

# Planlægning med LEGO agenter



Anders Rasmussen [s042410]  
Bo Kanstrup Hansen [s042319]

Kongens Lyngby 2007  
IMM-B.Sc-2007-7

Technical University of Denmark  
Informatics and Mathematical Modelling  
Building 321, DK-2800 Kongens Lyngby, Denmark  
Phone +45 45253351, Fax +45 45882673  
[reception@imm.dtu.dk](mailto:reception@imm.dtu.dk)  
[www.imm.dtu.dk](http://www.imm.dtu.dk)

# Abstrakt

---

Evnen til at tænke logisk er for de fleste mennesker en selvfølge som alle tager for givet. Hvornår og hvordan vi har fået denne evne eller hvordan den logisk fungerer er der ingen der rigtig ved. Robot industrien har i lang tid prøvet at efterligne menneskets hjerne og dermed dens logik, som lægger grundlaget for menneskets intelligens.

Denne rapport vil designe og implementere kunstig intelligens i form af planlægning i en LEGO Mindstorm NXT robot, der gør robotten i stand til at løse opgaver som kræver flere dertil følgende handlinger.



# Forord

---

Denne rapport dækker forløbet af at designe og implementere en planlægnings algoritme til styring af en LEGO Mindstorm NXT robot.

Rapporten er inspireret af midtvejsprojektet ”Developing Multi-Agent Lego Robotics”[7]. Vi forventer at læseren er bekendt med dette projekt.

Det vil ydermere være en fordel at have kendskab til algoritmer og data strukturer samt parallelle systemer svarende til pensum i kurserne Algoritmer og datastrukturer I og II samt Parallelle systemer på DTU.

Vi vil gerne takke vores vejleder assisterende professor Thomas Bolander for bidragelse til ide udvikling gennem forløbet.

Lyngby, Juni 2007

Anders Rasmussen [s042410]  
Bo Kanstrup Hansen [s042319]



# Indhold

---

<b>Abstrakt</b>	<b>i</b>
<b>Forord</b>	<b>iii</b>
<b>1 Introduktion</b>	<b>1</b>
1.1 Problem beskrivelse . . . . .	1
1.2 Projekt mål . . . . .	2
<b>2 Planlægning</b>	<b>3</b>
2.1 Introduktion . . . . .	3
2.2 Kravspecifikation . . . . .	3
2.3 Teori . . . . .	4
2.4 Analyse . . . . .	15
2.5 Design . . . . .	18
2.6 Implementering . . . . .	23

2.7	Test	24
2.8	Udvidelser	25
2.9	Diskussion	26
2.10	Konklusion	29
<b>3</b>	<b>LEGO Mindstorm NXT Robot</b>	<b>31</b>
3.1	Introduktion	31
3.2	Kravspecifikation	31
3.3	Analyse	32
3.4	Design	33
3.5	Implementering	35
3.6	Test	37
3.7	Diskussion	41
3.8	Konklusion	42
<b>4</b>	<b>Multiagenter</b>	<b>45</b>
4.1	Introduktion	45
4.2	Kravspecifikation	46
4.3	Teori	46
4.4	Analyse	49
4.5	Design	52
4.6	Implementation	56
4.7	Test	57

---

4.8 Diskussion . . . . .	59
4.9 Konklusion . . . . .	60
<b>5 Konklusion</b>	<b>61</b>
<b>A Pseudokode</b>	<b>65</b>
<b>B Grafisk brugergrænseflade</b>	<b>69</b>
<b>C Kausal link graf</b>	<b>71</b>
<b>D Kildekode</b>	<b>73</b>
D.1 Plan.cs . . . . .	73
D.2 POPstar.cs . . . . .	80
D.3 Action.cs . . . . .	86
D.4 Conditions.cs . . . . .	103
D.5 Variable.cs . . . . .	107
D.6 Program.cs . . . . .	108
D.7 NXTControl.cs . . . . .	115
D.8 Agent.cs . . . . .	119
D.9 World.cs . . . . .	121
D.10 PutOnShoe.cs . . . . .	132
D.11 ActionBlockWorld.cs . . . . .	135
D.12 ConditionBlockWorld.cs . . . . .	139
D.13 UnitTest.cs . . . . .	144

D.14 [Link.cs](#) . . . . . 152

# Introduktion

---

Robotter bliver en større og større del af hverdagen i fremtiden, hvor trivielle, og på sigt mindre trivielle, opgaver vil blive overtaget af robotter. Dette kræver at robotterne lærer at tænke helt eller delvist autonomt samt fortolke den verden de befinder sig i. For at robotter kan foretage sig noget fornuftigt, kræver det at de kigger frem i tiden og planlægger hvordan de vil løse en given opgave. Denne opgave kunne f.eks. være at en robot på et containerskib skal hente en container som befinder sig under andre containere og derved skal lægge en plan for hvordan den kan få flyttet containerne ovenover, så den kan nå ned til den givne container. Et andet eksempel kunne være en kranvogn robot som skal samle ødelagte biler op i Manhattan's gader og bringe dem til deres respektive værksteder. De ødelagte biler kan ikke bevæge sig selv og kranvognen kan ikke køre forbi andre ødelagte biler uden at flytte dem først.

## 1.1 Problem beskrivelse

Moderne computersystemer er i dag blevet så komplekse at de kan udfører beregninger og tal manipulationer med en kolossal hastighed, der langt overstiger menneskets fatteevne. De kan beregne  $\pi$  med en nøjagtighed, der hvis den skulle printes ud ville fylde en mindre samling af telefonbøger. De kan på stort set ingen tid finde alle primtal op til 1.000.000 og så videre.

Paradoksalt er det dog, at de ikke kan løse problemer som selv små børn kan finde ud af. Åbne en kagedåse, fjerne en kasse for at komme forbi, bygge et fort etc. Selv hvis computersystemet havde de samme handlings muligheder som barnet ville det ikke ane i hvilken rækkefølge tingene skulle gøres i, og hvilke ting der skal benyttes. At prøve sig frem ved at undersøge tilfældige rækkefølger af handlinger, vil lynhurtigt overstige selv en computers imponerende regne kraft. Før et computersystem kan løse selv sådanne mindre planlægnings opgaver, vil den kunstige intelligens alsidighed ikke kunne sammenlignes med menneskets.

I denne rapport vil vi undersøge en generel tilgang til problemet og forsøge at lave en planlægnings platform, der *forholdsvist* simpelt kan konverteres til at løse en række simple, men forskellige problemer. Idéen er at vha. en forenklet verdens beskrivelse og et sæt af handlings muligheder kan systemet *planlægge*; altså skildre hvorledes du kan opnå en ønskelig tilstand i dit miljø.

De alsidige mål som vi vil udsætte systemet for er, at bevæge sig rundt på et afgrænset kort og arrangere genstande på specificerende positioner; Bygge stabler af kasser i den rigtig rækkefølge; Sørge for at en professor tager sine sokker på før sine sko.

Vi vil undervejs kigge på muligheder og begrænsninger i planlægnings feltet.

## 1.2 Projekt mål

For at udspecificere problem beskrivelsen er vores overordnede mål at udvikle og implementere kunstig intelligens i form af planlægning til en LEGO Mindstorm NXT robot. Robotten skal kunne løse simple men ikke trivielle opgaver på en fuld kendt og lukket bane. Derudover vil vi undersøge hvordan planlægning tilføjes til BDI agenter samt en koncept løsning ”proof of concept“ til dette.

Første del består i at designe og udvikle en algoritme, som via et forholdsvist simpelt beskrivelses sprog kan lægge en plan for hvordan en given opgave løses med de handlinger som er til rådighed.

Anden del består af at transformere LEGO robotens verden ned i dette beskrivelses sprog og gøre det muligt at eksekvere en evt. plan.

Tredje del er en analyse af hvordan planlægning kan bruges i sammenhæng med andre robotter, hvor hver robot arbejder autonomt som en agent. Ressource og informationsdeling vil være nøgle termer.

# Planlægning

---

## 2.1 Introduktion

Planlægning er en gren af kunstig intelligens, som beskæftiger sig med at lægge strategier for hvordan et forløb skal udføres. Vi vil her give en gennemgang af hvordan sådanne problemer formelt beskrives, samt hvordan disse løses. Kapitlet ender ud med en færdig implementation af en planlægnings algoritme, som kan løse forskellige opgaver.

## 2.2 Kravspecifikation

Der ønskes designet og implementeret en algoritme, som via et beskrivelses sprog tolker problemstillingen og returnere en mulig løsning, hvis en sådan eksisterer. For at planlægningsdelen kan siges at være udviklet, specificerer vi følgende:

**Verdens beskrivelse:** Systemet skal implementere et sprog eller datasæt, som kan lave en simplificeret repræsentation af verden. Sproget skal endvidere kunne bruges til at specificere hvilke forhold der ønskes skal gælde efter eksekvering af en strategi.

**Handlings muligheder:** For at manipulere med tilstande i et miljø, må systemet kunne bruge et sæt handlings muligheder. En handling vil kunne fastlægge krav til miljøet før dens eksekvering og en virkning efter. Handlingerne skal designes af brugeren afhængigt af problem specifikationerne.

**Planlægning:** Udfra et sæt af handlings muligheder, en verdens beskrivelse og mål tilstand, skal der returneres en *plan*; Planen udgøres af en fastlagt rækkefølge af handlinger der sørger for at verden transformeres til en ønskelige tilstand. Systemet skal være deterministisk og returnere komplette, konsistente strategier. Givet kompleksiteten forventes dog mulighed for uforudsete anomaliteter.

**Heuristik** Da vi påberegner at planlægnings delen formentlig vil have en eksponentiel worst-case køretid, vil en fornuftig heuristik være et krav for tilfredsstillende præstations ydelse.

## 2.3 Teori

I dette afsnit gives et overblik over baggrundsteori indenfor planlægnings feltet. Hovedproblemet har været at skabe et automatiseret system, der ud fra et sæt af handlingsmuligheder og forudsætninger, kan kombinere mulighederne til at opnå et givent mål.

I undersøgelsen af et sådant system ligger det naturligt for, først at definere standard termer indenfor planlægning; Systemets opgave består i at generere en plan. dvs. en beskrivelse af et sæt af operationer, der ved eksekvering udgør en af løsningerne på et problem. Ved hjælp af operationerne ændrer systemet betingelser i en kendt simplificeret verden, over til en ønskeligt tilstand.

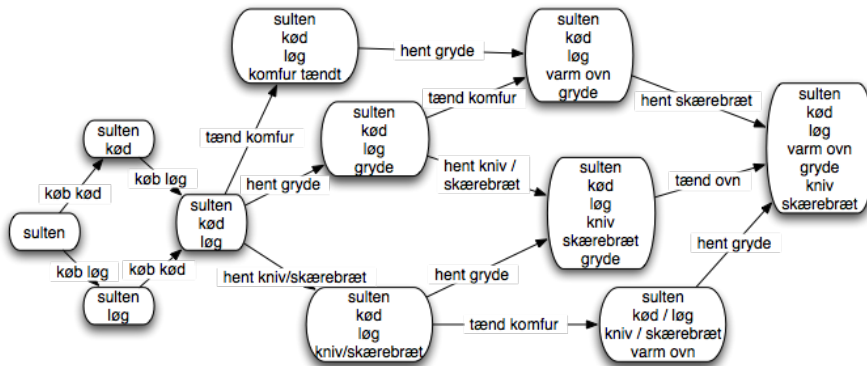
En operation er en handlingstype, der tager et sæt af forhånds-betingelser (Eng: preconditions) og hvis de er opfyldt, erstatter dem med et sæt af effektbetingelser (Eng: postconditions el. effects). Effektbetingelserne udgør de konjugerede virkninger af handlingen, der gælder efter udførsel. En handling er en instancieret operation - dvs med specifikke betingelser tilknyttet.

En betingelse kan betegnes som fakta om en simplificeret del af en tilstand. Et sæt af betingelser udgør en beskrivelse af en verden. I planlægningsmæssigt sammenhæng er betingelser ofte beskrevet vha. STRIPS el. lignende (se afsnit 2.3.1 om beskrivelses sprog). En betingelse er typisk betegnet enten som en boolsk værdi eller vha. en eller flere variabler. Variabler behøver ikke nødvendigvis at være bundet til en konstant under plansøgningen. En plan med alle variabler bundne kaldes fuldt instantieret. En plan med en eller flere ubundne variabler

kaldes delvist instantieret.

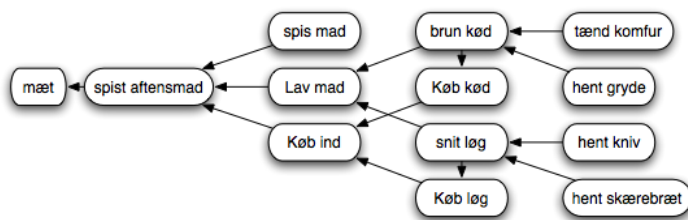
Et planlægningssystem der søger fremad og ud fra startbetingelserne forsøger at nå målbetingelserne kaldes et progressivt system. Et sådant system vil lide under at irrelevante konditioner i startbetingelserne vil generere et kæmpe søgetræ af handlinger der aldrig vil kunne bidrage til en løsning. Eksempelvis hvis problemet er, at man ønsker en bold 'Å' ind i et mål, vil det være omsonst først at undersøge om man opnår løsningen ved at sparke til boldene A til Ø. En væsentlig reduktion i antallet af søgetræets forgreninger kan opnåes ved i stedet at benytte en regressiv søgealgoritme; der søger baglæns ud fra målet mod start og forsøger at bestemme handlinger, der har ført til måltilstanden. [12]

Plan søgninger, i både progressive og regressive algoritmer, foregår enten ved tilstandsrum (Eng: state-space) eller ved plan-rum (Eng: plan-space). I Planlægningssystemer, der benytter tilstandsrum, varetages søgetræet ved at hver node i træet repræsenterer en komplet verdenstilstand. En kant i træet repræsenterer en handling, der manipulerer med betingelser over i en ny tilstand. Søgninger i tilstandsrum er en form for totalt rangerede plansøgninger, i og med at der kun undersøges en lineær rækkefølge af handlinger der fører fra start til mål (eller omvendt). Det har den betydning at hver af handlingerne, skal tage højde for samtlige betingelser imellem tilstande, der endnu ikke er håndteret. Søgningen mellem handlingerne skal foregå kronologisk.



Søgninger i planrum foregår ud fra del-og-hersk princippet. Hver node i træet er en plan for en delløsning på et tidligere problem i grafen. Denne plan kan være dekomponeret til mindre planer, der tilsammen udgør en løsning på den øvre plan. Hver kant i grafen repræsenterer derved et delproblem af fader-planens problem og den sammenhængende plan er løsningen på det. Det har den fordel at hvert delproblem kan løses individuelt og dele der har størst betydning for den

overordnede plan kan løses først. Eksempelvis hvis man planlægger turen for et containerskib, vil det være vigtigere at beregne tiden og ruten mellem havnene, før en plan for læs og losningen af containerne ordnes. Søgninger i planrum benytter sig af princippet om sidst mulig forpligtigelse (Eng. Least commitment principle), hvor ethvert valg udskydes, indtil det er absolut nødvendigt at træffe. Derved kan man spare at lægge planer for ting, der muligvis skal ændres alligevel.



En plans opbygning beskrives af fire komponenter.

- Et sæt af handlinger, der skal udføres
- En arrangering (Eng: Ordering) af handlingerne i planen, således at det kan bestemmes hvis en handling er afhængig af at andre handlinger er udført før eller efter. En arrangering af to handlinger betyder ikke nødvendigvis at de følger lige efter hinanden. Arrangeringen behøver ikke være totalt ordnet. Eksempelvis er det ikke nødvendigt at afgøre om man i en plan for at tage tøj på, tager venstre eller højre sok på først. Det er dog afgørende at venstre sok tages på før venstre sko.
- Et sæt af kausal links; Der forbinder to handlinger vha. de betingelser der opnåes imellem dem. Disse betingelser skal nødvendigvis være beskyttet imellem eksekvering af de to handlinger.
- En liste af åbne betingelser der mangler at blive opfyldt før en plan er komplet.

En konsistent komplet plan defineres som en plan hvor der ikke er nogen cyklusser i arrangeringen (ingen ring i handlinger der afhænger af andre handlinger). Hvor hver handlings forhånds-betingelser er opfyldt af en tidligere handlings effekt-betingelser; og hvor enhver handling der er i konflikt med et kausal link, er arrangeret således at det følger enten inden eksekvering af begge handlinger i linket, eller efter. Planen skal endvidere være fuldt instantieret, uden modstrid imellem bindingerne. En konsistent komplet plan der ved udførsel af handlingerne opnår en ønsket tilstand kaldes en løsning på et problem.

### 2.3.1 Beskrivelses sprog

For at lave en effektiv planlægger, kræves et godt modelleringssprog, som på en formel og logisk måde giver en beskrivelse af et problem. Ideen ved at bruge et sådant beskrivelses sprog er at have et sprog, som er fleksibelt nok til at kunne beskrive en lang række af situationer, men stadig restriktiv nok til at det er muligt at lave effektive algoritmer, der kan løse problemet.

For at være i stand til at lave en plan for et problem, kræves et udgangspunkt som typiske er den nuværende tilstand, samt hvilken opgave der ønskes opnået for at planen udfører hvad man ønsker. Disse to tilstande kaldes starttilstand og måltilstand, som hver især beskrives ved et antal betingelser. En startbetingelse  $S_{start}$  kan bestå af betingelserne ”Er indenfor“ eller ”Har sko på“. Hvis ønsket er at lægge en plan for hvordan man kommer udenfor, vil en oplagt betingelse være ”Er udenfor“ som dermed er indholdet af måltilstanden  $S_{finish}$ . Hver betingelse er dermed med til at beskrive en del af en tilstand.

Problemet er nu at transformere tilstanden  $S_{start}$  over til  $S_{finish}$ . Til dette kræves nogle handlinger som på hver deres måde ændrer en tilstand over til en anden. En handling f.eks. ”Gå udenfor“ eller ”Se TV“, indeholder hver nogle forhåndsbetingelser samt nogle effektbetingelser. For at en handling kan udføres kræves at den nuværende tilstand opfylder nogle forudsætninger, som er givet i handlingens forhåndsbetingelser. Det er f.eks. ikke muligt at udføre handlingen ”Se TV“, hvis forhåndsbetingelsen ”Har TV“ ikke eksisterer i din nuværende tilstand. Derimod hvis handlingen ”Gå udenfor“ har forhåndsbetingelsen ”Er indenfor“, vil denne handling være tilladt at blive udført. En handlingens effektbetingelser, som er de betingelser der bidrager til tilstandsændringen, tilføjes til nuværende tilstand og derved er en ny tilstand opstået.

Målet er nu at finde den rigtige rækkefølge af handlinger, for at starttilstanden  $S_{start}$  bliver transformeret til at indeholde  $S_{finish}$ .

Dvs. at et beskrivessprog består af

$P$ : et sæt af mulige betingelser

$O$ : et sæt af mulige handlinger

$I$ : en beskrivelse af start tilstanden

$G$ : en beskrivelse af den ønskede mål tilstand

Hvor  $I \subseteq P \wedge G \subseteq P$  er opfyldt

Antallet af mulige tilstande er  $2^P$ . [14]

Betingelser har mulighed for at have variabler tilknyttet som f.eks. "Er hos( $x$ )", hvor  $x$  er et navn på en person. Hvis  $x$  substitueres med f.eks. Bent, betegner betingelsen "Er hos(Bent)" at personen er på besøg hos Bent.

### 2.3.1.1 STRIPS

STRIPS var det første beskrivelses sprog der blev udviklet af Richard Fikes og Nils Nilsson tilbage i 1971 og har dannet grundlag for de fleste efterfølgende beskrivelses sprog [14].

STRIPS understøtter positive sammensatte betingelser og betragter alle ikke kendte betingelser som falske. Dette gør at verden antages fuld kendt og derved en lukket verden.

**Eksempel** For at give et eksempel på et simpelt STRIPS problem betragtes følgende scenario.

En professor har som udgangspunkt bare fødder. Han ønsker at have sine sko på. Han har mulighed for at gøre 4 forskellige ting

- Tage venstre sko på (*LeftShoe*)
- Tage højre sko på (*RightShoe*)
- Tage venstre sok på (*LeftSock*)
- Tage højre sok på (*RightSock*)

Han skal have en sok på den fod, han ønsker at tage sko på.

Vi har følgende betingelser

- Venstre bar fod (*LeftBareFoot*)
- Højre bar fod (*RightBareFoot*)
- Venstre sok på (*LeftSockOn*)
- Højre sok på (*RightSockOn*)
- Venstre sko på (*LeftShoeOn*)

- Højre sko på (*RightShoeOn*)

Udgangstilstanden  $S_{start}$  er  $LeftBareFoot \wedge RightBareFoot$  og den ønskede sluttetilstand  $S_{finish}$  er  $LeftShoeOn \wedge RightShoeOn$ .

Her ses følgende illustreret i STRIPS

Listing 2.1: STRIPS eksempel

```

Init (RightBareFoot  $\wedge$  LeftBareFoot)
Goal (RightShoeOn  $\wedge$  LeftShoeOn)
Action (RightShoe ,
  Precondition: RightSockOn ,
  Postcondition: RightShoeOn)
Action (RightSock ,
  Precondition: RightBareFoot ,
  Postcondition: RightSockOn)
Action (LeftShoe ,
  Precondition: LeftSockOn ,
  Postcondition: LeftShoeOn)
Action (LeftSock ,
  Precondition: LeftBareFoot ,
  Postcondition: LeftSockOn)

```

### 2.3.1.2 ADL

ADL er en udvidelse af STRIPS der har en række forbedringer, så det er muligt at udtrykke mere avancerede problemstillinger, som kan være for komplekst i STRIPS. ADL understøtter i forhold til STRIPS negative eller negerede betingelser såsom  $\neg LeftSockOn \wedge \neg LeftShoeOn$ , som vil beskrive det at personen hverken har en venstre sok eller sko, hvor vi førhen blev nød til at have en betingelse som betegner at det var en bar fod. Som en følge af at der understøttes negative betingelser, vil ikke opgivne betingelser som ikke optræder i tilstanden blive betragtet som ukendt. Dette gør ADL til en åben verden betragtning. Ydermere har ADL den fordel at den understøtter adskilte betingelser, lighedsudtryk og variable typer.[11]

### 2.3.2 Delvist rangeret plan

Ideen med planlægning er at man er i stand til at lægge en plan, som transformere en given verden fra en tilstand til en anden. Denne transformation består

af et antal handlinger, som hver især påvirker tilstanden når den udføres. Disse handlinger kan være afhængige af hinanden, så udover at finde de handlinger der skal bruges for at transformere tilstanden fra udgangspunktet til målet, kræves også at de udføres i den rigtige rækkefølge.

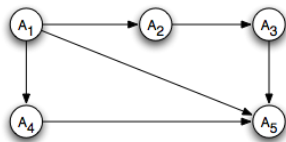
Hvis der under planlægningen fastholdes en total rangeret liste af handlinger til en delvist løst plan, kaldes dette for en total rangeret eller lineær planlægning. Hvis planlægningen derimod kun repræsenterer nødvendige orienterede arrangeringer (constraints) imellem handlinger, kaldes dette delvist rangeret (partial order) eller ikke lineær planlægning.

En sådan arrangering består af et par af handlinger, hvor  $A_1$  kommer før  $A_2$ , men ikke nødvendigvis lige før. Dvs. beskrevet i LTL (lineær temporal logik)  $A_1 \diamond A_2$ . Følgende notation vil blive brugt

$$A_1 \prec A_2$$

Formålet med at bruge disse arrangeringer er at følge sidst mulig forpligtelses (least commitments) princippet, som gør at der kun bliver lavet en arrangering imellem 2 handlinger, hvis denne arrangering er tvunget. Fordelen ved at bruge dette princip er fleksibilitet samt prøve at undgå at lave handlinger som senere måske skal laves om, og derved spare unødigt handlinger[13].

En delvist rangeret plan kan repræsenteres som en graf som beskriver arrangeringerne imellem handlingerne. Hver node repræsenterer en given handling og kanterne imellem noderne repræsenterer hver arrangering. For eksempel hvis vi har følgende arrangeringer  $A_1 \prec A_2$ ,  $A_1 \prec A_4$ ,  $A_1 \prec A_5$ ,  $A_2 \prec A_3$ ,  $A_3 \prec A_5$  og  $A_4 \prec A_5$  fås følgende graf



Denne delvist rangerede graf repræsenterer 4 total rangerede grafer, dette giver følgende lineær udstrækninger (linear extensions)

$$A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_4 \rightarrow A_5$$

$$A_1 \rightarrow A_4 \rightarrow A_2 \rightarrow A_3 \rightarrow A_5$$

$$A_1 \rightarrow A_2 \rightarrow A_4 \rightarrow A_3 \rightarrow A_5$$

$$A_1 \rightarrow A_2 \rightarrow A_3 \rightarrow A_4 \rightarrow A_5$$

Til at finde en af disse lineære udstrækninger kan bruges en topologisk sortering på den partial order graf.

For at løse et planlægnings problem, skal man kende den initiale tilstand og tilstanden man ønsker opnået, begge beskrevet ved betingelser.

Som udgangspunkt bliver der lavet to pseudo handlinger *Start* og *Finish*.

*Start* har ingen forhåndsbetingelser og den initiale tilstand som effektbetingelser.

*Finish* har betingelserne der ønskes opnået som forhåndsbetingelser og ingen effektbetingelser.

Der tilføjes følgende arrangering

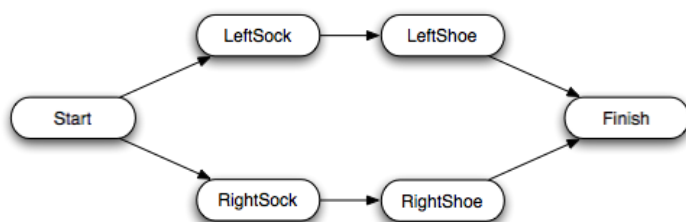
$$Start \prec Finish$$

### 2.3.2.1 Eksempel

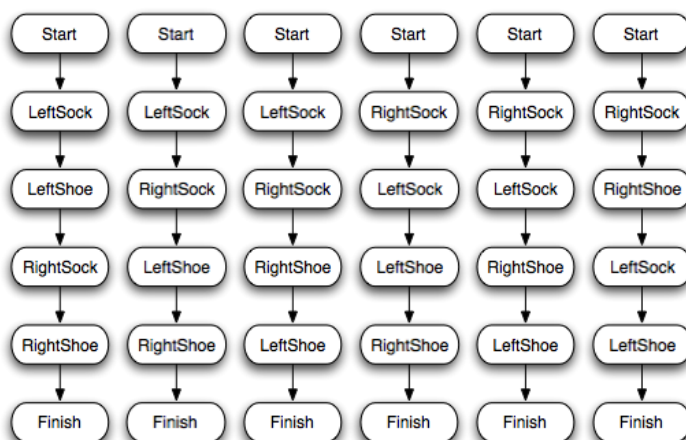
Der betragtes nu en løsning til STRIPS eksemplet i afsnit 2.3.1.1, hvor en person ønsker at tage sine sko på. Målet  $A_{finish}$  udgøres konjunktivt af de to betingelser:  $LeftShoeOn \wedge RightShoeOn$ .

Der vælges en af de to forhåndsbetingelser som findes i  $A_{finish}$ , f.eks.  $LeftShoeOn$ . Handlingen som kan opfylde den er  $LeftShoe$  og vi tilføjer arrangeringen  $LeftShoe \prec Finish$ . Efter denne handling er udført har vi nu en ny tilstand som skal løses, nemlig  $LeftSockOn \wedge RightShoeOn$ . Hvis der igen vælges delmålet  $LeftSockOn$ , ses det at handlingen  $LeftSock$  opfylder denne. På samme måde tilføjes arrangeringen  $LeftSock \prec LeftShoe$  og vi får tilstanden  $LeftBareFoot \wedge RightShoeOn$ . Nu bruges vores pseudo handling *Start* som har startbetingelser som effektbetingelser til at lave arrangeringen  $Start \prec LeftSockOn$  og  $LeftBareFoot$  fjernes fra tilstanden, da dette mål nu er opfyldt. På samme måde gøres med  $RightShoeOn$  som vælges fra  $A_{finish}$  som kan løses af  $RightShoe$  osv.

Hvis der tegnes en graf af alle arrangeringerne fås følgende



Da grafen ikke er total rangeret ses at der findes 6 lineære udstrækninger



Alle disse er hver især en løsning til problemet.

### 2.3.3 Planlægnings algoritme

Algoritmens opgave er iterativt at forfine en plans detaljer, indtil en plan er en løsning på et problem.

Udover at indeholde en rangeret liste af arrangeringer  $A_1 \prec A_2$ , har planen samtidigt en liste af åbne betingelser (open preconditions) som mangler at blive løst, samt en liste af kausal links. (Se ovenfor)

## Åbne betingelser

Denne liste udgør en række delmål, beskrevet ved betingelser, som mangler at blive opfyldt for at planen kan kaldes komplet (fuld løst). Som udgangspunkt ligger alle forhåndsbetingelser som findes i  $A_{finish}$  i denne liste. Algoritmen forsøger at omdanne denne liste så den matcher effektbetingelserne i  $A_{start}$ .

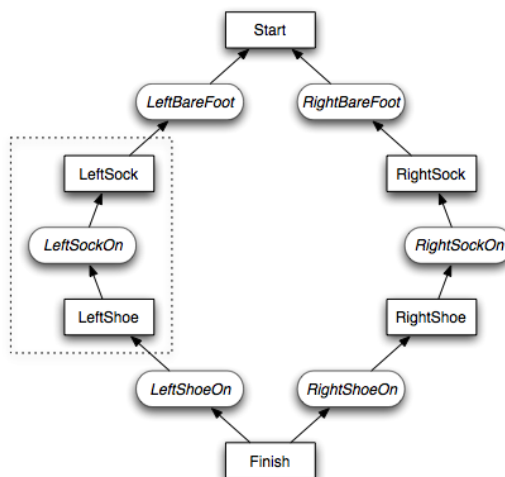
## Kausal link (causal link)

Et kausal link imellem to handlinger  $A_1$  og  $A_2$  hvor betingelsen  $p$  er bindeledet betegnes

$$A_1 \xrightarrow{p} A_2$$

Dette kan læses som “ $A_1$  opfylder  $p$  for  $A_2$ “. Dvs. at  $p$  er en effektbetingelse af  $A_1$  og en forhåndbetingelse af  $A_2$ . Et sådan link skal altid være opfyldt i tiden mellem handlingen  $A_1$  og handlingen  $A_2$ . Dvs. at der ikke må tilføjes en handling  $A_3$  som skaber en modstrid med  $p$ , hvis  $A_3$  kan komme efter  $A_1$  og før  $A_2$ . I det tilfælde siges at  $A_3$  udgør en trussel mod linket  $A_1 \xrightarrow{p} A_2$ .

En liste af kausal links danner en graf, hvor handlinger (firkantede i figur) bliver bundet sammen af deres pågældende betingelser (runde i figur). En sådan graf kan for eksemplet i afsnit 2.3.1.1 og 2.3.2.1, illustreres som vist nedenfor.



Her ses det at f.eks. det kausale link  $LeftShoe \xrightarrow{LeftSockOn} LeftSock$ , sørger for at fastholde at betingelsen  $LeftSockOn$  er opfyldt i perioden imellem  $LeftShoe$  og  $LeftSock$ .

### Trussel håndtering

Hvis en handling  $A_3$  truer et kausal link  $A_1 \xrightarrow{p} A_2$ , skal denne trussel håndteres på en måde, som sørger for at  $A_3$  ikke kan komme imellem  $A_1$  og  $A_2$ . Dette opnås ved oprykning eller nedrykning:

**Oprykning (promotion)** Tvinger den truende handling  $A_3$  til at komme efter det kausal link  $A_1 \xrightarrow{p} A_2$ , ved at tilføje arrangementen  $A_2 \prec A_3$ .

**Nedrykning (demotion)** Tvinger den truende handling  $A_3$  til at komme før det kausal link  $A_1 \xrightarrow{p} A_2$ , ved at tilføje arrangementen  $A_3 \prec A_1$ .

### Variabler

Som nævnt udgøres en betingelse enten af en boolsk værdi eller vha. en eller flere variabler, der ikke nødvendigvis er bundet til en konstant. Der kan være mange tilfælde hvori benyttelsen af ubundne variabler er effektivt; Eksempelvis handlingen Pickup med forhåndsbetingelserne  $Empty-Arm$  og  $Ball(A, x)$ , hvor  $x$  er en variabel for boldens position. Før at Pickup handlingen kan benyttes må de to betingelser være opfyldt. Med ubunden  $x$  betyder denne instans af Pickup at den skal samle bold  $A$  op fra en endnu ukendt destination, samt have en tom arm.

