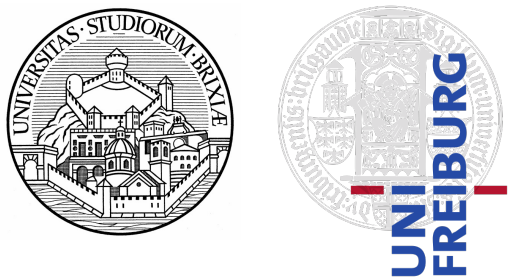
UNIVERSITÀ DEGLI STUDI DI BRESCIA
FACOLTÀ DI INGEGNERIA



CORSO DI LAUREA IN INGEGNERIA INFORMATICA
TESI DI LAUREA SPECIALISTICA

ANALISI DI FATTIBILITÀ ED IMPLEMENTAZIONE DI UN SISTEMA DI ROBOTICA COGNITIVA PER COMPITI DI NAVIGAZIONE

(Analysis of preconditions and implementation of a Cognitive Robotic System for navigation tasks)

Relatore:

Ch.mo Prof. Riccardo Cassinis

Correlatore:

Ch.mo Prof. Marco Ragni

Laureando:
Stefano Bennati
Matricola 72363

Anno Accademico 2010-2011

Under the sword lifted high,
There is hell making you tremble.
But go ahead,
And you have the land of bliss.
– Miyamoto Musashi

Contents

List of Figures

Introduzione

La Robotica Cognitiva

La Robotica Cognitiva è una disciplina di recente introduzione ed in pieno sviluppo. All'atto pratico è il punto d'incontro della Robotica tradizionale con le Scienze Cognitive, due discipline per molti versi fra loro complementari.
Lo scopo della Robotica Cognitiva è quello di affiancare alla tecnica tradizionale della Robotica le recenti soluzioni di controllo che le Scienze Cognitive offrono.
In particolare le Scienze Cognitive studiano il funzionamento del cervello umano ed i processi di ragionamento: numerose teorie sono già state formulate e stumenti software che le implementano sono già in fase avanzata.

La creazione di un sistema di controllo che funzioni come il cervello umano e che ne replichi il ragionamento, permetterebbe ad un robot dotato di tale sistema di interagire in modo più semplice e agevole con gli esseri umani.
Un'architettura cognitiva risulta, agli occhi di un essere umano, qualcosa di familiare e di più semplice utilizzo. Al contrario gli attuali sistemi, anche i più avanzati, sono visti dalla maggior parte degli utenti come dei meri sistemi meccanici, chiusi e statici, la cui logica e comportamento sono distanti da quelli ideali.

Una delle architetture cognitive più riconosciute ed utilizzate è senza dubbio ACT-R. ACT-R è stato creato e testato da centinaia di sviluppatori provenienti dalle Scienze Cognitive, dalla Psicologia e dall'Intelligenza Artificiale. Questo Software è utilizzato in numerosissimi ambiti da altrettanti centri di ricerca, e vanta all'attivo innumerevoli esperimenti che hanno contribuito a confermare ed affinare la teoria che vi sta dietro.

Due importanti progetti legati ad ACT-R sono stati portati a termine dai US Navy's Naval Research Laboratories (NRL) e dal Massachusetts Institute of Technology (MIT). Questi due centri di ricerca hanno portato avanti due differenti studi relativi alla Robotica Cognitiva, il cui scopo era creare degli automi in grado di rapportarsi agevolmente ed efficacemente con degli esseri umani, tramite comandi visivi e vocali.
I risultati sono stati importanti: i robot sono in grado di apprendere nozioni, sia tramite osservazione di oggetti, sia tramite l'interazione con gli esseri umani.

Per ottenere i risultati voluti, questa architettura è stata modificata ed ampliata dagli studiosi statunitensi per meglio adattarla ai loro obiettivi. Inoltre la gestione del robot è stata suddivisa fra questa ed altre architetture cognitive, meglio specializzate in alcuni compiti particolari (es. ragionamento spaziale, fisico e temporale [ST+04]).

Il risultato di queste ricerche è stato molto importante ed innovativo, ma d'altro canto molto complesso da realizzare e scarsamente documentato, a causa delle politiche di sicurezza delle forze armate statunitensi.

I concetti

Il lettore potrebbe chiedersi: "se un lavoro così completo già è stato portato a compimento, qual è l'utilità di calcare un'altra volta quella strada?"
Proseguendo con la lettura diverrà evidente la sostanziale differenza fra il lavoro brevemente descritto poco fa ed il lavoro presentato in questa tesi.
Le maggiori differenze sono due:

La più evidente è che il lavoro documentato da questa tesi non riguarda l'interazione uomo-macchina, tratta invece un argomento che negli esperimenti prima citati è stato trascurato: la navigazione.
Il robot realizzato dai NRL è in grado di navigare autonomamente, tuttavia per il ragionamento utilizza principalmente algoritmi di Intelligenza Artificiale classica, e si appoggia in minima parte all'architettura cognitiva Polyscheme [CC02].
La soluzione elaborata dall'MIT è invece un robot fisso, quindi non possiede alcun algoritmo di navigazione.

La seconda differenza, la più importante, è il metodo realizzativo. Entrambi gli esempi sopra citati utilizzano una complessa serie di sensori per una complessa serie di interazioni con l'ambiente e le persone.
Per gestire questa complessità gli stumenti software esistenti non erano sufficienti ed è stato necessario modificarli ed ampliarli con nuove funzionalità.

La forza di questo progetto sta nella semplicità di realizzazione.

Gli obiettivi

Questo lavoro parte con l'obiettivo di mantenersi il più semplice possibile, in modo da essere facilmente replicato, compreso ed adattato a nuove situazioni.
In particolare l'unica architettura cognitiva utilizzata è ACT-R, privo di qualsivoglia modifica. Le uniche aggiunte riguardano l'introduzione di moduli per l'interfacciamento con il robot che, grazie all'architettura modulare del software, non modificano neppure in minima parte il cuore del sistema.

La stessa filosofia KISS [1] è stata anche alla base del progetto del robot: si è deciso di utilizzare il minimo insieme di sensori sufficienti per la navigazione.
L'utilizzo di sensori più complicati, come una videocamera, avrebbe portato maggiore complessità nella realizzazione e poco significativi miglioramenti alla qualità del software.

Lo scopo finale di questo lavoro è realizzare un agente robotico mobile in grado di navigare all'interno di un labirinto alla ricerca di un obiettivo.
Il robot non possiede conoscenza pregressa dell'ambiente che gli sta attorno, quindi la navigazione seguirà inizialmente una strategia indipendente dalla conformazione del labirinto o dalla posizione dell'agente. Durante l'esplorazione il software memorizza informazioni utili, come le decisioni prese ed il percorso compiuto. Una volta trovato l'obiettivo il robot verrà ricollocato all'ingresso del labirinto e gli sarà nuovamente chiesto di trovare l'obiettivo nella stessa posizione iniziale.

Grazie alle caratteristiche dell'architettura cognitiva ACT-R, l'algoritmo di navigazione è in grado di imparare dall'esperienza, quindi col susseguirsi delle esplorazioni si costruirà una rappresentazione interna dell'ambiente e sarà in grado di migliorare, di volta in volta, le sue prestazioni.

[1] Keep It Simple, Stupid! http://catb.org/jargon/html/K/KISS-Principle.html

L'architettura cognitiva ACT-R

ACT-R è un'architettura cognitiva, un framework che modellizza la struttura ed il comportamento del cervello umano, cercando di spiegare come le varie aree della corteccia cerebrale lavorino assieme e formino la mente umana.
ACT-R implementa una teoria avanzata che permette di simulare complicate caratteristiche del cervello, come le variazioni BOLD (Blood-Oxygen-Level Dependance) di molte zone cerebrali e l'apprendimento.

La teoria alla base di ACT-R parte da un presupposto fondamentale, cioè che la conoscenza può essere divisa in due irriducibili tipi di rappresentazione:
memoria dichiarativa e memoria procedurale.
La memoria dichiarativa si riferisce a tutti quei ricordi che possono essere coscientemente richiamati, per esempio nozioni o fatti memorizzati tramite studio o esperienza.
La memoria procedurale invece, memorizza sequenze di azioni che portano ad un obiettivo. L'uomo le può apprendere dopo una lunga serie di ripetizioni.

Un chiaro esempio di come funziona la memoria procedurale lo si ritrova nello scrivere a macchina: gli esperti battitori sono in grado di scrivere a macchina senza guardare la tastiera, l'esperienza acquisita li porta a ricordare la posizione delle lettere sulla tastiera e a poter digitare in automatico. Tuttavia queste persone non sono in grado, se interrogate, di ricordare la posizione esatta di ciascun tasto.
La conoscenza della posizione delle lettere sulla tastiera non può essere richiamata coscientemente ma permette di raggiungere l'obiettivo di scrivere a macchina concentrandosi sul testo, senza dover prestare attenzione ai movimenti delle dita.

La conoscenza dichiarativa è rappresentata in ACT-R tramite *chunk*: strutture dati che contengono informazioni legate a specifici fatti o ricordi.
La conoscenza procedurale è rappresentata tramite *produzioni*: equivalenti alle funzioni nei linguaggi di programmazione, queste contengono sequenze di azioni che producono determinati effetti e che possono essere attivate solo nel caso in cui determinate precondizioni siano soddisfatte.

Il framework ACT-R è suddiviso in moduli, ciascuno dei quali rappresenta una o più funzioni del cervello e quindi una specifica zona della corteccia cerebrale.
Il modulo procedurale è la parte centrale di questa architettura e si occupa di coordinare tutti gli altri moduli e lo scambio di informazioni fra loro.

La struttura modulare di ACT-R permette ad un programmatore di estendere la teoria che sta dietro ad ACT-R scrivendo un nuovo modulo ed aggiungendolo senza sforzi all'architettura cognitiva.

Ciascun modulo è indipendente dagli altri e comunica con loro tramite uno scambio di chunk, attraverso i buffer.
Ogni modulo possiede almeno un buffer, che forma la sua interfaccia verso gli altri moduli. Un buffer può contenere un solo chunk alla volta: questo può essere generato dal modulo stesso e contenere le informazioni che vuole inviare agli altri moduli, oppure può essere stato creato da una produzione e contenere una richiesta per il modulo a cui appartiene.

In ogni momento solo una produzione per volta può essere avviata.
Perché quella specifica produzione sia avviabile occorre che le sue precondizioni, composte dallo stato dei buffer in quel momento, siano soddisfatte; non solo, essa deve presentare la maggiore utilità fra tutte le produzioni contemporaneamente avviabili.
Il valore di utilità è calcolato, per ciascuna produzione, sulla base di diversi parametri come la probabilità stimata di raggiungere il goal attraverso questa azione, il valore attuale del goal e il tempo stimato necessario a raggiungerlo.

Questo meccanismo di utilità è alla base del sistema di apprendimento di ACT-R: quanto più successo ha una produzione, tanto più cresce la stima della sua probabilità di raggiungere il goal, quindi la sua utilità.
Le produzioni che hanno più successo verranno scelte più frequentemente.

ACT-R è scritto in LISP e offre un linguaggio di modellazione simile al LISP tramite il quale scrivere modelli.
I modelli sono il mezzo tramite il quale un ricercatore può programmare ACT-R: ogni modello rappresenta un compito e specifica l'idea che il ricercatore ha riguardo i processi cognitivi che stanno dietro a quel compito.

La realizzazione

Per permettere all'architettura cognitiva ACT-R di collegarsi al robot e di controllarlo, è stato necessario scrivere un livello software di interfaccia.
Per il controllo del robot è stata scelta una libreria [Hir07] , scritta in LISP, che ha ridotto di molto la quantità e la complessità del lavoro.
Potendo contare su di un substrato di basso livello scritto nello stesso linguaggio di programmazione in cui anche i livelli superiori sono implementati, il lavoro per interfacciarli è stato relativamente semplice.
Come già accennato, la struttura modulare di ACT-R permette di espanderne le capacità in modo facile e pulito, semplicemente aggiungendo nuovi moduli. Ciascun modulo interfaccia un particolare tipo di sensore con il cuore dell'architettura cognitiva, mettendo a disposizione del modellatore le risorse necessarie per utilizzarlo.
In particolare le interazioni fra il modello ed i sensori sono basate sullo scambio di chunk, la struttura dati base di ACT-R, attraverso i buffer appartenenti ai diversi moduli.

Completata l'interfaccia si è proceduto a verificare la sua operatività tramite un programma di test che implementasse, nella sua più classica e deterministica forma, la strategia del perimetro.
Una volta constatata la bontà dell'interfaccia si è proceduto alla scrittura di un simulatore che permettesse al modello di navigare all'interno di labirinti virtuali.

I vantaggi di avere un simulatore sono innanzitutto la grande velocità di esecuzione e la flessibilità provenienti da un ambiente virtuale, che permette la programmazione "batch" e di raccogliere in breve tempo grandi quantità di dati; inoltre è facile apportare ingenti modifiche al labirinto in brevissimo tempo, il che ha permesso di testare il funzionamento del modello su numerosissimi labirinti di prova, molto diversi gli uni dagli altri.
Infine i test risultano molto più affidabili in quanto, all'interno di un ambiente virtuale, non si presentano errori di odometria o di misurazione che sono fisiologici per gli esperimenti nel mondo reale.

La realizzazione del modello vero e proprio ha richiesto la maggior parte del tempo: numerose versioni del modello sono state testate ed innumerevoli problemi corretti.
L'idea iniziale è stata quella di dare all'algoritmo differenti strategie d'esplorazione fra cui scegliere, e lasciare decidere all'esperienza quale fosse la strategia più performante.
Questo sistema si basa su una serie di "ricompense" associate a differenti produzioni (ossia azioni che il modello può compiere): più alta è la ricompensa associata ad una produzione, più alta è la possibilità che questa produzione venga in futuro riutilizzata.
Questo meccanismo di selezione è chiamato *Utility Learning* ed è fornito da ACT-R stesso.

Questo sistema si è dimostrato in grado di imparare, tuttavia numerosi problemi hanno pregiudicato la sua bontà:
per quanto il modello fosse in grado di capire se la strategia del perimetro verso destra fosse migliore, per esempio, di un random walk, questo sistema non è affatto flessibile. Si pensi al caso particolare in cui il percorso ottimo verso l'obiettivo si componga di N deviazioni a destra e, per ultima, di una deviazione a sinistra: il modello così realizzato non sarà mai in grado di accorgersi che la soluzione migliore implica curvare a sinistra all'ultimo bivio; esso continuerà a girare a destra, anche nel caso in cui questo comporti una penalità notevole.
Ancora più problematico è il caso di un random walk: essendo per definizione casuale, capiterà sempre il caso in cui la strada intrapresa sia più lunga di quella strada percorsa utilizzando la strategia del perimetro.
Ne risulta una situazione d'instabilità, e la possibilità che la strategia del perimetro venga scelta anche se, nel caso medio, peggiore.

Da queste considerazioni la decisione di provare un approccio totalmente nuovo: cambiare prospettiva, passando dalle ricompense applicate alle produzioni alle ricompense applicate agli stati: sicché invece di privilegiare genericamente un azione è consigliabile, per ogni stato, dare un valore alle prestazioni di ciascuna.
È stata introdotta una nuova struttura dati, che rappresenta gli incroci del labirinto. Ogni incrocio si differenzia per la lunghezza e la posizione dei corridoi che vi conducono.

Per ciascun incrocio si sono memorizzate le prestazioni di ciascuna direzione. Tramite questi valori è possibile stabilire, per ogni incrocio, quale sia l'azione più consigliabile da eseguire. Le prestazioni vengono valutate da un'apposita funzione che viene invocata ogni volta che si giunge all'obiettivo. La funzione implementa la routine *Policy-Iteration* [RN03, p.624], che consiste nell'aggiornamento delle prestazioni di tutti gli stati incontrati nell'esecuzione attuale.
Le prestazioni di ciascuno stato sono calcolate come la differenza fra il tempo a cui l'obiettivo è stato ritrovato ed il tempo a cui lo stato venne selezionato l'ultima volta; nel caso in cui questo valore sia migliore del precedente, esso viene aggiornato.
Questa routine garantisce che l'algoritmo si stabilizzi ad una soluzione sub-ottima entro un limite di tempo finito.

Questo approccio si è dimostrato molto più soddisfacente del precedente, ed ha permesso di affinare di molto il controllo nella navigazione: grazie a questo sistema è possibile riconoscere e marcare i vicoli ciechi in modo da escluderli da future esplorazioni.
Nel caso estremo il modello è in grado di potare un intero ramo del labirinto, escludendo dalle future scelte uno o più incroci.

In modo simile è possibile riconoscere una strada che è già stata percorsa e prendere provvedimenti in modo da interrompere il ciclo che si è venuto a creare.

Un ultimo passo è consistito nell'integrare nel modello il lavoro svolto da un altro gruppo di ricerca, relativo ad un esperimento piuttosto simile: quel progetto ha implementato un sistema per il calcolo di distanze fra punti di riferimento all'interno di un labirinto.

Il modello naviga nel labirinto, utilizzando un sistema di navigazione molto semplice, ed è in grado di riconoscere alcuni punti di riferimento sparsi per il percorso.
Per ciascun punto memorizza la distanza in passi dal punto di partenza, e calcola la distanza relativa fra quel punto d'interesse ed il precedente.
Alla fine dell'esplorazione il modello calcola, per via indiretta, la distanza fra ciascuna coppia di punti. Durante questo procedimento si possono verificare degli errori di calcolo dovuti alla "somiglianza" fra chunk.

La funzione di similarity è una funzionalità di ACT-R che permette di definire, per una coppia di chunk, una valore di somiglianza che porta ad una più alta probabilità di confondere i chunk durante il recupero dalla memoria. Questo sistema riproduce il comportamento degli esseri umani, quando richiesto loro di svolgere lo stesso compito.

Il modello sopra descritto è stato integrato in questo lavoro utilizzando gli incroci come punti di riferimento.
Il modello è quindi in grado di navigare in un labirinto, sfruttando il migliore algoritmo di navigazione sviluppato in questo progetto, ed una volta trovato il goal, calcolare la distanza fra ciascuna coppia di incroci incontrati durante l'esplorazione.

La valutazione

Una volta completato il modello, si è proceduto a valutarne le prestazioni tramite una serie di test effettuati in differenti labirinti virtuali, ed estrapolarne una statistica da confrontare con dati di riferimento. Questo riferimento proviene da esperimenti dello stesso tipo, compiuti su soggetti umani, svolti per mezzo di un sistema di realtà virtuale che mette in grado l'utente di navigare all'interno di un labirinto tridimensionale.
Nel nuovo esperimento sono stati riutilizzati gli stessi labirinti di prova adoperati nelle prime esperienze, grazie al simulatore è stato sufficiente costruire dei nuovi labirinti virtuali che ne replicassero le caratteristiche.

Per ciascun labirinto sono state simulate diverse sessioni, al termine di un tempo massimo la sessione è interrotta, le informazioni importanti salvate ed avviata una nuova. All'inizio di ogni sessione, l'agente non ha alcuna conoscenza dell'ambiente ed inizia una fase esplorativa applicando una strategia predefinita. Sono state testate diverse strategie d'esplorazione, e le loro prestazioni sono state messe in relazione alla conformazione di ciascun labirinto. Da questi risultati è apparso che alcune strategie funzionano meglio per alcune classi di labirinti. Terminata la fase esplorativa, con l'individuazione del goal, l'agente viene riposizionato all'ingresso e cerca nuovamente di raggiungere l'obiettivo, che si trova ancora nella stessa posizione, sfruttando l'esperienza accumulata all'interno della sessione corrente per migliorare le sue prestazioni.

Il confronto fra le statistiche del vecchio e del nuovo esperimento ha mostrato significativi punti di contatto. L'agente, allo stesso modo dell'essere umano, impara dalla propria esperienza: sceglie la strada più veloce che conosce per raggiungere il goal, non ritorna sui suoi passi ed evita i vicoli ciechi. Si è inoltre visto che utilizzando la strategia d'esplorazione chiamata "del perimetro", che consiste nel seguire il muro destro del labirinto ed è risultata la più utilizzata dall'uomo, le prestazioni della prima esecuzione sono pressoché identiche a quelle registrate con i soggetti umani.
Le prestazioni finali, ovverosia alla stabilizzazione dell'algoritmo presso una soluzione subottima, sono a volte migliori di un ordine di grandezza rispetto a quelle umane.
Questo miglioramento è dovuto alla differente strategia d'esplorazione, rispetto ai soggetti umani, adottata dal modello nelle successive esecuzioni: mentre gli umani preferiscono seguire una strada conosciuta e sicura, un agente robotico non ha paura di perdersi nel labirinto e quindi continua l'esplorazione alla ricerca di un percorso più veloce.
Per questo motivo, spesso la seconda esecuzione mostra prestazioni molto peggiori rispetto a quelle umane. Questo comportamento ha però il vantaggio di ampliare la conoscenza dell'ambiente e di portare quindi a trovare una soluzione migliore.

Conclusioni

La valutazione ha dimostrato che il modello è valido e riproduce con buona fedeltà il comportamento umano; che inoltre esso è in grado di imparare e, tramite la grande iniziativa che dimostra, in grado di avere prestazioni mediamente migliori di quelle umane.

Il difetto di questa implementazione è la necessità, per poter funzionare correttamente, di un ambiente molto particolare e con regole costruttive rigide.
Queste limitazioni si potrebbero risolvere dotando il robot di sensori più sofisticati.

Significativo è che il modello così costruito è totalmente indipendente dal tipo di sensori, quindi sarebbe teoricamente possibile implementare questo cambiamento senza apportare alcuna modifica al modello vero e proprio.

1. Abstract

Cognitive robotics is a fascinating field in its own right and comprises both key features of autonomicity and cognitive skills like learning behavior.

Cognitive Robotics aims at endowing the robot with intelligent behavior. An important property of a cognitive robot is the ability to learn and to behave in a complex scenario.

This is typically achieved by a cognitive architecture, which aims at mirroring human memory and assumptions about mental processes. One of the most widely recognized cognitive architectures is ACT-R, developed and tested by hundreds of researchers from Cognitive Science, Psychology, and AI [Act].

The major aspect of the proposed master thesis will consist of the analysis of necessary (pre-) conditions and subsequent implementation of the cognitive architecture ACT-R for mobile robots' control. The task of the robot will be to operate in an unknown environment, to search for a specific object and to execute specific operations depending on different objects. Towards this goal the robot must be able to explore space and, as it is a cognitive robot, to show learning behavior as provided by the cognitive architecture ACT-R. Especially, production rule compilation may provide a fruitful method.
In short, the robot must be able to:

- Explore (partially) unknown environment and search for a specific object of unknown position.
- Reach an object of known position, with or without any previous knowledge of the environment. The robot must be able to execute specific exploration strategies from AI or Cognitive Science (e.g. [RN03]).
- Show learning behavior. Possible test scenarios are the mazes used to investigate learning in rats [Tol48].

The robot will be built using as hardware a custom made Lego Mindstorms [Leg] and as software the ACT-R cognitive architecture. The Lego Mindstorms Robot has been used in several universities (e.g., MIT [KN00], RWTH, and numerous others). The software will be written using the LISP programming language.

2. Introduction

From the early beginning of robotics one line of research has tried to bring human cognition and robotics closer together [BBM+99].

Nowadays, technological progress in the field of robotics and the development of cognitive architectures allows for a leap forward: A robot able to navigate an environment, with the ability to learn and a human-like attention shift.

This new and exciting field is sometimes referred to as *Cognitive Robotics* [CG99].

This combination of two fields leads to a number of important research questions: What are the immediate advantages of cognitive robotics (a term we will use in the following for a robot controlled by a cognitive architecture) over classical robotics? Is the cognitive architecture (which is partially able to simulate human learning processes) restricting or improving navigation skills?
In cognitive science new research focuses on *embodied cognition*.

Embodied cognition claims that understanding (especially of spatial problems) is derived from the environment [And03].
In other words, cognition is not independent of its physical realization.
Taking as example the two labyrinths in Fig. 2.1, what are the minimal sensors necessary to navigate successfully through them?

The study described in [BHW09] used a virtual reality environment. Participants had to navigate through a labyrinth in the ego-perspective and find an initially specified goal (red dot). The study identified recurrent navigation strategies (which we will introduce later) used by the subjects.

Modeling navigation tasks, for instance in labyrinths, still poses a challenge for cognitive architectures: Although those architectures can model decision processes they typically abstract from metrical details of the world, from sensor-input, and from the integration processes of environmental input to actions (like move operations). Robotics, on the other hand, has captured all of these aspects, but does not necessarily make use of human-like learning and reasoning processes or even try to explain human errors or strategies.

Compared to humans a robotic agent has limited perceptions and capabilities.

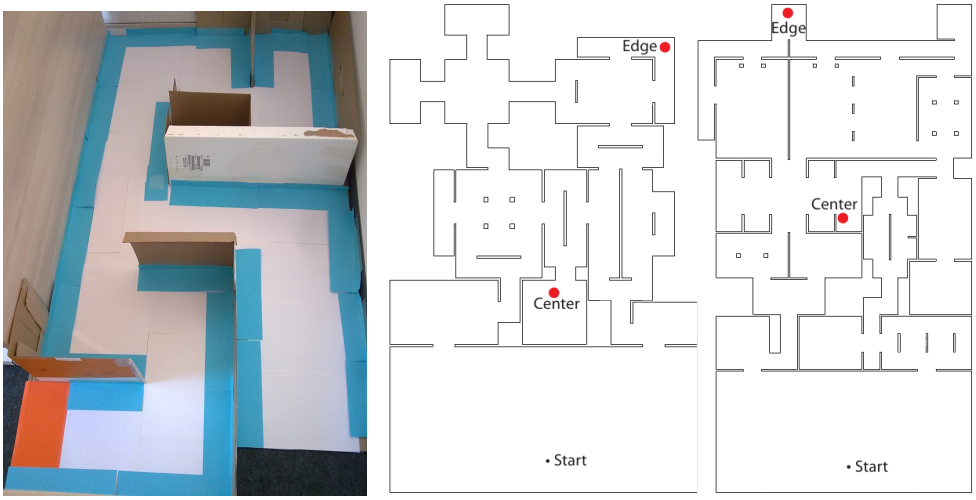


Figure 2.1.: **Left:** An example of a test labyrinth. **Right:** Layouts of two mazes: The task is to find the target object (red dot) at the edge or center [BHW09].

On the other hand, it can teach us something about the relevant perceptions that are already sufficient for the robotic agent to perform successfully. For instance, in the navigation task above (e.g., cf. Fig. 2.1), the distance from the wall (in the direction the robot is facing) and the color of the floor the robot is placed on are sufficient.

Much research is being done in the field of cognitive robotics; some prototypes of cognitive robots have already been built [1].

This research concentrates on the human-robot and robot-environment interaction, allowing the robots to recognize and interact with objects and people through their visual and auditory modules [ST+04]. The architecture proposed contains *Path Planning and Navigation routines* based on the Vector Field Histogram [BK91] that allow the robot to navigate avoiding obstacles and explore the environment.

Unlike the architecture proposed, the objective of this paper is to implement a *Cognitive Navigation System*, which is completely based on ACT-R and takes advantage of its cognitive features such as *Utility Learning*, that allows Reinforcement Learning [RN03, p.771] on productions.

No software other than ACT-R will be used to control the robot.

The previous experiment has the merit of having taken the first steps towards interfacing ACT-R with a mobile robot, but the data is still incomplete and tries to combine as many different abilities as possible (from natural language processing, to parallel computing) in a manner that is likely more complex than necessary.

[1] e.g., Cognitive Robotics with the architecture ACT-R/E http://www.nrl.navy.mil/aic/iss/aas/CognitiveRobots.php

Our approach starts at the other end, taking only the aspects and sensor data into account that are necessary to perform the task – the most simple robot.

Other research studied the possibility of interfacing ACT-R with a robot and giving it direct control over the robot's actions. That effort produced an extension of ACT-R called *ACT-R/E(mbodied)* [Tra09, HT10]. ACT-R/E contains some new modules that act as an interface between the cognitive model and the *Mobile-Dexterous-Social (MDS)* robot [BSB+08], allowing it to perceive the physical world through a video camera. However, no navigation was investigated, as the robot did not navigate an environment.

