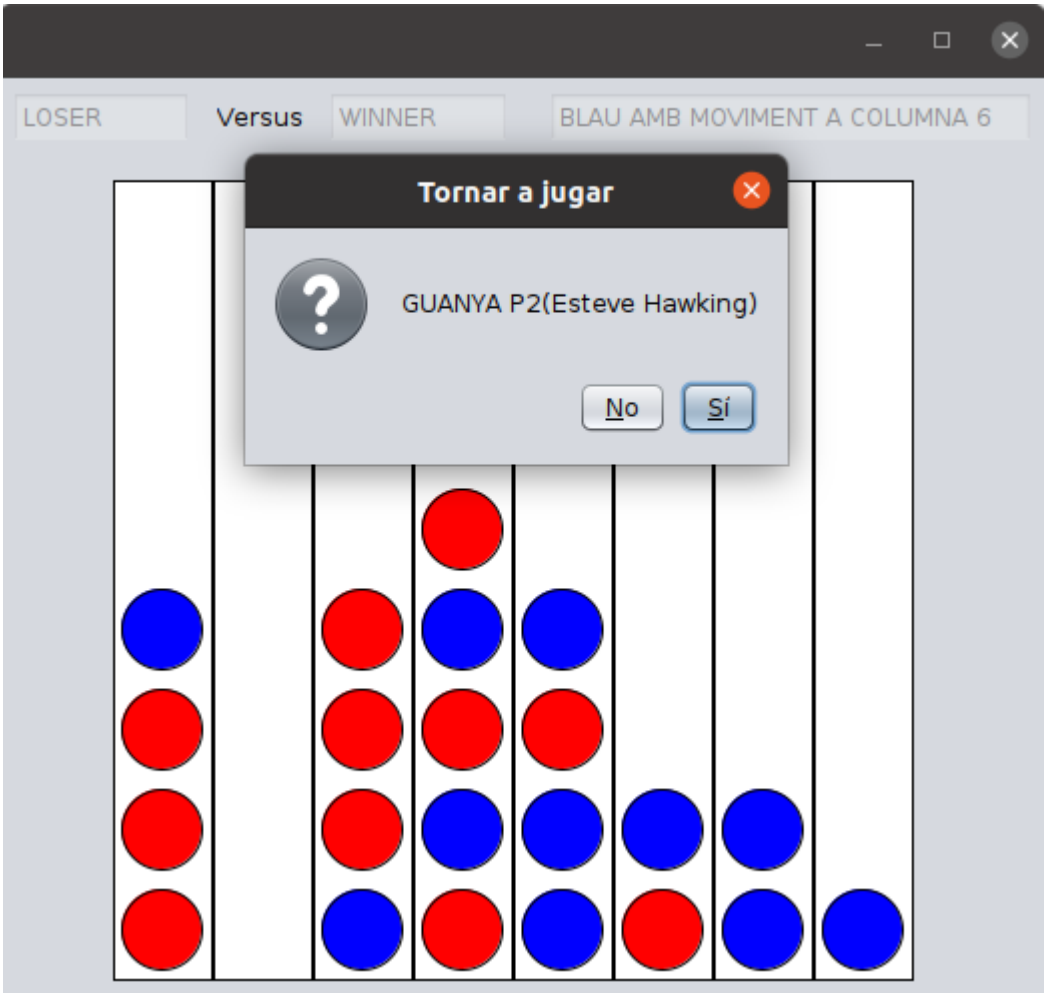
UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Projecte de Programació

Esteve Hawking

Documentació de la I.A. de Connecta-4



EPSEVG - Q5
Araceli Sandoval
Eduardo Pinto

Index

1. Algoritme min-max

L'algoritme min-max utilitzat está tret del pseudo-codi dels PDFs de classe, però amb petites modificacions per adaptar-lo al joc del connecta-4.

No explicarem gaire aquest algoritme, ja que ens centrarem més en explicar l'heuristica i el seu funcionament, pero farem una petita pinzellada amb aquests algoritme.

