Memòria pràctica 1

Mètodes de cerca local.

Sílvia Moià Canyelles - 43465685D
Martí Paredes Salom - 43460172Q

Index :

1. Cerca no informada per amplada.

Aquest algoritme en sí no ens va suposar gaires complicacions. El més destacable és que, com va ser el primer, ens va costar entendre com haviem de començar i com montar la classe estat des de zero. La major part de problemes que ens sorgiren foren perquè no agafavem la informació correctament, ens varem adonar a base de fer debuggins i veient que no es guardava la informació que pensàvem que estàvem guardant.

Per a resoldre l'algoritme tenim una llista d'estats oberts, una d'estats tancats i una altra d'accions.

Al principi la llista d'accions està buida, llavors entram dins la funció de cerca, on li passam l'estat actual i a partir d'aquí en va creant tots els fills i ficant-los a la llista d'oberts, de manera que es queda cercant sempre que hi ha algun element dins oberts. És a dir, no s'ha trobat la solució i, per tant, es segueixen creant fills. Un pic es troba la solució es surt del bucle i recorrem totes les accions (pare de cada estat) per saber el camí fet per arribar a la solució.

2. Algoritme d'A*.

Aquest algoritme també ens va sortir bastant ràpid ja una vegada varem arreglar tots els errors de la classe estat, només vàrem haver de canviar el mètode de cerca. La diferència d'aquest algoritme comparat amb l'algoritme de cerca per amplada és que ara sí tenim en compte el cost d'arribar fins un estat i per lo tant es troba un camí òptim.

La principal diferència és que la llista d'oberts és una cua amb prioritat, la qual és indexada per el pes de l'estat.

3. Algoritme MiniMax.

Per resoldre aquest algoritme vàrem haver de modificar bastant la classe estat de manera que tinguéssim en compte el torn i les coordenades de l'altre agent. Per fer-ho vàrem haver d'afegir un booleà per saber si era el torn de max o no (el de min) i així saber a partir de quin estat hem de crear els fills i modificar les coordenades del corresponent. També vàrem haver d'afegir una funció per avaluar cada estat. Aquesta mira la distància que estan min i max de la pizza i les resta respectivament: si min està a una distància menor, max està perdent, es retornarà un nombre negatiu. Si max està a una distància menor, max està guanyant, es retornarà un nombre positiu. A més retornam l'acció duta a terme. Aquí cal destacar

un problema, al principi retornàvem directament aquesta acció i aquesta era la que es duia a terme. Però les accions retornades aquí només tenien sentit al darrer nivell de l'arbre, ja que era la darrera acció abans de trobar la solució o haver arribat a la profunditat de l'arbre. Per aquest motiu, abans de retornar l'acció iteram els nodes i anam agafant les accions del pare, fins que arribam a la primera, llavors aquella és la que retornam.

Mentres cap dels dos agents arriba a una solució es va cridant la funció minimax, aquesta avalua. Cal destacar que el mètode avalua només retorna si és la meta o el límit de profunditat. Per tant, si no retorna res, simplement es creen els fills i es segueix fent el minimax per a cada un dels fills fins que s'arriba al màxim de profunditat o a la meta. Llavors avalua ja no retornarà None i maximitxarem o minimitzarem depenent del torn corresponent. Per maximitzar i minimitzar col·locam tots els elements (tuples de les puntuacions retornades per avalua + les accions corresponents) per ordre de mínima puntuació a màxima amb la funció sort de python. Així que a l'hora de maximitzar agafam el darrer element i a l'hora de minimitzar, el primer.

També mencionar que quan vàrem provar l'algoritme amb una profunditat diferent, les granotes anaven cap a la pizza però no se l'arribaven a menjar mai. Això ens passava perquè teniem definits malament les condicions per sortir del bucle recursiu i no acabava quan tocava. Teniem posat que quan trobava la solució s'aturàs, però trobar la solució a l'arbre no vol dir que l'haguem executada ja.

4. **Algoritme genètic.**

Per aquest algoritme varem estar bastant de temps ja que ens va costar començar a programar casi des de zero ja que ara no es tenen estats sinó individus. Hem plantejat que un individu és un conjunt d'accions. Per a la primera iteració del mètode de cerca cream als individus i a totes les iteracions següents el que feim es creuar als individus pares per a obtenir fills amb les característiques del dos pares. La quantitat de individus que cream i la longitud de la seqüència d'accions esta definit com atributs de la clase.