In both this and our implementation ACT-R has been extended. The vast difference between the two is the smaller amount of changes made to the standard ACT-R by our implementation, due to the sensors' higher complexity in the MDS.

2.1. Act-R

ACT-R is a cognitive architecture, a framework that models the structure and behavior of the human brain. This architecture tries to explain how all the brain's components collaborate and work together and form the human mind. The theory behind ACT-R is an unified theory and integrated system that tries to explain the overall behavior of the human mind through connections between its well-defined components.
The quote below comes from Allen Newell, the man whose work inspired J.R. Anderson in creating ACT-R, and explains the meaning of an integrated system.

> A single system (mind) produces all aspects of behavior. It is one mind that minds them all. Even if the mind has parts, modules, components, or whatever, they all mesh together to produce behavior. Any bit of behavior has causal tendrils that extend back through large parts of the total cognitive system before grounding in the environmental situation of some earlier times. If a theory covers only one part or component, it flirts with trouble from the start. It goes without saying that there are dissociations, independencies, impenetrabilities, and modularities. These all help to break the web of each bit of behavior being shaped by an unlimited set of antecedents. So they are important to understand and help to make that theory simple enough to use. But they do not remove the necessity of a theory that provides the total picture and explains the role of the parts and why they exist.

ACT-R not only offers a complete implementation of the theory, but also advanced features like automatic learning and quantitative predictions of the brain's behavior and activation. Many experiments with *Functional Magnetic Resonance Imaging (fMRI)* demonstrated that ACT-R can predict the *Blood-Oxygen-Level Dependence (BOLD)* response of several parts of the brain.

The main assumption of the theory behind ACT-R is that human knowledge can be divided into two irreducible kinds of representations: declarative and procedural.
Declarative memory refers to all the memories that can be consciously recalled, for example knowledge and facts memorized through experience, while procedural memory stores the information relative to how to do things: After repeating a sequence of actions that leads to a goal numerous times, this sequence is unified and memorized into the procedural memory. The next time the goal needs to be reached that knowledge will be retrieved and the actions done automatically, without the need to concentrate on what to do next.

A typical example that explains these two types of memory is the typewriter example. An expert in typewriting can type a text without looking at the keyboard; he knows the position of all the letters on the keyboard and can unconsciously access this information and touch type. The same expert, if questioned about the position of a defined letter on the keyboard, will have trouble in answering without looking at it.
The knowledge of the letters' position cannot be accessed consciously, in fact the process of typing is done automatically, without need of concentrating on the task. This knowledge is stored in the procedural memory: it does not contain a piece of information but the action (which finger and where to move it) needed to reach the goal of typing a letter.

Declarative knowledge is represented in the ACT-R's theory by *chunks*: data structures in the shape of a n-tuple of arbitrary size that contain information related to a specific fact or memory. A chunk is characterized by its *chunk-type*, that defines each tuple as a list of two elements: the first identifies the slot and the second its value.
Procedural knowledge is represented by *productions*, which are the equivalent of functions in ACT-R and contain a sequence of actions and operations on buffers, which produce determined effects and can take place only if determined preconditions are satisfied.

The ACT-R is a framework, structured in *modules* (as shown in Fig. 2.2), each module represents one or more than one function of the brain.
The visual module represents the visual cortex and is responsible for identifying the objects in the visual scene and shifting the attention on to them; the motor module controls the virtual hands, like the vocal module controls the voice, and can perform actions like pressing a button or moving the mouse.

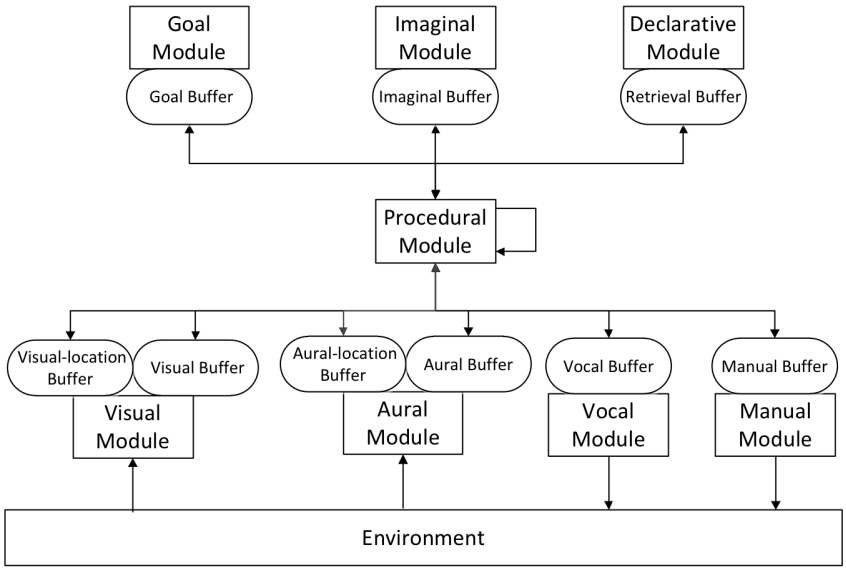


Figure 2.2.: The modular structure of ACT-R.

The brain's neural functions are localized: all the neurons that perform similar tasks are close together, for tha sake of ease in communication between neurons.
This implies that if each function is controlled by a determined brain region, each module must represent one of the brain's regions as well. The procedural module is the central part of the architecture and is responsible for the coordination of all the other modules. It controls the whole system through the exchange of information between the *buffers*.

Each module is independent from the others; modules do not share variables or information. Each module can communicate with the others through the use of buffers, which are its interface towards the other modules. A module can have no buffers. A module can read chunks from every buffer, but it is supposed to make changes only to the chunks in its own buffers. This typically happens when the module responds to a query or a request, creating of a new chunk in its buffer.

The processes inside the modules are parallelized; for example the visual module can keep track of many objects in the visual scene and the declarative module looks for a match between a huge number of records. The interaction between modules is, instead, strictly serial. There are two main reasons that cause this bottleneck:
The first reason is that each buffer can contain, in a specified moment, only one chunk. To store a new chunk in a buffer the previous one must be deleted, so for every cycle it is possible to read only one piece of information.
The second reason is due to the procedural system. For every cycle several productions could have the preconditions to be selected and fired by the procedural module.

In reality, because of the seriation in production execution, only one production can be fired at a time. The production to be selected is the one with higher *utility value*. This value is calculated, for every production, on the basis of several parameters such as the estimated probability of reaching the goal, its current value and the estimated time needed to achieve it. Those values are at the base of the learning mechanism embedded in ACT-R: the more successfully a production is, the more its estimated probability and so its utility grow, increasing the probability for that production to be selected again.

ACT-R is written in Lisp, and offers its own lisp-like language. Its modular structure allows the developer to write his own modules and add them effortlessly to the framework. The high flexibility of the framework makes it easy to extend the theory behind it.

A researcher can program the framework by writing models; each model represents one task and specifies the idea that the modeler has about the cognition behind that task. A model contains several productions that interact with the modules, querying them and reading their buffers.

2.2. The objective

The final objective of this work is to control a robot through ACT-R. To reach this goal an interface must first be written through ACT-R and the robot. This interface will be responsible for translating the commands in the model into lisp instructions.

The second step will consist of writing a model that will allow the robot to show human-like behavior while navigating through a labyrinth and searching for a goal, e.g., an exit from the maze.

For human-like behavior we mean the ability of creating a mental representation of the environment and adapting it to the new information that comes from the exploration. The model should also be able to learn from the experience and improve, run after run, its performance in finding the solution. Some reasoning should also be shown by the navigation algorithm, for example the ability to recognize a undesired situation, like a dead-end or a loop, and avoid it.

The model is going to be tested in several situations and its performance will be compared with the data from [BHW09], relative to an equivalent experiment previously done with humans subjects. The results of the two experiments should be somehow related and should show some point of contact that will prove the validity of the model in simulating the human behavior.

3. Robot

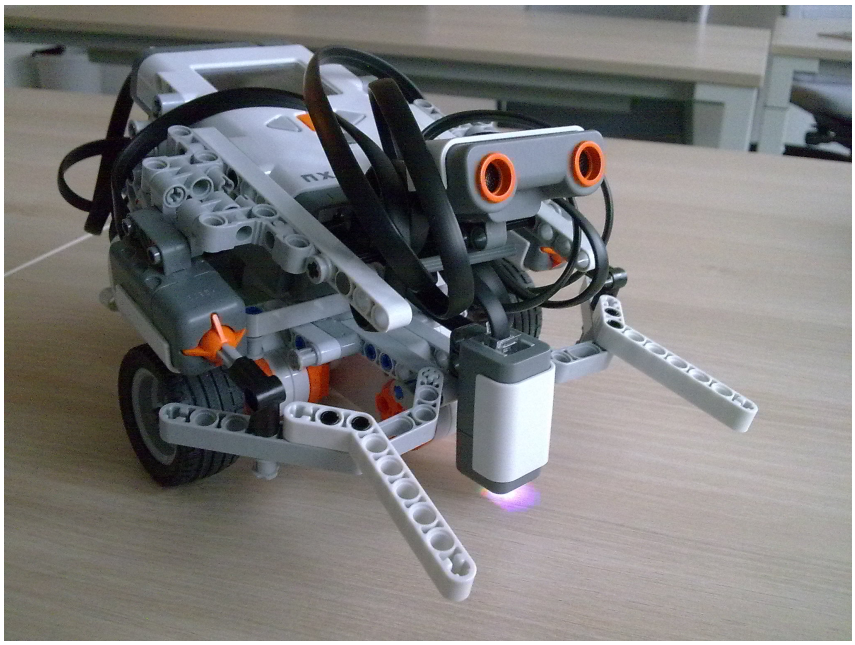


Figure 3.1.: The Mindstorms class robot. It is equipped with a color, ultrasonic and two touch sensors.

Our device of choice is a standard Mindstorms [1] class robot (cf. Fig. 3.1).
It consists of a central "brick" that contains the CPU and the batteries, it also features a small speaker, through which the robot can play some audio files, a microphone and a small display that reports the state of the robot and can be also used to navigate in the menus, through four directional buttons, or to visualize some graphics. This brick was designed in collaboration with the MIT.

At this core component several peripherals can be attached: it supports up to three step-to-step engines and up to four sensors. The robot is sold in a box that contains LEGO bricks, some sensors and the programming software.
The included software allows the user to program the robot with a graphical and easy to

[1] This type of robot is produced by The Lego Group

use interface: the programming consists of linking a series of virtual bricks together to create the desired behavior. Each brick represent an abstract entity, for example a logical port, a loop with condition, a sensor, the display, an engine, etc.

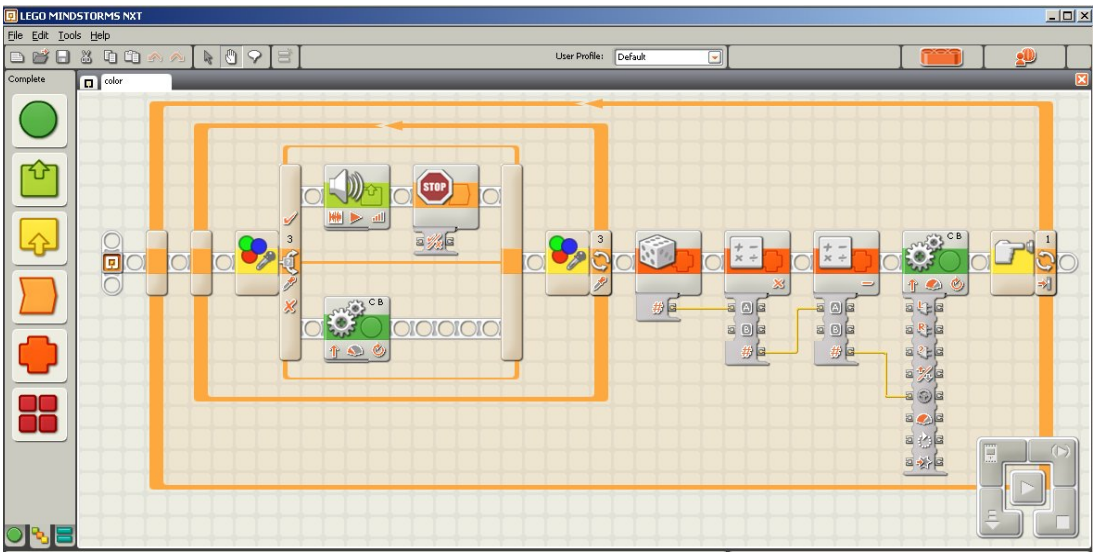


Figure 3.2.: The National Instruments' LabView development system.

The bricks have some properties that allows the programmer to set some parameters (e.g input values) and some output values can communicate some values to the next bricks in the sequence.
Such a programming IDE is a great tool to teach kids to program, but for our goals it is not sufficient.

The community developed many substitutes of this tool, that install a new firmware in the robot and allow the programmer to write its code in a programming language of common use. The most popular are the Java environment leJOS [2] and the C++ environment RobotC [3].

[2]leJOS: Java for LEGO Mindstorms http://lejos.sourceforge.net/
[3]RobotC: a C programming language for Robotics http://www.robotc.net/

3.1. Chassis design.

The chassis is the structure to which the central brick, the motors and the sensors are attached.
Our robot has a custom chassis based on the standard chassis proposed by The Lego Company for the use with the Mindstorms robot. The base design has been modified to room all the needed sensors and to respond to some design choices.
For example, instead of the suggested configuration with two caterpillar tracks, three wheels compose the robot's locomotion system (as seen in Fig. 3.3). Tracks are useful for off-road operations because avoid skidding, but in an indoor environment wheels are an appropriate choice. The use of two independent driving wheels helps decreasing the odometry errors, thanks to the smaller friction that the rubber wheel produce on the paper floor compared to the tracks.

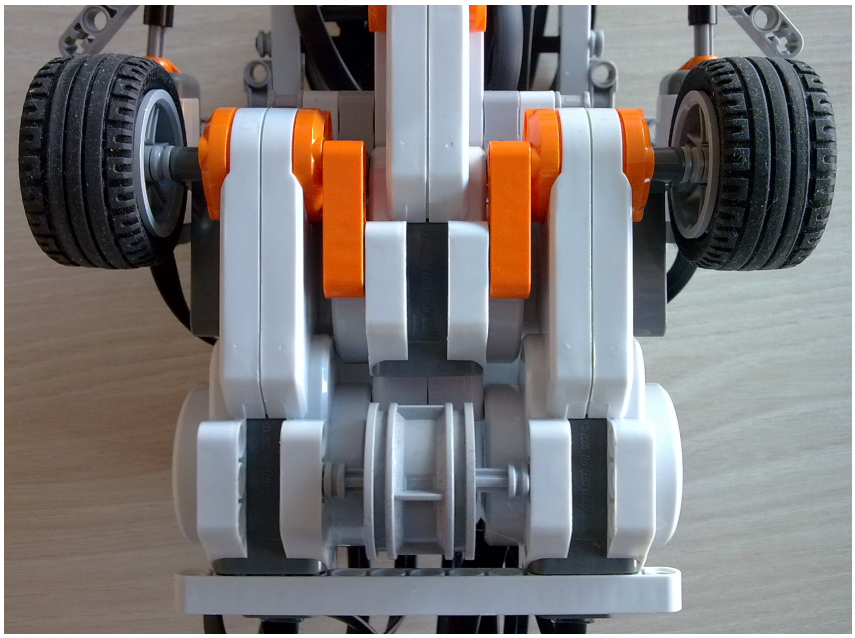


Figure 3.3.: The driving system.

A third central fixed idle wheel, placed on the back of the chassis, allows the robot to keep its balance.
The structure resembles a differential drive robot, with the exception that the third wheel cannot pivot. This compromise works well with the expedient of removing the tyre from the back wheel. The small friction between the floor surface and a small portion of the plastic wheel does not produce much of a friction and allows the robot to turn without too many difficulties.

3.2. Sensors.

A robot can be equipped with several kinds of sensors: from a simple sound or light sensor, to more complex ones like a compass or webcam. In addition to the standard sensors produced by LEGO, there are on the market several advanced sensors produced by third parties that are fully compatible with the NXT brick.

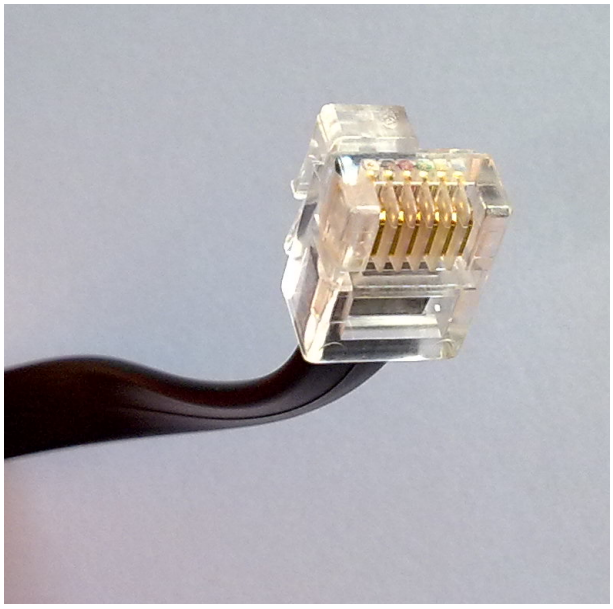


Figure 3.4.: The NXT sensor's 6-position modular connector.

Every sensor has the same physical interface towards the central brick (Fig. 3.4), this interface includes both an analogic and a digital communication channel, besides that the power supply. Thanks to this universal interface the sensors are interchangeable and can be attached to any port.
The engines have, on the other hand, three dedicated ports.

Following the modular approach of ACT-R, we decided to start bottom-up, using only the most necessary sensors to perform adequately. Our design makes only use of the most basic sensors:

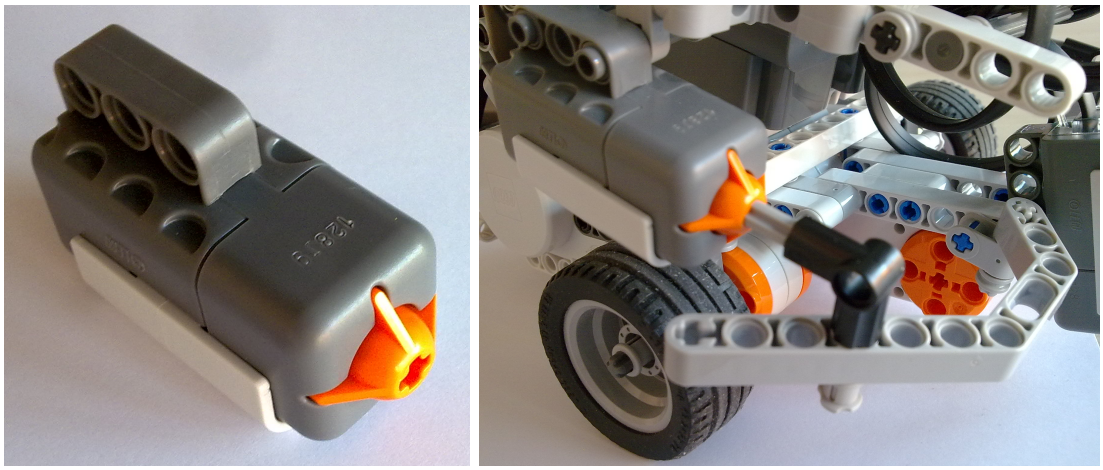


Figure 3.5.: Left: The touch sensor - Right: the structure it is linked to.

Two *touch sensors* (Fig. 3.5), they are nothing more than a switch that is activated every time that its top part is pressed. It has been used to have short range perceptions, as a security device that stops the movement when the robot hurts a wall.
Each touch sensor is placed on one side of the chassis and is linked to a mobile structure that covers the front of the robot on that side. When this structure hits an obstacle the sensor is activated. This allows it to identify obstacles within a range of 0 to 80 degrees, on both sides.

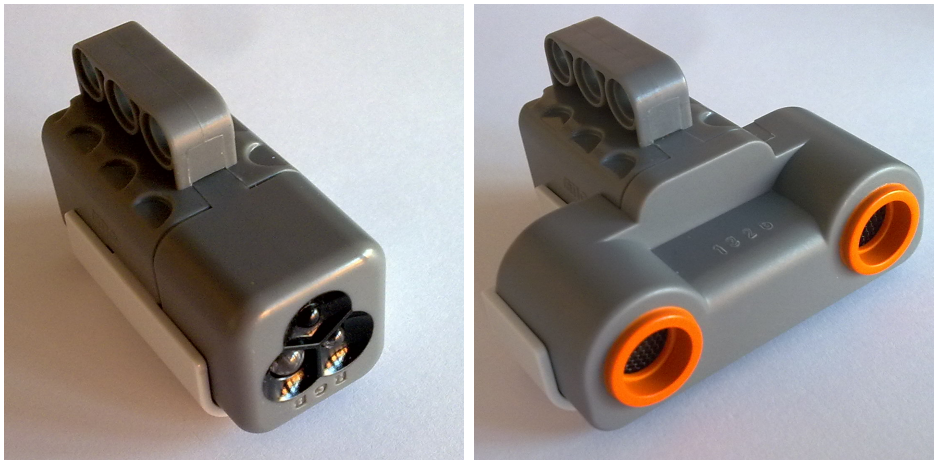


Figure 3.6.: Left: The color sensor - Right: The ultrasonic sensor.

A *color sensor* (Fig. 3.6 Left) that is composed by three LED forming a RGB light that can produce monochromatic light or white light, which is used to illuminate the target, and by a light sensor that measures the light intensity.
It is placed on the front part of the chassis and looks at the floor just in front of the robot. It has been used to read the floor's color, whose variation identify an obstacle nearby.
The sensor has two modes of operation:

- Black and White: The LED produces a monochromatic light that is used to illuminate the environment. The sensor reads the light intensity and gives as output a numerical value between 0 and 1023.
- Color: The LED produces a white light and the sensor reads the color of the reflexed light. In this mode of operations the sensor can discriminate between six tonalities of color. To work properly the sensor must form a right angle with the target and be not more that three centimeters far away.

An *ultrasonic sensor* (Fig. 3.6 Right), placed on the top of the chassis, measures the distance between the robot and the next obstacle in front of it. The sensor cannot pivot, so the robot must turn itself to measure the distance in a different direction.
It consists of a speaker that emits ultrasounds and a microphone that receives them.
The sensor compute the distance to the next obstacle applying a proportion on difference between the time at which the ultrasound has been sent and the time at which its echo has been received. The operational range goes from about 5 to 230 centimeters, with a declared precision of 3 centimeters.

Those last two sensors give the robot long-range perception, allowing it to obtain information about a part of the environment different from the point it is placed in that very moment.

On the back there are no sensors. This choice works in the hypothesis that the robot does not move backwards.

4. Environment

4.1. The Perceptions

Given the sensors described in the previous chapter, the agent is capable of having perceptions about the environment it is placed in.
Through those perceptions the agent can obtain some kind of information about some characteristics of the surrounding environment. The robot can realize, through a query to its touch sensors, if it is touching a wall. The sensors give information only about the front of the robot, while they cannot tell if the side is scraping against a wall.

The color sensor is able to recognize landmarks nearby. The landmarks are marked with a stripe of colored paper surrounding them, whose color the sensor decodes. The sensor can discriminate between 6 tonalities of color, from white to black:

	White is used to identify a clear way, with no landmarks in sight the robot will keep going forward. This color has been chosen because it assures the less possible degree of error: to read white, the sensor must be in a ideal situation (good brightness, perfectly white surface at less than 3 centimeters from the sensor). When a reading error happens the sensor can only report a darker tonality, in that case the algorithm will stop, preventing dangerous situations to occur.
	Blue is the last color before black, it has been chosen for representing a wall. This color assures the highest contrast possible, making at least likely to happen, because of a reading error, that the sensor reports erroneously a wall instead of a clear way.
	Red identifies the goal. It is in the middle between white and blue and so it assures the highest possible contrast and the smallest chance of error.
	Yellow identifies a junction. This color can be easily mistaken with White (even if the empirical experience proved that the most reading errors report the color Green) but that is not a major problem: the behavior programmed in response to Yellow don't involve possibly dangerous actions, and can recognize and handle such a reading error. This behavior will be deeply explained in the next chapters.

The ultrasonic sensor is used for long range perception, it measures the distance to the object that, typically, is the next wall. This feature is used to measure the length of the corridors that branch off a junction.

Given that set of perceptions, the agent must recognize and avoid obstacles in order to explore the environment, and take good decisions when it encounters a junction.
It is important to keep in mind that the robot has no a priori knowledge of the environment it is operating in. At the beginning the agent has no knowledge at all about it, the only information about the environment is the embedded knowledge of how it looks like: of what color are the walls, the goal and the junctions, how a junction is formed, etc.

4.2. The labyrinth

We decided to test the cognitive robot on a custom self-built labyrinth.
To permit the robot to have the perceptions described above, the environment has been build following some criteria that make errors less likely to happen (e.g., cf. Fig. 2.1).

The Labyrinth is built using cardboard, a material that is easy to find and manipulate. This solution permit rapid changes in the labyrinth's design.
The walls are taller than the robot and thick enough to resist an impact. That allows the ultrasonic sensor to read the distance from the wall and also to put up enough resistance and let the touch sensors operate.

The floor is covered in white paper. Paper is a good surface for odometry purposes and the white color is not common in a normal environment, this forces the model to work only when it is inside the labyrinth.
Blue paper, for a length of about 5 cm, is glued on the floor around the walls. When the robot is approaching a wall its color sensor will find the blue paper some seconds in advance of actually hurting the wall. The color sensor will recognize that change and report the danger to the navigation system, that will act to avoid the collision.

4.3. Properties of the environment

Before going forward we should define in a clear way the properties of the environment. The classification of the properties is taken from [RN03, p.41].

- The environment is *static*, so it does not change during the execution. Information about the surroundings is obtained by the agent through its sensors, as perceptions. Those perceptions stay valid for the whole execution.
- The environment is although *partially observable*, in fact the sensors cannot inform the agent about its orientation. The absolute orientation is anyways not necessary to navigate, is enough for the agent to know what is its orientation in relation to the environment. To compensate for this, the agent assumes to be observing north at the start of the exploration, and keeping track of the changes of its orientation in its internal state.
- Since the robot operates and moves freely into real space, the environment is to be considered *continuous*. A natural consequence of this characteristic is the presence of uncertainty and errors in perceptions.
- Because of this uncertainty the environment may appear *stochastic* to the agent.
- The tasks in this environment are *sequential*, that means that each action taken might have consequences on future decisions. This consideration is straightforward from the task of exploration: the agent cannot decide to move to an arbitrary place in the environment but is compelled to move from one position to the one physically next to it. Choosing a direction compared to another one excludes automatically a whole set of possible states, that could be accessed by choosing the other direction.
- Concluding, the environment is *safely explorable*: The agent can freely explore the environment without the risk of coming into a state that comports a loss. A more formal way to say that is that the goal can be reached from any possible state.

Such an environment determines that new information can be only obtained, through perceptions, after performing an action. Collecting of new information is subordinate to acting. This situation is a classical example of contingency problem [RN03, p.86].
For this class of problems the solution takes the form of a tree, whose branches identifies the possible alternatives that can be chosen depending on the received perceptions.
Each branch is associated with a condition that must be met in order to select it.

One example of this structure is a state in which the action of turning right can be performed only if the perceptions tell that there is no wall on that side.
Every branch has also a probability associated to it, in case more than one action can be performed from the same state this probability will be different from one and reflect the chance of the action to happen.

To reach the goal, the algorithm must solve an *exploration problem*. In this kind of problem the agent does not know what are the states and to what results its actions could bring to.
The way to solve such a problem is to observe what effects the actions bring and use that observation to decide what to do next. This execution, that interleaves an execution phase with an decision phase, is called *Online Search* [RN03, p.122].

