1 Innhold:

1 Innhold: 1
2 Innledning 2
3 Utstyrliste 2
4 Problembeskrivelse 2
5 Løsningsbeskrivelse 3
5.1 Brett og Astar 4
5.1.1 Sjekke om et brett er løselig 4
5.1.2 Å finne løsningen, bredde-først / A* 6
5.1.3 Manhatten Distance 7
5.2 GUI 8
5.2.1 Hvordan gjøre det spillbart 9
5.2.2 Spennende tilstander 10
5.2.3 Oppbygning av spillets GUI 11
5.2.4 Shuffle funksjonen 12
6 Konklusjon 12
6.1 Fremgangsmåte og tid: 12
6.2 Problemer: 13
6.3 Konklusjon: 14
7 Referanser & Kilder 14
7.1 A* 15
7.2 GUI og slikt 15
8 Vedlegg 15

2 Innledning

Denne siste oppgaven i år skal gi oss erfaring med å programmere grafer, søke i grafer og GUI programmering. Og sist, men ikke minst skal den gi oss litt spillteori å tygge på.

Vi skal programmere 8-puzzle.

3 Utstyrliste

- Microsoft Word 2003
- Microsoft Office Document Imaging
- Microsoft MSN Messenger
- Opera

- BlueJ
- Jcreator LE
- exe4j

- Adobe Photoshop
- Irfanview

4 Problembeskrivelse

Denne oppgaven går ut på å programmere 8-puzzle. 8-puzzle er et lite spill, der du har et brett med plass til 9 brikker. En av plassene er tom, og inn i det hullet kan man skyve en av nabobrikkene til hullet. Man skal altså omorganisere brettet til man kommer frem til en løsning.

Det var spillereglene. Over til mine regler. Vi skal først og fremst, naturligvis, lage et 8-puzzle som er spillbart. Et ikke spillbart 8-puzzle ville være heller kjedelig. Vi skal altså lage en GUI, med brikker, som skal kunne flyttes rundt til man finner en løsning. Da kommer vi over til neste lille problem som er å lage en funkjson som shuffler brettet på en slik måte at brettet fortsatt er løsbart. Å få et uløsbart 8-puzzle å løse er nemlig ikke

helt ok. Spilleren skal selvfølgelig også vite når han/hun har løst det. I tillegg til det fungerende spilllet, skal spilleren ha muligheten til å gi opp. Og når vi sier gi opp, mener vi ikke å ha muligheten til å lukke spillet for å finne på noe annet. Vi mener at spilleren skal kunne trykke på en knapp, og at spillet da skal løse seg selv, og ikke bare finne løsningen, men vise spilleren alle flyttene til løsningen, slik at spilleren ser at det var teit å gi opp da det var fult mulig å løse det selv.

Litt mer programmerisk er problemene som følger:

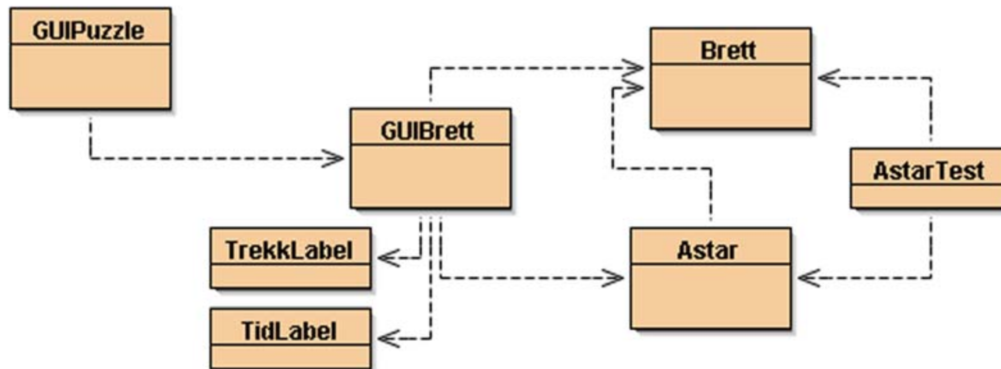
- Finne en god represetasjon av brettposisjonen spilleren er i
- Lage en funkjson som kan sikre at brettene spilleren får er løselige
- Finne en algoritme som kan ta brettposisjonen og finne løsningen ut i fra den
- Lage en fancy gui som gir spilleren lyst til å spille det c",)

Det var de grove trekkene. De mer konkrete problemene jeg møtte på kommer i løsningsbeskrivelsen.

