Algoritmen 2 – Test 2

Hoofdstuk 9 – Koppelen

Koppeling in een ongerichte graaf = deelverzameling van verbindingen waarin elke knoop hoogstens 1 keer voorkomt. De eindknopen zijn dan gekoppeld.

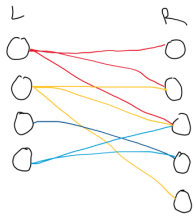
Een maximale koppeling is de koppeling met het hoogste aantal verbindingen (en dus gekoppelde knopen).

De verbindingen kunnen ook een gewicht hebben. Dan moet de koppeling met het grootste totale gewicht gezocht worden.

Tweeledige grafen

De knopen kunnen hier in twee deelverzamelingen verdeeld worden (L en R)
Alle verbindingen verbinden steeds een knoop uit L met eentje uit R.

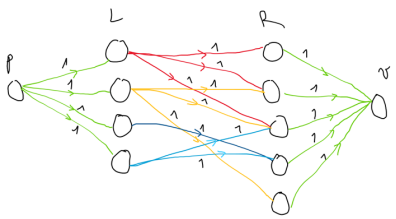
Deze grafen kunnen gebruikt worden om taken (R) aan een uitvoerder te koppelen (L).



Ongewogen

Elke uitvoerder kan slechts 1 taak aan. Maximale ongewogen koppeling = max aantal taken uitgevoerd.

Er is een nauw verband met maximale stroom. Er moet dan een producent en een verbruiker ingevoerd worden. Alle verbindingen krijgen capaciteit 1.



Het maximale koppelingsprobleem komt nu overeen met het zoeken van de maximale gehele stroomverdeling => Ford-Fulkerson.

Gewogen

Stabiele koppeling

Er zijn een of twee verzamelingen elementen gegeven. Elk element heeft een gerangschikte voorkeurslijst voor elk van de andere elementen. De elementen moeten gekoppeld worden op een wo goed mogelijke manier, de koppeling moet stabiel zijn.

Onstabiele koppeling = de koppeling bevat twee niet met elkaar gekoppelde elementen, die liever met elkaar zouden gekoppeld zijn.

3 problemen:

1) Stable marriage. Twee verzamelingen met dezelfde grootte (mannen en vrouwen). Elke man heeft (obviously) een voorkeurslijst van vrouwen.
2) Hospitals/residents. Twee verzamelingen. Elk hospitaal biedt een aantal stageplaatsen aan die moeten ingevuld worden door stagiaires. Elk hospitaal en elke stagiaire heeft een voorkeurslijst.
3) Stable roommates. Slechts één verzameling. Elk element heeft een voorkeurslijst van alle andere elementen.

Stable Marriage

Er is steeds tenminste één stabiele koppeling. Algoritme van Gale-Shapley vindt dit gegarandeerd.

Het algoritme

De mannen doen een reeks aanzoeken aan de vrouwen. Iedereen is op elk ogenblik ofwel verloofd, ofwel vrij. Eens een vrouw verloofd is blijft ze dit voor altijd. Ze kan enkel van verloofde veranderen indien ze een aanzoek krijgt van een man die hoger op haar voorkeuslijst staat. Een man kan afwisselend verloofd of vrij zijn. Hij wordt terug vrijgezel als zijn huidige vrouw verloofd wordt met een andere man. Een man die opnieuw vrijgezel wordt, mag als eerste een nieuwe vrouw kiezen (de volgende op zijn lijstje).

Elke mogelijke aanzoekvolgorde levert dezelfde oplossing op. De mannen krijgen steeds de beste mogelijke vrouw in de stabiele koppeling (de vrouwen niet, zij krijgt de slechtst mogelijke partner).

Implementatie

- Er zijn **voorkeurslijsten voor alle deelnemers**.
- Om snel te weten te komen welk van de 2 mannen een vrouw de voorkeur geeft wordt er een **ranglijst voor elke vrouw** opgesteld. Deze geeft de volgorde van elke man aan in haar voorkeurslijst.
- Vinden van een vrije man om het volgende aanzoek te doen: **lijst van vrije mannen**. Er worden daar enkel mannen uit verwijderd omdat een afgewezen man direct opnieuw een aanzoek mag doen.
- Algoritme stopt als de laatst overgebleven vrouw haar eerste aanzoek krijgt.