5. Interface

The first step towards a *cognitive robot* is to create a working interface between the ACT-R framework and the NXT platform. This interface allows an ACT-R model to control the robot and to receive sensor inputs from the robot. Due to the modular structure of ACT-R, this interface is composed by several modules with different functions.

5.1. Low level lisp function.

On the basis of these modules there is a library [1] that provides low-level lisp functions to execute simple tasks like interrogate a sensor or activate a motor.
The capabilities of this library are the following:

- It can read information about the environment from the light sensor, the distance sensor and the touch sensor.
- It can make the robot perform some actions, like move its motors, turn on and off the light and play some sounds.
- It can also check the robot's internal state, querying the battery or the motors about their states.

The library did not have the support for the color sensor, so it has been extended to support that sensor, necessary for our purposes, as well. The lisp library has been bundled in the code and its functions are called by these modules to control the robot.

[1] NXT-LSP: the NXT Controller in Lisp, developed by Tasuku Hiraishi http://super.para.media.kyoto-u.ac.jp/~tasuku/index-e.html

5.2. Module definition

The modules have a common structure. This is a standard module definition.

```
(define-module-fct 'name
                   '((buffer nil nil (query1 query2) nil))
(parameters))
```

Every module has an unique name, needed for its identification, it is the first parameter in the *define-module-fct* function. A module can have zero or more buffers, which are used to communicate with other modules though an exchange of chunks, whose names are written in a list in the second parameter.

If one of those elements names a list, not only the name but also the default values for the buffer's components are specified. The list can be up to five elements long, only the few that were useful for our purpose are going to be briefly handled, for a complete reference see the ACT-R Reference Manual.
The first element is the buffer's name and the fourth is the list of queries that the module will accept along with the standard "state" query.
The third element is the list of the module's parameters, for each one a define-parameter function is called.

```
function define-parameter returns [parameter | nil]
( param-name {:owner owner} {:default-value default}
{:documentation docs} {:valid-test test} {:warning warn} )
```

Every parameter has a unique name. There cannot be two parameters, even in two different modules, with the same name. Each parameter can have a documentation and a default value. Is also possible to specify a function that will test the validity of a new value, in case the parameter's value is changed by the modeler.

Every module has a set of functions that are called in response to some ACT-R events.

```
:request 'nxt-motor-requests
:query 'nxt-motor-queries
:creation 'nxt-motor-create
:reset 'nxt-motor-reset
:delete 'nxt-motor-delete
:params 'nxt-motor-params
```

5.2.1. Creation, reset and delete

Those functions are called by ACT-R when the module is created, reseted or deleted. They only have the task to initialize the module or destroy it.

5.2.2. Request

The request function is called every time that the model makes a request to one of the module's buffers.
It must accept three parameters:

- The *instance* of the module, it can be used to reference to the module's parameters.
- The *name* of the buffer. In case the module has more than one buffer, this is useful to know on which one of them the request was made to.
- The *chunk-spec*, the object contained into the buffer and, at the same time, the request itself. Testing the slots of this chunk the model can understand which kind of request it has been made.

There is an example of a request in a model:

```
+buffer-name>
    isa         chunk
    slot1       value1
    slot2       value2
```

5.2.3. Query

The query function is called when the model makes a query to the module.
A query does not imply an exchange of chunks, even if it is made on a specific buffer, it is only supposed to test something and return a boolean. A query gives back a boolean value, so it can be used to trigger a production.

Every module must respond to the query on its state, but it can also have other kinds of queries. For every buffer a module can respond to an indefinite number of queries.
Normally a module accepts only the standard "state" query. To tell it to accept more queries, it is necessary to write their names in the module's definition, as explained earlier in this chapter.

This functions must accept four parameters: the first one is the instance of the module, the second one is the name of the buffer, the third one is the name of the query and the fourth one is the value of this query.

5.2.4. Params

This function is used to modify the module's parameters.
It accepts only two parameters: the instance of the module and the the value to be modified.
If the second value is an element, it represents the name of the parameter. In this case the function will return the value of the specified parameter. If it is a list, the head will represents the name of the parameter and the body the new value to be set.
The modeler can get or set a parameter outside the module's definition using the *SGP* function.

5.3. Design studies

The design of the interface is of crucial importance for the overall project: a good design of the interface will bring not only better performance and capabilities in the modules themselves, but also better performance for the ACT-R model.

The first consideration has been to respect the common ACT-R design:
all the pre-existent modules had some common design rules that have been replicated in the new built modules. These rules include the standard "state" query and the possibility of both querying and modifying the parameters, that have been kept even if they were unnecessary.
An uniform naming rule has been applied to all the names, the prefix "nxt" identifies buffers and parameters owned by the nxt-specific modules.

Some other design choices have been taken, based on the common sense of good design: all the parameters have validity tests for type and range and their values can only be accessed through a query to the module, the chunk-types needed by one module have been kept private and each query produce an error message if the input value is malformed.

Each module takes care of one specific sensor and provides all the functions needed by it to operate. The commands to the sensor must be sent as a request to the buffer, or the buffers, owned by the module, a request is made placing a well-formed chunk in the buffer. The module responds to a valid query placing in its buffer a new chunk that contains the answer to the query.

A response is not mandatory, a module can just do what is requested to it and give no answer: this is the case of the query to the motors, there is no need for an answer because the result is deterministic and well visible through direct observation.

A special case of implementation is the touch module: no requests can be made to its buffer, it can only be queried about its status. The decision of not providing any request comes from the functioning of the touch sensor: the sensor cannot be driven or even turned on or off, its functioning is completely passive.
Since there is no need for giving any input to the sensor, there is consequently no need to make any request to the module. Every time that the model needs to query the touch sensor, will be enough to query the module and read the state of the sensors.

The design of the single modules is inspired by the design of the standard ACT-R modules that have similar functions: for example the module that controls the motors operates in a similar fashion as the ACT-R's motor module, responsible for the hand movements like typing or moving the mouse cursor. The sensor that controls the color sensor operates, as much as possible, like the ACT-R's vision module.

The standard ACT-R visual system is composed by two parts: the visual-location buffer accept requests containing some constraints about the visual object to find, like the position in the visual scene. Following these constraints the vision module look for a matching object in the visual scene.
The visual scene is represented in ACT-R by the "Visicon", that stands for Visual Icon, a real or a virtual device on where visual objects can be inserted and retrieved.

The vision module in addition operates a process called *buffer-stuffing*: every time that a new object appears on the visual scene the module creates automatically a new chunk in the visual-location buffer that contains a description of that object.
This system allows the model to react quickly to a new visual stimulus.

After a new object appears on the visual scene and is recognized by the visual module, the modeler can decide whether the event is worth of attention or not. In case the event is significant the model can send a request to the visual buffer, asking it to shift the attention to the new object and give some more information about it.
The vision module can concentrate its attention only on a single object at a time and every shift must be requested through the visual buffer, giving to it the appropriate visual-location chunk.
The vision module responds to an attention shift placing a new "visual-object" chunk in the visual buffer. This chunk contains more information about a defined visual object, like its color, the type of the object, its dimensions and its position on the screen.

Our implementation follows this pattern and overrides only the first half of the process.

```
(defun poll-sensor (instance)
        (setq port (nxt-vision-port instance))
        (let* ((number (colorsensor (nxt-vision-port instance)))
                (color (case number
                        (1 'black)
                        (2 'blue)
                        (3 'green)
                        (4 'yellow)
                        (5 'red)
                        (6 'white)
                        (t 'black))))
        (if (or (not (eq color *previous-color*)) ;if the color changes
                        (eq number 2)) ;of if it-s blue
                (let ()
                        (clear-exp-window)
                        (add-text-to-exp-window :text "O" :color color)
                        (proc-display :clear t)))
        (setq *previous-color* color)
        );let
)
```

The color sensor starts its operation after a request to the nxt-visual buffer is issued. The module respond to the request turning the sensor on and starting a new process that polls at a constant frequency the color sensor. When a change in the color is recognized, or the sensor reports the color blue that identifies an emergency situation, a new visual object is written on the Visicon. The information about the color read by the sensor is forwarded by the color of the newly created visual object.
After the object is drawn on the visual scene, the procedure to refresh the screen is called. The vision module will then rescan the visual scene and find the new object, in response to this event the module will do buffer stuffing and communicate to the module this visual change. The visual process will then go on as normal, with an attention shift and a subsequent analysis of the object.

5.4. Module implementation

5.4.1. nxt-motor

This module controls the engines, when a request is made to its buffer the module responds activating or deactivating the specified motors. This module assumes that normally two motors are linked, called left and right motors, and optionally a third motor can be linked and controlled separately. The module needs to know to what ports the engines are linked, it uses as default values the standard diagram suggested by LEGO. If the modeler wants to change the disposition of the engines, he must change the corresponding parameter through the *SGP* function.

```
(define-parameter :c-engine
        :documentation "nxt's third engine, normally not used for movement"
        :default-value #\a
        :valid-test (lambda (x) (characterp x))
        :warning "a character"
        :owner t)
(define-parameter :r-engine
        :documentation "nxt's right engine"
        :default-value #\b
        :valid-test (lambda (x) (characterp x))
        :warning "a character"
        :owner t)
(define-parameter :l-engine
        :documentation "nxt's left engine"
        :default-value #\c
        :valid-test (lambda (x) (characterp x))
        :warning "a character"
        :owner t)
```

There are three parameters, one for every engine, that encode the port to which the engine is registered. The parameters *r-engine* and *l-engine* correspond to the right and left engines, used for the robot's locomotion. All the movement functions in this module work on the two engines registered in these parameters.
The third parameter, *c-engine* is the auxiliary engine that can be used to control other functions, like a claw to pick up objects. These three parameters accept as value a character, the valid values are 'a', 'b' or 'c'.

The last parameter, *velocity*, accepts an integer that specifies the engines' velocity.

```
(define-parameter :velocity
        :documentation "max rotation velocity"
        :default-value *engine-velocity*
        :valid-test (lambda (x) (integerp x))
        :warning "an integer"
        :owner t)
```

This module has only one buffer called *nxt-move* that is used only to receive chunks, that are created by the modeler for controlling the engines.
These chunks order a movement for a certain duration, specified by the duration slot.

```
(chunk-type move-forward duration)
(chunk-type move-backwards duration)
(chunk-type turn-right duration)
(chunk-type turn-left duration)
```

The *activate-motor* chunk can be used to control a single engine, the chunk must contain in its slots a valid port, velocity, direction (t means forwards and *nil* means backwards) and duration values.

```
(chunk-type activate-motor port velocity direction duration)
```

The last commands are used to interrupt a movement: The *deactivate-motor* chunk stops the engine specified by the port in its slot, while the *emergency-stop* chunk brakes all the engines at the same time.

```
(chunk-type deactivate-motor port)
(chunk-type emergency-stop)
```

5.4.2. nxt-touch

This module controls the touch sensors, it can control up to four sensors but the default value is two. The value of the ports can be changed through a parameter request.
The only parameter of this module is called touch-port and it contains a list of numbers, between 1 and 4, that identifies the port to which the sensor is connected.

```
(define-parameter :touch-port
        :documentation "port to which the sensors are connected"
        :default-value '(1 2)
        :valid-test (lambda (x)
                (and
                (listp x) ;must be a list
                (dolist (var x t)
                        (if (or (not (numberp var)) (< var 1) (> var 4))
                                (return nil)) ;must be a number between 1 and 4
                )))
        :warning "a list of valid ports"
        :owner t)
```

This module has a buffer called nxt-touched, but it is only used to query the module:

```
(touched ;did it touch something?
        (setq result
                (dolist (var (nxt-touch-port nxt)) ;for every sensor
                        (if (touched? var) ;if it's pressed
                                (return t))
                );dolist
        );setq
        (case value
                (true result) ;true if it touched
                (false (not result)) ;false if it touched
                (t (print-warning "Bad state query to the ~s buffer" buffer))
        );case
);touched
```

If the nxt-touched buffer receives a query of type *touched*, with value true, it returns true if the sensor reports a pressure, if the value is false the answer is its logic negation.

```
(sensor ;did the specified sensor touch something?
        (if (member value (nxt-touch-port nxt))  ;checks if the specified port
   has a sensor attached
                (touched? value); reads from the sensor
                (print-warning "No touch sensor defined for port ~s, define a
   new sensor first" value)
        );if
);sensor
```

If the query has type *sensor*, with value a number that corresponds to a valid port number, it returns true if the sensor on that port reports a pressure.

```
(sensors ;are ALL the specified sensors activated? it must be a list!
        (dolist (var value t)
                (if (member var (nxt-touch-port nxt)) ;if the port is valid
                        (if (not (touched? var)) ;if one sensor is not pressed
                                (return nil)) ;finish and return nil
                        (return (print-warning "No touch sensor defined for
   port ~s, define a new sensor first" var)) ;print-warning returns nil
                );if
        );dolist
)
```

If the query is a *sensors*, with value a list of numbers that corresponds to a list of valid port numbers, it returns true if all the sensors (logic-AND) on those ports report a pressure.

5.4.3. nxt-vision

This module controls the color sensor. The default value of the port is the one suggested by LEGO, but it can be changed through a parameter request.
Its only parameter, called visual-port, contains a number between 1 and 4.

```
(define-parameter :vision-port
        :documentation "port to which the sensor is connected"
        :default-value 3
        :valid-test (lambda (x)
                (and (integerp x) (> x 0) (< x 5)))
        :warning "a valid port number"
        :owner t)
```

This module has a buffer called nxt-visual, but it does have no other purpose than to query the module.

```
(light ;used to turn on and off the sensor
        (case (third (car (chunk-spec-slot-spec chunk-spec 'turn)))
                (on (colorsensor-white (nxt-vision-port instance))
                        (setq polling-event
                        (schedule-periodic-event 50 'poll-sensor
                        :maintenance t :time-in-ms t :params (list instance)))
                )
                (off (colorsensor-off (nxt-vision-port instance))
                        (delete-event polling-event) ;stop polling
                )
                (t (print-warning "Wrong parameter for the \"light\" request"))
        );case
);light
```

If the nxt-vision buffer receives a query of type *light*, with value on, it will turn the sensor on and fork a new process that will run the polling function every 50ms [2] to check the sensor's state. If the value is off, the sensor will be turned off and the polling stopped. When the sensor identifies a change of color from the previous state, or the color blue, it draws on the visicon a letter of the same color. This action triggers a standard ACT-R procedure that will allow the model to realize that a change happened.
This solution fits perfectly into the ACT-R structure and allows the modeler to write a model that's independent from the kind of visual input.

[2]

5.4.4. nxt-distance

This module controls the ultrasonic sensor. The default value of the port is the one suggested by LEGO, but it can be changed through a parameter request.

```
(define-parameter :distance-port
        :documentation "port to which the sensor is connected"
        :default-value 4
        :valid-test (lambda (x)
                (and (integerp x) (> x 0) (< x 5)))
        :warning "a valid port number"
        :owner t)
```

The only parameter of this module is called distance-port and it contains a number, between 1 and 4, that identifies the port to which the sensor is connected.

```
(obstacle
        (EVAL (READ-FROM-STRING (format nil
                "(set-chunk-slot-value read distance ~D)"
                (distance (nxt-distance-port instance)))))
        (EVAL (READ-FROM-STRING (format nil
                "(set-chunk-slot-value read counter ~D)"
                (third (car (chunk-spec-slot-spec chunk-spec 'counter))))))
        ;scheduling the action to fill the buffers
        (schedule-event-relative 0.1 'set-buffer-chunk
                :priority :min
                :params '(nxt-distance read :requested t) ;put into the $1
   buffer the $2 chunk
                :output nil)
);obstacle
```

This module has a buffer called nxt-distance. When a request is made to this buffer, with a chunk of type obstacle, the module reads the distance from the sensor and updates the chunk in its buffer with the distance value returned by the sensor. This system takes two production to obtain the value, the first production must issue the request and the second can read the result from the nxt-distance buffer.

6. Model

6.1. The Model

Our first approach was to write a cognitive model that would use the perimeter strategy to navigate the labyrinth, replicating the experiment previously conducted with human subjects described in [BHW09].

The model was provided with three exploration strategies:

- The *right perimeter strategy* gives the model a right turn preference for all situations. When the agent meets a junction it first looks to its right for a free path which it follows id present; if the path is not free (e.g.. a wall is detected) it will continue on straight ahead if possible, and only as a worst case scenario it will turn left.
- The *left perimeter strategy* is identical to the right perimeter strategy, but with a preference for left turns.
- The *random walk* strategy chooses a direction at random until a free path is found.

To begin, all three strategies had equal probabilities of being selected by the model. Later these probabilities were modified by the model itself, based on the performance of the respective exploration strategy.

The *utility learning* function embedded in ACT-R, based on this learning process. There is a utility value associated with each strategy that records its performance. Associated with each goal is a "reward" that is handed out after the goal is reached, this reward is divided among the strategies used to reach the goal and is proportionate to the number of times each strategy was used and the duration of this use. If reaching the goal takes too long, the reward is negative and the strategies are penalized.

Several tests analyzed the performance of this model. The tests were executed in three special labyrinths: in the first one the goal was situated in the top-right part and the best path required turning right at all times. The second was identical the first mirrored on its vertical axis. In the last labyrinth the goal was two steps forwards from the start. Clearly each of these labyrinths catered to only one strategy as much as possible.

The model showed the ability to learn: in the 60% of the cases the correct strategy was chosen. This value is surprisingly low, considering the type of environment used. Furthermore, in the final labyrinth, the random walk strategy was chosen about 50% of the time. The other 50% the perimeter strategy was erroneously chosen. In practice the perimeter strategy finds a suboptimal solution, so the algorithm is technically not wrong; the problem is that, because of the aleatory nature of the strategy, random walk can lead to worse results than the deterministic perimeter strategy.

Another important disadvantage of this solution is the lack of adaptability: if, for example, the optimal path consists of N turns right and one turn left, the right perimeter strategy will be selected and so the last turn left will never be taken. This situation happens, because this approach uses a single strategy for the whole run, while the optimal path could be divided in several parts each of which needs a different strategy.

These considerations led us abandon this approach in favor a new, more flexible system.

6.2. A revised approach

To improve upon the approach described above, the idea was to shift the utility value from the strategy to the state: Instead of executing the best rated strategy in all situations, the best action for the current state should be chosen. The entire set of actions executed along the optimal path forms the best strategy for a particular maze.

For the second approach the model was refined: an *Active Reinforcement Learning* algorithm was implemented [RN03, p.771]. The task of Reinforcement Learning is to use the feedback, or *reward*, given by an evaluation function to discriminate between positive and negative actions. The ability of distinguish between good and bad is on the basis of learning a (sub-)optimal policy. The term *policy* refers to a solution that specifies, for every state, what action the agent must perform to reach the goal.

A task of Reinforcement Learning is *Active* if the policy is not fixed: the agent has a choice of action and must decide what to do. In our case the agent is supposed to follow the most promising path through the labyrinth and, when necessary, update the policy to a better performing one. The consecutive updates will eventually lead to the optimal policy, that can be easily found by solving the *Bellman equations* [RN03, p.620].

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s')U(s')$$

For this purpose a *Policy-Iteration* function was implemented.
Details follow in section 6.4.3 on page 57.

A standard implementation of an active agent depends on a utility function that assigns a utility to each state. To make the best decision, the agent needs to know to which state each action leads to, and thus must build a model of the environment. This is not desirable in the case of an exploration problem where the environment is completely unknown to the agent. The strategy forces the agent to learn, not only the utilities of all the states, but also how the states are linked together by means of the actions.

With this in mind, another possibility to implement such an agent is preferred: the *Q-learning* agent uses a Q-function to evaluate the possible moves. The Q-function associates a state with an action and returns the performance of that combination,

$$U(s) = \max_a Q(a, s)$$

where *U(s)* is the utility of the state s and *Q(a,s)* returns the utility of taking the action a from the state s. A Q-function agent has the advantage of not needing a model of the environment to operate: it can choose among the available alternatives without having to know the outcome. A disadvantage is that without having a model of the environment, a Q-learning agent does not learn a set of consistent utility values, a lack that can decrease its learning efficiency.

6.3. The internal representation

While exploring, the cognitive robot develops an internal representation of the environment in declarative memory. This representation contains all of the information that it knows about the already explored environment. For every step or turn the robot takes, information is stored in declarative memory in a chunk of type *movement*, these chunks have a slot called *direction* that encodes the direction of every movement:

```
(chunk-type movement direction counter)

MOVEMENT0-0
ISA MOVEMENT
  DIRECTION  FORWARDS
  COUNTER  1
```

If the robot turns itself to a new direction, a new movement is created and the name of the direction is written into its direction slot. If the robot keeps going the same direction, a new chunk with direction *forwards* is created.

Every movement chunk has another slot called *counter*, where a progressive number is stored. The numbers detail the sequence of movements of the current run. Through this trace, it is always possible to retrieve the direction it is currently facing: it is memorized in the last *movement* chunk with a direction value different from *forwards*.

The maze is represented in declarative memory as a graph, whose nodes are the junctions. The model does not care how long and how twisted the road from JunctionA to JunctionB is, because it does not involve decision making.
The only relevant information is the expected performance in taking a specific turn, which is represented in declarative memory by a chunk of type *junction*.

```
(chunk-type junction turn north east south west performance)

JUNCTION1-0
ISA JUNCTION
  TURN  EAST
  NORTH  0
  EAST  2
  SOUTH  2
  WEST  3
  PERFORMANCE  62
```

This chunk identifies a specific junction by the values in four of its slots (north, east, south and west) and a direction in its *turn* slot. The use of an absolute coordinate system allows the model to recognize a definite junction, no matter which direction it came from. This solution adds the complexity of keeping track of every state of the agent's orientation but avoids the overwhelming redundancy of memorizing several copies of the same chunks. Having multiple copies of the same chunk would introduce the disadvantage of not being able to realize, in a simple way, that they are related.
For every turn the model must know its quality, this value is encoded in the *performance* slot. A smaller number implies a better performance, the number *-1* means that this direction was taken but not yet rated.

This type of chunk virtually implements the above-quoted Q-function:
the *performance* slot rates the action of turning in a specific direction, written in the *turn* slot, while in a determined state, specified by the four distance measures.

When the model encounters a junction, all the *junction* chunks related to it are retrieved from declarative memory. The information stored in these chunks allows the model to know which routes have already been explored, and which is the best among them.

With such a complete knowledge the model is able to choose the most promising direction and find the shortest path to the goal.

These two chunk types form the model's *Knowledge Base*, through which the previous explorations can be reconstructed. The *movement* chunks contain the path history: They save, for every step, the direction in which the model was going; while the *junction* chunks compose the decisional history and save the order in which the junctions were taken and the performance value's updates of every junction.

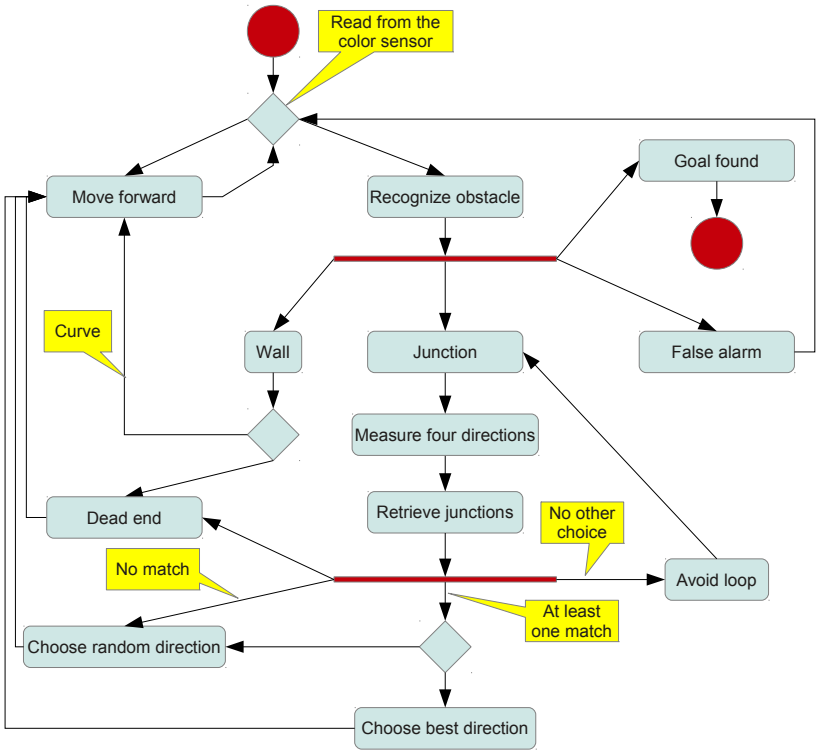


Figure 6.1.: The local search algorithm implemented in ACT-R.

6.4. Algorithm

The robot uses a local search algorithm (cf. Fig. 6.1) to find the best (shortest) path to the goal, it works as follows: If the sensors report no obstacle, the robot keeps moving forward step by step. After every movement a new *movement* chunk with direction *forwards* and a progressive number in the *counter* slot is saved in declarative memory. When the color sensor reports a change in the floor color, a *visual-location* chunk appears in the visual-location buffer. At this moment the robot stops and four different responses can follow: depending on the color read, one of four productions will be fired; each production deals with a different type of landmark (a junction, an obstacle, the goal or a false alarm).

6.4.1. Junction

When a *junction* is detected, by means of a yellow marker, the robot turns itself in the four directions and measures the distance to the nearest wall with its ultrasonic sensor. These four numbers allow the model to discriminate between junctions, because it is unlikely that any two junctions will have exactly the same values.
Once the distances have been retrieved, it tries to recognize the junction among the ones it has seen before. This process is completed by comparing the length of the four corridors that branch off from the junction with the ones memorized in declarative memory.
To make this process robust against measurement errors, these values do not need to be equal to match, but they need to differ less than a predefined threshold. Many retrieval requests to declarative memory are issued by calling the same production many times until the retrieval process fails.
Each chunk can only be retrieved once because the model checks the trace and excludes previously examined chunks in the next retrieval request.

This process makes use of *dynamic productions*: these productions are compiled in run-time and, unlike normal productions, can have variable symbols on their left sides. Every time a junction chunk matches, the value of its *turn* slot, that contains the direction it is referring to, is saved in the corresponding slot of the goal buffer:

```
=imaginal>
        isa     junction
        turn    =dir
=retrieval> ;junction retrieved from declarative memory
        isa     junction
        turn    =turn
        north   =n
        south   =s
        east    =e
        west    =w
        performance     =perf
==>
!bind! =back (turn-back =dir)
!bind! =temp (get-movement =dir =turn)
=goal> ;modifying the goal buffer
        =turn   =turn
        =back   closed ;mark the way it's coming from as closed
        =temp   =perf
```

For example: assume the retrieved junction contains the value *east* in its *turn* slot. The chunk in the goal buffer, that is of type ext-junction

```
(chunk-type ext-junction turn north east south west performance front right
   left)
```

is modified to contain the value *east* in the *east* slot. The function *turn_back* gives the opposite direction of its parameter as output. In this example assume that the current orientation is north, the function will return the value *south*.
The chunk in the goal buffer is updated to store the fact that the way the robot came from is virtually closed, because it is not supposed to go back on its steps.
The *ext-junction* chunk has three slots more than a normal junction, these are needed to keep track of the performance of each retrieval for later use.
The function *get_movement* returns the direction in which the robot must turn to go in the direction contained in its second parameter. In the current example the retrieved junction says to turn left (from north to east).

Once the goal buffer is properly modified, it is possible to go to the next step and look for another match. Since the same production will be called again next, the model needs to be sure not to match with the actual junction anymore.
The request made to the retrieval buffer is as follows:

```
+retrieval> ;tries to retrieve another chunk
        isa     junction
-       turn    =north
-       turn    =east
-       turn    =south
-       turn    =west
-       turn    =back
-       turn    =turn
        north   =n
        south   =s
        east    =e
        west    =w
-       performance     nil ;only a direction it took before
```

The request specifies that this time must not match junctions whose *turn* value is equal to the direction the robot is coming from (=back), it has been retrieved this time (=turn) or is memorized in the goal buffer (=north, =east, =south, =west represent the values contained in the homonym slot of the ext-junction chunk).

