values[[i,j]] {
194                     let reserve_price_edge_value = matching.utilities[i] + reserve_prices[[i,j]] -
                            values[[i, j]];
195                     graph[i][j].reserve_price_edge = Some(reserve_price_edge_value);
196                 }
197                 if matching.utilities[i] + max_prices[[i, j]] > values[[i,j]] {
198                     let max_price_edge_value = matching.utilities[i] + max_prices[[i, j]] -
                            values[[i,j]];
199                     graph[i][j].maximum_edge= Some(max_price_edge_value);
200                 }
201             }
202         }
203
204
```

```
205         return UpdateGraph{
206             graph,
207             terminal_arr: terminals
208         }
209     }
210 
211     fn update_graph_edges(&mut self) {
212         let n_bidders = self.values.shape()[0];
213         let n_slots = self.values.shape()[1];
214         let mut terminals: Vec<TerminalNode> = Vec::new();
215 
216         for i in 0..n_bidders {
217             if self.matching.utilities[i] > 0.0 {
218                 terminals.push(TerminalNode{terminal_edge : Some(self.matching.utilities[i])});
219             } else {
220                 terminals.push(TerminalNode{terminal_edge: None});
221             }
222         }
223         self.graph.terminal_arr = terminals;
224         for i in 0..n_bidders {
225             for j in 0..n_slots {
226                 if self.matching.prices[j] >= self.reserve_prices[[i,j]] && self.matching.prices[j] <
                        self.max_prices[[i, j]]{
227                     let forward_edge_value = self.matching.utilities[i] + self.matching.prices[j] -
                            self.values[[i, j]];
228                     if forward_edge_value > 0.0 {
229                         self.graph.graph[i][j].forward_edge = Some(forward_edge_value);
230                     } else {
231                         self.graph.graph[i][j].forward_edge = None;
232                     }
233                 }
234                 for (bidder, slot) in &self.matching.pairs{
235                     if *bidder == i && *slot == j {
236                         let backward_edge_value = self.values[[i, j]] - self.matching.utilities[i] -
                                self.matching.prices[j];
237                         self.graph.graph[i][j].backward_edge = Some(backward_edge_value);
238                         break;
239                     } else {
240                         self.graph.graph[i][j].backward_edge = None;
241                     }
242                 }
243                 if self.matching.utilities[i] + self.reserve_prices[[i, j]] > self.values[[i,j]] {
244                     let reserve_price_edge_value = self.matching.utilities[i] +
                            self.reserve_prices[[i,j]] - self.values[[i, j]];
245                     self.graph.graph[i][j].reserve_price_edge = Some(reserve_price_edge_value);
246                 } else {
247                     self.graph.graph[i][j].reserve_price_edge = None;
248                 }
249                 if self.matching.utilities[i] + self.max_prices[[i, j]] > self.values[[i,j]] {
250                     let max_price_edge_value = self.matching.utilities[i] + self.max_prices[[i, j]] -
                            self.values[[i,j]];
251                     self.graph.graph[i][j].maximum_edge= Some(max_price_edge_value);
252                 } else {
253                     self.graph.graph[i][j].maximum_edge = None;
```

```
254             }
255         }
256     }
257 }
258
259 pub fn update_graph(&self) -> UpdateGraph {
260     return Auction::update_graph_values(&self.matching, &self.values, &self.max_prices,
        &self.reserve_prices);
261 }
262
263 pub fn get_bidders_num(&self) -> usize {
264     return self.values.shape()[0];
265 }
266
267 pub fn get_slots_num(&self) -> usize {
268     return self.values.shape()[1];
269 }
270
271 pub fn is_feasible(&self, matching: Matching) -> bool {
272     for (i,j) in matching.pairs {
273         if matching.prices[j] < self.reserve_prices[[i, j]] ||
274             matching.prices[j] > self.max_prices[[i, j]] ||
275             matching.prices[j] + matching.utilities[i] != self.values[[i,j]] {
276             return false;
277         }
278     }
279
280     return true;
281 }
282
283 fn is_blocking(&self, matching: &Matching, i: usize, j: usize) -> bool {
284     return matching.utilities[i] + matching.prices[j] >= self.values[[i,j]] ||
285         matching.prices[j] >= self.max_prices[[i,j]] ||
286         matching.utilities[i] + self.reserve_prices[[i,j]] >= self.values[[i,j]];
287 }
288
289 pub fn get_blocking_pairs(&self, matching: &Matching) -> Array1<(usize, usize)> {
290     let blocking_pairs = matching
291         .pairs
292         .iter()
293         .filter(|(i,j)| self.is_blocking(matching, *i, *j))
294         .map(|(i,j)| (*i, *j));
295     return Array1::from_iter(blocking_pairs);
296
297 }
298
299 pub fn is_stable(&self, matching: Matching) -> bool {
300     let blocking_pairs = self.get_blocking_pairs(&matching);
301     return blocking_pairs.len() > 0;
302 }
303
304
305 fn get_unmatched_bidders(&self) -> Vec<usize> {
306     let n = self.get_bidders_num();
```