Uitbreidingen

- Verzamelingen van ongelijke grootte. De overschot van de grootste groep blijft ongekoppeld. Gale-Shapley toegepast op de kleinste groep vindt een stabiele koppeling.
- Onaanvaardbare partners. De voorkeurslijsten bevatten niet langer de volledige andere groep. Niet noodzakelijk iedereen krijg nu een partner. (Gale-Shapley aanpassen)
- Gelijke voorkeuren. Voorkeurslijsten bevatten ex aeqo's. Er zijn nu 3 soorten stabiliteit. Superstabiliteit, sterke stabiliteit, zwakke stabiliteit.

Hospitals/Residents

Merk de 2 uitbreidingen op: het totaal aantal plaatsen moet niet gelijk zijn aan aantal stagiaires en de voorkeurslijsten mogen onvolledig zijn. Gelijke voorkeuren zijn niet toegelaten.

Oplossing

Probleem transformeren: elk hospitaal met p plaatsen wordt vervangen door p dezelfde hospitalen: h_1, h_2, …, h_p. Al deze hospitalen hebben dezelfde voorkeurslijst. Nu gewoon Gale-Shapley toepassen.

Stable roommates

Veralgemening van stable marriage, maar met slechts 1 groep.

Hoofdstuk 11 – Zoeken in strings

T = tekst (hooiberg)
t = lengte T
P = patroon (naald)
p = lengte patroon

Formele talen

= verzameling eindige strings over een alfabet. Men bestudeert hoe men een taal kan beschrijven.

Generatieve grammatica's

$$\begin{array}{lcl} \langle \mathbf{S} \rangle & ::= & \langle \mathbf{AB} \rangle | \langle \mathbf{CD} \rangle \\ \langle \mathbf{AB} \rangle & ::= & a \langle \mathbf{AB} \rangle b | \epsilon \\ \langle \mathbf{CD} \rangle & ::= & c \langle \mathbf{CD} \rangle d | \epsilon \end{array}$$

$$\langle \mathbf{S} \rangle \Rightarrow \langle \mathbf{CD} \rangle \Rightarrow c\langle \mathbf{CD} \rangle d \Rightarrow cc\langle \mathbf{CD} \rangle dd \Rightarrow cc\langle \mathbf{CD} \rangle dd \Rightarrow ccc\langle \mathbf{CD} \rangle ddd \Rightarrow cccddd.$$

Reguliere uitdrukkingen

Elke regexp definieert een formele taal => Taal(regexp)

Ø = regexp met taal de lege verzameling.
ε = lege string, Taal(ε) = {ε}
Elke letter van het alfabet is ook een regexp met als taal: Taal('a') = {'a'}

Combineren:

- Concatenatie
- OF '|'
- Cleenesluiting '*'

Regexp	Grammatica
$\emptyset$	$\langle \mathbf{S} \rangle ::= \langle \mathbf{S} \rangle$
ϵ	$\langle \mathbf{S} \rangle ::= \epsilon$
a	$\langle \mathbf{S} \rangle ::= a$

Variabele tekst

We zoeken een naald in een hooiberg.

Eenvoudige methode

We leggen het patroon boven de tekst, links te beginnen. We vergelijken de overeenkomstige karakters, we stoppen met vergelijken als we een verschil gevonden hebben. Als er een verschil is gevonden schuiven we het patroon één plaats naar rechts op. We vergelijken opnieuw alle karakters. Als alle karakters uit het patroon gematcht zijn hebben we de naald in de hooiberg gevonden.

Knuth-Morris-Pratt

Prefixfunctie

Dit is nodig om efficient KMP toe te passen. Met de prefixfunctie q stellen we een tabel op die het resultaat van de prefixfunctie q bevat voor elk karakter uit het patroon.

Om de prefixtabel op te stellen heb je enkel het patroon P nodig. 'i' wordt gebruikt als index.

We kunnen string Q vóór i op P leggen ALS Q overeenkomt met een even lange deelstring van P eindigend vóór i.
(Eenvoudiger: we zoeken in de deelstring die voor i staat **een suffix, die ook een prefix is** van diezelfde deelstring)
De prefixfunctie op plaats i geeft dan de lengte van de langst mogelijke prefix die ook een suffix is van de deelstring vóór i.