Com que tant la creació dels individus com el creuament es fa de manera aleatoria el més segur és que la seqüència d'accions no sigui valida, pot sortir-se del tauler o pot xocar amb una paret, per això hem creat una funció "corregir_cami" la qual fa exactament això. Quan es troba un dels casos anteriors el que fa és esborrar el camí que ve a continuació i retorna el conjunt d'accions que si és vàlid, si almenys tenim una acció afegim l'individu a la llista, sinó el descartam. A més de corregir el camí aquesta funció retorna la puntuació de fitness per la posició final i si l'individu ha arribat a la meta o no.

La funció de fitness l'hem considerada com la distància de Manhattan i la longitud de la llista d'accions, les dues components no tenen el mateix pes, hem considerat un 80% i un 20% respectivament. D'aquesta manera premiam als individus que tenen una longitud més curta i intenam evitar voltes innecessàries.

Finalment tenim la funció de creuament, la funció reb per paràmetre una llista d'individus, aquests són els K millors de cada generació, aquest nombre es pot canviar modificant l'atribut de clase "sel_elite", si la funció reb 10 individus la funció retorna 10 fills. Els creuaments es fan fent un xap de longitud aleatoria de l'individu més curt i juntam el primer tros del pare amb el primer tros de la mare i el mateix per a la segona part, d'aquesta manera de dos pares generam dos individus nous. Aquests seran avaluats si són vàlids o no la següent vegada que es cridi a la funció de cerca.

Per aquest algoritme varem estar una bona estona pensant perquè no se mos executaven de manera correcte les accions fins que ens varem adonar que ara les accions estan en ordre invers ja que al treure de la pila es fa de manera inversa. Per tant, era tan simple com invertir l'ordre de les accions.

Sabem que aquesta funció a vegades dona un error que indica que s'ha arribat al màxim de recursivitat, això és perquè al crear tots els individus de manera aleatoria pots necesitar fer moltes crides a la funció de cerca i python per defecte quan has cridat moltes vegades a una mateixa funció (700-900) t'atura el programa.

Per assegurar-nos que el problema era això i no un problema de codi hem provat el cas extrem on l'individu apareix a la posició (0,0), la pizza està a la posició (7,7) i els individus comencen amb una sola acció i ens funciona, a vegades ens surt l'error esmentat anteriorment però altres ens troba la solució sense problema. Si augmenta el nombre d'accions inicials a cinc o sis aquest problema desapareix.

5. **Comparacions de resultats.**

Per a mesurar el temps d'execució del programa he guardat el temps just abans de la crida al mètode cerca i un altre al moment que me retorna el valor, finalment els he restat i he obtingut la següent taula. Tots el resultats estan en segons.

	AMPLADA	**A***	**GENÈTIC**
Execució 1	0,2604	0,0019	0,0230
Execució 2	19,4864	0,0069	0,0004
Execució 3	0,1300	0,0045	0,0009
Execució 4	0,0030	0,0020	0,0049
Execució 5	0,0060	0,0029	0,0009
Execució 6	7,2465	0,0099	0,0055
Execució 7	2,5315	0,0045	0,0035
Execució 8	0,1550	0,0075	0,0015
Execució 9	0,1390	0,1000	0,0005
Execució 10	0,0915	0,0100	0,0010
Mitjana aritmètica	*0,1470*	*0,0057*	*0,0013*

Com podem observar el mètode més lent es l'amplada cosa que té sentit ja que desplega tot l'arbre per amplada fins trobar una solució, no ens retorna el camí més òptim ni en el menor temps possible. El següent algoritme es A*, aquest és el segon més ràpid i moltíssim més ràpid que l'amplada. Aquest ens troba el camí més curt en passes però no el més òptim en temps. Finalment tenim el genètic és l'òptim en temps però no el més ràpid en passes.

L'algoritme MiniMax no l'hem cronometrat ja que cada vegada que avalua quina opció fer crida al mètode de cerca, per lo tant el temps final de cerca no el sabem amb exactitud.