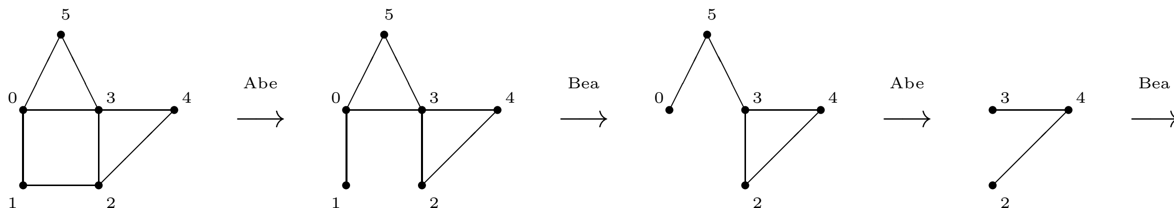
Eerste programmeeropgave — Brute force
Algoritmiek voorjaar 2013, Universiteit Leiden

In deze eerste opdracht bekijken we een spel waarbij twee personen om de beurt takken weghalen uit een ongerichte graaf en daarbij punten kunnen winnen. Degene die aan het eind de meeste punten heeft, heeft gewonnen. De bedoeling is om een programma te schrijven dat, via het doorrekenen van alle mogelijkheden (*brute force*), berekent welk van de twee spelers wint en met hoeveel verschil. Het spel wordt als volgt gespeeld.

Gegeven een ongerichte graaf zonder losse knopen. De graaf hoeft niet samenhangend te zijn. Er zijn twee spelers, Bea en Abe, die in principe om de beurt een tak weghalen. Als daarbij een knoop vrij komt, krijgt de betreffende speler een punt (als er twee knopen losraken twee punten) en moet hij/zij nog een tak weghalen. De losgeraakte knoop wordt verder buiten beschouwing gelaten. Als er geen takken meer zijn om weg te halen is het spel afgelopen. De speler die dan de meeste punten heeft, heeft gewonnen. Bij aanvang hebben de spelers uiteraard allebei nul punten. We nemen aan dat Abe begint.

Een **voorbeeld** van een reeks achtereenvolgende *beurten* wordt hieronder weergegeven.



Toelichting. In zijn eerste beurt haalt Abe de tak $(1, 2)$ weg. Dat levert hem geen punten op. Nu is Bea aan de beurt, en zij verwijdert $(0, 1)$. Dat levert haar 1 punt op. Daarna moet zij nog een tak verwijderen; ze kiest voor $(0, 3)$. Nu is de beurt weer aan Abe. Hij haalt eerst $(0, 5)$ en dan $(3, 5)$ weg, hetgeen hem 2 punten oplevert. Dan moet hij nog een tak wegnemen; hij kiest voor $(2, 3)$. Nu is de beurt weer aan Bea. Zij kan de twee resterende takken wegnemen, en behaalt daarmee 3 punten. Alle takken zijn verwijderd, dus het spel is afgelopen. Abe heeft in totaal 2 punten en Bea 4 punten, dus Bea heeft gewonnen met een verschil van 2 punten.

De opdracht bestaat uit twee delen, een C++-programma en een verslag.

1. Het **C++-programma** moet het volgende doen.
   - De gebruiker leest een ongerichte graaf in uit een invoerfile. In de invoerfile wordt de graaf als volgt weergegeven. Op de eerste regel staat het aantal knopen n. Vervolgens komen er n regels, met op elke regel n nullen en enen, gescheiden door spaties, die aangeven of de overeenkomstige tak aanwezig is in de graaf. De n rijen na de eerste regel vormen samen in feite de adjacency matrix van de graaf. De graaf wordt dus als adjacency matrix ingelezen (zie ook verderop).
   - Gebruik een constante `Nmax` (bijvoorbeeld `Nmax` $= 50$) om de maximale grootte van n aan te geven. Bij het inlezen moet gecontroleerd worden of n niet te groot is.