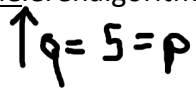
i	0	1	2	3	4	5	6	7	8	9
P	A	N	O	A	N	A	A	N	O	A
q	-1	0	0	0	1	2	1	1	2	3

q(0) is niet gedefinieerd
q(1) is altijd 0
q(i+1) is nooit groter dan q(i) + 1
q(i+1) = q(i) ALS karakters P[q(i)] = P[i]
Als P[q(i)] != P[i] dan kunnen we eventueel nog een korter prefix verlengen: q(i+1) = q(q(i)) +1 ALS karakters P[q(q(i))] = P[i]
Als P[q(q(i))] != P[i] kunnen we eventueel nóg een korter prefix verlengen: q(i+1) = q(q(q(i))) + 1 ALS P[q(q(q(i)))] = P[i]
Enz…

Een eenvoudige lineaire methode

Men stelt een string samen bestaande uit P gevolgd door T, gescheiden door een karakter dat in geen van beide voorkomt. Daarna de prefixtabel van deze string bepalen. Als q(i) gelijk is aan p (de lengte van P) dan hebben we op die plaats het patroon gevonden.

arnie#tarskeenarnielerenalgoritmenensnappenernietsvan



Knut-Morris-Pratt

Maakt ook gebruik van de prefixfunctie, maar beperkt het aantal binnenste operaties.
i = index in het patroon
j = index in de tekst

Bij de eenvoudige methode schoven we telkens 1 plaatsje naar rechts. KMP kan die sprong groter maken. Als een mismatch gevonden wordt kunnen we nu P met i - q(i) plaatsen opschuiven en dan verdergaan met vergelijken van T[j] met P[q(i)].

De sprong i - q(i) noemen we s. Opschuiven met sprong s is alleen zinvol als er dus een prefix is die ook een suffix is EN als **P[i – s] != P[i]** (anders treedt er onmiddellijk een fout op).
Om die tweede voorwaarde niet steeds te moeten checken stelt men voor KMP een licht andere prefixtabel op: de KMPtabel. Deze wordt berekend met q' en geeft telkens de meest zinvolle verschuiving s.

P	A	N	O	A	N	A	A	N	O	A	N	O	
q'	-1	0	0	0	0	2	1	0	0	0	0	5	3
q	-1	0	0	0	1	2	1	1	2	3	4	5	3

KMP[i] = q(i) ALS q(i+1) <= q(i)

Varianten

- Boyer-Moore: patroon van achteren naar voor onderzoeken: de sprong zal groter zijn dan bij KMP.
- Men kan ook eerst zo snel mogelijk een fout proberen ontdekken door eerst de karakters van het patroon te onderzoeken die nauwelijks in de tekst voorkomen.

Boyer-Moore

Lijkt goed op de eenvoudige methode. Ze gaat T en P op verschillende beginposities karakter per karakter vergelijken. Er zijn ook een aantal verschillen.

De vergelijkingen gebeuren van achter naar voor binnen het patroon. Als we een fout vinden bepalen 2 heuristieken de waarde van de grootst mogelijke verschuiving:

- Verkeerde karakter
- Juiste suffix

Heuristiek van het verkeerde karakter

Als een mismatch optreedt gaan we vergelijkingen overslaan tot een mismatch een match wordt OF tot P volledig voorbij het mismatchte karakter kan verschoven worden.

Bij stap 1 kunnen we P drie plaatsen opschuiven, dan wordt de mismatchte 'C' wel een match.

Bij stap 2 kan er van de mismatch geen match gemaakt worden, want de letter A (mismatch) komt niet voor aan de linkerkant van G in het patroon. In dit geval mag het volledige patroon voorbij de mismatch A geschoven worden (zie stap 3).

Om die verschuiving makkelijk te bepalen stellen we een tabel op die de meest rechtse positie van elk karakter in P bijhoudt.

P: ABCDABD

A	B	C	D
4	5	2	6

De verschuiving kan hier negatief worden: het verkeerde karakter komt voor aan de rechterkant van foutpositie i. Daarom moet er gebruik gemaakt worden van een tweede heuristiek die steeds een verschuiving van minstens 1 oplevert.