1.1. Funció Tria_moviment

A la funció `Tria_moviment` hem afegit la següent part:

```
Tauler aux = new Tauler(estat);
aux.afegeix(i, jugador);
if(aux.solucio(i, jugador)) {
       jugades_explorades++;
       jugades_explorades = 0;
       return i;
}
```

Aquesta part del codi lo que fa, és comprovar que si el moviment immediat es dona la victoria, returnem la `i`, que en aquest cas, l'`i` serà el moviment que es donará la victoria. Sense aquesta part del codi, érem capaços de trobar bones jugades per a un futur, però quan tenim la victoria amb un sol moviment, no la detectava i continuava buscant altres jugades cridant al `min_valor` i `max_valor`. Afegint aquest part, vam arreglar aquest error.

A més, com es pot veure al codi, utilitzem una variable anomenada `jugades_explorades` per comptar el número de jugades finals visitades, la qual cosa ens servirà per comparar més endavant com millora l'algoritme fent servir la poda alfa-beta.

1.2. Funció min_valor

La funció `min_valor` és molt similar a la del pseudo-codi del PDF amb la diferencia que hem d'afegir la variable `jugades_explorades` per comptar les jugades finales explorades i també hem afegit aquesta part dintre del bucle for:

```
Tauler aux = new Tauler(estat);
aux.afegeix(i, jugador);
if(aux.solucio(i, jugador)) {
      jugades_explorades++;
      return Integer.MIN_VALUE;
}
```

Seguint la mateixa lógica que la funció anterior, aquesta part mira si la jugada explorada es una solució, llavors en aquest cas, seria una jugada final i no faria falta crida a la següent funció que sería el max_valor. I en aquest cas retornem `Integer.MIN_VALUE` que seria l'equivalent a -∞.

1.3. Funció max_valor

Amb aquesta funció passa lo mateix que amb l'anterior, comprovem que el moviment fet sigui una solució o no.

```
Tauler aux = new Tauler(estat);
aux.afegeix(i, jugador);
if(aux.solucio(i, jugador)) {
      jugades_explorades++;
      return Integer.MAX_VALUE;
}
```

Tot lo altre, és exactament igual al pseudo-codi donat a classe.

2. Poda alfa-beta

Adaptar l'algoritme min-max per fer la poda alfa-beta es molt rápid i fácil, pero les millores en quant a velocitat d'execució que aporta son molt bones! És totalment recomanable utilitzar la poda alfa-beta.

Amb la poda alfa-beta el que fem és evitar visitar nodes que ens donaran resultats que sabem que no escollirem, com que sabem que no els escollirem, directament els podem podar i no visitarlos.

2.1. Funció Tria_moviment_alpha_beta

Es el mateix que la funció `Tria_moviment` anterior amb la diferència de que hem declarat dues noves variables:

```
int alpha = Integer.MIN_VALUE;
int beta = Integer.MAX_VALUE;
```

Aquestes dues variables s'afegiran a les altres dues funcions com a paràmetres.

2.2. Funció min_valor_alpha_beta

S'afegeixen aquestes línies a la funció `min_valor`:

```
if(valor <= alpha) {
      return valor;
}
beta = Math.min(valor, beta);
```

En cas de que `valor` sigui més petit o igual a `alpha`, retornem immediatament `valor` i deixem de explorar nodes fills. Si no, seguim l'execució i actualitzem el valor de `beta`.

2.3. Funció max_valor_alpha_beta

Fem el mateix que amb la funció `min_valor_alpha_beta` pero a l'inrevés.

```
if(beta <= valor) {
      return valor;
}
alpha = Math.max(valor, alpha);
```

2.4. Estudi de la incidència de la poda alfa-beta en el número de nodes explorats

Podem observar que l'algoritme min-max es més lent respecte a la poda alfa-beta. Pero volem observar de quina forma millora amb la poda alfa-beta. Per fer això, hem afegit algunes variables per comptar el número de nodes visitats i també el número de fulles visitades.

Compararem els resultats obtinguts amb l'algoritme min-max, la poda alfa-beta i també amb la poda alfa-beta ordenada.

L'algoritme min-max el que fa es recorrer tot l'arbre, és un cerca bastant brusca però ens asegura bons resultats. La poda alfa-beta ens dóna el mateix resultat que l'algoritme min-max però amb la diferència de que no visitem tots els nodes, ja que podem podar alguns resultats que no son necessaris visitar. I la poda alfa-beta ordenada, ens permeteix podar una part encara més gran, llavors ens estalviem visitar alguns nodes més.

