



SAMPLING FREQUENCY TUNING TOOL

Prosjektrapport
IKT-407 Web Mining

Laget av:
Carl Thomas Vatne
Lars Rune Haugen
Per Øyvind Hodøl

Innhold

1 Innledning	3
1.1 Forfase	3
1.2 Målsetting	3
1.3 Avgrensning	4
1.4 Verktøy	4
2 Tidligere arbeid	5
2.1 Standarder.....	5
2.2 Evalueringsverktøy	5
2.3 Metoder for detektering av endring på websider	6
3 Vårt arbeid : Crawler og justering av samplingfrekvens.....	7
3.1 Beskrivelse av crawleren	7
3.2 Algoritmen for justering av samplingfrekvens.....	8
3.3 Detektering av forandring i tilgjengelighet.....	11
4 Videre arbeid	13
4.1 Algoritmen.....	13
5 Konklusjon.....	14
6 Kildehenvisning.....	15
7 Kildekoden.....	16

1. Innledning

I dag benytter millioner av brukere Internett til enhver tid. Internett er blitt en egen liten verden, hvor vi stort sett kan gjøre alt vi kan gjøre i den virkelige verden. Her finnes butikker, banker, nyheter, chatrom, mailklienter, nyheter, mediasentre og informasjon om alt mellom himmel og jord. Men mange har kun begrenset tilgang, eller ikke tilgang i det hele tatt til alt dette. Det er for eksempel folk som

- ikke kan se, høre eller bevege seg
- har vanskeligheter med å lese eller forstå tekst
- ikke har eller har mulighet til å bruke mus eller tastatur
- har en skjerm som bare viser tekst, liten skjerm eller treg Internett-tilgang
- ikke forstår språket dokumentet er skrevet på
- befinner seg i en situasjon hvor øyne, øre eller hender er opptatt eller utilgjengelig (for eksempel en bråkete arbeidsplass)
- har en tidligere versjon av en browser, en annen browser, en lydbasert browser eller et annet operativsystem.

For at alle skal få tilgang til all informasjon må all software tolke koden likt. Det er derfor viktig å definere standarder og følge dem. Standardene er allerede definert, men langt fra alle følger dem. Det er flere grunner til det. Ettersom alle står fritt til å legge ut sider på Internett vil det være mange som har relativt liten kunnskap. Disse vet kanskje ikke hvordan en side kan gjøres mer tilgjengelig, eller hvorfor vi trenger tilgjengelighet. Det er også lett å skylde på Microsoft, som er den desidert største produsenten av software, og som nesten har monopol på operativsystemer. Et selskap på den størrelsen burde vise større grad av ansvar og få folk til å tenke mer på tilgjengelighet. I bunn og grunn skyldes mangel av tilgjengelighet økonomiske interesser. Det er rett og slett ikke økonomisk å bruke ressurser på å forme websider slik at de er tilgjengelige for alle. Det finnes pålegg fra staten om at etater skal ha websider med stor grad av tilgjengelighet, problemet er at svært få følger opp disse påleggene.

1.1 Forfase

Ettersom behovet for å følge standardene har økt, ønsker vi å finne de sidene som følger standardene. Vi ønsker også å finne ut hvor ofte sider forandrer seg, og ikke minst hvor ofte tilgjengeligheten forandrer seg. Ettersom antallet Internettsider er overveldende, ønsker vi ikke å sjekke alle sider til enhver tid. For å unngå å sjekke unødvendig mange sider ønsket vi å lage en algoritme for å optimalisere søkemotoren. Dette vil føre til en effektivisering i forhold til tilgjengelige ressurser både med hensyn på båndbredde og prosessorkraft.

1.2 Målsetting

Målet er å lage en crawler som går gjennom ulike websider og justerer en individuell oppdateringsfrekvens slik at søk kan gjennomføres mest mulig optimalt. Poenget er at søkemotoren skal slippe å søke gjennom en side hvis tilgjengeligheten ikke er oppdatert.

Crawleren skal hente ned webdokumenter og hashe dem for så å sammenlike den nye hashen mot en tidligere versjon. Tanken er da at vi ut ifra de forskjellige oppdateringsdatoene skal lage en algoritme som kan finne en gjennomsnittlig oppdateringsfrekvens for hver side – dersom dette finnes. I motsatt fall vil man i alle fall få en pekepinn på hvor ofte siden oppdateres.

1.3 Avgrensning

Vi skal se bort ifra alt innhold på sidene når vi sjekker om de er oppdatert. Det vil si at vi stripper HTML-koden for alt annet enn taggene. Det er ikke så viktig for oss at crawleren fungerer optimalt, vi har heller lagt vekt på å finne en god algoritme for optimalisering av søkene. Tidsbegrensingen er på ca 6 uker. På denne tiden skal vi lære oss et nytt programmeringsspråk, sette oss inn i tilgjengelighetsproblematikken og lage crawleren.

1.4 Verktøy

For å lage crawleren bruker vi programmeringsspråket Python. Det er et ganske “rett frem” programmeringsspråk. Det har en del likhetstrekk med Java og C++, som vi har jobbet med før. Databasen vil bli laget i PostgreSQL, og for å administrere databasen vil vi bruke gratisprogrammet PGAdmin.

2. Tidligere arbeid

2.1 Standarder

Section 508

Section 508 er en tilgjengelighetsstandard utviklet av “The Architectural and Transportation Barriers Compliance Board” (Access Board). Section 508 er utviklet for den Amerikanske staten først og fremst til bruk i offentlige etater. Dette er en mye brukt standard. Vi har derimot valgt å legge vekt på W3C sin WCAG.