Er bestaan ook varianten:

- Uitgebreide karakteristiek van het verkeerde karakter.
- Variant van Horspool.
- Variant van Sunday

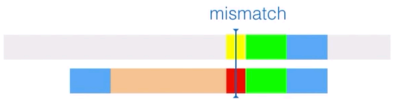
Heuristiek van het juiste suffix

Beide heuristieken geven een verschuiving. Boyer-Moore neemt de grootste verschuiving van de beide heuristieken.

Na een mismatch hebben we eigenlijk een suffix van P gevonden in T. Als we verschuiven moet het gedeelte dat tegenover die suffix in T komt, opnieuw overeenkomen. Anders heeft de verschuiving geen zin.
We moeten dus weten of die suffix nog eens in P voorkomt, en waar.
Daarvoor doen we weer wat voorbereidend werk.
We stellen een tabel op die voor elke index j in P de lengte s(j) van het grootste suffix van P bijhoudt, dat daar begint. s(j) noemen we de suffixfunctie. Alle indices j waar s(j) = p-i-1 zijn kandidaten voor de verschuiving. We hebben het meest rechtse voorkomen nodig, dus nemen we de grootste van deze waarden en noemen ze k. De verschuiving wordt dan gegeven door i+1+k.

Er zijn enkele speciale gevallen.

- Patroon P werd gevonden. Het juiste suffix in nu P zelf. Er is ook geen verkeerd karakter dus de eerste heuristiek kan geen verschuiving opleveren. We mogen P toch geen p posities opschuiven, want een nieuw voorkomen van P in T kan de vorige overlappen. De verschuiving is p - s(0).
- Er is geen juist suffix (dus direct een fout bij het laatste karakter uit P). De eerste heuristiek levert in dit geval zeker een positieve verschuiving op.
- Het juiste suffix komt niet meer in P voor. Het is nog altijd mogelijk dat er een suffix van het juiste suffix overeenkomt met een prefix van P. De verschuiving is in dit geval opnieuw gelijk aan p - s(0). Zie foto.



Can't find the matched good suffix in the pattern somewhere else

However the "suffix of the suffix" (the blue part) can be found as a prefix of the pattern

Het opzoeken van de verschuiving lijkt traag want telkens moet de grootste verschuiving gezocht worden in de tabel. We kunnen dit vermijden door voor elke index j de foutpositie i = p-s(j)-1 en verschuiving v[i] = i+1-j bepalen, en de kleinste verschuiving voor elke i bij te houden.

Onzekere algoritmen

Deze algo's leveren met een zekere waarschijnlijkheid een geheel foutief resultaat op. Dit is een Monte-Carlo algoritme.
Toch kunnen deze algo's nuttig zijn.

Bvb. bij de dolle koeienziekte. Er zijn 2 testen om te bepalen of een koe besmet is. Een van 200 euro, die altijd juist is, en een van 10 euro die in 1% van de gevallen fout is. De test van 10 euro geeft nooit een vals negatief antwoord (de test zegt dat het rund besmet is, terwijl dat wel zo is), maar alleen vals positieve antwoorden.

Men kan dus eerst opteren om de goedkope test te doen, en als deze positief is kan men nog een dure test erbij doen om het zeker te zijn. Als de goedkope test negatief is, moet je de dure test niet meer uitvoeren.

Karp-Rabin

Deze methode herleidt het vergelijken van strings tot het vergelijken van getallen. Enkel nuttig als getallen sneller kunnen vergeleken worden dan strings.

Aan elke string even lang als P => uniek geheel getal toekennen. Men gaat nu de overeenkomstige getallen vergelijken i.p.v. de tekst. Gelijke getallen = gelijke strings.

Er zijn d^p verschillende strings en dus worden getallen groot. Men beperkt zich tot getallen die in 1 processorwoord passen (w bits). Dit beperkt het aantal mogelijke getallen.
=> meerdere strings hebben hetzelfde getal. Cfr. hashing.

Gelijke strings = gelijke getallen MAAR gelijke getallen != gelijke strings
Het algoritme zal zich dus af en toe vergissen.

