Albert Espín Román

A continuació es mostren els comentaris sobre la pràctica d'enumeratius, que també apareixen a "enumeratius_Albert_Espin_Roman_C.py", però aquí són més llegibles.

Comentaris sobre l'elaboració de la pràctica:

Introducció:

En la realització d'aquesta pràctica s'ha experimentat amb moltes estratègies i metodologies en l'intent continu de fer el més eficient possible la resolució del problema NP-Complet de l'aparellament dels nodes d'un graf (que poden representar contactes o moltes altres coses) mitjançant ramificació i poda. Aquest arxiu conté el programa amb les heurístiques i sistemes que han demostrat ser més eficients. No obstant això, a continuació s'explica de forma general, en diferents àmbits o apartats, les diferents metodologies amb què s'ha experimentat i perquè unes han donat millors resultats que les altres. Al fitxer "heuristiquesSimples.py" s'inclou una versió menys eficient del programa amb algunes de les metodologies que a continuació es comenta que es van substituir per altres.

Organització del programa com un solucionador de problemes de ramificació i poda:

Un dels principis insígnia de la programació estructurada i dels hàbits recomanables a la programació orientada a objectes és la modularitat i capacitat de reutilització de blocs de codi, sovint encapsulats en la forma de funcions.

A l'hora de resoldre aquesta pràctica, s'ha volgut que la funció central per a solucionar el problema amb ramificació i poda (la funció "branchAndBound") fos el més reutilitzable possible en altres exercicis de ramificació i poda independentment que la temàtica o paràmetres d'altres problemes fossin semblants o molt diferents. Per això fins i tot operacions que són molt senzilles en aquest problema (com mirar si una solució és completa, ja que aquí només cal veure si hi ha tantes parelles a la solució candidata com la meitat de nodes del graf), però que podrien ser més complexes en altres problemes, s'han encapsulat dins de funcions pròpies. D'aquesta manera, per a un altre problema, pràcticament es pot reutilitzar el mètode central sense canvis, i només cal canviar la implementació de les funcions allí cridades, com la funció de cota, que pot variar d'un problema a un altre.

També s'ha respectat l'ús del concepte solució com a pilar central de l'algorisme, ja que si una cosa és comú als diferents problemes que poden ser solucionats per ramificació i poda és que

es pot abstraure un concepte de solució (parcial o completa) i treballar amb un arbre de solucions que l'algorisme va ramificant, podant i analitzant.

Format de les solucions:

Primerament, convé indicar el format de les solucions (parcials o completes) amb què treballa l'arbre de solucions al mètode de ramificació i poda. Les solucions es basen en l'estructura de dades diccionari, i les dades que inclouen són la llista de parelles que la solució porta considerades (guardades com a tuples de dos nodes, amb el de menor nombre davant), el rebuig o pes total acumulat per aquestes parelles i la cota inferior de la solució.

Estructura i construcció de l'arbre de solucions:

L'arbre de solucions té per element arrel un únic node, no una parella (l'algorisme està preparat per discernir, als seus diferents nivells, si s'està al cas del node inicial o no). Aquest node inicial s'ha considerat escollir-lo de moltes formes: el de menor grau del graf (menys veïns), el de major grau, un dels dos que té l'aresta de més pes o el primer en l'estructura en què el graf guarda els nodes. Experimentalment es van provar totes aquestes opcions, i la que va donar millors resultats d'eficiència va ser el de menor grau, que permet limitar la ramificació al primer nivell del graf.

A partir del següent nivell, totes les solucions estan formades per parelles, no per nodes individuals. El nivell 1 (si considerem que el 0 és el de l'arrel de l'arbre de solucions) està format per solucions amb les diferents parelles del node inicial.

Un factor altament determinant per a l'eficiència ha estat la forma d'estructurar els nivells posteriors al nivell 1. Inicialment es va considerar que cada nivell a partir del 2 fos format per solucions amb les parelles ja acumulades més cada una de les parelles que encara es podien considerar (perquè els seus nodes no havien estat aparellats encara). Això però, provoca una ramificació molt gran, i, tot i que el sistema de cota inferior sigui força eficient, la poda no aconsegueix que després de la seva acció quedin tan poques solucions candidates com amb altres opcions.

Posteriorment, així, es va considerar que a partir del nivell 2 de l'arbre de solucions, les solucions consistissin en les parelles ja realitzades més les diferents parelles encara possibles de realitzar fent servir un node fixat: això crea moltes més branques i no descarta cap solució, ja que tot node encara no aparellat haurà de ser emparellat en algun moment; posar en oposició a un mateix nivell un grup comú de parelles més una extra però que té un node nou comú a totes les solucions del nivell facilita molt una aplicació efectiva de la selecció per cota (s'examina primer la solució amb menor cota inferior). La forma d'escollir aquest node comú ha estat objecte

d'experiments, però de les diferents opcions considerades (com que fos el de més veïns) va demostrar ser més eficient escollir el de menor grau encara no aparellat: permet minimitzar la ramificació de l'arbre de solucions.