5 Løsningsbeskrivelse

Dette er en oppgave jeg slet veldig med i starten. Det var veldig mye å gjøre, og når man får en slik oppgave i fanget føler man at man skal gjøre alt på en gang. Man ser på det, og det virker som at alt trenger noe annet for å fungere. Noe som vil si at man ikke har noe sted å starte. Det var selvfølgelig feil, og etter råd fra faglærer Hallstein fikk jeg til slutt til å abstrahere og finne et sted å starte. Og ideen var ikke dum. Jeg startet med å lage de indrefunkjsonene. Altså finne en bra måte å representere et brett på, lage en metode til å opprette tilfeldige, løselige brett, og til slutt skrive algoritmen til å løse det tilfeldige brettet. Når alt det var i boks startet jeg å legge et GUI rundt det jeg allerede hadde laget. Noen forandringer måtte selvfølgelig til når GUIet kom inn i bildet, men det var jammen ikke mange. Og de fleste var stort sett fordi jeg fant ut jeg ville bruke funkjsoner til flere ting enn de var tilegnet til i førsteomgang. Men uansett var abstraksjonen og innkapslingen og alt det der temmelig vellykket, for det å putte guiet på etterpå var overraskende lett. Koden skal være greit kommentert, men jeg tenkte jeg skulle si litt om hver av de her, og nevne litt om hvorfor jeg valgte å gjøre ting som jeg gjorde. Spesielle metoder i klassen tenkte jeg også å kommentere litt. Under har jeg satt et klassediagram hentet fra BlueJ som viser litt hvordan de forskjellige klassene

avhenger av hverandre. Det viser også at jeg har lite kopling, det vil si at klassene ikke er avhengige av hverandre på kryss og tvers. Noe som igjen resulterer i at det er relativt enkelt å forandre på ting uten at det har katastrofale følger i andre klasser.



5.1 Brett og Astar

Dette er de to klassene som danner grunnmuren i mitt spill. Og det jeg startet med var å finne en god representasjon av et brett. Jeg startet med en to-dimensjonell da jeg tenkte det var naturlig siden et brett nettopp er todimensjonell. Dette viste seg fort å være vanskelig å jobbe med, så gikk over til en-dimensjonell byte array. Noe som viste seg å være ypperlig. Mitt brett representeres altså med en byte array på lengde 9, som skal inneholde alle tallene fra 0 til 8, der 0 representerer hullet på brettet.

Så startet jeg på de funksjonene jeg med en gang så at man trengte. En toString overkjøring som gjorde at man kunne se hvordan brettet så ut i tekstform var det første. På den måten kan man System.out.println(brett) og få ut hvordan det ser ut. Gjorde ting mye lettere! Så var det å lage tilfeldige brett å jobbe med. Hvordan å løse det var ikke helt klart for meg i starten. Å putte ut alle tallene fra 0 til 8 på et brett er jo i og for seg lett nok, men gjør du det så har du ikke peiling på om det du har laget er et løseligbrett. Vi trenger altså en metode for å sjekke om et brett er løselig eller ikke. Måten jeg gjør det på er å telle inversions.

5.1.1 Sjekke om et brett er løselig

Et brett som er løst vil se ut som brettet under:

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{matrix}$$

Som vi ser her står alle brikkene i riktig rekkefølge, og antall inversions er dermed 0. Vi bytter nå om på noen av brikkene og teller på nytt:

$$\begin{matrix} 6 & 0 & 3 \\ 5 & 4 & 7 \\ 8 & 1 & 2 \end{matrix}$$

For å gjøre det enklere å se, la oss nå skrive opp brettet på en rekke:

$$6 \quad 0 \quad 3 \quad 5 \quad 4 \quad 7 \quad 8 \quad 1 \quad 2$$

Hvis vi starter på den først ser vi at denne er 6, den står naturligvis helt feil, og vi teller nå hvor mange den står feil i forhold til. 0 skal ikke telles, så vi starter med 3. 3 skulle vært foran 6, altså har vi 1 inversion. 5, 4, 1 og 2 skulle også vært foran 6. Altså har vi for 6, 5 inversions. Vi hopper over 0 og ser på 3. 3 står feil i forhold til 1 og 2, de større tallene kommer som de skal etter 3. Nå har vi altså 5 + 2 = 7 inversions så langt. Legg foresten merke til at vi bare teller den ene veien. 3 står riktignok feil i forhold til 6, men den har vi allerede telt. Fortsetter vi sånn bortover får vi at antallet inversions er 5 + 2 + 3 + 2 + 2 + 2 = 16. 16 er et partall, og brettet er derfor løselig. Beviset for det kommer her. Hvis vi nå starter med start brettet og skyver litt på brikkene (det vil si bytter null og en nabo til null):

$$\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{matrix} \longrightarrow \begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 0 & 8 \end{matrix} \longrightarrow \begin{matrix} 1 & 2 & 3 \\ 4 & 0 & 6 \\ 7 & 5 & 8 \end{matrix} \longrightarrow \begin{matrix} 1 & 2 & 3 \\ 4 & 6 & 0 \\ 7 & 5 & 8 \end{matrix} \longrightarrow \begin{matrix} 1 & 2 & 3 \\ 4 & 6 & 8 \\ 7 & 5 & 0 \end{matrix}$$