De hashwaarde (op positie j+1) moet in O(1) uit de deelstring kunnen afgeleid worden. Dit kan enkel als we kunnen gebruik maken van de vorige berekende hashwaarde (deelstring op positie j).

$$H(P) = \sum_{i=0}^{p-1} P[i]d^{p-i-1} \quad \Rightarrow \quad H_r(P) = H(P) \bmod r$$

$$H_r(T_{j+1}) = ((H(T_j) - T[j]d^{p-1})d + T[j+p]) \bmod r$$

$$H_r(T_0) = \left(\sum_{i=0}^{p-1} T[i]d^{p-i-1}\right) \bmod r$$

Hoe kiezen we r?

- VAST. Een zo groot mogelijk priemgetal kiezen zodat rd <= 2^w.
- RANDOM. Is veiliger! r kiezen uit een vooraf opgesteld bereik.

Zoeken met automaten

Bij deterministische automaten is een staat = een toestand. Een staat beschrijft de gehele toestand van een eenheid. Er zijn evenveel staten als mogelijkheden.
Bij Niet deterministische automaten is een staat iets anders.
Een automaat heeft een invoerkanaal. De verzameling van invoerwaarden = alfabet.
Een nieuwe staat hangt af van een oude staat en van de invoer.

Zoeken naar strings: hooiberg is een reeks karakters die de automaat als invoer krijgt. Als de automaat een het einde van een gezochte string P ontmoet geeft hij een uitvoersignaal.

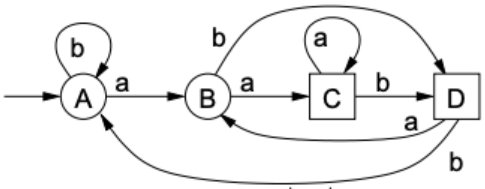
Deterministische automaten (DA)

Heeft een beperkt aantal toestanden.

- Eindige verzameling invoersymbolen
- Eindige verzameling toestanden S
- Begintoestand s_0 (s_0 element van S)
- Verzameling eindtoestanden F (F deelverzameling van S)
- Overgangsfunctie p(t,a), t = toestand, a = ontvangen symbool.

DA = gerichte geëtiketteerde multigraaf G. Knopen zijn de toestanden, verbindingen zijn overgangen, etiket is invoersymbool. Als de DA na het invoeren van een symbool zich in een eindtoestand bevindt is het patroon herkend.

Een snelle implementatie van de overgangsfunctie van een DA is een 2D tabel met als rijen de toestanden en als kolommen de symbolen. Met vernuftige technieken kan de tabel gecomprimeerd worden.



		a	b
→	$A = \{1\}$	B	A
	$B = \{1,2\}$	C	D
←	$C = \{1,2,3\}$	C	D
←	$D = \{1,3\}$	B	A

(a|b)*a(a|b)

Niet deterministische automaten (NA)

NA = een deterministisch mechanisme waarin toeval geen rol speelt.

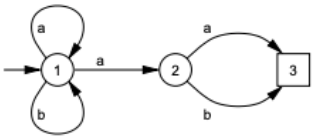
Een NA heeft een aantal statenbits die ‘aan’ of ‘uit’ kunnen aannemen. De staat wordt dan aangeduid door de verzameling statenbits die aan staan. De begintoestand wordt aangeduid door een speciale statenbit, de beginbit. Deze staat aan in het begin terwijl alle andere bits uit staan. De eindstaten worden aangeduid door eindbits. Een NA is in een eindstaat als 1 of meer eindbits aanstaan.

Overgang werkt bit per bit. Een statenbit die aan staat reageert op een invoersymbool stuurt een signaal naar nul of meer statenbits. Een statenbit die één of meer signalen binnenkrijgt zet zichzelf aan. Een statenbit die geen signaal krijgt gaat uit. s(i,a) is de verzameling statenbits die een signaal krijgen (en dus aangaan) van statenbit i als die een 'a' uit het invoeralfabet binnenkrijgt.

Een ε-overgang tussen i en j wil zeggen dat als i een signaal krijgt, j ook onmiddellijk aangaat.

De overgangstabel bevat dus verzamelingen van statenbits i.p.v. eenvoudige staten.