```
        let mut set: HashSet<usize> = HashSet::with_capacity(n);
        for (i, _) in &self.matching.pairs{
            set.insert(*i);
        }
        let bidders: Vec<usize> = (0..n).collect();
        let available_bidders: Vec<usize> = bidders
            .into_iter()
            .filter(|i| !set.contains(&i))
            .collect();
        return available_bidders;
    }

    fn get_unmatched_slots(&self) -> Vec<usize> {
        let m = self.get_slots_num();
        let mut set: HashSet<usize> = HashSet::with_capacity(m);
        for (_, j) in &self.matching.pairs{
            set.insert(*j);
        }
        let slots: Vec<usize> = (0..m).collect();
        let available_slots: Vec<usize> = slots
            .into_iter()
            .filter(|j| !set.contains(&j))
            .collect();
        return available_slots;
    }

    fn get_alternating_path(&self) -> AlternatingPath {
        let unmatched_bidders = self.get_unmatched_bidders();
        let unmatched_slots: Vec<usize> = (0..self.get_slots_num()).collect();
        let feasible_bidders: Vec<usize> = unmatched_bidders
            .iter()
            .filter(|&i| self.matching.utilities[*i] > 0.0)
            .map(|x| *x)
            .collect();
        const B: f64 = std::f64::MAX;
        let DUMMY_SLOT: usize = self.get_slots_num();
        let mut curr_min_path = AlternatingPath{
            path: Vec::new(),
            cost: B,
            final_edge: EdgeType::TERMINAL
        };
        for initial_bidder in feasible_bidders{
            let mut curr_path = AlternatingPath{
               path: Vec::new(),
               cost: B,
               final_edge: EdgeType::TERMINAL
            };
            curr_path.path.push(initial_bidder);
            let mut size = 0;
            let mut curr_node = initial_bidder;
            let mut visited_slots: HashSet<usize> = HashSet::new();
            let mut visited_bidders: HashSet<usize> = HashSet::new();
            visited_bidders.insert(curr_node);
            let mut end = false;
```

```
361             let mut is_viable = false;
362             while visited_bidders.len() + visited_slots.len() > size && !end {
363                 let mut candidate: Option<usize> = None;
364                 let mut candidate_cost = B;
365                 let mut edge_type = EdgeType::TERMINAL;
366                 let available_slots = unmatched_slots
367                     .iter()
368                     .filter(|&x| !visited_slots.contains(x));
369                 let curr_terminal = &self.graph.terminal_arr[curr_node];
370                 if curr_terminal.terminal_edge.is_some() {
371                     end = true;
372                     let max_cost = curr_terminal.terminal_edge.unwrap();
373                     if max_cost < candidate_cost{
374                         candidate = Some(DUMMY_SLOT);
375                         candidate_cost = max_cost;
376                         edge_type = EdgeType::TERMINAL;
377                     }
378                     is_viable = true;
379                 }
380                 for &j in available_slots {
381                     let node = &self.graph.graph[curr_node][j];
382                     if node.maximum_edge.is_some() {
383                         end = true;
384                         let max_cost = node.maximum_edge.unwrap();
385                         if max_cost < candidate_cost{
386                             candidate = Some(j);
387                             candidate_cost = max_cost;
388                             edge_type = EdgeType::MAXIMUM;
389                         }
390                         is_viable = true;
391                     }
392                     if node.reserve_price_edge.is_some() {
393                         end = true;
394                         let max_cost = node.reserve_price_edge.unwrap();
395                         if max_cost < candidate_cost{
396                             candidate = Some(j);
397                             candidate_cost = max_cost;
398                             edge_type = EdgeType::RESERVE;
399                         }
400                         is_viable = true;
401                     }
402                     if !end & node.forward_edge.is_some() {
403                         let max_cost = node.forward_edge.unwrap();
404                         if max_cost < candidate_cost{
405                             candidate = Some(j);
406                             candidate_cost = max_cost;
407                         }
408                         is_viable = false;
409                     }
410                 }
411                 if candidate.is_some(){
412                     visited_slots.insert(candidate.unwrap());
413                     size += 1;
414                     if curr_path.path.len() == 1{
```