WCAG

World Wide Web Consortium (W3C) er en organisasjon som utvikler spesifikasjoner og retningslinjer for Internett. Man kan si at W3C er et forum for informasjon, diskusjon og felles forståelse av Internett. Målet er å lede Internett til sitt fulle potensiale gjennom disse åpne forumene.

Web Content Accessibility Guideline (WCAG) er en av mange retningslinjer for Internett. WCAG er retningslinjer for tilgjengelighet. Målet er å skape tilgjengelighet for alle. Den består av 14 retningslinjer som har mellom 1 og 10 punkter. Hvert punkt har prioritet 1, 2 eller 3.

- Prioritet 1 - Punkter med prioritet 1 er relativt enkle å tilfredsstillte, og burde tilfredsstillte. Hvis disse punktene ikke er tilfredsstillte vil informasjonen på siden ikke være tilgjengelig for en eller flere bruker grupper. Å tilfredsstillte disse punktene er grunnleggende for at noen grupper skal ha tilgang til siden.
- Prioritet 2 - Prioritet 2 er noe vanskeligere å tilfredsstillte. Hvis disse punktene ikke er tilfredsstillte vil en eller flere grupper ha vanskelighet med å få tilgang til informasjonen på siden.
- Prioritet 3 - Prioritet 3 er vanskelig å tilfredsstillte, og regnes som et profesjonelt nivå. Hvis ikke prioritet 3 punktene er tilfredsstillte kan noen grupper ha vanskelighet med å få tilgang til noe av informasjonen på siden.

Ut fra disse punktene er det laget tre karakterer for hvor god tilgjengeligheten er på siden:

- Nivå A – Alle punkter med prioritet 1 er tilfredstilt
- Nivå AA – Alle punkter med prioritet 1 og 2 er tilfredstilt
- Nivå AAA – Alle punkter med prioritet 1, 2 og 3 er tilfredstilt

2.2 Evalueringsverktøy

Det finnes mange ulike verktøy som evaluerer tilgjengeligheten på en side. Det finnes også andre verktøy som forbedrer sidens tilgjengelighet. Disse verktøyene beskriver hvor i koden en eventuell forbedring kan komme, og hva som burde gjøres. Forbedringsverktøy er laget som støtte for programmereren, det finnes også filter og transformeringsverktøy som er til støtte for brukeren. Slike verktøy sjekker hvordan brukeren kan få bedre tilgang til en side ut ifra hans ståsted. Dette kan for eksempel dreie

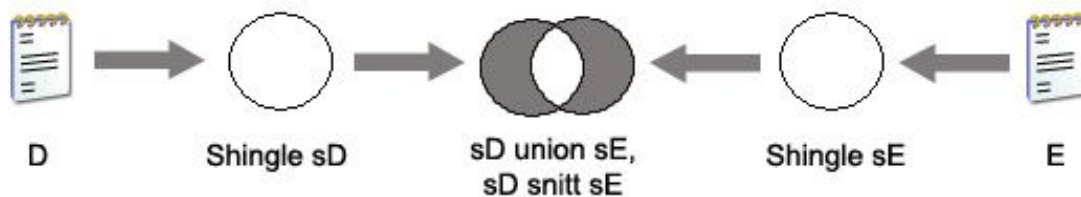
seg om å bruke en annen webbrowser eller tilleggsprogramvare. På W3C sine sider finnes en lang liste med slike verktøy [<http://www.w3.org/WAI/ER/existingtools.html>]

I prosjektet vårt var det mest aktuelt å evaluere tilgjengeligheten på en side. Vi har hovedsaklig sett på en test; Bobby. Det finnes mange verktøy som er grundigere og mer pålitelige enn denne testen, men vi brukte denne fordi den er gratis. Den er også online og enkel å bruke. Bobby sjekker tilgjengelighet på en webside i forhold til WCAG og Section 508. Testen er utviklet av Watchfire og er fritt tilgjengelig for alle [<http://bobby.watchfire.com/bobby/html/en/index.jsp>]. Bobby er ment som et redskap for webutviklere hvor tilgjengeligheten kan sjekkes, og forbedres ut ifra det. Testen går gjennom flere punkter, og hvis den oppdager et punkt som ikke er tilfredsstillt, viser den hvor i koden mangelen er og hvilken prioritet dette punktet har.

2.3 Metoder for detektering av endring på websider

Shingles

En shingle er en tekst som er bygd opp av en sekvens av tekst elementer. Shingles kan lages på bakgrunn av et dokument eller forskjellige valgte tekststrenger. Dette brukes blant annet til å finne likhet og ulikhet mellom to dokumenter. Et eksempel på dette er hvis vi har to dokumenter D og E, deretter brukes shingling og vi får to nye tekster sD og sE. Hvis vi tar union mellom sD og sE og deler dette på sD snitt sE får vi en likhetsverdi. Illustrasjonen under viser dette.



$$\frac{|S_1 \cap S_2|}{|S_1 \cup S_2|} = \text{Likhetsverdi}$$

Når det er funnet likhetsverdi må man evaluere hvor mye ulikhet man skal tolerere. Dette kan forbedres ved å “shingle” kun segmenter av en webside. Ved å bruke shingles kun på segmenter, vil man finne ut hvor stor og hvor ulikheten er.

Et annet bruksområde er å sjekke en egendefinert shingel mot en shingel fra et dokument. Den egendefinerte shingelen kan da inneholde spesielle sekvenser med tekst man ønsker å finne, eventuelt ikke finne i dokumentet.