Een string is herkend als op het einde van de string, een of meerdere statenbits aanstaan.



		a	b
→	1	{1, 2}	{1}
	2	{3}	{3}
←	3		

Figuur 11.1. Niet-deterministische automaat voor $(a|b)*a(a|b)$.

Deelverzamelingenconstructie

⇨ Omzetten NA naar DA

Een NA is een alternatieve voorstelling van een DA. NA is soms makkelijker (bvb bij regexpen).

Bij een DA moeten we bij elke binnenkomende letter een nieuwe staat opzoeken in een tabel. Als de tabel niet onhandelbaar groot is, is dit redelijk efficiënt.
Bij een NA moeten we bij elke binnenkomende letter alle statenbits die aanstaan afgaan, en andere bits die een signaal krijgen aanzetten.

Een NA gaan we dus meestal omzetten naar een DA (=deelverzamelingenconstructie). Daarvoor doorlopen we de multigraaf met diepte- of breedte eerst zoeken, vertrekkend van de beginstaat.

Buren kunnen we niet opzoeken in een burenlijst. Daarom 2 hulpoperaties:

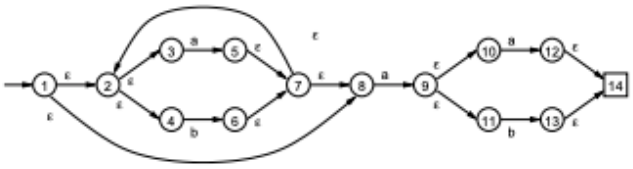
- Deelverzameling statenbits bereikbaar via ε-overgangen vanuit een verzameling statenbits T noemt men een ε-sluiting.
- Overgangsfunctie p(t,a) uitbreiden tot een verzameling statenbits T tot p(T,a). Resultaat is verz. statenbits rechtstreeks bereikbaar vanuit een toestand t uit T voor invoersymbool a.

Het bepalen van een ε-sluiting = zoeken van alle statenbits bereikbaar via een ε-overgang vanuit T. Code in de cursus.

Automaten voor regexps

Constructie van Thompson => NA opbouwen uit gegeven regexp, bottom-up. Eerst de primitieve NA's definiëren die overeenstemmen met de basiselementen van een regexp.

Primitieve elementen zijn symbolen uit het alfabet en de lege string ε.



Figuur 11.6. Constructie van Thompson: NA voor $(a|b)*a(a|b)$.

Minimalisatie van een automaat

Met de constructie van Thompson worden er onnodig veel statenbits gebruikt. Als we deze NA omzetten naar een DA zal deze nog meer toestanden bevatten. Elke DA kan men minimaliseren naar een equivalente DA met een minimaal aantal toestanden.

Equivalente toestanden tracht men te groeperen: elke groep bevat dan alle toestanden die hetzelfde gedrag vertonen en die niet van elkaar te onderscheiden zijn.

Twee toestanden zijn equivalent wanneer men vanuit beide toestanden, na invoer van om het even welke string, ofwel in twee gewone toestanden terechtkomt, ofwel in twee eindtoestanden (niet noodzakelijk dezelfde).

Toestanden equivalent? => alle paren toestanden opsporen die niet equivalent zijn. Om te beginnen weet je zeker dat elke eindtoestand zich onderscheid van elke gewone toestand. Wanneer men geen niet-equivalente toestanden meer kan vinden, zijn de overblijvende paren equivalent.

In elke groep kiest men een toestand als vertegenwoordiger. De vertegenwoordigers worden toestanden van de gereduceerde DA. Overgangen tussen toestanden worden dan overgangen tussen groepen. Tenslotte verwijdert men nog alle toestanden uit de nieuwe DA die onbereikbaar zijn vanuit de begintoestand.

De Shift-AND-methode

Bitgeoriënteerd. Zeer efficiënt voor kleine patronen. Eenvoudig aan te passen om patronen te vinden waarin fouten zijn geslopen.

We gaan het aantal vergelijkingen op zich niet verminderen, maar elke vergelijking maken we hier supersnel door gewoon bits te vergelijken i.p.v. karakters.

Voor elke positie j in de tekst T bijhouden welke prefixen van P overeenkomen met de tekst, eindigend op die positie j. => tabel R.