Ser vi på brett 1 så ser vi at dette er løsningsbrettet, alle brikkene står riktig, og inversions er lik 0. Ser vi på det neste brettet så har vi flyttet på 8. Vi har flyttet horisontalt, og hvis vi teller inversions, så ser vi at det fortsatt er 0! (husk at vi ikke teller med 0). I neste flytt, flytter vi vertikalt, hva skjer da? Teller vi inversions nå får vi 2. Neste flytt er horisontalt igjen, og vi ender opp med en inversions som fortsatt er 2. Flytter så vertikalt igjen, og ender opp med en inversions på 4! Hva ser vi av dette? Jo, horisontal flytting forandrer ikke inversions, noe som er logisk nok. Flytter du vertikalt derimot, forandrer du inversions med 2. Dette er faktisk også ganske logisk, når du ser at det er to plasser imellom hvert vertikale flytt. Altså, hvis du tenker deg brettet som en lang rekke, vil det siste flyttet vårt resultere i at 8, som var etter 7 og 5, nå kom foran, og

økte inversions med 2. Vi ser derfor at uansett hvordan vi flytter rundt på brettet så vil inversions alltid være et partall. Hvis vi nå utfører et ulovlig trekk, og for eksemlpel bytter plass på 7 og 8 i det første brettet ser fort at inversions vil bli 1. Hvis vi så prøver å flytte rundt som vanlig, med lovlige trekk vil vi alltid ende opp med et oddetall bare fordi vi byttet om på de to brikkene. Og vi vil dermed aldri klare å komme oss tilbake til 0 inversions der brettet er løst.

Tilbake til spillet. Det finnes 9! = 362880 forskjellige brett. Halvparten av disse er løselige, den andre halvparten ikke. Med en metode som lager tilfeldigebrett er det derfor 50:50 sjangse for å treffe et løselig et. Og da en datamaskin er temmelig rask, fungerer det greit å bare lage tilfeldige brett til man finner et løselig et, og problemet er løst.

Vi har nå en funkjson til å lage løselige brett, neste steg er funksjonen som løser dem.

5.1.2 Å finne løsningen, bredde-først / A*

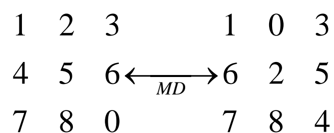
Her kommer vi til dette med grafer og algoritmer. Her slet jeg mye i starten. Jeg forstod hvordan disse dybde- og bredde-først metodene fungerte, men fatta ikke hvordan jeg skulle få programmert de. Jeg surra rundt på internet og endte opp på to sider om A* som var utrolig gode. Det disse sidene også hadde var pseudokode, som gjorde at jeg klarte å programmere A* algoritmen. God pseudokode klarte jeg nemlig ikke å finne for bredde-først eller dybde-først søking. Kanskje fordi de ikke er spesielt gode. A* er en pathfinding algoritme som, ved hjelp av diverse verktøy jeg kommer tilbake til, finner den korteste veien fra en node til en annen. I vårt tilfelle er noden et brett, og veien er trekkene i mellom dem:

```
          1 2 3
          4 5 6
          7 8 0
        /       \
  1 2 3           1 2 3
  4 5 6           4 5 0
  7 0 8           7 8 6
```

Pathfinding algoritmens oppgave blir da å finne veien fra et brett til et annet. Pathfinding er en særs komplisert affære. Og skal jeg forklare den veldig her kommer denne rapporten til å bli ufattelig lang. Jeg henviser derfor til kildene mine der det er to internetsider og en bok der man kan lese om det. Var også de jeg brukte for å programmere min A*. Er Koden er kommentert også, så det eneste jeg har valgt å kommentere mer her er Manhatten Distance.