```
                    curr_path.cost = candidate_cost;
                } else {
                    curr_path.cost += candidate_cost;
                }
                curr_path.final_edge = edge_type;
                curr_path.path.push(candidate.unwrap());
                curr_node = candidate.unwrap();
            }
            if !end && candidate.is_some(){
                let available_bidders = unmatched_bidders
                    .iter()
                    .filter(|x| !visited_bidders.contains(x));
                candidate = None;
                candidate_cost = B;
                for &i in available_bidders{
                    let backward_node = &self.graph.graph[curr_node][i];
                    if backward_node.backward_edge.is_some() {
                        let max_cost = backward_node.backward_edge.unwrap();
                        if max_cost < candidate_cost{
                            candidate = Some(i);
                            candidate_cost = max_cost;
                        }
                        is_viable = false;
                    }
                }
                if candidate.is_some(){
                    if candidate.is_some(){
                        visited_bidders.insert(candidate.unwrap());
                        size += 1;
                        curr_path.cost += candidate_cost;
                        curr_path.path.push(candidate.unwrap());
                        curr_node = candidate.unwrap();
                    }
                }
            }
        }

        if is_viable && curr_min_path.cost > curr_path.cost {
            curr_min_path = curr_path;
        }
    }
    return curr_min_path;
}

fn compute_distances(&self, start: usize) -> Distances {
    let n_slots = self.get_slots_num();
    let n_bidders = self.get_bidders_num();
    let B = 10.0;
    let mut slots_distances: Array1<f64> = Array::from_elem(n_slots,B);
    let mut bidders_distances: Array1<f64> = Array::from_elem(n_bidders,B);
    let mut visited_slots : HashSet<usize> = HashSet::new();
    let mut visited_bidders: HashSet<usize> = HashSet::new();
    let mut queue: VecDeque<(usize, f64, bool)> = VecDeque::new();
    queue.push_front((start, 0.0, true));
```

```
        while !queue.is_empty() {
            let (node, cost, is_bidder) = queue.pop_back().unwrap();
            if is_bidder {
                for j in 0..n_slots{
                    if !visited_slots.contains(&j){
                        match self.graph.graph[node][j].forward_edge {
                            None => (),
                            Some(x) => {
                                slots_distances[j] = x + cost;
                                queue.push_front((j, x + cost, false));
                                visited_slots.insert(j);
                            }
                        }
                    }
                }
            } else {
                for i in 0..n_bidders{
                    if !visited_bidders.contains(&i){
                        match self.graph.graph[i][node].backward_edge {
                            None => (),
                            Some(x) => {
                                bidders_distances[i] = x + cost;
                                queue.push_front((i, x + cost, true));
                                visited_bidders.insert(i);
                            }
                        }
                    }
                }
            }
        }
        bidders_distances[start] = 0.0;
        return Distances{
            slots_distances,
            bidders_distances
        }
    }

    fn fill_pairs(&self, path: &AlternatingPath, length: usize) -> Array1<(usize, usize)> {
        let mut pairs : Vec<(usize, usize)> = Vec::new();
        for i in (0..length).step_by(2) {
            let pair = (*path.path.get(i).unwrap(), *path.path.get(i+1).unwrap());
            pairs.push(pair);
        }
        return Array::from_vec(pairs);
    }

    fn update_matches(&self, new_pairs: Array1<(usize, usize)>) -> Array1<(usize, usize)> {
        let mut new_matches : Vec<(usize, usize)> = Vec::new();
        let mut bidders_set: HashSet<usize> = HashSet::with_capacity(new_pairs.len());
        let mut slots_set: HashSet<usize> = HashSet::with_capacity(new_pairs.len());

        for i in 0..new_pairs.len() {
            new_matches.push(new_pairs[i]);
            bidders_set.insert(new_pairs[i].0);
```