3. Løsning

3.1 Beskrivelse av crawleren

Vi delte crawlerens arbeid inn i fire faser.

1. Innsamle HTML dokumenter og renske dokumentene for uinteressant tekst.
2. Hashe det renskede dokumentet.
3. Sjekk den nye hashverdien mot en tidligere lagret hash.
4. Implementere algoritmen.

Informasjon om URLen (frekvens, hash, med mer) lagres i en database.

Fase 1

I den første fasen samles alle data inn i HTML-format. Disse dataene er på mange måter ubrukelige for oss, siden de inneholder litt for mye informasjon. De fleste websider har for eksempel en besøksteller, og denne vil forandre seg for hver gang man besøker siden. Det samme gjelder datoer og reklamebannere. Dermed er hashing av hele innholdet en dårlig løsning. Vi vurderte flere mulige måter å omgå dette på, men omfanget av oppgaven begrenset hva vi hadde mulighet for å implementere.

Løsningen vi endte opp med, parser taggene ved å renske bort all tekst og absolutt alt utenfor BODY-området i koden. Dermed håper vi også å ha fjernet en del reklamebannere, da disse kan være plassert i skriptkode utenfor dette området. Hashingfunksjonen blir på denne måten matet med kun tagger – og kun de av dem som er innenfor BODY-området.

Fase 2

Når en side blir strippet for tekst, sitter man igjen med bare taggene. Disse legger crawleren inn etter hverandre i en string. For å sjekke om siden har endret seg må man sammenligne denne stringen mot den forrige som er lagret for den gitte URLen. Siden stringene kan bli veldig lange, egner de seg dårlig for sammenligning – og særdeles dårlig for lagring. Derfor valgte vi å hashe stringene med en md5-hash. Md5 hash-algoritmen krever lite prosessorkraft og er den korteste av de mest brukte hash-algortimene. Python har en egen md5-hash-klasse som inneholder flere metoder. Hashingen ble derfor relativt enkel å implementere.

Fase 3

Når stringen med taggene er hashet må crawleren sammenligne hashverdien med den siste verdien som er lagret for den gitte URLen. Først må det sjekkes om det i det hele tatt er lagret noen hash for denne URLen tidligere. En try-catch tar seg av dette. Har URLen en gammel hashverdi, sjekkes den nye mot den gamle i en if-setning. Er disse like, har siden ikke blitt endret. Er de ikke like, har siden blitt endret. Da må den nye hashverdien lagres i databasen. Algoritmen vil regne ut hvor mange dager crawleren skal vente før den skal sjekke siden igjen. Denne verdien vil følgelig justere seg etter oppdateringsfrekvensen på hver enkelt webside. Vi la imidlertid inn en manuell styring for å unngå store variasjoner i crawlerens besøksfrekvens. Denne styringen bestod i to

variable som definerer med hvilke hopp besøksfrekvensen skal justeres ved henholdsvis økning og reduksjon i sidens oppdateringsfrekvens. I tillegg hadde vi fått styringer fra faglærer og veileder med minimumsverdi på ett døgn og maksimumsverdi på tretti dager.

For å synliggjøre hvordan dette fungerer i praksis, kan vi ta et eksempel. La oss si at websiden www.domain.com oppdateres hvert femte døgn. I begynnelsen vil crawleren sjekke siden hver dag. Det femte døgnet oppdater crawleren en endring, og besøksfrekvensen justeres opp med +2 døgn (som er vårt valg). Den nye verdien er da tre døgn mellomrom. Etter hvert vil crawleren finne ut at den skal øke med +2 en gang til, og den vil da sjekke siden hvert femte døgn. I tillegg vil verdien for sidens oppdateringsfrekvens justeres. Sistnevnte beregnes hver gang crawleren finner en endring i hashverdien, og da beregnes gjennomsnittet av forrige gjennomsnittsverdi og antall dager mellom forrige oppdatering og denne (se under punktet om databasen).

Fase 4

Selve implementasjonen var en god del mer omfattende enn hva vi hadde trodd. Det viste seg at intelligensen i systemet måtte utvides for å ta hånd om alle mulige problemstillinger. Vi endte derfor opp med å omstrukturere systemet i forhold til hvordan vi hadde tenkt i utgangspunktet. Python viste seg å være et godt valg som programmeringsspråk, da svært mange avanserte funksjoner er implementert gjennom biblioteker. En stor del av Pythons funksjoner kan benyttes uten at man behøver å skrive mange linjer med kode.

Databasen

Databasen har naturlig nok forandret seg ettersom prosjektet har tatt form. Vi endte opp med to tabeller; henholdsvis url og urlsearch. Den første inneholder ei liste over alle websidene som vi benyttet under testingen. I tillegg inneholder den felter for oppdateringsfrekvens, søkeintervall og neste gjennomsøkingsdato. Tanken er at skriptet skal kjøres en gang i døgnet, og at det selv skal finne ut hvilke URLer som antas endret og som bør sjekkes denne aktuelle datoen. Den andre tabellen inneholder resultatet av hashtesten for hver gang siden er endret, i tillegg til en fremmednøkkel mot url-tabellen og en datostempling. Sistnevnte benyttes hver gang siden er endret til å beregne hvor lang tid det er siden sist oppdatering.

I ettertid viste det seg at det ville vært enklere og brukt MySQL, fordi PostgreSQL ikke har en innebygd autonummer-funksjon. Dette medfører ekstra databasetrafikk grunnet ekstra SQL-statements.

