ANÀLISI I DISSENY D'ALGORISMES (CODI: 102783)
PRIMER PARCIAL: 11/01/2019

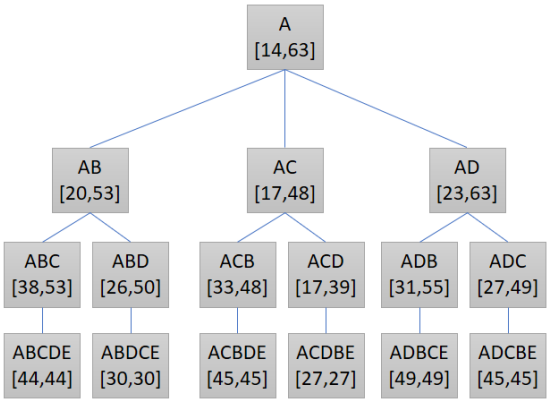
NOM I COGNOMS: ..... NIU: .....

IMPORTANT: Responeu en l'espai disponible a sota de cada pregunta

1. (3 punts) Són certes o falses, les següents afirmacions? Cada afirmació val 0.25 punts. Cada resposta incorrecta resta 0.25 punts.
   1) A l'algorisme de branch&bound, per obtenir una solució òptima, acaba l'execució quan treu una solució complerta de la cua de prioritat si l'heurística utilitzada és una cota inferior de la funció de cost i coincideix amb aquesta per solucions complertes.
   2) Es pot podar una solució parcial o complerta amb cota inferior més gran que la cota inferior d'una altra solució parcial.
   3) La cota superior ens permet podar les solucions que tinguin una cota inferior més gran.
   4) Les cotes inferior i superior convergeixen cap a la funció de cost conforme anem completant les solucions
   5) Podem construir una algorisme probabilístic q-correcte que executa n vegades un algorisme probabilístic p-correcte. Pel mateix n, q pujarà més ràpidament si l'algorisme p-correcte és un algorisme de Las Vegas que si és un algorisme de Monte Carlo.
   6) Un algorisme de Monte Carlo sempre dona una resposta correcta.
   7) La Programació Dinàmica permet resoldre problemes usualment en temps polinòmics.
   8) La Programació Dinàmica resol els problemes a partir de problemes més petits, però sempre els hem de recalcular tots.
   9) La Programació Dinàmica té en compte les diferents alternatives en la presa de decisions.
   10) La Programació Dinàmica necessita que els problemes es puguin dividir en subproblemes independents del mateix tipus i de mida més petita.
   11) Greedy sempre dóna una complexitat superior que Programació Dinàmica.
   12) Les mesures de complexitat són fonamentals per comparar diferents dissenys de solucions a un mateix problema.

	1	2	3	4	5	6	7	8	9	10	11	12
V	x		x	x	x		x		x			x
F		x				x		x		x	x	

2. (2.5 punts) Donat el següent arbre de l'espai de solucions amb les cotes inferior i superior de la funció de cost, indica a cada iteració de l'algorisme de branch & bound quin és el contingut de la cua de prioritat,



quins nodes es creen per ramificació i quins es poden. Considera que es poden podar els nodes que hi ha a la cua i els que es generen per ramificació. Els primers passos de l'algorisme serien:

A[14,63]	PODA: cap
AC[17,48],AB[20,53],AD[23,63]	PODA: cap

SOLUCIÓ:
CAMÍ ÒPTIM: ACDBE[27,27]

A[14,63]	PODA: cap
AC[17,48],AB[20,53],AD[23,63]	PODA: cap
ACD[17,39],AB[20,53],AD[23,63],ACB[33,48]	PODA: cap
AB[20,53],AD[23,63],ACDBE[27,27]	PODA: ACB[33,48]
AD[23,63],ABD[26,50], ACDBE[27,27]	PODA: ABC[38,53]
ABD[26,50], ACDBE[27,27]	PODA: ADB[31,55], ADC[27,49]
ACDBE[27,27]	PODA: ABDCE[30,30]