Substituting the variable terms with their actual values, the request will look like this:

```
+retrieval>
        isa     junction
-       turn    nil
-       turn    east
-       turn    nil
-       turn    nil
-       turn    south
-       turn    east
        ...
```

This process can repeat itself from zero to three times, depending on how many chunks will match overall. Most of the time there is at least one direction that has not been rated yet and so for that direction no match will be found.

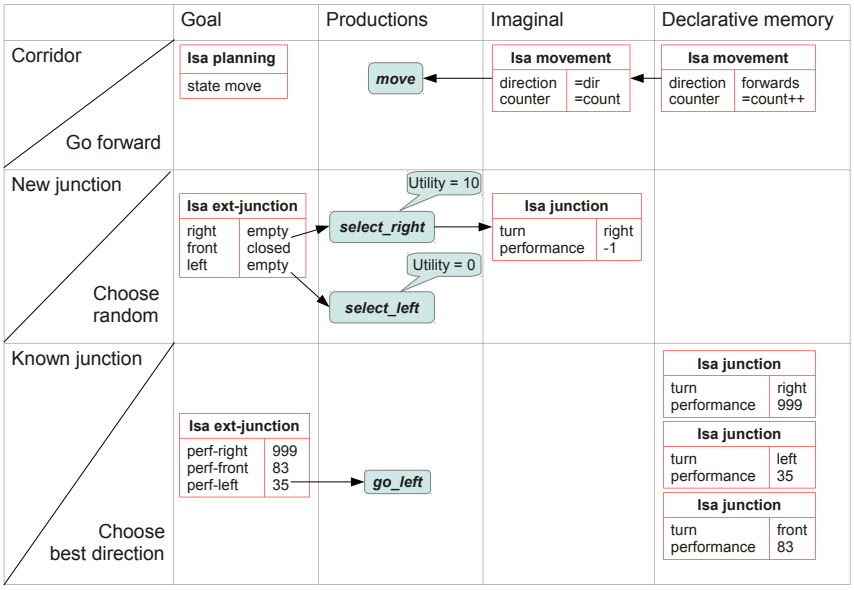


Figure 6.2.: Decision process for corridor, new and known junctions.

Once the retrieval process is finished, the model remains with a certain number of directions that did not match. The reasons for the lack of a corresponding chunk in declarative memory can be one of two:

- That path has not been explored yet.
- That direction is blocked by a wall.

The next step is to exclude from further actions all directions that lead to a wall: the model checks every direction that did not match its corresponding distance and, if it is less than a certain threshold, it detects a wall and marks that direction as not selectable.

```
=imaginal>
        isa     junction
        turn    =dir
<=      north   =threshold
==>
=goal>
        north   closed
```

At this point the only directions marked in the goal buffer with the value *nil* are the ones not yet explored. If there is still at least one direction that has not been tried yet, the model has two possibilities:

The first possibility is to select that direction and explore a new branch of the labyrinth; if more than one path has not been rated yet, the model chooses the one to explore following the defined exploration strategy. At the beginning of the exploration, when the knowledge of the environment is almost none, the model has no other choice than always follow the exploration strategy. At this point of the execution four different productions are ready to get fired by the procedural module: each of these concurrent productions aims to move towards one of the four cardinal points. The decision is made to fire the production with the highest utility. The exploration strategies are implemented through variable utility bonuses for each of these four productions.
An overview of the available exploration strategies and a deeper analysis of the implementation will be discussed in the followings chapters.
At this point the decision making ends and the physical moving begins; for this reason the absolute coordinate system becomes obsolete and the model switches to a relative coordinate system that discriminates between right, front and left.

Instead of preferring a new exploration that could bring to worse performance the model can choose a more conservative option: select the way with the best performance among the directions that matched in declarative memory and go on a safe path. This decision is made looking at the values saved in the *front, left and right* slots in the goal buffer. The smaller the performance value, the better the option.

The decision between an enterprising and a conservative approach is made randomly, with a probability of taking a conservative path proportional to the performance rating: The better the performance, the more likely the conservative approach is to be selected. This system encourages exploration in the first stage, when it is more likely that a shorter way can be found; and is more conservative towards the end, when exploration will bring a minimal increment of performance. A complete knowledge of the environment will make the algorithm completely deterministic.

This random choice is implemented by giving the productions a random utility bonus.

The *calculate_utility* function is set as the *:utility_offset* function, which is called every time the utility value of a production must be computed. The number given as output by this function is then added to the actual utility value of the production. The production with the higher utility value will then be fired by the production system. When the function is called with the name *best_front* as argument, it saves in the *utility* variable the value of the *front* slot in the goal buffer, which contains the performance read in the matching chunk during the retrieval phase. At this point the function *act-r-random* is called with the ratio between 100 and that performance.

```
(defun calculate-utility (prod_name)
  (case prod_name
    ('best-front
      (with-output-to-string (*STANDARD-OUTPUT*) (setq utility
        (EVAL (READ-FROM-STRING (format nil "(chunk-slot-value ~s front)"
          (car (BUFFER-CHUNK GOAL)))))))
      (- (act-r-random (/ 100.0 utility))
        *utility_thresh*))
    ...
```

The random function gives a random number between zero and its argument as output: the bigger the utility value, the smaller the range in which the random number can fall. A random threshold is then subtrated from this random number this allows the utility value to become negative. In the code the value of the utility threshold is set at 5. This value gives a mean utility bonus of 0 for a performance value of 10, a positive mean bonus for values smaller than 10 and a negative mean bonus for those greater than 10.

If no further exploration is possible in any of the selectable directions matched in declarative memory, the model chooses the way with the best performance.
The last possible situation is that the junction does not have any selectable direction, because there are only walls or dead-ends. Since this junction cannot lead to a solution, there is no sense returning: the model purges the branch that leads to the junction, disallowing the model to come back in the future. The model traces back into its memory to the last selected junction, the one that brought the model to this dead-end and updates the performance value of that direction. During the next run, that direction will be remembered as leading to a dead-end and avoided.

In the particular case in which the only selectable direction has already been taken during the current run, that is the robot retracing its steps, a loop is detected. Loops are dangerous situations in which the agent keeps running in circle and choosing the same directions again and again. To break a loop it is necessary to let the model change its choices, that means changing the performance value of one of the junctions it encounters during the loop: the last unrated junction that the robot took is marked as a dead-end. The model will then follow the loop once again and again find the previous junction. This time the usual direction will be marked as a dead-end, and thus not selected. The model is forced to find another way or purge that junction, interrupting the loop.

6.4.2. Wall

If a *wall* is nearby and the robot is moving towards it, the sensor reports the color blue. Because no junction has been detected, there is only one possible valid direction: a right or left turn, or a dead-end.

Special productions are called for a respond more quickly, because the whole process of reading and memorizing the distances is not needed.
The robot will simply follow the curve or, in case of a dead-end, go back on its steps.

This quick response routine first turns the robot to the right and measures the distance to the wall. If the distance is greater than a certain threshold, the direction is considered free and the robot moves forward and continues exploration along this path.

If the measure is less then the threshold, that direction is registered as a wall and the routine continues to look for a free path turning to the left.

The left side is the only remaining possibility for a free path.
If the distance sensor detects a wall here as well, the model recognizes a dead end; it turns back and take the corridor it came from.

When a dead-end is encountered the last junction in the trace, the one which led to this dead-end, is marked with a extremely low performance value, which will tell the model to avoid that direction in the future.

This is the only case in which the declarative memory is accessed.
In case of a curve a new junction chunk is not created because no decision was made.

6.4.3. Goal

If a *goal* is detected, all junction chunks used in the last run are rated by a rating function that implements the *Policy-Iteration* routine [RN03, p.624].

The Policy-Iteration algorithm finds an optimal policy in a finite number of iterations from an initial policy. During each iteration the current policy is improved until no further improvement can be done.

Each iteration is divided in two steps:

- Policy evaluation: the utility of every state is calculated as if the current policy were to be executed.
- Policy improvement: a new policy that increases the performance, compared to the previously calculated utility values, is created.

In this model the evaluation is done during the run: the utility of every junction is represented by its performance value.

During the run the performance of each new junction is saved in declarative memory, and can have value *-1*, in case of newly explored paths, or *9999*, that identifies a dead-end. The performances assigned in previous runs are not modified. At the end of the current run the rating function is called by the model and performs the improvement step.

At first the new performances of the junctions encountered during the exploration are calculated by the rating function as the time difference between the goal discovery and the last access to the chunks representing the junctions in declarative memory. This rating system has the advantage of being precise, because the ACT-R's virtual time needed to complete every operation is fixed and constant, so reliable.
After that the new policy is updated indirectly, by updating the utility values of the junctions. A value is updated if certain preconditions are met:

```
(if (and (numberp old_performance) ;if the value is nil no need to update it
    (not (eq old_performance 9999)) ;it isn't a dead end
    (or (eq old_performance -1) ;if it wasn't rated yet
    (< new_performance old_performance))) ;the new value is better
  (EVAL (READ-FROM-STRING (format nil "(set-chunk-slot-value ~s performance ~D)
    " junction new_performance)))
);if
```

If the old performance value is higher, the junction is not a dead-end, or has not been rated yet, the function updates the performance with the new value.

After the rating function has completed the simulation starts again.
For this run the model will count on more information about the environment and on a better policy. The use of the experience accumulated during the previous explorations leads to better choices. Each run in the labyrinth entails an iteration of the Policy-Iteration routine. This routine will, after some time and iterations, converge to a sub-optimal solution, and will find a policy that cannot be improved upon anymore.

The algorithm does not always find the optimal, instead sometimes a sub-optimal solution whose performance may change from a execution to the other is found. On the other hand, Policy-Iteration assures convergence to an optimal solution.
The reason why our implementation is not always optimal can be found in the environment: the model always finds the optimal solution based on current knowledge of the environment, different levels of knowledge lead to different optimal solutions. Since the utility of the junctions is computed and might vary during the execution, the model always operates with different sets of utility values.

Furthermore, the knowledge is generally incomplete. If the environment were known a-priori and totally observable, then the model would be able to solve it optimally.

6.4.4. False alarm

The final possible situation, in which the sensor gives a *false alarm*, is caused by a reading error: the robot is lifted from the ground or the start point is seen during exploration. This production is fired when the other three do not match.

In case of a false alarm the model ignores the reading and goes back to the initial loop of movements forwards, until it receives a new reading from the color sensor.

6.5. Distance computation

The Act-Rientierung Project [DK+11] implemented an algorithm that reconstructs the distance between non-adjacent waypoints, using information about adjacent waypoints gleaned from exploration. This feature has been integrated into our model to provide distance computation capabilities. The original project used a basic navigation algorithm to explore a maze and find waypoints placed in it. The model saves the distance of each waypoint from the start position and calculates the distance from the previous waypoint, the last found during the current exploration. After exploration, once all duplicates have been removed from the list of visited waypoints, the model computes the distance between every pair of elements in the list.

During this last phase, if the distance between two determined waypoints, A and Z, is not know at the moment, the model tries to compute with an indirect method. If the waypoints are not adjacent, it means that there is at least one waypoint on the path between the two. The simplest example is a waypoint B, that is directly connected to both A and Z. All more complex situations can be traced back to a recursive application of this simplest case. When direct computation is not sufficient, the model expands its knowledge by joining the information about two pairs of directly connected waypoints (e.g. A-B and B-Z) in a single entity that represent a direct connection between the two extremes. With this expedient it becomes possible to calculate the distance between every pair of waypoints.

The computation is made using chunks that represent numbers. Adjacent numbers have a high similarity value, this value tells ACT-R's retrieval system how closely the two chunks are related. If two chunks are very similar, the retrieval mechanism has a higher probability of confusing the chunks, and thus of retrieving the wrong one. This situation leads to counting errors that reproduce human behavior, seen in similar experiments done with human subjects. The integration in our model entails the use of the junctions as waypoints, and allows us to take advantage of the new and sophisticated navigation algorithm in conjunction with the distance computation feature.

The model, during exploration, stores distances of adjacent junctions in specific *waypoint* chunks, these values keep track of how many steps it takes to go from one adjacent waypoint to the other. When the model finds the goal, it goes through the list of known junctions, which includes only those found during the current run of exploration, and calculates the distance between each pair of elements. With consecutive runs the distance values may change, because of the retrieval errors in the counting process.

6.5.1. Example case

Given: the chunk type *waypoint* that stores the distance between two different junctions, represented by the length of the corridors that branch off it in the four directions.

```
(chunk-type waypoint
  ;first waypoint
  n1 e1 s1 w1
  ;second waypoint
  n2 e2 s2 w2
  distance)
```

The start point and the goal are treated as special waypoints. As they aren't junctions, they do not have corridors whose distance must be read. The start point is represented by the value -2 in the four directions, the goal by the value -1 in the four directions. Assume that during the current exploration, only two junctions have been encountered: The following chunks are present in declarative memory:

```
WAYPOINT0-0
 ISA WAYPOINT
  N1  -2
  E1  -2
  S1  -2
  W1  -2
  N2  NIL
  E2  NIL
  S2  NIL
  W2  NIL
  DISTANCE  0
```

This chunk representing the starting point is automatically created by the model on every start, to assure that there will always be a well-formed chunk to begin with. The distance of this waypoint from the start is, of course, zero.

```
WAYPOINT2-0
 ISA WAYPOINT
  N1  0
  E1  2
  S1  2
  W1  3
  N2  NIL
  E2  NIL
  S2  NIL
  W2  NIL
  DISTANCE  2
```

The first junction is found, a new chunk is saved in declarative memory to memorize that the junction (0,2,2,3) is two steps away from the beginning.

```
WAYPOINT0-0-0           WAYPOINT1-0
  ISA WAYPOINT            ISA WAYPOINT
   N1  -2                  N1  0
   E1  -2                  E1  2
   S1  -2                  S1  2
   W1  -2                  W1  3
   N2  0                   N2  -2
   E2  2                   E2  -2
   S2  2                   S2  -2
   W2  3                   W2  -2
   DISTANCE  2             DISTANCE  2
```

This pair of chunks immediately saved in declarative memory, which tells us that the first junction (0,2,2,3) and the starting point are directly connected and separated by a distance of two. The only difference between the two is the position of the coordinates: the junctions have been swapped to highlight the commutativity of the distance.

```
WAYPOINT4-0
 ISA WAYPOINT
  N1  1
  E1  4
  S1  1
  W1  0
  N2  NIL
  E2  NIL
  S2  NIL
  W2  NIL
  DISTANCE  5
```

The second junction (1,4,1,0) is encountered: it has a distance of five from the start.

```
WAYPOINT2-0-0          WAYPOINT3-0
  ISA WAYPOINT           ISA WAYPOINT
   N1  0                  N1  1
   E1  2                  E1  4
   S1  2                  S1  1
   W1  3                  W1  0
   N2  1                  N2  0
   E2  4                  E2  2
   S2  1                  S2  2
   W2  0                  W2  3
   DISTANCE  3            DISTANCE  3
```

This pair of chunks tells us that the first and the second junction are three steps apart.

```
WAYPOINT5-0
  ISA WAYPOINT
   N1  1
   E1  4
   S1  1
   W1  0
   N2  -1
   E2  -1
   S2  -1
   W2  -1
   DISTANCE  1
```

The last chunk tells us that the goal (-1,-1,-1,-1) has been found and that it has a distance of one from the second junction. Now the exploration is done and the distance computation algorithm will start.
The algorithm gives chunks of type Antwort ("answer" in German) as output:

```
(chunk-type Antwort
  ;first waypoint
  n1 e1 s1 w1
  ;second waypoint
  n2 e2 s2 w2
  Wert
  Entfernung)
```

which are actually the same as a waypoint chunk, but with the difference that the distance is represented by another chunk of type Zahl ("number" in German):

```
(chunk-type Zahl Name Wert Nachfolger)
(Eins isa Zahl Name "Eins" Wert 1 Nachfolger "Zwei")
```

The algorithm starts its computation from the first waypoint:
the start point (-2,-2,-2,-2)

```
ANTWORT0-0
 ISA ANTWORT
  WERT  ZWEI
  ENTFERNUNG  "Zwei"
  N1  -2
  E1  -2
  S1  -2
  W1  -2
  N2  0
  E2  2
  S2  2
  W2  3
```

The first distance can be calculated directly, because the needed information is stored in declarative memory into the chunk WAYPOINT0-0-0

```
ANTWORT1-0                ANTWORT2-0
  ISA ANTWORT               ISA ANTWORT
   WERT  FUENF               WERT  SECHS
   ENTFERNUNG  "Fuenf"       ENTFERNUNG  "Sechs"
   N1  -2                    N1  -2
   E1  -2                    E1  -2
   S1  -2                    S1  -2
   W1  -2                    W1  -2
   N2  1                     N2  -1
   E2  4                     E2  -1
   S2  1                     S2  -1
   W2  0                     W2  -1
```

Now the algorithm tries to calculate the distance between the start point and the second junction. This information is not stored in declarative memory, so the model needs to get it indirectly from the information it already has. A retrieval is issued to the declarative module, that requests a chunk that has as first waypoint (-2,-2,-2,-2) and as second waypoint any valid junction. If the retrieval is successful, the unknown waypoint (in our example (0,2,2,3), the first junction) is used as first element of a new retrieval request. The second waypoint is again not specified. The retrieval module will give back a waypoint among those directly connected to the middle waypoint (in our example it will necessarily be (1,4,1,0), the second junction).

```
WAYPOINT6-0
 ISA WAYPOINT
  N1  -2
  E1  -2
  S1  -2
  W1  -2
  N2  1
  E2  4
  S2  1
  W2  0
  DISTANCE  5
```

Now a new chunk of type waypoint will be stored in declarative memory, with (-2,-2,-2,-2) as the first waypoint, and (1,4,1,0) as the second waypoint and as distance the sum of the retrieved distances.

With this information the algorithm can fall back to the direct search and proceed with the calculations.

```
ANTWORT3-0                ANTWORT4-0                ANTWORT5-0
  ISA ANTWORT               ISA ANTWORT               ISA ANTWORT
   WERT  DREI                WERT  VIER                WERT  EINS
   ENTFERNUNG  "Drei"        ENTFERNUNG  "Vier"        ENTFERNUNG  "Eins"
   N1  0                     N1  0                     N1  1
   E1  2                     E1  2                     E1  4
   S1  2                     S1  2                     S1  1
   W1  3                     W1  3                     W1  0
   N2  1                     N2  -1                    N2  -1
   E2  4                     E2  -1                    E2  -1
   S2  1                     S2  -1                    S2  -1
   W2  0                     W2  -1                    W2  -1
```

Once all the distances from the first waypoint are computed, the algorithm skips to the second waypoint and the process continues until the algorithm has finished.

7. The Simulator

Once the model was completed, our interest shifted in comparing its performance against the humans. For this reason we needed a test scenario that could speed up the development of the model and collect precise numerical results in a short period of time. Such a task is necessarily completed through the creation of a simulated environment.
The simulator consists of a set of functions that substitute the lisp functions provided by the nxt-lsp library. Those functions control a virtual robot that travels through a virtual maze, described by a two-dimensional list that contains different characters with different meanings. The "virtual" functions interface themselves with the modules in the same way as the "real" functions do, so there is no need of changes in the module itself.

The virtual robot has the same perceptions as the real one: it can recognize the distance from the next wall as the number of free cells in front of it, and the color as the character of the next cell. It can also move in the labyrinth and turn itself, simply modifying the value of 3 variables.

The advantages of testing in a simulated environment are, first of all, the high speed at which the simulator can run: the virtual robot does not need to wait for the sensors to respond or the motors to move, all the physical waiting times are nullified.
Besides the simulator changes the field of operation from a partially observable and continuous environment to a Fully observable and discrete one. This change takes all the uncertainty away: the model does not suffer anymore of odometry or measurement errors, because the position is always exact and discrete.

Another advantage of a simulated environment is its flexibility: substantial changes can be applied to the maze in a really short time and without leaving the keyboard, allowing the tester to create a vast amount of test labyrinths in a very short time. Also the robot can be instantly placed by modifying a couple of variables or can go back to the beginning without the need for the operator to place it. This last feature is useful to create batch tests, that automatically start again from the beginning after finding the goal.

The simulator was build following some compatibility criteria. The provided functions presented a signature identical to the corresponding "real-world" function, that allow those functions to be effortlessly substituted in the modules.

The model's code itself needs no modification at all to be run on a virtual robot. With this simulator it was possible to quickly obtain large amount of data about multiple runs, and build a statistic on them. Also the experiments done with it are perfectly repeatable with the real robot, thanks to the equivalence between the two solutions.

7.1. Implementation

In our implementation the labyrinth is represented as a two-dimensional list (array) that contains characters.

```
(x x x x x x x x x x)
(x x x - - - - x x x)
(x x x - x x - x - x x)
(x s - j x x j - j g x)
(x x x - x x - x x x x)
(x x x - - - - x x x x)
(x x x x x x x x x x)
```

Each character has a different meaning:

- The character *'X'* identifies a wall. The robot can not walk over walls so it checks every time, before making a movement, that the next cell does not contain a wall.
- The character *'-'* identifies a corridor over which the robot can move.
- The character *'S'* represents the start point. There can be only one start point in each labyrinth. The simulator looks for it before starting a new run and places the robot over it. During the navigation, the model gives no importance to it and treat it as a normal way.
- The character *'G'* represents the goal. Like the start point, it is unique. When the model finds the goal, the navigation process is interrupted.
- The character *'J'* identifies a junction. When the robot finds a junction, it stop and start to look around.

The virtual robot's state is composed by three variables: its X and Y coordinates, as indexes in the labyrinth's array, and it's orientation. The robot can explore the environment thanks to the functions that control the robot's movements:

```
(defun next_cell ()
        (case *userDir*
                (0 ;north
                        (setf row (- (car *userPos*) 1))
                        (setf col (cadr *userPos*))
                        (if (checkPosition row col)
                                (list row col)
                                *userPos*))
                (1 ;east
                        (setf row (car *userPos*))
                        (setf col (+ (cadr *userPos*) 1))
                        (if (checkPosition row col)
                                (list row col)
                                *userPos*))
                (2 ;south
                        (setf row (+ (car *userPos*) 1))
                        (setf col (cadr *userPos*))
                        (if (checkPosition row col)
                                (list row col)
                                *userPos*))
                (3 ;west
                        (setf row (car *userPos*))
                        (setf col (- (cadr *userPos*) 1))
                        (if (checkPosition row col)
                                (list row col)
                                *userPos*))
                (t (prin1 "ERROR"))))

(defun checkPosition (row col) ;; userPos should be inside maze
        (if (not (or (< col 0) (< row 0)
                (> col (- (array-dimension *mazeArray* 1) 1))
                (> row (- (array-dimension *mazeArray* 0) 1))))
                t
                nil))
```

The *next_cell* function updates the user's position according to its actual orientation. This function calls the *checkPosition* routine that controls if the new position is valid.

```
(defun move_forward ()
        (when (not (eq (read_color) 'blue)) ;if the next cell is not a wall
                (setf *userPos* (next_cell)) ; it can move forward
))
```

The *move_forward* function looks in the next cell and, if the cell contains no wall and is within the labyrinth's boundaries, updates the agent's coordinates to the new position.

```
(defun turn_right ()
        (if (eq *userDir* *west*)
                (setf *userDir* *north*)
                (setf *userDir* (+ *userDir* 1))))

(defun turn_left ()
        (if (eq *userDir* *north*)
                (setf *userDir* *west*)
                (setf *userDir* (- *userDir* 1))))
```

Other functions allow the robot to turn left or right, by modifying its actual orientation.

The simulator provides functions that clone the behavior of the sensors and give the virtual robot the perceptions it needs.

```
(defun read_color ()
        (setf coords (next_cell)) ;read the coords of the cell it's facing
        (setf cell (aref *mazeArray* (car coords) (cadr coords)))
        (case cell
                ('s 'green)
                ('g 'red)
                ('x 'blue)
                ('* 'blue)
                ('j 'yellow)
                ('- 'white)
                ('d 'white)
                (t (prin1 "WRONG SYMBOL"))
        );case
)
```

The function *read_color* reads the character in the next cell, in the direction given by the robot's orientation, and outputs a different color according to the read character: A wall is associated with the color Blue, while a free way with the color White, the character S is related to the color Green, the goal G has the color Red, and the color Yellow is paired with the junctions J.

```
(defun get_distance ()
        (setf dist 0)
        (setf bak *userPos*)
        (while (not (eq (read_color) 'blue))
                (setf dist (+ dist 1))
                (setf *userPos* (next_cell))
        )
        (setf *userPos* bak)
        dist
)
```

The function *get_distance* counts how many cells there are, from the actual agent's position until the first wall, in the direction given by the current orientation.
The touch sensor is not implemented as no material walls are present, emergency situations just cannot happen into a virtual environment.

8. Evaluation

To be able to compare the results the test has been set up as a replica of the Büchner experiment [BHW09], recreating the same labyrinths inside the simulator (cf. Fig. 2.1); in both the configuration with the goal on the edge and in the center. The performance was measured in PAO (Percentage Above Optimum).

$$PAO = ((d_{walked} - d_{shortest})/d_{shortest}) \times 100$$

This unit of measurement represents how worse the found solution is, in percentage, respect to the optimal, shortest solution. For example if the optimal solution has a value of X and the solution found by the algorithm has a value of $X + \frac{X}{2}$, then the PAO value of the suboptimal solution will be 50%.
The same unit of measurement was used in this experiment as well, so that the results of both can be easily compared.

During every simulation, for each of the four environments, the model had a time limit of 4000 units of ACT-R's virtual time to find an optimal solution. Each time that the model finds a goal the agent is placed again on the start point and the exploration starts again. After each run the information about the environment and the goal position still exists in the declarative memory, and is used to find a shorter path to the goal. When the 4000 units of time are expired the simulation ends, the results are printed and the declarative memory is cleared. In the beginning of each simulation the agent has no information whatsoever about the environment, so each simulation is run in the same working conditions.
For every environment 50 simulations have been run and a statistic extrapolated from the results. Within this time constraint the model could find a stable suboptimal solution 84% of the time. The diagram in Fig. 8.1 shows the mean quality of the consecutive solutions.

The curves in Fig. 8.1 show similar learning behavior for all four labyrinths: The model starts with an uninformed exploration, according to the actual strategy, until it finds the goal. This phase can take more or less time depending on the effectiveness of

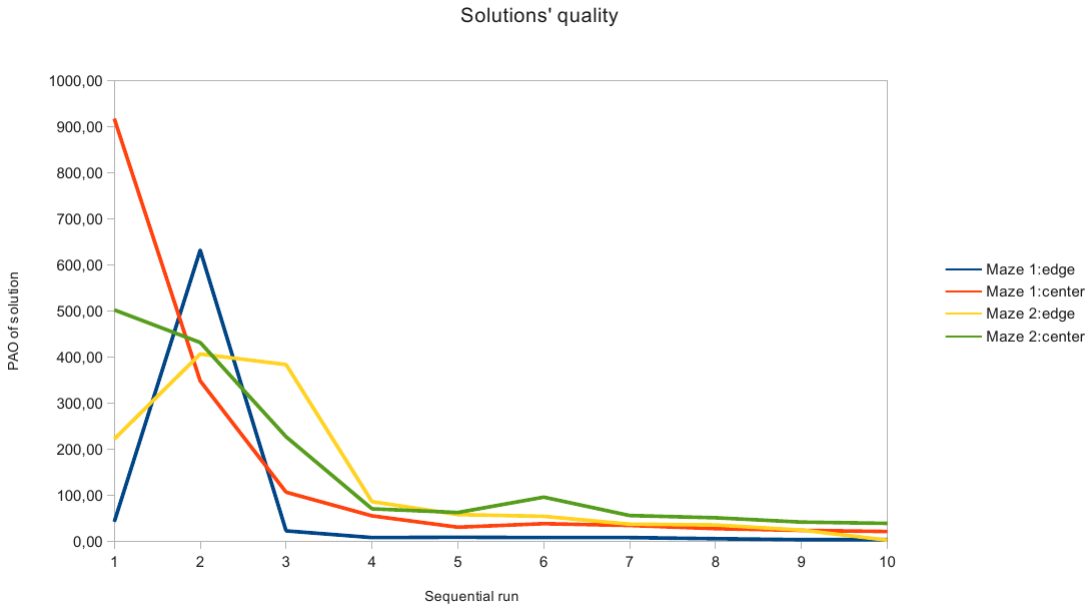


Figure 8.1.: Behavior of the model, showing a learning curve similar to the humans' [BHW09].

the selected strategy in the actual maze configuration. In the graph is blatant to realize that the actual strategy performed way better in the configurations with the goal on the edge than in the ones with the goal in the center.