Gestió i priorització de les solucions candidates:

És la funció de ramificació i poda ("branchAndBound") qui s'encarrega d'escollir l'ordre en què les noves solucions que es consideren al llarg de l'execució són explorades i ampliades amb solucions filles. Això s'ha provat a estructurar de tres maneres diferents.

La primera va ser de forma recursiva: d'una solució (si no és completa) es generen solucions filles, i es van explorant recursivament per ordre de prioritat, si no són descartades abans (podades), i quan tota una branca s'ha acabat d'explorar, es torna enrere (backtracking), i es continuen explorant solucions de nivells més alts, els fills de les quals tornaran a guanyar profunditat amb nivells fills, fins a trobar la solució òptima. El problema d'aquesta metodologia és que només és prioritza per nivells, no per totes les solucions encara no descartades de l'arbre de solucions: si, per exemple, un nivell té dos solucions candidates de cota inferior 10 i 20, el mètode explorarà primer la de 10. Si aquesta té 2 fills viables (no descartats) de cotes inferiors de 30 i 40, idealment voldríem tornar enrere i analitzar abans la solució del nivell de la solució mare que tenia cota inferior 10, però aquesta implementació exploraria primer la de 30 i els seus fills, la de 40 i els seus fills, i finalment acabaria tornant al nivell en què podria analitzar la de 10 i els seus fills.

La forma més senzilla de posar solució a aquest problema i assolir una priorització global a l'arbre de solucions en tot moment és fer servir una cua de prioritats en comptes de la metodologia recursiva descrita al paràgraf anterior, de tal manera que mentre la cua de prioritats no és buida (i partir de la solució arrel de l'arbre), se selecciona la solució candidata amb menor cota inferior, es generen els fills, que s'analitzen: si un fill és complet i millor que l'òptim actual passa a ser el nou òptim actual i la seva cota inferior la cota superior (i s'eliminen de la cua les solucions candidates amb cota inferior no menor a la superior, ja que ja no són viables per ser òptimes), i si no és complet però sí viable com a candidat (amb cota inferior inferior a la superior i completable) és afegit a la cua, tot plegat fins que aquesta es buida i es retorna la millor solució trobada.

Fins ara, però, s'ha esmentat una cua de prioritats però no l'estructura de dades exacta amb què s'implementa. Inicialment (i això constitueix la segona opció considerada per gestionar les solucions, recordem que la primera era la recursiva) es va fer servir una llista com a cua de prioritats (sent la cota inferior de les solucions la seva prioritat a la cua), però el cost d'eliminar elements (amb la consegüent reestructuració de la llista) fent servir aquesta llista penalitzava l'eficiència fins al punt que la opció recursiva provava ser més eficient per als grafs provats.

La solució va ser implementar la cua de prioritats com un heap binari, que es reestructura en eliminar els elements amb complexitat logarítmica, no pas lineal, de manera que eliminava la penalització patida amb la implementació com a llista i provava ser com a mínim tan eficient com la opció recursiva, i conceptualment més encertada, de manera que es va decidir utilitzar aquesta tercera opció, la cua com a heap, en la versió definitiva del programa, fent servir la llibreria "heapq" per les operacions bàsiques d'afegir i extreure elements del heap sense haver d'utilitzar una classe heap pròpia.

Comprovacions inicials sobre la possibilitat de trobar una solució:

Abans de cap altra cosa, comprovem que el graf sobre el qual aparellar els nodes té un nombre parell de nodes: altrament serà impossible fer un matching perfecte, aquell que inclou tots els nodes del graf un sol cop, òbviament només amb parelles que existeixen sota la forma d'arestes del graf.

Estratègies per maximitzar l'eficiència:

Una primera metodologia amb què es millora l'eficiència es realitza abans de fer servir el mètode de ramificació i poda pròpiament, i consisteix en aparellar els nodes que només tenen una parella dins del graf, i eliminar-los del mateix: això es fa així perquè aquests nodes només tenen una opció possible d'aparellament viable, si la comptabilitzem ja d'entrada i evitem considerar-la repetidament a l'arbre de solucions aconseguim reduir l'espai de cerca.

Una altra estratègia aplicada pràcticament a l'inici és la de dividir el graf en subgrafs de components connexos, i realitzar l'emparellament de cada component connex per separat, i simplement acumular les parelles i pes total en una llista general. Aquesta metodologia de dividir i vèncer parteix del fet que entre sectors del graf sense connexió per arestes no té sentit la idea de trobar una solució òptima barrejant nodes dels diferents sectors, ja que són recíprocament inaccessibles. Fent aquesta separació assolim un guany d'eficiència extraordinari en grafs grans amb varis components connexos: d'aquesta manera amb grafs més de 50 nodes que sense aplicar aquesta metodologia es podria trigar setmanes en aparellar, simplement dividint en components connexos podem resoldre el problema en menys d'1 segon en un ordinador de prestacions mitjanes; això està íntimament relacionat amb la complexitat del problema, que no és d'ordre polinòmic, sinó major. Imaginem un graf G1 amb certa quantitat de nodes i arestes i un graf G2 que consisteix en ser dues vegades el graf G1 (amb identificadors diferents pels nodes duplicats): G2 té dos components connexos anàlegs a G1: dividint el seu aparellament en dos grafs com G1, la complexitat total per G2 serà aproximadament 2*x, sent x la complexitat de G1; si per contra es resol tot G2 sense dividir en components connexos, la complexitat d'aparellar és de y, amb y >> x si G1 té una quantitat elevada de nodes i arestes, de manera que y >> 2*x: es guanya molt temps amb grafs grans que tenen més d'un component connex.