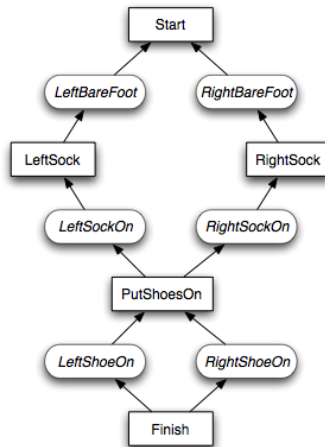
Ved at benytte en ubunden variabel  $x$ , kan man binde  $x$  til et koordinat hvor eksempelvis start handlingen dikterer at bold  $A$  befinder sig. Dette er en del af sidst muligt princippet at overlade beslutningen om hvad  $x$  skal bindes til, indtil skridt i planen, med mere information, kan træffe beslutningerne. Det er let at se at det vil være mere effektivt end at lede igennem alle lokationer for bold  $A$ .

### Genbrug af handlinger

Når en betingelse  $B_{goal}$  udgør et delmål der endnu ikke er håndteret (en åben betingelse), behøves som nævnt en handling der har en effektbetingelse  $B_{goal} \subseteq A_{eff}$ . Handlingen med effektbetingelserne  $A_{eff}$ , vil typisk være en helt ny

instantieret operator. For at kunne løse ikke-trivielle problemer skal tidligere benyttede handlingers effektbetingelser  $Tidlig_{eff}$  kunne genbruges; Betydende at betingelser kan opfylde delmål, hvis  $B_{goal} \subseteq Tidlig_{eff}$ .

Som eksempel sammensætter vi handlingerne  $RightShoe$  og  $LeftShoe$  til en handling  $PutShoesOn$ , som har forhånds-betingelserne  $LeftSockOn \wedge RightSockOn$  og effektbetingelserne  $LeftShoeOn \wedge RightShoeOn$ . Hvis delmålet som ønskes opnået er  $B_{goal} : LeftShoeOn$ , vil den bruge  $PutShoesOn$ , da  $B_{goal} \subseteq PutShoesOn_{eff}$ . Når delmålet  $B_{goal} : RightShoeOn$  skal løses, genbruges den tidligere handling  $PutShoesOn$ , da  $B_{goal} \subseteq PutShoesOn_{eff}$  også gælder. I stedet for at oprette en ny instans af  $PutShoesOn$ .



## 2.4 Analyse

### 2.4.1 Planlægning med variabler

Vi definerer betingelsen  $BoksPå(a, b)$  som en beskrivelse af at boks  $a$  står på boks  $b$ . For at udvikle en fuldt instantieret plan, udfra handlinger med ubundne variabler, vil det kræve en metode til at binde ubundne variabler til respektive konstanter; Således at hvis en handling dikterer at den ønsker opfyldt betingelsen  $BoksPå(A, x)$ , hvor  $A$  er en boks og  $x$  er en variabel, der skal bindes til en korrekt boks. For at binde variabel  $x$  til en korrekt boks benyttes en *foreningsfunktion*  $UF(B_{fuld}, B_{delvis})$ . Hvor  $B_{fuld}$  og  $B_{delvis}$  er henholdsvis en fuldt

instantieret betingelse og en delvist instantieret betingelse. Funktionen afgør om variablerne i  $B_{delvis}$  er forenelige med de instantierede konstanter i  $B_{fuld}$  således at  $B_{delvis} \subseteq B_{fuld}$ . Er dette tilfælde kan de ubundne variabler i  $B_{delvis}$  bindes til respektive konstanter i  $B_{fuld}$ . Det vil eksempelvis betyde at den delvist instantierede betingelse  $BoksPå(A, x)$  kan forenes med  $BoksPå(A, B)$  og binde variabelen  $x$  til boks  $B$ . Ligeledes vil  $BoksPå(A, x)$  ikke kunne forenes med  $BoksPå(C, D)$  og variabel  $x$  vil ikke blive bundet.

I forbindelse med trusler vil en delvist instantieret betingelse ikke udgøre en trussel mod en fuld instantieret. En *mulig* trussel er ikke en real trussel før bundne variabler i betingelserne identificerede den som en sådan [11].

En plan er ikke komplet og konsistent blot ved at alle åbne betingelser er blevet håndteret. Planen skal også være fuldt instantieret, med alle variabler bundne. Det må nødvendigvis være et krav, da alle reelle trusler skal være kendt. Da trusler ikke regnes for reelle hvis de består af ubundne variabler, kan konsekvensen af at acceptere delvist instantierede planer være at man eksempelvis har  $BoksPå(A, x)$  i sin endelige plan. Vælger eksekveringsalgoritmen at  $x$  betyder boks  $C$  vil du stå med en uforståelig plan, hvis boks  $A$  ikke står på boks  $C$ . En sikring mod endelige delvist instantierede planer, kan ifølge [10] opnåes ved at start betingelsen ikke består af variabler og at hver variabel i en handlings effektbetingelser også er inkluderet i dens forhåndsbetingelser.

## 2.4.2 Heuristik

POP er et PSPACE-komplet problem i dens worst-case betragtning (se afsnit 2.4.3). I praksis kan den gennemsnitlige køretid nedbringes ved at udregne et kvalificeret estimat på hvilken prioritering handlingerne skal undersøges. Heuristikken bruges herefter til at vælge den plan der ønskes udviklet for evt. at finde en komplet plan.

Som følge af at der bruges en delvist rangeret plan, er det meget svært at estimere hvor langt en plan er fra at opnå dens mål, da planen ingen direkte tilstand har[11].

En simple måde at lave en heuristisk funktion  $h()$  er at bruge antallet af åbne betingelser, der ikke direkte kan opfyldes af nogle af start betingelserne, som  $h()$ . Ved valg af en betingelse der skal løses, fra listen over åbne betingelser, kan det med fordel vælges den som de færreste handlinger kan opfylde. Dette vil detektere hvis der er en betingelse som ingen handlinger kan opfylde og derfor umuligt at løse for at finde en komplet plan. Ydermere hvis der kun findes én handling til at opnå en betingelse, kan den med fordel vælges først, da denne

handling alligevel er uundgåelig. [11]

I [9] argumenteres for forskellige metoder der kan forbedre heuristikken eller mindske søgetræet. Adskilte arrangeringer kan bruges, når en handling  $A_3$  truer et kausal link  $A_1 \xrightarrow{p} A_2$ . I stedet for at først forsøge at lave oprykning og derefter nedrykning af hvor den truende handling skal ligge, kan en adskilt arrangering  $A_2 \prec A_3 \vee A_3 \prec A_1$  tilføjes. På den måde udelukker det ene ikke det andet og giver derfor mere fleksibilitet i den delvist rangerede plan og derved bedre mulighed for at finde en lineær udstrækning.

### 2.4.3 Ydelse

Ydeevnen for planlæggeren afhænger i alvorlig høj grad af problemstillingen og hvor god heuristikken er til at vurdere hvilken plan der skal arbejdes videre på for at finde en løsning til problemet [8]. Dette skyldes bl.a. at algoritmen har en worst-case køretid som er PSPACE-komplet [3].

En planlægers effektivitet bestemmes af 2 faktorer

- tidsforbrug pr. undersøgt plan.
- størrelsen af (del-)søgetræet som bliver undersøgt.

POP har en fordel i forhold til total rangeret planlægger (TOP), da størrelsen af søgerummet altid vil være mindre end eller lig med størrelsen til den total rangeret planlægger  $size(POP) \leq size(TOP)$ . Det eneste tidspunkt søgerummet for POP er lig med søgerummet for TOP  $size(POP) = size(TOP)$  er når den delvist rangerede plan er en total rangeret plan (ved kun at have en mulig lineær udstrækning). Når dette ikke er tilfældet vil størrelsen af søgetræet for POP være stærkt mindre end størrelsen af søgetræet for TOP  $size(POP) \ll size(TOP)$  og muligt eksponentielt mindre  $exp(size(POP)) = size(TOP)$ . [8]

Kort summeret op fra [8], har POP et lidt højere tidsforbrug pr. undersøgt plan på  $\Theta(e)$  imod TOP på  $\Theta(n)$ , hvor  $e$  er antallet af kanter i den partiel rangeret graf (med worst-case  $O(n^2)$  og best-case  $\Omega(n)$ ) og  $n$  er antallet af specifikke handlinger i planen (længden af planen).

Selvom søgerummet for POP er mindre end for TOP, betyder dette dog ikke nødvendigvis at den er mere effektiv. Dette afhænger af søgestrategien samt heuristikken som planlæggeren følger. [8]

### 2.4.4 Andre algoritmer

Der findes andre algoritmer indenfor automatiseret planlægning, her tænkes på CSP (Constraint Satisfaction Problem) baserede algoritmer såsom Graphplan og andre tilstands rum (state-space) algoritmer. Disse algoritmer betragtes generelt som hurtigere end POP algoritmer, da det er muligt at lave bedre heuristik, da deres fulde tilstand er kendt undervejs. Dog har POP algoritmer en stor fordel i deres måde at håndtere eksekveringen af planerne, idet den benytter sidst mulig forpligtigelse (least commitments) princippet. Det gør den genererede komplette plan meget fleksibel, da det ikke er givet på forhånd præcis hvilken rækkefølge nogle af handlingerne kan blive eksekveret i. Dette ses ved at de fleste eksisterende planlægnings systemer som involvere eksekvering, informations indsamling og koordinering er baseret på POP algoritmer [9].

## 2.5 Design

Algoritmen der bruges til at generere den partiel rangeret plan, er inspireret af pseudo algoritmen fra artiklen "Intelligent Control of Autonomous Underwater Vehicles - A Partial Order Planner for the Orca Project"[10]. I det følgende beskrives vores design af POP\*.

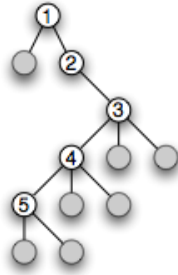
### 2.5.1 POP\*

POP algoritmen i [10] som der er brugt som udgangspunkt er rekursiv. Dette bevirker at den laver en form for dybde først søgning igennem plan-rummet. Algoritmen backtracker (går tilbage til en tidligere tilstand) ikke før den enten ikke har flere mulige handlinger den kan udføre, eller at der eksisterer nogle trusler som den ikke kan løse. Det betyder at hvis heuristikken vælger en forkert handling  $A_1$  fremfor en anden handling  $A_2$  i en given delplan  $P$ , som ville have været mere rentabel, vil  $P'$  som er delplanen efter  $A_1$  er udført, have en dårligere heuristik end  $P$  med handlingen  $A_2$ . Da  $P'$  endnu ikke er ekspanderet opdages en bedre plan ikke før der eventuelt bliver backtracket, hvis heuristikken kommer til at vælge en mindre optimal handling.

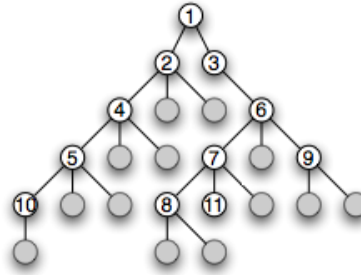
Med inspiration af  $A^*$  søgealgoritmen [4], hvor der benyttes en prioriteret kø, som altid har den bedste estimerede plan liggende først i køen, vil det være muligt altid at undersøge den plan som regnes for at have den bedste mulige løsning. Ved at benytte  $A^*$  princippet er det helt op til heuristikken at vurdere hvilken plan algoritmen skal udvide i næste iteration også selvom den næste

plan er et andet sted i træet. Ulempen ved at bruge denne type i forhold til en rekursiv, er at hukommelsesforbruget øges, da der kan være en masse halvt udforskede grene i træet og derved et større antal af blade.

Rekursiv med heuristik  
(dybde-først søgning)



Iterativ med heuristik  
(bedste-først søgning)



Den prioriterede kø i POP\*, har altid den plan med den mindste estimerede heuristiske værdi  $h()$  forrest i køen.

### 2.5.1.1 Pseudo algoritme

POP\* algoritmen får som argument en liste af startbetingelser, en liste af målbetingelserne samt en liste af handlinger som er til rådighed.

Der laves en tom plan, som kun indeholder pseudo handlingerne *Start* og *Finish* med en arrangement  $\{start \prec goal\}$ , hvor efter den bliver tilføjet til den prioriterede kø.

Hvis der findes en plan i køen som mangler at blive undersøgt, vælges den første på listen og dermed den med den bedste heuristik. I hver iteration bliver der valgt et delmål, som er betingelsen  $c$ , der tilhører handlingen  $S_{need}$ . For hver handling  $a \in \omega$  som kan opfylde betingelsen  $c$ , laves der en kopi  $P'$  af den nuværende plan  $P$  og  $a$  tilføjes til  $P'$ . Hvis der ingen cykluser er i  $O$  efter  $a$  er tilføjet, tilføjes det kausal link  $a \xrightarrow{c} S_{need}$ . Der undersøges om der opstår nogle trusler. Hvis dette er tilfældet bliver *ResolveThreats* udført. Hvis det lykkes at løse konflikten lægges den nye forfinede plan ind i køen. Hvis der dog eksisterer en trussel som ikke kan løses, bliver hele planen droppet, ved simpelthen ikke at lægge nogen plan i køen.

Lad  $\omega$  være en liste af handlingsklasser

Lad *goal* og *start* være handlinger med forhåndsbetingelserne *pre* og effektbetingelserne *eff*

Lad *P* være en plan.

Lad *Q* være en prioriteret liste af planer *P*, med metoderne *enqueue(P)* og *dequeue(P)*.

Lad *I* (*InitState*) være en liste af start betingelser.

Lad *G* (*GoalState*) være en liste af mål betingelser.

Lad *O* være en liste af bindinger  $S_1 \prec S_2$ .

Lad *L* være en liste af kausal links  $S_1 \xrightarrow{P} S_2$ .

```

1: procedure POPSTAR(I, G ,  $\omega$ )
2:   eff(goal) := G;                                ▷ goals effektbetingelser initialiseres
3:   pre(start) := I;                                ▷ starts forhåndsbetingelser initialiseres
4:   var plan := empty-plan;
5:   add(O(plan), {start  $\prec$  goal});
6:   enqueue(Q, plan);

7:   while Q is not empty do
8:     P := dequeue(Q);
9:     if P is a solution then
10:      return P;
11:    end if
12:    Sneed, c := Select-Subgoal(P);
13:    saddlist := Choose-All-Operators(P, Sneed, c);

14:    for a in saddlist do
15:      var P' := P;
16:      add(O(P'), {a  $\prec$  Sneed});
17:      if no cycles in O then
18:        add(L, {a  $\xrightarrow{c}$  Sneed});
19:        var threats := GetThreats(P', Sneed, c);
20:        var P'' := ResolveThreats(P', Sneed, threats);
21:        if P''  $\neq$  nil then
22:          enqueue(Q, P'');
23:        end if
24:      end if
25:    end for
26:  end while
27: end procedure

```

### POPstar

Den mest simple måde at vælge et delmål er at tage den første betingelse i

listen over åbne betingelser som mangler at blive opfyldt. Der kan dog med fordel udvides med metoden beskrevet i afsnit 2.4.2, for at forbedre chancen for at give en bedre plan.

```

1: procedure SELECT-SUBGOAL(Plan p)
2:    $S_{need}, c := p.OpenPreconditions[0];$ 
    $\triangleright$  Vælg handling  $S_{need}$  som afhænger af at betingelsen  $c$  bliver opfyldt
3:   return  $S_{need}, c;$ 
4: end procedure

```

For at finde handlinger som kan opfylde betingelsen  $c$  kigges der først om der kan genbruges nogle tidligere handlinger, derefter ses om den kan opfyldes af pseudo handlingen  $Start$  og til sidst kigges der på nye operatører.

Lad  $\omega$  være en liste af handlingsklasser.

Lad  $P$  være en plan,

Lad Steps være en liste med alle tidligere tilføjede handlinger i planen.

Lad bindings være en samling af bindinger i planen

```

1: procedure CHOOSE-ALL-OPERATORS( $P, S_{need}, c$ )
   /* For alle tidligere handlinger i  $P$ : */
2:   for each step  $S_{add}$  from Steps( $P$ ) that has ' $c_{add}$ ' as an effect do
3:     if ( $c \cap c_{add} \cap bindings(P)$ ) then
4:       add( $S_1, S_{add}$ );  $\triangleright$  tilføj  $S_{add}$  til liste  $S_1$ 
5:     end if
6:   end for

   /* For start-handlingen i  $P$ : */
7:   for each condition  $c_{start}$  in eff( $start$ ) do
8:     if ( $c \cap c_{start} \cap bindings(P)$ ) then
9:       add( $S_2, start$ );  $\triangleright$  tilføj  $start$  til liste  $S_2$ 
10:    end if
11:  end for

   /* For alle handlingsklasser i  $P$ : */
12:  for each step  $S_{add}$  from  $\omega(P)$  that has ' $c_{add}$ ' as an effect do
13:    if ( $c \cap c_{add} \cap bindings(P)$ ) then
14:      add( $S_3, S_{add}$ );  $\triangleright$  tilføj  $S_{add}$  til liste  $S_3$ 
15:    end if
16:  end for
17:   $saddlist := S_1 \cup S_2 \cup S_3;$ 
18:  return  $saddlist;$ 
19: end procedure

```

Når en trussel opdages skal den håndteres efter fremgangsmåden beskrevet i afsnit 2.3.3. Kort fortalt fortsætter algoritmen rekursivt med at tilføje ar-

rangeringer. Den forsøger først at flytte handlingen frem i planen, hvis dette ikke lykkes forsøger den at flytte den tilbage i planen.

```

1: procedure RESOLVE-THREATS( $P, T, Llist$ )
2:   if length( $Llist$ )=0 then
3:     return  $P$ ;
4:   end if

```

Promotion

```

5:   var  $P' := P$ ;                                     ▷ lav en kopi af planen
6:   add( $O(P), \{T \prec Llist[1]\}$ );                   ▷ tilføj  $T \prec Llist[1]$  til rangeringen
7:   if no cycles in  $O$  then
8:      $P := \text{ResolveThreats}(P, T, Llist[2..n])$ ;
9:     if  $P \neq nil$  then
10:      return  $P$ ;
11:    end if
12:  end if

```

Demotion

```

13:  var  $P := P'$ ;                                     ▷ tilbage til forrige plan
14:  add( $O(P), \{Llist[1] \prec T\}$ );                   ▷ tilføj  $Llist[1] \prec T$  til rangeringen
15:  if no cycles in  $O$  then
16:     $P := \text{ResolveThreats}(P, T, Llist[2..n])$ ;
17:    if  $P \neq nil$  then
18:      return  $P$ ;
19:    end if
20:  end if
21:  return  $nil$ ;
22: end procedure

```

### 2.5.1.2 Heuristik

Som en følge af at den iterative POP\* algoritme bruger en prioriteret kø, til at vælge den næste plan der skal undersøges, vil det være muligt ud fra heuristikken at bestemme om POP\* algoritmen skal fungere som en dybde-først søgning eller en bredde-først søgning. For at skabe en dybde-først søgning (som den rekursive algoritmen, der blev brugt som udgangspunkt), sættes  $h'() = h() - 1$ , hvor  $h()$  er heuristikken for den den plan der undersøges og  $h'()$  er den nye heuristik der tilføres dens børn. I dette tilfælde vil planens børn altid ligge før alt andet og derved vil børn altid blive undersøgt før deres forældre. For derimod at skabe en bredde-først søgning, sættes  $h'() = h() + 1$ . Her vil børnene først blive evalueret efter alle forældrene i samme dybde af træet.

## 2.6 Implementering

Algoritmen samt beskrivelsessproget er implementeret i Microsoft's programmeringssprog C# .NET.

Beskrivelsessprogets handlinger og betingelser bliver implementeret ved at nedarve fra henholdsvis *Action* og *Condition* (se evt. afsnit 3.5.1). Hvis en betingelse kan indeholde variabler, skal denne dog nedarve *ConditionVariable* som er en udvidelse af *Condition*.

Hver handling (se bilag D.3), har en metode *ActionAchieves()* til at forespørge på om den kan opfylde et givent delmål. Endvidere kan handlingen transformere verdenen ved at optage dens forhåndsbetingelser og erstatte dem med dens effektbetingelser (dette foregår via metoden *NewOpenPreconditions()*). Forhånds- og effektbetingelser er givet ved typer af de forskellige betingelser, dvs. ikke instansierede betingelser. Når en ny operator benyttes, instancieres disse til rigtige betingelser.

Selve algoritmen kan findes i bilag D.2 og datastrukturen som er selve planen findes i bilag D.1.

Standardbiblioteket fra .NET benyttes, samt en implementering af en prioriteret kø, da en sådan ikke findes i standardbiblioteket. Den implementation der bruges er lavet af BenDi fundet på [www.codeproject.com](http://www.codeproject.com) [2].

### 2.6.1 Heuristik

Den implementerede heuristikfunktion  $h()$  er forholdsvis simpel. Hver handling er vægtet som et estimat af udførselstiden af den pågældende handling. Der tages summen af alle fundne handlinger fra start tilstanden til den nuværende tilstand  $g()$ . Dertil tages antallet af åbne betingelser (open preconditions)  $k()$ , så der fås

$$h() = g() + k()$$

Dette gør algoritmen til en tilnærmelsesvis bredde-først søgning, da længden af delplanen har indflydelse på heuristikken. Denne fremgangsmåde har vist sig til dette LEGO eksempel at give en acceptabel heuristik.

Hvis der udelukkende bruges en heuristik  $h() = k()$ , som altid tager den plan med mindst åbne betingelser, skal der tages højde for at der nemt kan komme cykler. Dette kan forekomme hvis 2 handlinger som er hinandens inverse, er de

handlinger som giver den mindste liste af åbne betingelser. Der vil den prøve at lave en uendelig lang plan.

## 2.7 Test

For at teste om systemet opfører sig stabilt og korrekt, benyttes strukturel test i form af unit test[15], via programmet NUnit. Unit test bruges til at teste om individuelle dele af et program fungerer efter hensigten. Målet med en unit test er at isolere essentielle afsnit af programmet og sikre de er rigtige. Delene skal så vidt muligt være uafhængige og for at validere dem, kan man specificere en række parametre de skal opfylde. Testen forløber 'bottom-up' således at for at testet større sektioner, skal de enkelte dele først testes uafhængigt.

Vi har benyttet unit test på et lille repræsentativt udvalg af sektioner i vores program. I.e. oprettelse, binding, lig med, forespørgelse på om delvist eller fuldt instantieret etc. på en betingelse. Samme fremgangsmåde med en handlings type.

De enkelte algoritmer, funktioner, metoder er, hvis de ikke er inkluderet i NUnitTest(for de flestes vedkommende), funktionelt testet. Typisk ved deres kreation. For en fuldgældig test, skulle disse funktioner naturligvis være med i den strukturelle test.

For det overordnede system har vi lavet en test strategi, hvor en serie af planlægnings test kan valideres; ligeledes i NUnitTest. Dette har vist sig uvurderligt da selv små ændringer i en handling, i måden ting bliver genbrugt på, heuristikken etc. har stor indflydelse på plan træet. Konsekvenser der kan være meget svære at forudse og overskue. Når en mindre defekt er fundet og løst, vil det muligvis kunne ødelægge planlægningen for et andet problem. Af den grund er det rart konstant at kunne få en hurtig validering på de problemer vi har udvalgt.

Selve planlægningsprocessen er fyldt med skjult heuristik (rækkefølgen af åbne betingelser, valg af delmål, binding af variabler etc.) som ikke umiddelbart håndteres i selve heuristik funktionen. Det gør det svært at styre og også i den sammenhæng har test strategien været nyttig. Igen har små ændringer som synes ubetydelige, været skyld i at tiden for at køre testen er steget fra 4-5 sekunder til 2-3 minutter(!).

Teststrategien med udvalgte problemer kan findes i appendix D.13. Strategien har hovedsageligt været udført i LEGO verdenen(se kapitel 3).

## 2.8 Udvidelser

For at POP kan bruges i mere virkelighedstro systemer vil det være nødvendigt med nogle modifikationer og udvidelser. STRIPS er begrænset på fire områder[11]:

**Hierarkiske planer:** Det er givet at skal POP planlægge hvorledes et hus skal opføres, vil den blive overvældet af detaljer omkring handlinger for at banke søm i, købe mørtel, få byggetilladelser osv. For at simplificere dens opgave, er det logisk at STRIPS kan udvides til først at beskrive generelle problemer siden gå i detaljer. Altså først ligger en plan med handlingerne:

få tilladelser  $\rightarrow$  købe jorden  $\rightarrow$  bygge hus.

Dernæst kan den kigge på hver enkel handling og gå ned i hierarkiet og kigge på eksempelvis at bygge huset. Denne dekomponering kan fortsætte indtil en plan alene består af atomare handlinger, som at ligge en mursten.

To ting skal modificeres for at benytte hierarkiske planer. STRIPS skal tillade mere komplekse handlinger(ikke atomare), som består af andre komplekse handlinger eller af atomare handlinger. POP skal kunne erstatte komplekse handlinger, med de underhandling den udgøres af. Ingen af delene er alvorlige ændringer af vores hidtidige fundament.

**Tid:** I de fleste miljøer vil tiden for udførelsen af en handling spille en vigtig rolle. Alene deadlines og heuristik mæssigt vil inkludering af tid være væsentlig for en effektiv planlægning. Da STRIPS er baseret på logiske konstruktioner, antages det at alle handlinger sker øjeblikkeligt. Man kan indbygge et estimat af tids forholdet imellem handlinger, men dette ville ikke give et godt billede af planens samlede eksekverings tid. Ligger professoren en plan for at komme på arbejde (I.e. stå op, gøre sig klar, tage bussen etc.) vil det være nødvendigt at vide hvor lang tid de enkelte handlinger tager, for at han kan nå bussen.

**Ressourcer:** Moderne planlæggere benyttes ofte i fabriks lignende miljøer til at fastlægge arbejdsfordelingen. STRIPS er ikke gearet til at beskrive hvor mange arbejdere, maskiner, råstoffer osv. der er til rådighed. Handlinger skal kunne forbruge og generere ressourcer og afgive betingelser på antallet af enheder den skal bruge.

**Komplekse betingelser:** På trods af STRIPS evne til at benytte variabler, er der stadig mange situationer den ikke kan beskrive. Et mere informativt sprog (se afsnit 2.3.1) vil være en fornuftig udvidelse.

Implementeringen af ovenstående udvidelser er uden for rapportens ramme og der henvises til andet litteratur om emnet. Særligt [11] beskriver hvorledes udvidelserne kan inkorporeres i POP platformen.

## 2.9 Diskussion

Vi har som nævnt i test afsnittet lavet test både for enkelte funktioner og hele test scenarier. I begge tilfælde ville en mere vidtrækkende test strategi være lækkert, men skønnes for omfattende. Opstår der et problem undervejs i planlægningen, er det at finde hvor ulykken opstår ganske vanskeligt. Outputtet fra plan-grafen er gigantisk og selv ved mindre problemer dannes træer med 150 noder der er filtret sammen på kryds og tværs. Findes der ikke en løsning, må man afbryde træet når det eksempelvis har undersøgt 50.000 noder - hvor gik det galt?. Debugge hele systemet er derved meget tidskrævende. Eventuelle problemer der opstår er vi, belært af erfaring, ret sikre på skyldes måden hvorpå handlinger sletter og opretter åbne betingelser. Specificeres en handling ikke ordentligt (og det kan være svært) vil der ikke returneres en brugbar plan. Selve opætningen omkring algoritmen har efter indledende test ikke givet nævneværdige problemer.

**Anvendelighed af POP med STRIPS** Vi har gennem implementeringen af POP fået en god føling med algoritmen og STRIPS. For at systemets virkeligt kan siges at være alsidigt, vil nogle af de nævnte udvidelser skulle indbygges. STRIPS giver en god og intuitiv tilgang til planlægning. Idéen med dens anvendelse til snart sagt alle problemer, hvori du kan beskrive verden med betingelser og handlinger der manipulerer med disse er genial. De hovedsagelige problemer der gør at STRIPS (og dens overbygninger) ikke ses i alt mekanik, er to-foldig. Naturligvis er algoritmens meget dårlige worstcasekøretid grundstammen i dens begrænsede anvendelsesmuligheder. Som nævnt forværres beregningstiden alvorligt med antallet af handlinger og betingelser. Der skal ikke alt for mange komponenter til før køretiden er for slem til at benyttes i realtime systemer. Dette problem eksistere selvsagt i al planlægning.

Det andet problem er lidt mere bekymrende for STRIPS o.lig. sprog. Det er desværre ikke let 'blot' at tilføje en række handlinger og betingelser, så klarer algoritmen det selv. Designet og sammenhængen mellem disse kan være meget svært. Kunne gennem talrige test kan man se om de rigtige forbindelser i plantræet bliver oprettet, så at ens ønskede problemstillinger kan løses. Som nævnt er disse test meget tidskrævende. Alene vores simple eksempel med LEGO, hvor antallet af handlinger er forholdsvis begrænsede, har vi måtte bruge anseeligt mere tid på debug end rent design.

### 2.9.1 Implementerings vanskeligheder

At give et generelt indblik i nogle af de problemer der kan opstå i et planrums søgetræ, kan være vanskeligt da de typisk er meget situations afhængige. Enkelte nævneværdige problemer vil dog blive listet i nedenstående. De tjener mere til at vise rent praktiske vanskeligheder end bidrage til det teoretiske grundlag; af samme grund kan afsnittet springes over eller let læses.

**Vedligeholdelse af åbne betingelser** Når en ny operator opfylder en åben betingelse, skal den sørge for at vedligeholde og opdatere listen over åbne betingelser. Dette gøres dels ved at slette den netop opnåede betingelse fra listen, samt ved at oprette de nye åbne betingelser operatoren kræver opfyldt. Endvidere kan det ses at det er fornuftigt, at operatoren sletter de betingelser fra listen som den yderligere opfylder.

De fleste handlinger tager et sæt af betingelser med variabler, manipulerer med disse og smider dem tilbage ind på den åbne liste. Tager handlingen flere betingelser, med forskellige variabler, er det ikke trivielt at få afgjort hvilke betingelser der konjunktivt udgør handlingens effekt.

Eksempelvis ses handlingen '*SeperateItems*' der deler et objekt i to. Handlingen har forhåndsbetingelsen  $item(x, y)$  og effektbetingelserne  $item(x)$  &  $item(y)$ . Hvis handlingen oprettes på baggrund af  $item(A)$ , må den finde  $item(A)$  i åbne betingelser, slette den og oprette en ny åben betingelse  $item(A, y)$ . Kan den endvidere opfylde en anden åben betingelse  $item(B)$ , vil  $item(A, B)$  kunne instantieres med det samme i åbne betingelser. Det er dog ikke ligetil at gennemgå åbne betingelser for at finde tidligere ting der kan opfyldes. Hvis der eksisterer flere objekter  $item(C)$ ,  $item(D)$  etc. er det ikke let at afgøre hvilken en handling der skal benyttes sammen med  $item(A)$ .

Det har vist sig ganske svært at lave en generel funktion, der uafhængigt af den kontekst en handling oprettes på baggrund af, kan vedligeholde listen over åbne betingelser. Ideelt set skal der kunne slettes alle åbne betingelser en vilkårlig handling, med et uspecificeret antal betingelser og variabler, opfylder; kunne oprettes nye betingelser med en korrekt sammenhæng mellem variablerne i forhåndsbetingelser og effektbetingelserne; kunne tilføjes handlingens effektbetingelser til liste over tidligere handlinger osv. Selvom de korrekte betingelser bliver tilføjet listen over åbne betingelser er rækkefølgen langt fra ligegyldig. Alene heuristikmæssigt kan en ombytning af rækkefølgen af betingelser der tilføjes til listen, have anseelig betydning. Fald og stigninger fra 100 til 2000 noder der undersøges i træet, har været observeret. En specifik håndtering af hver enkelt handling er derfor ofte nødvendigt og tidskrævende (konsekvenserne på plantræet skal undersøges til bunds), og gør at det ikke er en simpel sag at tilføje

nye handlinger og betingelser.

**Information til rådighed** En node i en planrums graf (i modsætning til i tilstandsrum) har som nævnt ovenfor den egenskab at den kun skal koncentrere sig om sin lille del af en overordnet plan. Den har ikke oplysninger om konsekvenser andre dele af træet kan have på verdenstilstanden. Der er i POP\* trusselshåndtering, men det består ikke i at den enkelte handling, overvejer hvad hele træets kumulerede effekt er og træffer valg ud fra dette. Det har den fordel at rangeringen ikke er rigid men fleksibel. Ulempen er dog at en node ikke kan få oplysninger nogen steder fra om tilstanden af verden.

Eksempelvis ses  $Move(x, y)$  handling der flytter fra et koordinat  $x$  til et andet  $y$ . Det kan ikke ses ud fra den initiale tilstand om der er et objekt i vejen mellem de to koordinater - dette objekt kan være flyttet eller lagt i vejen af en anden gren i træet. Ligeledes kan det heller ikke afgøres af slutbetingelserne, da de blot beskriver egenskaber der ønskes verden skal tilføjes efter eksekveringen af planen (ikke hele verdens tilstanden). Endvidere er det ikke muligt at søge igennem det midlertidige træ og beregne effekten af disse handlinger, da det jo naturligt ikke er afgjort om de handlinger overhovedet er med i den endelige løsning.

