Laskuharjoitus 5

14.2.2002

1. Jatkoa edellisen laskuharjoituksen 1(b) tehtävään: Etsi keinoja tehostaa peräytyvää hakua miinaharavapelin konsistenttien joukkojen määrittämiseen.

2. Kirjan tehtävä 4.13: *Latinalainen neliö* on $n \times n$-taulukko, jonka jokaisessa rivissä ja sarakkeessa kukin luvuista $1, \ldots, n$ esiintyy tasan kerran. Latinalaista neliötä kutsutaan redusoiduksi latinalaiseksi neliöksi, jos sen ensimmäisen rivin ja ensimmäisen sarakkeen alkiot esiintyvät järjestyksessä $1, 2, \ldots, n$. Kirjoita ohjelma, joka laskee peräytymishaulla redusoitujen $n \times n$ latinalaisten neliöiden lukumäärän.

3. Esitä jokin mielekäs kustannusfunktio ja naapuristo seuraaville vaikeille kombinatorisille optimointiongelmille.

   (a) Symmetrinen kauppamatkustajan ongelma.

   (b) Graafinväritys n värillä siten, että naapurisolmuilla on eri väri.

   (c) Graafin ositus: Jaa graafin solmut, joita on parillinen määrä, kahteen yhtäsuureen osaan siten, että osien välisten kaarien määrä on mahdollisimman pieni.

   (d) Lukujen ositus: jaa joukko $A = \{a_1, a_2, \ldots, a_n\}$ kahteen osaan A_1 ja A_2 siten, että $A_1 \cup A_2 = A$, $A_1 \cap A_2 = \emptyset$ ja

$$\sum_{a \in A_1} a = \sum_{a \in A_2} a.$$

Laskuharjoituksen 5 vastaukset

Tehtävä 1

Jatketaan edellisen viikon (7.2.2002) tehtävän 1 (b) yhteydessä määriteltyjen käsitteiden käyttöä.

Tehtävänä on siis löytää annetusta miinaharavan peliasemasta joukon $\mathsf{boundary}$ konsistentit osajoukot, ts. kaikki mahdolliset tavat asetella miinat joukon $\mathsf{boundary}$ ruutuihin siten, että miina-asetelma vastaa miinaharavan antamia lukemia.

Naiivi ratkaisu toimii "tuota ja testaa"-periaatteella ("generate and test"), ts. tuottaa kaikki joukon $\mathsf{boundary}$ osajoukot, ja testaa jälkikäteen onko tuotettu joukko konsistentti. Tämä ei luonnollisesti ole kovin tehokasta, koska miinaharavan antamat lukemat joukon $\mathsf{boundary}$ ympäristössä rajoittavat konsistenttien joukkojen lukumäärän tyypillisesti vain murto-osaan kaikista joukon $\mathsf{boundary}$ osajoukoista. Erityisesti, miinaharavan lukemat $d(x, y)$ joukon

$$\mathsf{constraints} = \{(x, y) \in \mathsf{known} \ : \ N(x, y) \cap \mathsf{boundary} \neq \emptyset\}$$

ruuduissa asettavat (kaikki) rajoitteet konsistenttien joukkojen rakenteelle.

Muotoillaan nyt perätyvä hakualgoritmi siten, että joukon $\mathsf{constraints}$ määräämien ehtojen toteutumista (toteutumattomuutta) tarkkaillaan jo osittaisratkaisuille, jolloin läpikäytävästä hakuavaruudesta tulee huomattavasti pienempi kuin naiivin ratkaisun tapauksessa. Algoritmin pseudokoodi on alla:

```
CONSISTENT_SETS(E, R)
    if R = ∅
        check consistency of E, output solution if consistent.
        return
    for (x, y) ∈ constraints
        if d(x, y) = |E ∩ N(x, y)| and R ∩ N(x, y) ≠ ∅                        (i)
            CONSISTENT_SETS(E, R \ N(x, y))
            return
        if d(x, y) = |E ∩ N(x, y)| + |R ∩ N(x, y)| and R ∩ N(x, y) ≠ ∅        (ii)
            CONSISTENT_SETS(E ∪ (R ∩ N(x, y))), R \ N(x, y))
            return
        if d(x, y) > |E ∩ N(x, y)| and R ∩ N(x, y) = ∅                        (iii)
            return
        if d(x, y) < |E ∩ N(x, y)|                                            (iv)
            return
    P ← any point from R
    CONSISTENT_SETS(E, R \ {P})
    CONSISTENT_SETS(E ∪ {P}, R \ {P})
    return
```

Algoritmi CONSISTENT_SETS ottaa kaksi argumenttia E, R, missä

- $E \subseteq \mathsf{boundary}$ on joukko, joka sisältää ruudut (x, y), joihin algoritmi on päättänyt asettaa miinan.
- $R \subseteq \mathsf{boundary}$ on joukko, joka sisältää ruudut (x, y), joihin algoritmi ei ole vielä ottanut kantaa (asetetaanko miina vaiko ei).

