Pràctica 2
MaxSAT

Ferran Aran Domingo
Oriol Agost Batalla

48253731F
48257094N

Oriol Agost Batalla 48257094N
Ferran Aran Domingo 48253731F

Índex

Oriol Agost Batalla 48257094N
Ferran Aran Domingo 48253731F

Introducció

En aquesta pràctica se'ns demana plasmar i demostrar els coneixements apresos a classe dels problemes de optimització MaxSAT

Problema 1

En aquest problema se'ns demana modelitzat un seguit de problemes relacionats amb grafs, com hem vist en classe, però modelitzant-los en codi.

Minimum vertex cover

El que busquem es treure el màxim nombre de vèrtex de manera que es mantinguin les arestes que hi havia en el graf original, entenent que una aresta "desapareix" quan els dos vertex que té deixen d'estar en el subconjunt.

Per a resoldre aquest problema usarem una clàusula de tipus "Weighted Partial Max Sat", ja que tenim clàusules *soft* i *hard* i a més a més aquestes tenen pesos.

Per això, el primer que fem es declarar una nova fórmula i hi afegim tantes variables com nodes tinguem.

Tot seguit posarem les clàusules *soft*. Recordem que aquestes seran per cada node, afegir-lo en positiu amb un pes de 1:

```
for n in nodes:
    formula.add_clause([-n], weight=1)
```

Ara, comencem amb les clàusules *hard*. Que seran, per totes les arestes, afegir els dos literals en positiu amb un pes de "pes màxim":

```
for e1, e2 in self.edges:
    v1, v2 = nodes[e1 - 1], nodes[e2 - 1]
    formula.add_clause([v1, v2], weight=wcnf.TOP_WEIGHT)
```

Després, resolem la fórmula passant-li al MaxSAT solver que tenim, obtenint el nombre de òptimes i el model. Finalment, retornem aquells nodes que són positius en el model, la qual cosa ens indica que són els que es satisfan, la solució:

Max clique

En aquest cas el que busquem és el major sub-graf complet, entenent per complet aquell graf en el que els nodes tenen totes les arestes possibles entre ells.

Com a clàusules *soft* en tindrem una per cada node amb pes d'1, com en l'apartat anterior, on la presència d'aquest en el sub-graf anirà determinada per una variable (en cas de ser certa el node estarà en el sub-graf, altrament no).

Tot seguit afegim les *hard*, on una clàusula *hard* tindrà com a literals aquelles dues variables que representin dos nodes els quals no comparteixen una aresta. Els literals apareixeran negats i el pes serà d'infinit.

Només contemplarem aquelles combinacions de parelles de vèrtex en les que la primera variable tingui un valor inferior a la segona, de manera que evitarem afegir clàusules duplicades que només difereixen en l'ordre dels literals.

Finalment li passem la formula resultant al solver per a que ens retorni el model resultant, del qual ens quedarem amb les variables que representin vèrtex que formen part del sub-graf (aquelles que siguin positives), com en l'apartat anterior, un cop més.

Max cut

El següent problema que se'ns planteja és el del *maximum cut*. En aquest se'ns demana que donat un graf amb els seus nodes i les arestes que aquests tenen entre ells, retornem una partició dels nodes, un subconjunt; tal que per cada aresta, aïllem cada un dels nodes que pertanyen a ella en una partició diferent.

En aquesta, entenem que en el model de la solució, els nodes en positiu representaran formar part d'una partició i els nodes en negatiu, part de l'altra partició.

En aquest problema, igual que ens els anteriors, primer creem una fórmula del tipus *Weighted Partial Max Sat* i després, afegim tantes variables en aquesta com nombre de nodes tenim.

A continuació, declararem les clàusules *soft*, que si recordem són l'únic tipus de clàusules que hi ha en aquest problema. Per cada aresta que tenim en el nostre node, generarem dues clàusules, una en la que estan els dos literals en positiu amb pes 1 i una en que es el contrari, amb els dos literals en negatiu i un pes d'1 també. D'aquesta manera, estarem especificant que volem que els nodes vagin a particions diferents, com hem explicat prèviament.

