Proposta de solució al problema 1

(a) Sí. Per una banda la qüestió de si P = NP o no és un problema obert. Per una altra se sap que **CNF-SAT** és un problema NP-complet. Si existís un algorisme de cost polinòmic que decidís **CNF-SAT**, llavors tindríem que P = NP!

(b) Ho podem raonar per inducció sobre el nombre de voltes del **while**: es compleix abans d'entrar al **while** perquè a *propagated* només hi fiquem els literals de les clàusules unitàries. I a cada volta només fiquem a *propagated* un literal quan apareix en una clàusula on la resta de literals han de ser necessàriament falsos.

(c) Sí, el cost és polinòmic.

El primer **for** recorre totes les clàusules de la fórmula. Processar una clàusula té cost polinòmic en la mida de l'entrada. En efecte, el cost més gran és el de consultar i afegir en un *set*<**int**>, que és com a molt logarítmic en el seu nombre d'elements. Aquest serà com a molt $2n$, on n és el nombre de variables. El cost és polinòmic perquè n és més petit o igual a la mida de l'entrada.

Per altra banda, a cada volta el **while** propaga un nou literal. Per tant, com a molt es fan $2n$ voltes. El cost de cadascuna d'aquestes voltes és de nou polinòmic en la mida de l'entrada. A banda d'operacions de cost constant, copiar una clàusula té cost lineal en la mida de la clàusula (que està fitada superiorment per la mida de l'entrada), i propagar un literal en una clàusula (funció *propagate*) té cost logarítmic en la mida de la clàusula. Pel mateix raonament d'abans, el cos de **if** $(propagate(nC, l))$ té cost polinòmic en la mida de la fórmula en aquella volta, que no és més gran que l'original. I inserir la nova clàusula propagada té cost polinòmic en la mida de la fórmula en aquella volta. Finalment l'assignació $F = nF$ té cost polinòmic en la mida de l'entrada, perquè a cada volta la mida de les fórmules va decreixent.

(d) Sí.

Com que tota assignació que satisfaci la fórmula ha de fer certs els literals de *propagated*, quan es retorna **false** és perquè s'ha trobat una clàusula tal que o bé és buida o bé tots els seus literals han de ser falsos. Per tant, en aquest cas no pot existir cap assignació que satisfaci la fórmula. Així doncs, quan es retorna **false** la fórmula és insatisfactible.

(e) No.

La fórmula $\{\{\neg x_1, \neg x_2\}, \{\neg x_1, x_2\}, \{x_1, \neg x_2\}, \{x_1, x_2\}\}$ és insatisfactible. Però en cridar $sat(F)$, el primer **for** no troba cap clàusula buida ni empila res a la pila *pending*, i no s'entra al **while**. Per tant, *sat* retorna **true** (incorrectament).

Proposta de solució al problema 2

(a) Suposem que hi ha un isomorfisme $\rho : V_1 \to V_2$ entre $G_1 = (V_1, E_1)$ i $G_2 = (V_2, E_2)$, i sigui $u \in V_1$. Sigui $U_1 = \{\{u, v_1\}, \cdots, \{u, v_k\}\}$ el conjunt d'arestes de E_1 que incideixen en u. Llavors, per definició d'isomorfisme, $U_2 = \{\{\rho(u), \rho(v_1)\}, \cdots, \{\rho(u), \rho(v_k)\}\}$ és el conjunt d'arestes de E_2 que incideixen en $\rho(u)$. En particular, U_1 i U_2 tenen el mateix nombre d'elements, de forma que el grau de u en G_1 és el mateix que el grau de $\rho(u)$ en G_2.

(b) Existeix un únic isomorfisme ρ entre els dos grafs, definit per $\rho(0) = 1, \rho(1) = 6, \rho(2) = 0, \rho(3) = 3, \rho(4) = 5, \rho(5) = 2, \rho(6) = 4, \rho(7) = 7$.