After this first "blind" run the model starts an exploration phase that will try new ways until all possibilities have been tried and rated. This phase takes, in average, more time to find the goal than the first run and than the human counterpart. Because the possible different paths are in a finite number it happens that the length of the second run is in inverse proportion to the first one: if the first run is short the number of unrated variations to the original path is high and the exploration will take much time to rate them all; if the first run takes long time to complete more paths are going to be rated and so, to complete the exploration, just a small number of alternatives need to be covered.

After the second run the model has enough information to start the learning process and improve, in every run, its performance. After less than 4 runs, on average, the model has gathered enough information about the environment and stabilizes on a suboptimal solution that is covered until the time elapses.

In the experiment done by Büchner the test subjects shown a preference for the perimeter strategy to navigate in the maze and search for the goal. In the current experiment different strategies were tested:

8.1. Deterministic perimeter strategy.

The first strategy uses a right-preference perimeter strategy. Each application of turn-right production rules yields a utility bonus, a smaller bonus is given for the go-straight production while for the turn-left no bonus is assigned. The model will then prefer to turn right over turning left, because it assures a higher reward. This means that the agent will try, for every junction, to take the rightmost free corridor.

The test showed that the first run is completely deterministic and in every run the agent always take the same path, that implies a much smaller degree of choice during the rest of the exploration. Even if some differences can be seen during the exploration phase, every run leads to the same suboptimal solution. In conclusion the tests demonstrated that this strategy is too deterministic and dependent on the specific maze to be suitable for a real exploration task.

8.2. Random walk.

Another strategy was a random strategy with utility learning, in the beginning all the productions have the same utility, which can change during the run according to the utility rewards gathered during the exploration. A positive utility reward is given by the action of finding the goal, with a negative reward for each dead-end.

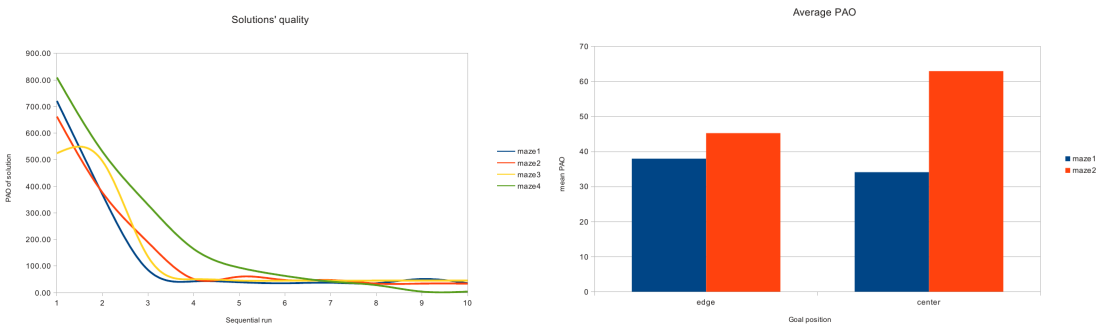


Figure 8.2.: Left: Mean performance of the random walk in the four test environments
Right: Solution quality of the random walk strategy in the four environments.

The graph on the left of Fig. 8.2 shows that, for each of the four environments, the model behave in the same way. This result can be reasonably expected from a random walk that makes no differences between the four mazes, because of its aleatory nature.

On the other hand the quality of the solution differs from a maze to the other, as seen in the right of Fig. 8.2 .

This difference is the outcome of the learning process in the navigation algorithm. Even if the performance changes between the four environment, it is always much better (more than an order of magnitude) than the initial performance.

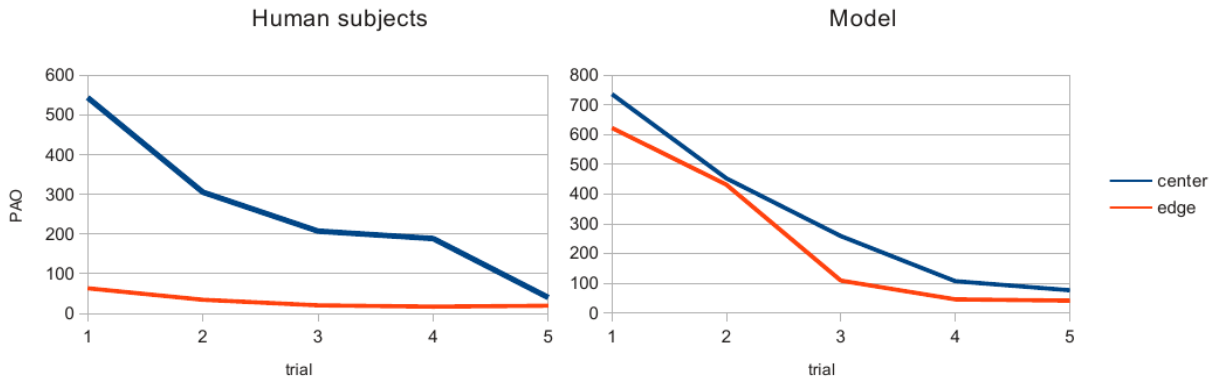


Figure 8.3.: Human (left) and model performance (right) with random walk, the curves have a correlation of 0,962 and 0,954.

The comparison in Fig. 8.3 places side by side the mean performance of the human subjects and of the model. The results are grouped by labyrinth's class, to be comparable to the results of the Büchner experiment [BHW09]. The time needed to find the goal in the center of the maze is almost identical to the one scored by the human beings, while the humans outperform the model in finding the goal on the edge. The human behavior shows an advantage over our implementation in solving the second class of problem. This result can be explained by the lean of the humans, as proved by our reference experiment, to prefer the perimeter strategy while exploring an unknown environment. Because of this reason, a third strategy was implemented and tested.

8.3. Gaussian perimeter strategy.

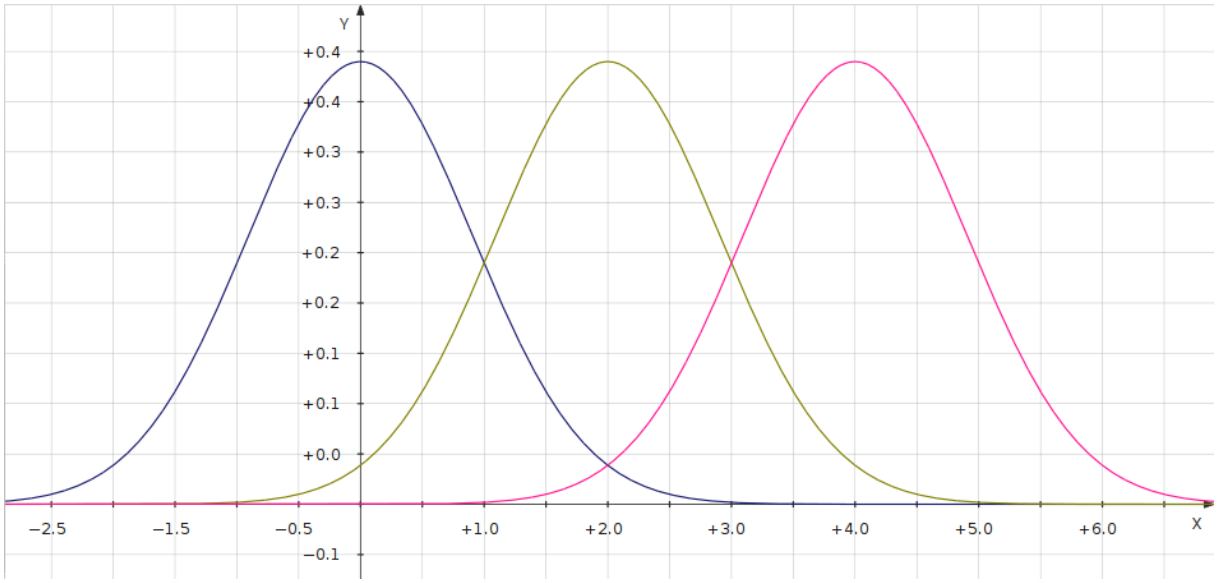


Figure 8.4.: The Gaussian error.

The goal here is to implement a system that prefers the perimeter strategy but is also shows some degree of randomness. The implementation provides for a random utility bonus applied to the three choices (Fig. 8.4), the more desirable is the action to perform, the higher the center of the Gaussian will be. The three curves, identifying the action of turning left, going straight on and turning right, are centered respectively in 0, 2 and 4. In practice, the action of turning right will be the favorite and the action of turning left the least likely to happen.
This bonus is added to the standard production's utility by the mean of the *:utility-offsets* function [Bot04, p.187].

```
(defun calculate-utility (prod_name)
  (case prod_name
    ('go-north-random
      (if (eq *userDir* *west*);
        (+ (act-r-noise *s*) 4) ;right
        (if (eq *userDir* *north*);
          (+ (act-r-noise *s*) 2) ;straight
          (act-r-noise *s*) ;left
        )))
    ...
```

The Gaussian functions are calculated by the *act-r-noise* function [Bot04, p.138] with parameter $s = 0.5$, that correspond to a variance $\sigma^2 = 0,82246$.
The *calculate_utility* function is set as the *:utility_offset* function, which is called every time that the utility value of a production must be computed. The number given as output by this function is then added to the actual utility value of the production. The production with the higher utility value will then be fired by the production system. The function is invoked with, in its parameter, the name of the current production to rate. Taking as example the "go-north-random" production, the function reads the current user direction and generates a Gaussian curve with the "act-r-noise" function. If the current direction is "west", the action of turning north will be the favorite because implies a turn right, for this reason the center of the Gaussian curve is shifted to 4. In case the current direction is "north", the action of going north will need no turns and the Gaussian curve will be centered in 2. The last case, when the current direction is "east", will get no offset in its utility bonus.

The Gaussian curves intersect one to the other, this means that there is some probability that one action will be preferred over another action that is normally considered better. This behavior normally occurs also in the human reasoning, which is never deterministic. Varying the breadth of the curves, this probability changes. The tests report that, with this probability distribution, the perimeter strategy is chosen 47% of the time in the first maze and the 43% in the second.

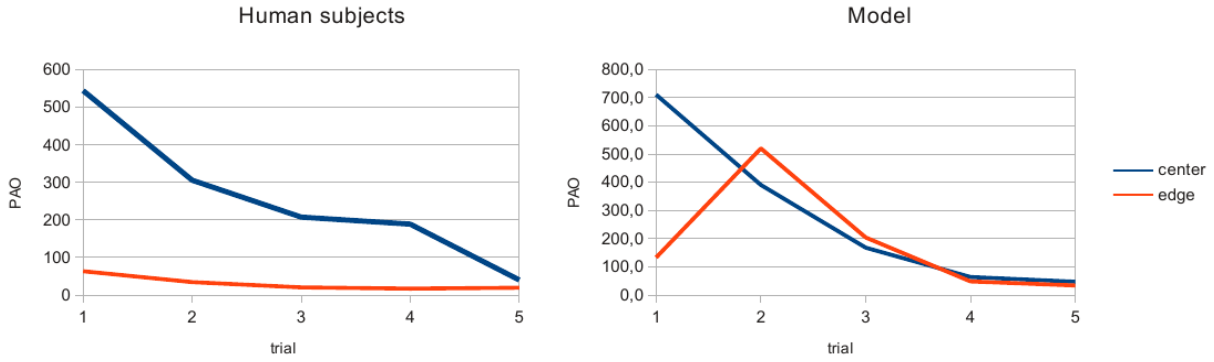


Figure 8.5.: Gaussian perimeter strategy performance. Left: Human - Right: model.

The comparison in Fig. 8.5 shows that, compared to the previous strategy, the initial results are more similar to the one registered with humans. In particular the first run in the edge configuration is now comparable to the human result.
In that particular case, the model shows a behavior very different from the expected: while the humans find quickly a good solution and keep following that path, eventually

making small improvements, the model goes on with exploration and register a worse score in the second run. In the end the learning mechanism brings the solution back to the same initial value or to a better one.

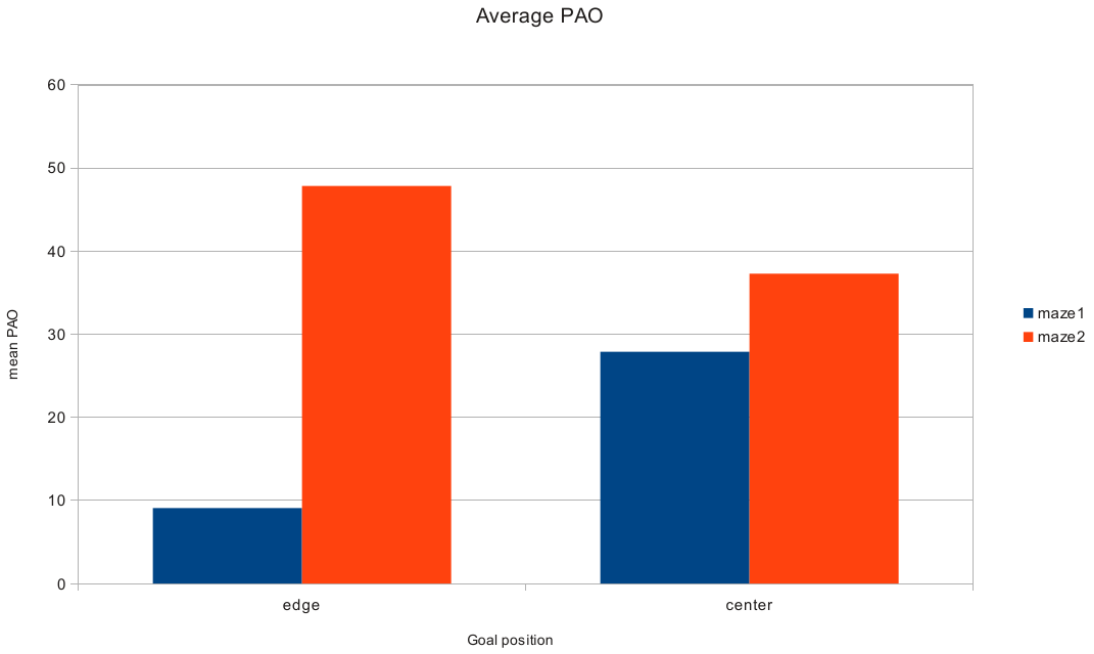


Figure 8.6.: Solution quality of the Gaussian perimeter strategy in the four environments.

The graph in Fig. 8.6 highlights the solution's quality. The solution found in the first maze is almost optimal, and all the values are quite close to the humans results.
Unlike the random walk, this strategy performs always better in the first maze, in both edge and central configurations. This result can be explained by the structure of the labyrinth, in fact the goal is placed in the top-right, an agent that uses a simple perimeter strategy can find the goal with an almost optimal path.
Having a good solution in the beginning makes easier for the model to find its way through the maze, with less need for further exploration.
While low performance is registered in the second maze in edge configuration. Again the maze structure can explain this result: The goal is placed on the top-left edge, in a position difficult to reach for the implemented strategy. That implies more exploration in the first run and so less possibility of finding a shorter solution and improving the performance.

9. Conclusions

This experiment's main achievement is to have demonstrated that ACT-R can be successfully used to control the navigation of a mobile robot.
We applied a bottom-up method starting with a restricted number of sensors and computational power relying more on the cognitive aspects of ACT-R.
Our cognitive model is not only able to navigate in a labyrinth, but also to learn from its experience and improve its performance. Also this experiment showed that, despite using a simple robot with limited perceptions, is possible to reach human performance or do even better. There is no need for complex architectures and expensive hardware to successfully model the human behavior.

A disadvantage of this simplicity is the need of a more strictly defined environment, the model is not ready for navigating outside its labyrinth, in the real world.
Obtaining more flexibility is not a matter of model but a matter of perceptions: with a more complex hardware that is able to handle more complex perceptions, the same model will theoretically be still able to handle the extra complexity and show the same level of performance.

Even if the results have proved to be satisfying, there is still room for improvement. The most practical improvement is to enhance the robot's overall design:

The use of different and more sophisticated sensor will allow the algorithm to be further improved, for example the use of a webcam could allow the model to recognize landmarks visually, from a distance, and act consequently (e.g. recognize and avoid a dead-end).
Improvement of this type are countless and depends entirely on the available budget.

Another useful improvement is to study a new, better designed chassis. It will contribute in reducing the odometry and measurement errors, which are still a problem during the execution in a real environment.
For example the sensors could be mounted on a rotating turret that allow then to measure in all the directions without the need of the robot to turn itself.
A sensor like a compass could allow the robot to measure its absolute orientation, a really important information for the navigation in open terrains that will let the robot to move not only along the Cartesian axis, but with every angle.

On the Artificial Intelligence point of view a valuable improvement is to move to a *generalized reinforcement learning* algorithm [RN03, p.777]. A generalized approach differs in the internal representation: it drops the tabular Q-function, that encodes the performance of every state, in favor of a parametric Q-function. A parametric Q-function has the advantage of compressing the information needed to handle with huge state spaces, also it allows to generalize and do inference from known to unknown states. With the help of a *Policy Search* algorithm the Q-function's parameter can be learned and optimized until they produce good performances.

On the other hand, the cognitive model could be as well further enhanced by implementing a better simulation of human memory, for example adding uncertainty in the memorization process. Humans tend to remember less efficiently complicated information or to confuse similar information. At the moment the model has a perfect memory and the only source of error is the ambiguity between junctions.

A first step in this direction is to introduce a utility threshold under which the chunks cannot be retrieved, some tests in this direction have been executed, but to get valuable results more work is needed on the parameter's tuning.

A further step is to introduce a difficulty level that makes memorization more difficult. Humans tend to have more difficulties in memorizing a complex information than a simple information, for example is more likely to forget the length of a corridor in a junction with four corridors then in a junction with only one corridor.
The complexity level could be proportional, for example, to the number of open ways in that junction or the length of the corridors.

Another interesting enhancement is to allow the model to randomly forget or switch junction performance, as well as the user direction, like humans tend to do when the quantity of information they have to remember is too great
This feature could be easily implemented using the similarity function embedded in ACT-R. This function, already used in the distance calculation process, allows to set some similarity between chunks, the bigger the similarity the higher the chance that the retrieval process will load the wrong chunk.

Acknowledgments

First af all I would like to thank my supervisor at the Center for Cognitive Science of the Freiburg University, Prof. Dr. Marco Ragni for giving me this project, the resources to work on it and helping me whenever I needed.

I would also offer my gratitude to the people that helped me during my work:
Thanks to Dan Bothell for extensive discussions about the working of ACT-R and help during the coding phase.
Tasuku Hiraishi who developed the nxt-lsp library, that made my life incredibly easier.
Simon Büchner for discussions about his study and human navigation in general.
Franz Dietrich for introducing me his mboxACT-Rientierung project and giving to me hints about the integration with my project.
Rebecca Albrecht for her help about the modeling in ACT-R.

Special thanks go as well to Prof. Riccardo Cassinis, for accepting his role of supervisor, and to Prof. Gerhard Strube, for accepting me in his department.

Finally I want to sincerely thank my family for supporting me, morally and economically, throughout my studies and for giving me the possibility of following my urge to study abroad.

A. Source code

A.1. nxt-motor

```
(defvar *engine-velocity* (/ *max-speed* 4) "The␣default␣engine␣velocity")
(defvar portlist '(:a :b :c)) ;the available engine ports

#+:packaged-actr (in-package :act-r)
#+(and :clean-actr (not :packaged-actr) :ALLEGRO-IDE) (in-package :cg-user)
#-(or (not :clean-actr) :packaged-actr :ALLEGRO-IDE) (in-package :cl-user)

(require-compiled "DMI" "ACT-R6:support;dmi")
(require-compiled "GENERAL-PM" "ACT-R6:support;general-pm")

;;;; ------------------------------------------------------------------------ ;;;;
;;;; module definition
;;;; ------------------------------------------------------------------------ ;;;;

(defstruct nxt-motor velocity c-engine r-engine l-engine busy)

(defun nxt-motor-create (model-name)
 (declare (ignore model-name))
  (make-nxt-motor))

(defun nxt-motor-reset (nxt)
  (declare (ignore nxt))
  (chunk-type move-forward duration)
  (chunk-type move-backwards duration)
  (chunk-type turn-right duration)
  (chunk-type turn-left duration)
  (chunk-type activate-motor port velocity direction duration) ;direction t=forward nil=backwards
  (chunk-type deactivate-motor port)
  (chunk-type engine port)
  (chunk-type emergency-stop)
)

(defun nxt-motor-delete (nxt)
 (declare (ignore nxt))
)

(defun nxt-motor-params (nxt param) ;getter and setter for the parameters
  (if (consp param)
    (case (car param)
    (:busy
      (setf (nxt-motor-busy nxt) (cdr param)))
```

```
(:velocity
  (setf (nxt-motor-velocity nxt) (cdr param)))
(:c-engine
  (case (cdr param)
    (#\a
      (setf (nxt-motor-c-engine nxt) (first portlist)))
    (#\b
      (setf (nxt-motor-c-engine nxt) (second portlist)))
    (#\c
      (setf (nxt-motor-c-engine nxt) (third portlist)))
    (t (print-warning "Wrong␣parameter"))
  ));case
(:r-engine
  (case (cdr param)
    (#\a
      (setf (nxt-motor-r-engine nxt) (first portlist)))
    (#\b
      (setf (nxt-motor-r-engine nxt) (second portlist)))
    (#\c
      (setf (nxt-motor-r-engine nxt) (third portlist)))
    (t (print-warning "Wrong␣parameter"))
  ));case
(:l-engine
  (case (cdr param)
    (#\a
      (setf (nxt-motor-l-engine nxt) (first portlist)))
    (#\b
      (setf (nxt-motor-l-engine nxt) (second portlist)))
    (#\c
      (setf (nxt-motor-l-engine nxt) (third portlist)))
    (t (print-warning "Wrong␣parameter"))
  ))
(t (print-warning "Wrong␣parameter")));case
(case param
(:busy
  (nxt-motor-busy nxt))
(:velocity
  (nxt-motor-velocity nxt))
(:c-engine
  (case (nxt-motor-c-engine nxt)
    (:a #\a)
    (:b #\b)
    (:c #\c)
  ));case
(:l-engine
  (case (nxt-motor-l-engine nxt)
    (:a #\a)
    (:b #\b)
    (:c #\c)
  ));case
(:r-engine
  (case (nxt-motor-r-engine nxt)
    (:a #\a)
    (:b #\b)
```

```
        (:c #\c)
      ));case
    (t (print-warning "Wrong␣parameter")))
))

(defun nxt-motor-requests (instance buffer-name chunk-spec)
(if (nxt-motor-busy instance)
  (model-warning "Request␣made␣to␣the␣~S␣buffer␣while␣the␣nxt-motor␣module␣was␣busy.␣New␣request␣ignored."
     buffer-name)
  (let
    ((setf (nxt-motor-busy instance) t))
    (case buffer-name
      (nxt-move
        (case (chunk-spec-chunk-type chunk-spec)
          (move-forward
            (nxt-move-forward instance (third (car (chunk-spec-slot-spec chunk-spec 'duration))))) ;returns
      the value of the duration buffer in the request (chunk-spec)
          (move-backwards
            (nxt-move-backwards instance (third (car (chunk-spec-slot-spec chunk-spec 'duration)))))
          (turn-right
            (nxt-turn-right instance (third (car (chunk-spec-slot-spec chunk-spec 'duration)))))
          (turn-left
            (nxt-turn-left instance (third (car (chunk-spec-slot-spec chunk-spec 'duration)))))
          (activate-motor
            (nxt-movement   (third (car (chunk-spec-slot-spec chunk-spec 'port)))
                    (third (car (chunk-spec-slot-spec chunk-spec 'velocity)))
                    (third (car (chunk-spec-slot-spec chunk-spec 'direction)))
                    (third (car (chunk-spec-slot-spec chunk-spec 'duration)))))
          (deactivate-motor ;TODO useful??
            (nxt-movement   (third (car (chunk-spec-slot-spec chunk-spec 'port))) :brake t 0))
          (emergency-stop (nxt-stop-motors))
        )
      )
    )
  )
)
)

(defun nxt-motor-queries (nxt buffer query value) ;handles the queries to the buffers
  (declare (ignore buffer))
  (case query ;type of the query
    (state
     (case value
       (busy (nxt-motor-busy nxt))
       (free (not (nxt-motor-busy nxt)))
       (error nil)
       (t (print-warning "Bad␣state␣query␣to␣the␣~s␣buffer" buffer))))
  (velocity
    (nxt-motor-velocity nxt))
    (t (print-warning "Invalid␣query␣~s␣to␣the␣~s␣buffer" query buffer))
    ))


;;;; ------------------------------------------------------------------- ;;;;
```

```
;;;; special nxt functions
;;;; ------------------------------------------------------------------------ ;;;;

(defun nxt-movement (port velocity direction duration) ;port is a keyword, element of portlist
  (if direction
    (motor port velocity)
    (let ((vel (- velocity))) ;else
      (motor port vel))
  )
  (sleep duration)
  (motor port :brake)
)

(defun nxt-move-forward (nxt duration)
  (motor (nxt-motor-l-engine nxt) *engine-velocity*)
  (motor (nxt-motor-r-engine nxt) *engine-velocity*)
  (sleep duration))

(defun nxt-move-backwards (nxt duration)
  (setq vel (- *engine-velocity*))
  (motor (nxt-motor-l-engine nxt) vel)
  (motor (nxt-motor-r-engine nxt) vel)
  (sleep duration))

(defun nxt-turn-right (nxt duration)
  (setq vel (- *engine-velocity*))
  (motor (nxt-motor-l-engine nxt) *engine-velocity*)
  (motor (nxt-motor-r-engine nxt) vel)
  (sleep duration))

(defun nxt-turn-left (nxt duration)
  (setq vel (- *engine-velocity*))
  (motor (nxt-motor-l-engine nxt) vel)
  (motor (nxt-motor-r-engine nxt) *engine-velocity*)
  (sleep duration))

(defun nxt-stop-motors ()
  (motor-reset))

(define-module-fct 'nxt-motor ;name
                   '(nxt-move) ;buffers
  (list
    (define-parameter :busy
      :documentation "state"
      :default-value nil
      :owner t)
    (define-parameter :c-engine ;cannot set a keyword as default value
      :documentation "nxt's␣third␣engine,␣normally␣not␣used␣for␣movement"
      :default-value #\a
      :valid-test (lambda (x) (characterp x))
      :warning "a␣character"
      :owner t)
    (define-parameter :r-engine
      :documentation "nxt's␣right␣engine"
```

```
      :default-value #\b
      :valid-test (lambda (x) (characterp x))
      :warning "a␣character"
      :owner t)
    (define-parameter :l-engine
      :documentation "nxt's␣left␣engine"
      :default-value #\c
      :valid-test (lambda (x) (characterp x))
      :warning "a␣character"
      :owner t)
    (define-parameter :velocity
      :documentation "max␣rotation␣velocity"
      :default-value *engine-velocity*
      :valid-test (lambda (x) (integerp x))
      :warning "an␣integer"
      :owner t)
    )

   :request 'nxt-motor-requests
   :query 'nxt-motor-queries

   :version "0.1"
   :documentation "Motor␣module␣for␣the␣lego␣nxt␣brick"
   :creation 'nxt-motor-create
   :reset 'nxt-motor-reset
   :delete 'nxt-motor-delete
   :params 'nxt-motor-params
)
```