A més a més, cal destacar que es posen com clàusules *soft* i no com clàusules *hard* ja que no sempre se'ns donarà la possibilitat de complir-les totes, com mostrem a continuació:

```
for e1, e2 in self.edges:
    v1, v2 = nodes[e1 - 1], nodes[e2 - 1]
    formula.add_clause([v1, v2], weight=1)
    formula.add_clause([-v1, -v2], weight=1)
```

Finalment, calculem la solució i retornem aquells nodes en positiu com en els altres casos; que aquí seran aquells que pertanyin tots a una partició.

Problema 2

En aquest segon problema, se'ns demana implementar el procediment to_13wpm(). El primer que fem es declara una nova formula, de tipus WCNF.

A continuació, generarem tantes variables en la nova fórmula com nombre de variables tenim en la formula vella, ja que sabem que serà el nombre mínim de variables que tindrà la nova fórmula.

Clàusules *hard*

Per a fer la reformulació de les clàusules *hard* el que farem serà cridar a una altra funció, que s'encarregarà de gestionar aquest tipus de clàusules, ja que ens ajudarà després perquè farem una crida recursiva en ella.

Primerament, desempaquetarem la primera tongada de literals, i mirarem la seva longitud, com s'ha mencionat. Si aquesta es menor que tres, simplement posarem en la nostra nova formula la que ja teníem, però repetint algun literal fins que n'hi hagi tres, la qual cosa ens assegurarà de que no es canvia el resultat i que ara la longitud si que sigui tres; i el pes al màxim.

Si altrament la longitud es 3, el que farem serà posar aquesta a les clàusules *hard* de la nova fórmula, amb el pes al màxim.

D'altre manera, la qual cosa voldrà dir que la longitud de la clàusula passada es major que tres, el que farem serà generar una nova variable que ens servirà per enllaçar el conjunt de clàusules d'ara, amb el que ens quedarà fer després.

A continuació, afegim una nova clàusula amb els dos primers literals de la clàusula que estem tractant i la variable nova generada en positiu.

Finalment, farem una crida recursiva sobre aquesta funció, passant-li una nova clàusula que serà idèntica a l'anterior però com a primer literal tindrà la nova variable que tot just hem creat en negatiu, i se li hauran eliminat els dos primers literals (ja que ja estan introduïts en la clàusula anterior).

Clàusules *soft*

Per a fer la transformació de les clàusules *soft*, el que fem és crear una nova variable, que ens servirà per representar els literals de la *soft* que ja teníem i introduir-la negada en una nova clàusula *soft* de la fórmula que estem generant, amb el pes que ens ha entrat.

A continuació, cridarem a la funció que ens transforma les clàusules *hard*, passant-li com a paràmetre una clàusula formada per el literal que acabem d'introduir en positiu i la pròpia clàusula *soft* que ens havia entrat (ja que tal com hem vist en les sessions de teoria, aquesta clàusula es l'únic resultat significatiu de la conversió).

```
self.to_13wpm_hard(formula13, [[-formula13.num_vars] + clause[2:]])
```

Problema 3

Se'ns demana que elaborem un programa capaç de resoldre el problema *Software Package Upgrades* mitjançant la seva conversió a WPM. Per tal d'assolir l'objectiu, creem una classe la qual representarà el problema, i estarà composada per una sèrie de mètodes i paràmetres que aniran resolent-lo.

"Parsejar" el fitxer d'entrada

D'obtenir i tractar els paràmetres d'entrada se'n encarrega el mètode *parse_args()*, que no és part de la classe *SPU* (a diferència de la resta), de forma que li retorna al *main*, que s'encarrega d'instanciar un objecte *SPU* i passar-li els arguments "parsejats". Hem implementat aquest amb l'ajuda del mòdul *argparse*.