Algoritme min-max	Poda alfa-beta	Poda alfa-beta ordenada
Jugades explorades totals: 120 475 392 Nodes explorats totals: 137 148 958	Jugades explorades totals: 3 852 488 Nodes explorats totals: 4 904 186	Jugades explorades totals: 1 703 773 Nodes explorats totals: 2 214 529

Podem observar que el número de nodes ha decrementat moltíssim entre l'algoritme min-max i la poda alfa-beta. I entre la poda alfa-beta i l'alfa-beta ordenada també ha disminuït considerablement el nombre de nodes visitats.

En aquest tres casos, ha sigut la mateixa partida desenvolupada amb les mateixes jugades. El resultat d'aquest partida es el que es troba en la *Figura 2.4.1.*

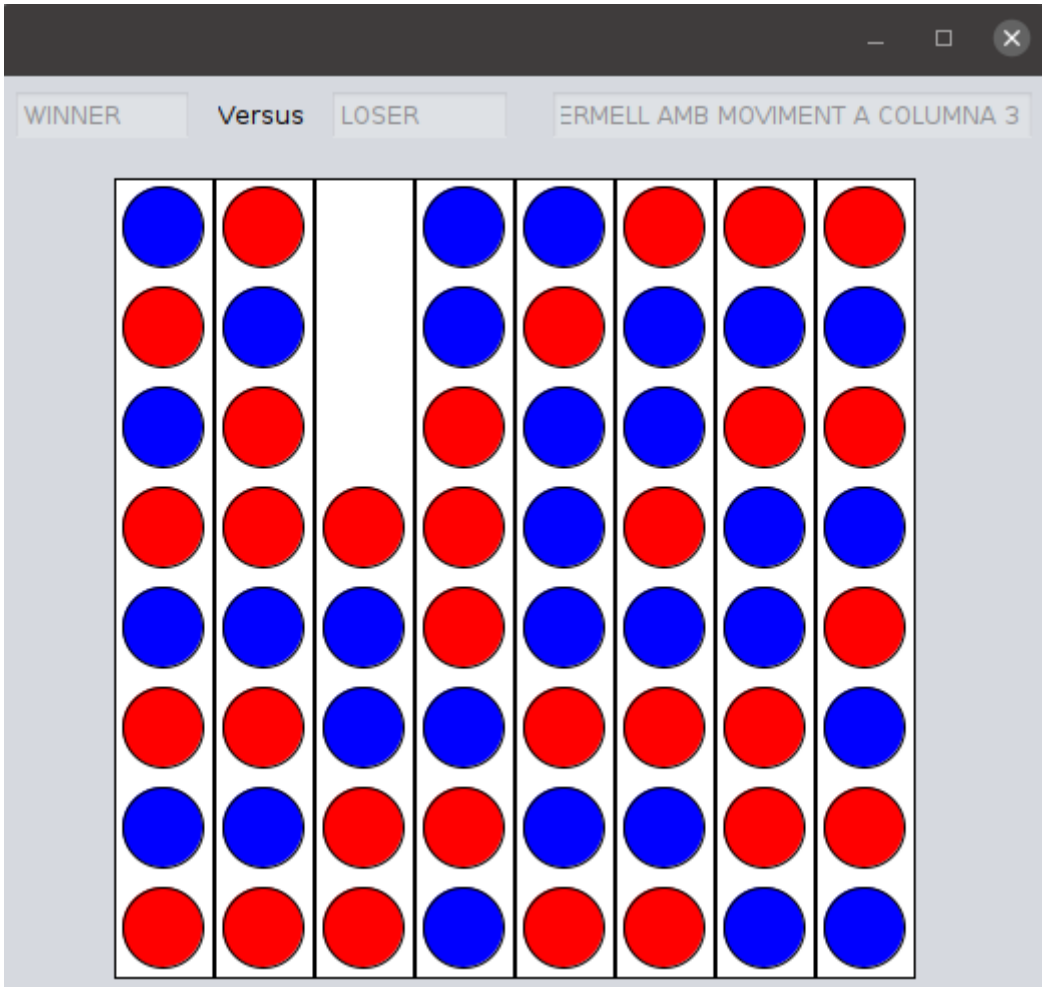


Figura 2.4.1. Partida intensa

Ara explicarem com hem fet la poda alfa-beta ordenada. Simplement hem començat a comprovar el tauler per la segona columna, ja que les probabilitats de trobar millors jugades solen estar per el centre del tauler, llavors si desde el principi trobem una millor jugada, podem podar una part encara mayor.