Rj = waarde tabel die hoort bij tekstpositie j.

Het i-de element Rj[i-1] is waar als de eerste i karakters van P overeenkomen met de i tekstkarakters eindigend in j.
Het patroon is gevonden als Rj[p-1] waar is. Het komt dan voor op positie j-p+1 in de tekst.

De nieuwe tabel Rj+1 kan telkens uit Rj afgeleid worden.

$$R_{j+1}[0] = \begin{cases} \text{waar als } P[0] = T[j+1] \\ \text{niet waar als } P[0] \neq T[j+1] \end{cases}$$

$$R_{j+1}[i] = \begin{cases} \text{waar als } R_j[i-1] \text{ waar is en } P[i] = T[j+1] \\ \text{niet waar in de andere gevallen} \end{cases} \qquad \text{voor } 1 \leq i < p$$

Veel werk, tabel op elke positie j bepalen? NEE -> De elementen zijn gewoon logische waarden, en er bestaan instructies die op alle bits van processorwoorden tegelijk inwerken. Het patroon mag dus niet langer zijn dan de lengte w van een processorwoord.

Rj+1 wordt:

Rj+1 = Schuif(Rj) EN S[T[j+1]]

S is een tabel die een entry heeft voor elke letter van het alfabet, en per entry wordt daar bijgehouden op welke plek in de naald die letter voorkomt.

Shift-AND Benaderende overeenkomst

Laat fouten in het patroon toe.

Hoofdstuk 14 – Metaheuristieken

Vuistregels bij het zoeken naar een oplossing van een probleem.

In principe kan men wel alle mogelijkheden afgaan en de beste oplossing daaruit kiezen, maar dit is soms niet praktisch haalbaar. Daarom stelt men benaderende algoritmes op die in een redelijke tijd **een** oplossing geven, maar niet noodzakelijk de beste.

Combinatorische optimisatie

Optimisatie = uit een gegeven verzameling S halen we het beste individu. S wordt niet gedefinieerd door een opsomming, maar door een aantal voorwaarden waaraan de elementen uit S moeten voldoen. Het beste individu wordt bepaald door een evaluatiefunctie f (beste individu = kleinste waarde f).

Bij TSP: S = verzameling verbindingen die allemaal aan elkaar hangen en alle steden worden bezocht. Het beste individu is die met de kortste lengte.

Het probleem is te groot om op te lossen met backtracking. Is dit niet het geval gebruik je beter backtracking.

Een metaheuristiek is een manier waarop individuen worden uitgekozen. Bij alle metaheuristieken wordt er daarvoor een steekproef genomen uit de totale verzameling S. Voor elk individu wordt de f-waarde berekend, waarbij het beste individu dat we ooit ontmoet hebben bewaard wordt.

Vaak wordt gestopt met zoeken als de tijd op is, of als er een oplossing gevonden is die niet meer te verbeteren valt.

Vooronderstellingen

De methodieken werken niet voor alle problemen. Er zijn een aantal voorwaarden.

Het moet zinvol zijn om op zoek te gaan naar betere individuen in de buurt van een gegeven individu.
Individuen opstellen maakt bijna steeds gebruik van een mate van randomisatie.
Je kan het individu opbouwen vanaf de grond, of aan de hand van vorige individuen door die licht aan te passen.
Soms is het niet zinvol om uit te gaan van vorige individuen. Dan worden individuen volledig random samengesteld.

```
S RandomZoekBeste(){
    S beste=maakRandom();
    while (nog tijd){
        S nieuw=maakRandom();
        if (f(nieuw) < f(beste))
            beste=nieuw;
    };
    return beste;
};
```

Een nieuw individu vanaf de grond opbouwen:
- Constructie: Opbouwen uit componenten (bv steeds verbindingen toevoegen aan een graaf)
- Rechtstreeks aanmaken

Soms beide: bvb in TSP:
- Verbindingen toevoegen individu = pad in graaf
- Direct een permutatie nemen van het aantal te bezoeken winkels

Er zijn meerdere goeie kandidaten als beste individu -> kiezen:
- Shortlisting: men neemt de beste kandidaten en kiest daaruit met gelijke waarschijnlijkheid.
- Gewogen keuze: Elke mogelijke keuze krijgt een gewicht.