Traduir el problema a una formula MaxSAT i resoldre la formula amb el solver

Per resoldre aquesta part, partim de que ja tenim una instància creada de *SPU* i en cridem el mètode *generate_formula_and_parse()* que s'encarrega de llegir el fitxer com un *stream* i passar-li al mètode *parse_and_solve()* que comprova quin és el primer caràcter de cadascuna de les línies que composen *l'stream*. D'aquesta manera, discernim entre els diferents tipus d'input (declaració de paquet, dependència o conflicte) i altres caràcters com comentaris o bé la primera de les línies.

Un cop distingit el tipus d'input es crida a la funció que s'encarrega de manejar-lo. Tot seguit es comprova que el nombre de paquets llegits quadra amb el nombre de paquets que diu que hi ha d'haver a la primera línia del fitxer d'input.

Finalment es retorna la solució obtinguda amb el solver.

Tractar les línies d'input

Per tal de manipular els paquets, la classe *SPU* disposarà de dos diccionaris, un que tindrà com a clau els noms dels paquets i com a valor el número de la variable associada a aquest, i un altre diccionari que farà l'invers al primer.

Paquets

Per cadascuna de les línies que representin paquets a instal·lar, calcularem l'id que li pertoca (valor que tindrà la variable que representa si aquest s'instal·larà o no) i posarem els valors (tant el nom del paquet com l'id) en els corresponents diccionaris.

A continuació afegirem una nova variable a la fórmula i una nova clàusula soft que senzillament tindrà un literal (variable associada al paquet en positiu) i un pes d'1:

```
self.formula.new_var()  # Package 1 == Var 1 of the formula13
self.formula.add_clause([self.packages[package]], weight=1)
```

Dependències

Quan tractem amb una dependència el primer que farem es mirar si el paquet depenent ha esta declarat (es a dir, està en el diccionari), un cop feta la comprovació passarem a crear la clàusula que representarà les dependències.

Per a fer-ho, recorrem les dependències de la línia d'input i les anem afegint a una llista, un cop obtingudes totes, afegirem una nova clàusula formada per la variable associada al paquet depenent negada, i a continuació les variables associades a les dependències sense negar. La clàusula serà *hard*.

```
for dependency in dependencies[1:]:
    self.check_package(dependencies[1])
    new_clause += [self.packages[dependency]]

dependent = self.packages[dependencies[0]]
self.formula.add_clause([-dependent] + new_clause, weight=wcnf.TOP_WEIGHT)
```

Conflictes

En aquesta part, el primer que fem és mirar si aquest conflicte està en el diccionari amb la funció *check_package()*, i de no ser així, aquesta imprimirà el error per pantalla i acabarà l'execució del programa.

Altrament, obtindrem el paquet que "crea" el conflicte i després, el paquet conflictiu.

Ara, afegirem una clausula en la nostra fórmula , amb el número de paquet que crea el conflicte en negatiu i el número del paquet conflictiu també; amb el pes al màxim, com a clàusula *hard*. Això ho fem per a representar que el que no vull que passi és que estiguin els dos instal·lats a l'hora.

Interpretar la solució del maxSAT

Per a interpretar aquesta solució i imprimir els resultats per pantalla, el primer que fem és imprimir "o <nom de paquets no instal·lats>", que és el nombre de paquets òptims trobat en calcular la solució, com s'ha mencionat prèviament.

A continuació, se'ns demanava que imprimíssim també "v ", seguit de tots els paquets que no han pogut ser instal·lats en ordre alfabètic i separats per espais.