(c) Una forma de completar el codi:

```
typedef vector<vector<bool>> Graph;

bool compatible(const Graph& G1, int x1, const Graph& G2, vector<int>& p) {
  for (int y1 = 0; y1 ≤ x1; ++y1)
    if (G1[y1][x1] ≠ G2[p[y1]][p[x1]]) return false;
  return true;
}

bool rec(int x1, const Graph& G1, const Graph& G2, vector<int>& p, vector<bool>& used) {
  if (x1 == G1.size()) return true;
  for (int x2 = 0; x2 < G1.size(); ++x2) {
    if (not used[x2]) {
      used[x2] = true;
      p[x1] = x2;
      if (compatible(G1, x1, G2, p) and rec(x1+1, G1, G2, p, used)) return true;
      used[x2] = false;
  } }
  return false;
}

bool iso(int n1, int m1, const Graph& G1, int n2, int m2, const Graph& G2) {
  if (n1 ≠ n2 or m1 ≠ m2) return false;
  vector<int> p(n1);
  vector<bool> used(n1, false);
  return rec(0, G1, G2, p, used);
}

void read(int& n, int& m, Graph& G) {
  cin >> n >> m;
  G = Graph(n, vector<bool>(n, false));
  for (int k = 0; k < m; ++k) {
    int x, y;
    cin >> x >> y;
    G[x][y] = G[y][x] = true;
} }
```

```
int main() {
  int n1, m1, n2, m2;
  Graph G1, G2;
  read(n1, m1, G1);
  read(n2, m2, G2);
  if ( iso(n1, m1, G1, n2, m2, G2)) cout << "true"  << endl;
  else                              cout << "false" << endl;
}
```

(d) Abans de cridar la funció de backtracking *rec*, a partir de les matrius d'adjacència es pot precalcular, per cada vèrtex de $G1$ i de $G2$ respectivament, quin és el seu grau. Aquesta informació es pot guardar en dos *vector*<**int**> $d1$ i $d2$, respectivament. Llavors, a la funció de backtracking *rec*, es descarta *x2* com a candidat a imatge de *x1* quan $d1[x1] \neq d2[x2]$.

Una altra manera de fer més eficient el programa és precalcular a més de *d1* i *d2* un diccionari *dict* amb claus naturals i valors llistes de vèrtexs de *G2*, de forma que *dict*$[d]$ és la llista dels vèrtexs de *G2* amb grau d. Llavors, a la funció de backtracking *rec*, quan es recorren els candidats a ser imatge de *x1*, en lloc de recórrer tots els vèrtexos de *G2*, només es recorren directament els vèrtexos de *dict*$[d1[x1]]$.

Proposta de solució al problema 3

(a) Els certificats són els subconjunts de C amb almenys k elements, els quals són disjunts dos a dos. Tenen mida polinòmica en l'entrada: si C té n elements, només cal un vector de n bits que indiqui quins elements de C s'agafen i quins no. A més, donat un subconjunt P de C, es pot comprovar en temps polinòmic que $|P| \geq k$, i que $S \cap S' = \emptyset$ per tot $S, S' \in P$.

(b) La reducció es defineix de la manera següent. Donat un graf $G = (V, E)$ i un natural k, prenem $U = E$, i per cada vèrtex $v \in V$, definim S_v com el conjunt d'arestes d'E que incideixen en v. Finalment prenem $C = \{S_v \mid v \in V\}$. El natural k és el mateix.

Demostrem a continuació que efectivament aquesta és una reducció polinòmica de **INDEPENDENT** a **EMPAQUETAMENT**.

En primer lloc, és clar que la reducció costa temps polinòmic.

Vegem ara que S és un conjunt independent de V si i només si $P = \{S_v \mid v \in S\}$ és un empaquetament; o equivalentment, S **no** és un conjunt independent de V si i només si $P = \{S_v \mid v \in S\}$ **no** és un empaquetament. En efecte, donats dos vèrtexs $u, v \in V$, tenim que $\{u, v\} \in E$ si i només si $S_u \cap S_v \neq \emptyset$.

Així doncs, hi ha un conjunt independent de V amb almenys k elements si i només si hi ha un empaquetament de C amb almenys k elements. Això conclou la demostració que la transformació donada és una reducció.

(c) Finalment, com que **INDEPENDENT** és un problema NP-complet, en particular NP-difícil, i hem trobat una reducció polinòmica de **INDEPENDENT** a **EMPAQUETAMENT**, tenim que **EMPAQUETAMENT** és NP-difícil. Tenint en compte a més que a l'apartat anterior hem vist que **EMPAQUETAMENT** pertany a la classe NP, podem concloure que **EMPAQUETAMENT** és també NP-complet.