```
523             slots_set.insert(new_pairs[i].1);
524         }
525
526         for i in 0..self.matching.pairs.len(){
527             let curr_match = self.matching.pairs[i];
528             if !bidders_set.contains(&curr_match.0) && !slots_set.contains(&curr_match.1) {
529                 new_matches.push(curr_match);
530             }
531         }
532
533         return Array::from_vec(new_matches);
534     }
535
536     fn update_terminal_assignment(&self, path: &AlternatingPath) -> Option<Array1<(usize, usize)>>{
537         let new_pairs = self.fill_pairs(path, path.path.len() - 2);
538         return Some(self.update_matches(new_pairs));
539     }
540
541     fn update_maximum_assignment(&self, path: &AlternatingPath) -> Option<Array1<(usize, usize)>>{
542         let n = path.path.len();
543         if n >= 4 && path.path.get(n-1).unwrap() != path.path.get(n-3).unwrap() {
544             return None;
545         }
546         let new_pairs = self.fill_pairs(path, path.path.len() - 2);
547         return Some(self.update_matches(new_pairs));
548     }
549
550     fn update_reserve_assignment(&self, path: &AlternatingPath, updated_prices: &Array1<f64>) ->
          Option<Array1<(usize, usize)>> {
551         let last_item = path.path.last().unwrap();
552         let is_last_item_matched = self.matching.pairs.iter().filter(|(_, j)| j ==
              last_item).last().is_some();
553         if !is_last_item_matched {
554             let pairs = self.fill_pairs(path, path.path.len());
555             return Some(concatenate(ndarray::Axis(0),&[self.matching.pairs.view(),
                  pairs.view()]).unwrap());
556         } else if
               self.graph.graph[*path.path.get(path.path.len()-2).unwrap()][*last_item].reserve_price_edge.unwrap_or_else(||
               0.0) < updated_prices[*last_item]{
557             return None;
558         } else {
559             let new_pairs = self.fill_pairs(path, path.path.len() - 2);
560             return Some(self.update_matches(new_pairs));
561         }
562     }
563
564     fn update_assignments(&self, updated_prices: &Array1<f64>, path: &AlternatingPath) ->
          Option<Array1<(usize, usize)>> {
565         match path.final_edge {
566             EdgeType::TERMINAL => self.update_terminal_assignment(path),
567             EdgeType::MAXIMUM => self.update_maximum_assignment(path),
568             EdgeType::RESERVE => self.update_reserve_assignment(path, updated_prices)
569         }
570     }
```

```
571
572     pub fn stable_match(mut self) -> usize {
573         let n = self.get_bidders_num();
574         let k = self.get_slots_num();
575         const B : f64 = std::f64::MAX;
576         let mut iter : usize = 0;
577         let mut cost: f64 = 0.0;
578         while cost != B {
579             iter = iter + 1;
580             let path: AlternatingPath = self.get_alternating_path();
581             if path.path.len() < 2 || iter > n*(2*k+1){
582                 break;
583             }
584             cost = path.cost;
585             let max_f64 = |f1, f2| {
586                 if f1 >= f2 {
587                     return f1
588                 } else {
589                     return f2
590                 }
591             };
592
593             let first_item = path.path.get(0).unwrap_or_else(|| &(0 as usize));
594             let distances: Distances = self.compute_distances(*first_item);
595             let mut updated_utilities = self.matching.utilities.clone();
596             let mut updated_prices = self.matching.prices.clone();
597             for i in 0..n{
598                 updated_utilities[i] = updated_utilities[i] - max_f64(path.cost -
                        distances.bidders_distances[i], 0.0);
599             }
600             for j in 0..k{
601                 updated_prices[j] = updated_prices[j] + max_f64(path.cost -
                        distances.slots_distances[j], 0.0);
602             }
603             match path.final_edge {
604                 EdgeType::RESERVE => {
605                     let last_slot : usize = *path.path.get(path.path.len()-1).unwrap();
606                     let last_bidder : usize = *path.path.get(path.path.len()-2).unwrap();
607                     updated_prices[last_slot] = max_f64(updated_prices[last_slot],
                            self.reserve_prices[[last_bidder, last_slot]]);
608                 }
609                 _ => ()
610             }
611
612             let updated_assignments = self.update_assignments(&updated_prices, &path);
613             let new_assignments = match updated_assignments {
614                    None => self.matching.pairs.clone(),
615                    Some(x) => x
616             };
617             self.matching = Matching {
618                 utilities: updated_utilities,
619                 prices:   updated_prices,
620                 pairs:    new_assignments
621             };
```