Com hem dit abans, hem començat per la segona columna. Llavors el que fem en aquest bucle, es mirar dues columnes més endavant, fent servir mòdul de 8, ja que si ens sortim del rang, mirarem les columnes inicial que no vam mirar al començar.

```
for (int i = 0; i < estat.getMida(); i++) {
      int millora = (i+1)%8;
      if (estat.movpossible(millora)) {
            ...
      }
}
```

També hem comprovat quin seria el millor valor per començar a estudiar les columnes, pero vam arribar que la començant per la segona columna obtindrem millors resultats.

Possibles valor per `millora = (i+x)%8;`

	`(i + 1) % 8`	`(i + 2) % 8`	`(i + 3) % 8`	`(i + 4) % 8`	`(i + 5) % 8`
Jugades explorades	1 703 773	2 786 746	5 482 058	9 493 043	9 981 487
Nodes explorats	2 214 529	3 471 870	6 636 981	10 716 911	11 819 632

Aquí podem observar que l'ordenació és millor si comencem per la segona columna, que es el que hem fet.

3. Heurística

Per fer la nostra heurística, el que hem és cridar quatre mètodes que s'encarreguen de comprovar i puntuar els fitxes seguides que hi han tant en horitzontal, com vertical.

```
int score = 0;
score += comprova_horitzontal(estat, jugador);
score += comprova_vertical(estat, jugador);
return score;
```

Cada mètode retorna una puntuació per heurística i sumen la puntuació total.

Teniem pensat fer les comprovacions en diagonal, pero hem vist que la nostra heurística era bastant forta, i hem decidir deixar aquest marge de millora per la IA, ja que ens es suficient per complir amb els nostres objectius.

3.1. Puntuació de l'heurística

Abans d'explicar com funciona cada mètode que s'utilitza en la funció per calcular l'heurística, volem explicar com puntuem, ja que aquest lógica s'aplica als quatre mètodes anteriors.

Per a que una fitxa pugui puntuar, aquesta no ha d'estar **bloquejada**. A que ens referim amb això? Bueno, ha de tenir espai suficient per fer quatre en ratlla. Si tenim 3 fitxes del mateix color consecutives, pero no hi ha espai fer quatre en ratlla, la puntuació d'aquestes serà un 0.

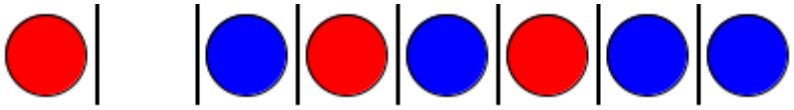


Figura 3.1.1. Exemple de fitxes bloquejades.

En l'exemple de la *Figura 3.1.1.* podem veure que les fitxes blaves tenen 2 fitxes consecutives, i sabem que tenir 2 consecutives és millor que només una, pero aquesta posició del tauler no ens dona possibilitat de fer 4 en ratlla de forma horitzontal, per tant, la puntuació serà un 0. I totes les altres fitxes també, perquè no hi ha espai suficient per fer 4

en ratlla. Llavors, el que farem serà calcular l'espai l'espai que ens queda lliure per saber si es pot fer 4 en ratlla o no. En cas de que si hi hagi espai, si que puntuarem la jugada.

També considerem important aclarar el que nosaltres considerem com fitxes consecutives. Per a nosaltres, una casella es consecutiva quan per enmig, no hi ha cap altres fitxa del color del rival. Observem la *Figura 3.1.2.* i *Figura 3.1.3.*, per a nosaltres és el mateix, ja que les dues ens dona la mateixa possibilitat de fer 4 en ratlla. Llavors en comptes de considerar la *Figura 3.1.2.* com dues fitxes consecutives i la *Figura 3.1.3.* com una fitxa consecutiva però dues vegades, considerem les dues jugades con 2 fitxes consecutives i retornem el mateix valor. Si fiquem una fitxa blava en qualsevol lloc podrem bloquejar fàcilment el 4 en ratlla en els dos casos, i com que no hi hauria espai per fer 4 en ratlla, la puntuació seria de 0.