5.1.3 Manhatten Distance

Veldig kort, fungerer A* ved å ha forskjellige brett på to lister. Den henter ut et brett fra listen Open og ser om det er løsningen. Er det ikke det finner den etterkommerne til brettet, ser om de har vært undersøkt før, og hvis ikke putter de på Open, og putter så det brettet den startet med på Closed. Forskjellen mellom A* og for eksempel Brede-først, er at den velger å sjekke de brettene som ser ut til å føre til løsningen raskest. Og til dette bruker vi Manhatten Distansen. Har en link til det i kildene også, så skal bare forklare litt om den her. Manhatten Distance mellom to brett er i virkeligheten summen av avstandene mellom hver brikke på to brett. Vanligvis måler vi avstanden mellom to punkter, i vårt tilfelle brikker ved formelen $\sqrt{(x2-x1)^2+(y2-y1)^2}$. Dette vil naturligvis ikke fungere, da vi ikke kan flytte vertikalt. Dessuten om vi hadde gjort det ville vi som vi så i 5.1.1, ende opp i et uløseligbrett. Vi har derfor en annen metode å måle avstand på. Den kommer faktisk fra taxi-sjåfører som jobbet på Manhatten. Der kan du nemlig ikke kjøre diagonalt mellom to punkter, da det er svære blokker der, du må kjøre i rette linjer, i en x og en y akse. Den korteste avstanden mellom to punkter målt med MD blir da $|x2-x1|+|y2-y1|$. Kan ta et eksempel for å vise hvordan det fungerer. Vi har de to brettene under. Det til venstre er løsningen, og det til høyre er byttet litt rundt på. (brettet er faktisk også uløselig, men det ser vi bort i fra her, da vi uansett bare skal finne avstanden i mellom de).



$$\begin{matrix}1 & 2 & 3\\4 & 5 & 6\\7 & 8 & 0\end{matrix}\xleftrightarrow[MD]{}\begin{matrix}1 & 0 & 3\\6 & 2 & 5\\7 & 8 & 4\end{matrix}$$

Vi tar så hver brikke for seg, unntatt 0, og måler avstanden mellom der brikken er, og der den skulle ha vært, ved hjelp av MD. 1 står på riktig plass. 2 skulle vært 1 opp. 5 skulle vært en til venstre. 4 skulle vært en opp pluss 2 til venstre. Gjør vi slik for alle brikkene, og summerer får vi 0 + 1 + 0 + 3 + 1 + 2 + 0 + 0 = 7. Dette er et estimat, og vi naturligvis

undervurdere avstanden litt mellom to brett, siden man vanligvis må flytte rundt en del ekstra for å få en brikke dit den skal. Men det er det som er så bra, for når vi underestimerer er vi garantert at A* alltid velger den som er kortest i fra målbrettet, og ikke overburderer og finner en som er lengre unna enn man egentlig hadde trengt.

Grunnet dette estimatet og den selektive undersøkelsen blir A* til en av verdenes raskeste Pathfinding algoritmer. Jeg lagde en test klasse som satte A* metoden til å løse 6000 tilfeldige brett. Klassen tok tiden, summerte antall brett A* undersøkte hver gang, og hvor mange trekk løsningen var på. Når den var ferdig regnet jeg, dvs klassen, ut gjennomsnittet. Har sett andres oppgaver der de har brukt Bredde-først søk, og sett hvor lang tid de bruker og hvor mange brett de undersøker. Og det tok ofte en god stund og det ble undersøkt ofte flere ti-talls tusen brett. Da jeg så resultatene fra testen min skjønte jeg hvor effektiv A* faktisk var. Jeg fikk nemlig en gjennomsnittelig løsningstid på 635 ms! Og antall brett den gjennomsnittlig undersøkte var 1779. Og med tanke på at det er 181 440 mulige brett å komme til (9! / 2, da den andre halvparten er uløselige og man ikke kommer til de) så er det ganske så fantastisk synes i all fall jeg. Noe annet som er litt stilig er at denne A* algoritmen er tilsvarende i mange andre typer spill og oppgaver også, så når man har mestret den er veien åpen for mye morsom programmering c",)

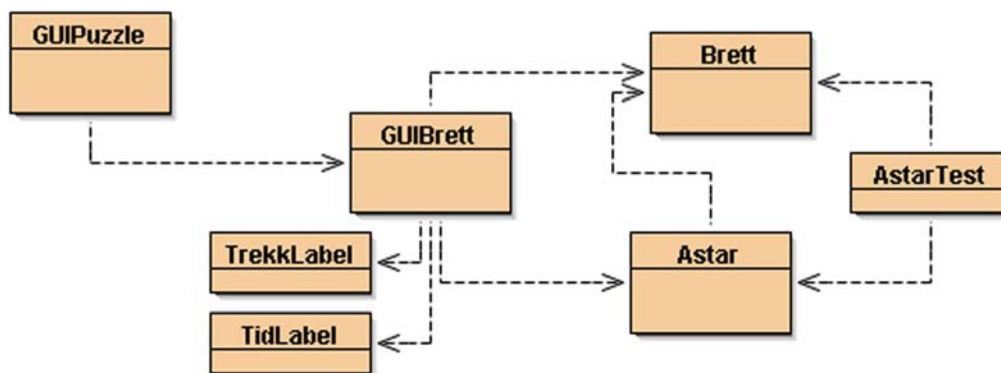
Men nå tenkte jeg å henvise til kildekoden som er lagt ved, i papirform og på cd, og fortsette til skallet utenpå.