$Move(x, y)$  må derfor lave betingelser der søger for at der ikke er objekter på vejen, og det komplicerer systemet en del. Det ses at da verdens tilstanden ikke kan oplyses under planlægningsfasen, er en effektiv heuristik vanskelig. I en verden hvor man rykker rundt på et kort og samler ting op, kunne man lave et globalt mål koordinat som alle  $Move(x, y)$  handlinger vil søge hen mod for at mindske heuristikken. Start koordinatet for robotten kunne naturligt vælges (start da robotten er regressiv). Starter robotten samme sted som den slutter, men skal hente et objekt på et endnu ikke specificeret sted, vil heuristikken betyde at algoritmen undersøger muligheder tæt rundt om start, i stedet for at gå direkte efter objektet.

**Gentagelse af tilstand** Teoretisk kan der være store forbedringer at opnå i plansøgningen, hvis tilstande der allerede er ekspanderet, ikke behøves undersøgt igen. Hvis en gren af søgningen har nået en tilstand (ligegyldigt hvordan) og en anden del af grenen når den samme tilstand på en alternativ måde, vil den videre søgning være ens for de to grene og de må formodes lige langt fra en løsning. Der er i designet forsøgt adskillelige forsøg på at definere en tilstand så den er restriktiv nok til at ens tilstande bliver anerkendt, uden at udelukke tilstande der kan føre til en løsning. En delpans instans af definitionen er undervejs vha. en strengs hashkode gemt i et hashmap. Når en plan udvælges fra den prioriterede kø, checkes først om en plan med ens tilstand er undersøgt først. Er den det, behøves planen ikke at blive ekspanderet.

Uden at gå i detaljer med selve forsøgende på definitionerne af en tilstand,

er konklusionen at det ikke har vist sig rentabelt. Tiden det tager at gemme tilstande og slå op i hashmappet er alt for kostbart i forhold til hvor mange noder der ellers kunne ekspanderes. Heller ikke at forglemme implementeringsvanskelighederne ved at definere en sådan tilstand. Heuristikken må sørge for at søgningen ikke går i ring og konstant prøver  $MoveNorth \rightarrow MoveSouth \rightarrow MoveNorth \rightarrow \dots$ .

## 2.10 Konklusion

Vi har i ovenstående beskrevet og implementeret en delvis rangeret planlægger, der understøtter STRIPS beskrivelsessprog. Systemet udgør en grundlæggende platform til planlægning og til senere udvidelser.

Vi kan lave en simplificeret verdensbeskrivelse vha. STRIPS betingelser og manipulere med betingelserne vha. handlingstyper. Vi har afprøvet algoritmen på et hurtigt PutShoesOn eksempel og klargjort den til at implementere en planlægger i LEGO verdenen. Endvidere har vi tilføjet en heuristik funktion til POP der, afhængigt af miljøet, hurtigt kan kalibrere forholdet mellem handlinger. Heuristiken har vist at give store forbedringer til køretiden.

Undervejs har vi konstateret at STRIPS kan være meget tungt at arbejde med, på trods af dets intuitivt simple design. Fremtidigt arbejde vil afgjort kræve en revision af beskrivelses sproget.



## KAPITEL 3

# LEGO Mindstorm NXT Robot

---

### 3.1 Introduktion

Med planlæggeren på plads, beskrives nu LEGO verdenen og dens betingelser og handlinger, så præcist som muligt, for at planlæggeren kan lægge fornuftige planer til hvert problem den får givet. I dette kapitel ses der kun på planlægning med én robot på banen.

### 3.2 Kravspecifikation

Der ønskes et system, hvor en robot kan navigere rundt i en LEGO verden, hvor der befinder sig genstande. Robotten får en opgave den skal løse, men den må helt på egen hånd finde ud af hvordan den løser denne opgave med de midler den har til rådighed. Hvis der ligger en genstand i vejen for robotten eller hvis den vurderer at det tager længere tid at bevæge sig uden, kan robotten vælge at samle genstanden op for at skabe en bedre vej til dens mål.

## 3.3 Analyse

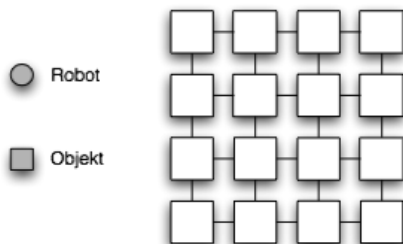
### 3.3.1 LEGO verdenen

For at være forberedt på nogle af de udfordringer som vi kan støde på senere, opstilles derfor nogle krav om hvordan Lego verdenen ser ud og hvordan den skal fortolkes.

#### 3.3.1.1 Banen

Banen er et 4x4 kvadratisk gitter (grid) hvor robotten kun kan køre imellem noder horisontalt og vertikalt i en lige linie ved at følge en streg på underlaget. Ved hver node kan robotten ved hjælp af en markering på underlaget registrere at den står ved en node og eventuelt skifte retning.

Her illustreres en tom bane, samt en robot og et objekt. Denne notation vil blive brugt igennem resten af rapporten.



#### 3.3.1.2 Robotten

Robotten er en modificeret Tribot [6] som beskrevet i [7]. Den har 3 primitive egenskaber den kan udføre

**Bevægelse** hen ad en linie ved hjælp af lyssensorer og en linie på underlaget.

**Løfte** en genstand så den kan transporteres.

**Smide** en genstand igen efter den er blevet samlet op.

Robotten er udstyret med en lyssensor, som kan registrere en genstand, i form af en kasse, som befinder sig på robottens vej. Robotten er derudover udstyret med en gribe arm, som kan løfte og smide en genstand. Vi vil i vores opstilling ikke bruge lyssensoren som kan opfange en genstand på vejen, da vores udgangspunkt er en kendt verden, dvs. hvor robotten kender til dens fulde start tilstand.

De 3 egenskaber kan omdannes til STRIPS handlingerne *Move*, *Pickup* og *Release*.

Verdenens fysiske konstruktion medfører nogle restriktioner som skal overholdes

1. En robot kan kun køre med en genstand af gangen.
2. En robot kan kun bevæge sig på felter hvor der ikke er genstande.
3. En robot kan kun bevæge sig mod en genstand, hvis den ikke allerede har en genstand.
4. Hvis robotten bevæger sig hen mod en genstand, skal den samle den op.
5. Når robotten smider en genstand, kan den kun køre baglæns.

Følgende betingelser bruges til at beskrive verdenen

$At(x, y)$  er robottens position  $x, y$ .

$BoxAt(x, y)$  en genstand på position  $x, y$ .

$NotBoxAt(x, y)$  betegner at der ingen genstand er på positionen  $x, y$ .

$HaveBall$  robotten har opsamlet en genstand.

$NotHaveBall$  robotten har ingen opsamlet genstand.

Vi ser her, at nogle betingelser har brug for to variable, som er  $x$  og  $y$  koordinatet på banen.

## 3.4 Design

*Pickup* handlingen, der sørger for at opsamle et objekt hvis der ligger et foran robotten, ser således ud i STRIPS notation.

```

Action(Pickup ,
  Precondition: At(x, y) ∧ BoxAt(x, y) ∧ NotHaveBall ,
  Postcondition: At(x, y) ∧ NotBoxAt(x, y) ∧ HaveBall
)

```

Her sørges der for at robotten er på samme position som et objekt, ved at de har ens variabler, samt at robotten ikke samtidig bærer på et objekt allerede. Robotten tilstand ændres til at den nu bærer på et objekt, samt at der bliver markeret at der intet objekt er på positionen mere.

Handlingen *Move* udbygges til 4 handlinger, som hver står for at bevæge robotten i hver sin retning, så det bliver til *MoveNorth*, *MoveSouth*, *MoveEast* og *MoveWest*. Dette gøres på baggrund af at STRIPS ikke tillader at have en handling som giver forskellige effektbetingelser (postconditions), afhængig af hvilken vej den gerne vil køre. Grunden til den har en *NotBoxAt* betingelse, er at den ikke kan flytte fra et punkt hvis den står foran en genstand.

```

Action(MoveNorth ,
  Precondition: At(x, y) ∧ NotBoxAt(x, y) ,
  Postcondition: At(x, y+1) ∧ NotBoxAt(x, y)
)
Action(MoveSouth ,
  Precondition: At(x, y) ∧ NotBoxAt(x, y) ,
  Postcondition: At(x, y-1) ∧ NotBoxAt(x, y)
)
Action(MoveEast ,
  Precondition: At(x, y) ∧ NotBoxAt(x, y) ,
  Postcondition: At(x+1, y) ∧ NotBoxAt(x, y)
)
Action(MoveWest ,
  Precondition: At(x, y) ∧ NotBoxAt(x, y) ,
  Postcondition: At(x-1, y) ∧ NotBoxAt(x, y)
)

```

Den samme opdeling gælder også for *Release*. Når en robot vil lægge en genstand kører den først hen til feltet hvor den vil lægge genstanden, placerer genstanden i midten af krydset og vender rundt og tilbage til udgangspunktet. For at handlingen kan udføres skal der hverken være nogen genstand på feltet hvor robotten står eller hvor den gerne vil lægge genstanden. Det er naturligvis også et krav at den allerede har en genstand.

```

Action(ReleaseNorth ,
  Precondition: At(x, y) ∧ NotBoxAt(x, y) ∧ NotBoxAt(x, y+1)
    ∧ HaveBall ,
  Postcondition: At(x, y) ∧ NotBoxAt(x, y) ∧ BoxAt(x, y+1) ∧
    NotHaveBall
)

```

```

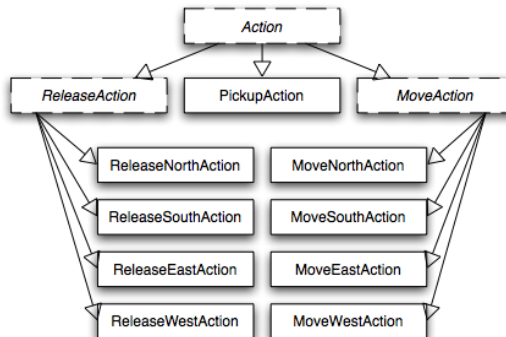
)
Action(ReleaseSouth ,
  Precondition: At(x, y) ∧ NotBoxAt(x, y) ∧ NotBoxAt(x, y-1)
    ∧ HaveBall ,
  Postcondition: At(x, y) ∧ NotBoxAt(x, y) ∧ BoxAt(x, y-1) ∧
    NotHaveBall
)
Action(ReleaseEast ,
  Precondition: At(x, y) ∧ NotBoxAt(x, y) ∧ NotBoxAt(x+1, y)
    ∧ HaveBall ,
  Postcondition: At(x, y) ∧ NotBoxAt(x, y) ∧ BoxAt(x+1, y) ∧
    NotHaveBall
)
Action(ReleaseWest ,
  Precondition: At(x, y) ∧ NotBoxAt(x, y) ∧ NotBoxAt(x-1, y)
    ∧ HaveBall ,
  Postcondition: At(x, y) ∧ NotBoxAt(x, y) ∧ BoxAt(x-1, y) ∧
    NotHaveBall
)

```

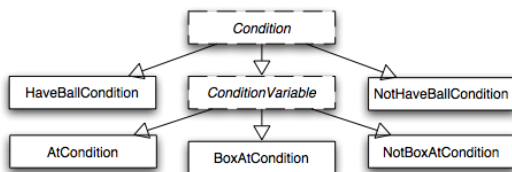
## 3.5 Implementering

### 3.5.1 Struktur

Alle handlinger fra afsnit 3.4 er implementeret som hver deres klasse alle nedarvet fra *Action*. *Move* handlingerne er nedarvet af en abstrakt *MoveAction* klasse som indeholder metoder som er tilfældes for alle *Move* handlinger. Det samme er tilfældet for *Release* handlingerne.



Alle betingelser er nedarvet fra *Condition*, hvor dog *At*, *BoxAt* og *NotBoxAt* er nedarvet af subklassen *ConditionVariable*, da disse indeholder  $x, y$  variabler.



### 3.5.2 Heuristik

Udover den generelle heuristik som beskrevet i afsnit 2.6.1, bliver der i det tilfælde hvor delmålet er at nå en given position udtrykt via en *AtCondition* eller en *BoxAtCondition*, udregnet Manhattan distancen fra robotens position til den ønskede position. Denne vægt indgår i udtrykket som  $j()$  på følgende måde

$$h() = g() + j() + k()$$

### 3.5.3 Hardware implementering

NXTMover er programmet der ligger nede på NXT'en, der sørger for at kommunikere med computeren om hvad den ser og hvad den skal foretage sig. Via computeren har MXTMover en række low-level handlinger den kan udføre, som svarer til de handlinger som er i STRIPS ovenover. Dog lever den tidligere implementation af release handlingen ikke op til hvad der ønskes, nemlig at genstanden bliver stillet på krydset igen. NXTMover er derfor blevet udvidet så den også understøtter en *releaseMoveback* handling, som fungerer på følgende måde:

1. Kør lige ud indtil den finder et kryds
2. Kør en smule baglæns så genstanden er ovenover krydset
3. Slip genstanden og kør lidt mere baglæns så kloen er uden for radius af genstanden
4. Drej 180 grader
5. Kør lige ud indtil den finder et kryds

### 3.5.4 Grafisk brugergrænseflade

Til implementationen er der udviklet en lille GUI som kan udregne forskellige scenarier inden for legoverdenen og eksekvere dem på en NXT Robot. På bilag B ses et skærmbillede af programmet. Ved start vælges der hvilke startbetingelser samt hvilke slutbetingelser som ønskes opnået. I listen nedenunder ses alle de handlinger som er understøttet af robotten. For at det er muligt at udregne en plan skal man vælge en At betingelse samt en HaveBall eller NotHaveBall betingelse.

Ved udregning af planen, vises grafisk det søgetræ som dynamisk bliver udvidet. Når en komplet plan er fundet, findes der en liniære udstrækninger af den delvist rangerede plan og vises under "Actions". Det er nu muligt at vælge en comport hvor robotten er tilknyttet, hvor ved robotten begynder at eksekvere planen ved tryk på "Execute".

Planer kan gemmes til og åbnes fra en fil ved brug af menulinien øverst.

## 3.6 Test

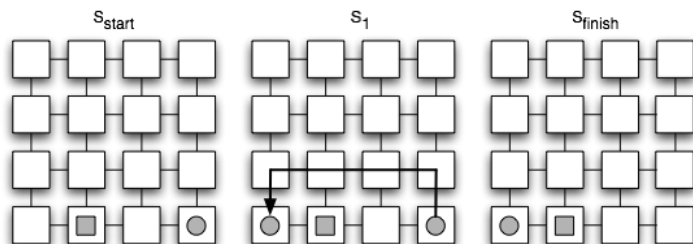
Funktionelle test benyttes til at teste LEGO implementationen, for at bekræfte at den fungerer efter hensigten. Et eksempel på en resulterende kausal graf er vedlagt i appendix C

Der er udarbejdet 6 test cases som hver illustrere forskellige situationer som kan forekomme i LEGO verdenen. Til hver testcase illustreres forskellige tilstande som viser forløbet, heriblandt start tilstanden  $S_{start}$ , sluttilstanden  $S_{finish}$  samt en eller flere mellem tilstande som viser robotens vej fra  $S_{start}$  til  $S_{finish}$ .

Derudover findes til hver testcase en film på den medlagte CD, som viser de forskellige testcases blive udført.

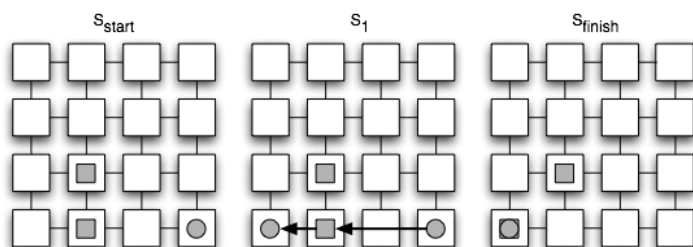
Endvidere er der lavet yderligere testscenarier, der kan ses i Appendix D.13.

### 3.6.1 Testcase 1



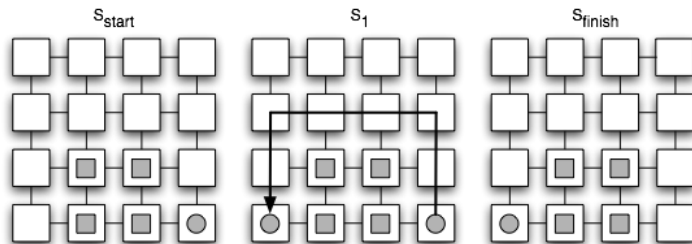
Dette senarie viser at robotten er i stand til at finde hen til et andet koordinat. Hvis der lægger noget i vejen for robotten kan den vælge at køre uden om eller flytte genstanden for at skabe fri passage. I dette tilfælde vælger robotten at køre uden om, da det kun er en forhindring.

### 3.6.2 Testcase 2



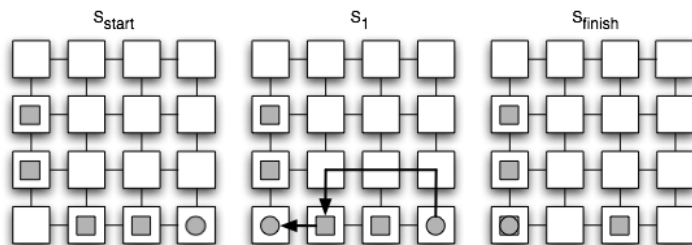
Nu er der kommet en ekstra forhindring ind på banen og robotten skal derfor, hvis den vælger at køre uden om, rimelig langt rundt om begge forhindringer. Den kan dog, som den gør i dette tilfælde, også vælge at køre hen og fjerne den ene kasse for derefter at køre i mål.

## 3.6.3 Testcase 3



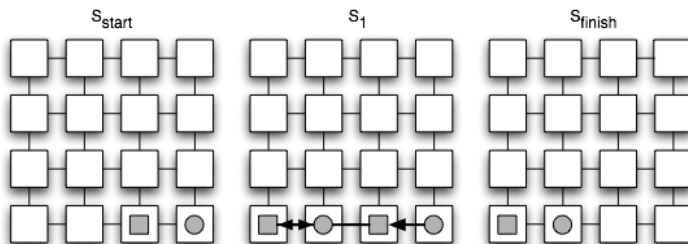
I denne testcase har robotten fået mange forhindringer så den ikke umidbart kan tage en hurtig vej direkte i mål. Den kan vælge at flytte forhindringerne, men pga. der ligger 2 i vejen, bliver den nød til først at løfte og flytte den første forhindring væk fra dens rute også flytte forhindring nr to. I dette tilfælde vælger robotten at køre uden om alle forhindringerne, da den vurderer at dette er den hurtigste vej.

## 3.6.4 Testcase 4



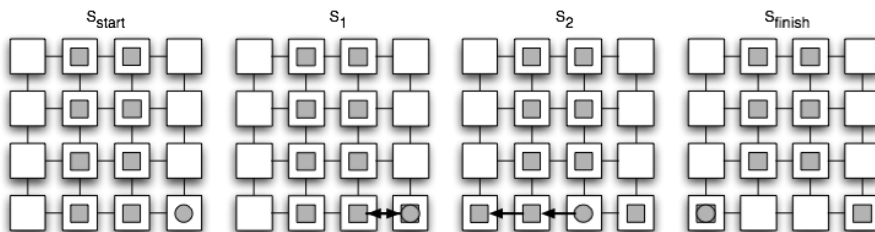
Her vælger robotten først at køre uden om den første forhindring, for derefter at flytte forhindring nr to så den kan nå sit mål.

### 3.6.5 Testcase 5



I dette tilfælde skal robotten flytte genstanden fra udgangspunktet  $S_{start}$  til  $S_{finish}$ . Den kører først hen til genstanden for at samle den op. Derefter kører den hen for at smide genstanden på destinationen og kører hen på den forrige position (da den ikke kan stå hvor den lige har smidt en genstand).

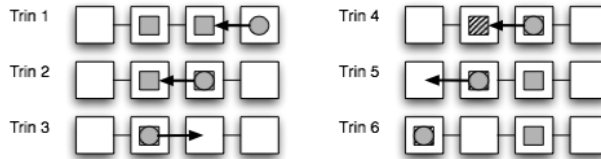
### 3.6.6 Testcase 6



I denne test skal robotten igennem 2 forhindringer for at komme til målet. Illustrationen viser hvordan denne testcase skulle have foregået. Først kører robotten hen til den første genstand og lægger den tilbage på den position hvor robotten kom fra, for ikke at spærre dens bane. Derefter kan den frit køre hen til genstand nr to og flytte den med i mål.

Dog følger robotten dog ikke denne fremgangsmåde. Robotten vælger efter den har samlet den første genstand op, pga. hvordan dens STRIPS handlinger er konstrueret, at køre hen til genstand nr to, for derefter at lave en *ReleaseEast* så den lægger den første genstand på samme position. Dog har den, da den kørte hen på genstand nr to's plads, skubbet den genstand væk fra banen og dermed kommer robotens indre verden ud af synkronisering med den virkelige verden.

Følgende illustration viser hvordan robotten valgte at løse problemet (der vises kun de 4 nederste felter, da de andre ikke ændres)



Fejlen sker i trin 2, hvor robotten vælger at fortsætte lige ud, i stedet for at vende rundt og lægge genstanden på robottens udgangspunkt. Ved at robotten kører hen på et felt med en genstand, hvor der allerede står en genstand, bliver den genstand der allerede stod på feltet skubbet ud af banen. Genstanden eksisterer stadig i robottens interne hukommelse, så den vil fortsætte som intet var hændt og opsamle den nu manglende genstand for at fortsætte i mål.

Hvorfor dette sker og hvordan dette løses beskrives i diskussionen afsnit 3.7.

### 3.7 Diskussion

På trods af at LEGO planerne genereres i et delvis rangeret plan miljø, er de endelige planer ikke fleksible. Der findes kun en mulig eksekverings rækkefølge. Dette skyldes ikke design i algoritmen, men er bestemt af forholdene i LEGO. Alle handlingerne er afhængige af koordinater og deres indbyrdes rækkefølge er derved rigid. Det kan umiddelbart ses, at der ikke på noget tidspunkt eksisterer en situation hvor det er lige meget hvilken rækkefølge to handlinger udføres i. I.e. hvis du står ved en bold skal du først samle op og så køre - ikke omvendt. Det er fristende at antage at rækkefølgen af eksempelvis *MoveNorth* og *MoveEast* kan være lige meget - de ender jo samme sted. Men har planen returneret *MoveNorth*  $\rightarrow$  *MoveEast*, ligger det i beregningen, at der ikke er et objekt på positionen nord for robotten. Planen har ikke regnet på om der forefindes et objekt øst for robotten, den kunne støde sammen med, hvis *MoveEast* vælges først. De to ruter er således ikke ens selvom de ender samme sted. Havde der ikke været forhindringer ville ruten være valgfri.

Som en følge af at der kun eksisterer en type af *BoxAt* betingelsen, bevirker det at robotten ikke kan se forskel på de forskellige genstande. Man kunne tilføje flere betingelser for hver genstand, men fra afsnit 2.4.3 vides at jo flere betingelser

der er i spil, vil antallet af mulige tilstande stige enormt. Der kunne derimod gives endnu en variabel til *BoxAt* som betegner hvilken box der er på feltet.

Testcase nr 7 gav ikke det forventede resultat, da det blev eksekveret i LEGO verdenen. Robotten valgte at lægge den første opsamlede genstand direkte ovenpå genstanden foran den, i stedet som forventet at køre baglæns og lægge den bagved på et felt som ikke allerede er optaget. Problemet opstår på grund af *MoveWest* handlingens (samt de andre *Move* handlingers) opbygning, da der ikke skælnes imellem om robotten har en genstand eller ej. En løsning til problemet kunne være at oprette endnu 4 *Move* handlinger, som bruges i de tilfælde hvor robotten har en genstand og de gamle handlinger bruges når den ikke har nogen genstand. Disse kunne opnås på følgende måde, her illustreret med *MoveEast*:

```

Action(MoveEastWithObject ,
  Precondition: At(x, y) ∧ NotBoxAt(x, y) ∧ HaveBall ∧
    NotBoxAt(x+1, y) ,
  Postcondition: At(x+1, y) ∧ NotBoxAt(x, y) ∧ HaveBall ∧
    NotBoxAt(x+1, y)
)
Action(MoveEastWithoutObject ,
  Precondition: At(x, y) ∧ NotBoxAt(x, y) ∧ NotHaveBall ,
  Postcondition: At(x+1, y) ∧ NotBoxAt(x, y) ∧ NotHaveBall
)

```

En anden og måske bedre løsning kunne være at benytte et ADL lignende sprog i stedet for STRIPS. ADL har den fordel (se afsnit 2.3.1.2) at den understøtter adskilte betingelser. En *MoveEast* handlingens forhånds-betingelser vil kunne udtrykkes på følgende måde:

$$\text{At}(x, y) \wedge \text{NotBoxAt}(x, y) \wedge (\text{NotHaveBall} \vee \text{NotBoxAt}(x+1, y))$$

Dette vil sikre at robotten kun kan bruge denne handling hvis den ikke allerede bærer på nogen bold eller hvis der ingen bold er på det næste felt.

## 3.8 Konklusion

I LEGO eksemplet her ses det at en af de fordele som delvist rangeret planlægning giver, nemlig at der skabes fleksibilitet i planerne forsvinder, da som beskrevet i afsnit 3.7 bliver fastholdt en total rangeret plan.

Det ses også at STRIPS har sine begrænsninger i dens måde at udtrykke betingelser og handlinger på som kan give nogle rimelig komplekse løsninger i nogle tilfælde.

Af den årsag vil en evt fremtidig udbygning være at implementere ADL som beskrivelses sprog.



# Multiagenter

---

## 4.1 Introduktion

Projektets fokusområde har primært været koncentrere omkring planlægning for en enkelt autonom agent. Hvor en agent er defineret som (jvf. [16]) et computersystem der befinder sig i et miljø, hvori systemet har kompetence til at udføre autonome handlinger, for at kunne opfylde dets designkriterier. Det vil i korte træk betyde at agenten egenrådigt skal være i stand til at træffe og udføre beslutninger - Uden menneskelig styring. Dette problem er for en simpel verden løst vha. POPstar implementeringen. I et mere komplekst og virkelighedsnært miljø eksisterer der dog nogle problemstillinger der er værd at kigge på.

I planlægningsalgoritmen POPstar antages en statisk kendt verden, hvori de eneste ændringer der kan forekomme er konsekvenser af en agents handlinger. Brydes disse forudsætninger op betyder det ikke at POPstar ikke vil fungere, men blot at der skal ligges en struktur rundt om systemet for at håndtere problemstillinger, som ændringer i verden der ikke er forudsaget af agenten; ændring af mål, deling af ressourcer med andre agenter etc. Vi vil i denne sektion og i programmet give en konceptløsning på hvordan et sådan system kan udformes baseret på LEGO verdenen.

## 4.2 Kravsspecifikation

Der ønsker at lave et grundlæggende rammesystem til styring af en agent i et dynamisk miljø, hvori andre agenter kan interagere og ændre miljøets tilstande.

I den kontekst skal den enkelte agent kunne opfylde følgende:

- Kunne agere og operere autonomt. Således at en agent selv kan planlægge og udføre løsninger på alsidige opgaver. Forudsættende naturligvis at en løsning eksisterer.
- Kunne modtage og behandle nye opgaver løbende.
- Varetage og vedligeholde en individuel repræsentation af verden, som kan opdateres løbende fra en, af agenterne, fælles repræsentation af verden eller fra agentens egne opfattelser.
- Reservere ressourcer (Det være sig typisk genstande og felter på kortet) således at der ikke vil opstå kollisioner og konflikter mellem agenterne.

Agenterne vil løbende under eksekvering, skulle opdatere deres individuelle repræsentationer af verden, samt informere andre agenter om ændringer foretaget. Et system til formidling af informationer om verden, samt om ressource tildelinger er deraf nødvendigt.

Da agenterne har mulighed for at reservere ressourcer, vil der opstå mulighed for deadlock situationer. Situationer hvori to eller flere agenter er låst fast i tomgang mens der ventes på ressourcer der aldrig frigives. Derfor er det et krav med et overordnet system, der kan identificere en deadlock situation og løse hårdknuden.

## 4.3 Teori

Generelt skal en agent (tilpasset efter [17]) udvise følgende karakteristika:

**Autonomi:** . Agenten skal kunne løse sine opgaver uden direkte hjælp fra menneskelige input eller med hjælp fra andre agenter. Agenten skal have egenkontrol over eget system og egne operationer.

**Reaktionsevne:** Agenten skal kunne opfatte og tage højde for ændringer i verden, hvorefter den skal reagere i henhold til situationen.

**Proaktivitet:** Agenten skal ikke alene udvise et reaktionært handlingsmønster, hvor kun øjeblikkelige situationer udgør agentens beslutningsgrundlag; men også planlægge og afgøre frem mod et givent mål

**Kommunikation:** Agenten skal kunne kommunikere med omverdenen og andre agenter for at sikre fair ressourcefordeling, samt evt. kunne dele opgaver imellem agenterne

Proaktivitetsdelen har den betydning at agenten, ikke kun kan beslutte sine handlinger reaktionært - altså hvor der handles ud fra observationer af den nære omverden på et givent tidspunkt. Den vil også skulle skabe og følge en plan, for derved at kunne løse problemer der ikke kan eller skal manuelt programmeres. Umiddelbart problem ved at sammensætte en plan af en korrekt sekvens af handlinger, der opfylder et vilkårligt antal betingelser er et PSPACE komplet problem. En nedskaleret plan for handlinger der må kunne forventes engang i fremtiden at bidrage til en løsning, kan derfor være en nødvendighed. En del af reaktionstiden for en agent nedsættes væsentligt ved planlægningsfasen og derfor er agenter oftest bygget op som både en reaktionær del, der eksempelvis holder robotten væk fra kantstenen, og en proaktiv del der i eksemplet søger for vejplanlægningen fra Vojens til Struer.

### 4.3.1 BDI styring af agenter

BDI er et system til rationel styring af en agent, baseret på Belief-Desire-Intention (DK: Opfattelse, Ønske, Hensigt) arkitekturen udviklet af Rao og Georgeff. BDI modellen er tildels udviklet ud fra en filosofisk undersøgelse af hvorledes den menneskelige hjerne i praksis resonere. En BDI agent vil konstant resonere over sin opfattelse af verden, opgaver og hensigter og, hvis det er påkrævet, ændre sin opførsel. BDI udgør en standard arkitektur til agenter der opererer i miljøer, hvor agentens opfattelse af verden er begrænset og miljøet skifter dynamisk (ændringer behøver ikke være introduceret af agentens egne handlinger). BDI-Arkitekturen baseres på fire vigtige komponenter:

**Opfattelsen** af en agent, er information om miljøet. Informationen er muligvis ikke korrekt og sandsynligvis begrænset. Opfattelsen af miljøet skal derfor af disse grunde konstant revideres.

**Ønsker** består af agentens tildelte opgaver og mål. Disse udgør agentens motivation for at frembringe bestemte betingelser i verden. Opgaverne er begrænset

til ikke at være hinandens negation.

**Hensigter** er de mål som agenten har lavet en plan for og er dedikeret til opnå. Typisk vil hensigter bestå af en plan under udførelse.

**Planer** er defineret som i ovenstående sektioner, som en sekvens af handlinger agenten har mulighed for at udføre for at løse et problem.

### 4.3.2 Deling af ressourcer i multiagentmiljø

I et multiagentmiljø, hvor flere agenter konkurrerer, om et sparsomt antal ressourcer (det vil være i LEGO-verdenen knudepunkter på kortet eller bokse), vil der være mulighed for deadlock situationer. En agent forespørger på en ressource og så længe ressourcen ikke tildeles venter den i tomgang. Hvis ressourcen aldrig bliver ledig for agenten, fordi den er holdt af en anden agent i tomgang, så eksisterer der en deadlocksituation.

Ressource tildeling benyttes for undgå konflikter ved benyttelse af objekter; Som kollisioner i et vejnet. Hvis en agent ønsker at benytte en ressource(eksempelvis køre over et knudepunkt) kaldes en forespørgsel på ressourcen. Hvis denne ikke kan tildeles øjeblikkeligt, fordi den er allokeret til anden side, må agenten vente. Når ressourcen endelig tildeles, kan agenten frit lave operationer på denne. Efter endt brug, skal agenten frigive ressourcen, således den kan benyttes af andre agenter.

$$\text{Forespørgsel} \Rightarrow \text{Benyttelse} \Rightarrow \text{Frigivelse}$$

Tildeling og frigivelse af ressourcer klares typisk vha. et overordnet system instantieret med en semafor for hver ressource. En deadlocksituation defineres formelt som et sæt af agenter, hvori hver agent venter på frigivelsen af en ressource, der holdes af en anden agent i sættet. Følgende fire betingelser skal være opfyldt samtidig for at en deadlocksituation kan opstå:

**Indbyrdes eksklusion** (Eng: Mutual exclusion): Der skal eksistere mindst en ressource som på ethvert tidspunkt kun kan benyttes af en agent. Hvis flere ønsker at benytte ressourcen, må de vente.