- Bij het inlezen van getallen uit een file mag je gebruik maken van `invoer>>getal`; er hoeft dus niet karakter voor karakter uit de file te worden gelezen. Je mag verder aannemen dat de graaf goed in de invoerfile staat.
- Het programma moet voor een ingelezen graaf bepalen of bovenbeschreven spel winnend of verliezend is voor de speler die begint (Abe dus), of remise zal worden. Tevens moet worden aangegeven met welk verschil gewonnen wordt door Abe danwel Bea (0 bij remise). We veronderstellen hierbij zoals gebruikelijk dat beide spelers optimaal spelen, dat wil zeggen altijd de voor hen beste zet doen. Bepaal daartoe voor elke stand een waarde `uitslag`, die aangeeft wat het puntenverschil zal zijn voor die betreffende stand bij optimaal spel. Bijvoorbeeld `uitslag` = aantal punten voor de speler die aan de beurt is - aantal punten voor de tegenstander, of `uitslag` = aantal punten voor Abe - aantal punten voor Bea. De waarde 0 betekent dan dat de stand tot remise leidt. De uitslag van het spel moet naar het scherm geschreven worden. Geef een duidelijke mededeling, bijvoorbeeld: "Het spel is winnend voor Bea (de tegenstander). Zij wint met een verschil van 5 punten."
- Het programma moet gebruik maken van BRUTE FORCE. Dat betekent dat het programma vanuit elke stand steeds alle mogelijke zetten een voor een doet en de vervolgstand bekijkt. (Een zet is hier het weghalen van een enkele tak door degene die aan de beurt is. Een beurt kan dus bestaan uit meerdere zetten na elkaar.) Van elk van deze vervolgstanden ga je dan RECURSIEF na wat daarvan de uitslag is. Het is nadrukkelijk niet de bedoeling om via optimalisaties het zoeken naar de beste zetten te versnellen.
- Behalve het bepalen van de uitslag van het spel voor verschillende grafen, moet ook het verschil onderzocht worden tussen het gebruik van de adjacency matrix representatie en de adjacency list representatie. Er moet dus onder andere een functie geschreven worden die een graaf in de adjacency matrix representatie omzet in dezelfde graaf in adjacency list representatie. Vervolgens moeten enige experimenten worden uitgevoerd om te bekijken wanneer welke representatie efficiënter is. Dit kan afhankelijk zijn van het aantal knopen en het aantal takken van de graaf. Gebruik als voorbeelden bij de experimenten vooral grafen waarvoor je programma nog net binnen redelijke tijd de uitslag kan bepalen. Voor heel kleine grafen zal het verschil in representatie niet of nauwelijks meetbaar zijn.
- Opmerkingen:
  - Kies bij elk van beide graafrepresentaties een verstandige klasse-structuur, met als memberfuncties in elk geval de recursieve functie die de uitslag bepaalt, en daarbij eventueel gebruikte hulpfuncties. De inleesfunctie kan als memberfunctie in de ene klasse, de functie die de adjacency matrix omzet in de adjacency list in de andere. Uiteraard moet de recursieve functie in de ene respectievelijk andere klasse *alleen* verschillen in de graafrepresentatie; het achterliggende algoritme moet hetzelfde zijn! Denk verder aan de constructor en de destructor (indien nodig).
  - Ten behoeve van de experimenten over de invloed van de graafrepresentatie op de snelheid waarmee de uitslag van het spel op een gegeven graaf wordt bepaald, zal op de website van Algoritmiek (`http://www.liacs.nl/~graaf/ALGO/`) een programmaatje komen te staan dat voor een gegeven aantal takken en knopen een random graaf genereert.

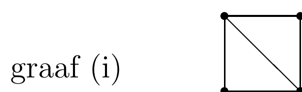
- Boven elke functie moet een commmentaarblokje komen met daarin een (zeer) korte beschrijving van wat de functie doet. Noem daarin ook de gebruikte parameters: geef hun betekenis en geef aan hoe ze eventueel veranderd worden door de functie. Geef bij memberfuncties ook aan wat deze met de membervariabelen van het object doen. Let verder op de layout (consequent inspringen) en op het overige commentaar bij de programmacode (alleen zinvol en kort commentaar).
- Het programma moet (ook) onder Linux werken.