A.2. nxt-touch

```
#+:packaged-actr (in-package :act-r)
#+(and :clean-actr (not :packaged-actr) :ALLEGRO-IDE) (in-package :cg-user)
#-(or (not :clean-actr) :packaged-actr :ALLEGRO-IDE) (in-package :cl-user)

(require-compiled "DMI" "ACT-R6:support;dmi")
(require-compiled "GENERAL-PM" "ACT-R6:support;general-pm")

;;;; ------------------------------------------------------------------------ ;;;;
;;;; module definition
;;;; ------------------------------------------------------------------------ ;;;;

(defstruct nxt-touch port)

(defun nxt-touch-create (model-name)
  (declare (ignore model-name))
  (make-nxt-touch))

(defun nxt-touch-reset (nxt)
  (declare (ignore nxt))
)

(defun nxt-touch-delete (nxt)
 (declare (ignore nxt))
)

(defun nxt-touch-params (nxt param) ;getter and setter for the parameters
(if (consp param)
    (case (car param)
    (:touch-port
      (setf (nxt-touch-port nxt) (cdr param))
      (dolist (var (nxt-touch-port nxt))
        (nxt-define-sensor var)
      )
      ; (princ "And the ports are: ")
      ; (prin1 (nxt-touch-port nxt))
    )
    (t (print-warning "Wrong␣parameter")))
    (case param
    (:touch-port
      (nxt-touch-port nxt))
    (t (print-warning "Wrong␣parameter")))
))

(defun nxt-touch-queries (nxt buffer query value) ;handles the queries to the buffers
(declare (ignore buffer))
(case query ;type of the query
  (state
    (case value
      (busy nil)
      (free t)
      (error nil)
      (t (print-warning "Bad␣state␣query␣to␣the␣~s␣buffer" buffer))
```

```
    );case
  );state
  (touched ;did it touch something?
    (setq result
      (dolist (var (nxt-touch-port nxt)) ;for every sensor
        (if (touched? var) ;if it's pressed
          (return t))
      );dolist
    );setq
    (case value
      (true result) ;true if it touched
      (false (not result)) ;false if it touched
      (t (print-warning "Bad state query to the ~s buffer" buffer))
    );case
  );touched
  (sensor ;did the specified sensor touch something?
    (if (member value (nxt-touch-port nxt))  ;checks if the specified port has a sensor attached
      (touched? value); reads from the sensor
      (print-warning "No touch sensor defined for port ~s, define a new sensor first" value)
    );if
  );sensor
  (sensors ;did ALL the specified sensors touch something? the value must be a list!
    (dolist (var value t)
      (if (member var (nxt-touch-port nxt)) ;if the port is valid
        (if (not (touched? var)) ;if one sensor is not pressed finish and return nil
          (return nil))
        (return (print-warning "No touch sensor defined for port ~s, define a new sensor first" var)) ;
     print-warning returns nil
      );if
    );dolist
  );TODO useful???
    (t (print-warning "Invalid query ~s to the ~s buffer" query buffer))
))


;;;; ------------------------------------------------------------------------ ;;;;
;;;; special nxt functions
;;;; ------------------------------------------------------------------------ ;;;;

(defun nxt-define-sensor (port)
(touch-on port)
)

(define-module-fct 'nxt-touch ;name
                   '((nxt-touched nil nil (touched sensor sensors) nil)) ;buffers, need to define the query
      the buffer will respond to
(list
  (define-parameter :touch-port
    :documentation "port to which the sensors are connected"
    :default-value '(1 2)
    :valid-test (lambda (x) (and
            (listp x) ;must be a list
            (dolist (var x t)
              (if (or (not (numberp var)) (< var 1) (> var 4))
```

```
                (return nil)) ;must be a number between 1 and 4
           )
          ))
    :warning "a list of valid ports"
    :owner t
  )
)
;;    :request 'nxt-touch-requests
   :query 'nxt-touch-queries

   :version "0.1"
   :documentation "Touch sensor module for the lego nxt brick"
   :creation 'nxt-touch-create
   :reset 'nxt-touch-reset
   :delete 'nxt-touch-delete
   :params 'nxt-touch-params
)
```

A.3. nxt-vision

```
#+:packaged-actr (in-package :act-r)
#+(and :clean-actr (not :packaged-actr) :ALLEGRO-IDE) (in-package :cg-user)
#-(or (not :clean-actr) :packaged-actr :ALLEGRO-IDE) (in-package :cl-user)


(require-compiled "DMI" "ACT-R6:support;dmi")
(require-compiled "GENERAL-PM" "ACT-R6:support;general-pm")

(defvar *previous-color* 0)

(defun poll-sensor (instance) ; a process who checks periodically the sensor and draws a colored letter on
     the experiment window
  (setq port (nxt-vision-port instance))
  (let* ((number (colorsensor (nxt-vision-port instance)))
    (color (case number
      (1 'black)
      (2 'blue)
      (3 'green)
      (4 'yellow)
      (5 'red)
      (6 'white)
      (t (print-warning "Error␣reading␣the␣color,␣setting␣it␣to␣black") 'black)
      );case
    );color
    );let-params
  (if (or (not (eq color *previous-color*)) ;if the color changes
      (eq number 2)) ;of if it-s blue, better to insist if it's close to the wall
    (let ()
      (clear-exp-window)
      (add-text-to-exp-window :text "O" :color color)
      (proc-display :clear t)
    );let
  );if
  (setq *previous-color* color)
  );let
)

(defstruct nxt-vision port)

(defun nxt-vision-create (model-name)
(declare (ignore model-name))
(make-nxt-vision))

(defun nxt-vision-reset (nxt)
(declare (ignore nxt))
(define-chunks (color-chunk isa visual-object))
(chunk-type light turn)
)

(defun nxt-vision-delete (nxt)
(declare (ignore nxt))
)
```

```
(defun nxt-vision-requests (instance buffer-name chunk-spec)
  (case buffer-name
    (nxt-visual
      (case (chunk-spec-chunk-type chunk-spec)
        (light ;used to turn on and off the sensor
          (case (third (car (chunk-spec-slot-spec chunk-spec 'turn)))
            (on (colorsensor-white (nxt-vision-port instance))
              (setq polling-event (schedule-periodic-event 50 'poll-sensor :maintenance t :time-in-ms t :
     params (list instance)))
            )
            (off (colorsensor-off (nxt-vision-port instance))
              (delete-event polling-event) ;stop polling
            )
            (t (print-warning "Wrong␣parameter␣for␣the␣\"light\"␣request"))
          );case
        );light
        (t (print-warning "wrong␣chunk-type␣~s␣for␣the␣buffer␣~s" (chunk-spec-chunk-type chunk-spec)
     buffer-name))
      );case
    );nxt-visual
    (t (print-warning "requested␣the␣wrong␣buffer␣~s" buffer-name))
  );case
)

(defun nxt-vision-queries (nxt buffer query value) ;handles the queries to the buffers
(declare (ignore buffer))
(case query ;type of the query
  (state
    (case value
      (busy nil)
      (free t)
      (error nil)
      (t (print-warning "Bad␣state␣query␣to␣the␣~s␣buffer" buffer))
    );case
  );state
  (t (print-warning "Invalid␣query␣~s␣to␣the␣~s␣buffer" query buffer))
))

(defun nxt-vision-params (nxt param) ;getter and setter for the parameters
(if (consp param)
  (case (car param)
    (:vision-port
      (setf (nxt-vision-port nxt) (cdr param))
    );port
    (t (print-warning "Wrong␣parameter"))
  );case
  (case param
    (:vision-port
      (nxt-vision-port nxt))
    (t (print-warning "Wrong␣parameter"))
  );case
))
```

```
;;;; ------------------------------------------------------------------------ ;;;;
;;;; special nxt functions
;;;; ------------------------------------------------------------------------ ;;;;

(define-module-fct  'nxt-vision ;name
          '(nxt-visual) ;buffers
(list
  (define-parameter :vision-port
    :documentation "port␣to␣which␣the␣sensor␣is␣connected"
    :default-value 3
    :valid-test (lambda (x)
      (and (integerp x) (> x 0) (< x 5)))
    :warning "a␣valid␣port␣number"
    :owner t)
)

   :request 'nxt-vision-requests
   :query 'nxt-vision-queries

   :version "0.7"
   ;;changelog
   ;;v 0.7  now the display is refreshed only if there is a change of color, or if the color is blue: trying
      to save computational time
  ;;    and go closer to the human behaviour
  ;;    corrected a bug into the condition in poll-sensor
   ;;v 0.6  removed the fake device and now using the standard act-r device (a window) and drawing a colored
      letter on that
   ;;     removed the polling process and implemented the same thing through the "schedule-periodic-event"
     act-r function
   ;;v 0.5 = 0.2 + 0.3
   ;;v 0.3  rollbacked to v0.1 and modified the buffer stuffing so that resemble the normal act-r behaviour
   ;;     instead of stuffing a chunk directly in the visual buffer it creates a new visicon device and
     wites in it an oval whose color is the one the sensor reads
   ;;v 0.2  added multiprocessing, now the polling to the color sensor is made by another process
   ;;     this process is created when the sensor is turned on and killed when it's turned off
   ;;     when the function recognizes a change of color, or that the color is blue, stuffs a new chunk into
      the vision buffer
   ;;     the request visual-location has been removed
   :documentation "Motor␣module␣for␣the␣lego␣nxt␣brick"
   :creation 'nxt-vision-create
   :reset 'nxt-vision-reset
   :delete 'nxt-vision-delete
   :params 'nxt-vision-params
)
```

A.4. nxt-distance

```
#+:packaged-actr (in-package :act-r)
#+(and :clean-actr (not :packaged-actr) :ALLEGRO-IDE) (in-package :cg-user)
#-(or (not :clean-actr) :packaged-actr :ALLEGRO-IDE) (in-package :cl-user)

;;documentation

;;this module reads the distance from the nxt's ultrasonic sensor
;;when a request is made the module reads an integer value from the sensor and produces as output a chunk in
      it's own buffer
;;the chunk is of type "obstacle" and has a slot called "distance" whose value is the one given by the
     sensor
;;
;;because the function (set-chunk-slot-value) accepts as parameter only strings the module creates a string
     containing the command
;;to execute, including the right numeric value, and then evaluates the string
;;after the value has been changed it issues a buffer stuffing in it's own buffer

(require-compiled "DMI" "ACT-R6:support;dmi")
(require-compiled "GENERAL-PM" "ACT-R6:support;general-pm")

(defstruct nxt-distance port)

(defun nxt-distance-create (model-name)
(declare (ignore model-name))
(make-nxt-distance))

(defun nxt-distance-reset (nxt)
(declare (ignore nxt))
(chunk-type obstacle distance counter)
(define-chunks (read isa obstacle))
)

(defun nxt-distance-delete (nxt)
(declare (ignore nxt))
)

(defun nxt-distance-requests (instance buffer-name chunk-spec)
  (case buffer-name
    (nxt-distance
      (case (chunk-spec-chunk-type chunk-spec)
        (obstacle
          (setq measures 0.0)
          (dotimes (x 10) ;make 10 reads
            (setf measures (+ measures (distance (nxt-distance-port instance))))
          );dotimes
          (setf measures (/ measures 10.0));mean value
          (EVAL (READ-FROM-STRING (format nil "(set-chunk-slot-value␣read␣distance␣~D)" measures))) ;
     reading the distance
          (EVAL (READ-FROM-STRING (format nil "(set-chunk-slot-value␣read␣counter␣~D)" (third (car (
     chunk-spec-slot-spec chunk-spec 'counter)))))) ;setting the counter
          (schedule-event-relative 0.1 'set-buffer-chunk ;scheduling the action to fill the buffer, so that
      it will be available for the next production
```

```
            :priority :min
            :params '(nxt-distance read :requested t) ;put into the $1 buffer the $2 chunk
            :output nil)
        );distance
        (t (print-warning "wrong␣chunk-type␣~s␣for␣the␣buffer␣~s" (chunk-spec-chunk-type chunk-spec)
     buffer-name))
      );case
    );nxt-distance
    (t (print-warning "requested␣the␣wrong␣buffer␣~s" buffer-name))
  );case
)

(defun nxt-distance-queries (nxt buffer query value) ;handles the queries to the buffers
(declare (ignore buffer))
(case query ;type of the query
  (state
    (case value
      (busy nil)
      (free t)
      (error nil)
      (t (print-warning "Bad␣state␣query␣to␣the␣~s␣buffer" buffer))
    );case
  );state
  (t (print-warning "Invalid␣query␣~s␣to␣the␣~s␣buffer" query buffer))
))

(defun nxt-distance-params (nxt param) ;getter and setter for the parameters
(if (consp param)
  (case (car param)
    (:distance-port
      (setf (nxt-distance-port nxt) (cdr param))
      (distance-on (nxt-distance-port nxt))
    );port
    (t (print-warning "Wrong␣parameter"))
  );case
  (case param
    (:distance-port
      (nxt-distance-port nxt))
    (t (print-warning "Wrong␣parameter"))
  );case
))

;;;; ------------------------------------------------------------------------ ;;;;
;;;; special nxt functions
;;;; ------------------------------------------------------------------------ ;;;;

(define-module-fct  'nxt-distance ;name
          '(nxt-distance) ;buffers
(list
  (define-parameter :distance-port
    :documentation "port␣to␣which␣the␣sensor␣is␣connected"
    :default-value 4
    :valid-test (lambda (x)
      (and (integerp x) (> x 0) (< x 5)))
```

```
    :warning "a␣valid␣port␣number"
    :owner t)
)

   :request 'nxt-distance-requests
   :query 'nxt-distance-queries

   :version "0.1"
   :documentation "module␣for␣the␣lego␣nxt␣brick's␣ultrasonic␣sensor"
   :creation 'nxt-distance-create
   :reset 'nxt-distance-reset
   :delete 'nxt-distance-delete
   :params 'nxt-distance-params
)
```

A.5. Simulator

```
;;changelog
;;v 0.2 added the fast_raw option, if the variable is false it will draw the dynamic map

(setf *mazeList* (list
  ;; index 0: first maze, goal at edge position
  (make-array '(26 19) ;; define the number of rows and columns
  ;; here comes the actual content: x=wall, -=accessible, g=goal, d=door
  ;; rooms have to be rectangular!
  :initial-contents'(
    (x x x x x x x x x x x x x x x x x x x)
    (x x x - * - x x x x x x - - j - g x x)
    (x x x x d x x x x x x x x x d x d x x)
    (x - x - - - x x x x x - d - j x - x x)
    (x * d - j - d - - - d j x * j d j x x)
    (x - x - - - x x x x x - d - j x - x x)
    (x x x x d x x x x x x x x x d x x x x)
    (x x x x - * x x x x x x - - j - - x x)
    (x x x x - - x x x x x x d x x x d x x)
    (x x x x x d x x x x x x - j - j - x x)
    (x x x - - j - - x x x x x d x d x x x)
    (x - x - - * - - x - - - x - x - x x x)
    (x - x - - - - - x d x d x - x j d - x)
    (x * d j * - * j d j x j d j x - x - x)
    (x - x - - - - - x d x d x - x - x - x)
    (x - x d x x x d x - * - x - x j d - x)
    (x x x - - j - - x - j - x d x d x x x)
    (x x x x x d x x x x d x x - j - x x x)
    (x x x x x - x x x x - x x x d x x x x)
    (x - * - x - x x x x d x x x - - - * x)
    (x * - - d - x x - - * - x x * - - - x)
    (x - - - x x x x - - - - x x - - - - x)
    (x x d x x x x x x x x x x x x x d x x)
    (x - - - - - - - - * - - - - - - - - x)
    (x * - - - - - - - s - - - - - - - * x)
    (x x x x x x x x x x x x x x x x x x x))
  )
  ) ;; list
) ;; setf

;; the distance of the optimal path
(setf *mazeOptDist* (list 29.052809 23.727083 44.988667 32.03491))
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                   TOOLS                                    ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun init (num try)
  (setf index num)
  (setf window_name (format nil "maze␣~D␣run␣~D" num try))
  (setf *mazeArray* (nth index *mazeList*))
  (setf *shortest* (nth index *mazeOptDist*))
  (setf width (array-dimension *mazeArray* 1))
  (setf height (array-dimension *mazeArray* 0))
```

```
  (setf *north* 0)
  (setf *east* 1)
  (setf *south* 2)
  (setf *west* 3)
  (reset_sym)
  ;; the way that the user took
  (setf *userTrace* (list *userPos*))
  (setf *listPAO* ())
  (setf textSize 8)
  (setf last_visloc ())
  (setq fast_draw t)
  (setq *previous-color* 0)
)

(defun reset_sym ()
  (setf *userPos* (getStartPos)) ;;first is the row, second the column
  ;; the direction in which the user looks: N/E/S/W = 0/1/2/3/
  (setf *userDir* *north*)
  (setf *userTrace* (list *userPos*))
)

;; search start position from left bottom
(defun getStartPos ()
  (getStartPosHelper *mazeArray* (- (array-dimension *mazeArray* 0) 1) 0)
  ;'(34 14)
)

(defun getStartPosHelper (arr row col)
  (cond
    ((>= col (array-dimension arr 1)) ;; too far right
      (getStartPosHelper arr (- row 1) 0)
    )
    ((< row 0) ;; too far up, no start position
      (print "ERROR: no start position found")
      nil
    )
    ((string= (aref arr row col) "S")
      (list row col)
    )
    (t
      (getStartPosHelper arr row (+ col 1))
    )
  )
)

(defun checkPosition (row col)
  ;; userPos should be inside maze
  (if (not (or (< col 0) (< row 0) (> col (- (array-dimension *mazeArray* 1) 1)) (> row (- (array-dimension
      *mazeArray* 0) 1))))
    t ;(or (string= (aref *mazeArray* row col) "D") (not *initDone*))
    nil)
)

(defun getDistance (traceList dist)
```

```
  (let ((cur (pop traceList)))
    (if (null traceList)
      dist
      (getDistance traceList (+ dist (euclidianDistance cur (first traceList))))
    )
  )
)

(defun getPAO ()
  (* 100 (/ (- (getDistance *userTrace* 0) *shortest*) *shortest*))
)

(defun euclidianDistance (pos1 pos2)
  (sqrt (+ (square (- (first pos1) (first pos2))) (square (- (second pos1) (second pos2)))))
)

(defun square (num)
  (* num num)
)

(defun copy-array (arr)
  (let ((dims (array-dimensions arr)))
    (adjust-array
      (make-array dims :displaced-to arr)
      dims)
  )
)

(defun listSum (sum theList)
  (if (null theList)
    sum
    (listSum (+ (pop theList) sum) theList)
  )
)

(defun listMean (theList)
  (/ (listSum 0 theList) (list-length theList) 1.0)
)

(defun switchmodel ()
  (setf *actr-enabled-p* (not *actr-enabled-p*))
  (format t "ACT-R␣is␣~Denabled.~%" (if *actr-enabled-p* "" "not␣"))
)

(defun help ()
  (format t "~%~%MAZES:␣(format:␣index=number,targetPosition)~%0=first,edge;␣1=first,center;␣2=second,edge;
    ␣3=second,center~%")
  (format t "~%Start␣experiment:␣(e␣maxTime␣doRealTime␣mazeIndex␣verbose)~%Start␣series␣of␣experiments:␣(
    sampleDataInd␣numExperiments␣mazeIndex␣maxTime)~%")
  (format t "~%Switch␣ACT-R␣model␣ON/OFF:␣(switchmodel)~%~%Currently,␣ACT-R␣is␣~Denabled.~%" (if *
    actr-enabled-p* "" "not␣"))
  (format t "~%To␣show␣this␣help␣again,␣type␣(help)~%")
)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                                FUNCTIONS                                ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun next_cell ()
  (case *userDir*
    (0 ;north
      (setf row (- (car *userPos*) 1))
      (setf col (cadr *userPos*))
      (if (checkPosition row col)
        (list row col)
        *userPos*)
    )
    (1 ;east
      (setf row (car *userPos*))
      (setf col (+ (cadr *userPos*) 1))
      (if (checkPosition row col)
        (list row col)
        *userPos*))
    (2 ;south
      (setf row (+ (car *userPos*) 1))
      (setf col (cadr *userPos*))
      (if (checkPosition row col)
        (list row col)
        *userPos*))
    (3 ;west
      (setf row (car *userPos*))
      (setf col (- (cadr *userPos*) 1))
      (if (checkPosition row col)
        (list row col)
        *userPos*))
    (t (prin1 "ERROR"))
  )
)

(defun draw_user_pos_fast ()
  (clear-exp-window)
  (add-text-to-exp-window :text "O" :color color
    :x (* (cadr *userPos*) textSize)
    :y (* (car *userPos*) textSize)
    :width textSize)
  (proc-display :clear t)
)

(defun draw_user_pos ()
  (setf coords (next_cell)) ;read the coords of the cell it's facing
  (setf cell (aref *mazeArray* (car coords) (cadr coords)))
  (case cell
    ('s
        (add-text-to-exp-window :text "S"
        :x (+
          (* width (* textSize 2))
          (* (cadr coords) (* textSize 2)))
        ;(* (cadr coords) (* textSize 2))
```

```
    :y (* (car coords) (* textSize 2))
    :width textSize
    :color 'green)
  ; )
  )
('g
  (add-text-to-exp-window :text "G"
    :x (+
      (* width (* textSize 2))
      (* (cadr coords) (* textSize 2)))
    ;(* (cadr coords) (* textSize 2))
    :y (* (car coords) (* textSize 2))
    :width textSize
    :color 'red)
  ; )
  )
('x
  (add-text-to-exp-window :text "X"
    :x (+
      (* width (* textSize 2))
      (* (cadr coords) (* textSize 2)))
    ;(* (cadr coords) (* textSize 2))
    :y (* (car coords) (* textSize 2))
    :width textSize
    :color 'blue)
  ; )
  )
('*
  (add-text-to-exp-window :text "X"
    :x (+
      (* width (* textSize 2))
      (* (cadr coords) (* textSize 2)))
    ;(* (cadr coords) (* textSize 2))
    :y (* (car coords) (* textSize 2))
    :width textSize
    :color 'blue)
  ; )
  )
('j
  (add-text-to-exp-window :text "J"
    :x (+
      (* width (* textSize 2))
      (* (cadr coords) (* textSize 2)))
    ;(* (cadr coords) (* textSize 2))
    :y (* (car coords) (* textSize 2))
    :width textSize
    :color 'yellow)
  ; )
  )
('-
  (add-text-to-exp-window :text "-"
    :x (+
      (* width (* textSize 2))
      (* (cadr coords) (* textSize 2)))
```

```
        ;(* (cadr coords) (* textSize 2))
        :y (* (car coords) (* textSize 2))
        :width textSize
        :color 'white)
      ; )
      )
    ('d
      (add-text-to-exp-window :text "-"
        :x (+
          (* width (* textSize 2))
          (* (cadr coords) (* textSize 2)))
        ;(* (cadr coords) (* textSize 2))
        :y (* (car coords) (* textSize 2))
        :width textSize
        :color 'white)
      ; )
      )
    (t (prin1 "WRONG SYMBOL"))
  );case
  (remove-items-from-exp-window last_visloc)
  (setf last_visloc (add-text-to-exp-window :text "O" :color color
    :x (* (cadr *userPos*) (* textSize 2))
    :y (* (car *userPos*) (* textSize 2))
    :width textSize))
)

(defun move_forward ()
  (when (not (eq (read_color) 'blue)) ;if the next cell is not a wall
    (if (not fast_draw)
      (draw_user_pos)
    );if
    (setf *userPos* (next_cell)) ; it can move forward
    (when (eq 'd (aref *mazeArray* (car *userPos*) (cadr *userPos*)))
      (push *userPos* *userTrace*)
    );if
    (when (eq 'g (aref *mazeArray* (car *userPos*) (cadr *userPos*)))
      (push *userPos* *userTrace*)
    );if
  );when
)

(defun move_backwards ()
  (turn_right)
  (turn_right)
  (move_forward)
  (turn_right)
  (turn_right)
)

(defun turn_right ()
  (if (eq *userDir* *west*)
    (setf *userDir* *north*)
    (setf *userDir* (+ *userDir* 1))
  )
```

```
)

(defun turn_left ()
  (if (eq *userDir* *north*)
    (setf *userDir* *west*)
    (setf *userDir* (- *userDir* 1))
  )
)

(defun read_color ()
  (setf coords (next_cell)) ;read the coords of the cell it's facing
  (setf cell (aref *mazeArray* (car coords) (cadr coords)))
  (case cell
    ('s 'green)
    ('g 'red)
    ('x 'blue)
    ('* 'blue)
    ('j 'yellow)
    ('- 'white)
    ('d 'white)
    (t (prin1 "WRONG␣SYMBOL"))
  );case
)

(defun get_distance ()
  (setf dist 0)
  (setf bak *userPos*)
  (while (not (eq (read_color) 'blue))
    (setf dist (+ dist 1))
    (setf *userPos* (next_cell))
  )
  (setf *userPos* bak)
  dist
)
```

