ALGORITMIEK: opgaven werkcollege 5

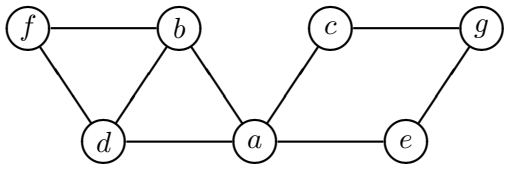
Brute force, exhaustive search en backtracking

Opgave 1. (Levitin: opgave 3.2.8)
Formuleer bij **a.** en **b.** je algoritme in woorden of in C++.
Beschouw het probleem om in een gegeven tekst het aantal substrings te tellen dat begint met een `A` en eindigt met een `B`. (Bijvoorbeeld: er zijn vier van zulke substrings in `CABAAXBYA`.)
a. Ontwerp een brute-force algoritme voor dit probleem en bepaal zijn complexiteit.
Merk op dat het aantal gevraagde substrings beginnend met een `A` die op plek i ($0 \leq i < n-1$) in de tekst staat, precies gelijk is aan het aantal `B`'s in de tekst rechts van positie i. Baseer je brute-force algoritme op deze observatie.
b. Ontwerp een efficiënter algoritme voor dit probleem.

Opgave 2. (Levitin: opgave 3.4.6) Beschouw het **partition problem** (opsplitsingsprobleem): gegeven n positieve integers, splits ze op in twee disjuncte deelverzamelingen met elk dezelfde som van hun elementen. (Natuurlijk kent dit probleem niet altijd een oplossing.) Ontwerp een exhaustive-search algoritme voor dit probleem. Probeer het aantal deelverzamelingen dat het algoritme moet genereren te minimaliseren.
Er hoeft geen C++-code geschreven te worden. Formuleer je algoritme gewoon in woorden.

Opgave 3. (Levitin: opgave 3.4.10, a, b)
Magische vierkanten. Een magisch vierkant van orde n is een plaatsing van de getallen 1 tot en met n^2 in een $n \times n$ matrix, zodat elk getal precies één keer voorkomt, en elke rij, elke kolom en elke hoofddiagonaal (linksboven-rechtsonder en rechtsboven-linksonder) dezelfde som heeft.
a. Toon aan dat als er een magisch vierkant van orde n bestaat, de genoemde som gelijk moet zijn aan $n(n^2+1)/2$.
b. Ontwerp een exhaustive-search algoritme om alle magische vierkanten van orde n te genereren.
Er hoeft geen C++-code geschreven te worden: formuleer je algoritme in woorden.

Opgave 4. Beschouw de volgende ongerichte graaf G:



a. (Levitin, opgave 3.5.1.b)
Maak een DFS wandeling door G, startend in knoop a. Wanneer een knoop meerdere buren heeft, handel die dan in alfabetische volgorde af.
Construeer ook de bijbehorende DFS boom. Geef daarin aan in welke volgorde de knopen voor de eerste keer worden bereikt (en op de stapel gezet), en in welke volgorde ze helemaal zijn afgehandeld (van de stapel worden gehaald).

b. (Levitin, opgave 3.5.4)
Maak een BFS wandeling door G, startend in knoop a. Wanneer een knoop meerdere buren heeft, handel die dan in alfabetische volgorde af.
Construeer ook de bijbehorende BFS boom. Geef daarin aan in welke volgorde de knopen worden bezocht.

Opgave 5.
a. Pas de pseudo-code voor een DFS wandeling door een ongerichte graaf (zie slides van het hoorcollege, of Levitin paragraaf 3.5) zo aan, dat het algoritme controleert of de graaf acyclisch is. Als dat het geval is, moet het algoritme **true** teruggeven. Als dat niet het geval is (de graaf bevat dus minstens één kring), moet het algoritme **false** teruggeven en tevens de knopen van de gevonden kring afdrukken, in hun volgorde in de kring.
b. (naar Levitin, opgave 3.5.6.a)
Pas de pseudo-code voor een BFS wandeling door een ongerichte graaf (zie slides van het hoorcollege, of Levitin paragraaf 3.5) zo aan, dat het algoritme controleert of de graaf acyclisch is. Als dat het geval is, moet het algoritme **true** teruggeven. Als dat niet het geval is (de graaf bevat dus minstens één kring), moet het algoritme **false** teruggeven.
c. (Levitin, opgave 3.5.6.b)
We kunnen dus zowel DFS als BFS gebruiken om te ontdekken dat een ongerichte graaf kringen bevat. Is het zo dat een van de twee algoritmes hiermee altijd minstens zo snel klaar is als de ander?
Zo ja, welk van de twee is minstens zo snel als de andere, en waarom is dat zo? Zo nee, geef een voorbeeld van een graaf waarbij DFS sneller ontdekt dat er een kring is, en een voorbeeld van een graaf waarbij BFS sneller ontdekt dat er een kring is.