**Hold og vent** (Eng: Hold and wait): Der eksisterer en agent, som besidder en ressource og venter på at få allokeret ekstra ressourcer - der igen holdes af andre agenter.

**Ingen ufrivillig frigivelse** (Eng: No preemption): En ressource kan kun frigives frivilligt af den agent der besidder den givne ressource.

**Cirkulær ventetilstand** (Eng: Cirkular wait): Der eksisterer et cirkulært sæt af agenter ( $A_1, A_2 \dots A_n$ ) i tomgang, hvor  $A_1$  venter på  $A_2$ ,  $A_2$  på  $A_3$  etc. og  $A_n$  venter på  $A_1$ .

To metoder findes til at undgå deadlocksituationer:

**Forebyggelse af deadlocks** (Eng: Deadlock prevention) foregår ved at sikre at blot en af de fire ovenstående krav til en deadlock, ikke kan forekomme. Mulig sideeffekt ved en sådan procedure kan være at ressourcerne ikke udnyttes optimalt og hele multiagentsystemets produktivitet forringes.

**Undgåelse af deadlocks** (Eng: Deadlock avoidance) er en alternativ metode til at undgå deadlocks. Idéen er her at overvåge måden hvorpå ressourcer tildeles. Kort fortalt må systemet ved hver eneste forespørgsel overveje nuværende ledige ressourcer, ressourcer allokeret til agenter, fremtidige forespørgsler og frigivelser af enhver agent for at afgøre om en forespørgslen kan godkendes eller at agenten må vente for at undgå fremtidige mulige deadlocksituationer. Banker's algoritme, med en worst-case køretid på  $\Theta(mXn^2)$ , benyttes typisk til at foretage sådanne afgørelser (hvis der kun findes en instans af alle ressourcerne kan en simplificeret version benyttes, med køretid på  $\Theta(n^2)$  hvor  $n$  er antallet af agenter). For en fyldig gennemgang se [1] og [5]

Hvis hverken forebyggelse eller undgåelse kan eller er effektivt at benytte, vil et system der overvåger om agenterne er gået i deadlock være nødvendigt. Herefter benyttes en funktion som bryder dødvandet. Dette kan foregå ved at en operatør klarer det manuelt. En anden mulighed er at systemet selv har en automatisk procedure til at genskabe et sikkert stadie for sig selv. Proceduren vil enten 'afbryde' en agent ad gangen indtil den cirkulære ventetilstand brydes, eller alternativt at tvinge en agent til at opgive nogle af dens rekvirerede ressourcer.

## 4.4 Analyse

Problemstillingen i forbindelse med implementering af et system af multiagenter kan præciseres som følgende: Der ønsker at lave en konceptløsning på implementering af et system der kan håndtere et endeligt antal agenter. Der vil i konceptet tage udgangspunkt i LEGO miljøet, beskrevet tidligere i rapporten. Agenterne vil udgøres af adskillelige, ens styrede og opererede LEGO NXT robotter. Disse agenter vil benytte POPstar implementeringen i planlægningsfasen,

samt BDI arkitekturen til agentstyring og ræsonnering. Ud over agenterne vil der på banen være et antal bokse, robotterne kan samle op og flytte.

Opgaverne som agenterne løbende skal løse, skal udformes vha. STRIPS betingelser og begrænses til ikke at må være negerede. De handlingsmuligheder agenterne vil have til rådighed er fire varianter af flyt et felt på kortet, fire varianter af smid boks et felt væk og ryk tilbage, samt saml bold op.

Kendskab til verden kan agenterne enten få fra et centralt register, der indsamler og uddelegerer oplysninger om verden, imens robotterne udfører handlinger. Disse oplysninger kan ligeledes være beskrevet vha. STRIPS. Det centrale register skal være sikret med indbyrdes eksklusion og mod race-conditions (DK: løbs betingelser). En alternativ løsning er at lade agenterne kommunikere indbyrdes og sammen synkronisere opfattelserne af verden. En kommunikationsprotokol for udveksling af oplysninger skal udvikles, men et central register der **altid** skal være tændt kan undgås.

Konflikter omkring ressourcer kan styres af et overordnet system, der vha. metaforer skal besidde metoder til forespørgelse og frigivelse af ressourcer. Systemet skal sørge for fair tildeling af ressourcer, samt forebygge, undgå eller løse deadlocksituationer. Agenterne kunne indbyrdes synkronisere deres brug af ressourcer, vha en protokol, men håndtering af deadlocksituationer vil givetvis være svært.

#### 4.4.1 BDI arkitektur

**Opfattelse:** Agentens opfattelse af verden vil være udgjort af de oplysninger om verden der modtages fra centralt hold. Det vil typisk bestå af STRIPS betingelser omkring hvor den selv befinder sig og hvor på banen der befinder sig kasser.

**Ønsker:** Er de brugerspecificerede opgaver der endnu ikke er løst. Det kan være opgaver som kun denne specifikke agent skal løse, eller opgaver der er fri for alle. Opgaverne vil være beskrevet som STRIPS betingelser, der skal gælde for at opgaven regnes som løst.

**Hensigter:** Består af den opgave som agenten er dedikeret til at løse. Nye hensigter begyndes der ikke på, før agenten har løst problemet eller at opgaven regnes som umulig.

## 4.4.2 Ressource-delning

Som pointeret i ovenstående er en håndtering af deadlocks nødvendig i et multi-agentmiljø. I denne sektion vil vi analysere på hvilke muligheder der kan benyttes i LEGO verdenen. Man kunne med en hvis ret hævde at undgåelse af deadlocks burde ligge i selve planlægningen. Vores plan-del, der som bekendt udgøres af en POP planlægger der understøtter STRIPS, er ikke gearret til fleksibiliteten omkring at undgå andre robotter der er i bevægelse, kasser der muligvis først skal frigives af andre agenter, tidsfaktorer etc. Da agenterne antages at have autonomt selvstyre og forventes at løse og udføre planer alene, vil det være vanskeligt at lægge langsigtede planer for hvad andre agenter tænker at påtage sig. De kan have skjulte motiver, hensigter og udgangspunkter etc; derfor vælges der en løsning hvori planlægningen foregår som i ovenstående kapitler. Ressource-delningen, og de dertil hørende deadlocksituationer skal derved håndteres på anden vis. I teorisektionen forefandttes der 3 muligheder:

### 4.4.2.1 Forebyggelse af deadlocks

Her kræves at enten '*indbyrdes eksklusion*', '*hold og vent*', '*ingen ufrivillig frigivelse*' eller '*cirkulær ventetilstand*' ikke holder. Herefter følger en analyse af hvorfor denne fremgangsmetode ikke kan benyttes i LEGO verdenen:

*Indbyrdes eksklusion* må nødvendigvis holde da det er en forudsætning for at ressourcer kun kan benyttes af en agent ad gangen.

*Hold og vent* må gælde da agenterne i nogle tilfælde skal optage en ny ressource før den kan slippe en tidligere. I.e. en agent holder på et knudepunkt på kortet og ønsker at køre videre til et nyt knudepunkt. Ressourcen for det første knudepunkt kan først frigives idet den har optaget(og dermed fået tilladelse til at benytte) ressourcen for den nye knudepunkt.

*Ingen ufrivillig frigivelse* vil betyde at en agent skal kunne tvinges til at opgive alle sine ressourcer. Dette er ikke hensigtsmæssigt da det vil betyde at agenten helt skulle kunne forlade brættet(kunne implementeres ved at lave særlige hold-epladser til hver agent på brættet, men ville medføre en stort spild i agenternes samlede output).

*Cirkulær ventetilstand* kan man sørge for ikke holder ved at påtvinge en total rangering af alle ressourcerne, og derefter kræve at agenten kun forespørger på ressourcer med en højere numerisk værdi, end dem den allerede har i besiddelse [1]. For at optage en ressource med lavere værdi, må agenten opgive de numerisk

højere ressourcer først. Det er givet ud fra LEGO miljøet, at en sådan rangering ikke kan påtvinges systemet. Det ville betyde at robotten ikke kunne vende tilbage til et tidligere besøgt knudepunkt, men eksempelvis kun bevæge sig Nord-Øst. Rangeringen mellem bokse og knudepunkter kan det også let ses at det vil give nogle modsætninger med hensigten af systemet. I.e har bokse en højere værdi end knudepunkter, vil du ikke kunne holde en bold og køre et andet sted hen. Har boksen en lavere værdi er det ikke muligt at samle en bold op, hvis du befinder dig på et knudepunkt(!).

#### 4.4.2.2 Undgåelse af deadlocks

Som i teori afsnittet vil vi overlade en fyldig beskrivelse af Banker's algoritme til [1]. Blot bemærker vi at algoritmen, for at træffe en beslutning om en forespørgelse kan resultere i en deadlock, benytter en *claim edge* (Dk: Anmeldelseskant). En *claim edge*  $Agent_i \rightarrow Ressource_j$  indikerer at  $Agent_i$  ønsker at benytte  $Ressource_j$  på et tidspunkt under eksekveringen af dens plan. Claim edge skal ikke forveksles med en forespørgelse, der direkte reserverer en ressource. En agent kan kun forespørge på en ressource hvis den allerede har indikeret en *claim edge*. Ifølge [1] kan en *claim edge*  $A_i \rightarrow R_j$  kun tilføjes til grafen i tilfælde af at agent  $A_i$  ikke holder nogle ressourcer. Dette er en umulighed i LEGO verdenen, da en agent altid vil holde mindst en ressource (I.e. altid være på et knudepunkt).

#### 4.4.2.3 Overvågning af deadlocks

Da forebyggelse og undgåelse af deadlocks har vist sig uhensigtsmæssige, må der benyttes en overvågningsalgoritme til at opdage deadlocks i systemet. Systemet vil kræve en to delt løsning; en del til at detektere deadlocken og en del til at udrede problemet. Til opdagelse af deadlocken må der skabes en monitor klasse, der holder styr på agenterne og deres ressourceforbrug.

## 4.5 Design

#### 4.5.0.4 BDI agent

Designet af BDI agenten vil baseres på Wooldridge's forslag til en agentløkke givet i nedstående figur. Ved kendskab til [18] samt [7]'s beskrivelse af pseudoko-

den, kan denne sektion let læses.

---

**Algorithm 1** BDI Agent-Løkke
 

---

```

1:  $B := B_0;$  ▷  $B_0$  er startopfattelsen af verden
2:  $I := I_0;$  ▷  $I_0$  er agentens starthensigter
3:  $D := D_0;$  ▷  $D_0$  er agentens startønsker
4: while (true) do
5:    $D := world.options(this);$ 
6:    $B := world.brf(this);$ 
7:    $I := world.filter(this);$ 
8:    $plan := MakePlan(B, I);$ 
9:   while (not ( $empty(plan)$  or  $impossible(plan)$ )) do
10:     $action := hd(plan);$ 
11:     $plan := tail(plan);$ 
12:     $B := world.brf(this);$ 
13:    if  $reconsider(I, B)$  then
14:       $D := world.options(this);$ 
15:       $I := world.filter(this);$ 
16:    end if
17:    if  $not-sound(plan)$  then
18:       $B := world.brf(this);$ 
19:       $plan := MakePlan(B, I);$ 
20:      continue;
21:    end if
22:     $execute(action);$ 
23:  end while
24: end while

```

**Figur:** Tilpasset pseudokode til BDI Agentløkke.

---

BDI strukturen udgøres kort fortalt i to while-løkker. En overordnet løkke der aldrig ender, og som instancieret et nyt mål agenten skal dedikeres til. Samt en underløkke der løber igennem hvert skridt i planen for det dedikerede mål, og konstant evaluerer om planen stadig holder eller skal revideres. Pseudokoden for agentstyringen udgøres af følgende funktioner:

**world.Options** der søger for at holde agents mål eller ønsker opdateret. Disse oplysninger fås fra et centralt register hvor brugerspecificerede mål opbevares. Det kan være opgaver som kun denne specifikke agent skal løse, eller opgaver der er fri for alle agenter.

**world.BRF** der er BDI's 'believe-revision function' (funktion til revision af agentens opfattelse). Dette er en funktion, der søger for at agentens opfattelse af

verden, konstant holdes opdateret. I denne implementering fås informationerne om verdens tilstand fra et centralt system, der holder styr på konsekvenserne af agenternes handlinger.

**world.Filter** er en funktion til at returnere den opgave, agenten skal dedikeres til. Denne opgave kunne selvsagt også hentes fra agentens ønsker, men ved at benytte world kan der sørges der for, at den pågældende opgave ikke deles ud til andre agenter.

**MakePlan** er hele systemets planlægningsfase, hvor der ud fra en verdensopfattelse og en hensigt, begge defineret ved betingelser, lægges en plan til opnå hensigten. Dette implementeres vha. POPstar der returnerer en skridtvis plan.

**Empty(plan)** og **Impossible(plan)** er givet ved navnene. Hvis planen ikke har flere skridt tilbage, forventes hensigten at være opnået. Impossible kan oversættes til at POPstar ikke har fundet nogle mulige planer og returnerer null.

**Not-sound(plan)** er en metode til at afgøre om en plans forudsætninger stadig holder - hvis ikke må den nødvendigvis rekalkuleres. I POPstar findes som bekendt en starthandling, der indeholder alle de forudsætninger der gælder før planen eksekveres. Undervejs i planlægningen binder forskellige handlinger sig med nogle af disse forudsætninger i kausallinks. Hvis betingelsen i et kausallink bundet til starthandlingen ændres før eksekveringen af posthandlingen, af andre aktører end agenten, må planen formodes at være ugyldig.

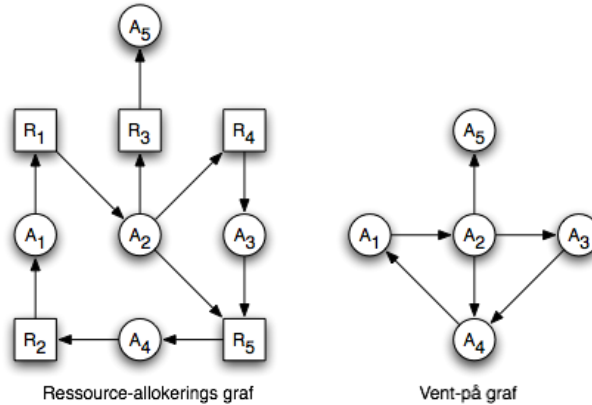
**Reconsider(plan)** benyttes til at afgøre om et bedre mål opstår, som det vil være mere fordelagtigt at forfølge. Funktionen er i programmet blevet nedprioriteret, men ville kunne implementeres ved at lægge end plan for samtlige af agentens ønsker og vælge den mest optimale plan - formentlig med færrest handlinger. Kalkulationer af planer for samtlige ønsker vil dog selvsagt være en meget kostbar manøvre og vil introducere unødvendige forsinkelser.

**Execute(action)** vil sørge for at agenter eksekverer en handling. Indbygget i 'execute' ligger metoder til at reservere nødvendige ressourcer, samt frigive tidligere brugte ressourcer.

### 4.5.1 Detektion af deadlocks

**Til opdagelse af deadlocks** kreeres en monitor klasse der opretter og vedligeholder en orienteret graf over agenter, ressourcer, samt sammenhængen mellem disse; grafen benævnes ressource-allokerings graf (Eng: Resource-allocation graph). Noder i grafen består af alle agenter ( $A_1, A_2..A_n$ ) og alle ressourcer ( $R_1, R_2..R_n$ ).

En orienteret kant ( $A_i \rightarrow R_j$ ) symboliserer at agent  $A_i$  forespørger på ressource  $R_j$  og venter indtil den frigives. Når  $R_j$  tildeles  $A_j$  indikeres det i grafen med kanten  $R_j \rightarrow A_j$ . En *forespørgelseskant*  $A_i \rightarrow R_j$  konverteres til en *tildelingskant* når  $R_j$  allokeres til  $A_i$ . Tildelingskanten slettes når  $R_j$  frigives.



Det følger umiddelbart, at hvis der eksisterer en cyclus i grafen, så er systemet i deadlock (Her antaget at der kun forefindes en instans af hver ressource type). For at optimere søgningen på en sådan cyklus oprettes og vedligeholdes en 'Vent-på graf' (Eng: Wait-for graph) som bygges ud fra ressource-allokerings grafen (RAG). Grafen illustrerer hvilke agenter det holder i tomgang - hver node repræsenterer en agent der venter på en eller flere ressourcer. En kant i grafen består af agent  $A_i$  i RAG der venter på en ressource som agent  $A_j$  holder. Parallelt til RAG ses det umiddelbart, at hvis der eksisterer en cyclus i Vent-på grafen, så eksisterer der en deadlock.

En metode til at gennemsøge en graf for cyklusser er givet i [4]. Her benyttes en Depth-First Search (Dk: Dybde-først søgning), og når en *back-edge* findes så eksisterer der en cyklus. Da [4] antages bekendt vil vi ikke gå i nærmere detaljer med DFS. Pseudokoden til algoritmen har vi dog listet i appendix A. Endvidere bemærkes at når en 'grå' node støder på en anden grå node under udforskningen af grafen, så er der fundet en back-edge.

Hyppigheden af antallet af søgninger igennem systemet, for at opdage en deadlock afhænger af hvor ofte en deadlock kan forventes at opstå; samt hvor vigtigt tidsfaktoren for opdagelsen er. I de fleste systemer vil det være overkill at gennemsøge for deadlock, hver gang en agent skal vente på en ressource. Da man, afhængigt af situationen, alt andet end lige må forvente at ressourcen frigives

på et tidspunkt. En tidsinstans for hyppigheden af søgningen virker derved fornuftigt, og vil i vores LEGO verden ligge på omkring 3 sekunder.

**Udredning af deadlocks** kan som nævnt foregå ved at en operatør klarer det manuelt; altså fysisk fjerner en eller flere robotter, samt frigiver disses ressourcer. Alternativet er at systemet implementerer en automatisk procedure til at genskabe et sikkert stadie for sig selv. Da LEGO verdenen dikterer at en agent ikke kan 'afbrydes', som man ville kunne gøre det med en trådstyret process, må proceduren løse det vha. faste reetableringsmetoder. Afbrydelsen af en agent kan simuleres ved at på LEGO banen lave særlige holdepladser for agenterne, hvor de helt kan slippe alle holdte ressourcer og ad den vej frigøre de andre agenter. Når deadlocken er løst, må agenten(muligvis flere) lægge en ny plan for hvorledes den vil løse dens *gamle* opgave. Agenternes samlede ydelse vil være alvorligt påvirket af en sådan løsning. Istedet kunne man forsøge at udrede problemet. Da det er muligt at finde de implicerede agenter i deadlocken (alle agenter der er en del af en *back edge* i DFS søgningen), kunne man lægge en ny plan for en agent ad gangen indtil deadlocken var løst. I den nye plan skulle de knudepunkter, der optages af andre agenter i deadlocken, ikke være mulige at betræde. Disse løsninger vil i givet fald bryde deadlocken.

## 4.6 Implementation

I denne implementationssektion vil vi kun beskrive specifikke problemstillinger, hvor kodeløsningen ikke umiddelbart følger af ovenstående sektioner.

Til uddeling af jobs, håndtering af ressourcer og overvågning om systemet er i en deadlock situation, har vi udviklet klassen `world`. Heri gemmes betingelser der beskriver verdens antagede 'sande' tilstand. Disse oplysninger opdateres vha. proceduren `UpdateWorld`, hver gang en agent udfører en handling. `UpdateWorld` beskyttes mod at parallel tråde (flere agenter) for adgang til metoden med C#'s indbyggede trådmonitor (`Thread monitor`). Ved at benytte en trådmonitor istedet for semaforer kan løbsbetingelser (Eng: *race-conditions*) undgås. Uddeling af jobs foregår simpelt ved at gemme målbetingelser i `Jobs`; et 'Dictionary' (eller `HashMap`), hvor nøglen udgøres af den agent der er udvalgt til at løse opgaven. Hvis ingen special agent er påkrævet tilføjes opgaven til en default værdi. En agent modtager sine *ønsker* fra dette hashmap, ved at få alle opgaver med eget 'navn' som nøgle, samt alle opgaver med defaultværdi. En opgave slettes fra hashmappet når en agent dedikere sig til problemet (i.e. benytter filter metoden). Hashmappet jobs er beskyttet mod multibenyttelse (Eng: *Concurrent use*) ligeledes med en trådmonitor.

**Tilegnelse af ressourcer** for agenter foregår som nævnt efter parolen:

*Forespørgsel  $\Rightarrow$  Benyttelse  $\Rightarrow$  Frigivelse*

Og kaldes i `world.ExecuteAction`. Før en ressource kan optages undersøges allokeringssgraften for om ressourcen er benyttet. Hvis den benyttes til anden side låses agenten af en semafor dedikeret til agenten. Denne semafor frigives når ressourcen er ledig. Efter optagelse af ressourcen tilføjes agenten i ressourceallokeringsgraften som ejer og handlinger på denne kan frit udføres. Når handlingerne er udført frigives ressourcen. Forefindes der en agent som *kun* har ventet på denne ressource, frigives agentens semafor.

### 4.6.1 Deadlockmonitor

Deadlockmonitoren er implementeret som en selvstændig tråd, der kører i uendelig lykke. Monitoren undersøger hvert tredje sekund om der eksisterer en deadlock. Det gøres som nævnt ved at søge igennem Vent-på graffen med en dybde-først søgning og registrere alle agenter der indgår i en *back edge*. Disse agenter modtager alle en meddelelse om at de er gået i deadlock.. Besøgte 'agenter' i Vent-på træet (noder med farven sort), gemmes i et hashmap 'visited'. Ligeledes gemmes agentnoder der er under undersøgelse i et hashmap 'greys'. Hvide noder udgøres af agenter i ingen af de to nævnte.

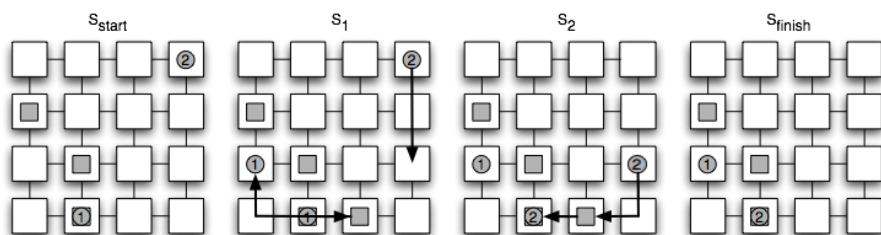
## 4.7 Test

Vi har gennemført en serie tests i selve LEGO verdenen med op til tre agenter. B-DI'en fungerede fint i disse opgaver og var i stand til at genplanlægge, så snart en tidligere plan viste sig ugyldig. World opdateringen og uddelegeringen af opgaver fungerede også efter hensigten, og datadelingen mellem det fælles register og de enkelte agenter gav ikke anledning til at revidere multitråds beskyttelsen; ved en noget mere omfattende test ville man have testet beskyttelsen vha. spin og se om der eksempelvis kunne findes løbsbetingelser. Metoderne til ressourcedelegering (forespørg, vent, frigiv) lod til at virke således agenterne ikke kolliderede.

Deadlockmonitoren opdagede de nødvendige deadlocks, og agenter fik besked på at de befandt sig i deadlock. Hvis en agent stødte til en allerede etableret deadlock, blev det korrekt opfanget af monitoren og den fik givet den korrekte besked.

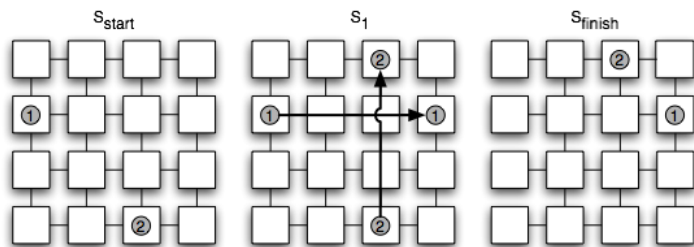
Ved en udbygning af systemet, vil det formentlig være nødvendigt med en mere omfattende teststrategi, der dog vil være tidsmæssigt krævende da det hovedsageligt vil foregå med 'live' LEGO forsøg.

### 4.7.1 Testcase 1



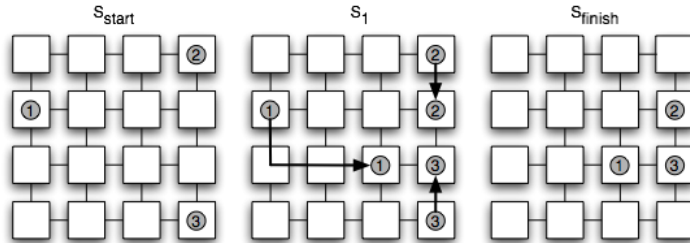
Denne testcase omhandler genplanlægning der viser, som på illustrationen, hvordan at robot nr 1 flytte en genstand over på et nyt felt. Robot nr 2 finder ud af at dens plan nu ikke fungerer og lægger en ny plan, som indbefatter at opsamle genstanden som robot nr 1 har lagt i vejen.

### 4.7.2 Testcase 2



Her testes ressourceuddeling. De 2 robotter har lagt en plan, som indebærer at de skal krydse hinandens bane. Robot nr 2 når først op og robot nr 1 venter så på det forrige felt indtil robot nr 2 har forladt det spærrede felt.

## 4.7.3 Testcase 3



Denne testcase ender i en deadlock, da vores implementation kun har en deadlock detection og ikke kender til nogen måde at løse problemet på. Når en robot sidder fast i en deadlock vil den signalere med lyd og lys at den sidder fast. I dette eksempel vil 2 & 3 signalere at de er i deadlock, siden støder 1 til.

## 4.8 Diskussion

Som nævnt er det implementerede multiagent system endnu i udviklingsfasen - læs en konceptløsning. Deraf findes der problemsituationer som er bekendte, men kunne bruge en mere fuldstændig løsning. Her nævnes enkelte ting der vil udgøre åbenlyse forbedringer.

I teori afsnittet omkring [Woolridge]'s definition på korrekt agent opførsel, er det under kommunikationen nævnt at agenter evt. skal kunne dele opgaver imellem sig. Et ønskeligt scenarie ville være hvis agenterne udviste såkaldt '*social adfærd*' og sammen argumenterede for hvem der havde de bedste forudsætninger for at løse en opgave, samt hjælpe hinanden med at løse problemer, der ellers ville være umulige. POP kan, *forholdsvis* uden de store ændringer, implementere at planlægge for flere agenter af gangen. Agenterne vil derved være styret af en fælles hjerne og er derfor, per definition, ikke en løsning for sociale multiagenter; da en agent defineres som autonomt styret. En løsning vil i højere grad centreres omkring en kommunikations og adfærdsprotokol, som ville koordinere adfærden mellem agenterne. En nærmere undersøgelse må vente til en lejlighed byder sig, eller overlades til læseren.

I en fuldtimplementeret BDI struktur, vil en del agentens opfattelse stamme fra sensorer på og omkring agenten. Oplysninger fra disse vil, sammen med info fået gennem kommunikation med andre agenter og registre, udgøre agentens

idé om verden. LEGO robotterne har rig mulighed for at blive udstyret med forskellige sensorer, men i den nuværende udformning mangler de evnen til at opfatte bokse der står i vejen, når robotten selv holder en bold. Derved er verden nød til at være statisk kendt, hvori der ikke kan foretages ændringer af andre end LEGO agenterne. Disse ændringer skal gemmes og distribueres videre til de nødvendige aktører.

Deadlockudredningen skal naturligvis fuldt implementeres. Til dette kunne de nævnte løsninger eventuelt benyttes. I øvrigt bemærkes det at ved at have valgt en centralt system til repræsentation af verden, ressourcestyring, jobdelegering etc. er hele processen meget sårbar overfor sammenbrud i dette register; et subsidaritetsprincip kunne med fordel benyttes, men ville tilføje en del til kompleksiteten uden de store landvindinger.

## 4.9 Konklusion

I dette kapitel har vi kreeret en konceptløsning på hvorledes multiagenter i LEGO verdenen kan håndteres. Systemet har implementeret en modificeret B-DI arkitektur, således at agenterne kan operere i en dynamisk skiftende verden og stadig benytte POP\* i planlægningsmodulet. Vi har analyseret systemet for hvorledes deadlocks og ressourcestyring vil kunne benyttes.

Til dette har vi lavet en overordnet system der uddelegere opgaver, vedligeholder en repræsentation af verden, styrer ressourcer og overvåger deadlocks. Udvidelser kan med fordel tilføjes systemet og af dem vi har nævnt, vil det ikke umiddelbart kræve en revision af den basale platform.

# Konklusion

---

Vi har udviklet og implementeret en kunstig intelligens, der kan generere en handlingssekvens til at opnå givne betingelser. Til dette har vi designet og implementeret et delvist rangeret planlægningsmodul, der understøtter STRIPS. Vi har undervejs afprøvet planlægningen og eksekvering på LEGO Mindstorm NXT robotter.

Vi har vist hvorledes planlægningsmodulet kan anvendes til at udgøre planlægningsfasen i agentstyring. Det er blevet demonstreret ved at inkorporere POP i BDI arkitekturen. Endvidere har vi undersøgt teoretiske aspekter ved flere agenter der opererer i samme miljø.

Vi har undervejs konstateret at det kræver *meget* nøje design af handlinger og betingelser for at planlægningen bliver konsistent. STRIPS er intuitivt at forstå, men dets resulterende planrums grafer er ganske komplekse. Et højere abstraktionsniveauet ville udgøre en stor fordel for designeren.

LEGO verdenen har vist sig ikke at være idéel til STRIPS planlægning. Omvendt kunne man konkludere at STRIPS, uden nødvendige udvidelser, ikke er alsidig nok til let at påføre selv en simpel LEGO verden. Udvidelserne ville skulle starte med en revision af beskrivelsessproget.

Idéer og design af POP, BDI strukturen og agentkoordineringen kan med fordel

videreudvikles og benyttes i fremtidigt arbejde.

# Litteratur

---

- [1] P. B. Galvin A. Silberschatz, J. L. Peterson. *Operating System Concepts trd. edi.*, pages 195–224.
- [2] BenDi. Priority queue.  
<http://www.codeproject.com/csharp/PriorityQueue.asp>.
- [3] Tom Bylander. The computational complexity of propositional strips planning. Technical report, Matematics, Computer Science, and Statstics, 1994.
- [4] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. MIT Press/McGraw-Hill, second edition, 2001.
- [5] Edsger W. Dijkstra. Bankers algorithm.  
<http://www.cs.utexas.edu/users/EWD/ewd06xx/EWD623.PDF>.
- [6] LEGO. Lego mindstorm nxt tribot.  
[http://mindstorms.lego.com/Overview/MTR\\_Tribot.aspx](http://mindstorms.lego.com/Overview/MTR_Tribot.aspx).
- [7] Anders Lemke, Johan Lidlaw, and Lars Zilmer-Pedersen. Developing multi-agent lego robotics. Technical report, IMM, 2007.
- [8] Steven Minton, John Bresina, and Mark Drummond. Total-order and partial-order planning: A comparative analysis. Technical report, NASA Ames Research Center, 1994.
- [9] XuanLong Nguyen and Subbarao Kambhampati. Reviving partial order planning. In *IJCAI*, pages 459–466, 2001.
- [10] Prabha Ramakrishnan. Intelligent control of autonomous underwater vehicles - a partial order planner for the orca project. Master's thesis, University of Maine System, 1997.

- 
- [11] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach*. Prentice Hall International Series in Artificial Intelligence. Prentice Hall, 2003.
- [12] Daniel S. Weld. An introduction to least commitment planning. *AI Magazine*, 15(4):27–61, 1994.
- [13] Wikipedia. Partially ordered set.  
[http://en.wikipedia.org/wiki/Partially\\_ordered\\_set](http://en.wikipedia.org/wiki/Partially_ordered_set).
- [14] Wikipedia. Strips.  
<http://en.wikipedia.org/w/index.php?title=STRIPS&oldid=121703234>.
- [15] Wikipedia. Unit testing.  
[http://en.wikipedia.org/wiki/Unit\\_testing](http://en.wikipedia.org/wiki/Unit_testing).
- [16] Michael Wooldridge. *Introduction to MultiAgent Systems*. John Wiley and Sons, 2002.
- [17] Michael Wooldridge and Nicholas R. Jennings. Intelligent agents: Theory and practice. *Knowledge Engineering Review*, 10(2):115–152, 1995.
- [18] Michael J. Wooldridge. *Reasoning about Rational Agents*. The MIT Press, Cambridge, Massachusetts, 2000.

## BILAG A

# Pseudokode

---

I følgende listes pseudokode for forskellige algoritmer der har været benyttet i programmet.