Kaikkien konsistenttien joukkojen määrittämiseksi algoritmia kutsutaan argumenteilla

$$E = \emptyset \quad \text{ja} \quad R = \mathsf{boundary}.$$

Algoritmin jokaisella rekursioaskeleella joukko R pienenee, kunnes lopulta $R = \emptyset$, jolloin tarkistetaan joukon E konsistenssi ja tämän jälkeen palataan takaisin edelliselle tasolle hakupuussa. "for"-silmukan tehtävänä on toisaalta estää haun haarautumista (ehdot (i) ja (ii)) ja toisaalta katkaista haku, jos osittaisratkaisu E on mahdoton täydentää konsistentiksi joukoksi (ehdot (iii) ja (iv)). Huomaa, että algoritmi toimii (kuten em. naiivi algoritmi) vaikka koko "for"-silmukka jätettäisiin pois.

Tarkastellaan ehtoja (i)–(iv) yksi kerrallaan. Ehdot (i) ja (ii) havaitsevat nk. "pakkotilanteet", joissa osittaisratkaisua E on jatkettava täsmälleen tietyllä tavalla, jotta lopputuloksena syntyvästä joukosta tulisi konsistentti. Erityisesti,

- ehto (i) havaitsee tilanteen, jossa miinaharavan lukema $d(x, y)$ ruudussa $(x, y) \in \mathsf{constraints}$ vastaa täsmälleen ruudun ympäristöön $N(x, y)$ jo asetettujen miinojen lukumäärää $|E \cap N(x, y)|$. Nyt jos johonkin joukon $N(x, y)$ ruutuun ei vielä ole otettu kantaa, on selvää, että tällaisiin ruutuihin *ei* voida asettaa miinaa, koska muutoin tunnettu miinaharavan lukema $d(x, y)$ ylittyisi; toisaalta
- ehto (ii) havaitsee tilanteen, jossa miinaharavan lukema $d(x, y)$ ruudussa $(x, y) \in \mathsf{constraints}$ vastaa täsmälleen ruudun ympäristöön $N(x, y)$ jo asetettujen miinojen lukumäärää $|E \cap N(x, y)|$ lisättynä niiden ympäristön $N(x, y)$ ruutujen lukumäärällä, joihin ei ole vielä otettu kantaa. On selvää, että tällöin kaikkiin ruutuihin joukossa $R \cap N(x, y)$ on asettava miina, jotta lopullisessa ratkaisussa olisi riittävästi miinoja toteuttamaan lukema $d(x, y)$.

Ehdot (iii) ja (iv) karsivat osittaisratkaisuja, joita ei voida mitenkään täydentää konsistentiksi joukoksi.

- Ehto (iii) havaitsee tilanteen, jossa ruudun (x, y) ympäristöön on asetettu liian vähän miinoja, eikä tilannetta voida enää korjata, koska kaikkiin ruutuihin ympäristössä $N(x, y)$ on jo otettu kantaa.

- Ehto (iv) havaitsee tilanteen, jossa ruudun (x, y) ympäristössä on liikaa miinoja.