5.2 GUI

GUI er en ting jeg aldri har forstått mye av. Det vil si før denne oppgaven, for under denne oppgaven har det gått opp mange lys for meg. Og selv om jeg fortsatt ikke helt liker greia (muligens litt bortsjemt fra Macromedia Flash, som er laget for nettopp animering og grafikk), så har jeg blitt mye mer venner med Java sin GUI api, eller hva man kaller det.

Første poeng som bør sies er at jeg valgte å lage hele GUIen min i en klasse som arvet fra JPanel. På den måten kunne jeg enkelt lage en liten klasse som arvet fra JFrame, og putte spillet inn i den. Dette virker kanskje litt rart, men det hadde en baktanke. Jeg

hadde nemlig lyst til å også lage en JApplet som kunne inneholde samme JPanel og som jeg kunne putte ut på hjemmesiden min. Det har jeg da ikke klart. JFramen min fungerer ypperlig, med min spill panel inni. JApplet, not so much. Siden lastet, applet kom opp, men der sto det ikke annet enn Applet Crashed. Jeg begriper ikke hvorfor, og har ikke klart å finne ut av det heller, men det akter jeg å gjøre senere etter at denne oppgaven er levert inn. Når jeg har brukt så mye tid på å lage et stilig 8-puzzle spill vil jeg nemlig veldig gjærne ha det inn på siden min. Men nok om det jeg ikke fikk til. Over til hvordan og hvorfor jeg gjorde mitt GUI som jeg gjorde. Her er klasse diagrammet igjen:



Når man skal lage et spillbart 8-puzzle er det først og fremst en ting man trenger, og det er brikker som man man skyve rundt på et brett. Å utføre dette var en tanke jeg slet med en liten stund. Det er for det første litt komplisert, og for det andre vil det bli litt rotete kode om man skal drive å flytte rundt på brikker. Men til slutt gikk det et lys opp for megog jeg vente meg til Brett klassen min, som jeg jo allerede hadde skrevet. Brett klassen inneholder jo i teorien allerede alt det man trenger! Den representerer et brett, og den har en funkjson som finner mulige trekk ut fra brettet (den Astar bruker til å søke i brett-grafen). Og mer enn det trenger en jo ikke. Det eneste man da trenger i GUI klassen er å vise Brettet man er på, grafisk, og å kunne bytte til et annet brett når spilleren trykker på en knapp som er et lovlig trekk.

5.2.1 Hvordan gjøre det spillbart

Det jeg gjorde var å ha 9 knapper i et JPanel inni GUIBrett. Disse knappene er i en array, slik at det er lett å gå igjennom de. Bildene oppbevarer jeg i en annen array (det vil si, egentlig to, en for aktive brikker, og en for ikke aktive.). Skal vi så vise brettet, som i en en-dimensjonell array er representert ved { 3,5,1,4,2,6,7,8,0}, kan vi bare gå igjennom knappene i knappe arrayet. Putte bilde 3 på knapp 1, bilde 5 på knapp 2, og så

videre. Så var det å kunne skifte brett. Her tenkte jeg på muligheter som å ha en switch og en case for hver knapp, og lignende, men jeg synes det måtte finnes en enklere metode, og jeg tror jeg fant en ganske optimal en. Jeg gikk til Brett klassen og laget tilleggs metode som brukte metoden som fant naboene. Det den gjør er å gi en Brett array, der de ulovlige trekkene er null, og de lovlige representert ved Brettet man ville komme til hvis man flyttet den brikken. Altså vil for eksempel brettet under gi arrayet ved siden av:

$$\begin{matrix} 5 & 1 & 2 \\ 6 & 4 & 7 \\ 3 & 8 & 0 \end{matrix} \xrightarrow{gir} \left\{ null, null, null, null, null, \begin{matrix} 5 & 1 & 2 \\ 6 & 4 & 0 \\ 3 & 8 & 7 \end{matrix}, null, \begin{matrix} 5 & 1 & 2 \\ 6 & 4 & 7 \\ 3 & 0 & 8 \end{matrix}, null \right\}$$