3. (1.5 punts) Planteja un algorisme probabilístic per solucionar un Sudoku. Com a guia es pot utilitzar l'algorisme probabilístic que soluciona el problema de les n-reines. Aquest algorisme es basa en generar aleatòriament configuracions del taulell que després s'aniran modificant per minimitzar una funció (algorisme de descens del gradient). En el cas del problema de les n-reines teníem:
   - **Representació d'una configuració**. La representació d'una configuració és un array de mida n que conté la fila de cada reina (de la 0 a la n-1). L'índex de l'array és la columna.
     Objectiu: representació senzilla i de fàcil manipulació.
     ```
     int posicions[n];
     ```
   - **Generació de la configuració inicial**. Col·locar aleatòriament n reines al taulell de forma que hi ha una reina a cada columna i la fila es selecciona aleatòriament. Les configuracions diferents que es poden generar per un taulell de nxn on es col·loquen n reines són: n^n configuracions.
     Objectiu: minimitzar el número de configuracions diferents que es poden generar aleatòriament.
     ```
     for (i=0; i<n; ++i) posicions[i]=srand() % 8;
     ```
   - **Funció a minimitzar**. Es calcula el número d'atacs entre reines per cada configuració.
     Objectiu: Indicar com es calcula la funció i que serveixi per decidir si una configuració es solució o no.
     ```
     int atacs(int posicions[n]) {
         int atacs=0;
         for (int i=0; i<n; ++i) {
             for (int j=i+1; j<n; ++j) {
                 if (posicions[i]==posicions[j]) ++atacs;
                 if (abs(posicions[i]-posicions[j])==j-i) ++atacs;
             }
         }
         return atacs;
     }
     ```
   - **Generació de les configuracions veïnes**. Donada una configuració, es generen totes les configuracions possibles on es canvia la fila d'una sola reina. Per tant, si tenim n reines, cada reina la podem moure a les n-1 posicions lliures que queden dintre de la seva columna (nomes canvia de fila). Com que tenim n reines, generarem (n-1)*n configuracions veïnes.
     Objectiu: Especificar com es generen les noves configuracions.
     ```
     for (int i=0; i<n;++i) {
         for (j=0; j<n; ++j) {
             if (posicions[i]!=j) {
                 novaConfiguracio=posicions;
                 novaConfiguracio[i]=j;
                 //Valorar la nova configuració generada
             }
         }
     }
     ```
   - **Selecció de la millor configuració veïna**. La configuració veïna amb la que continuarà l'algorisme serà la que tingui menys atacs.
     Objectiu: Indicar el criteri de selecció de la millor configuració veïna.
   - **Configuracions solució**. L'algorisme pararà quan arribi a una configuració o el número d'atacs entre reines sigui 0.
     Objectiu: Indicar la condició que ha de complir una configuració per ser solució.
   - **Generar un nou intent**. En el cas que les configuracions veïnes de la configuració actual no redueixin el número d'atacs entre reines, es generarà aleatòriament una nova configuració inicial.
     Objectiu: Indicar en quina situació s'ha de fer un nou intent.

   La vostra explicació de l'algorisme ha de contestar els mateixos punts anteriors. Considereu que el taulell del Sudoku es de 9x9 caselles, que en cada casella poden anar els dígits de l'1 al 9 i que hi ha caselles on s'ha fixat el dígit que contenen. L'objectiu és aconseguir una configuració on no es repeteixin dígits ni en la mateixa fila, ni en la mateixa columna, ni en el mateix quadrant de 3x3.
   Exemple de sudoku:

			2	4			3	5
				5	7			
2			6		8			
				8				6
	7	4				3		8
		8	7	9	1	4		
	5			6			8	
			4			6		
	2	9	8	7	3			1

4. (0.5 punts) En el problema de la motxilla, comparant backtracking i programació dinàmica, raona quin algorisme és més eficient.

   RESPOSTA: Usualment els algorismes de programació dinàmica són més eficients que els de backtracking pel fet que programació dinàmica fa un divide & conquer reaprofitant el càlcul de les solucions prèvies (i per tant es redueix a omplir una taula o un vector) mentre que backtracking explora tot l'arbre de solucions factibles.