Tehtävä 2

Oheinen ohjelma toteuttaa tehtävän yksinkertaisella tavalla. Koska redusoidussa latinalaisessa neliössä ensimmäinen rivi ja sarake ovat määrätyt, $n \times n$-kokoisessa latinalaisessa neliössä riittää sijoittaa $(n-1)^2$ lukua, mikä siis on haun maksimisyvyys. Ohjelma ei pidä yllä konstruoitavaa latinalaista neliötä, vaan ainoastaan boolean-muuttujia `n_in_row_i` ja `n_in_col_j`, joista nähdään, joko luku n on laitettu riville i tai sarakkeeseen j.

```
/* Etsi kertalukua N olevat redusoidut latinalaiset neliöt */
#define N 4

int n_in_row_i[N+1][N+1]; /* onko numero n jo rivissä i     */
int n_in_col_j[N+1][N+1]; /* onko numero n jo sarakkeessa j */

int place(int depth) {
  int i,j,n,sum;

  if (depth==(N-1)*(N-1)) /* yksi neliö löytyi, koska on */
    return 1;             /* sijoitettu (N-1)**2 numeroa */

  i=2+depth%(N-1); /* lasketaan depthistä, mihin kohtaan */
  j=2+depth/(N-1); /* valitaan numero seuraavaksi       */

  sum=0;
  for (n=1; n<=N; n++) {
    if(!n_in_row_i[n][i] && !n_in_col_j[n][j]) {
      n_in_row_i[n][i]=1;  /* kullekin n:lle vuorollaan:         */
      n_in_col_j[n][j]=1;  /* jos n:ä ei ole laitettu riviin i   */
      sum+=place(depth+1); /* eikä sarakkeeseen j, laitetaan se  */
      n_in_row_i[n][i]=0;  /* näiden risteyskohtaan ja jatketaan */
      n_in_col_j[n][j]=0;  /* hakua; lopuksi otetaan luku pois   */
    }
  }
  return sum;
}

int main(int argc, char **argv) {
  int i,j;
```

```
  for(i=1;i<=N;i++) {      /* nollataan taulukot */
    for(j=1;j<=N;j++) {
      n_in_row_i[i][j]=n_in_col_j[i][j]=0;
    }
  }

  for(i=1;i<=N;i++) {   /* asetetaan luvut 1...n     */
    n_in_row_i[i][i]=1; /* ensimmäiseen sarakkeeseen */
    n_in_col_j[i][i]=1; /* ja riviin                 */
  }
  printf("%d\n", place(0));
  return 0;
}
```

Tehtävä 3 (a)

Kustannusfunktiona luonnollisin valinta on reitin pituus. Eräs mahdollisuus naapuristoksi on ns. 2-OPT -siirrot: muodostetaan nykyistä reittiä kuvaavasta graafista uusi reitti poistamalla kaksi kaarta ja lisäämällä kaksi kaarta.

Tehtävä 3 (b)

Tässä kuvataan ns. fixed-k lähestymistapa. Ratkaisu on mikä tahansa solmujen ositus k osaan (tyhjät osat sallitaan). Kaksi väritystä ovat toistensa naapurit, jos toinen saadaan toisesta siirtämällä jokin solmu osasta toiseen. Kustannusfunktiona esim. niiden kaarien lukumäärä, joiden molemmat solmut ovat samassa osituksen osassa. Naapuristoa voidaan rajoittaa seuraavasti: nykyistä ratkaisua voidaan muuttaa vain niiden solmujen osalta, joista lähtee ainakin yksi väritysehtoa rikkova kaari; tällöin tosin naapuristo ei enää ole symmetrinen.

Tehtävä 3 (c)

Yksinkertainen lähestymistapa: ratkaisujoukkona on kaikki solmujen ositukset kahteen osaan. Siirto on kahden solmun vaihto osien välillä. Kustannusfunktiona kaarien määrä osien välillä.

Parempi suorituskyky on saatu simuloidulla jäähdytyksellä, kun on sallittu mielivaltaiset ositukset kahteen osaan V_1 ja V_2. Siirtoina on nyt yhden solmun siirto osasta toiseen, ja kustannusfunktiossa on lisäksi sakkona summatermi

$$\alpha(|V_1| - |V_2|)^2,$$

joka lisää kustannusta aina kun osat ovat erikokoiset. Optimointialgoritmi saattaa sakkotermistä huolimatta päätyä epätasapainoiseen ratkaisuun, jolloin voidaan joko kasvattaa α:n arvoa tai lisätä jokin yksinkertainen heuristiikka, joka muutamaa solmua siirtämällä pääsee tasapainotettuun ratkaisuun.

Tehtävä 3 (d)

Otetaan ratkaisujoukoksi kaikki ositukset kahteen osaan. Määritellään k-naapuristo sellaiseksi, jossa yhdellä siirrolla saadaan siirtää enintään k alkiota osituksesta toiseen. Kustannusfunktioksi voidaan ottaa

$$\left| \sum_{a \in A_1} a - \sum_{a \in A_2} a \right| .$$

Tämä ongelma on hiukan hankala sikäli, että siinä on pahimmillaan hyvin paljon lokaaleja optimeja. Simuloitu jäähdytys esitetyllä naapuristolla ei pärjää erityisesti tähän ongelmaan kehitetylle Karmarkar-Karp -heuristiikalle, mutta parempaa on kehitetty.