Per dividir el graf en subgrafs de components connexos, cal primer de tot saber quins nodes formen part de cada component connex, i això ho descobrim de forma senzilla i gens costosa computacionalment mitjançant la cerca per profunditat (DFS): explorem cada node del graf no visitat encara fent que cada crida a explorar porti a la seva vegada a visitar els nodes accessibles des dels de les crides anteriors (encara no visitats); cada cop que acaba la crida inicial des del mètode de la cerca en profunditat, obtenim una llista de nodes que són accessibles entre ells perquè formen un component connex. Quan tenim els nodes de tots els components connexos, anem extraiem aquests nodes i les seves arestes del graf inicial i formem subgrafs que representen els components connexos, sobre els quals posteriorment trobarem el matching òptim amb ramificació i poda, i formarem una solució final combinant les solucions dels diferents components connexos.

Un recurs que s'ha ideat per podar amb més freqüència solucions candidates que no poden portar a una solució òptima és el fet d'afegir una comprovació extra (a la comprovació de si la cota inferior és inferior a la superior) per poder afirmar que una solució candidata és viable (i per tant es pot afegir a la cua): comprovar que, amb les parelles ja comptabilitzades a la solució candidata, no hagi quedat aïllat o abandonat cap node encara no aparellat, és a dir, que no hi hagi cap node no emparellat tots els veïns del qual ja ho hagin estat: si és dóna aquest escenari, amb les parelles acumulades no es podrà arribar a una solució completa vàlida, ja que és impossible incloure el node aïllat: cal descartar la solució per inviabilitat.

Pel que fa a la cota inferior, moltes han estat les opcions amb què s'ha experimentat per calcular-la per a cada solució que es construeix per a l'arbre de solucions. Per a trobar una cota eficient i correcta s'ha seguit el principi del concepte de cota inferior: un valor de pes o penalització que, sigui com sigui que es combinin les variables restants amb les ja avaluades a una solució parcial, no podrà ser rebaixat: pel camí que sorgeix a partir dels fills d'una solució parcial, no es podrà trobar una solució d'un valor millor que la cota inferior d'aquesta solució parcial.

Inicialment es va provar a fer que el pes de les parelles acumulades fos la cota inferior d'una solució, però ràpidament es va veure que es podia millorar aquesta opció, que, tot i ser correcta, distava de ser la més eficient. Seguidament es va provar el mètode suggerit a classe de teoria, consistent en fer que la cota superior fos el rebuig acumulat per les parelles explorades més la meitat de valors de rebuig més petits, sorgits de seleccionar un valor per node d'entre els nodes que encara no s'han aparellat, sent el valor escollit per node el de la seva aresta de menor pes. Fàcilment es pot observar que es pot fer una mica més realista aquesta metodologia si s'agafa per a cada node el menor pes de les arestes que no involucrin nodes ja aparellats a la solució candidata per a la qual es calcula la cota inferior.

Finalment, però, es va optar per una opció una mica diferent, que consisteix en fer que la cota inferior d'una solució candidata sigui el rebuig acumulat per les parelles explorades més el rebuig de les parelles amb menor rebuig formades exclusivament per nodes encara no aparellats, agafant tants valors de rebuig com quedin perquè la solució sigui completa (tingui tantes parelles com la meitat de nodes del graf). Aquesta és una heurística que ha demostrat, tot i la

seva aparent simplicitat, ser molt eficient: als grafs provats el descens per l'arbre de solucions és molt precís i no es troben gaires solucions completes en qualitat de provisionalment òptimes abans que es trobi la definitiva i es pugui podar el que queda de l'arbre de solucions sencer. Aquesta cota es calcula en temps superlineal, ja que cal iterar per totes les parelles i comprovar per a cada una que els seus nodes no formin part dels ja aparellats (un conjunt inferior en nombre al de tots els nodes, durant la major part de l'algorisme). En l'intent de millorar aquesta cota es van haver de cancel·lar diverses versions alternatives, ja que s'incorria en pràctiques pròpies de metodologia voraç, com no repetir nodes a les parelles que analitzem pel rebuig extra (i fer-ho en ordre ascendent de pes, començant per la parella de nodes no explorats amb menor pes), de manera que es va mantenir el model prèviament explicat. Una addició interessant a la cota inferior actual és que, mentre és calculada, pot calcular en certs casos si la solució que tindria aquesta cota és inviable (per raons més desenvolupades que no ser inferior a la cota superior), de manera que estalvia crear una solució candidata no desitjada (per més detalls al respecte es recomana anar a la funció "obtainInferiorHeight").