Når jeg hadde mekka denne metoden var resten like enkelt, som det etter min mening er genialt. Knappene i GUIet er merket med setActionCommand() til å ha ActionCommand verdi lik plassen knappen har i knapp arrayet. Alle knappene har også samme ActionListener. Hva som skjer kan kanskje lettes forklares ut i fra eksemplet over. Hvis nå spilleren klikker på 6, altså knapp nummer 3 i arrayet. Da vil ActionListeneren hente arrayen over og se om det er et brett i plass 3. Det er det ikke, altså er det ikke et lovlig trekk, og vi lar være å gjøre noe med det. Hvis spilleren istedet klikker på 7, som er knapp nummer 5, ser vi at i arrayen som brettet gir så er det et brett i plass 5! Da bytter vi brettet vårt til det, putter bilder på knappene etter det nye brettet, og vi har utført et trekk. Kildekoden bør gjøre dette enda klarere.

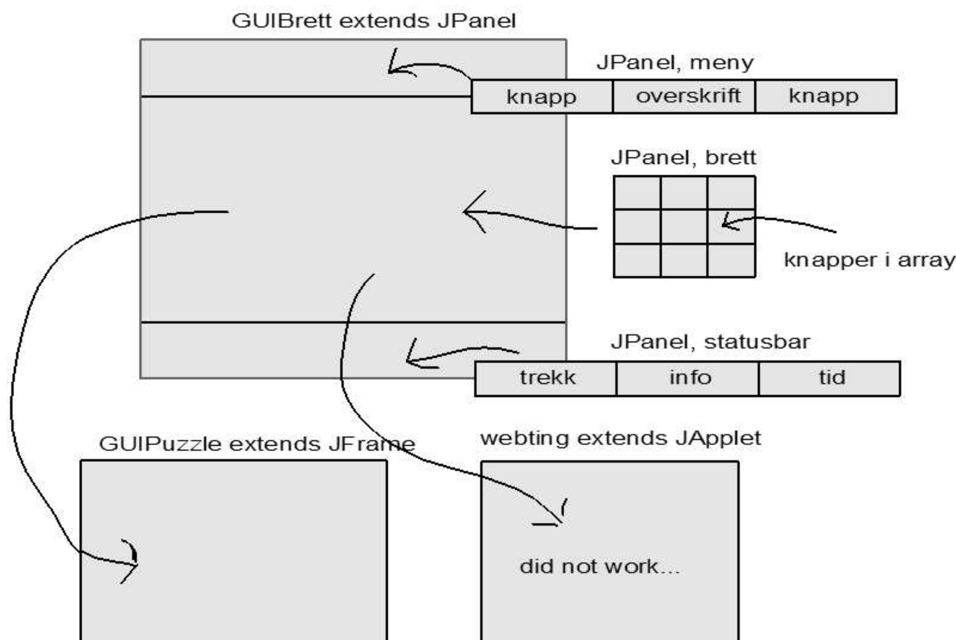
5.2.2 Spennende tilstander

Andre ting jeg har valgt å gjøre er opprerere med tilstander. Innspirert av en maskinvare oppgave der vi laget en tilstandsmaskin, som skulle opperere et lyskryss. Når noe skjedde så den hvlken tilstand den var i, og skiftet lysene og et par andre ting ettersom hvilken tilstand den er i. Det fant jeg ut av at jeg kunne bruke her også. Jeg satte meg ned med pen og papir og tenkte meg igjennom hvilke tilstander jeg trengte, og fant ut at jeg ikke trengte flere enn 5. Disste tilstandene har jeg definert som tall-konstanter. På den måten kan jeg i metoden som oppdaterer brettet og slikt ha en switch som setter knapper etter hvilken tilstand man er i. Hva som skjer i hver tilstand ser man i koden, men kan jo ta et eksempel. En av tilstandene er BRETT_AKTIV. Det er tilstanden som man er i når spilleren kan spille. De to knappene på brettet blir da satt til Nytt Brett, til hvis du blir lei av det du holder på med og til Finn Løsning, til å få brettet til å løse seg selv. Når det gjelder brikkene vil de i BRETT_AKTIV bli satt til å ha et fargebilde, og til å

være klikk-bare. Unntatt brikken som representerer 0, som vil være deaktivert og med et grått bilde. Har delt det i to metoder, slik at man kan skifte brett uten å skifte tilstand. Skifter man tilstand, skiftes forøvrig brettet også. Det vil si, om knappene er aktive eller slikt skiftes. Ikke brikke oppsettet. Høres kanskje avansert ut, men er relativt enkelt når man ser i kildekoden.