---

```

1: procedure DFS(G)
2:   for each vertex  $u \in V[G]$  do
3:     color[u] := WHITE;
4:      $\pi[u]$  := NIL; ▷  $\pi$  is node  $u$ 's parent
5:   end for
6:   for each vertex  $u \in V[G]$  do
7:     if color[u] = WHITE then
8:       DFS-VISIT(u);
9:     end if
10:  end for
11: end procedure

```

```

1: procedure DFS-VISIT(u)
2:   color[u] := GREY; ▷ White vertex has just been discovered
3:   for each  $v \in \text{Adj}[u]$  do ▷ Explore edge(u, v)
4:     if color[v] = GREY then
5:       PRINT "Back-Edge Found";
6:       terminate DFS;
7:     else if color[v] = WHITE then
8:        $\pi[v]$  := u;
9:       DFS-VISIT(v);
10:    end if
11:  end for
12:  color[u] := BLACK; ▷ Blacken u; it is finished
13: end procedure

```

### Depth First Search

---

---

```
1: procedure TOPOLOGICAL SORT( $Q$ )
2:    $Q \leftarrow$  Set of all nodes with no incoming edges
3:   while  $Q$  is non-empty do
4:     remove a node  $n$  from  $Q$ 
5:     output  $n$ 
6:     for each node  $m$  with an edge  $e$  from  $n$  to  $m$  do
7:       remove edge  $e$  from the graph
8:       if  $m$  has no other incoming edges then
9:         insert  $m$  into  $Q$ 
10:      end if
11:    end for
12:  end while
13:  if graph has edges then
14:    output error message (graph has a cycle)
15:  end if
16: end procedure
```

### Topological Sort

---

```

1: procedure MAX-HEAPIFY( $A, i$ )
2:    $l \leftarrow LEFT(i)$ ;
3:    $r \leftarrow RIGHT(i)$ ;
4:   if  $l \leq heap-size[A]$  and  $A[l] > A[i]$  then
5:      $largest \leftarrow l$ ;
6:   else  $largest \leftarrow i$ ;
7:   end if
8:   if  $r \leq heap-size[A]$  and  $A[r] > A[largest]$  then
9:      $largest \leftarrow r$ ;
10:  end if
11:  if  $largest \neq i$  then
12:    exchange  $A[i] \leftrightarrow A[largest]$ ;
13:  end if
14:  Max-Heapify( $A, largest$ );
15: end procedure

```

```

1: procedure HEAP-MAXIMUM( $A$ )
2:   return  $A[1]$ ;
3: end procedure

```

```

1: procedure HEAP-EXTRACT-MAX( $A$ )
2:   if  $heap-size[A] < 1$  then
3:     print error 'heap underflow';
4:   end if
5:    $max \leftarrow A[1]$ ;
6:    $A[1] \leftarrow A[heap-size[A]]$ ;
7:    $heap-size[A] \leftarrow heap-size[A] - 1$ ;
8:   Max-Heapify( $A, 1$ );
9:   return  $max$ ;
10: end procedure

```

```

1: procedure HEAP-INCREASE-KEY( $A, i, key$ )
2:   if  $key < A[i]$  then
3:     print error 'new key is smaller than current key';
4:   end if
5:    $A[i] \leftarrow key$ ;
6:   while  $i > 1$  and  $A[PARENT(i)] < A[i]$  do
7:     exchange  $A[i] \leftrightarrow A[PARENT(i)]$ ;
8:      $i \leftarrow PARENT(i)$ ;
9:   end while
10: end procedure

```

```

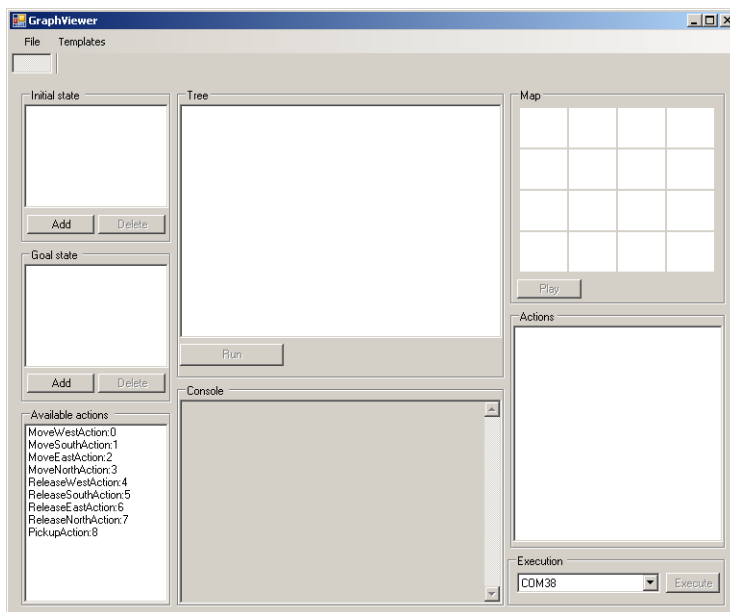
1: procedure MAX-HEAP-INSERT( $A, key$ )
2:    $heap-size[A] \leftarrow heap-size[A] + 1$ ;
3:    $A[heap-size[A]] \leftarrow -\infty$ 
4:   Heap-Increase-Key( $A, heap-size[A], key$ );
5: end procedure

```

# BILAG B

## Grafisk brugergrænseflade

---



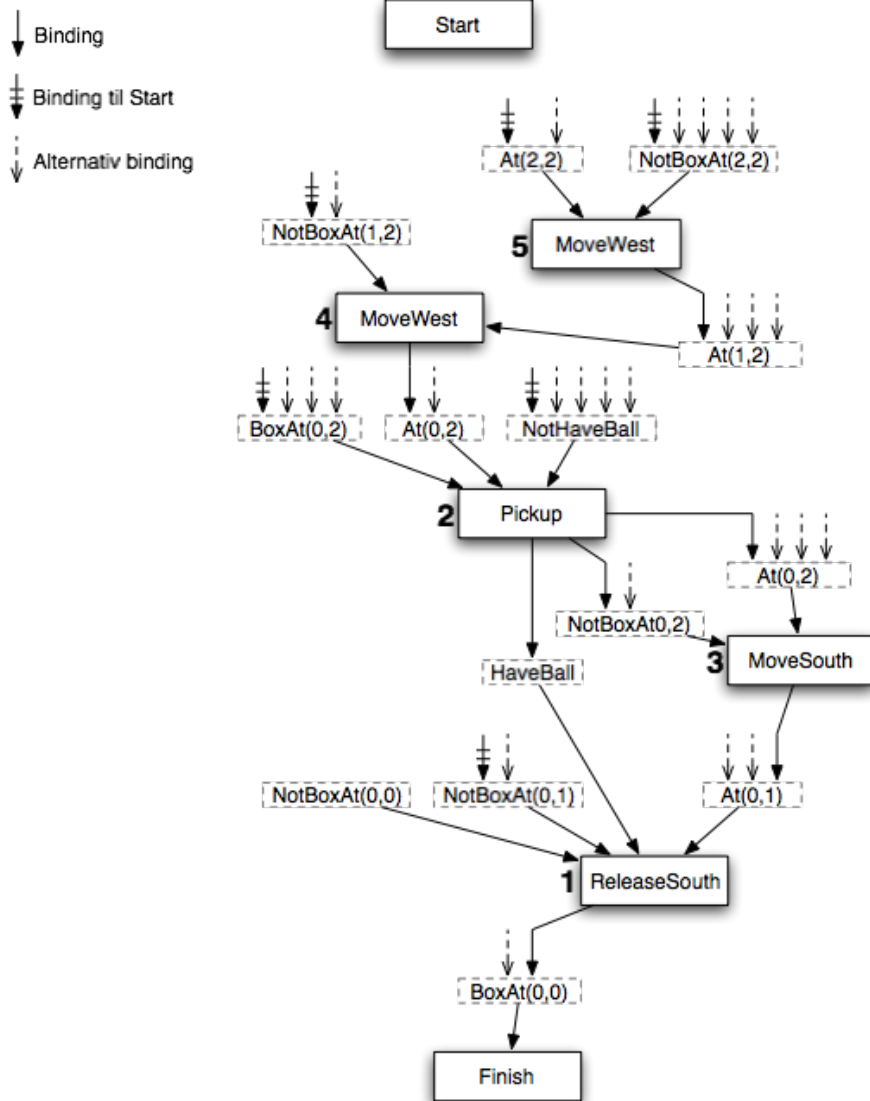


## BILAG C

# Kausal link graf

---

Følgende graf viser et hvordan kausal link dannes i eksemplet. Bindingerne til Start og alternative bindinger, vises som små pile for at lette overskueligheden.



## D.1 Plan.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4 using MultiRoboticAgentsPlanning;
5
6 namespace MultiRoboticAgentsPlanning
7 {
8     public class Plan : IComparable, IComparable<Plan>
9     {
10         // List of actions used in the plan
11         public List<Action> actions = new MyList<Action>();
12
13         // List of previous actions as a link
14         public List<Link> previousActions = new MyList<Link>();
15
16         // The StartAction
17         public Action StartAction;
18
19         // List of causal links
20         public List<CausalLink> CausalLinks = new MyList<CausalLink>();
21
22         // List of ordering pairs.
23         // The key node has a directed edge to all of the nodes in the
24         list
```

```

24     public Dictionary<Action, List<Action>> orders = new Dictionary
        <Action, List<Action>>();
25
26     // All conditions there are bound to StartAction's conditions
27     public Dictionary<Action, List<Condition>>
        EssentielStartConditions = new Dictionary<Action, List<
            Condition>>();
28
29     /*
30     * A set of open preconditions. A precondition is open if it is
        not achieved by some action PRECONDITIONS in the plan.
31     * Planners will work to reduce the set of open preconditions
        to the empty set, without introducing a contradiction.
32     */
33     public List<Link> OpenPreconditions = new MyList<Link>();
34
35     // Constructor
36     public Plan(Action StartAction, Action FinishAction, List<
        Condition> FinishConditions)
37     {
38         this.StartAction = StartAction;
39
40         // Add the first action to actions
41         actions.Add(FinishAction);
42
43         // Place all conditions from Finish in the OpenPrecondition
            list
44         foreach (Condition c in FinishConditions)
45         {
46             Link FinishLink = new Link(null, c, FinishAction);
47             OpenPreconditions.Add(FinishLink);
48         }
49
50         // Add the initial order Start -> Finish
51         AddOrder(StartAction, FinishAction);
52     }
53
54     // Private constructor, used by Clone()
55     private Plan()
56     {
57     }
58
59     /*
60     * Add a ordering constraints only if there does not exists a
        cycle after
61     * There is using a DFS to find if there is a potential cycle
62     */
63     public bool AddOrder(Action before, Action after)
64     {
65         if (after is StartAction) return false;
66
67         // Search the graph for any cycles
68         Queue<Action> Q = new Queue<Action>();
69         Dictionary<Action, bool> visited = new Dictionary<Action,
            bool>();

```

```
70     Q.Enqueue(after);
71     int i = 0;
72     while (Q.Count != 0)
73     {
74         i++;
75         Action a = Q.Dequeue();
76
77         if (a.Equals(before))
78         {
79             return false;
80         }
81
82         visited.Add(a, true);
83         try
84         {
85             foreach (Action ap in orders[a])
86             {
87                 if (!visited.ContainsKey(ap) && !Q.Contains(ap))
88                 {
89                     Q.Enqueue(ap);
90                 }
91             }
92         }
93         catch (KeyNotFoundException)
94         {
95         }
96     }
97
98     List<Action> list;
99     try
100    {
101        list = orders[before];
102    }
103    catch (KeyNotFoundException)
104    {
105        list = new List<Action>();
106        orders.Add(before, list);
107    }
108    if (!list.Contains(after))
109    {
110        list.Add(after);
111    }
112
113    return true;
114 }
115
116 /**
117  * Check if the plan is a solution to the whole problem
118  * Use a topological sort to determine if there exist a linear
119     extension
120 */
121 public bool IsSolution()
122 {
123     if (OpenPreconditions.Count == 0 && GetOrdersTopologicalSort
124         () != null)
```

```

123     {
124         return true;
125     }
126     return false;
127 }
128
129 // Add an action to the plan.
130 public bool AddAction(Link preActionLink, Condition
    subgoalCondition, Action subgoalAction)
131 {
132     Action preAction = preActionLink.ActionPre;
133
134     // Add ordering constrains
135     if (!AddOrder(preAction, subgoalAction))
136     {
137         return false;
138     }
139
140     // Add the causal link
141     CausalLink newlink = new CausalLink(preAction,
    subgoalCondition, subgoalAction);
142     CausalLinks.Add(newlink);
143
144     // Update plan:
145     Console.WriteLine("Use:_" + preAction);
146     if (preActionLink.Condition == null)
147     {
148         OpenPreconditions = preAction.NewOpenPreconditions(
    OpenPreconditions, newlink, this);
149         actions.Add(preAction);
150     }
151     else
152     {
153         Console.WriteLine("Previous_action_used");
154         //previousActions.Remove(preActionLink);
155         foreach (Link link in OpenPreconditions)
156         {
157             if (link.Condition.Equals(subgoalCondition))
158             {
159                 OpenPreconditions.Remove(link);
160                 break;
161             }
162         }
163     }
164     return true;
165 }
166
167 // Returns a list of threats
168 public List<CausalLink> GetThreatLinks(Link unresolvedLink,
    Link selectedPreActionLink)
169 {
170     List<CausalLink> Threats = new MyList<CausalLink>();
171     Action selectedPreAction = selectedPreActionLink.ActionPre;
172
173     Condition unresolvedCondition = unresolvedLink.Condition;

```

```

174     foreach (CausalLink causallink in new List<CausalLink>(
175         CausalLinks))
176     {
177         if (unresolvedCondition.Threatens(causallink.Condition) &&
178             !selectedPreAction.Equals(causallink.PostAction)
179             && !selectedPreAction.Equals(causallink.PreAction))
180         {
181             Threads.Add(causallink);
182         }
183         else if( //Make sure the condition C1 used by an action A1
184                 that changes it, is not used by another action A2.
185                 causallink.PreAction.Equals(selectedPreAction) //start ==
186                     start
187                 && ! causallink.PostAction.Equals(unresolvedLink.
188                     ActionPost) //pickup1 != pickup2
189                 && causallink.Condition.GetType().Equals(
190                     unresolvedCondition.GetType()) //nothaveball ==
191                     nothaveball
192                 && unresolvedLink.ActionPost.GetAlterPreList().Contains(
193                     unresolvedCondition.GetType()) //pickup2.alternates(
194                     nothaveballcondition)
195                 && (unresolvedCondition is ConditionVariable
196                 && ((ConditionVariable)unresolvedCondition).IsSamePlace(
197                     ((ConditionVariable)causallink.Condition)
198                 || !(unresolvedCondition is ConditionVariable))) //
199                 boxat(1,0) == boxat(1,0) el. nothaveball og ikke
200                 boxat(1,0) == boxat(0,0)
201         {
202             CausalLinks.Remove(causallink);
203             OpenPreconditions.Add(new Link(causallink.PreAction,
204                 causallink.Condition, causallink.PostAction));
205         }
206     }
207     return Threads;
208 }
209
210 // Make a copy of the plan
211 public Plan Clone()
212 {
213     Plan plan = new Plan();
214     plan.actions = new MyList<Action>(this.actions);
215     plan.CausalLinks = new MyList<CausalLink>(this.CausalLinks);
216     plan.orders = new Dictionary<Action, List<Action>>();
217     foreach (KeyValuePair<Action, List<Action>> k in this.orders)
218         plan.orders.Add(k.Key, new List<Action>(k.Value));
219     plan.OpenPreconditions = new MyList<Link>(this.
220         OpenPreconditions);
221     plan.previousActions = new MyList<Link>(this.previousActions)
222         ;
223     plan.StartAction = this.StartAction;
224
225     return plan;
226 }

```

```

214
215     public override string ToString()
216     {
217         StringBuilder s = new StringBuilder();
218         foreach (Action a in actions)
219             {
220                 s.Append(a+"\n");
221             }
222
223         return s.ToString();
224     }
225
226     // Make a linear extension from the ordering constraints via a
227     // topological sort algorithm
228     public LinkedList<Action> GetOrdersTopologicalSort()
229     {
230         Dictionary<Action, List<Action>> ordercopy = new Dictionary<
231             Action, List<Action>>();
232         foreach (KeyValuePair<Action, List<Action>> k in this.orders)
233             {
234                 ordercopy.Add(k.Key, new List<Action>(k.Value));
235             }
236
237         LinkedList<Action> result = new LinkedList<Action>();
238
239         Queue<Action> Q = new Queue<Action>();
240         Q.Enqueue(StartAction);
241
242         while (Q.Count > 0)
243         {
244             Action n = Q.Dequeue();
245             result.AddLast(n);
246
247             try
248             {
249                 foreach (Action m in new List<Action>(ordercopy[n]))
250                 {
251                     ordercopy[n].Remove(m);
252                     if (ordercopy[n].Count == 0)
253                         ordercopy.Remove(n);
254
255                     bool haveothers = false;
256                     foreach (KeyValuePair<Action, List<Action>> l in
257                         ordercopy)
258                     {
259                         if (l.Value.Contains(m))
260                             haveothers = true;
261                     }
262                     if (!haveothers) Q.Enqueue(m);
263                 }
264             }
265             catch (KeyNotFoundException) {

```

```

266     if (ordercopy.Count != 0)
267         return null;
268
269     return result;
270 }
271
272 public int CompareTo(object p)
273 {
274     if (p is Plan)
275         return this.CompareTo((Plan)p);
276     return 0;
277 }
278
279 // Compare two plans and rank the plan with the lowest
    // heuristics first
280 public int CompareTo(Plan p)
281 {
282     return this.Heuristic.CompareTo(p.Heuristic);
283 }
284
285 // Calculated heuristic
286 private int calculatedHeuristic = -1;
287
288 // Return the heuristic value.
289 // This value is only calculated once and stored in the private
    // variable calculatedHeuristic
290 public int Heuristic
291 {
292     get
293     {
294         if (calculatedHeuristic == -1)
295         {
296             calculatedHeuristic = 0;
297
298             // The weight of the current plan
299             foreach (Action a in actions)
300                 calculatedHeuristic += a.Weight;
301
302             // Estimate a weight of how many actions there is needed
    // for all conditions are resolved
303             foreach (Link c1 in OpenPreconditions)
304             {
305                 // Calculate the distance to a object and let this
    // weight include in the heuristic
306                 if (c1.Condition is AtCondition)
307                     foreach (Condition c2 in ((StartAction)StartAction).
    // PostConditions)
308                         if (c2 is AtCondition)
309                             {
310                                 int distance = ((AtCondition)c1.Condition).
    // DistanceTo(((AtCondition)c2));
311                                 if (distance != int.MaxValue)
312                                 {
313                                     calculatedHeuristic += distance;
314                                     break;

```

```

315         }
316     }
317     calculatedHeuristic++;
318 }
319 }
320 return calculatedHeuristic;
321 }
322 }
323
324 // Find all essential start conditions
325 internal void CalcEssentialStartConditions()
326 {
327     foreach (CausalLink calink in CausalLinks)
328     {
329         if (calink.PreAction is StartAction)
330         {
331             if (EssentielStartConditions.ContainsKey(calink.
332                 PostAction))
333             {
334                 EssentielStartConditions[calink.PostAction].Add(calink.
335                     Condition);
336             }
337             else
338             {
339                 List<Condition> temp = new List<Condition>();
340                 temp.Add(calink.Condition);
341                 EssentielStartConditions.Add(calink.PostAction, temp);
342             }
343         }
344     }
345 }

```

## D.2 POPstar.cs

```

1 using System;
2 using System.Collections.Generic;
3 using BenTools.Data;
4 using System.Threading;
5
6 namespace MultiRoboticAgentsPlanning
7 {
8     public class POPstar
9     {
10         // Events
11         public event EventHandler OnStarted;
12         public event EventHandler OnStoped;
13         public event EventHandler<PlanEventArgs> OnDequeue;
14         public event EventHandler<PlanEventArgs> OnNewPlanInQueue;
15         public event EventHandler<PlanEventArgs> OnSuccesFinish;
16         public event EventHandler OnFailedFinish;
17

```

```

18     // Prioritized queue with the best plan on top
19     private IPriorityQueue Queue = new BinaryPriorityQueue();
20
21     // Returns the best plan there exists in the queue. It is not
22         garantered that it is a complete plan
23     public Plan BestPlan
24     {
25         get
26         {
27             return (Plan)Queue.Peek();
28         }
29     }
30
31     // List of all available actions
32     private List<Action> Actions;
33
34     // List of all conditions in the initial state
35     List<Condition> InitialState;
36
37     // List of all conditions in the goal state
38     List<Condition> GoalState;
39
40     public POPstar(List<Condition> StartState, List<Condition>
41         GoalState, List<Action> Actions)
42     {
43         // Store all available actions, initial state conditions and
44             goal state conditions
45         this.Actions = Actions;
46         this.InitialState = StartState;
47         this.GoalState = GoalState;
48
49         // The initial and goal actions
50         StartAction StartAction = new StartAction(InitialState);
51         FinishAction FinishAction = new FinishAction(GoalState);
52
53         // Making the global plan
54         Plan plan = new Plan(StartAction, FinishAction, GoalState);
55
56         // Pushing the first element in to the queue
57         Queue.Push(plan);
58     }
59
60     internal void Start()
61     {
62         // Call the event OnStarted
63         if (OnStarted != null) OnStarted(this, new EventArgs());
64
65         // Do as long as there exists a plan in the queue
66         while (Queue.Count > 0)
67         {
68             // Extract the currently best plan
69             Plan plan = (Plan)Queue.Pop();
70
71             // Call the OnDequeue event

```

```

69     if (OnDequeue != null) OnDequeue(this, new PlanEventArgs(
70         plan));
71     Console.WriteLine("Look_at_" + plan.actions[plan.actions.
72         Count - 1]);
73     // If the current plan is a solution to the problem, then
74     // return plan
75     if (plan.IsSolution())
76     {
77         if (OnSuccessFinish != null) OnSuccessFinish(this, new
78             PlanEventArgs(plan));
79         Queue.Push(plan);
80         return;
81     }
82     else if (plan.OpenPreconditions.Count == 0)
83         continue;
84     //Selecting a subgoal to solve
85     Link SubGoal = SelectSubgoal(plan);
86     Console.WriteLine("SubGoal:_" + SubGoal);
87     // Selecting all actions there possible can solve the
88     // subgoal
89     List<Link> saddlist = ChooseAllOperators(SubGoal, plan);
90
91     // Foreach action there achieves the current subgoal
92     foreach (Link PreAction in saddlist)
93     {
94         // Make a copy of the plan
95         Plan newplan = plan.Clone();
96
97         // If it is possible to add the action without violating
98         // the order
99         if (newplan.AddAction(PreAction, SubGoal.Condition,
100             SubGoal.ActionPost))
101         {
102             // Retrieve all threats
103             List<CausalLink> threats = newplan.GetThreatLinks(
104                 SubGoal, PreAction);
105
106             // Try to resolve the threats. If succeed return a plan
107             // without threats else return null
108             newplan = ResolveThreats(newplan, PreAction.ActionPre,
109                 threats);
110
111             // If it was possible to remove the treats
112             if (newplan != null)
113             {
114                 // Put the new plan in the queue
115                 Queue.Add(newplan);
116                 // Call the OnNewPlanInQueue event
117                 if (OnNewPlanInQueue != null) OnNewPlanInQueue(this,
118                     new PlanEventArgs(plan, newplan));
119             }
120         }
121     }

```

```

113         }
114     }
115 }
116     Console.WriteLine();
117 }
118
119 // Call the OnFailedFisish event
120 if (OnFailedFinish != null) OnFailedFinish(this, new
    EventArgs());
121 return;
122 }
123
124 #region POP internal methods
125
126 // Selecting a subgoal to solve
127 private Link SelectSubgoal(Plan plan)
128 {
129     // Returns the first condition in the list
130     return plan.OpenPreconditions[0];
131 }
132
133 // Find all actions there achieves the subgoal and return them
134 private List<Link> ChooseAllOperators(Link SubGoal, Plan plan)
135 {
136     List<Link> list = new MyList<Link>();
137
138     // Try to find any previous actions there can be reused to
139     // achieve the subgoal
140     foreach (Link link in plan.previousActions)
141     {
142         if (link.ActionPre == SubGoal.ActionPre)
143         {
144             continue;
145         }
146
147         if (link.Condition.Equals(SubGoal.Condition)
148             && !link.ActionPre.Equals(SubGoal.ActionPost)
149             && !(link.ActionPre is MoveAction && SubGoal.ActionPost
150                 is MoveAction)
151             && !(link.ActionPre is MoveAction && SubGoal.ActionPost
152                 is ReleaseAction)
153         )
154         {
155             list.Add(link);
156         }
157     }
158     //if subgoal is not bound the unmatched handle can still
159     //achieve the goal.
160     else if (!SubGoal.Condition.IsBothBound() && link.Condition
161         .IsBothBound() && SubGoal.Condition.GetType().Equals(
162             link.Condition.GetType())
163         && !(link.ActionPre.GetType().Equals(SubGoal.ActionPost.
164             GetType()))))
165     {
166         //(((ConditionVariable)SubGoal.Condition).BindVariables((
167             ConditionVariable)link.Condition);

```

```

159         ConditionVariable sc = ((ConditionVariable)SubGoal.
160             Condition);
161         ConditionVariable lk = ((ConditionVariable)link.Condition
162             );
163
164         if (sc.GetVariableX().IsBound())
165         {
166             if (!((ConditionVariable)SubGoal.Condition).GetIntegerX
167                 ().Equals(lk.GetIntegerX()))
168                 continue;
169         }
170         if (sc.GetVariableY().IsBound())
171         {
172             if (!((ConditionVariable)SubGoal.Condition).GetIntegerY
173                 ().Equals(lk.GetIntegerY()))
174                 continue;
175         }
176         list.Add(link);
177     }
178 }
179
180 // Add startaction if its achieves the subgoal
181 if (plan.StartAction.Achieves(SubGoal, plan) && plan.
182     StartAction != SubGoal.ActionPre)
183 {
184     list.Add(new Link(plan.StartAction, null, null));
185 }
186
187 // Add actions from all available actions it its achieves the
188     subgoal
189 foreach (Action action in Actions)
190 {
191     if (action.Achieves(SubGoal, plan))
192     {
193         list.Add(new Link((Action)Activator.CreateInstance(action
194             .GetType()), null, null));
195     }
196 }
197 return list;
198 }
199
200 // Try recursion to resolve the threats by making ordering
201     constrains
202 private Plan ResolveThreats(Plan plan, Action threat, List<
203     CausalLink> threats)
204 {
205     if (threats.Count == 0)
206     {
207         return plan;
208     }
209
210     // Promotion
211     Plan plan2 = plan.Clone();
212     if (plan.AddOrder(threat, threats[0].PreAction))
213     {

```

```

205         plan = ResolveThreats(plan, threat, threats.GetRange(1,
206             threats.Count - 1));
207         if (plan != null)
208         {
209             return plan;
210         }
211     }
212     // Demotion
213     if (plan2.AddOrder(threats[0].PostAction, threat))
214     {
215         plan2 = ResolveThreats(plan2, threat, threats.GetRange(1,
216             threats.Count - 1));
217         if (plan2 != null)
218         {
219             return plan2;
220         }
221     }
222     Console.WriteLine("Threat_Could_Not_Be_Resolved");
223     return null;
224 }
225 #endregion
226
227 #region Event arguments
228 public class PlanEventArgs : EventArgs
229 {
230     private Plan _parent;
231     private Plan _plan;
232
233     public Plan Plan
234     {
235         get
236         {
237             return _plan;
238         }
239         private set
240         {
241             _plan = value;
242         }
243     }
244
245     public Plan Parent
246     {
247         get
248         {
249             return _parent;
250         }
251         private set
252         {
253             _parent = value;
254         }
255     }
256
257     public PlanEventArgs(Plan plan) : base()

```

```

258     {
259         this.Parent = null;
260         this.Plan = plan;
261     }
262
263     public PlanEventArgs(Plan parent , Plan plan) : base()
264     {
265         this.Parent = parent;
266         this.Plan = plan;
267     }
268 }
269 #endregion
270 }
```

## D.3 Action.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Collections;
5  using NXTRemoteControl;
6
7  namespace MultiRoboticAgentsPlanning
8  {
9      public abstract class Action
10     {
11         protected static int numbercount = 0;
12         protected int number = 0;
13         protected List<Type> preconditionsTypes = new MyList<Type>();
14         protected List<Type> addConditionTypes = new MyList<Type>();
15         protected List<Type> alterAddConditionTypes = new MyList<Type>
16             >();
17         protected List<Type> alterPreConditionTypes = new MyList<Type>
18             >();
19         protected bool extraLinks1 = true;
20         protected bool extraLinks2 = false;
21
22         protected Action()
23         {
24             number = numbercount++;
25         }
26
27         public virtual void Execute(NXTControl nxt)
28         {
29
30         }
31         protected virtual bool ActionAchieves(Link subgoal , Plan p)
32         {
33             return true;
34         }
35
36         public virtual List<Link> NewOpenPreconditions(List<Link>
37             OpenPreconditions , CausalLink SubGoalLink , Plan p)
```

```

35     {
36         Condition SubGoalCondition = SubGoalLink.Condition;
37         MyList<Link> result = new MyList<Link>(OpenPreconditions);
38         MyList<Type> temp = new MyList<Type>(addConditionTypes);
39
40         Variable newVarX = new Variable();
41         Variable newVarY = new Variable();
42
43         Variable oldVarX = new Variable();
44         Variable oldVarY = new Variable();
45
46         foreach (Link l in OpenPreconditions)
47         {
48             Condition c = l.Condition;
49
50             /*
51              * For hver condition i OpenPreconditions, fjern alle som
52                matcher typerne
53              * i addconditionType listen, hvis conditionen er ubundet
54                eller
55              * hvis conditionen har samme koordinat (variabel) som
56                SubGoalCondition
57              */
58             if (SubGoalLink.PostAction.Equals(l.ActionPost) &&
59                 addConditionTypes.Contains(c.GetType()))
60             {
61                 if (!c.IsBothBound())
62                 {
63                     result.Remove(l);
64                     if (extraLinks1.p.CausalLinks.Add(new CausalLink(
65                         SubGoalLink.PreAction, c, SubGoalLink.PostAction)
66                     ));
67                 }
68                 else if (SubGoalCondition.IsBothBound())
69                 {
70                     if (SubGoalCondition is ConditionVariable)
71                     {
72                         if (((ConditionVariable)SubGoalCondition).IsSamePlace(
73                             ((ConditionVariable)c))
74                         )
75                         {
76                             result.Remove(l);
77                             if (extraLinks1.p.CausalLinks.Add(new CausalLink(
78                                 SubGoalLink.PreAction, c, SubGoalLink.
79                                 PostAction));
90                             }
91                         }
92                     }
93                 }
94             }
95         }
96
97         if (SubGoalCondition is ConditionVariable)
98         {
99             newVarX = newXVariable(((ConditionVariable)SubGoalCondition
100                 ).GetVariableX());

```

```

80         newVarY = newYVariable(((ConditionVariable)SubGoalCondition
81             ).GetVariableY());
82         oldVarX = ((ConditionVariable)SubGoalCondition).
83             GetVariableX();
84         oldVarY = ((ConditionVariable)SubGoalCondition).
85             GetVariableY();
86     }
87     foreach (Type PrecondType in preconditionsTypes)
88     {
89         Condition c;
90         if (PrecondType.IsSubclassOf(typeof(ConditionVariable)))
91         {
92             object [] args = new object [2];
93             args [0] = newVarX;
94             args [1] = newVarY;
95             c = (Condition) Activator.CreateInstance(PrecondType, args
96                 );
97         }
98         else
99         {
100             c = (Condition) Activator.CreateInstance(PrecondType);
101         }
102         result.Add(new Link(null, c, this));
103     }
104
105     object [] args1 = new object [2];
106     if (oldVarX.IsBound() && oldVarY.IsBound())
107     {
108         args1 [0] = oldVarX;
109         args1 [1] = oldVarY;
110     }
111     else
112     {
113         args1 [0] = newVarX;
114         args1 [1] = newVarY;
115     }
116
117     foreach (Type type in temp)
118     {
119         Condition c;
120         if (type.IsSubclassOf(typeof(ConditionVariable)))
121         {
122             c = (Condition) Activator.CreateInstance(type, args1);
123         }
124         else
125         {
126             c = (Condition) Activator.CreateInstance(type);
127         }
128         p.previousActions.Add(new Link(this, c, null));
129     }