B. ACT-R Model

```
(setq *performances* ()) ;the number of steps to complete every run
(setq *minDist* 0) ;the minimum distance under which a wall is detected. for the simulator 0, robot 30
(setq *stepFwd* 0.1) ;how much go forward at every step. for the simulator 0.1
(setq *stepTurn* 1) ;how much turn at every turn. for the simulator 1.0, robot 1.1
(setq *forgetRate* 1) ;the bigger, the quicker junctions are forgotten
(setq *similThresh* 0) ;the max difference in the distances over which a junction won'b be retrieved
(setq *utility_thresh* 5.0) ;with a threshold of 1 a way utility of 50 will have the same probability, under
     50 the known way is more probable and over 50 the exploration is prefered
;;gaussian parameters
(setq *s* 0.5)
(setq *frontCenter* 2)
(setq *rightCenter* 4)
;waypoints
(setq *wplist* '((-2 -2 -2 -2)))
(setq *wp1index* 0)
(setq *wp2index* 1)

(defun next-waypoint ()
  (setq *wp2index* (1+ *wp2index*))
  (setq lng (- (length *wplist*) 1))
  (if (> *wp2index* lng)
    (let ()
      (setq *wp1index* (1+ *wp1index*))
      (if (>= *wp1index* lng)
        (let ()
          (setq *wp1index* lng)
          (setq *wp2index* 0)
          nil
        );let
        (setq *wp2index* (1+ *wp1index*))
      );if
    );let
    t
  ))

(defun reduce_activ (n1 e1 s1 w1 n2 e2 s2 w2)
  (with-output-to-string (*STANDARD-OUTPUT*) ;suppress output
    (setq real-name (EVAL (READ-FROM-STRING
      (format nil "(sdm␣isa␣waypoint␣=␣n1␣~s␣=␣e1␣~s␣=␣s1␣~s␣=␣w1␣~s␣=␣n2␣~s␣=␣e2␣~s␣=␣s2␣~s␣=␣w2␣~s)"
      n1 e1 s1 w1 n2 e2 s2 w2))))
    (EVAL (READ-FROM-STRING (format nil "(set-base-levels␣(␣~s␣~D))" (car real-name) 1)))))

(defun get-n1 ()
  (car (nth *wp1index* *wplist*)))
```

```
(defun get-e1 ()
  (cadr (nth *wp1index* *wplist*)))

(defun get-s1 ()
  (caddr (nth *wp1index* *wplist*)))

(defun get-w1 ()
  (cadddr (nth *wp1index* *wplist*)))

(defun get-n2 ()
  (car (nth *wp2index* *wplist*)))

(defun get-e2 ()
  (cadr (nth *wp2index* *wplist*)))

(defun get-s2 ()
  (caddr (nth *wp2index* *wplist*)))

(defun get-w2 ()
  (cadddr (nth *wp2index* *wplist*)))

(defun turn-right (direction)
  (case direction
    ('north 'east)
    ('east  'south)
    ('south 'west)
    ('west  'north)
  ))

(defun turn-left (direction)
  (case direction
    ('north 'west)
    ('west  'south)
    ('south 'east)
    ('east  'north)
  ))

(defun turn-back (direction)
  (case direction
    ('north 'south)
    ('east  'west)
    ('south 'north)
    ('west  'east)
  ))

(defun get-movement (start end) ;which movement must i do to go from start direction to end direction?
  (case start
    ('north
      (case end
        ('north 'front)
        ('east 'right)
        ('west 'left)
      )
```

```
    )
    ('east (case end
        ('east 'front)
        ('south 'right)
        ('north 'left)
      )
    )
    ('south (case end
        ('south 'front)
        ('west 'right)
        ('east 'left)
      )
    )
    ('west (case end
        ('west 'front)
        ('north 'right)
        ('south 'left)
      )
    )
    (t nil) ;matches with the values "closed" and (in case of an error) "empty"
  ))

(defun give-rewards (past current) ;counter of the last and current goals
  (with-output-to-string (*STANDARD-OUTPUT*) ;suppress output
    (setq junction-list (sdm isa junction)) ;save all the crossings
    (setq name-last (EVAL (READ-FROM-STRING (format nil "(CAR␣(SDM␣ISA␣GOAL␣=␣COUNTER␣~s))" past))))
    (setq time-last (EVAL (READ-FROM-STRING (format nil "(CaAR␣(SDP␣~s␣:creation-time))" name-last)))) ;
     creation time of the last goal
  );with-output-to-string
  (setq last-run ()) ;the list containing the junctions of the last run
  (dolist (junction junction-list t) ;selects the chunks that have been used in the last run
    (setq element-time nil)
    (with-output-to-string (*STANDARD-OUTPUT*) (setf element-time (EVAL (READ-FROM-STRING (format nil "(
     CaaAR␣(SDP␣~s␣:reference-list))" junction)))))
    (if (not element-time) ;it haven't been referenced yet
      (with-output-to-string (*STANDARD-OUTPUT*) (setf element-time (EVAL (READ-FROM-STRING (format nil "(
     CaAR␣(SDP␣~s␣:creation-time))" junction)))))
    );if
    (if (> element-time time-last) ;the current chunk belongs to the last run
      (let (  (new_performance (round (- (mp-time) element-time))) ;difference between the actual (goal)
     time and the chunk's execution time
           (old_performance (EVAL (READ-FROM-STRING (format nil "(chunk-slot-value␣~s␣performance)" junction
     )))))
        (if   (and (numberp old_performance) ;if the value is nil it isn't a valid junction, no need to
     update the value
           (not (eq old_performance 9999)) ;it isn't a dead end
           (or (eq old_performance -1) ;if it wasn't rated yet
           (< new_performance old_performance))) ;the new value is better than the last one
          (EVAL (READ-FROM-STRING (format nil "(set-chunk-slot-value␣~s␣performance␣~D)" junction
     new_performance)))
        );if
      );let
      ; (setf last-run (append last-run (list junction)))
    );if
```

```
  ))


;; When in a junction the model retrieves all the chunks about that junction that there are in dm.
;; If the retrieval fails it means that a way is not being tried yet, or it's a wall.
;; If the junction is new, or there's a wall in one side, the model won't fire the best-* productions
     because the performances in the goal buffer will be 9999
;; If a chunk matches, one of those performances will take a positive value, less than 9999.
;; In that case both the productions best-* and go-*-random can fire, an offset is applied on the utility
     calculation
;; this offset is a random positive number proportional to the disance from the goal, minus a threshold.
;; If the known way brings quickly to the goal, the utility of the best-* production will likely be a
     positive number, so the model will follow that way, ignoring the unexplored one
;; If the known way is a long way to the goal, the utility will likely be a negative number, so the model
     will explore the new way


(defun calculate-utility (prod_name) ;add to the utility a random number, in inverse proportion to the
     distance from the goal
  (case prod_name
    ('best-front
      (with-output-to-string (*STANDARD-OUTPUT*) (setq utility
        (EVAL (READ-FROM-STRING (format nil "(chunk-slot-value␣~s␣front)" ;read the utility in that
     direction
          (car (BUFFER-CHUNK GOAL)) ;from the chunk in the goal buffer
        )))))
      (- (act-r-random (/ 100.0 utility)) ;a random positive number between 1/100 and 100
        *utility_thresh*) ;the threshold under which the random production is choosen
    )
    ('best-right
      (with-output-to-string (*STANDARD-OUTPUT*) (setq utility
        (EVAL (READ-FROM-STRING (format nil "(chunk-slot-value␣~s␣right)" ;read the utility in that
     direction
          (car (BUFFER-CHUNK GOAL)) ;from the chunk in the goal buffer
        )))))
      (- (act-r-random (/ 100.0 utility)) ;a random positive number between 1/100 and 100
        *utility_thresh*) ;the threshold under which the random production is choosen
    )
    ('best-left
      (with-output-to-string (*STANDARD-OUTPUT*) (setq utility
        (EVAL (READ-FROM-STRING (format nil "(chunk-slot-value␣~s␣left)" ;read the utility in that
     direction
          (car (BUFFER-CHUNK GOAL)) ;from the chunk in the goal buffer
        )))))
      ( - (act-r-random (/ 100.0 utility)) ;a random positive number between 1/100 and 100
        *utility_thresh*) ;the threshold under which the random production is choosen
    )
    ;implementing the perimeter strategy
    ('go-north-random
      (if (eq *userDir* *west*);
        (+ (act-r-noise *s*) *rightCenter*) ;right
        (if (eq *userDir* *north*);
          (+ (act-r-noise *s*) *frontCenter*) ;straight
          (act-r-noise *s*) ;left
```

```
        );if
      );if
    )
    ('go-east-random
      (if (eq *userDir* *north*)
        (+ (act-r-noise *s*) *rightCenter*) ;right
        (if (eq *userDir* *east*);
          (+ (act-r-noise *s*) *frontCenter*) ;straight
          (act-r-noise *s*) ;left
        );if
      );if
    )
    ('go-south-random
      (if (eq *userDir* *east*)
        (+ (act-r-noise *s*) *rightCenter*) ;right
        (if (eq *userDir* *south*);
          (+ (act-r-noise *s*) *frontCenter*) ;straight
          (act-r-noise *s*) ;left
        );if
      );if
    )
    ('go-west-random
      (if (eq *userDir* *south*)
        (+ (act-r-noise *s*) *rightCenter*) ;right
        (if (eq *userDir* *west*);
          (+ (act-r-noise *s*) *frontCenter*) ;straight
          (act-r-noise *s*) ;left
        );if
      );if
    )
  ))

(define-model reinforced_learning
(install-device (open-exp-window window_name ;"Model's view"
          :visible t
          :width (+ 10 (* (array-dimension *mazeArray* 1) (* 4 textSize)))
          :height (+ 100 (* (array-dimension *mazeArray* 0) (* 2 textSize)))
        ))

(sgp :trace-detail low
  :esc  t ;activation enabled
  :er   t ;random choice of chunks with same activation
  :le   0 ;latency exponent parameter, set to 0 to avoid underflows with high activation values
  :bll  0.2 ;activates learning: needed to record the references of a chunk
  :blc  5
  :act  nil ;activation trace disabled
  :ol 1 ;enables the recording of chunk references
  :ul t :v t  ;enables utility learning
  ; :rt 0.1
  :ult nil ;utility trace
  :utility-offsets calculate-utility ;adds a random component to the utility calculation
  :test-feats nil ;speeds up the draw process in proc-display
  ; :epl t ;production compilation)
```

```
(chunk-type planning state)
(chunk-type goal counter)
(chunk-type fork direction front right left) ;support chunk, used to store temporarly useful information
(chunk-type junction turn north east south west performance)
(chunk-type ext-junction turn north east south west performance front right left)
(chunk-type movement direction counter)
;possible chunks for the direction slot
(chunk-type north)
(chunk-type east)
(chunk-type south)
(chunk-type west)
(chunk-type forwards)
;waypoint calculation
(chunk-type waypoint ; values of -1 indicate the goal and values of -2 indicate the start point
  ;first waypoint
  n1 e1 s1 w1
  ;second waypoint
  n2 e2 s2 w2
  distance)
(chunk-type Ausgabe Unterstatus
  ;first waypoint
  n1 e1 s1 w1
  ;second waypoint
  n2 e2 s2 w2)
(chunk-type Antwort Wert Entfernung
  ;first waypoint
  n1 e1 s1 w1
  ;second waypoint
  n2 e2 s2 w2)
(chunk-type Zahl Name Wert Nachfolger)                              ; Typ zur Repraesentierung der Zahlen.
(chunk-type Addition Wert1 Wert2 Ergebniss)

(add-dm (goal isa planning) (start isa goal counter 0) (move isa movement direction north counter 0)
  (Eins   isa Zahl Name "Eins"    Wert 1  Nachfolger "Zwei")
  (Zwei   isa Zahl Name "Zwei"    Wert 2  Nachfolger "Drei")
  (Drei   isa Zahl Name "Drei"    Wert 3  Nachfolger "Vier")
  (Vier   isa Zahl Name "Vier"    Wert 4  Nachfolger "Fuenf")
  (Fuenf    isa Zahl Name "Fuenf"   Wert 5  Nachfolger "Sechs")
  (Sechs    isa Zahl Name "Sechs"   Wert 6  Nachfolger "Sieben")
  (Sieben   isa Zahl Name "Sieben"    Wert 7  Nachfolger "Acht")
  (Acht   isa Zahl Name "Acht"    Wert 8  Nachfolger "Neun")
  (Neun   isa Zahl Name "Neun"    Wert 9  Nachfolger "Zehn")
  (Zehn   isa Zahl Name "Zehn"    Wert 10 Nachfolger "Elf")
  (Elf    isa Zahl Name "Elf"     Wert 11 Nachfolger "Zwoelf")
  (Zwoelf   isa Zahl Name "Zwoelf"    Wert 12 Nachfolger "Dreizehn")
  (Dreizehn isa Zahl Name "Dreizehn"  Wert 13 Nachfolger "Vierzehn")
  (Vierzehn isa Zahl Name "Vierzehn"  Wert 14 Nachfolger "Fuenfzehn")
  (Fuenfzehn  isa Zahl Name "Fuenfzehn" Wert 15 Nachfolger "Sechzehn")
  (Sechzehn isa Zahl Name "Sechzehn"  Wert 16 Nachfolger "Siebzehn")
  (Siebzehn isa Zahl Name "Siebzehn"  Wert 17 Nachfolger "Achtzehn")
  (Achtzehn isa Zahl Name "Achtzehn"  Wert 18 Nachfolger "Neunzehn")
  (Neunzehn isa Zahl Name "Neunzehn"  Wert 19 Nachfolger "Zwanzig")
  (Zwanzig  isa Zahl Name "Zwanzig"   Wert 20 Nachfolger "Einundzwanzig")
  (Einundzwanzig    isa Zahl    Name "Einundzwanzig"    Wert 21   Nachfolger "Zweiundzwanzig")
```

```
(Zweiundzwanzig   isa Zahl   Name "Zweiundzwanzig"    Wert 22   Nachfolger "Dreiundzwanzig")
(Dreiundzwanzig   isa Zahl   Name "Dreiundzwanzig"    Wert 23   Nachfolger "Vierundzwanzig")
(Vierundzwanzig   isa Zahl   Name "Vierundzwanzig"    Wert 24   Nachfolger "Fuenfundzwanzig")
(Fuenfundzwanzig    isa Zahl   Name "Fuenfundzwanzig" Wert 25   Nachfolger "Sechundzwanzig")
(Sechsundzwanzig  isa Zahl  Name "Sechsundzwanzig"    Wert 26   Nachfolger "Siebenundzwanzig")
(Siebenundzwanzig isa Zahl Name "Siebenundzwanzig"    Wert 27   Nachfolger "Achtundzwanzig")
(Achtundzwanzig   isa Zahl   Name "Achtundzwanzig"    Wert 28   Nachfolger "Neunundzwanzig")
(Neunundzwanzig   isa Zahl   Name "Neunundzwanzig"    Wert 29   Nachfolger "Dreisig")

(Dreisig      isa Zahl Name "Dreisig"         Wert 30        Nachfolger "Einunddreisig")
(Einunddreisig    isa Zahl    Name "Einunddreisig"    Wert 31     Nachfolger "Zweiunddreisig")
(Zweiunddreisig   isa Zahl   Name "Zweiunddreisig"    Wert 32     Nachfolger "Dreiunddreisig")
(Dreiunddreisig   isa Zahl   Name "Dreiunddreisig"    Wert 33     Nachfolger "Vierunddreisig")
(Vierunddreisig   isa Zahl   Name "Vierunddreisig"    Wert 34     Nachfolger "Fuenfunddreisig")
(Fuenfunddreisig    isa Zahl   Name "Fuenfunddreisig" Wert 35     Nachfolger "Sechunddreisig")
(Sechsunddreisig  isa Zahl  Name "Sechsunddreisig"    Wert 36   Nachfolger "Siebenunddreisig")
(Siebenunddreisig isa Zahl Name "Siebenunddreisig"    Wert 37     Nachfolger "Achtunddreisig")
(Achtunddreisig   isa Zahl   Name "Achtunddreisig"    Wert 38     Nachfolger "Neununddreisig")
(Neununddreisig   isa Zahl   Name "Neununddreisig"    Wert 39   Nachfolger "Vierzig")
(Vierzig      isa Zahl Name "Vierzig"         Wert 40        Nachfolger "Einundvierzig")
(Einundvierzig    isa Zahl    Name "Einundvierzig"    Wert 41     Nachfolger "Zweiundvierzig")
(Zweiundvierzig   isa Zahl   Name "Zweiundvierzig"    Wert 42     Nachfolger "Dreiundvierzig")
(Dreiundvierzig   isa Zahl   Name "Dreiundvierzig"    Wert 43     Nachfolger "Vierundvierzig")
(Vierundvierzig   isa Zahl   Name "Vierundvierzig"    Wert 44     Nachfolger "Fuenfundvierzig")
(Fuenfundvierzig    isa Zahl   Name "Fuenfundvierzig" Wert 45     Nachfolger "Sechundvierzig")
(Sechsundvierzig  isa Zahl  Name "Sechsundvierzig"    Wert 46   Nachfolger "Siebenundvierzig")
(Siebenundvierzig isa Zahl Name "Siebenundvierzig"    Wert 47     Nachfolger "Achtundvierzig")
(Achtundvierzig   isa Zahl   Name "Achtundvierzig"    Wert 48     Nachfolger "Neunundvierzig")
(Neunundvierzig   isa Zahl   Name "Neunundvierzig"    Wert 49   Nachfolger "Fuenfzig")
(Fuenfzig     isa Zahl Name "Fuenfzig"          Wert 50        Nachfolger "Einundfuenfzig")
(Einundfuenfzig   isa Zahl    Name "Einundfuenfzig"   Wert 51     Nachfolger "Zweiundfuenfzig")
(Zweiundfuenfzig    isa Zahl   Name "Zweiundfuenfzig" Wert 52     Nachfolger "Dreiundfuenfzig")
(Dreiundfuenfzig    isa Zahl   Name "Dreiundfuenfzig" Wert 53     Nachfolger "Vierundfuenfzig")
(Vierundfuenfzig    isa Zahl   Name "Vierundfuenfzig" Wert 54     Nachfolger "Fuenfundfuenfzig")
(Fuenfundfuenfzig   isa Zahl   Name "Fuenfundfuenfzig"  Wert 55     Nachfolger "Sechundfuenfzig")
(Sechsundfuenfzig isa Zahl  Name "Sechsundfuenfzig"   Wert 56   Nachfolger "Siebenundfuenfzig")
(Siebenundfuenfzig  isa Zahl Name "Siebenundfuenfzig"   Wert 57     Nachfolger "Achtundfuenfzig")
(Achtundfuenfzig    isa Zahl   Name "Achtundfuenfzig" Wert 58     Nachfolger "Neunundfuenfzig")
(Neunundfuenfzig    isa Zahl   Name "Neunundfuenfzig" Wert 59   Nachfolger "Sechzig")
(Sechzig      isa Zahl Name "Sechzig"         Wert 60        Nachfolger "Einundsechzig")
(Einundsechzig    isa Zahl    Name "Einundsechzig"    Wert 61     Nachfolger "Zweiundsechzig")
(Zweiundsechzig   isa Zahl   Name "Zweiundsechzig"    Wert 62     Nachfolger "Dreiundsechzig")
(Dreiundsechzig   isa Zahl   Name "Dreiundsechzig"    Wert 63     Nachfolger "Vierundsechzig")
(Vierundsechzig   isa Zahl   Name "Vierundsechzig"    Wert 64     Nachfolger "Fuenfundsechzig")
(Fuenfundsechzig    isa Zahl   Name "Fuenfundsechzig" Wert 65     Nachfolger "Sechundsechzig")
(Sechsundsechzig  isa Zahl  Name "Sechsundsechzig"    Wert 66   Nachfolger "Siebenundsechzig")
(Siebenundsechzig isa Zahl Name "Siebenundsechzig"    Wert 67     Nachfolger "Achtundsechzig")
(Achtundsechzig   isa Zahl   Name "Achtundsechzig"    Wert 68     Nachfolger "Neunundsechzig")
(Neunundsechzig   isa Zahl   Name "Neunundsechzig"    Wert 69   Nachfolger "Siebzig")
(Siebzig      isa Zahl Name "Siebzig"         Wert 70        Nachfolger "Einundsiebzig")
(Einundsiebzig    isa Zahl    Name "Einundsiebzig"    Wert 71     Nachfolger "Zweiundsiebzig")
(Zweiundsiebzig   isa Zahl   Name "Zweiundsiebzig"    Wert 72     Nachfolger "Dreiundsiebzig")
(Dreiundsiebzig   isa Zahl   Name "Dreiundsiebzig"    Wert 73     Nachfolger "Vierundsiebzig")
(Vierundsiebzig   isa Zahl   Name "Vierundsiebzig"    Wert 74     Nachfolger "Fuenfundsiebzig")
```

```
  (Fuenfundsiebzig    isa Zahl   Name "Fuenfundsiebzig" Wert 75     Nachfolger "Sechundsiebzig")
  (Sechsundsiebzig  isa Zahl  Name "Sechsundsiebzig"    Wert 76   Nachfolger "Siebenundsiebzig")
  (Siebenundsiebzig isa Zahl Name "Siebenundsiebzig"    Wert 77     Nachfolger "Achtundsiebzig")
  (Achtundsiebzig   isa Zahl   Name "Achtundsiebzig"    Wert 78     Nachfolger "Neunundsiebzig")
  (Neunundsiebzig   isa Zahl   Name "Neunundsiebzig"    Wert 79   Nachfolger "Achtzig")
  (Achtzig      isa Zahl Name "Achtzig"         Wert 80        Nachfolger "Einundachtzig")
  (Einundachtzig    isa Zahl    Name "Einundachtzig"    Wert 81     Nachfolger "Zweiundachtzig")
  (Zweiundachtzig   isa Zahl   Name "Zweiundachtzig"    Wert 82     Nachfolger "Dreiundachtzig")
  (Dreiundachtzig   isa Zahl   Name "Dreiundachtzig"    Wert 83     Nachfolger "Vierundachtzig")
  (Vierundachtzig   isa Zahl   Name "Vierundachtzig"    Wert 84     Nachfolger "Fuenfundachtzig")
  (Fuenfundachtzig    isa Zahl   Name "Fuenfundachtzig" Wert 85     Nachfolger "Sechundachtzig")
  (Sechsundachtzig  isa Zahl  Name "Sechsundachtzig"    Wert 86   Nachfolger "Siebenundachtzig")
  (Siebenundachtzig isa Zahl Name "Siebenundachtzig"    Wert 87     Nachfolger "Achtundachtzig")
  (Achtundachtzig   isa Zahl   Name "Achtundachtzig"    Wert 88     Nachfolger "Neunundachtzig")
  (Neunundachtzig   isa Zahl   Name "Neunundachtzig"    Wert 89   Nachfolger "Neunzig")
  (Neunzig      isa Zahl Name "Neunzig"         Wert 90        Nachfolger "Einundneunzig")
  (Einundneunzig    isa Zahl    Name "Einundneunzig"    Wert 91     Nachfolger "Zweiundneunzig")
  (Zweiundneunzig   isa Zahl   Name "Zweiundneunzig"    Wert 92     Nachfolger "Dreiundneunzig")
  (Dreiundneunzig   isa Zahl   Name "Dreiundneunzig"    Wert 93     Nachfolger "Vierundneunzig")
  (Vierundneunzig   isa Zahl   Name "Vierundneunzig"    Wert 94     Nachfolger "Fuenfundneunzig")
  (Fuenfundneunzig    isa Zahl   Name "Fuenfundneunzig" Wert 95     Nachfolger "Sechundneunzig")
  (Sechsundneunzig  isa Zahl  Name "Sechsundneunzig"    Wert 96   Nachfolger "Siebenundneunzig")
  (Siebenundneunzig isa Zahl Name "Siebenundneunzig"    Wert 97     Nachfolger "Achtundneunzig")
  (Achtundneunzig   isa Zahl   Name "Achtundneunzig"    Wert 98     Nachfolger "Neunundneunzig")
  (Neunundneunzig   isa Zahl   Name "Neunundneunzig"    Wert 99   Nachfolger "Hundert"))

(p boot
  =goal>
    isa planning
    state nil
  ==>
  =goal>
    state move
  +retrieval> ;fill in the retrieval buffer for the next production
    isa   movement
  - direction nil
  +nxt-visual>
    isa   light
    turn  on
  ;create the first waypoint
  +imaginal>
    isa   waypoint
    n1    -2
    e1    -2
    s1    -2
    w1    -2
    n2    nil
    e2    nil
    s2    nil
    w2    nil
    distance  0 ;actual counter value
);boot

;;;-----------------------------------------------------------------------
```

```
;;;-----------------------------Moving forward--------------------------
;;;---------------------------------------------------------------------

(p move ;start a new movement
  =goal>
    isa   planning
    state move
  =retrieval>
    isa     movement
    counter   =num ;number of the last movement
  ?visual-location>
    buffer empty ;no communications from the visual sensor
  ?imaginal>
    state free ;the last movement completed
  ?retrieval>
    state free
  ==>
  =goal>
    state track
  !bind! =next (+ =num 1)
  !bind! =move *stepFwd*
  =retrieval>
    direction nil
    counter nil
  +imaginal> ;ask to the imaginal module to create a new chunk
    isa     movement
    direction forwards ;with this direction
    counter   =next ;and the next number
  +nxt-move>
    ISA move-forward
    duration =move)

(p track-movement ;saves in declarative memory the new movement
  =goal>
    ISA   planning
    state   track
  =imaginal> ;now it will be saved in dm
    isa     movement
  ?imaginal>
    state free ;the last movement completed
  ?retrieval>
    state free
  ==>
  =goal>
    state move ;start from the beginning
  +retrieval> ;filling the buffer for the next production
    isa   movement
  - direction nil)

;;;---------------------------------------------------------------------
;;;--------------------------finds an obstacle--------------------------
;;;---------------------------------------------------------------------

(p close ;approaches the wall
```

```
  =goal>
    isa   planning
    state move
  =visual-location> ;something in sight
    ISA visual-location
  =retrieval>
    isa movement
  ?imaginal>
    state free ;the last movement completed
  ?retrieval>
    state free
  ?visual>
    state      free
  ==>
  =goal>
    state close ;stop and investigate
  =retrieval>
  +visual>
    ISA         move-attention
    screen-pos  =visual-location
  +nxt-move>
    isa   emergency-stop)

(p avoid ;avoids the curve
  =goal>
    isa   planning
    state close
  =visual> ;if it's a wall
    isa   text
    value "O"
  = color blue
  =retrieval>
    isa movement
    counter =count
  ?visual>
    state      free
  ?retrieval>
    state free
  ==>
  =goal>
    state avoid-curve
  =retrieval>
    direction nil
    counter nil
  !bind! =move *stepTurn*
  +nxt-move>
    ISA turn-right
    duration =move
  +retrieval> ;load the last infos about the user direction
    isa     movement
  - direction forwards
  - direction nil)

(p junction ;there's a junction
```

```
  =goal>
    isa   planning
    state close
  =visual> ;if it's a wall
    isa   text
    value "O"
  = color yellow
  =retrieval>
    isa   movement
    counter =num
  ?visual>
    state      free
  ?retrieval>
    state free
  ==>
  =goal>
    state approach-junction
  !bind! =next (+ =num 1)
  !bind! =move *stepFwd*
  =retrieval>
    direction nil
    counter nil
  +imaginal> ;ask to the counter module to create a new chunk
    isa     movement
    direction forwards ;with this direction
    counter   =next ;and the next number
  +nxt-move>
    ISA move-forward
    duration =move)

(p approach-junction ;go one step forward to reach the junction and start measuring
  =goal>
    ISA   planning
    state   approach-junction
  =imaginal> ;now it will be saved in dm
    isa     movement
    counter   =count
  ?imaginal>
    state free ;the last movement completed
  =retrieval>
    isa movement
  ==>
  =goal>
    state start-measuring
  =retrieval>
    direction nil
    counter nil
  +imaginal>
    isa   junction
    performance =count ;save the value temporary
  +nxt-move>
    isa emergency-stop)

(p false-alarm
```

```
  =goal>
    isa   planning
    state close
  =visual> ;if it's not a wall
    isa   text
    value "O"
  - color blue
  - color red
  - color yellow
  ?visual>
    state      free
  =retrieval>
    isa movement
  ==>
  =goal>
    state move
  =retrieval>
    ; direction nil
    ; counter nil
  ; +retrieval> ;filling the buffer for the next production
    ; isa   movement
  ; - direction nil)

;;;-------------------------------------------------------------------------
;;;---------------------------finds a goal----------------------------------
;;;-------------------------------------------------------------------------

(p recognise-goal ;recognises a goal
  =goal>
    isa   planning
    state close
  =visual> ;if it's a goal
    isa   text
    value "O"
  = color red
  ?visual>
    state      free
  =retrieval>
    isa movement
  ==>
  =goal>
    state goal
  +nxt-move>
    ISA emergency-stop
  =retrieval>)

(p goal2
  =goal>
    isa   planning
    state goal
  =retrieval>
    isa     movement
    counter   =num ;number of the last movement
  ?visual-location>
```

```
    buffer empty ;no communications from the visual sensor
  ?imaginal>
    state free ;the last movement completed
  ?retrieval>
    state free
  ==>
  =goal>
    state     track-goal
  =retrieval>
    direction nil
    counter nil
  !bind! =next (+ =num 1)
  +imaginal> ;ask to the counter module to create a new chunk
    isa     goal
    counter   =next ;and the next number
  +retrieval>
    isa     goal
  - counter   nil
  ; !eval! (print-warning "PAO: ")
  ; !eval! (prin1 (getPAO))
  ; !eval! (setf *listPAO* (append *listPAO* (list (getPAO))))
  !eval! (reset_sym)  ;reset the user position and orientation in the simulated labyrinth)

(p track-goal ;stores the goal in dm, prints an output in console and starts the simulation from the
     beginning
  =goal>
    isa     planning
    state   track-goal
  =retrieval>
    isa     goal
    counter   =last-goal
  =imaginal> ;now it will be saved in dm
    isa     goal
    counter   =current-goal
  ?retrieval>
    state free
  ==>
  =goal>
    state   take-my-time
  =retrieval>
    counter nil
  +imaginal>
    isa     movement
    direction north ;reset the user orientation in dm
    counter   =current-goal
  +retrieval> ;retrieve the last incomplete waypoint
    isa   waypoint
  - n1    nil
  - e1    nil
  - s1    nil
  - w1    nil
    n2    nil
    e2    nil
    s2    nil
```

```
    w2    nil
  - distance  nil
  !eval! (print-warning "Simulation␣completed␣in␣~D␣steps" (- =current-goal =last-goal))
  !eval! (setf *performances* (append *performances* (list (- =current-goal =last-goal))))
  !eval! (give-rewards =last-goal =current-goal))

(p take-my-time
  =goal>
    isa     planning
    state   take-my-time
  =imaginal>
    isa movement ;saving in dm
    counter   =count ;current counter
  =retrieval>
    isa waypoint
    distance  =old_count
  ==>
  !bind! =dist (- =count =old_count)
  !bind! =n1 (get-n1)
  !bind! =e1 (get-e1)
  !bind! =s1 (get-s1)
  !bind! =w1 (get-w1)
  !bind! =n2 (get-n2)
  !bind! =e2 (get-e2)
  !bind! =s2 (get-s2)
  !bind! =w2 (get-w2)
  +goal>  ; Und aktualisiere den goal buffer.
    isa       Ausgabe
    Unterstatus   Suche_Direkt
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  =retrieval>
    n2  -1
    e2  -1
    s2  -1
    w2  -1
    distance  =dist
  -retrieval>
  !eval! (setq *wplist* (APPEND *WPLIST* '((-1 -1 -1 -1))))
  ;create the first waypoint
  +imaginal>
    isa   waypoint
    n1    -2
    e1    -2
    s1    -2
    w1    -2
    n2    nil
    e2    nil
```

```
    s2    nil
    w2    nil
    distance  =count ;actual counter value
  -imaginal>)

;;;-------------------------------------------------------------------------
;;;-------------------------measuring the distances-------------------------
;;;-------------------------------------------------------------------------

(p start-measuring
  =goal>
    isa   planning
    state start-measuring
  =imaginal>
    isa   junction
    performance =count
  ==>
  =goal>