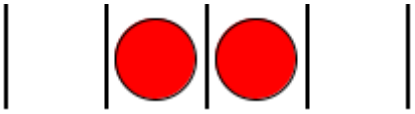


Figura 3.1.2. Exemple de 2 fitxes consecutives.

Figura 3.1.3. Això també ho considerem com 2 fitxes consecutives.

Encara que els exemplem siguin mirant el tauler de forma horitzontal, tot això també s'aplica quan ho comprovem de forma vertical.

Aquesta serà la nostra forma de puntuar l'estat de la partida:

	1 fitxa consecutiva	2 fitxes consecutives	3 o més fitxes consecutives	Victoria
Puntuació	**+ 1 punt**	**+ 3 punts**	**+ 7 punts**	**+ ∞ punts**

Tot això, suposant que tenim espai per fer 4 en ratlla, sino, la puntuació sumarà 0.

3.2. Funció comprova_horitzontal

Comprovem la matriu sencera i fem el recorregut per files. Cada cop que explorem una nova fila reiniciem les dades de cada fila. Farem servir la variable `puntuacio` per calcular la puntuació de cada fila, `seguides` per contar les fitxes d'un mateix color que tenim consecutives, `colorAnt` per recordar el color de l'última fitxa visitada, `lliure` per comptar els espais lliures que encara permeten ficar una fitxa que sumi a les `seguides`, i `lliuresConsecutives` per contar els espais lliures consecutius que hi han abans de que la fitxa canviï de color, ja que ens servirá per actualizar la variable `lliure` quan hi hagi canvi de color.

```
int score = 0;
for(int i = estat.getMida()-1; i > -1; i--) {
      int puntuacio = 0, seguides = 0, colorAnt = 0;
```

```
	int lliure = 0, lliuresConsecutives = 0;
	for(int j = 0; j < estat.getMida(); j++) {
		...
	}
}
```

Quan mirem la primera posició d'una fila, sempre `colorAnt` serà igual a 0, i será 0 fins que trobi la primera fitxa de qualsevol jugador. Llavors el que farem será actualizar el valor de `colorAnt`, un cop el valor sigui diferent a 0, no tornará a ser 0 fins que explori una nova fila. Quan trobi la primera fitxa de la fila, a `lliure` se li donarà el valor de `seguides`, que aquesta variable contindrà el valor dels espais lliures que ha trobat. I com que hem trobat un color nou, reiniciem el valor de `seguides`.

```
if(colorAnt == 0) {
	colorAnt = estat.getColor(i, j);
	if(colorAnt != 0) {
		lliure = seguides;
		seguides = 0;
	}
}
```

Cada cop que mirem la següent posició d'una fila, existeix tres possibilitats:

1) Hi ha una fitxa del mateix color
2) Hi ha una fitxa del color contrari
3) No hi ha cap fitxa

Llavors el que farem, será contemplar els 3 casos.

Si hi ha una fitxa del mateix color, augmenten el valor de `seguides` i sabem que no hi han més `lliuresConsecutives`, perquè l'espai està ocupat per una fitxa, per tant li donem valor de 0.

```
if(estat.getColor(i, j) == colorAnt) {
	seguides++;
	lliuresConsecutives = 0;
}
```

Si la fitxa es del color contrari, sumarem a la `puntuació` el valor de les fitxes seguides que hem trobat del `colorAnt`, sempre i quan hi hagi espai per fer 4 en ratlla. I sumarem aquesta puntuació a l'`score`, o en cas que el color sigui del `jugador` rival, li restarem la `puntuacio` a l'`score`.

També actualitzarem el valor de `colorAnt`, i com que hem trobar la primera fitxa d'un color, a `seguides` li donarem valor de 1. I reiniciem el valor de `lliuresConsecutives`.

```
else if (estat.getColor(i, j) == -colorAnt && colorAnt != 0) {
	if(seguides == 1 && lliure >= 3) {
		puntuacio = 1;
```

```
        }
        else if(seguides == 2 && lliure >= 2) {
                puntuacio = 3;
        }
        else if(seguides >= 3 && lliure >= 1){
                puntuacio = 7;
        }
        else puntuacio = 0;

        if(colorAnt != 0) {
                lliure = lliuresConsecutives;
        }

        if(colorAnt == jugador) {
                score += puntuacio;
        }
        else if(colorAnt == -jugador) {
                score -= puntuacio;
        }
        colorAnt = estat.getColor(i, j);
        seguides = 1;
        lliuresConsecutives = 0;
}
```