```

```
130         if (extraLinks2) p.CausalLinks.Add(new CausalLink(
131             SubGoalLink.PreAction, c, SubGoalLink.PostAction));
132     }
133     return result;
134 }
135 public bool Achieves(Link subgoal, Plan plan)
136 {
137     foreach (Type effect in alterAddConditionTypes)
138     {
139         if (effect.Equals(subgoal.Condition.GetType()))
140         {
141             return ActionAchieves(subgoal, plan);
142         }
143     }
144     return false;
145 }
146
147 public bool IsApplicable(Condition goal)
148 {
149     return addConditionTypes.Contains(goal.GetType());
150 }
151
152 protected virtual Variable newXVariable(Variable oldvar)
153 {
154     return oldvar;
155 }
156
157 protected virtual Variable newYVariable(Variable oldvar)
158 {
159     return oldvar;
160 }
161
162 public List<Type> GetPreconditions()
163 {
164     return preconditionsTypes;
165 }
166
167 public List<Type> GetAddList()
168 {
169     return addConditionTypes;
170 }
171
172 public List<Type> GetAlterAddList()
173 {
174     return alterAddConditionTypes;
175 }
176
177 public List<Type> GetAlterPreList()
178 {
179     return alterPreConditionTypes;
180 }
181
182 private int weight = 1;
183 public int Weight
```

```
184     {
185         get
186         {
187             return weight;
188         }
189
190         protected set
191         {
192             weight = value;
193         }
194     }
195
196     public override string ToString()
197     {
198         return this.GetType().Name + ":" + this.number;
199     }
200 }
201
202 public abstract class MoveAction : Action
203 {
204
205
206     private static int y;
207     private static int x;
208
209     public int deltaX = 0;
210     public int deltaY = 0;
211
212
213     public MoveAction(): base()
214     {
215         preconditionsTypes.Add(typeof(NotBoxAtCondition));
216         preconditionsTypes.Add(typeof(AtCondition));
217
218         addConditionTypes.Add(typeof(NotBoxAtCondition));
219         addConditionTypes.Add(typeof(AtCondition));
220
221         alterAddConditionTypes.Add(typeof(AtCondition));
222
223         alterPreConditionTypes.Add(typeof(AtCondition));
224     }
225
226     public static void SetMaxY(int value)
227     {
228         y = value;
229     }
230
231     public static int GetMaxY()
232     {
233         return y;
234     }
235
236     public static void SetMaxX(int value)
237     {
238         x = value;
```

```
239     }
240
241     public static int GetMaxX()
242     {
243         return x;
244     }
245
246     protected override Variable newXVariable(Variable oldvar)
247     {
248         return new Variable(oldvar.GetValue() + deltaX);
249     }
250
251     protected override Variable newYVariable(Variable oldvar)
252     {
253         return new Variable(oldvar.GetValue() + deltaY);
254     }
255 }
256
257 public class MoveSouthAction : MoveAction
258 {
259     public MoveSouthAction(): base()
260     {
261         deltaY = 1;
262     }
263
264     protected override bool ActionAchieves(Link subgoallink, Plan p
265         )
266     {
267         Condition subgoal = subgoallink.Condition;
268
269         if (subgoal is AtCondition)
270         {
271             return ((AtCondition)subgoal).GetIntegerY() < MoveAction.
                GetMaxY();
272         }
273         return false;
274     }
275
276
277     public override void Execute(NXTControl nxt)
278     {
279         nxt.Move(NXTControl.SOUTH);
280     }
281 }
282
283 public class MoveNorthAction : MoveAction
284 {
285     public MoveNorthAction() : base()
286     {
287         deltaY = -1;
288     }
289
290     protected override bool ActionAchieves(Link subgoallink, Plan p
        )
```

```
291     {
292         Condition subgoal = subgoallink.Condition;
293         if (subgoal is AtCondition)
294         {
295             return ((AtCondition)subgoal).GetIntegerY() > 0;
296         }
297         return false;
298     }
299
300     public override void Execute(NXTControl nxt)
301     {
302         nxt.Move(NXTControl.NORTH);
303     }
304 }
305
306 public class MoveWestAction : MoveAction
307 {
308     public MoveWestAction() : base()
309     {
310         deltaX = 1;
311     }
312
313     protected override bool ActionAchieves(Link subgoallink, Plan p
314         )
315     {
316         Condition subgoal = subgoallink.Condition;
317         if (subgoal is AtCondition)
318         {
319             return ((AtCondition)subgoal).GetIntegerX() < MoveAction.
320                 GetMaxX();
321         }
322         return false;
323     }
324
325     public override void Execute(NXTControl nxt)
326     {
327         nxt.Move(NXTControl.WEST);
328     }
329 }
330
331 public class MoveEastAction : MoveAction
332 {
333     public MoveEastAction() : base()
334     {
335         deltaX = -1;
336     }
337
338     protected override bool ActionAchieves(Link subgoallink, Plan p
339         )
340     {
341         Condition subgoal = subgoallink.Condition;
342         if (subgoal is AtCondition)
343         {
344             return ((AtCondition)subgoal).GetIntegerX() > 0;
```

```
343     }
344     return false;
345 }
346
347
348 public override void Execute(NXTControl nxt)
349 {
350     nxt.Move(NXTControl.EAST);
351 }
352 }
353
354 public class PickupAction : Action
355 {
356     public PickupAction() :base()
357     {
358         Weight = 3;
359
360         preconditionsTypes.Add(typeof(BoxAtCondition));
361         preconditionsTypes.Add(typeof(NotHaveBallCondition));
362         preconditionsTypes.Add(typeof(AtCondition));
363
364         addConditionTypes.Add(typeof(HaveBallCondition));
365         addConditionTypes.Add(typeof(NotBoxAtCondition));
366         addConditionTypes.Add(typeof(AtCondition));
367
368         alterAddConditionTypes.Add(typeof(HaveBallCondition));
369         alterAddConditionTypes.Add(typeof(NotBoxAtCondition));
370
371         alterPreConditionTypes.Add(typeof(BoxAtCondition));
372         alterPreConditionTypes.Add(typeof(NotHaveBallCondition));
373     }
374
375
376     public override void Execute(NXTControl nxt)
377     {
378         nxt.pickupBall();
379     }
380 }
381
382 public abstract class ReleaseAction : Action
383 {
384     public int deltaXneg = 0;
385     public int deltaYneg = 0;
386
387     public ReleaseAction() :base()
388     {
389         Weight = 3;
390
391         preconditionsTypes.Add(typeof(HaveBallCondition));
392         preconditionsTypes.Add(typeof(NotBoxAtCondition));
393         preconditionsTypes.Add(typeof(AtCondition));
394
395         addConditionTypes.Add(typeof(BoxAtCondition));
396         addConditionTypes.Add(typeof(NotHaveBallCondition));
397     }

```

```

398     addConditionTypes.Add(typeof(AtCondition));
399
400     alterAddConditionTypes.Add(typeof(BoxAtCondition));
401     alterAddConditionTypes.Add(typeof(NotHaveBallCondition));
402
403     alterPreConditionTypes.Add(typeof(HaveBallCondition));
404     alterPreConditionTypes.Add(typeof(NotBoxAtCondition));
405 }
406
407 public override void Execute(NXTControl nxt)
408 {
409     //nxt.Release();
410 }
411
412 protected override Variable newXVariable(Variable oldvar)
413 {
414     return new Variable(oldvar.GetValue() + deltaXneg);
415 }
416
417 protected override Variable newYVariable(Variable oldvar)
418 {
419     return new Variable(oldvar.GetValue() + deltaYneg);
420 }
421
422 public override List<Link> NewOpenPreconditions(List<Link>
    OpenPreconditions, CausalLink SubGoalLink, Plan p)
423 {
424     Condition SubGoalCondition = SubGoalLink.Condition;
425     MyList<Link> result = new MyList<Link>(OpenPreconditions);
426
427     Variable newVarX = new Variable();
428     Variable newVarY = new Variable();
429
430     Variable oldVarX = new Variable();
431     Variable oldVarY = new Variable();
432
433     foreach (Link l in OpenPreconditions)
434     {
435         Condition c = l.Condition;
436
437         /*
438          * For hver condition i OpenPreconditions, fjern alle som
439          * matcher typerne
440          * i addconditionType listen, hvis conditionen er ubundet
441          * eller
442          * hvis conditionen har samme koordinat (variabel) som
443          * SubGoalCondition
444          */
445         if (SubGoalLink.PostAction.Equals(l.ActionPost) &&
            addConditionTypes.Contains(c.GetType()))
446         {
447             if (!c.IsBothBound())
448             {
449                 result.Remove(l);
450             }
451         }
452     }
453 }

```

```

447         if (extraLinks1) p.CausalLinks.Add(new CausalLink(
448             SubGoalLink.PreAction, c, SubGoalLink.PostAction));
449     }
450     else if (SubGoalCondition.IsBothBound())
451     {
452         if (SubGoalCondition is ConditionVariable)
453         {
454             if (SubGoalCondition is BoxAtCondition && c is
455                 AtCondition && ((ConditionVariable)
456                     SubGoalCondition).IsSamePlace((ConditionVariable)
457                         c, deltaXneg, deltaYneg)
458                 || SubGoalCondition is AtCondition && c is
459                     BoxAtCondition && ((ConditionVariable)
460                         SubGoalCondition).IsSamePlace((
461                             ConditionVariable)c, -deltaXneg, -deltaYneg))
462             {
463                 result.Remove(1);
464                 if (extraLinks1) p.CausalLinks.Add(new CausalLink(
465                     SubGoalLink.PreAction, c, SubGoalLink.
466                         PostAction));
467             }
468         }
469         else if (c.GetType().Equals(SubGoalCondition.GetType()
470             ()) && ((ConditionVariable)SubGoalCondition).
471             IsSamePlace((ConditionVariable)c))
472         {
473             result.Remove(1);
474             if (extraLinks1) p.CausalLinks.Add(new CausalLink(
475                 SubGoalLink.PreAction, c, SubGoalLink.
476                     PostAction));
477         }
478     }
479 }
480 }
481 else if (!(SubGoalCondition is ConditionVariable))
482 {
483     if (c is AtCondition)
484     {
485         oldVarX = ((AtCondition)c).GetVariableX();
486         oldVarY = ((AtCondition)c).GetVariableY();
487
488         deltaXneg *= -1;
489         deltaYneg *= -1;
490         newVarX = newXVariable(((AtCondition)c).GetVariableX()
491             ());
492         newVarY = newYVariable(((AtCondition)c).GetVariableY()
493             ());
494         deltaXneg *= -1;
495         deltaYneg *= -1;
496     }
497     else //c is boxatcondition
498     {
499         newVarX = ((AtCondition)c).GetVariableX();
500         newVarY = ((AtCondition)c).GetVariableY();
501     }
502 }

```

```

486         oldVarX = newXVariable(((AtCondition)c).GetVariableX
487             ());
488         oldVarY = newYVariable(((AtCondition)c).GetVariableY
489             ());
490     }
491     result.Remove(1);
492     if (extraLinks1) p.CausalLinks.Add(new CausalLink(
493         SubGoalLink.PreAction, c, SubGoalLink.PostAction));
494 }
495 }
496 }
497
498 if (SubGoalCondition is ConditionVariable)
499 {
500     newVarX = newXVariable(((ConditionVariable)SubGoalCondition
501         ).GetVariableX());
502     newVarY = newYVariable(((ConditionVariable)SubGoalCondition
503         ).GetVariableY());
504
505     oldVarX = ((ConditionVariable)SubGoalCondition).
506         GetVariableX();
507     oldVarY = ((ConditionVariable)SubGoalCondition).
508         GetVariableY();
509 }
510
511 if (SubGoalCondition is BoxAtCondition)
512 {
513     result.Add(new Link(null, new HaveBallCondition(), this));
514     result.Add(new Link(null, new NotBoxAtCondition(oldVarX,
515         oldVarY), this));
516     result.Add(new Link(null, new NotBoxAtCondition(newVarX,
517         newVarY), this)); //
518     result.Add(new Link(null, new AtCondition(newVarX, newVarY)
519         , this));
520
521     p.previousActions.Add(new Link(this, new AtCondition(
522         newVarX, newVarY), null));
523     p.previousActions.Add(new Link(this, new
524         NotHaveBallCondition(), null));
525
526     if (extraLinks2)
527     {
528         p.CausalLinks.Add(new CausalLink(SubGoalLink.PreAction,
529             new AtCondition(newVarX, newVarY), SubGoalLink.
530                 PostAction));
531         p.CausalLinks.Add(new CausalLink(SubGoalLink.PreAction,
532             new NotHaveBallCondition(), SubGoalLink.PostAction));
533     }
534 }
535 }
536 else //subgoalcondition is NotHaveBallCondition
537 {
538     result.Add(new Link(null, new AtCondition(oldVarX, oldVarY)
539         , this));

```

```

525     result.Add(new Link(null, new HaveBallCondition(), this));
526     result.Add(new Link(null, new NotBoxAtCondition(newVarX,
527         newVarY), this));
528     result.Add(new Link(null, new NotBoxAtCondition(oldVarX,
529         oldVarY), this));//
530
531     p.previousActions.Add(new Link(this, new AtCondition(
532         oldVarX, oldVarY), null));
533     p.previousActions.Add(new Link(this, new BoxAtCondition(
534         newVarX, newVarY), null));
535
536     if (extraLinks2)
537     {
538         p.CausalLinks.Add(new CausalLink(SubGoalLink.PreAction,
539             new AtCondition(oldVarX, oldVarY), SubGoalLink.
540                 PostAction));
541         p.CausalLinks.Add(new CausalLink(SubGoalLink.PreAction,
542             new BoxAtCondition(newVarX, newVarY), SubGoalLink.
543                 PostAction));
544     }
545     }
546     return result;
547 }
548
549 protected override bool ActionAchieves(Link subgoallink, Plan p
550 )
551 {
552     if (subgoallink.Condition is ConditionVariable)
553     {
554         return TestInsideMap(((ConditionVariable)subgoallink.
555             Condition).GetVariableX(),
556             ((ConditionVariable)subgoallink.Condition).GetVariableY()
557             );
558     }
559
560     Action a = this;
561     foreach (Link l in p.OpenPreconditions)
562     {
563         Condition c = l.Condition;
564
565         if (this.Equals(l.ActionPost)
566             && addConditionTypes.Contains(c.GetType()))
567         {
568             if (c is AtCondition)
569             {
570                 deltaXneg *= -1;
571                 deltaYneg *= -1;
572                 Variable newVarX = newXVariable(((AtCondition)c).
573                     GetVariableX());
574                 Variable newVarY = newYVariable(((AtCondition)c).
575                     GetVariableY());
576                 deltaXneg *= -1;
577                 deltaYneg *= -1;
578             }
579
580             return TestInsideMap(newVarX, newVarY);
581         }
582     }
583 }

```

```
567     }
568     else if (c is BoxAtCondition)
569     {
570         return TestInsideMap(((AtCondition)c).GetVariableX(),
571                               ((AtCondition)c).GetVariableY());
572     }
573 }
574 return true;
575 }
576 protected abstract bool TestInsideMap(Variable x, Variable y);
577
578 }
579
580 public class ReleaseNorthAction : ReleaseAction
581 {
582
583     public ReleaseNorthAction(): base()
584     {
585         deltaYneg = -1;
586     }
587
588     protected override bool TestInsideMap(Variable x, Variable y)
589     {
590         if (y.IsBound())
591         {
592             return y.GetValue() > 0;
593         }
594         return true;
595     }
596
597     public override void Execute(NXTControl nxt)
598     {
599         nxt.Release(NXTControl.NORTH);
600     }
601
602 }
603
604
605 public class ReleaseSouthAction : ReleaseAction
606 {
607     public ReleaseSouthAction()
608     : base()
609     {
610         deltaYneg = 1;
611     }
612
613     protected override bool TestInsideMap(Variable x, Variable y)
614     {
615         if (y.IsBound())
616         {
617             return y.GetValue() < MoveAction.GetMaxY();
618         }
619         return true;
620     }
621 }
```

```
621
622     public override void Execute(NXTControl nxt)
623     {
624         nxt.Release(NXTControl.SOUTH);
625     }
626 }
627
628 public class ReleaseEastAction : ReleaseAction
629 {
630     public ReleaseEastAction()
631         : base()
632     {
633         deltaXneg = -1;
634     }
635
636     protected override bool TestInsideMap(Variable x, Variable y)
637     {
638         if (x.IsBound())
639         {
640             return x.GetValue() > 0;
641         }
642         return true;
643     }
644
645     public override void Execute(NXTControl nxt)
646     {
647         nxt.Release(NXTControl.EAST);
648     }
649 }
650
651 public class ReleaseWestAction : ReleaseAction
652 {
653     public ReleaseWestAction()
654         : base()
655     {
656         deltaXneg = 1;
657     }
658
659     protected override bool TestInsideMap(Variable x, Variable y)
660     {
661         if (x.IsBound())
662         {
663             return x.GetValue() < MoveAction.GetMaxX();
664         }
665         return true;
666     }
667
668     public override void Execute(NXTControl nxt)
669     {
670         nxt.Release(NXTControl.WEST);
671     }
672 }
673
674
675
```

```

676
677 public class StartAction : Action
678 {
679     public List<Condition> PostConditions;
680     private NotBoxAtCondition[,] notboxatlist;
681
682
683     public StartAction(List<Condition> startCond)
684     {
685         notboxatlist = new NotBoxAtCondition[4, 4];
686         Weight = 0;
687
688         PostConditions = startCond;
689
690         for (int i = 0; i < notboxatlist.GetLength(0); i++)
691         {
692             for (int j = 0; j < notboxatlist.GetLength(1); j++)
693             {
694                 notboxatlist[i, j] = new NotBoxAtCondition(i, j);
695             }
696         }
697
698         alterAddConditionTypes.Add(typeof(NotBoxAtCondition));
699         foreach (Condition c in PostConditions)
700         {
701             alterAddConditionTypes.Add(c.GetType());
702
703             if(c is BoxAtCondition)
704             {
705                 notboxatlist [((BoxAtCondition)c).GetIntegerX(), ((
706                     BoxAtCondition)c).GetIntegerY()] = null;
707             }
708         }
709         foreach (Condition c in notboxatlist)
710         {
711             if (c != null)
712             {
713                 PostConditions.Add(c);
714             }
715         }
716
717     protected override bool ActionAchieves(Link subgoallink, Plan p
718         )
719     {
720         Condition subgoal = subgoallink.Condition;
721
722         if (subgoal is ConditionVariable && (!((ConditionVariable)
723             subgoal).GetVariableY().IsBound() || !((ConditionVariable)
724             subgoal).GetVariableX().IsBound()))
725         {
726             foreach (Condition startcondition in PostConditions)
727             {
728                 if (startcondition.GetType().Equals(subgoal.GetType()) )
729                     //ES !((ConditionVariable)subgoal).IsBound())

```

```

726     {
727         if (((ConditionVariable) subgoal).GetVariableX().
            IsBound() && ((ConditionVariable) startcondition).
            GetIntegerX().Equals(
728             ((ConditionVariable) subgoal).GetIntegerX()) && !((
                ConditionVariable) subgoal).GetVariableY().IsBound
                ())
729             ||
730             (((ConditionVariable) subgoal).GetVariableY().IsBound
                ()) && ((ConditionVariable) startcondition).
                GetIntegerY().Equals(
731             ((ConditionVariable) subgoal).GetIntegerY()) && !((
                ConditionVariable) subgoal).GetVariableX().IsBound
                ())
732             ||
733             (!((ConditionVariable) subgoal).GetVariableY().IsBound
                ()) && !((ConditionVariable) subgoal).GetVariableX
                ().IsBound())
734         )
735         {
736             return true;
737         }
738     }
739 }
740
741     return false;
742 }
743 else
744 {
745     return PostConditions.Contains(subgoal);
746 }
747 }
748
749 public override List<Link> NewOpenPreconditions(List<Link>
    OpenPreconditions, CausalLink SubGoalLink, Plan p)
750 {
751     List<Link> result = new MyList<Link>(OpenPreconditions);
752     Link l = OpenPreconditions[0];
753
754     result.RemoveAt(0);
755
756     if (extraLinks1) p.CausalLinks.Add(new CausalLink(SubGoalLink
        .PreAction, l.Condition, SubGoalLink.PostAction));
757
758     if (l.Condition is ConditionVariable && !((ConditionVariable)
        l.Condition).IsBothBound())
759     {
760         foreach (Condition c in PostConditions)
761         {
762             if (l.Condition.GetType().Equals(c.GetType()))
763             {
764                 if (!((ConditionVariable) l.Condition).GetVariableX().
                    IsBound() &&
765                 !((ConditionVariable) l.Condition).GetVariableY().
                    IsBound())

```

```

766         {
767             ((ConditionVariable)l.Condition).GetVariableX().Bind
768                 (((ConditionVariable)c).GetIntegerX());
769             ((ConditionVariable)l.Condition).GetVariableY().Bind
770                 (((ConditionVariable)c).GetIntegerY());
771             break;
772         }
773     else
774     {
775         if (!((ConditionVariable)l.Condition).GetVariableX().
776             IsBound()
777             && ((ConditionVariable)l.Condition).GetIntegerY().
778                 Equals(((ConditionVariable)c).GetIntegerY())
779             )
780             ((ConditionVariable)l.Condition).GetVariableX().
781                 Bind(((ConditionVariable)c).GetIntegerX());
782         else if (!((ConditionVariable)l.Condition).
783             GetVariableY().IsBound()
784             && ((ConditionVariable)l.Condition).GetIntegerX().
785                 Equals(((ConditionVariable)c).GetIntegerX())
786             )
787             ((ConditionVariable)l.Condition).GetVariableY().
788                 Bind(((ConditionVariable)c).GetIntegerY());
789         else
790             continue;
791         break;
792     }
793 }
794 }
795 }
796 return result;
797 }
798
799 public override void Execute(NXTControl nxt)
800 {
801 }
802 }
803
804 public class FinishAction : Action
805 {
806     private List<Condition> PreConditions;
807     public FinishAction(List<Condition> goalConds)
808     {
809         Weight = 0;
810
811         PreConditions = goalConds;
812         foreach (Condition c in goalConds)
813         {
814             preconditionsTypes.Add(c.GetType());
815         }
816     }
817
818     protected override bool ActionAchieves(Link subgoal, Plan p)
819     {
820         return true;
821     }
822 }

```

```
813     }
814
815     public override void Execute(NXTControl nxt)
816     {
817     }
818
819 }
820
821 }
```

## D.4 Conditions.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace MultiRoboticAgentsPlanning
6 {
7     public abstract class Condition
8     {
9         public override bool Equals(object c)
10        {
11            return this.GetHashCode() == c.GetHashCode();
12        }
13
14        public override int GetHashCode()
15        {
16            return this.ToString().GetHashCode();
17        }
18
19        public virtual bool Threatens(Condition c)
20        {
21            return false;
22        }
23
24        public override string ToString()
25        {
26            return this.GetType().Name;
27        }
28
29        public virtual bool IsBothBound()
30        {
31            return false;
32        }
33    }
34
35    public abstract class ConditionVariable : Condition
36    {
37        protected Variable xVar, yVar;
38
39        public override bool IsBothBound()
40        {
41            if (xVar.IsBound() && yVar.IsBound())
```

```
42     {
43         return true;
44     }
45     return false;
46 }
47
48 public void BindVariables(ConditionVariable c)
49 {
50     xVar.Bind(c.GetIntegerX());
51     yVar.Bind(c.GetIntegerY());
52     this.IsBothBound();
53 }
54
55 public void BindVariables(int x, int y)
56 {
57     xVar.Bind(x);
58     yVar.Bind(y);
59 }
60
61 public int GetIntegerX()
62 {
63     return xVar.GetValue();
64 }
65
66 public int GetIntegerY()
67 {
68     return yVar.GetValue();
69 }
70
71 public Variable GetVariableX()
72 {
73     return xVar;
74 }
75
76 public Variable GetVariableY()
77 {
78     return yVar;
79 }
80
81 public bool IsSamePlace(ConditionVariable c)
82 {
83     return IsSamePlace(c, 0, 0);
84 }
85
86 public bool IsSamePlace(ConditionVariable c, int deltaX, int
87     deltaY)
88 {
89     if (this.IsBothBound() && c.IsBothBound())
90     {
91         return this.GetIntegerX() == c.GetIntegerX()+deltaX && this
92             .GetIntegerY() == c.GetIntegerY()+deltaY;
93     }
94     return false;
95 }
```

```

95     public int DistanceTo(ConditionVariable c)
96     {
97         if (this.IsBothBound() && c.IsBothBound())
98         {
99             return Math.Abs(this.GetIntegerX() - c.GetIntegerX()) +
100                Math.Abs(this.GetIntegerY() - c.GetIntegerY());
101         }
102         return int.MaxValue;
103     }
104
105     public class AtCondition : ConditionVariable
106     {
107
108         public AtCondition(Variable x, Variable y)
109         {
110             this.xVar = x;
111             this.yVar = y;
112         }
113
114         public AtCondition(int x, int y)
115         {
116             xVar = new Variable(x);
117             yVar = new Variable(y);
118         }
119
120
121         public override bool Threatens(Condition c)
122         {
123             /* if (c.GetType().Equals(typeof(AtCondition)))
124             {
125                 AtCondition atc = (AtCondition)c;
126                 return !(atc.GetIntegerX() == this.GetIntegerX() && atc.
127                    GetIntegerY() == this.GetIntegerY());
128             }*/
129             return false;
130         }
131
132         public override String ToString()
133         {
134             return "At(" + (this.GetVariableX().IsBound() ? this.
135                GetIntegerX() + "," : "") + (this.GetVariableY().IsBound
136                () ? "" + this.GetIntegerY() : "") + ")";
137         }
138     }
139
140     public class HaveBallCondition : Condition
141     {
142
143         public override bool Threatens(Condition c)
144         {
145             return c.GetType().Equals(typeof(NotHaveBallCondition));
146         }
147
148         public override string ToString()

```

```
146     {
147         return "HaveBall";
148     }
149 }
150
151 public class NotHaveBallCondition : Condition
152 {
153
154     public override bool Threatens(Condition c)
155     {
156         return c.GetType().Equals(typeof(HaveBallCondition));
157     }
158
159     public override string ToString()
160     {
161         return "NotHaveBall";
162     }
163 }
164
165 public class BoxAtCondition : ConditionVariable
166 {
167     public BoxAtCondition(Variable x, Variable y)
168     {
169         this.xVar = x;
170         this.yVar = y;
171     }
172
173     public BoxAtCondition(int x, int y)
174     {
175         xVar = new Variable(x);
176         yVar = new Variable(y);
177     }
178
179     public override bool Threatens(Condition c)
180     {
181         if (c.GetType().Equals(typeof(NotBoxAtCondition)))
182         {
183             NotBoxAtCondition nboxc = (NotBoxAtCondition)c;
184             return (nboxc.GetIntegerX() == this.GetIntegerX() && nboxc.
185                 GetIntegerY() == this.GetIntegerY());
186         }
187         return false;
188     }
189
190     public override String ToString()
191     {
192         return "BoxAt(" + (this.GetVariableX().IsBound() ? this.
193             GetIntegerX() + ", " : "") + (this.GetVariableY().IsBound
194             () ? "" + this.GetIntegerY() : "") + ")";
195     }
196 }
197
198 public class NotBoxAtCondition : ConditionVariable
199 {
200     public NotBoxAtCondition(Variable x, Variable y)
```

```

198     {
199         this.xVar = x;
200         this.yVar = y;
201     }
202
203     public NotBoxAtCondition(int x, int y)
204     {
205         xVar = new Variable(x);
206         yVar = new Variable(y);
207     }
208
209     public override bool Threatens(Condition c)
210     {
211         if (c.GetType().Equals(typeof(BoxAtCondition)))
212         {
213             BoxAtCondition boxc = (BoxAtCondition)c;
214             return (boxc.GetIntegerX() == this.GetIntegerX() && boxc.
                GetIntegerY() == this.GetIntegerY());
215         }
216         return false;
217     }
218
219     public override String ToString()
220     {
221         return "NotBoxAt(" + (this.GetVariableX().IsBound() ? this.
                GetIntegerX() + "," : "") + (this.GetVariableY().IsBound
                () ? "" + this.GetIntegerY() : "") + ")";
222     }
223 }
224 }

```

## D.5 Variable.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace MultiRoboticAgentsPlanning
6  {
7      public class Variable
8      {
9          int value;
10         bool isset = false;
11
12         public Variable(int x)
13         {
14             isset = true;
15             value = x;
16         }
17         public Variable()
18         {
19         }
20     }

```

```
21     public void Bind(int x)
22     {
23         isset = true;
24         value = x;
25     }
26
27     public int GetValue()
28     {
29         return value;
30     }
31     public bool IsBound()
32     {
33         return isset;
34     }
35
36
37
38 }
39 }
```

## D.6 Program.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Windows.Forms;
4  using NXTRemoteControl;
5  using BenTools.Data;
6  using System.Threading;
7
8  namespace MultiRoboticAgentsPlanning
9  {
10     static class Program
11     {
12         [STAThread]
13         static void Main()
14         {
15             //new PutOnShoes();
16
17             Application.Run(new GraphViewer());
18             return;
19             Console.WriteLine("Hello_World!");
20             Console.WriteLine();
21             DateTime dt = System.DateTime.Now;
22
23             //Tests.popLEGOtest();
24             //Tests.WorldAgentTest();
25             //Tests.BlockWorldtest();
26             //PlanTest pt = new PlanTest();
27             //pt.Init();
28             //pt.MoveThroughBallTest();
29             //pt.BoxAtTest1();
30
31             Console.WriteLine(System.DateTime.Now.Subtract(dt));
```

```
32
33     Console.WriteLine("Done!");
34     Console.ReadKey();
35 }
36 }
37
38 public static class Tests
39 {
40
41     public static void popLEGOtest()
42     {
43         MoveAction.SetMaxX(3);
44         MoveAction.SetMaxY(3);
45
46         List<Action> actions = new MyList<Action>();
47         actions.Add(new MoveWestAction());
48         actions.Add(new MoveSouthAction());
49         actions.Add(new MoveEastAction());
50         actions.Add(new MoveNorthAction());
51         actions.Add(new ReleaseEastAction());
52         actions.Add(new ReleaseNorthAction());
53         actions.Add(new ReleaseSouthAction());
54         actions.Add(new ReleaseWestAction());
55
56
57
58         actions.Add(new PickupAction());
59
60         //boxatTest1star(actions);
61
62         boxatTest1(actions);
63         //boxatTest2(actions);
64         //haveballTest(actions);
65         //atTest(actions);
66         //nothaveballTest(actions);
67         //movethroughballTest(actions);
68
69         //heuristicTest(actions);
70
71
72         //multiTest doesn't actually make sense if ordering is
73         //enabled
74         //multiTest(actions);
75     }
76
77     public static void WorldAgentTest()
78     {
79         MoveAction.SetMaxX(3);
80         MoveAction.SetMaxY(3);
81
82         List<Action> actions = new MyList<Action>();
83         actions.Add(new MoveWestAction());
84         actions.Add(new MoveSouthAction());
85         actions.Add(new MoveEastAction());
86         actions.Add(new MoveNorthAction());
```

```
86     actions.Add(new ReleaseEastAction());
87     actions.Add(new ReleaseNorthAction());
88     actions.Add(new ReleaseSouthAction());
89     actions.Add(new ReleaseWestAction());
90     actions.Add(new PickupAction());
91
92     Dictionary<Agent, List<Condition>> agents = new Dictionary<
93         Agent, List<Condition>>();
94     List<Condition> agentstartconditions = new List<Condition>();
95
96     NXTControl nxt = new NXTControl("COM39");
97     Agent a39 = new Agent(actions, nxt);
98     //nxt.pickupBall();
99     agentstartconditions.Add(new AtCondition(3, 3));
100    agentstartconditions.Add(new NotHaveBallCondition());
101    agents.Add(a39, agentstartconditions);
102    agentstartconditions = new List<Condition>();
103
104    nxt = new NXTControl("COM43");
105    Agent a43 = new Agent(actions, nxt);
106    agentstartconditions.Add(new AtCondition(3, 0));
107    agentstartconditions.Add(new NotHaveBallCondition());
108    agents.Add(a43, agentstartconditions);
109    agentstartconditions = new List<Condition>();
110
111
112    nxt = new NXTControl("COM42");
113    Agent a42 = new Agent(actions, nxt);
114    agentstartconditions.Add(new AtCondition(0, 2));
115    agentstartconditions.Add(new NotHaveBallCondition());
116    agents.Add(a42, agentstartconditions);
117    agentstartconditions = new List<Condition>();
118
119    List<Condition> startboxatconditions = new List<Condition>();
120    //startboxatconditions.Add(new BoxAtCondition(2, 2));
121    //startboxatconditions.Add(new BoxAtCondition(2, 0));
122    //startboxatconditions.Add(new BoxAtCondition(0, 1));
123    //startboxatconditions.Add(new BoxAtCondition(2, 1));
124
125
126    World world = new World(startboxatconditions, agents);
127
128    List<Condition> agentgoalconditions = new List<Condition>();
129
130    agentgoalconditions.Add(new AtCondition(3, 3));
131    world.AddJob(a43, new List<Condition>(agentgoalconditions));
132    agentgoalconditions.Clear();
133
134
135    agentgoalconditions.Add(new AtCondition(3, 0));
136    //agentgoalconditions.Add(new AtCondition(0, 0));
137    world.AddJob(a39, new List<Condition>(agentgoalconditions));
138    agentgoalconditions.Clear();
139
```