5.2.3 Oppbygning av spillets GUI

Under har jeg forsøkt å tegne de forskjellige lagene på spillet mitt og tenkte å si litt om hver bit, slik at når man ser på kildekoden så er det litt enklere å se hvordan det henger sammen:



Som du ser har jeg to knapper øverst. Disse har funksjon og navn ettersom hvilken tilstand man er i. Overskriften er en vanlig Jlabel som stort sett ikke er annet enn en overskrift. Samme med info, som bare viser navnet mitt og litt sånt. Trekk og tid er derimot litt mer interessante. Jeg mener nemlig at når man spiller et spill bør man kunne vite hvor lang tid man har brukt, og hvor bra man har gjort det. Uansett, så synes jeg det ville bli surr å putte deres funksjoner inn i GUIBrett. Jeg tok derfor å laget to separate klasser kalt TidLabel, og TrekkLabel. Disse to klassen arver begge fra Jlabel og kan derfor settes enkelt inn som vanlig. Forskjellen er at de har tillegsmetoder som å starte

og stoppe tiden, og å øke og resette telleren. Igjen henviser jeg til kildekoden, men nå har i hvert fall nevnt at de er der for en grunn.

Nå tror jeg egentlig jeg har kommentert det meste som man bør kommentere, men kan nevne en siste ting jeg valgte å gjøre for å få det litt stilig.

5.2.4 Shuffle funksjonen

Som sagt har brettet mitt en metode for å legge ut brett. Min shuffle (nytt brett) knapp tok i starten kun å la ut et randombrett. Men så slo det meg at jeg kunne med små justeringer få en animert shuffling. Det jeg gjorde var å gjøre så Manhatten Distanse metoden i Brett ikke var låst til å finne avstanden til løsningsbrettet. Når det var gjort kunne jeg så justerte A* til kunne ta inn et start og et slutt brett, istedet for bare et start brett, og på den måten finne veien mellom to hvilke som helst brett og ikke kun et brett til løsningsbrettet. Med andre ord bruker jeg A* både til å finne løsningsbrettet og til å shuffle brettet. Så der har vi jo et godt eksempel på konsistens (cohesion og veldefinerte enheter) og gjenbruk av kode c",)

6 Konklusjon

6.1 Fremgangsmåte og tid:

Dette er en oppgave jeg egentlig har brukt ufattelig lang tid på. Men det er fordi jeg har vært så usikker, og det er mye jeg ikke har kunnet fra før. Rota foreksempel masse med GUI, før jeg i det hele tatt begynte så vidt med GUI på denne oppgaven. Tiden jeg har brukt er derfor litt vag og udefinert. Men jeg vet at jeg etter en halv dags lesing på A* klarte å programmere Brett og Astar klassen på en halv dag til (ja jeg var oppe særs lenge den natta). Det samme skjedde egentlig med GUI. Rota rundt på dagen, og plutselig spratt ideene inn i hodet som perler på en snor, og hadde det relativt fungerende etter en del timer. Så har jeg jo finpussa og lagt til litt funksjoner her og der etter det igjen. Rapporten derimot har jeg tiden på. Startet en gang mellom 15-16 i dag, og klokken er nå straks 23. Har altså brukt en 7-8 timer nå, og blir vel kanskje litt til når jeg leser over etterpå. Men når man er så dum å bruke masse spess løsninger på

problemene så er det vel ikke annet å vente enn å måtte bruke litt mer tid på å forklare dem alle sammen... c”,)

6.2 Problemer:

Problemer var det drøssevis av, men de fleste var av mindre størrelse og relativt greie å løse. Eneste jeg ikke har klart å løse er det å få det inn i en JApplet. Men for ordensskyld kan jeg jo nevne de to problemene som har skapt mest hodebry for meg, og som jeg har lært mest av. Lærte i alle fall en viktig lekse på den første av de:

Hodebry #1: Astar klarte ikke alltid å finne en løsning. Fant løsning av og til, mens andre ganger ble den bare stående å surre. Jeg forstod lite, og begynte å strekke Astar metoden ut (dele skritt i to for å få det mer oversiktlig og slkt) , System.out.print’e opp og ned og i mente, debugge og surre å stå i. Måtte til slutt få Hallstein til å se på det, og det viste seg at Astar metoden min var bra. Problemet lå nemlig i funksjonen som testa om brett var løselige eller ikke. Resultatet den ga var feil. Når den var fiksa fungerte alt som det skulle. Så nå har jeg lært å sjekke de små metoden først. Vet ikke hvor god sammenligneingen er, men hvis huset du prøver å bygge faller sammen gang på gang, kan det ofte være en ide å sjekke om mursteinene du bruker holder mål, før du begynner å tegne om på huset. Uansett, fra nå av tar jeg det ikke forgitt at en metode gjør som den skal. En liten test holder ikke alltid