   En el problema de la motxilla hem vist a classe que si comparem programació dinàmica amb un backtracking pur serà més eficient programació dinàmica, però en aquest cas també hem vist que si apliquem forward checking la complexitat del backtracking es redueix fins a millorar els temps de programació dinàmica. Això és degut al fet que amb la funció cota (que té en compte la capacitat lliure que li queda a la motxilla) no permet que es despleguin la majoria de les branques.
5. (2.5 punts) Volem omplir un camió amb caixes de mercaderies de 5 tipus diferents. De cada mercaderia tenim un nombre il·limitat de caixes i la capacitat del camió és de 10 Tones. Volem omplir el camió de manera que maximitzem el seu benefici sense superar la seva capacitat màxima. La taula que hi ha a continuació detalla els diferents pesos (en tones) i beneficis (en milers d'euros) que obtenim per cada un dels tipus de mercaderia que tenim.

Mercaderia	Pes	Benefici
M1	3	7
M2	2	5
M3	4	12
M4	7	23
M5	6	20

   Dissenyar la solució minimitzant la complexitat espacial i especificant
   (a) els valors de la taula de solucions en aquest cas
   (b) la funció que ens permet omplir cada cel·la de la taula.

   Quina serà la configuració de la solució final i el benefici màxim obtingut?

   SOLUCIÓ:

   En aquest cas trenquem el nostre problema en problemes dependents més petits del mateix tipus, per això trenquem la capacitat del camió en tones i les mercaderies a posar-les d'una en una. En el cas d'intentar posar la primera mercaderia, per cada capacitat parcial posarem tantes caixes com càpiguen. En el cas general d'una mercaderia quan ja n'hem mirat algunes, i per una capacitat parcial concreta, mirarem quin és el màxim benefici entre no posar cap caixa nova fins a posar-ne el màxim que ens hi càpiguen, passant per totes les possibilitats (si en caben 3 mirarem quin és el màxim benefici de posar-ne 0, 1, 2 o 3). Per saber el benefici de posar cada nova caixa d'una mercaderia concreta caldrà anar a mirar quin era el benefici que hi havia abans de posar-la.

   Per això, si w és la capacitat parcial del camió, m_i correspon a la mercaderia i, b_i el benefici de la mercaderia i, i p_i el pes de cada caixa de la mercaderia i, tenim la següent fórmula:

$$B(m_i, w) = \begin{cases} 0, & \text{si } w = 0; \\ b_i * \lfloor \frac{w}{p_i} \rfloor, & \text{si } i = 1; \\ B(m_{i-1}, w), & \text{si } w < p_i; \\ \max_{0 \leq j \leq \lfloor \frac{w}{p_i} \rfloor} \{B(m_{i-1}, w - p_i * j) + b_i * j\}, & \text{en la resta de casos.} \end{cases}$$

Si apliquem la fórmula en aquest cas concret tindríem una taula de màxims beneficis i una taula que va desant la configuració de caixes de mercaderies. Si volem reduir la complexitat espacial, observem que no necessitem desar tots els beneficis màxims ni totes les configuracions que anem trobant, sinó que com anem escribint d'esquerra a dreta i de dalt a baix, podem anar sobreescribint la taula. D'aquesta forma, la configuració final de la taula quedarà:

	0	1	2	3	4	5	6	7	8	9	10
Benefici total	0	0	5	7	12	12	20	23	25	28	32
M1	0	0	0	1	0	1	0	0	0	0	0
M2	0	0	1	0	0	1	0	0	1	1	0
M3	0	0	0	0	1	0	0	0	0	0	1
M4	0	0	0	0	0	0	0	1	0	1	0
M5	0	0	0	0	0	0	1	0	1	0	1

D'on podem veure que el benefici màxim en aquest cas serà de 32 per la configuració d'una caixa de la mercaderia M3 i una caixa de la mercaderia M5.