```
140     agentgoalconditions.Add(new AtCondition(3, 1));
141     world.AddJob(a42, new List<Condition>(agentgoalconditions));
142     agentgoalconditions.Clear();
143
144     //Thread T1
145     Thread t1 = new Thread(a39.BDILoop);
146     t1.Start();
147
148     //Thread T2
149     Thread t2 = new Thread(a43.BDILoop);
150     t2.Start();
151
152     //Thread T3
153     Thread t3 = new Thread(a42.BDILoop);
154     t3.Start();
155 }
156
157
158 private static void boxatTest1(List<Action> actions)
159 {
160
161     List<Condition> startstate = new MyList<Condition>();
162
163     startstate.Add(new AtCondition(0, 1));
164     startstate.Add(new BoxAtCondition(0, 1));
165     startstate.Add(new NotHaveBallCondition());
166     //startstate.Add(new NotHaveBallCondition());
167
168     List<Condition> goalstate = new MyList<Condition>();
169
170     goalstate.Add(new BoxAtCondition(0, 0));
171     goalstate.Add(new AtCondition(new Variable(), new Variable())
172 );
173     //goalstate.Add(new AtCondition(0, 0));
174     POPstar pop = new POPstar(startstate, goalstate, actions);
175     pop.Start();
176 }
177
178 private static void boxatTest1star(List<Action> actions)
179 {
180
181     List<Condition> startstate = new MyList<Condition>();
182
183     startstate.Add(new AtCondition(3, 3));
184     startstate.Add(new NotHaveBallCondition());
185
186     List<Condition> goalstate = new MyList<Condition>();
187
188     goalstate.Add(new AtCondition(0, 0));
189     goalstate.Add(new NotHaveBallCondition());
190
191     new POPstar(startstate, goalstate, actions);
192 }
193
```

```
194 private static void boxatTest2(List<Action> actions)
195 {
196
197     List<Condition> startstate = new MyList<Condition>();
198
199     startstate.Add(new AtCondition(0, 2));
200     startstate.Add(new HaveBallCondition());
201
202
203     List<Condition> goalstate = new MyList<Condition>();
204
205     goalstate.Add(new BoxAtCondition(2, 3));
206     //goalstate.Add(new AtCondition(0, 0));
207
208     POPstar pop = new POPstar(startstate, goalstate, actions);
209 }
210
211
212 private static void haveballTest(List<Action> actions)
213 {
214
215     List<Condition> startstate = new MyList<Condition>();
216
217     startstate.Add(new AtCondition(3, 2));
218     //startstate.Add(new BoxAtCondition(1, 0));
219     startstate.Add(new NotHaveBallCondition());
220
221     startstate.Add(new BoxAtCondition(0, 0));
222     //startstate.Add(new NotHaveBallCondition());
223
224
225     List<Condition> goalstate = new MyList<Condition>();
226
227     goalstate.Add(new HaveBallCondition());
228     //goalstate.Add(new BoxAtCondition(0, 0));
229     goalstate.Add(new AtCondition(new Variable(), new Variable())
230         );
231     //goalstate.Add(new AtCondition(0, 0));
232     POP pop = new POP(startstate, goalstate, actions);
233 }
234
235 private static void atTest(List<Action> actions)
236 {
237
238     List<Condition> startstate = new MyList<Condition>();
239
240     startstate.Add(new AtCondition(3, 3));
241     //startstate.Add(new BoxAtCondition(1, 0));
242     //startstate.Add(new NotHaveBallCondition());
243
244     List<Condition> goalstate = new MyList<Condition>();
245
246     //goalstate.Add(new HaveBallCondition());
247     //goalstate.Add(new BoxAtCondition(0, 0));
248     goalstate.Add(new AtCondition(0, 0));
```

```
248     //goalstate.Add(new AtCondition(0, 0));
249     new POPstar(startstate, goalstate, actions);
250 }
251
252 private static void nothaveballTest(List<Action> actions)
253 {
254     List<Condition> startstate = new MyList<Condition>();
255
256     startstate.Add(new AtCondition(1, 0));
257     //startstate.Add(new BoxAtCondition(1, 0));
258     startstate.Add(new HaveBallCondition());
259
260     List<Condition> goalstate = new MyList<Condition>();
261
262     goalstate.Add(new NotHaveBallCondition());
263     goalstate.Add(new AtCondition(0, 0));
264
265     //goalstate.Add(new BoxAtCondition(0, 0));
266
267     //goalstate.Add(new AtCondition(0, 0));
268     POP pop = new POP(startstate, goalstate, actions);
269 }
270
271 private static void multiTest(List<Action> actions)
272 {
273     List<Condition> startstate = new MyList<Condition>();
274
275     startstate.Add(new AtCondition(3, 0));
276     startstate.Add(new NotHaveBallCondition());
277     startstate.Add(new BoxAtCondition(3, 0));
278     startstate.Add(new BoxAtCondition(1,0));
279
280     List<Condition> goalstate = new MyList<Condition>();
281
282     goalstate.Add(new BoxAtCondition(2, 0));
283     goalstate.Add(new AtCondition(1, 0));
284     goalstate.Add(new HaveBallCondition());
285
286     POP pop = new POP(startstate, goalstate, actions);
287 }
288
289 private static void movethroughballTest(List<Action> actions)
290 {
291     MoveAction.SetMaxY(0);
292
293     List<Condition> startstate = new MyList<Condition>();
294
295     startstate.Add(new AtCondition(3, 0));
296     startstate.Add(new BoxAtCondition(1, 0));
```

```

303     startstate.Add(new NotHaveBallCondition());
304
305     List<Condition> goalstate = new MyList<Condition>();
306
307     //goalstate.Add(new HaveBallCondition());
308     //goalstate.Add(new BoxAtCondition(0, 0));
309     goalstate.Add(new AtCondition(0, 0));
310     //goalstate.Add(new AtCondition(0, 0));
311     POP pop = new POP(startstate, goalstate, actions);
312 }
313
314 private static void heuristicTest(List<Action> actions)
315 {
316     List<Condition> startstate = new List<Condition>();
317     List<Condition> goalstate = new List<Condition>();
318
319     startstate.Add(new AtCondition(1, 1));
320     goalstate.Add(new AtCondition(0, 0));
321
322     new POP(startstate, goalstate, actions);
323 }
324
325 public static void BlockWorldtest()
326 {
327
328     List<Action> actions = new MyList<Action>();
329     actions.Add(new PickupTableAction());
330     actions.Add(new PickupBlockAction());
331     actions.Add(new PutdownAction());
332     actions.Add(new PutdownTableAction());
333
334
335     blockworldHolding(actions);
336     //blockworldReverse(actions);
337 }
338
339 private static void blockworldHolding(List<Action> actions)
340 {
341
342     List<Condition> startstate = new MyList<Condition>();
343
344     startstate.Add(new OnTableCondition(1, 0));
345     //startstate.Add(new OnTableCondition(3, 0));
346     startstate.Add(new EmptyHandCondition());
347
348     startstate.Add(new OnBoxCondition(2, 1));
349     startstate.Add(new OnBoxCondition(3, 2));
350     startstate.Add(new OnBoxCondition(4, 3));
351     startstate.Add(new OnBoxCondition(5, 4));
352     //startstate.Add(new ClearBoxCondition(1, 0));
353     startstate.Add(new ClearBoxCondition(5, 0));
354
355     List<Condition> goalstate = new MyList<Condition>();
356
357     goalstate.Add(new HoldingCondition(1,0));

```

```

358
359     Console.WriteLine("Plan:");
360     POPstar pop = new POPstar(startstate, goalstate, actions);
361     pop.Start();
362     Plan plan = pop.BestPlan;
363
364     foreach (Action action in plan.GetOrdersTopologicalSort())
365     {
366         Console.WriteLine("___" + action);
367     }
368 }
369
370 private static void blockworldReverse(List<Action> actions)
371 {
372
373     List<Condition> startstate = new MyList<Condition>();
374
375     startstate.Add(new OnTableCondition(1, 0));
376     //startstate.Add(new OnTableCondition(3, 0));
377     startstate.Add(new EmptyHandCondition());
378
379     startstate.Add(new OnBoxCondition(2, 1));
380     //startstate.Add(new OnBoxCondition(3, 2));
381     startstate.Add(new ClearBoxCondition(2, 0));
382
383     List<Condition> goalstate = new MyList<Condition>();
384
385     //goalstate.Add(new HoldingCondition(1,0));
386     goalstate.Add(new OnBoxCondition(1, 2));
387     //goalstate.Add(new OnTableCondition(2, 0));
388
389     Console.WriteLine("Plan:");
390     POPstar pop = new POPstar(startstate, goalstate, actions);
391     pop.Start();
392     Plan plan = pop.BestPlan;
393
394     foreach (Action action in plan.GetOrdersTopologicalSort())
395     {
396         Console.WriteLine("___" + action);
397     }
398 }
399 }
400 }

```

## D.7 NXTControl.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4 using NXTRemoteControl;
5
6 namespace MultiRoboticAgentsPlanning
7 {

```

```
8 public class NXTControl
9 {
10     private NXT nxt;
11     private Agent agent;
12     private World world;
13
14     private bool HaveBall = false;
15
16     private int CurrentDirection = NORTH;
17
18     public const int NORTH = 0;
19     public const int EAST = 1;
20     public const int SOUTH = 2;
21     public const int WEST = 3;
22
23
24
25     public NXTControl(String comport)
26     {
27         nxt = new NXT(comport);
28     }
29
30     public NXTControl(String comport, Agent agent, World world)
31     {
32         nxt = new NXT(comport);
33         this.agent = agent;
34         this.world = world;
35     }
36
37     public void Close()
38     {
39         nxt.close();
40     }
41
42     public void Move(int NewDirection)
43     {
44         Turn(NewDirection);
45         if (!HaveBall)
46         {
47             nxtDoAndWaitForTermination("followLine");
48         }
49         else
50         {
51             nxtDoAndWaitForTermination("followLineWithBall");
52         }
53     }
54
55     public void Turn(int NewDirection)
56     {
57         switch (CurrentDirection)
58         {
59             case NORTH:
60                 switch (NewDirection)
61                 {
62                     case NORTH:
```

```
63         break;
64     case SOUTH:
65         turnAround();
66         break;
67     case EAST:
68         turnRight();
69         break;
70     case WEST:
71         turnLeft();
72         break;
73
74     default: break;
75 }
76 break;
77 case SOUTH:
78     switch (NewDirection)
79     {
80     case NORTH:
81         turnAround();
82         break;
83     case SOUTH:
84         break;
85     case EAST:
86         turnLeft();
87         break;
88     case WEST:
89         turnRight();
90         break;
91
92     default: break;
93 }
94 break;
95 case EAST:
96     switch (NewDirection)
97     {
98     case NORTH:
99         turnLeft();
100        break;
101    case SOUTH:
102        turnRight();
103        break;
104    case EAST:
105        break;
106    case WEST:
107        turnAround();
108        break;
109
110    default: break;
111 }
112 break;
113 case WEST:
114     switch (NewDirection)
115     {
116     case NORTH:
117         turnRight();
```

```
118         break;
119     case SOUTH:
120         turnLeft();
121         break;
122     case EAST:
123         turnAround();
124         break;
125     case WEST:
126         break;
127
128         default: break;
129     }
130     break;
131
132     default: break;
133 }
134 CurrentDirection = NewDirection;
135 }
136
137 private void turnRight()
138 {
139     nxtDoAndWaitForTermination("right");
140 }
141 private void turnLeft()
142 {
143     nxtDoAndWaitForTermination("left");
144 }
145 private void turnAround()
146 {
147     nxtDoAndWaitForTermination("turnAround");
148 }
149
150 public void Release(int NewDirection)
151 {
152     Turn(NewDirection);
153     HaveBall = false;
154
155     nxtDoAndWaitForTermination("releaseMoveback");
156
157     switch (CurrentDirection)
158     {
159     case NORTH:
160         CurrentDirection = SOUTH;
161         break;
162     case SOUTH:
163         CurrentDirection = NORTH;
164         break;
165     case EAST:
166         CurrentDirection = WEST;
167         break;
168     case WEST:
169         CurrentDirection = EAST;
170         break;
171     }
172 }
```

```

173
174     public void pickupBall()
175     {
176         HaveBall = true;
177         nxtDoAndWaitForTermination("grab");
178     }
179
180     private String nxtDoAndWaitForTermination(String p)
181     {
182         nxt.Bluetooth.sendMessage(p, 0);
183         String received = nxt.Bluetooth.readMessage(11, 19, true);
184         while (received.Equals("0"))
185         {
186             received = nxt.Bluetooth.readMessage(11, 19, true);
187         }
188         return received;
189     }
190
191     private void nxtExecute(String p)
192     {
193         nxt.Bluetooth.sendMessage(p, 0);
194     }
195
196     private String getNextState()
197     {
198         return nxt.Bluetooth.readMessage(11, 19, true);
199     }
200
201     public void WarnDeadlock()
202     {
203         int wait = 500;
204         int tone = 1200;
205         nxt.Sound.playTone(tone, wait);
206         nxt.Sensor4.setTypeAndMode(NXTRemoteControl.Communication.
                Protocol.LIGHT_ACTIVE, NXTRemoteControl.Communication.
                Protocol.PCTFULLSCALEMODE);
207         System.Threading.Thread.Sleep(wait);
208         nxt.Sensor4.setTypeAndMode(NXTRemoteControl.Communication.
                Protocol.LIGHT_INACTIVE, NXTRemoteControl.Communication.
                Protocol.PCTFULLSCALEMODE);
209         System.Threading.Thread.Sleep(wait);
210     }
211 }
212 }
213 }

```

## D.8 Agent.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Threading;
5

```

```

6 namespace MultiRoboticAgentsPlanning
7 {
8
9     public class Agent
10    {
11        private World world;
12        public NXTControl NXT;
13        private List<Action> actions;
14
15        protected bool continueExecuting = true;
16        private static object locker = new object();
17        private Semaphore beginExecuting = new Semaphore(1, 1);
18
19        private static int agentCount = 0;
20        private int agentNumber;
21
22        public Agent(List<Action> actions , NXTControl NXT)
23        {
24            this.NXT = NXT;
25            this.actions = actions;
26            agentNumber = agentCount++;
27        }
28
29        public void SetWorld(World world)
30        {
31            this.world = world;
32        }
33
34        public void BDILoop()
35        {
36
37            List<Condition> Believes = null; // Initial beliefs
38            List<List<Condition>> Desires = null; // Initial desires
39            List<Condition> Intentions = null; // Initial intentions
40
41            while(true)
42            {
43                Desires = world.Options(this);
44                Believes = world.BRF(this);
45                Intentions = world.Filter(this);
46
47                Plan Plan = MakePlan(Believes , Intentions , actions);
48                if (Plan == null) continue;
49                LinkedList<Action> PlanSteps = Plan.
                    GetOrdersTopologicalSort();
50
51                //while not (empty(plan) or succeeded(plan) or impossible(
                    plan))
52                while (Plan != null && PlanSteps.Count > 0)
53                {
54                    Action action = PlanSteps.First.Value;
55                    PlanSteps.RemoveFirst();
56
57                    /*if reconsider(I,B)
58                    {

```

```

59         D := options(B,I);
60         I := filter(B,D,I);
61     }*/
62     if (world.PlanNotSound(Plan)) //not sound(plan)
63     {
64         Believes = world.BRF(this);
65         Plan = MakePlan(Believes, Intentions, actions);
66         if (Plan == null) break;
67         PlanSteps = Plan.GetOrdersTopologicalSort();
68         continue;
69     }
70
71     //Execute
72     world.ExeCuteAction(action, this, Plan, NXT);
73 }
74 }
75 }
76
77 private Plan MakePlan(List<Condition> Believes, List<Condition>
78     Intention, List<Action> actions)
79 {
80     POPstar pop = new POPstar(Believes, Intention, actions);
81     pop.Start();
82     pop.BestPlan.CalcEssentialStartConditions();
83     return pop.BestPlan;
84 }
85
86 public override String ToString()
87 {
88     return "Agent:_" + agentNumber;
89 }
90 }

```

## D.9 World.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Threading;
5
6  namespace MultiRoboticAgentsPlanning
7  {
8      public class World
9      {
10         private Dictionary<Agent, List<Condition>> agentsConditions;
11         private object WorldConditionLocker = new object();
12         private Dictionary<Agent, List<List<Condition>>> Jobs;
13         private Agent NonSpecifiedAgent = new Agent(new List<Action>(),
14             new NXTControl(""));
15
16         public World(List<Condition> worldstartconditions, Dictionary<
17             Agent, List<Condition>> agentsConditions)

```

```

16     {
17         WorldConditions = worldstartconditions;
18         this.agentsConditions = agentsConditions;
19
20
21
22         Jobs = new Dictionary<Agent, List<List<Condition>>>();
23         Jobs.Add(NonSpecifiedAgent, new List<List<Condition>>());
24
25         //initialize resources in resource-allocation graph
26         for (int i = 0; i < MoveAction.GetMaxX()+1; i++)
27         {
28             for (int j = 0; j < MoveAction.GetMaxY()+1; j++)
29             {
30                 RAG.Add("At("+i+", "+j+ ")", new List<string>());
31             }
32         }
33     }
34
35     foreach (Agent agent in this.agentsConditions.Keys)
36     {
37         agent.SetWorld(this);
38         //initialize agents in resource-allocation graph and wait-
39         for graph
40         RAG.Add(agent.ToString(), new List<String>());
41         WaitForGraph.Add(agent.ToString(), new List<String>());
42
43         //initialise agent locks
44         AgentLocks.Add(agent.ToString(), new Semaphore(0, 1));
45
46         //Reserve ressource at start position
47         foreach (Condition c in this.agentsConditions[agent])
48         {
49             if (c is AtCondition)
50             {
51                 System.Threading.Monitor.Enter(RAG);
52                 RAG[c.ToString()].Add(agent.ToString());
53                 System.Threading.Monitor.Exit(RAG);
54                 break;
55             }
56         }
57
58
59         NotBoxAtCondition[,] notboxatlist = new NotBoxAtCondition[
60             MoveAction.GetMaxX()+1, MoveAction.GetMaxY()+1];
61
62         for (int i = 0; i < notboxatlist.GetLength(0); i++)
63         {
64             for (int j = 0; j < notboxatlist.GetLength(1); j++)
65             {
66                 notboxatlist[i, j] = new NotBoxAtCondition(i, j);
67             }
68         }

```

```

69     foreach (Condition c in WorldConditions)
70     {
71         if(c is BoxAtCondition)
72         {
73             notboxatlist [((BoxAtCondition)c).GetIntegerX(), ((
74                 BoxAtCondition)c).GetIntegerY()] = null;
75         }
76     }
77     foreach (Condition c in notboxatlist)
78     {
79         if (c != null)
80         {
81             WorldConditions.Add(c);
82         }
83     }
84
85     //Thread DeadLockMonitor
86     Thread t = new Thread(DeadLockMonitor);
87     t.Start();
88 }
89
90 private Dictionary<String, bool> visited;
91 private Dictionary<String, bool> greys;
92 private Dictionary<String, bool> deadlockedAgents;
93 private void DeadLockMonitor()
94 {
95     deadlockedAgents = new Dictionary<string, bool>();
96     while (true)
97     {
98         //sleep 15 sec before chekking for a cycle
99         System.Threading.Thread.Sleep(3000);
100
101         //Depth-First search for a cycle in wait-for graph
102         visited = new Dictionary<String, bool>();
103         greys = new Dictionary<String, bool>();
104
105         bool deadlock = false;
106         foreach (String U in WaitForGraph.Keys)
107         {
108             if (!visited.ContainsKey(U)) //color[U] = WHITE
109             {
110                 if (DfsVisit(U)) //Backedge found
111                 {
112                     deadlock = true;
113                     //Console.ReadKey();
114                 }
115             }
116         }
117         if (deadlock)
118         {
119             HandleDeadlock();
120         }
121     }
122 }

```

```
123
124 private bool DfsVisit(string u)
125 {
126     visited.Add(u, true);
127     greys.Add(u, true);
128     bool dl = false;
129
130     try
131     {
132         foreach (String v in WaitForGraph[u])
133         {
134             if (greys.ContainsKey(v))
135             {
136                 //graph contains cycle
137                 try
138                 {
139                     deadlockedAgents.Add(u, true);
140                     deadlockedAgents.Add(v, true);
141                 }
142                 catch (ArgumentException)
143                 {
144                 }
145                 dl = true;
146             }
147             else if (!visited.ContainsKey(v))
148             {
149                 if (!dl)
150                     dl = DfsVisit(v);
151             }
152         }
153     }
154     catch (KeyNotFoundException)
155     {
156     }
157     if (dl)
158     {
159         return true;
160     }
161     greys.Remove(u);
162     return false;
163 }
164
165
166 private void HandleDeadlock()
167 {
168     foreach (Agent agent in agentsConditions.Keys)
169     {
170         bool temp = false;
171         temp = deadlockedAgents.ContainsKey(agent.ToString());
172         /*temp = greys.ContainsKey(agent.ToString());
173
174         if (!temp)
175             foreach (String s in WaitForGraph[agent.ToString()])
176             {
177                 if (greys.ContainsKey(s))
```

```

178         {
179             temp = true;
180             break;
181         }
182     }*/
183     if (temp)
184     {
185         agent.NXT.WarnDeadlock();
186         System.Threading.Thread.Sleep(200);
187     }
188 }
189 Console.WriteLine("
    *****
    ");
190 Console.WriteLine("WARNING");
191 Console.WriteLine("DeadLock!");
192 Console.WriteLine("Kernel_Panic!");
193 Console.WriteLine("
    *****
    ");
194 deadlockedAgents = new Dictionary<string, bool>();
195 }
196
197 internal List<Condition> BRF(Agent agent)
198 {
199     List<Condition> startstate = new List<Condition>(
200         WorldConditions);
201     startstate.AddRange(agentsConditions[agent]);
202     return startstate;
203 }
204
205 internal void AddJob(List<Condition> job)
206 {
207     //No special agent required
208     System.Threading.Monitor.Enter(Jobs);
209     Jobs[NonSpecifiedAgent].Add(job);
210     System.Threading.Monitor.Exit(Jobs);
211 }
212 internal void AddJob(Agent agent, List<Condition> job)
213 {
214     //Specific agent should do the job
215     System.Threading.Monitor.Enter(Jobs);
216
217     if (Jobs.ContainsKey(agent))
218     {
219         Jobs[agent].Add(job);
220     }
221     else
222     {
223         Jobs.Add(agent, new List<List<Condition>>());
224         Jobs[agent].Add(job);
225     }
226     System.Threading.Monitor.Exit(Jobs);
227 }
228 internal List<Condition> Filter(Agent agent)

```

```

228     {
229         System.Threading.Monitor.Enter(Jobs);
230         List<Condition> result = null;
231         if (Jobs.ContainsKey(agent))
232             {
233                 if (Jobs[agent].Count > 0)
234                     {
235                         result = Jobs[agent][0];
236                         Jobs[agent].RemoveAt(0);
237                     }
238             }
239         else
240             {
241                 if (Jobs[NonSpecifiedAgent].Count > 0)
242                     {
243                         result = Jobs[NonSpecifiedAgent][0];
244                         Jobs[NonSpecifiedAgent].RemoveAt(0);
245                     }
246             }
247         System.Threading.Monitor.Exit(Jobs);
248         return result;
249     }
250
251     internal bool PlanNotSound(Plan PopPlan)
252     {
253         //Decide if the plan assumptions in PopPlan still holds in
254         the real world
255
256         foreach (List<Condition> list in PopPlan.
257             EssentielStartConditions.Values)
258             {
259                 foreach (Condition condition in list)
260                     {
261                         if ((condition is BoxAtCondition || condition is
262                             NotBoxAtCondition) && !WorldConditions.Contains(
263                                 condition))
264                             {
265                                 return true;
266                             }
267                     }
268             }
269         return false;
270     }
271
272     public void UpdateWorld(Action action, Agent agent, Plan plan)
273     {
274         System.Threading.Monitor.Enter(WorldConditions);
275
276         plan.EssentielStartConditions.Remove(action);
277         if (action is MoveAction)
278             {
279                 foreach (Condition c in agentsConditions[agent])
280                     {
281                         if (c is AtCondition)
282                             {

```

```

279         ((AtCondition)c).BindVariables(((AtCondition)c).
280             GetIntegerX() - ((MoveAction)action).deltaX,
                ((AtCondition)c).GetIntegerY() - ((MoveAction)
                action).deltaY);
281     }
282     }
283 }
284 }
285 else if (action is PickupAction)
286 {
287     AtCondition atcondition = new AtCondition(new Variable(),
                new Variable());
288     foreach (Condition c in new List<Condition>(
                agentsConditions[agent]))
289     {
290         if (c is HaveBallCondition)
291         {
292             agentsConditions[agent].Remove(c);
293             agentsConditions[agent].Add(new NotHaveBallCondition());
294             ;
295         }
296         else if (c is AtCondition)
297         {
298             atcondition = (AtCondition)c;
299         }
300     }
301     foreach (Condition c in new List<Condition>(WorldConditions
                ))
302     {
303         if (c is BoxAtCondition && ((BoxAtCondition)c).
                IsSamePlace(atcondition))
304         {
305             WorldConditions.Remove(c);
306             WorldConditions.Add( new NotBoxAtCondition(atcondition.
                GetIntegerX(), atcondition.GetIntegerY()));
307             break;
308         }
309     }
310 }
311 else if (action is ReleaseAction)
312 {
313
314     AtCondition atcondition = new AtCondition(new Variable(),
                new Variable());
315     foreach (Condition c in new List<Condition>(
                agentsConditions[agent]))
316     {
317         if (c is NotHaveBallCondition)
318         {
319             agentsConditions[agent].Remove(c);
320             agentsConditions[agent].Add(new HaveBallCondition());
321         }
322         else if (c is AtCondition)
323         {

```

```

324         atcondition = (AtCondition)c;
325     }
326 }
327 AtCondition newAtCondtion = new AtCondition(atcondition.
    GetIntegerX() + ((ReleaseAction)action).deltaXneg*-1,
328     atcondition.GetIntegerY() + ((ReleaseAction)action).
        deltaYneg*-1);
329
330 foreach (Condition c in new List<Condition>(WorldConditions
    ))
331 {
332     if (c is NotBoxAtCondition && ((NotBoxAtCondition)c).
        IsSamePlace(newAtCondtion))
333     {
334         WorldConditions.Remove(c);
335         WorldConditions.Add(new BoxAtCondition(newAtCondtion.
            GetIntegerX(), newAtCondtion.GetIntegerY()));
336         break;
337     }
338 }
339 }
340 System.Threading.Monitor.Exit(WorldConditions);
341 }
342
343
344 internal void ExecuteAction(Action action, Agent agent, Plan
    plan, NXTControl NXT)
345 {
346     //Execute with NXT
347     if (action is FinishAction || action is StartAction)
348         return;
349
350     if (action is PickupAction)
351     {
352         //if action is pickup no new resources are required
353         UpdateWorld(action, agent, plan);
354         action.Execute(NXT);
355         Console.WriteLine(action.ToString());
356         return;
357     }
358
359     //Find current at position:
360     String currentAtCondition = null;
361     AtCondition newCondition = null;
362
363     foreach (Condition c in agentsConditions[agent])
364     {
365         if (c is AtCondition)
366         {
367             currentAtCondition = (String)((AtCondition)c).ToString().
                Clone();
368             if (action is MoveAction)
369             {
370                 newCondition = new AtCondition(((AtCondition)c).
                    GetIntegerX() - ((MoveAction)action).deltaX,

```

```

371         ((AtCondition)c).GetIntegerY() - ((MoveAction)action)
           .deltaY);
372     }
373     else if (action is ReleaseAction)
374     {
375         newCondition = new AtCondition(((AtCondition)c).
           GetIntegerX() + ((ReleaseAction)action).deltaXneg *
           -1,
376         ((AtCondition)c).GetIntegerY() + ((ReleaseAction)
           action).deltaYneg * -1);
377     }
378     break;
379 }
380 }
381 //request resource:
382 if (! RequestResource(agent, newCondition.ToString()))
383     //Lock Agent
384     AgentLocks[agent.ToString()].WaitOne();
385
386 //Execute action:
387 UpdateWorld(action, agent, plan);
388 Console.WriteLine(action.ToString());
389 action.Execute(NXT);
390
391 //Release resource:
392 if (action is MoveAction)
393     ReleaseResource(agent, currentAtCondition.ToString());
394 else if (action is ReleaseAction)
395     ReleaseResource(agent, newCondition.ToString());
396 }
397
398 internal List<List<Condition>> Options(Agent agent)
399 {
400     List<List<Condition>> result = new List<List<Condition>>();
401     if (Jobs.ContainsKey(agent))
402         result.AddRange(Jobs[agent]);
403     result.AddRange(Jobs[NonSpecifiedAgent]);
404
405     while (result.Count == 0)
406     { //wait for new jobs to arrive
407         System.Threading.Thread.Sleep(500);
408         if (Jobs.ContainsKey(agent))
409             result.AddRange(Jobs[agent]);
410         result.AddRange(Jobs[NonSpecifiedAgent]);
411     }
412     return result;
413 }
414
415 private bool RequestResource(Agent agent, String resource)
416 {
417     System.Threading.Monitor.Enter(RAG);
418     if (RAG[resource].Count == 0)
419     { //Resource not allocated
420         RAG[resource].Add(agent.ToString());
421         System.Threading.Monitor.Exit(RAG);

```

```

422     return true;
423 }
424 else
425 { //Resource allocated to other agent
426   RAG[agent.ToString()].Add(resource);
427   WaitForGraph[agent.ToString()].Add(RAG[resource][0]);
428   System.Threading.Monitor.Exit(RAG);
429   return false;
430 }
431 }
432
433 private void ReleaseResource(Agent agent, String resource)
434 {
435   String tempAction = null;
436
437   //Update Resource allocation graph
438   RAG[resource].Remove(agent.ToString());
439
440   //Update wait-for graph
441   foreach(String key in WaitForGraph.Keys)
442   {
443     if (WaitForGraph[key].Contains(agent.ToString()) && RAG[key]
444         .Contains(resource))
445     {
446       WaitForGraph[key].Remove(agent.ToString());
447       if (WaitForGraph[key].Count == 0)
448       {
449         if (tempAction != null)
450         {
451           WaitForGraph[key].Add(tempAction);
452         }
453         else
454         {
455           //if the Agent(key) waits for no one else it is
456             released
457           AgentLocks[key].Release();
458           tempAction = key;
459         }
460       }
461     }
462   }
463
464   private Dictionary<String, List<String>> rag = new Dictionary<
465     String, List<String>>();
466   public Dictionary<String, List<String>> RAG
467   {
468     get
469     {
470       lock (rag)
471       return rag;
472     }
473     set
474     {
475       lock (rag)

```

```
474         rag = value;
475     }
476 }
477
478 private Dictionary<String, List<String>> wfg = new Dictionary<
479     String, List<String>>();
480 private Dictionary<String, List<String>> WaitForGraph
481 {
482     get
483     {
484         lock (wfg)
485             return wfg;
486     }
487     set
488     {
489         lock (wfg)
490             wfg = value;
491     }
492 }
493 private Dictionary<String, Semaphore> al = new Dictionary<
494     string, Semaphore>();
495 private Dictionary<String, Semaphore> AgentLocks
496 {
497     get
498     {
499         lock (al)
500             return al;
501     }
502     set
503     {
504         lock (al)
505             al = value;
506     }
507 }
508 private List<Condition> wc = null;
509 private List<Condition> WorldConditions
510 {
511     get
512     {
513         lock (WorldConditionLocker)
514             return wc;
515     }
516     set
517     {
518         lock (WorldConditionLocker)
519             wc = value;
520     }
521 }
522 }
523 }
```

## D.10 PutOnShoe.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using NUnit.Framework;
5
6  namespace MultiRoboticAgentsPlanning
7  {
8      class LeftBareFoot : Condition
9      {
10     }
11
12     class RightBareFoot : Condition
13     {
14     }
15
16     class LeftSockOn : Condition
17     {
18     }
19
20     class RightSockOn : Condition
21     {
22     }
23
24     class LeftShoeOn : Condition
25     {
26     }
27
28     class RightShoeOn : Condition
29     {
30     }
31
32     class LeftSock : Action
33     {
34         public LeftSock() : base()
35         {
36             preconditionsTypes.Add(typeof(LeftBareFoot));
37             addConditionTypes.Add(typeof(LeftSockOn));
38
39             alterPreConditionTypes.Add(typeof(LeftBareFoot));
40             alterAddConditionTypes.Add(typeof(LeftSockOn));
41         }
42     }
43
44     class LeftShoe : Action
45     {
46         public LeftShoe() : base()
47         {
48             preconditionsTypes.Add(typeof(LeftSockOn));
49             addConditionTypes.Add(typeof(LeftShoeOn));
50
51             alterPreConditionTypes.Add(typeof(LeftSockOn));
52             alterAddConditionTypes.Add(typeof(LeftShoeOn));
```

```
53     }
54 }
55
56 class RightSock : Action
57 {
58     public RightSock() : base()
59     {
60         preconditionsTypes.Add(typeof(RightBareFoot));
61         addConditionTypes.Add(typeof(RightSockOn));
62
63         alterPreConditionTypes.Add(typeof(RightBareFoot));
64         alterAddConditionTypes.Add(typeof(RightSockOn));
65     }
66 }
67
68 class RightShoe : Action
69 {
70     public RightShoe() : base()
71     {
72         preconditionsTypes.Add(typeof(RightSockOn));
73         addConditionTypes.Add(typeof(RightShoeOn));
74
75         alterPreConditionTypes.Add(typeof(RightSockOn));
76         alterAddConditionTypes.Add(typeof(RightShoeOn));
77     }
78 }
79
80 class BothShoes : Action
81 {
82     public BothShoes() : base()
83     {
84         preconditionsTypes.Add(typeof(LeftSockOn));
85         preconditionsTypes.Add(typeof(RightSockOn));
86         addConditionTypes.Add(typeof(LeftShoeOn));
87         addConditionTypes.Add(typeof(RightShoeOn));
88
89         alterPreConditionTypes = preconditionsTypes;
90         alterAddConditionTypes = addConditionTypes;
91     }
92 }
93
94 [TestFixture]
95 public class PutOnShoes
96 {
97     public PutOnShoes()
98     {
99     }
100 }
101
102 [Test]
103 public void Test1()
104 {
105     List<Action> actions = new List<Action>();
106     actions.Add(new LeftSock());
107     actions.Add(new LeftShoe());
```