3.2 Algoritmen for justering av samplingfrekvens

Algoritmen skal regne ut en optimal søkefrekvens for crawleren. Frekvensen skal variere fra 1 til 30. Det vil si at crawleren skal søke gjennom en side maksimum en gang om dagen og minimum hver 30. dag. Vi baserer algoritmen på forandring av HTML-tagger. I starten av prosjektet ville vi bruke forandring i tilgjengelighet som et viktig element. Hvis tilgjengeligheten forandret seg ofte, ville frekvensen minke, og dermed ville crawleren sjekke siden oftere. Det viste seg at det å sjekke tilgjengelighet var et veldig mye mer omfattende prosjekt enn hva vi trodde. Etter samtale med veileder, bestemte vi oss for å skjære ned på denne delen og heller fokusere på om siden var oppdatert i vesentlig grad. Detektering av forandring i tilgjengelighet vil bli beskrevet senere.

Vi vurderte også å bruke to hashtester. Den første testen skulle hashe hele webdokumentet for så å sammenlikne det mot et tidligere eksemplar. Det vi skulle oppnå med denne testen var å finne ut om siden var statisk. Den andre testen skulle luke ut tekst på siden og hashe bare taggene. Hvis siden feilet første hash test (hashverdiene var like), ville siden bli kategorisert som en statisk side. Hvis siden passerte første hashtest (hashverdiene var ulike), men feilet andre, ville det bety at bare teksten på siden var forandret. Formålet med denne testen var å luke ut sider som ikke forandret tagger, kun tekst. Dette er sider som for eksempel forandrer dato automatisk. Hvis siden passerte begge testene ville sidens tagger ha vært forandret. Forandring i tagger tyder på en litt større og mer omfattende oppdatering. Hvis en side hadde forandret tagger ville frekvensen sunket. Dersom siden bare forandret tekst ville også frekvensen sunket, men i mindre grad. Men etter hvert som prosjektet formet seg droppet vi den første testen. Grunnen til dette er at algoritmen ikke ville blitt optimal, siden den andre testen også skiller ut statiske sider. Vi implementerte kun den andre testen hvor vi sjekket etter forandring i taggene.

I starten så vi også for oss en algoritme som halverte eller fordoblet frekvensen hvis tilgjengeligheten hadde forandret seg. Dette var noe vi slo fra oss ganske tidlig. Det ville ikke blitt en optimal søkefrekvens fordi hoppene ville blitt for store. Vi fant ut at vi måtte bruke en form for gjennomsnitt. Vi vurderte flere ulike typer gjennomsnitt.

1. Gjennomsnitt hvor alle oppdateringer vektet likt
2. Gjennomsnitt hvor den siste oppdateringen vektet mest
3. Gjennomsnitt hvor alle oppdateringene blir vektet etter hvor lang tid det er siden oppdateringen kom.

Alle gjennomsnittene er ment som en del av algoritmen og kan ikke stå alene.

1. Gjennomsnitt hvor alle oppdateringer vektet likt

Når crawleren oppdager forandring på siden, lagrer den datoen i databasen ved hjelp av timestamp. Etter hvert vil databasen bestå av mange slike datoer. Meningen var da ganske enkelt å regne ut antall dager mellom oppdateringene, for så å ta gjennomsnitt av disse.

Dato	Dager siden forandring
2005.01.01	
2005.01.10	10
2005.01.25	15
2005.02.06	12
2005.02.12	6
2005.02.22	12

Tabell 3.2

Snittet ville da blitt $10+15+12+6+12=55/5 = 11.1$. Den nye frekvensen ville da blitt 11. Hvis vi kaller intervallet i_n ($1 < n < \infty$) og gjennomsnittet g blir funksjonen slik:

$$g = (\sum i_n) / n$$

Denne algoritmen er ganske enkel å komme frem til og å implementere i crawleren. Men hvis vi sjekker over veldig lang tid vil antall oppdateringer bli for stort og frekvensen forandre seg lite, selv om intervallet er stort. Dette ville også ført til at utregningsprosessen ville blitt tung. I tillegg kan det være et problem å oppdage tette oppdateringer hvis intervallet mellom oppdateringene i lengre tid har vært høy. Generelt kan vi si at det blir vanskelig å forandre en trend. Problemet kunne løses ved å bruke et vindu, da slipper man også for mange poster i databasen, og det blir enklere å endre trenden. Hvis vi skulle brukt denne metoden måtte vi i tillegg hatt en metode som slettet alle lagrede postene utenfor vinduet.

2. Gjennomsnitt hvor den siste oppdateringen vektet mest

At den siste oppdateringen vektet mest vil si at intervallet mellom den siste og nest siste oppdateringen vektet mer enn intervallet mellom de tidligere oppdateringene. Denne metoden tar først gjennomsnittet av de to første verdiene. Hvis vi fortsetter med tall fra tabell 3.2 blir det $10 + 15 = 25/2 = 12.5$ som blir rundet opp til 13. Hvis det neste intervallet er 12 blir får vi $12 + 13 = 25/2 = 12.5$ som igjen blir rundet opp til 13. Deretter får vi $13 + 6 = 19/2 = 9,5 \approx 10$. Hvis vi kaller intervallet i_n ($2 < n < \infty$) og gjennomsnittet g_n blir funksjonen slik:

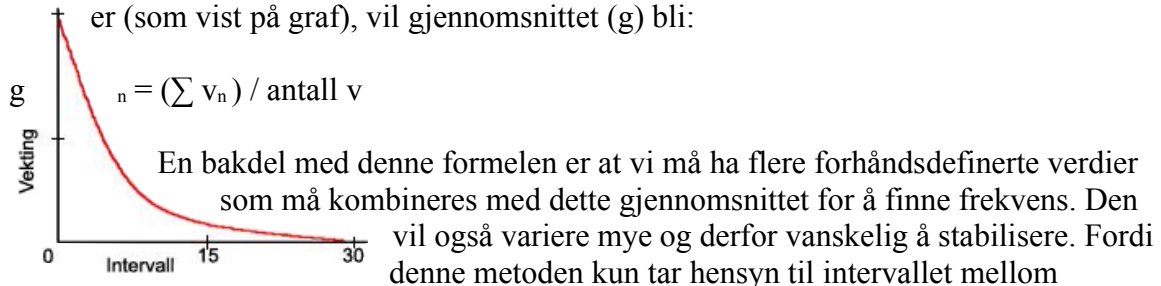
$$g_n = (i_n + g_{n-1}) / 2$$

Å legge mest vekt på den siste oppdateringen kan ha både fordeler og ulemper. En ulempe kan være hvis vi har en stabil frekvens og vi får et intervall som skiller seg veldig fra frekvensen. Da vil gjennomsnittet endres betraktelig. Det kan også være en fordel, fordi det da vil være enklere å snu en trend. Denne metoden blir også enklere å implementere i og med at vil slipper bruk av vindu og at vi kun bruker to verdier i utregningen.

3. Gjennomsnitt hvor alle oppdateringene blir vektet etter hvor lang tid det er siden oppdateringen kom

Her ville vi lage et gjennomsnitt som vektet alle oppdateringene i stigende grad. Jo mindre intervall mellom dagens dato og forrige oppdateringsdato, desto høyere vektning. Si for eksempel en side ble oppdatert for 3 og 21 dager siden. Da ville den oppdateringen som skjedde for 3 dager siden bli vektet mer enn oppdateringen som skjedde for 21 dager siden. Dette ville ført til at mange nylige oppdateringer ville gått tapt. Det vil med andre ord si at lite intervall mellom oppdateringen ville bli oppdaget raskere.

Hvis vi da sier at hver oppdatering får en vektning (v_n), som øker etter hvor lite intervallet er (som vist på graf), vil gjennomsnittet (g) bli:



Vi valgte å bruke gjennomsnittsmetode 2. Gjennom testing fant vi ut at denne metoden var den som ga best resultater. Vi har ikke fått testet gjennomsnittsmetode 1 med vindu. Gjennomsnittsmetode 3 var veldig interessant; vi kunne utviklet og finjustert den noe mer hvis vi hadde hatt tid.

Disse tre gjennomsnittene kan som sagt ikke stå alene, og må derfor kombineres med andre formler for å danne en god algoritme. Det vi først og fremst ser er at etter ett gjennomsnitt er funnet og hashen er ulik, vil frekvensen aldri minke. Derfor må vi ha en metode for å minske frekvensen. Hvis det siste oppdateringsintervallet er mer enn +2 eller -4 i avvik fra det forrige, har vi valgt henholdsvis å øke frekvensen med 2 eller å redusere den med 4. Hvis den er innenfor dette intervallet har vi valgt å beholde den forrige verdien. Tallene +2 og -4 er tall som vi har implementert i crawleren, men som lett kan forandres senere, da de initialiseres i toppen av koden. Vi valgte å redusere med et høyere tall fordi vi anser det som viktigere at side blir sjekket en gang for mye enn en gang for lite.

3.3 Detektering av forandring i tilgjengelighet

Detektering av forandring i tilgjengelighet viste seg å bli et veldig omfattende tema. Noen punkter kan relativt enkelt sjekkes ved å gå gjennom HTML-taggene til en side. Dette kan for eksempel være å sjekke om siden har en "title"-attributt i hvert ramme-element. For eksempel:

```
<FRAMESET ROWS="100,*">  
  <FRAME SRC="sidebar.html" TITLE="Sidemeny">  
  <FRAME SRC="content.html" TITLE="Hovedside">  
</FRAMESET>
```

Et annet eksempel kan være å sjekke om "img"-taggen har en "alt" attributt, slik som dette:

```
<IMG SRC="sidemeny_knapp1.gif" ALT="Gå til side x">
```

Når vi sjekker et dokument for likheter og ulikheter som dette kan shingles være et godt redskap. Shingles er beskrevet i kapittel 2.3. Vi kan også bruke shingles til å sjekke om informasjonen er relevant ved å plukke ut elementer som er irrelevante. Dette kan komme til nytte hvis vi skal sjekke om teksten til en link er relevant. Det vil ikke bli en fullgod test, men ved å plukke ut noen bestemte ord kan den bli nokså god. Vi ser da for oss at vi plukker ut ord og fraser som "link", "les mer", "les hele saken" og "her" (som i "klikk her for å gå til...").

Noen punkter er vanskeligere å sjekke og krever litt spesiell kunnskap. Dette omfatter for eksempel skript, animasjoner (flash mm), CSS og liknende. For eksempel skal animasjoner ikke flikke i for høy hastighet, ettersom dette kan føre til at personer med fotosensitiv epilepsi kan få anfall. Vi har valgt å droppe disse områdene fordi de er veldig omfattende og tidskrevende.

4. Videre arbeid

4.1 Algoritmen

Når det gjelder det videre arbeidet med algoritmen, er det mye som kan gjøres. En interessant problemstilling er å se nærmere på å kombinere denne algoritmen med en algoritme som luker ut like dokumenter.