Keuze van een nieuw individu:
- Kleine wijzigingen aanbrengen in een reeds bekeken individu
- Kruisen van 2 al bekeken individuen
- Vermengen van grote hoeveelheden individuen

Lokaal vs globaal zoeken

Heeft het zin om een individu van S te verbeteren door het aanbrengen van kleine wijzigingen? Wat is een kleine wijziging?

Begrip omgeving invoeren: een omgeving N(s) is de verzameling individuen bekomen door een kleine wijziging aan s aan te brengen. s is een element uit S.

In de omgeving van een individu zitten meestal meerdere andere goeie individuen, maar ook enkele veel slechtere individuen. De methode wordt zinvol als het vaak voorkomt dat er veel goeie (betere) individuen in de omgeving van het vorige individu zitten, na een kleine wijziging. Dit noemt men lokale optimisatie.

Als dit niet voorkomt -> randomisatie toepassen. Men neemt een random verandering en kijkt of dit een beter resultaat oplevert. Dit doet men tot een beter individu is gevonden. Indien geen beter individu gevonden wordt zit men in een lokaal minimum.

Als er maar 1 lokaal minimum is, is dit het beste individu. Er kunnen echter meer lokale minima zijn. Er moet dus ook kunnen ontsnapt worden uit een lokaal minimum. -> exploratie (globaal).
- Helemaal opnieuw beginnen, random opbouw van een individu.
- Grote wijzigingen aanbrengen in reeds bekeken individuen.

Methodes zonder recombinatie

Gewone random zoeken 2 elementen toevoegen:
- Lokaal zoeken
- Heuristieken die constructie van individuen kunnen sturen

VB: ILS en GRASP

Simulated Annealing

Lokale optimisatie -> we zoeken een individu in de buurt van het vorige en bewaren het als het beter is dan het vorige.

Bij simmulated annealing is er een kans dat we het nieuwe individu bewaren, ookal is het slechter dan het vorige. De kans is afhankelijk van de evaluatiefunctie, en van een temperatuur T.

De kans = $\rho(T, f(s'), f(s)) = \exp\left(\frac{f(s') - f(s)}{T}\right).$

De begintemperatuur is vrij hoog, dit bevordert de exploratie. Dan laten we T dalen tot bijna nul. De eindfase van simmulated annealing benaderd dan lokaal zoeken.
Het algoritme vindt zeer waarschijnlijk een globaal optimum.

Tabu Search

Houdt een taboelijst bij: een lijst al gecontroleerde individuen die niet opnieuw mogen bezocht worden.

```
S TabuSearch(){
    S s=maakRandom();
    lijst<S> Taboe;
    while (!stopconditie){
        Taboe.voegtoe(s);
        s=besteNietTaboe(N(s),Taboe);
    };
    return s;
};
```

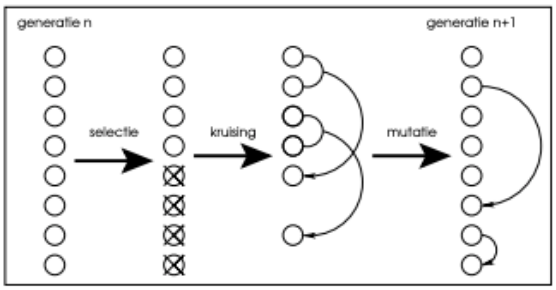
De taboelijst zorgt ervoor dat de methode voldoende exploreert. De lijst moet groot genoeg zijn zodat men ver genoeg uit de buurt van een lokaal optimum komt om het basin of attraction te verlaten. Dit is meestal niet haalbaar, daarom meestal eigenschappen bewaren i.p.v. echte individuen.

Genetische algoritmes

= metaheuristieken gebruik makend van kruising.
We gaan 2 individuen vermengen.
Enkel zinvol als dit kan leiden tot betere individuen.
2 opties:

- Gerichte kruising. Men heeft een idee van de algemene structuur van individuen en men combineert zinvolle componenten ervan.
- Blinde kruising. Random bepaalde elementen van de individuen combineren.

Zo gaat men telkens over naar de volgende generatie.



Een mutatie is een kleine wijziging.

Vermenging

Geen tijd meer.