```
            self.update_graph_edges();
        }
        self.write_results("rust-results2.txt".to_string(), iter);
        return iter;
    }


    fn write_results(self, path: String, iterations: usize) -> () {
        let mut file = fs::OpenOptions::new()
          .write(true)
          .append(true)
          .open(path)
          .unwrap();
        let n = self.get_bidders_num();
        let k = self.get_slots_num();
        let mut result : String =
            n.to_string() + "," +
            &k.to_string() + "," +
            &iterations.to_string() + ",";

        for (i,j) in self.matching.pairs {
            result = result +
                &i.to_string() + "-" + &j.to_string() + "/";
        }
        result = result + ",";
        for x in self.matching.utilities {
            result = result + &x.to_string() + "-";
        }
        result = result + ",";
        for x in self.matching.prices {
            result = result + &x.to_string() + "-";
        }
        result = result + "\n";
        file.write_all(result.as_bytes()).expect("Unable to write to file");
    }
}

fn main() {
    let vals_opt = Auction::read_file("./rust-dataset.txt".to_string()).unwrap();
    let auction = Auction::new(vals_opt.0, vals_opt.1, vals_opt.2);
    println!("Auction");
    println!("{:?}",auction.values);
    println!("{:?}",auction.reserve_prices);
    println!("{:?}",auction.max_prices);
    println!("{:?}",auction.graph);
    let iter = auction.stable_match();
    println!("Number of iterations: {:?}", iter);
}
```

Código B.2: Implementación do mecanismo xeral de subasta.

```
n=(3 5 8 10 20 50 100)
len=${#n[@]}
iter=100

```

```
for((j=0; j < $iter; j++))
do
    for((i=0; i < $len; i++))
    do
        echo "Execucion: ${j} - n: ${n[$i]}"
        python3 generate_auction.py ${n[$i]} ${n[$i]}
        cargo run
    done
done
```

Código B.3: Script usado para o primeiro experimento do mecanismo xeral de subastas.

```
n=30
iter=100

for((j=0; j < $iter; j++))
do
    for((i=0; i < 30; i++))
    do
        echo "Execucion: ${j} - k: $i"
        python3 generate_auction.py $n $i
        cargo run
    done
done
```

Código B.4: Script usado para o primeiro experimento do mecanismo xeral de subastas.

Bibliografía

[1] D. Gale e L. S. Shapley. (1962). *College admissions and the stability of marriage.* American Mathematical Monthly, 69: 9–15.

[2] Gui-Yuan Shi, Yi-Xiu Kong, Bo-Lun Chen, Guang-Hui Yuan e Rui-Jie Wu. (2018). *Instability in Stable Marriage Problem: Matching Unequally Numbered Men and Women.* Complexity, vol. 2018, Article ID 7409397, 5 pages, 2018.

[3] A. E. Roth. (1985). *The College Admissions Problem Is Not Equivalent to the Marriage Problem.* Journal of Economic Theory 36 (August 1985): 277–288.

[4] Manuel Eberl. (2016). *Fisher-Yates shuffle.* Arch. Formal Proofs 2016.

[5] Makoto Matsumoto e Takuji Nishimura. (1998). *Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator.* ACM Trans. Model. Comput. Simul. 8, 1 (Jan. 1998), 3–30.

[6] Gagan Aggarwal, Senthilmurugan Muthukrishnan, Dávid Pál e Martin Pál. (2008). *General Auction Mechanism for Search Advertising.* WWW'09 - Proceedings of the 18th International World Wide Web Conference.

[7] L. S. Shapley e M. Shubik. (1971). *The assignment game I: The core.* Int J Game Theory 1, 111–130.

[8] Edelman, Benjamin, Michael Ostrovsky e Michael Schwarz. (2007). *Internet Advertising and the Generalized Second-Price Auction: Selling Billions of Dollars Worth of Keywords.* American Economic Review, 97 (1): 242-259.

[9] Gagan Aggarwal, Jon Feldman, e S. Muthukrishnan. (2006). *Bidding to the Top: VCG and Equilibria of Position-Based Auctions.* Google Inc.

[10] Giorgos D. Birbas. *Generalized Second-Price Auctions under Advertisement Settings.* (2013). National Kapodistrian University of Athens.