Vi vet at ca 30 % av alle websider er en form for duplikat eller nesten duplikat. Dette skyldes i stor grad voksende bruk av forum og genererte produktsider for nettbutikker. En god algoritme for dette vil da redusere antall sider som må søkes gjennom betraktelig. Det er gjort mye forskning innen dette temaet og det burde være lett å hente frem slik informasjon. For slike metoder kan shingles bli en viktig ingrediens.

En annen interessant problemstilling kan være å kombinere denne algoritmen med verdier fra WCAG-tester. Vi ser da for oss en algoritme som gir en side "poeng" etter hvor god tilgjengeligheten er. Det vil være ønskelig at poengene er delt opp i en større skala enn WCAG sin prioritering, som består av tre nivåer. Det kan for eksempel beregnes i prosent i forhold til hvor mange av punktene som er tilfredsstilt. Dette prosenttallet kan da legges til prioriterings nivået. For eksempel A80 kan bety at du har tilfredstilt 80% av alle prioritet A punktene. Ved å bruke en slik metode vil vi også kunne måle verdier for alle nivåene og implementere disse i algoritmen. Det vil også bli mulig å oppdage forandring i tilgjengelighet.

I tillegg kunne det vært interessant å se på mønstergjenkjenning. Hvis vi sjekker en side som oppdateres hver uke i 6 måneder. Anta at når sommerferien kommer blir den ikke oppdatert på 6 uker. Da vil man få en stor økning i frekvensen; noe vi ikke ønsker. Derfor ville det kanskje vært gunstig å gjenkjenne mønstre, i dette tilfellet at det kommer en oppdatering hver uke. Vi ønsker da at frekvensen øker seint, dessuten vil vi at frekvensen skal komme ned på det gamle nivået så raskt som mulig.

Ettersom tidsrammen var relativt liten, fikk vi ikke testet algoritmen over lengre tid. Dette er noe som ville gitt en endelig vurdering og kanskje peke på hva som må justeres for at algoritmen skal bli optimal.

5. Konklusjon

Målet var å lage en crawler som gikk gjennom ulike websider og justerte en individuell oppdateringsfrekvens slik at søk kunne gjennomføres mest mulig optimalt. Ideen var da å bruke denne frekvensen i søk for å unngå å søke på sider hvor tilgjengeligheten ikke var forandret.

I begynnelsen hadde vi skissert en løsning som tok veldig detaljert hensyn til WCAG-retningslinjene til W3C. Ved nærmere studier fant vi ut at en god del av punktene var meget tidkrevende å implementere. Da vi støtte på en del andre problemstillinger i forhold til resten av skriptet, måtte vi omprioritere og skjære ned på valideringen av tilgjengelighet. I samtale med veileder fikk vi forståelsen av at denne valideringen ikke egentlig var en viktig del av vår oppgave, men mer en separat funksjon som enten allerede var laget eller som skulle lages i et fremtidig prosjekt.

For at tilgjengeligheten skal endre seg må nødvendigvis også taggene endre seg. Vår crawler går derfor gjennom sidene og sjekke om taggene har blitt endret for å bestemme om siden er blitt oppdatert. Ettersom sidene var oppdatert eller ikke, skal en algoritme bestemme hvor mange dager crawleren skal vente før den sjekker de forskjellige URLene igjen.

Vi mener at programkoden vår er oversiktlig og bra strukturert, i tillegg til at den er relativt kort og konkret. Vi har også lagt vekt på å kommentere koden godt med tanke på videreutvikling.

Mye tid har blitt brukt på å finne en algoritme som gir ønsket resultat. Etter mange forsøk og mye regning kom vi frem til at et gjennomsnitt var den beste løsningen. Målet var at antall dager mellom sjekkene skulle gå mot en jevn verdi, eller et lite intervall. Algoritmen gir nå en fornuftig verdi i forhold til hvor ofte sidene endrer tilgjengelighet. Vi har diskutert tre typer gjennomsnitt. Gjennomsnitt hvor alle oppdateringer vektet likt, gjennomsnitt hvor den siste oppdateringen vektet mest og gjennomsnitt hvor alle oppdateringene blir vektet etter hvor lang tid det er siden oppdateringen kom. Vi bestemte oss for å bruke gjennomsnitt hvor den siste oppdateringen vektet mest.

Vi har også sett på tilgjengelighetsproblematikken og hvordan den etter hvert kan kombineres med algoritmen. Det viste seg at tilgjengelighet var et omfattende område. For at vi skulle komme i mål med prosjektet unnlot vi å sjekke forandring i tilgjengelighet. Vi har derimot vist noen enkle eksempler på hvordan enkelte tilgjengelighetssjekker kan implementeres i crawleren.

Vi mener vi har nådd målet vårt, og er fornøyd med sluttproduktet. Crawleren er relativt omfattende og fungerer utmerket. Samtidig har vi kommet frem til en algoritme som gir et reelt og saklig resultat. Vi føler også at vi har lært mye om Python og tilgjengelighetsproblematikken.