Per a fer això, el primer que fem es recórrer el model de solució que se'ns ha donat i si el paquet apareix en negatiu (paquet no instal·lat), afegim en una llista el nom del paquet. Aquest, l'obtenim buscant el valor que tenim en el diccionari "package_ids" per la clau corresponent al valor absolut del nombre de paquet en el model.

```
for package in model:
    if package < 0:
        installed_packages += [self.package_ids[abs(package)]]
```

A continuació, ordenem la llista amb:

```
installed_packages.sort()
```

I finalment, imprimim els paquets

```
for package in installed_packages:
    print(" " + package, end="")
print()
```

Problema 4

En aquest últim problema se'ns presentava una millora opcional al problema anterior. Se'ns demanava que afegíssim un nou paràmetre possible a la crida, "--validate" i en aquest cas, a més de realitzar tot el procediment anterior, s'havia de validar també la solució obtinguda.

El primer que hem fet és modificar la funció *parse_args()*, que s'ocupa de "parsejar" els paràmetres, afegint aquest paràmetre mencionat com una opció amb la línia:

```
parser.add_argument("--validate", action="store_true",
                    help="Specify if the user wants to check the solution.")
```

A continuació, hem declarat dos diccionaris, en els que emmagatzemarem les dependències i els conflictes de cada paquet, afegint-los en aquests quan "parsejem" la dependència o conflicte pertinent de l'arxiu d'entrada.

El diccionari de dependències serà de l'estil: {'myapp': [[gcc][python2, python3]]}

En aquest cas, això representa que el programa myapp té dependència de gcc $\wedge$ (python2 $\vee$ python3).

El diccionari de conflictes serà de l'estil: {'c': [myapp][rust]}

Cal destacar que en aquest cas mai se'ns donarà el cas en que un programa tingui conflicte en un o un altre, sempre seran amb un i un altre.

Llavors, el que fem és, en acabar la execució del *main*, mirar si *args.validate* es cert i si es així, cridar a la funció *check_solution()*

```
if args.validate:
    spu_problem.check_solution(solution)
```

En aquesta funció el que fem es desempaquetar la solució, i declarar un *bool*, que es dirà *correct*. Si aquest és cert, voldrà dir que la solució és correcta i altrament, que és incorrecta.

Llavors, el que fem és mirar per cada paquet de la solució positiu (paquet instal·lat), i si la nostra funció que comprova dependències o la que comprova paquets retorna fals, canviem el *bool* a *false* i

fem un break, ja que si trobem un cas en el que no es compleixi ja podem assegurar que la solució no serà correcta. Altrament, en acabar, imprimim cert.

ok_dependencies()

Aquesta funció mira si donat un paquet, es compleixen totes les seves dependències.

Per a fer-ho, primer declarem una variable que ens servirà de comptador.

Després, mirem si el paquet passat per paràmetre no té cap dependència i si és així, retornem cert, ja que segur que es compleixen les seves dependències al no tenir-ne cap.

Altrament, per cada dependència en la posició corresponent del diccionari de dependències per aquest paquet, i per cada ítem en aquesta dependència, mirem si aquest es troba en el model de la solució i es positiu (està instal·lat). Si és així, sumem 1 al comptador i fem un break, per a passar a la següent dependència

Finalment, retornem cert si el comptador és igual a la longitud dels valors del diccionari de dependències d'aquest paquet:

```
sat_dependencies = 0
if self.dependencies.get(package) is None:
    return True
for dependance in self.dependencies[package]:
    for item in dependance:
        if item in model and item > 0:
            sat_dependencies += 1
            break
return sat_dependencies == len(self.dependencies[package])
```

ok_conflicts()

Aquesta funció mira si donat un paquet, es compleixen totes els seus conflictes.

Per a fer-ho, mirem si existeix algun conflicte per a aquest paquet en qüestió i llavors, mirem per cada conflicte si aquest està en el model positiu (instal·lat). Si fos així, retornem fals (conflicte instal·lat).

Altrament, un cop hagin acabat totes les iteracions, retornem cert

```
if self.conflicts.get(package) is not None:
    for conflict in self.conflicts[package]:
        if conflict in model and conflict > 0:
            return False
return True
```