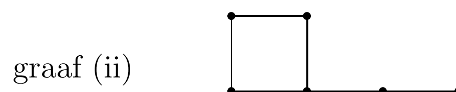
• Er is een **bonus** van 0,5 punt te verdienen als je, behalve de uitslag van het spel, tevens een serie optimale zetten genereert die tot die uitslag leidt. Deze zettenserie wordt naar het scherm geschreven op de manier zoals hieronder (het begin van) de zettenserie uit het voorbeeld op pagina 1 is afgedrukt:

```
(1,2) wordt weggenomen door Abe
(0,1) wordt weggenomen door Bea
(0,3) wordt weggenomen door Bea
...
```

2. Het **verslag** moet getypt zijn in LaTeX, en moet een introductie bevatten, de probleemstelling, een duidelijk antwoord op onderstaande vragen (a) t/m (c) en een hoofdstukje waarin je uitlegt hoe de recursieve functie voor de bepaling van de uitslag van het spel werkt. Ten slotte bevat het verslag nog een hoofdstukje over de experimenten, waarin een korte uitleg over de opzet, de resultaten en de conclusies daarvan. De resultaten bestaan uit tabellen met gegevens over de tijd die de berekening kost voor beide representaties. Neem als voorbeelden grafen waarvoor je programma nog net binnen redelijke tijd de uitslag kan bepalen.
De toestand-actie-ruimtes mogen met de hand getekend zijn (mits leesbaar).
Zie `http://www.liacs.nl/~rvvliet/algoritmiek/programmeeropdrachten` voor enkele richtlijnen bij het maken van het verslag.

   (a) Teken de toestand-actie-ruimtes voor het spel zoals dat gespeeld wordt op de volgende twee grafen. Geef bij elke stand aan wat de uitslag ervan is. Merk op dat sommige standen in feite hetzelfde zijn, ook al worden ze anders getekend. In de voorbeeldgraaf op pagina 1 levert bijvoorbeeld het wegnemen van tak $(0,1)$ dezelfde stand (graaf) op als het weghalen van $(1,2)$. In zo'n geval hoef je maar een van deze vervolgstanden (te tekenen en) uit te werken. Ook als je een stand tegenkomt die je al gehad hebt, hoef je deze niet nogmaals helemaal uit te werken.

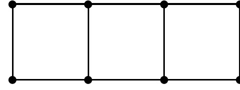
graaf (i) 

graaf (ii) 

Van graaf (i) moet de volledige toestand-actie-ruimte worden gegeven; er zijn in essentie twee verschillende directe vervolgstanden, die volledig moeten worden uitgewerkt (alle mogelijke zetten). Graaf (ii) heeft vier verschillende vervolgstanden. De vervolgstanden waarvoor direct duidelijk is wat de uitslag is hoef je niet uit te werken (maar wel uitleggen hoe de uitslag bereikt wordt), de andere moet je wel uitwerken.

(b) Leg uit wat de uitslag van het spel zal zijn als je begint met een graaf die een boom is. Idem voor een bos.

(c) Beredeneer, door de verschillende directe vervolgstanden te bekijken, wat de uitslag is als we beginnen met onderstaande graaf. Er moet dus geen volledige toestand-actie-ruimte getekend worden.



Voor eventuele aanvullingen of tips bij de programmeeropdracht of andere informatie, zie: `http://www.liacs.nl/~graaf/ALGO/`. Hier komen t.z.t. ook de behaalde cijfers te staan.

Uiterste inleverdatum: maandag 18 maart 2013. Voor elke week te laat inleveren gaat er een punt van het cijfer af. Het programma per e-mail sturen naar: graaf@liacs.nl. Listing en verslag moeten op papier worden ingeleverd en in de daartoe bestemde doos met opschrift Algoritmiek in de postkamer van Informatica (kamer 156) worden gedeponeerd. Vermeld overal duidelijk de namen van de makers.

Normering:
verslag 3 (waarvan 1 punt voor de experimenten)
commentaar en layout 1
modulaire opbouw en OOP 1
werking 5 (o.a. bepaling van de uitslag, voor beide graafrepresentaties)