I l'últim cas seria trobar un espai on no hi hagi fitxa, en aquest cas l'únic que fem es aumentar el valor de `lliuresConsecutives` i `lliure`.

```
else {
        lliuresConsecutives++;
        lliure++;
}
```

Després d'haver visitat totes les posicions d'una fila, no ens hem d'oblidar de sumar els punts a l'score de les últimes fitxes trobades, ja que normalment les sumem quan hi ha canvi de color, però quan hem explorat tota la fila, no hi ha canvi de l'últim color, llavors afegim aquesta part quan finalitza el recorregut de la fila:

```
if(seguides == 1 && lliure >= 3) {
        puntuacio = 1;
}
else if(seguides == 2 && lliure >= 2) {
        puntuacio = 3;
}
else if(seguides >= 3 && lliure >= 1){
        puntuacio = 10;
}
else puntuacio = 0;
```

```
if(colorAnt == jugador) {
      score += puntuacio;
}
else if(colorAnt == -jugador) {
      score -= puntuacio;
}
```

3.3. Funció comprova_vertical

Aquesta funció segueix la mateixa lógica que el `comprova_horitzontal`. Fem un recorregut de la matriu per columnes, on reiniciem el valor d'algunes variables per cada nova columna visitada.

```
int score = 0;
for(int i = estat.getMida()-1; i > -1; i--) {
      int puntuacio = 0;
      int seguides = 0;
      int colorBuscat = 0;
      for(int j = 0; j < estat.getMida(); j++) {
            …
      }
}
```

Si la casella visitada no conté cap fitxa, és totalment imposible que més adalt i hagi un altra fitxa, llavors, sumen la puntuació de les fitxes trobades, si es que n'hi han, i canviem de columna.

```
if(estat.getColor(j, i) == 0) {
      j = estat.getMida();
      if(seguides == 1) {
            puntuacio = 1;
      }
      else if(seguides == 2) {
            puntuacio = 3;
      }
      else if(seguides >= 3){
            puntuacio = 10;
      }
      else puntuacio = 0;
      if(colorBuscat != 0) {
            if(colorBuscat == jugador) {
                  score += puntuacio;
            }
            else {
                  score -= puntuacio;
            }
      }
}
```

Si la casella visitada no está buida, només queden dues opcions:

1) Conté una fitxa del mateix color que busquem
2) Conté una fitxa del color contrari al que busquem

I això li sumen la possibilitat de que el `colorBuscat` sigui igual a 0, que això només passa en la primera iteració del bucle, ja que `colorBuscat` tindrá valor 0. En aquest cas, li assignarem el valor de la primera fitxa trobada.

En el cas de trobar una fitxa del color contrari, si estem en una fila superior a l'altura 4(sent 0 la primera altura), només queden l'altura 5, 6 i 7, per tant, no hi hauria espai per fer 4 en ratlla, i simplement ens estalviem mirar aquestes caselles i saltem a la següent columna.

```
else {
      if(j == 0) {
            colorBuscat = estat.getColor(j, i);
            seguides = 1;
      }
      else if(estat.getColor(j, j) == colorBuscat) {
            seguides++;
      }
      else if(estat.getColor(j, i) == -colorBuscat){
            if(j > 4) {
                  j = estat.getMida();
            }
            else {
                  colorBuscat = estat.getColor(j, i);
                  seguides = 1;
            }
      }
}
```

4. Resultats d'enfrontaments contra l'IA de Profe

Hem fet algunes comprovacions d'algunes partides de la nostra IA enfrontant-se contra l'IA anomeda `Profe`. Els resultats han sigut bons. `esteve_hawking` s'ha enfrontat contra les dues heurístiques de Profe amb profunditat 8, i jugant amb blaves i vermelles i ha sortit guanyador en tots els casos excepte en un. Hem de tenir en compte, que `esteve_hawking` té un marge de millora si es decideix implementar la comprovació en diagonal.

Vermell	**Blau**	**Guanyador**
esteve_hawking	Profe(8, false)	esteve_hawking
esteve_hawking	Profe(8, true)	esteve_hawking
Profe(8, false)	esteve_hawking	Profe
Profe(8, true)	esteve_hawking	esteve_hawking