6. Kildehenvisning

<http://www.w3.org/>

Mye relevant informasjon er hentet derfra. (18.11.2004)

<http://www.w3.org/TR/WAI-WEBCONTENT/>

W3C sin Web Content Accessibility Guidline (18.11.2004)

<http://eaccess.rince.ie/white-papers/2004/warp-2004-00/warp-2004-00.html>

Studie av tilgjengelighet på nett (18.11.2004)

<http://www2003.org/cdrom/papers/refereed/p097/P97%20sources/p97-fetterly.html>

Studie av nettets evolusjon (18.11.2004)

<http://www.access-board.gov/sec508/508standards.htm>

Beskrivelse av Section 508 (18.11.2004)

<http://www.python.org/>

Dokumentasjonen for programmeringsspråket Python (18.11.2004)

O'Reilly – Learning Python

O'Reilly – Programming Python

7. Kildekoden

HTMLparser.py:

```
"""
Class which parses HTML document and returns all the tags
in a string variable. This result is to be hashed and
compared to existing value in database.
Based on a pre-made class called BaseHTMLProcessor.py in the book "Dive Into Python" by Mark Pilgrim
"""

from sgmlib import SGMLParser
import htmlentitydefs
from string import upper

class BaseHTMLProcessor(SGMLParser):
    def reset(self):
        # extend (called by SGMLParser.__init__)
        self.pieces = []
        SGMLParser.reset(self)
        self.inBody = 0

    def unknown_starttag(self, tag, attrs):
        # called for each start tag
        # attrs is a list of (attr, value) tuples
        # e.g. for <pre class="screen">, tag="pre", attrs=[("class", "screen")]
        # Next line produces this (example): ' href="www.vg.no" target=_new'
        if upper(tag) == "BODY":
            self.inBody = 1

        if self.inBody == 1:
            strattrs = "".join([' %s="%s"' % (key, value) for key, value in attrs])
            # Next line takes the tag name (ie. 'a'), appends '<'
            self.pieces.append("<%s%s>" % (tag, strattrs))

    def unknown_endtag(self, tag):
        # called for each end tag, e.g. for </pre>, tag will be "pre"
        if self.inBody == 1:
            # Reconstruct the original end tag.
            self.pieces.append("</%s>" % tag)

        if upper(tag) == "BODY":
            self.inBody = 0

    def output(self):
        """Return processed HTML as a single string"""
        return "".join(self.pieces)

if __name__ == "__main__":
    for k, v in globals().items():
        print k, "=", v
```

Slutt HTMLparser.py

Algorithm.py (bygget fra bunnen av):

```
from _pg import *
import md5
import urllib
import time
from datetime import date, timedelta
from HTMLparser import BaseHTMLProcessor
from string import lower

class DBHash:

    def __init__(self):
        # If the url is new, use this frequency in robot [default = 1 day]
        self.newhashfrequency = 1
        self.increaseBy = 2 # If increase in update frequency, look again in n days
        self.decreaseBy = 4 # If decrease in update frequency, look again in n days

        # Connect to database:
        self.conn('ikt407_04', 'steinbit.agder-ikt.hia.no', 5432, "", 'postgres', 'pg123')

    def forceAll(self):
        # Checks all urls and re-hashes all of that are changed

        urls = (self.db.query("SELECT url, nextsearchdate FROM url")).getresult()
        print "Checking all urls ..."
        for var in urls:
            self.testurl(var[0])

    def doAll(self):
        # Checks all urls and treats those that are to be treated today
        # as indicated in table 'url'

        urls = (self.db.query("SELECT url, nextsearchdate FROM url")).getresult()
        print "Checking all urls ..."
        for var in urls:
            if self.DateIsToday(var[1]) == 1:
                self.testurl(var[0])
            else:
                print "Scheduled for another day: %s" % var[0]

    def conn(self, *args, **kw):
        # Database connection wrapper
        self.db = connect(*args, **kw)

    def DateIsToday(self, date):
        # Returns 1 if string date is today, else 0.

        # Parse date-string to ints:
        year = int(date[0:4])
        month = int(date[5:7])
        day = int(date[8:10])

        # Retrieve today's date:
        tm = time.localtime()

        # Compare the dates
```

```

        if ((year == tm[0]) and (month == tm[1]) and (day == tm[2])):
            return 1
        return 0

def hashit(self, tobehashed):
    # Hashes tobehashed and returns it
    m1 = md5.new()

    # Hashes input
    m1.update(tobehashed)
    return m1.hexdigest()

def gettime(self):
    # Returns local time formatted for timestamp in database
    tm = time.localtime()
    stime = "%s-%s-%s %s:%s:%s" % (tm[0],self.zeroconv(tm[1]),self.zeroconv(tm[2]),
self.zeroconv(tm[3]),self.zeroconv(tm[4]),self.zeroconv(tm[5]))
    return stime

def zeroconv(self, var):
    # Formats number in var to two-digit string and returns result
    if var < 10:
        return "0%s" % str(var)
    else:
        return str(var)

def gettimediff(self, strPrevious):
    # Calculates the difference in days between strPrevious (date-string) and today
    # Required format: "YYYY-MM-DD hh:mm:ss"

    if cmp(lower(strPrevious), None): return 1
    if cmp(lower(strPrevious), ""): return 1
    print "strPrevious: %s" % strPrevious

    pyear = int(strPrevious[0:4])
    pmonth = int(strPrevious[5:7])
    pday = int(strPrevious[8:10])

    # Retrieve today's date
    tm = time.localtime()

    if (tm[0] - pyear) > 0: return 30
    elif (tm[1] - pmonth) > 1: return 30
    else:
        d1 = date(pyear, pmonth, pday)
        d2 = date(tm[0], tm[1], tm[2])
        delta = d2 - d1
        if delta.days <= 30: return delta.days
        else: return 30

def getnextdate(self, newinterval):
    # Calculates the next search date based on what getnewinterval() calculated (!)

    # Retrieves the current time:
    tm = time.localtime()

    # Adds the interval to current date:

```

```

d = date(tm[0],tm[1],tm[2]+newinterval)

# Returns the new date as string:
return "%s-%s-%s 00:00:00" % (d.year, d.month, d.day)

def getnewinterval(self, timediff, oldinterval):
# Calculates the new interval based on the old interval, timediff, increaseBy and
decreaseBy
# (as defined in __init__)

newinterval = oldinterval

# If the time span between the current date and the previous update exceeds increaseBy,
# then increase the interval by the pre-defined varibale increaseBy
if (timediff - oldinterval) >= self.increaseBy:

    if (oldinterval + self.increaseBy) <= 30:
        newinterval += self.increaseBy
    else:
        # But if the new interval would be more than 30 days,
        # forget it and set the interval to 30 days manually
        newinterval = 30

# If the time span between the current date and the previous update is less than
decreaseBy,
# then decrease the interval by the pre-defined varibale decreaseBy
elif (oldinterval - timediff) >= self.decreaseBy:
    if (oldinterval - self.decreaseBy) > 1:
        newinterval -= self.decreaseBy
    else:
        # But if the new interval would be less than one day,
        # forget it and set the interval to one day manually
        newinterval = 1

return newinterval

def gettags(self, text):
# Cleans the HTML source from everything but the tags
# Also removes tags outside <BODY> ... </BODY> - section

parser = BaseHTMLProcessor()
parser.feed(text)
parser.close()
return parser.output()

def testurl(self, url):
# Reading URL from web:
# -----
opener = urllib.FancyURLopener({})
filehandle = opener.open(url)

# Immediately parsing html source so that only changes in
# tags in BODY-section are validated
txt = self.gettags(filehandle.read())

# Retrieving old hash from db

```

```

# First, let's find urlid for the given url:
dbuid = self.db.query("SELECT urlid, hashfrequency, searchinterval, nextsearchdate
FROM url WHERE url='%s'" % url)
uid = 1
oldhash = None
dburlid = None
dbldate = None
dblhf = None
dblsi = None
dblsd = None

try:
    # If this doesn't fail, the url already exists in db:
    uid = dbuid.getresult()[0][0]
    dblhf = int(dbuid.getresult()[0][1]) # Old hash frequency
    dblsi = int(dbuid.getresult()[0][2]) # Old search interval
    dblsd = dbuid.getresult()[0][3] # Old search date (today)

    # Second, let's find the index of the last (hash-) record for this url in urlsearch:
    dbusmax = self.db.query("SELECT MAX(urlsearchid) FROM urlsearch
WHERE urlid='%s'" % str(uid))
    usid = dbusmax.getresult()[0][0]

    # Third, let's retrieve the actual record (to get the previous hash value):
    dbq = "SELECT hashvalue, date, urlid FROM urlsearch WHERE
urlsearchid='%s'" % str(usid)
    dbresult = self.db.query(dbq)
    dbresultlist = dbresult.getresult()
    oldhash = dbresultlist[0][0] # 0 is index for hashvalue in tuple
    dbldate = dbresultlist[0][1] # 1 is index for date in tuple
    dburlid = dbresultlist[0][2] # 2 is index for url in tuple

except:
    print "%s: Url does not exist in db." % url
    oldhash = None
    dburlid = None

# Comparing the two hashes:
# -----
newhash = self.hashit(txt)
if oldhash == newhash:
    print "No changes for url %s" % url
else:
    # ** If url has not been registered in table 'url' before
    if dblsi == None:
        # New url: Updating table 'url'
        # Retrieving largest value for column 'urlid' in table 'url':
        dbrcount = self.db.query("SELECT MAX(urlid) FROM url")
        dburlid = ((dbrcount.getresult()[0][0] + 1)
        # Updating table url

```

```

        self.db.query("INSERT INTO url (urlid, url, hashfrequency,
nextsearchdate, searchinterval) VALUES('%s','%s','%s','%s','%s')" % ( str(dburlid), url,
self.newhashfrequency, self.getnextdate(1), str(1) ) )
        # Printing status message
        print "Created new entry in db for url %s with id %s" % (url, str(urlid) )

# ** If url is already in table 'url', calculate new hashfrequency:
else:
    # Calculate number of days since last update
    numofdays = self.gettimediff(str(dbldate))

    # Calculate new average
    newhashfrequency = (dblhf + numofdays) / 2

    # Calculate new search interval
    newinterval = self.getnewinterval(numofdays, dblsi)

    # Calculate new search date
    newdate = self.getnextdate(newinterval)

    # Put new hash frequency, nextsearchdate, searchinterval into table 'url'
    updquery = "UPDATE url SET hashfrequency='%s', " % str
(newhashfrequency)
    updquery += "nextsearchdate='%s', searchinterval='%s' " %
(newdate,str(newinterval))
    updquery += "WHERE urlid='%s'" % (str(uid))
    self.db.query(updquery)

# Now - add this new hash value to table 'urlsearch'
# -----

# Retrieving largest value for column 'urlsearchid' in table 'urlsearch':
dbrcount2 = self.db.query("SELECT MAX(urlsearchid) FROM urlsearch")
dburlsid = (dbrcount2.getresult())[0][0]
if dburlsid == None:      dburlsid = 1
else:                    dburlsid += 1

# Retrieving system time [NOW]:
tm = self.gettime()

# Updating table 'urlsearch' with new hash value:
self.db.query("INSERT INTO urlsearch (urlsearchid, urlid, hashvalue, date)
VALUES('%s','%s','%s','%s')" % (str(dburlsid), str(uid), str(newhash), tm ))

# Printing status message
print "Updated hash value for url %s." % url

# End of code.

```