```
108     actions.Add(new RightSock());
109     actions.Add(new RightShoe());
110
111     List<Condition> startstate = new List<Condition>();
112     List<Condition> goalstate = new List<Condition>();
113
114     startstate.Add(new LeftBareFoot());
115     startstate.Add(new RightBareFoot());
116
117     goalstate.Add(new LeftShoeOn());
118     goalstate.Add(new RightShoeOn());
119
120     POPstar popstar = new POPstar(startstate, goalstate, actions)
121         ;
122
123     popstar.Start();
124
125     Assert.IsNotNull(popstar.BestPlan);
126
127     Console.WriteLine("Final_plan:");
128     foreach (Action a in popstar.BestPlan.
129         GetOrdersTopologicalSort())
130     {
131         Console.WriteLine("_" + a);
132     }
133 }
134 [Test]
135 public void Test2()
136 {
137     List<Action> actions = new List<Action>();
138     actions.Add(new LeftSock());
139     actions.Add(new RightSock());
140     actions.Add(new BothShoes());
141
142     List<Condition> startstate = new List<Condition>();
143     List<Condition> goalstate = new List<Condition>();
144
145     startstate.Add(new LeftBareFoot());
146     startstate.Add(new RightBareFoot());
147
148     goalstate.Add(new LeftShoeOn());
149     goalstate.Add(new RightShoeOn());
150
151     POPstar popstar = new POPstar(startstate, goalstate, actions)
152         ;
153
154     popstar.Start();
155
156     Assert.IsNotNull(popstar.BestPlan);
157
158
159
```

```

160     Console.WriteLine("Final_plan:");
161     foreach (Action a in popstar.BestPlan.
           GetOrdersTopologicalSort())
162     {
163         Console.WriteLine("_" + a);
164     }
165 }
166 }
167 }

```

## D.11 ActionBlockWorld.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4  using System.Collections;
5  using NXTRemoteControl;
6
7  namespace MultiRoboticAgentsPlanning
8  {
9
10     public abstract class BlockWorldAction : Action
11     {
12         public BlockWorldAction():base()
13         {
14         }
15
16         public override void Execute(NXTControl nxt)
17         {
18             throw new Exception("The_method_or_operation_is_not_
           implemented.");
19         }
20     }
21
22     public class PickupBlockAction : BlockWorldAction
23     {
24         public PickupBlockAction()
25         : base()
26         {
27
28             preconditionsTypes.Add(typeof(ClearBoxCondition));
29             preconditionsTypes.Add(typeof(OnBoxCondition));
30             preconditionsTypes.Add(typeof(EmptyHandCondition));
31
32             addConditionTypes.Add(typeof(HoldingCondition));
33             addConditionTypes.Add(typeof(ClearBoxCondition));
34
35             alterPreConditionTypes.Add(typeof(ClearBoxCondition));
36             alterPreConditionTypes.Add(typeof(OnBoxCondition));
37             alterPreConditionTypes.Add(typeof(EmptyHandCondition));
38
39             alterAddConditionTypes.Add(typeof(HoldingCondition));
40

```

```

41     alterAddConditionTypes.Add(typeof(ClearBoxCondition));
42 }
43
44 public override List<Link> NewOpenPreconditions(List<Link>
45     OpenPreconditions, CausalLink SubGoalLink, Plan p)
46 {
47     ConditionVariable SubGoalCondition = (ConditionVariable)
48         SubGoalLink.Condition;
49     MyList<Link> result = new MyList<Link>(OpenPreconditions);
50
51     Action a = result[0].ActionPost;
52     result.RemoveAt(0);
53
54     Variable boxA = new Variable();
55     Variable boxB = new Variable();
56
57     if (SubGoalCondition is ClearBoxCondition)
58     {
59         boxA = SubGoalCondition.GetVariableX();
60
61         result.Add(new Link(null, new OnBoxCondition(boxB, boxA),
62             this));
63         result.Add(new Link(null, new ClearBoxCondition(boxB, new
64             Variable(0)), this));
65         result.Add(new Link(null, new EmptyHandCondition(), this));
66     }
67     else if (SubGoalCondition is HoldingCondition)
68     {
69         boxB = SubGoalCondition.GetVariableX();
70
71         result.Add(new Link(null, new ClearBoxCondition(boxB, new
72             Variable(0)), this));
73         result.Add(new Link(null, new EmptyHandCondition(), this));
74         result.Add(new Link(null, new OnBoxCondition(boxB, boxA),
75             this));
76     }
77
78     Condition temp = new ClearBoxCondition(boxA, new Variable(0))
79         ;
80     p.previousActions.Add(new Link(this, temp, null));
81     //p.CausalLinks.Add(new CausalLink(this, temp, a));
82     temp = new HoldingCondition(boxB, new Variable(0));
83     p.previousActions.Add(new Link(this, temp, null));
84     //p.CausalLinks.Add(new CausalLink(this, temp, a));
85
86     return result;
87 }
88
89 public class PickupTableAction : BlockWorldAction
90 {
91     public PickupTableAction()
92         : base()
93     {

```

```

89
90     preconditionsTypes.Add(typeof(ClearBoxCondition));
91     preconditionsTypes.Add(typeof(OnTableCondition));
92     preconditionsTypes.Add(typeof(EmptyHandCondition));
93
94     addConditionTypes.Add(typeof(HoldingCondition));
95
96     alterPreConditionTypes.Add(typeof(ClearBoxCondition));
97     alterPreConditionTypes.Add(typeof(OnTableCondition));
98     alterPreConditionTypes.Add(typeof(EmptyHandCondition));
99
100    alterAddConditionTypes.Add(typeof(HoldingCondition));
101 }
102
103 public override List<Link> NewOpenPreconditions(List<Link>
104     OpenPreconditions, CausalLink SubGoalLink, Plan p)
105 {
106     ConditionVariable SubGoalCondition = (ConditionVariable)
107         SubGoalLink.Condition;
108     MyList<Link> result = new MyList<Link>(OpenPreconditions);
109
110     result.RemoveAt(0);
111
112     Variable boxA = SubGoalCondition.GetVariableX();
113
114     result.Add(new Link(null, new ClearBoxCondition(boxA, new
115         Variable(0)), this));
116     result.Add(new Link(null, new EmptyHandCondition(), this));
117     result.Add(new Link(null, new OnTableCondition(boxA, new
118         Variable(0)), this));
119
120     p.previousActions.Add(new Link(this, new HoldingCondition(
121         boxA, new Variable(0)), null));
122
123     return result;
124 }
125
126 public class PutdownAction : BlockWorldAction
127 {
128     public PutdownAction()
129         : base()
130     {
131         //preconditionsTypes.Add(typeof(ClearBoxCondition));
132         preconditionsTypes.Add(typeof(HoldingCondition));
133
134         addConditionTypes.Add(typeof(OnBoxCondition));
135         addConditionTypes.Add(typeof(ClearBoxCondition));
136         addConditionTypes.Add(typeof(EmptyHandCondition));
137
138         alterPreConditionTypes.Add(typeof(HoldingCondition));
139         //alterPreConditionTypes.Add(typeof(ClearBoxCondition));

```

```

139     alterAddConditionTypes.Add(typeof(OnBoxCondition));
140     //alterAddConditionTypes.Add(typeof(EmptyHandCondition));
141     //alterAddConditionTypes.Add(typeof(ClearBoxCondition));
142 }
143
144
145 public override List<Link> NewOpenPreconditions(List<Link>
    OpenPreconditions, CausalLink SubGoalLink, Plan p)
146 {
147     Condition SubGoalCondition = SubGoalLink.Condition;
148     MyList<Link> result = new MyList<Link>(OpenPreconditions);
149
150     result.RemoveAt(0);
151
152     Variable boxA = new Variable();
153     Variable boxB = new Variable();
154
155     if (SubGoalCondition is ClearBoxCondition)
156     {
157         boxA = ((ConditionVariable)SubGoalCondition).GetVariableX()
158             ;
159     }
160     else if (SubGoalCondition is OnBoxCondition)
161     {
162         boxA = ((ConditionVariable)SubGoalCondition).GetVariableX()
163             ;
164         boxB = ((ConditionVariable)SubGoalCondition).GetVariableY()
165             ;
166     }
167
168     result.Add(new Link(null, new HoldingCondition(boxA, new
        Variable(0)), this));
169     result.Add(new Link(null, new ClearBoxCondition(boxB, new
        Variable(0)), this));
170
171     p.previousActions.Add(new Link(this, new OnBoxCondition(boxA,
        boxB), null));
172     p.previousActions.Add(new Link(this, new ClearBoxCondition(
        boxA, new Variable(0)), null));
173     p.previousActions.Add(new Link(this, new EmptyHandCondition()
        , null));
174
175     return result;
176 }
177
178 public class PutdownTableAction : BlockWorldAction
179 {
180     public PutdownTableAction()
181     : base()
182     {
183         preconditionsTypes.Add(typeof(HoldingCondition));
184         addConditionTypes.Add(typeof(OnTableCondition));

```

```

185     addConditionTypes.Add(typeof(EmptyHandCondition));
186     addConditionTypes.Add(typeof(ClearBoxCondition));
187
188     alterPreConditionTypes.Add(typeof(HoldingCondition));
189
190     alterAddConditionTypes.Add(typeof(OnTableCondition));
191     alterAddConditionTypes.Add(typeof(EmptyHandCondition));
192     //alterAddConditionTypes.Add(typeof(ClearBoxCondition));
193 }
194 public override List<Link> NewOpenPreconditions(List<Link>
    OpenPreconditions, CausalLink SubGoalLink, Plan p)
195 {
196     Condition SubGoalCondition = SubGoalLink.Condition;
197     MyList<Link> result = new MyList<Link>(OpenPreconditions);
198
199     result.RemoveAt(0);
200
201     Variable boxA = new Variable();
202
203     if (SubGoalCondition is ConditionVariable)
204     {
205         boxA = ((ConditionVariable)SubGoalCondition).GetVariableX()
                ;
206     }
207
208     result.Add(new Link(null, new HoldingCondition(boxA, new
        Variable(0)), this));
209
210     p.previousActions.Add(new Link(this, new OnTableCondition(
        boxA, new Variable(0)), null));
211     p.previousActions.Add(new Link(this, new ClearBoxCondition(
        boxA, new Variable(0)), null));
212     p.previousActions.Add(new Link(this, new EmptyHandCondition()
        , null));
213
214     return result;
215 }
216 }
217
218 }

```

## D.12 ConditionBlockWorld.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Text;
4
5  namespace MultiRoboticAgentsPlanning
6  {
7      /* holding(B): the gripper is holding block B
8         empty: the gripper is not holding a block
9         on(B1,B2): block B1 is on top of block B2
10        ontable(B): block B is on the table

```

```

11      clear(B): block B has no blocks on top of it and is not being
12      held by the gripper
13      */
14      public class HoldingCondition : ConditionVariable
15      {
16
17          //holding(B): the gripper is holding block B
18
19          public HoldingCondition(Variable box, Variable y)
20          {
21              this.xVar = box;
22              this.yVar = y;
23          }
24
25          public HoldingCondition(int x, int y)
26          {
27              xVar = new Variable(x);
28              yVar = new Variable(y);
29          }
30
31          public override bool Threatens(Condition c)
32          {
33              if (c.GetType().Equals(typeof(EmptyHandCondition)))
34              {
35                  return true;
36              }
37              if (c.GetType().Equals(typeof(OnBoxCondition)))
38              {
39                  OnBoxCondition onbc = (OnBoxCondition)c;
40                  if (onbc.GetIntegerX() == this.GetIntegerX())
41                      return true;
42                  else
43                      return onbc.GetIntegerX() == this.GetIntegerY();
44              }
45              if (c.GetType().Equals(typeof(OnTableCondition)))
46              {
47                  //if box b from ontable(b) == b from holding(b) then there
48                  is a threat.
49                  OnTableCondition onTabc = (OnTableCondition)c;
50                  return onTabc.GetIntegerX() == this.GetIntegerX();
51              }
52              if (c.GetType().Equals(typeof(ClearBoxCondition)))
53              {
54                  //if box b from clearbox(b) == b from holding(b) then there
55                  is a threat.
56                  ClearBoxCondition clearbc = (ClearBoxCondition)c;
57                  return clearbc.GetIntegerX() == this.GetIntegerX();
58              }
59              return false;
60          }
61
62          public override bool IsBothBound()
63          {

```

```

63     return this.GetVariableX().IsBound();
64 }
65
66 public override String ToString()
67 {
68     return "Holding(" + (this.GetVariableX().IsBound() ? this.
        GetIntegerX() + " : ") + ")";
69 }
70 }
71
72 public class EmptyHandCondition : Condition
73 {
74
75     public override bool Threatens(Condition c)
76     {
77         return c.GetType().Equals(typeof(HoldingCondition));
78     }
79
80     public override string ToString()
81     {
82         return "EmptyHand";
83     }
84 }
85
86 public class OnBoxCondition : ConditionVariable
87 {
88
89     //OnBox(B1,B2): block B1 is on top of block B2
90
91     public OnBoxCondition(Variable box1, Variable box2)
92     {
93         this.xVar = box1;
94         this.yVar = box2;
95     }
96
97     public OnBoxCondition(int box1, int box2)
98     {
99         xVar = new Variable(box1);
100        yVar = new Variable(box2);
101    }
102
103    public override bool Threatens(Condition c)
104    {
105        if (c.GetType().Equals(typeof(HoldingCondition)))
106        {
107            HoldingCondition holdc = (HoldingCondition)c;
108            if (holdc.GetIntegerX() == this.GetIntegerX())
109                return true;
110            else
111                return holdc.GetIntegerX() == this.GetIntegerY();
112        }
113
114        if (c.GetType().Equals(typeof(OnTableCondition)))
115        {

```

```

116         //if box b from ontable(b) == a from onbox(a,b) then there
           is a threat.
117         OnTableCondition onTabc = (OnTableCondition)c;
118         return onTabc.GetIntegerX() == this.GetIntegerX();
119     }
120
121     if (c.GetType().Equals(typeof(ClearBoxCondition)))
122     {
123         //if box b from clear(b) == b from onbox(a,b) then there is
           a threat.
124         ClearBoxCondition clearc = (ClearBoxCondition)c;
125         return clearc.GetIntegerX() == this.GetIntegerY();
126     }
127
128     return false;
129 }
130
131 public override String ToString()
132 {
133     return "OnBox(" + (this.GetVariableX().IsBound() ? this.
           GetIntegerX() + " " : " ") + ", "
134         + (this.GetVariableY().IsBound() ? this.GetIntegerY() + " "
           : " ") + ")";
135 }
136 }
137
138
139 public class OnTableCondition : ConditionVariable
140 {
141
142     //ontable(B): block B is on the table
143
144     public OnTableCondition(Variable box, Variable y)
145     {
146         this.xVar = box;
147         this.yVar = y;
148     }
149
150     public OnTableCondition(int x, int y)
151     {
152         xVar = new Variable(x);
153         yVar = new Variable(y);
154     }
155
156     public override bool Threatens(Condition c)
157     {
158         if (c.GetType().Equals(typeof(HoldingCondition)))
159         {
160             HoldingCondition holdc = (HoldingCondition)c;
161             return holdc.GetIntegerX() == this.GetIntegerX();
162         }
163
164         if (c.GetType().Equals(typeof(OnBoxCondition)))
165         {

```

```

166         //if box a from onbox(a,b) == b from ontable(b) then there
           is a threat.
167         OnBoxCondition onbc = (OnBoxCondition)c;
168         return onbc.GetIntegerX() == this.GetIntegerX();
169     }
170
171     return false;
172 }
173
174 public override bool IsBothBound()
175 {
176     return this.GetVariableX().IsBound();
177 }
178
179 public override String ToString()
180 {
181     return "OnTable(" + (this.GetVariableX().IsBound() ? this.
           GetIntegerX() + " : ") + ")";
182 }
183 }
184
185
186
187 public class ClearBoxCondition : ConditionVariable
188 {
189
190     //clear(B): block B has no blocks on top of it and is not being
           held by the gripper
191
192     public ClearBoxCondition(Variable box, Variable y)
193     {
194         this.xVar = box;
195         this.yVar = y;
196     }
197
198     public ClearBoxCondition(int x, int y)
199     {
200         xVar = new Variable(x);
201         yVar = new Variable(y);
202     }
203
204     public override bool Threatens(Condition c)
205     {
206         if (c.GetType().Equals(typeof(HoldingCondition)))
207         {
208             HoldingCondition holdc = (HoldingCondition)c;
209             return holdc.GetIntegerX() == this.GetIntegerX();
210         }
211
212         if (c.GetType().Equals(typeof(OnBoxCondition)))
213         {
214             //if box b from onbox(a,b) == b from clearbox(b) then there
           is a threat.
215             OnBoxCondition onbc = (OnBoxCondition)c;
216             return onbc.GetIntegerY() == this.GetIntegerX();

```

```
217     }
218
219     return false;
220 }
221
222 public override String ToString()
223 {
224     return "Clear(" + (this.GetVariableX().IsBound() ? this.
225         GetIntegerX() + " " : "") + ")";
226 }
227
228 public override bool IsBothBound()
229 {
230     return this.GetVariableX().IsBound();
231 }
232 }
```

## D.13 UnitTest.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4 using NUnit.Framework;
5
6 namespace MultiRoboticAgentsPlanning
7 {
8
9
10
11     [TestFixture]
12     public class PlanTest
13     {
14         private List<Action> actions;
15         private Plan FinalPlan;
16
17         [SetUp]
18         public void Init()
19         {
20             MoveAction.SetMaxX(3);
21             MoveAction.SetMaxY(3);
22
23             actions = new MyList<Action>();
24             actions.Add(new MoveWestAction());
25             actions.Add(new MoveSouthAction());
26             actions.Add(new MoveEastAction());
27             actions.Add(new MoveNorthAction());
28             actions.Add(new ReleaseWestAction());
29             actions.Add(new ReleaseSouthAction());
30             actions.Add(new ReleaseEastAction());
31             actions.Add(new ReleaseNorthAction());
32             actions.Add(new PickupAction());
33     }
```

```

34
35     [Test]
36     public void BoxAtTest1()
37     {
38
39         int startx = 3, starty = 0;
40         int startBoxX = 3, startBoxY = 3;
41         int finishBoxX = 0, finishBoxY = 0;
42
43         List<Condition> startstate = new MyList<Condition>();
44         startstate.Add(new AtCondition(startx, starty));
45         startstate.Add(new BoxAtCondition(startBoxX, startBoxY));
46         //startstate.Add(new BoxAtCondition(2,2));
47         //startstate.Add(new BoxAtCondition(1,1));
48         startstate.Add(new NotHaveBallCondition());
49
50         List<Condition> goalstate = new MyList<Condition>();
51         goalstate.Add(new BoxAtCondition(finishBoxX, finishBoxY));
52         goalstate.Add(new AtCondition(new Variable(), new Variable())
53             );
54
55         POPstar popstar = new POPstar(startstate, goalstate, actions)
56             ;
57         popstar.Start();
58         this.FinalPlan = popstar.BestPlan;
59
60         Assert.IsNotNull(FinalPlan);
61         LinkedList<Action> topoSort = FinalPlan.
62             GetOrdersTopologicalSort();
63
64         Console.WriteLine(topoSort.Count);
65
66         foreach (Action action in topoSort)
67         {
68             if (action is MoveAction)
69             {
70                 finishBoxX += ((MoveAction)action).deltaX;
71                 finishBoxY += ((MoveAction)action).deltaY;
72             }
73             else if (action is ReleaseAction)
74             {
75                 finishBoxX += ((ReleaseAction)action).deltaXneg;
76                 finishBoxY += ((ReleaseAction)action).deltaYneg;
77             }
78             Console.WriteLine("___" + action);
79             Assert.Contains(action, FinalPlan.actions);
80         }
81
82         //Assert.AreEqual(finishBoxX, startx);
83         //Assert.AreEqual(finishBoxY, starty);
84     }
85
86     [Test]
87     public void BoxAtTest2()

```

```

86     {
87         int startx = 0, starty = 2;
88         int finishBoxX = 2, finishBoxY = 3;
89
90         List<Condition> startstate = new MyList<Condition>();
91
92         startstate.Add(new AtCondition(startx, starty));
93         startstate.Add(new HaveBallCondition());
94
95
96         List<Condition> goalstate = new MyList<Condition>();
97
98         goalstate.Add(new BoxAtCondition(finishBoxX, finishBoxY));
99         //goalstate.Add(new AtCondition(0, 0));
100
101
102         POPstar popstar = new POPstar(startstate, goalstate, actions)
103             ;
104         popstar.Start();
105         this.FinalPlan = popstar.BestPlan;
106
107         Assert.IsNotNull(FinalPlan);
108         LinkedList<Action> topoSort = FinalPlan.
109             GetOrdersTopologicalSort();
110
111         Assert.AreEqual(topoSort.Count, 5);
112
113         foreach (Action action in topoSort)
114         {
115             if (action is MoveAction)
116             {
117                 finishBoxX += ((MoveAction)action).deltaX;
118                 finishBoxY += ((MoveAction)action).deltaY;
119             }
120             else if (action is ReleaseAction)
121             {
122                 finishBoxX += ((ReleaseAction)action).deltaXneg;
123                 finishBoxY += ((ReleaseAction)action).deltaYneg;
124             }
125             Console.WriteLine("___" + action);
126             Assert.Contains(action, FinalPlan.actions);
127         }
128         Assert.AreEqual(finishBoxX, startx);
129         Assert.AreEqual(finishBoxY, starty);
130     }
131
132
133     [Test]
134     public void AtTest()
135     {
136         List<Condition> startstate = new MyList<Condition>();
137         startstate.Add(new AtCondition(3, 3));
138     }

```

```

139     List<Condition> goalstate = new MyList<Condition>();
140     goalstate.Add(new AtCondition(0, 0));
141
142     POPstar popstar = new POPstar(startstate, goalstate, actions)
143         ;
144     popstar.Start();
145     this.FinalPlan = popstar.BestPlan;
146
147     Assert.IsNotNull(FinalPlan);
148     LinkedList<Action> topoSort = FinalPlan.
149         GetOrdersTopologicalSort();
150
151     Assert.IsNotNull(topoSort);
152     Assert.IsTrue(topoSort.Count == 8);
153     int moveWestCount = 0;
154     int moveSouthCount = 0;
155
156     foreach (Action action in topoSort)
157     {
158         Assert.Contains(action, FinalPlan.actions);
159         Assert.IsTrue((action is MoveAction) || (action is
160             FinishAction) || (action is StartAction));
161
162         if (action is MoveWestAction)
163             moveWestCount++;
164         else if (action is MoveSouthAction)
165             moveSouthCount++;
166
167         Console.WriteLine("_ _ _" + action);
168     }
169     Assert.AreEqual(moveSouthCount, 3);
170     Assert.AreEqual(moveWestCount, 3);
171 }
172
173 [Test]
174 public void HaveBallTest()
175 {
176     int startx = 0, starty = 0;
177     int startBoxX = 3, startBoxY = 3;
178
179     List<Condition> startstate = new MyList<Condition>();
180     startstate.Add(new AtCondition(startx, starty));
181     startstate.Add(new NotHaveBallCondition());
182     startstate.Add(new BoxAtCondition(startBoxX, startBoxY));
183
184     List<Condition> goalstate = new MyList<Condition>();
185     goalstate.Add(new HaveBallCondition());
186     //goalstate.Add(new AtCondition(new Variable(), new Variable
187     ());
188
189     POPstar popstar = new POPstar(startstate, goalstate, actions)
190         ;
191     popstar.Start();
192     this.FinalPlan = popstar.BestPlan;
193

```

```

189     Assert.IsNotNull(FinalPlan);
190     LinkedList<Action> topoSort = FinalPlan.
        GetOrdersTopologicalSort();
191
192     Console.WriteLine(topoSort.Count);
193
194     foreach (Action action in topoSort)
195     {
196         if (action is MoveAction)
197         {
198             startBoxX += ((MoveAction) action).deltaX;
199             startBoxY += ((MoveAction) action).deltaY;
200         }
201         else if (action is ReleaseAction)
202         {
203             startBoxX += ((ReleaseAction) action).deltaXneg;
204             startBoxY += ((ReleaseAction) action).deltaYneg;
205         }
206         Console.WriteLine("_ _ _" + action);
207         Assert.Contains(action, FinalPlan.actions);
208     }
209
210     Assert.AreEqual(startBoxX, startx);
211     Assert.AreEqual(startBoxY, starty);
212 }
213
214 [Test]
215 public void MoveThroughBallTest()
216 {
217     int startx = 3, starty = 0;
218     //int startBoxX1 = 1, startBoxY1 = 0;
219     //int startBoxX2 = 2, startBoxY2 = 0;
220     int finishx = 0, finishy = 0;
221
222     List<Condition> startstate = new MyList<Condition>();
223     startstate.Add(new AtCondition(3, 0));
224     startstate.Add(new BoxAtCondition(1, 0));
225     startstate.Add(new BoxAtCondition(2, 0));
226     startstate.Add(new BoxAtCondition(1, 1));
227     startstate.Add(new BoxAtCondition(0, 1));
228     startstate.Add(new BoxAtCondition(0, 2));
229     startstate.Add(new NotHaveBallCondition());
230
231     List<Condition> goalstate = new MyList<Condition>();
232     goalstate.Add(new AtCondition(0, 0));
233
234
235     POPstar popstar = new POPstar(startstate, goalstate, actions)
        ;
236     popstar.Start();
237     this.FinalPlan = popstar.BestPlan;
238
239     Assert.IsNotNull(FinalPlan);
240     LinkedList<Action> topoSort = FinalPlan.
        GetOrdersTopologicalSort();

```

```
241
242     Console.WriteLine(topoSort.Count);
243
244     foreach (Action action in topoSort)
245     {
246         if (action is MoveAction)
247         {
248             finishx += ((MoveAction)action).deltaX;
249             finishy += ((MoveAction)action).deltaY;
250         }
251         Console.WriteLine("___" + action);
252         Assert.Contains(action, FinalPlan.actions);
253     }
254
255     Assert.AreEqual(finishx, startx);
256     Assert.AreEqual(finishy, starty);
257 }
258
259 [Test]
260 public void NotHaveBallTest()
261 {
262     int startx = 1, starty = 1;
263     int finishx = 0, finishy = 0;
264
265     List<Condition> startstate = new MyList<Condition>();
266     startstate.Add(new AtCondition(startx, starty));
267     startstate.Add(new HaveBallCondition());
268
269     List<Condition> goalstate = new MyList<Condition>();
270     goalstate.Add(new NotHaveBallCondition());
271     goalstate.Add(new AtCondition(finishx, finishy));
272
273
274     POPstar popstar = new POPstar(startstate, goalstate, actions)
275     ;
276     popstar.Start();
277     this.FinalPlan = popstar.BestPlan;
278
279     Assert.IsNotNull(FinalPlan);
280     LinkedList<Action> topoSort = FinalPlan.
281         GetOrdersTopologicalSort();
282
283     foreach (Action action in topoSort)
284     {
285         if (action is MoveAction)
286         {
287             finishx += ((MoveAction)action).deltaX;
288             finishy += ((MoveAction)action).deltaY;
289         }
290         Assert.Contains(action, FinalPlan.actions);
291         Console.WriteLine("___" + action);
292     }
293
294     Assert.AreEqual(finishx, startx);
```

```
294     Assert.AreEqual(finishy, starty);
295 }
296
297 [Test]
298 public void LongWay()
299 {
300     List<Condition> startstate = new MyList<Condition>();
301     startstate.Add(new AtCondition(0, 0));
302     startstate.Add(new BoxAtCondition(1, 0));
303     startstate.Add(new BoxAtCondition(1, 1));
304     startstate.Add(new BoxAtCondition(2, 1));
305     startstate.Add(new BoxAtCondition(1, 2));
306     startstate.Add(new BoxAtCondition(2, 2));
307
308     List<Condition> goalstate = new MyList<Condition>();
309     goalstate.Add(new AtCondition(2,0));
310
311     POPstar popstar = new POPstar(startstate, goalstate, actions)
312     ;
313     popstar.Start();
314     this.FinalPlan = popstar.BestPlan;
315
316     Assert.IsNotNull(FinalPlan);
317 }
318
319 [TestFixture]
320 public class ActionTest
321 {
322     PickupAction pickup;
323     ReleaseSouthAction releasenorth;
324     ReleaseWestAction releaseeast;
325     ReleaseNorthAction releasesouth;
326     ReleaseEastAction releasewest;
327     MoveNorthAction movenorth;
328     MoveEastAction moveeast;
329     MoveSouthAction movesouth;
330     MoveWestAction movewest;
331     StartAction start;
332     FinishAction finish;
333
334     [SetUp]
335     public void Init()
336     {
337         pickup = new PickupAction();
338         releasenorth = new ReleaseSouthAction();
339         releaseeast = new ReleaseWestAction();
340         releasesouth = new ReleaseNorthAction();
341         releasewest = new ReleaseEastAction();
342         movenorth = new MoveNorthAction();
343         moveeast = new MoveEastAction();
344         movesouth = new MoveSouthAction();
345         movewest = new MoveWestAction();
346
347         List<Condition> startcond = new List<Condition>();
```

```
348     startcond.Add(new AtCondition(0,0));
349     start = new StartAction(startcond);
350
351
352     List<Condition> finishcond = new List<Condition>();
353     finishcond.Add(new AtCondition(0, 1));
354     finish = new FinishAction(finishcond);
355 }
356
357 [Test]
358 public void PickupAction()
359 {
360
361 }
362 }
363
364 [TestFixture]
365 public class ConditionTest
366 {
367     [SetUp]
368     public void Init()
369     {
370     }
371
372     [Test]
373     public void AtCondition()
374     {
375         AtCondition c1 = new AtCondition(3,4);
376         AtCondition c2 = new AtCondition(new Variable(3), new
            Variable(4));
377         AtCondition c3 = new AtCondition(new Variable(), new Variable
            ());
378
379         Assert.IsTrue(c1.IsBothBound());
380         Assert.AreEqual(c1.GetIntegerX(), 3);
381         Assert.AreEqual(c1.GetIntegerY(), 4);
382
383         Assert.AreEqual(c1, c2);
384
385         StringAssert.AreEqualIgnoringCase(c1.ToString(), "At(3,4)");
386
387
388         Assert.IsFalse(c3.IsBothBound());
389         Assert.AreNotEqual(c1, c3, "C1_and_C3_is_equals");
390         StringAssert.AreEqualIgnoringCase(c3.ToString(), "At()");
391
392         Assert.IsTrue(!c1.IsSamePlace(c3));
393
394         c3.BindVariables(3, 4);
395         Assert.AreEqual(c1, c3);
396     }
397
398     public void HaveBallCondition()
399     {
400
```

```
401     }  
402   }  
403 }
```

## D.14 Link.cs

```
1  using System;  
2  using System.Collections.Generic;  
3  using System.Text;  
4  
5  namespace MultiRoboticAgentsPlanning  
6  {  
7      public class Link  
8      {  
9          public Action ActionPre;  
10  
11         public Action ActionPost;  
12  
13         public Condition Condition;  
14  
15         public Link(Action ActionPre, Condition Condition, Action  
16             ActionPost)  
17         {  
18             this.ActionPre = ActionPre;  
19             this.ActionPost = ActionPost;  
20             this.Condition = Condition;  
21         }  
22  
23         public override string ToString()  
24         {  
25             return ActionPre+"-"+Condition+"-"+ActionPost;  
26         }  
27  
28         public abstract class LinkBase  
29         {  
30         }  
31  
32         public class CausalLink : LinkBase  
33         {  
34             public Action PreAction;  
35             public Condition Condition;  
36             public Action PostAction;  
37  
38             public CausalLink(Action PreAction, Condition Condition, Action  
39                 PostAction)  
40             {  
41                 this.PreAction = PreAction;  
42                 this.Condition = Condition;  
43                 this.PostAction = PostAction;  
44             }  
45  
46             public override string ToString()
```

```
46     {
47         return "Link("+PreAction+",_"+Condition+",_"+PostAction+)";
48     }
49 }
50
51 public class OrderingConstraint : LinkBase
52 {
53     public Action PreAction;
54     public Action PostAction;
55
56     public OrderingConstraint(Action PreAction, Action PostAction)
57     {
58         this.PreAction = PreAction;
59         this.PostAction = PostAction;
60     }
61
62     public override string ToString()
63     {
64         return "Order(_<_" + PreAction + "_" + PostAction + ")";
65     }
66 }
67
68 }
```