Opgave 6. Pas de pseudo-code voor een DFS wandeling door een ongerichte graaf (zie slides van het hoorcollege, of Levitin paragraaf 3.5) zo aan, dat het algoritme controleert of de graaf samenhangend is. Als dat zo is, moet het algoritme **true** teruggeven, en anders **false**. Tevens moet(en) de samenhangende component(en) van de graaf genummerd worden, te beginnen bij nummer 1, en elke knoop moet het nummer van zijn component krijgen.

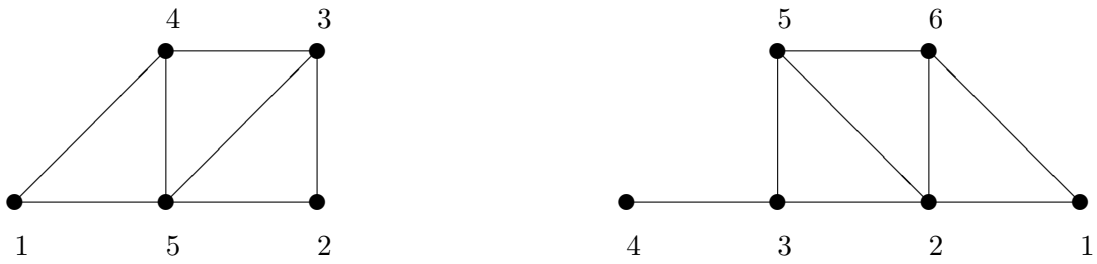
Opgave 7. Geef (in woorden) een backtracking algoritme voor het genereren van magische vierkanten. Geef duidelijk aan hoe je magische vierkanten stap voor stap opbouwt, en welke restrictie je in elke stap controleert.

Opgave 8. Graafkleuringsprobleem: gegeven een ongerichte graaf met n knopen en een positief geheel getal m. Kunnen de knopen van de graaf zodanig gekleurd worden met hooguit m kleuren, dat geen tweetal aangrenzende knopen dezelfde kleur heeft? Zo ja, geef zo'n kleuring.
a. Hoeveel verschillende kleuringen zijn er maximaal mogelijk met m kleuren? Zijn er grafen die op zoveel manieren gekleurd kunnen worden?
b. Wat is het minimale aantal kleuren dat nodig is om een complete graaf met n knopen te kleuren?
c. Geef (in woorden) een exhaustive search algoritme dat bepaalt of de knopen van een gegeven graaf met hooguit m kleuren kunnen worden gekleurd, en dat zo ja, ook zo'n kleuring oplevert.

Opgave 9.
a. Geef (in woorden) een backtracking algoritme voor het graafkleuringsprobleem.
b. Veronderstel, dat we bij het genereren van een graafkleuring elke knoop steeds eerst met kleur 1, dan met kleur 2, etc. proberen te kleuren. Er kan dan nog gekozen worden in welke volgorde de knopen worden doorlopen (en gekleurd). Deze volgorde heeft invloed op het aantal stappen dat het kost om een oplossing te vinden. We bekijken twee gevallen:

1. de knopen worden in de volgorde 1 t/m n doorlopen
2. vanuit de knoop die we nu kleuren kiezen we als volgende knoop de eerste buur (d.w.z. die met het laagste knoopnummer) die we nog niet gehad hebben. Van daaruit op dezelfde manier verder naar de volgende knoop. Begin vanuit knoop 1.

Pas beide methoden toe op onderstaande grafen met $m = 3$ en bekijk het verschil.



Opgave 10. Een *Latijns vierkant* van orde n is een n bij n vierkant (matrix) dat in elke rij en elke kolom de getallen 1 t/m n bevat. Er geldt dus dat in elke rij en in elke kolom elk van die getallen slechts één keer voorkomt. De bedoeling is nu om alle Latijnse vierkanten te genereren: zonder beperking der algemeenheid mogen we aannemen dat in de eerste rij en de eerste kolom de getallen 1 t/m n in die volgorde reeds vaststaan.

a. Een voor de hand liggende manier om een Latijns vierkant stap voor stap op te bouwen is om het vierkant rij voor rij, en per rij van links naar rechts te vullen. In elke stap wordt dan gecontroleerd of aan de restricties is voldaan. Gebruik dit backtracking algoritme om (met de hand) alle Latijnse vierkanten van orde 3 (dit is er maar 1) en van orde 4 (dit zijn er 4) te genereren (waarbij de eerste rij en kolom reeds als hierboven vastliggen).
b. Schrijf nu een recursieve C++-functie `void latijnsvierkant(n, A, i, j)` die alle Latijnse vierkanten van orde n m.b.v. backtracking genereert en afdrukt. We nemen weer aan dat de eerste rij en de eerste kolom vastliggen. De i en j geven aan dat je nu het vakje met coördinaten (i, j) probeert te vullen, waarbij alle eerdere vakjes reeds goed gevuld zijn. Neem aan dat een functie `void drukaf(n, A)`, die een n bij n matrix afdrukt, reeds bestaat.