Hodebry #2: Neste problem jeg hadde med A* var tiden den brukte. Kjørte en test på den, som løste 4000 tilfeldige brett, og den jobbet \relativt kjapt, et snitt på 3,2 sekunder på å finne en løsning. Og det kunne jeg ha vært fornøyd med, hadde det ikke vært for at den hadde et worst-case på 220 sekunder! Hva dette kom av har jeg faktisk ikke direkte funnet ut av, men jeg har klart å løse det, og har en teori. For da jeg tok programmet ut av BlueJ og kjørte det fra Jcreator istedet, så gikk det så det suste! Et snitt på ca 600 ms, og bare 11 av de 6000 gangene jeg kjørte den tok det mer enn 20 sekunder. Min teori er at BlueJ er et Java program, og at koden man kjører på en måte blir kjørt inni det igjen. Mens i Jcreator kjøres det som et eget Java program. Kommer ikke til å gå helt over til Jcreator av den grunn, men kommer nok til å flytte meg over dit i sluttfasen av programmer. Jcreator og andre programmer mangler nemlig det å kunne teste ut metoder og objekter uten å ha et fungerende program med main metoder og slikt. Det har BlueJ og det er rett og slett genialt!

6.3 Konklusjon:

Dette er den hittill morsomste oppgaven jeg har møtt på i min tid som dataingeniørstudent. Jeg har lært så utrolig mye, og skal jeg være helt ærlig har jeg ofte sittet å vært imponert over hva jeg faktisk har klart å utrette. Ting jeg ikke trodde jeg skulle klare, eller forstå noe som helst av, fallt plutselig på plass. Plutselig flyttet brikkene på seg og brettet var løst, og det var jeg som hadde fiksa det! I tillegg til at jeg har fått til mye, har jeg forstått mye. Spesielt fordi jeg har hjulpet en 3-4 stykker med en del. Har ikke gitt dem noe kode direkte, men har forsøkt å forklare hvordan jeg har gjort ting. Og en ting er å skjønne noe selv, en annen ting er å forklare det til andre så de skjønner det. En sak jeg var usikker på i starten var måten å finne nabobrett på. Dette med at når null ikke er øverst så kan man flytte opp og greier (se kildekode), var noe jeg slet med lenge. Fikk til å implementere det til slutt, men det var fortsatt litt sånn halvveis uklart. Men etter å forklart det til 3-4 stykker, så har jeg faktisk blitt veldig sikker på det. Og nå tror jeg at jeg skal klare å forklare folk det uten å måtte se bort på min egen kode hele tiden c",)

Har brukt ganske mye tid på denne oppgaven, men mye av tiden har gått med til å lese på ting som jeg ikke skjønte. A* og gui for eksempel. Men en ting som er bra og for meg litt imponerende og gøy, er at enda jeg har lest og jobbet så utrolig mye, så har jeg hatt det gøy hele tiden og egentlig aldri blitt lei. Det er faktisk sånn at jeg nå er litt trist over å ha den ferdig og måtte levere den fordi den var så gøy å jobbe med. Og dette beviser minst en ting: Jeg er på riktig høyskole og på riktig studie :-D

7 Referanser & Kilder

- Karoline Moholth
- Hallstein Hansen
- Eivind Eriksen
- Hans Engebretsen
- JavaDoc
- http://java.sun.com/docs/books/tutorial/essential/threads/

7.1 A*

- http://en.wikipedia.org/wiki/Manhattan_distance
- http://en.wikipedia.org/wiki/A-star_search_algorithm
- http://www.policyalmanac.org/games/aStarTutorial.htm
- http://www.geocities.com/jheyesjones/astar.html
- ”New Riders Publishing: Core Tecniques and Algorithms in Game Programming”
  - Kapittel 8, Tactical Thinking Explained

7.2 GUI og slikt

- http://java.sun.com/docs/books/tutorial/applet/index.html (blæ)
- Deitel & deitel, ”Java: How To Program, fifth edition”
  - Kapittel 13, Graphical User Interface Components: Part 1

8 Vedlegg

- Cd med følgende:
  - - Puzzle: fungerende 8-puzzle med exe-fil laget med exe4J
  - Dokumenter: this, ++
  - Source: kildekoden til mitt 8-puzzle
  - Utskrift: utskriften (ms document imaging)