    state measure-front
  +retrieval> ;load the last infos about the user direction
    isa     movement
  - direction forwards
  - direction nil
  +nxt-distance>
    isa obstacle
    counter nil
  =imaginal> ;setting the default value for all the 4 slots to 0
    north 0
    east  0
    south 0
    west  0)

(p* measure-front ;measuring the front
  =goal>
    isa   planning
    state measure-front
  =retrieval>
    isa     movement
    direction =dir
  =nxt-distance>
    isa obstacle
    distance  =dist
  =imaginal>
    isa junction
  ==>
  =goal>
    state turn-right
  =retrieval>
  =imaginal> ;write the distance in the right slot
    =dir  =dist
  !bind! =move *stepTurn*
  +nxt-move>
    ISA turn-right
    duration =move)
```

```
(p turn-right
  =goal>
    isa   planning
    state turn-right
  =retrieval>
    isa     movement
  =imaginal>
    isa junction
  ==>
  =goal>
    state measure-right
  =retrieval>
  =imaginal>
  +nxt-move>
    ISA emergency-stop
  +nxt-distance> ;reads the distance
    isa obstacle
    counter nil)

(p* measure-right ;measuring the right side
  =goal>
    isa   planning
    state measure-right
  =retrieval>
    isa     movement
    direction =dir ;the direction it was facing when it saw the obstacle
  =imaginal>
    isa   junction
  =nxt-distance>
    isa     obstacle
    distance  =dist
  ==>
  =goal>
    state turn-back
  =retrieval>
  !bind! =side (turn-right =dir)
  =imaginal>
    =side =dist
  !bind! =move *stepTurn*
  +nxt-move>
    ISA turn-right
    duration =move)
(p turn-back
  =goal>
    isa   planning
    state turn-back
  =retrieval>
    isa     movement
  =imaginal>
    isa junction
  ==>
  =goal>
    state measure-back
  =retrieval>
```

```
  =imaginal>
  +nxt-move>
    ISA emergency-stop
  +nxt-distance> ;reads the distance
    isa obstacle
    counter nil)

(p* measure-back ;measuring the back
  =goal>
    isa   planning
    state measure-back
  =retrieval>
    isa     movement
    direction =dir ;the direction it was facing when it saw the obstacle
  =imaginal>
    isa   junction
  =nxt-distance>
    isa     obstacle
    distance  =dist
  ==>
  =goal>
    state turn-left
  =retrieval>
  !bind! =side (turn-back =dir)
  =imaginal>
    =side =dist
  -nxt-distance>
  !bind! =move *stepTurn*
  +nxt-move>
    ISA turn-right
    duration =move)
(p turn-left
  =goal>
    isa   planning
    state turn-left
  =retrieval>
    isa     movement
  =imaginal>
    isa junction
  ==>
  =goal>
    state measure-left
  =retrieval>
  =imaginal>
  +nxt-move>
    ISA emergency-stop
  +nxt-distance> ;reads the distance
    isa obstacle
    counter nil)

(p* measure-left ;measuring the left side
  =goal>
    isa   planning
    state measure-left
```

```
  =retrieval>
    isa     movement
    direction =dir ;the direction it was facing when it saw the obstacle
  =imaginal>
    isa   junction
  =nxt-distance>
    isa     obstacle
    distance  =dist
  ==>
  =goal>
    state load-last-waypoint ;retrieve-junction
  =retrieval>
  !bind! =side (turn-left =dir)
  =imaginal>
    =side =dist
  -nxt-distance>
  !bind! =move *stepTurn*
  +nxt-move>
    ISA turn-right
    duration =move
  +nxt-distance> ;reads the distance
    isa obstacle
    counter nil)

;;;-------------------------------------------------------------------------
;;;----------------------quick response functions-------------------------
;;;-------------------------------------------------------------------------

(p avoid-curve
  =goal>
    isa   planning
    state avoid-curve
  =retrieval>
    isa   movement
  ==>
  =goal>
    state check-wall-right
  =retrieval>
  +nxt-move>
    ISA emergency-stop
  +nxt-distance> ;reads the distance
    isa obstacle
    counter nil)

(p free-right ;there is a wall in front and on the right side, tries left
  =goal>
    isa   planning
    state check-wall-right
  !bind! =min *minDist*
  =nxt-distance>
    isa     obstacle
  > distance  =min ;a wall is not present
  =retrieval>
    isa     movement
```

```
    direction =dir
  ==>
  =goal>
    state track-turn
  !bind! =side (turn-right =dir)
  +imaginal> ;record the movement
    isa   junction
    turn  =side
  =retrieval>
    direction nil
    counter nil
  +retrieval>
    isa   movement
  - direction nil)

(p wall-right ;there is a wall in front and on the right side, tries left
  =goal>
    isa   planning
    state check-wall-right
  !bind! =min *minDist*
  =nxt-distance>
    isa     obstacle
  <=  distance  =min ;a wall is present
  =retrieval>
    isa     movement
    direction =dir
  ==>
  =goal>
    state check-wall-left
  =retrieval>
  !bind! =move (* 2 *stepTurn*)
  +nxt-move>
    ISA turn-left
    duration =move)

(p check-wall-left ;turn left and stop
  =goal>
    isa   planning
    state check-wall-left
  =retrieval>
    isa movement
  ==>
  =goal>
    state wall-right
  =retrieval>
  +nxt-move>
    ISA emergency-stop
  +nxt-distance> ;reads the distance
    isa obstacle
    counter nil)

(p left-free ;there isn't a wall on the left, proceeding this way
  =goal>
    isa   planning
```

```
    state wall-right
  !bind! =min *minDist*
  =nxt-distance>
    isa     obstacle
  > distance  =min ;a wall is not present
  =retrieval>
    isa     movement
    direction =dir ;the direction it was facing
  ==>
  =goal>
    state track-turn
  !bind! =side (turn-left =dir)
  +imaginal> ;record the movement
    isa   junction
    turn  =side
  =retrieval>
    direction nil
    counter nil
  +retrieval>
    isa   movement
  - direction nil)

(p going-back ;there is a wall on the left, going back
  =goal>
    isa   planning
    state wall-right
  !bind! =min *minDist*
  =nxt-distance>
    isa     obstacle
  <=  distance  =min ;a wall is present
  =retrieval>
    isa     movement
    direction =dir ;coming from this direction
  ==>
  =goal>
    state dead-end
  =retrieval>
    direction nil
    counter nil
  +retrieval> ;the last junction before the dead-end
    isa   junction
  - performance nil ;only a valid junction
;;it needs to mark the last called junction as dead end, not always it has value -1 because in case the way
     has been followed a run before, but a dead end
;;is found then the robot boes back on its steps and if it founds a goal later, that junction will be marked
      with a valid performance value
  !bind! =move *stepTurn*
  +nxt-move>
    ISA turn-left
    duration =move ;going back
  !bind! =side (turn-back =dir)
  +imaginal> ;record the movement
    isa   junction
    turn  =side)
```

```
(p dead-end ;update the last action with a terrible performance value
  =goal>
    isa   planning
    state dead-end
  =retrieval> ;the most recent junction chunk, the last choice it made
    isa   junction
  =imaginal>
    isa junction
  ==>
  =goal>
    state track-turn
  =retrieval>
    performance 9999 ;give a terrible performance value, next time it won't turn this way
  =imaginal>
  +retrieval>
    isa movement
  - direction nil)

;;;-------------------------------------------------------------------------
;;;-----------------------choose the direction--------------------------
;;;-------------------------------------------------------------------------

(p track-turn ;stores the decision in dm, needs as inputs the direction in the turn slot of the imaginal
     buffer and the last counter in the counter slot of the retrieval buffer
  =goal>
    isa   planning
    state track-turn
  =imaginal> ;store in dm
    isa   junction
    turn  =dir
  =retrieval>
    isa   movement
    counter =num
  ==>
  =goal>
    state track
  =retrieval>
    direction nil
    counter nil
  !bind! =next (+ =num 1)
  +imaginal> ;ask to the imaginal module to create a new chunk
    isa     movement
    direction =dir ;with this direction
    counter   =num ;=next ;and the next number
  +nxt-move>
    ISA emergency-stop)

(p load-last-waypoint ;loads the last uncomplete waypoint saved in memory
  =goal>
    isa   planning
    state load-last-waypoint
  =imaginal>
    isa   junction
```

```
  =retrieval>
    isa     movement
  ==>
  =goal>
    state save-waypoint
  =imaginal>
  =retrieval>
    direction nil
    counter   nil
  +retrieval>
    isa   waypoint
  - n1    nil
  - e1    nil
  - s1    nil
  - w1    nil
    n2    nil
    e2    nil
    s2    nil
    w2    nil)

(p discard-waypoint ;loop detected
  =goal>
    isa   planning
    state save-waypoint
  =imaginal>
    isa   junction
    north =n
    south =s
    east  =e
    west  =w
    performance =count ;actual counter value
  =retrieval>
    isa     waypoint
    distance  =old-count ;first junction's counter value at creation time
    n1  =n ;it's going in circle
    e1  =e
    s1  =s
    w1  =w
  ==>
  =goal>
    state retrieve-junction
  =retrieval>
    distance =count ;update the counter, don't have to count the steps made to walk in circle
  =imaginal>
  +retrieval> ;load the last infos about the user direction
    isa     movement
  - direction forwards
  - direction nil)

(p save-waypoint ;updates the last waypoint with the second junction and the distance
  =goal>
    isa   planning
    state save-waypoint
  =imaginal>
```

```
    isa   junction
    north =n2
    south =s2
    east  =e2
    west  =w2
    performance =count ;actual counter value
  =retrieval>
    isa     waypoint
    distance  =old-count ;first junction's counter value at creation time
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
  !eval! (not (and (eq =n1 =n2) (eq =e1 =e2) (eq =s1 =s2) (eq =w1 =w2))) ;the two junctions aren't the same
  ==>
  =goal>
    state duplicate-waypoint
  !bind! =dist (- =count =old-count) ;calculate the correct distance
  =retrieval> ;save the actual junction as second waypoint
    n2    =n2
    e2    =e2
    s2    =s2
    w2    =w2
    distance =dist
  !eval! (setq *wplist* (APPEND *WPLIST* '((=n2 =e2 =s2 =w2))))
  =imaginal> ;clear the chunk
    north nil
    south nil
    east  nil
    west  nil
    performance nil
  +imaginal> ;duplicate the chunk inverting the junctions
    isa   waypoint
    n1  =n2
    e1  =e2
    s1  =s2
    w1  =w2
    n2  =n1
    e2  =e1
    s2  =s1
    w2  =w1
    distance  =count ;need the value in the next production)

(p duplicate-waypoint ;updates the last waypoint with the second junction and the distance
  =goal>
    isa   planning
    state duplicate-waypoint
  =imaginal> ;duplicated chunk
    isa   waypoint
    distance  =count
  =retrieval> ;complete waypoint
    isa     waypoint
    n2  =n
    e2  =e
```

```
    s2  =s
    w2  =w
    distance  =dist
  ==>
  =goal>
    state new-waypoint
  =imaginal>
    distance  =dist ;set the correct distance
  +imaginal> ;create the next waypoint
    isa   waypoint
    n1    =n
    e1    =e
    s1    =s
    w1    =w
    distance  =count ;actual counter
    n2    nil
    e2    nil
    s2    nil
    w2    nil)

(p new-waypoint ;saves the new waypoint and prepares the buffers for the next steps
  =goal>
    isa   planning
    state new-waypoint
  =imaginal> ;new incomplete wayoint
    isa   waypoint
    n1  =n
    e1  =e
    s1  =s
    w1  =w
    distance  =count ;actual counter value
  ==>
  =goal>
    state retrieve-junction
  +imaginal>
    isa   junction
    north =n
    south =s
    east  =e
    west  =w
    performance =count ;actual counter value
  +retrieval> ;load the last infos about the user direction
    isa     movement
  - direction forwards
  - direction nil)

(p retrieve-junction
  =goal>
    isa   planning
    state retrieve-junction
  =imaginal>
    isa   junction
    north =n
    south =s
```

```
    east  =e
    west  =w
    performance =count ;actual counter value
  =retrieval>
    isa     movement
    direction =dir
  ==>
  +goal>
    isa   ext-junction
    turn  0
    north empty
    south empty
    east  empty
    west  empty
    front 9999
    right 9999
    left  9999
    performance 0
  =imaginal>
    turn  =dir ;saving the information for future use
  !bind! =side (turn-back =dir)
  ;calculate the acceptables ranges of distance
  !bind! =nl (- =n *similThresh*)
  !bind! =nh (+ =n *similThresh*)
  !bind! =el (- =e *similThresh*)
  !bind! =eh (+ =e *similThresh*)
  !bind! =sl (- =s *similThresh*)
  !bind! =sh (+ =s *similThresh*)
  !bind! =wl (- =w *similThresh*)
  !bind! =wh (+ =w *similThresh*)
  =retrieval>
    direction nil
    counter nil
  +retrieval>
    isa   junction
  - turn  =side ;match with all possible directions
  ;between val - thresh and val + thresh
  >=  north =nl
  <=  north =nh
  >=  south =sl
  <=  south =sh
  >=  east  =el
  <=  east  =eh
  >=  west  =wl
  <=  west  =wh
  - performance nil ;only a direction it took before
  +nxt-move>
    ISA emergency-stop)

;;;-------------------------------------------------------------------------
;;;--------------------there is at least one match-----------------------
;;;-------------------------------------------------------------------------
;;;this function calls itself until all the matches are retrieved, every time it saves which ways matched
     already and delete them from the next request
```

```
(p* junction-match ;found a match
  =goal>
    isa   ext-junction
    turn  =back
    north =v1
    east  =v2
    south =v3
    west  =v4
    performance =temp
  =imaginal>
    isa   junction
    turn  =dir
  =retrieval>
    isa   junction
    turn  =turn
    north =n
    south =s
    east  =e
    west  =w
    performance =perf
  - performance -1 ;not a loop
  ==>
  !bind! =back (turn-back =dir)
  !bind! =temp (get-movement =dir =turn)
  =goal>
    =turn =turn
    =back closed ;mark the way it's coming from as closed
    =temp =perf ;saves the performance of the retrieved direction, in relative coordinates
  =imaginal>
  ;deleting all the information, this way when the chunk will be removed from the buffer to make room for
     the next request it won't match with any junction
  ;this guarantees the reward mechanism to work correctly
  =retrieval>
    turn  nil
    north nil
    south nil
    east  nil
    west  nil
    performance nil
  +retrieval> ;tries to retrieve another chunk
    isa   junction
  - turn  =v1 ;if they contain empty, if won't have any effect. If that direction it has been explored, the
     value will contain the name of that direction and will prevent another retrieval
  - turn  =v2
  - turn  =v3
  - turn  =v4
  - turn  =back
  - turn  =turn
    north =n
    south =s
    east  =e
    west  =w
  - performance nil);only a direction it took before
```

```
(p* loop-match ;found a loop, mark it as a closed way
  =goal>
    isa   ext-junction
    turn  =back
    north =v1
    east  =v2
    south =v3
    west  =v4
    performance =temp
  =imaginal>
    isa   junction
    turn  =dir
  =retrieval>
    isa   junction
    turn  =turn
    north =n
    south =s
    east  =e
    west  =w
    performance -1
  ==>
  !bind! =back (turn-back =dir)
  !bind! =temp (get-movement =dir =turn)
  =goal>
    =turn =turn
    =back closed ;mark the way it already took as closed, it will be handled later
    =temp 10000
  =imaginal>
  ;deleting all the information, this way when the chunk will be removed from the buffer to make room for
     the next request it won't match with any junction
  ;this guarantees the reward mechanism to work correctly
  =retrieval>
    turn  nil
    north nil
    south nil
    east  nil
    west  nil
    performance nil
  +retrieval> ;tries to retrieve another chunk
    isa   junction
  - turn  =v1 ;if they contain empty, if won't have any effect. If that direction it has been explored, the
     value will contain the name of that direction and will prevent another retrieval
  - turn  =v2
  - turn  =v3
  - turn  =v4
  - turn  =back
  - turn  =turn
    north =n
    south =s
    east  =e
    west  =w
  - performance nil);only a direction it took before

;;;-----------------------------------------------------------------------
```

```
;;;--------------------one direction does not match----------------------
;;;---------------------------check for walls-----------------------------
;;;-----------------------------------------------------------------------
;;when all matches has been retrieved this function checks all the direction still marked with "empty" and,
     if a wall is present, marks them
(p north-wall
  =goal>
    isa   ext-junction
    north empty ;this way is free
  !bind! =min *minDist*
  =imaginal>
    isa   junction
    turn  =dir
  <=  north =min ;there is a wall on the front
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  =goal>
    north closed
  =imaginal>)

(p east-wall
  =goal>
    isa   ext-junction
    east  empty ;this way is free
  !bind! =min *minDist*
  =imaginal>
    isa   junction
    turn  =dir
  <=  east  =min ;there is a wall on the front
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  =goal>
    east  closed
  =imaginal>)

(p west-wall
  =goal>
    isa   ext-junction
    west  empty ;this way is free
  !bind! =min *minDist*
  =imaginal>
    isa   junction
    turn  =dir
  <=  west  =min ;there is a wall on the front
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
```

```
  ==>
  =goal>
    west  closed
  =imaginal>)

(p south-wall
  =goal>
    isa   ext-junction
    south empty ;this way is free
  !bind! =min *minDist*
  =imaginal>
    isa   junction
    turn  =dir
  <=  south =min ;there is a wall on the front
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  =goal>
    south closed
  =imaginal>)

;;;-----------------------------------------------------------------------
;;;--------------------one direction does not match----------------------
;;;---------------------choose a random direction-------------------------
;;;-----------------------------------------------------------------------

(p go-north-random
  =goal>
    isa   ext-junction
    north empty ;this way is free
  !bind! =min *minDist*
  =imaginal>
    isa   junction
    turn  =dir
  - turn  south ;can't go back
  > north =min ;there isn't a wall on the front
    north =n
    south =s
    east  =e
    west  =w
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  !bind! =side (get-movement =dir 'north) ;the way it need to turn
  +goal>
    isa   planning
    state =side
  =imaginal>)

(p go-east-random
```

```
  =goal>
    isa   ext-junction
    east  empty ;this way is free
  !bind! =min *minDist*
  =imaginal>
    isa   junction
    turn  =dir
  - turn  west ;can't go back
  > east  =min ;there isn't a wall on the front
    north =n
    south =s
    east  =e
    west  =w
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  !bind! =side (get-movement =dir 'east) ;the way it need to turn
  +goal>
    isa   planning
    state =side
  =imaginal>)

(p go-south-random
  =goal>
    isa   ext-junction
    south empty ;this way is free
  !bind! =min *minDist*
  =imaginal>
    isa   junction
    turn  =dir
  - turn  north ;can't go back
  > south =min ;there isn't a wall on the front
    north =n
    south =s
    east  =e
    west  =w
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  !bind! =side (get-movement =dir 'south) ;the way it need to turn
  +goal>
    isa   planning
    state =side
  =imaginal>)

(p go-west-random
  =goal>
    isa   ext-junction
    west  empty ;this way is free
  !bind! =min *minDist*
```

```
  =imaginal>
    isa   junction
    turn  =dir
  - turn  east ;can't go back
  > west  =min ;there isn't a wall on the front
    north =n
    south =s
    east  =e
    west  =w
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  !bind! =side (get-movement =dir 'west) ;the way it need to turn
  +goal>
    isa   planning
    state =side
  =imaginal>)

(p go-right
  =goal>
    isa   planning
    state right
  =imaginal>
    isa   junction
    turn  =dir
  ?retrieval>
    state free
  ==>
  =goal>
    state track-turn
  +retrieval>
    isa   movement
  - direction nil ;filling in the buffer for the next production
  !bind! =side (turn-right =dir)
  =imaginal>
    turn  =side  ;record the movement, going right
    performance -1
  !bind! =move *stepTurn*
  +nxt-move>
    ISA turn-right
    duration =move);turning right

(p go-left
  =goal>
    isa   planning
    state left
  =imaginal>
    isa   junction
    turn  =dir
  ?retrieval>
    state free
  ==>
```

```
  =goal>
    state track-turn
  +retrieval>
    isa   movement
  - direction nil ;filling in the buffer for the next production
  !bind! =side (turn-left =dir)
  =imaginal>
    turn  =side  ;record the movement, going left
    performance -1
  !bind! =move *stepTurn*
  +nxt-move>
    ISA turn-left
    duration =move);turning left

(p go-front
  =goal>
    isa   planning
    state front
  =imaginal>
    isa   junction
    turn  =dir
  ?retrieval>
    state free
  ==>
  =goal>
    state track-turn
  +retrieval>
    isa   movement
  - direction nil ;filling in the buffer for the next production
  =imaginal>
    turn  =dir  ;record the movement, going right
    performance -1)
;;;-------------------------------------------------------------------------
;;;--------------------there's at least on match-------------------------
;;;-------------------------find the best way-----------------------------
;;;-------------------------------------------------------------------------

(p best-front ;decides which way is more promising
  =goal>
    isa   ext-junction
  ; - north empty
  ; - east  empty
  ; - south empty
  ; - west  empty
    right =r
    left  =l
  ;;front is preferred over right and left
  <=  front =r
  <=  front =l
  < front 9999 ;can't ba a dead end or a loop
  =imaginal>
    isa   junction
    turn  =dir
    north =n
```

```
    south =s
    east  =e
    west  =w
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  +goal>
    isa   planning
    state go-best-dir
  =imaginal>
    ; turn  =dir
    north nil
    east  nil
    south nil
    west  nil
    performance nil
  +retrieval> ;retrieve the correct chunk
    isa   junction
    turn  =dir
    north =n
    south =s
    east  =e
    west  =w
  - performance nil ;only a direction it took before
  - performance 9999)

(p best-right ;decides which way is more promising
  =goal>
    isa   ext-junction
  ; - north empty
  ; - east  empty
  ; - south empty
  ; - west  empty
    front =f
    left  =l
  ;;right is preferred over left
  < right =f
  <=  right =l
  < right 9999 ;can't ba a dead end or a loop
  =imaginal>
    isa   junction
    turn  =dir
    north =n
    south =s
    east  =e
    west  =w
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  +goal>
```

```
    isa   planning
    state go-best-dir
  !bind! =side (turn-right =dir)
  =imaginal>
    turn  =side
    north nil
    east  nil
    south nil
    west  nil
    performance nil
  !bind! =move *stepTurn*
  +nxt-move>
    isa     turn-right
    duration  =move
  +retrieval> ;retrieve the correct chunk
    isa   junction
    turn  =side
    north =n
    south =s
    east  =e
    west  =w
  - performance nil ;only a direction it took before
  - performance 9999)

(p best-left ;decides which way is more promising
  =goal>
    isa   ext-junction
  ; - north empty
  ; - east  empty
  ; - south empty
  ; - west  empty
    right =r
    front =f
  < left  =r
  < left  =f
  < left  9999 ;can't ba a dead end or a loop
  =imaginal>
    isa   junction
    turn  =dir
    north =n
    south =s
    east  =e
    west  =w
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  +goal>
    isa   planning
    state go-best-dir
  !bind! =side (turn-left =dir)
  =imaginal>
    turn  =side
```

```
    north nil
    east  nil
    south nil
    west  nil
    performance nil
  !bind! =move *stepTurn*
  +nxt-move>
    isa     turn-left
    duration  =move
  +retrieval> ;retrieve the correct chunk
    isa   junction
    turn  =side
    north =n
    south =s
    east  =e
    west  =w
  - performance nil ;only a direction it took before
  - performance 9999)

(p go-best-dir
  =goal>
    isa   planning
    state go-best-dir
  =retrieval> ;this chunk will now match with another in dm, so that in the end this chunk will be rewarded
    isa   junction
  =imaginal>
    isa   junction
  ==>
  =goal>
    state track-turn
  -retrieval>
  =imaginal>
  +retrieval>
    isa     movement ;needs to know the last counter
  - direction nil)

;;;-----------------------------------------------------------------------
;;;----------------------------nowhere to go------------------------------
;;;-------------------------a loop is detected----------------------------
;;;-----------------------------------------------------------------------

(p loop-front
  =goal>
    isa   ext-junction
  ;all other directions have been already explored, the only chance is to follow the loop
  - north empty
  - east  empty
  - south empty
  - west  empty
  >=  right 9999 ;even if there is a wall, instead of a dead-end, the value will be 9999
  >=  left  9999
    front 10000
  =imaginal>
    isa   junction
```

```
    turn  =dir
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  +goal>
    isa   planning
    state track-loop-front
  =imaginal>
  +retrieval>
    isa junction
    performance -1 ;the last way it took)

(p loop-right
  =goal>
    isa   ext-junction
  ;all other directions have been already explored, the only chance is to follow the loop
  - north empty
  - east  empty
  - south empty
  - west  empty
  >=  front 9999 ;even if there is a wall, instead of a dead-end, the value will be 9999
  >=  left  9999
    right 10000
  =imaginal>
    isa   junction
    turn  =dir
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  +goal>
    isa   planning
    state track-loop-right
  =imaginal>
  +retrieval>
    isa junction
    performance -1 ;the last way it took)

(p loop-left
  =goal>
    isa   ext-junction
  ;all other directions have been already explored, the only chance is to follow the loop
  - north empty
  - east  empty
  - south empty
  - west  empty
  >=  right 9999 ;even if there is a wall, instead of a dead-end, the value will be 9999
  >=  front 9999
    left  10000
  =imaginal>
    isa   junction
```

```
    turn  =dir
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  +goal>
    isa   planning
    state track-loop-left
  =imaginal>
  +retrieval>
    isa junction
    performance -1 ;the last way it took)

(p track-loop-front ;if in the already explored direction is in front, take that way
  =goal>
    isa   planning
    state track-loop-front
  =imaginal>
    isa junction
    turn  =dir ;actual direction
  =retrieval>
    isa junction
    performance -1 ;the last way it took
  ==>
  =goal>
    state track-turn
  =retrieval>
    performance 9999 ;give a terrible performance value, next time it won't turn this way
  =imaginal>
    turn  =dir
    performance -1 ;create and save (later) in dm a chunk to mark this junction as the most recent
  +retrieval>
    isa movement
  - direction nil)

(p track-loop-right ;if in the already explored direction is on the right, take that way
  =goal>
    isa   planning
    state track-loop-right
  =imaginal>
    isa junction
    turn  =dir ;actual direction
  =retrieval>
    isa junction
    performance -1 ;the last way it took
  ==>
  =goal>
    state track-turn
  !bind! =turn (turn-right =dir)
  =imaginal>
    turn  =turn
    performance -1 ;create and save (later) in dm a chunk to mark this junction as the most recent
  =retrieval>
```

```
    performance 9999 ;set that way as a dead end
  !bind! =move *stepTurn*
  +nxt-move>
    ISA turn-right
    duration =move ;going right
  +retrieval>
    isa movement
  - direction nil)

(p track-loop-left ;if in the already explored direction is on the left, take that way
  =goal>
    isa   planning
    state track-loop-left
  =imaginal>
    isa junction
    turn  =dir ;actual direction
  =retrieval>
    isa junction
    performance -1 ;the last way it took
  ==>
  =goal>
    state track-turn
  !bind! =turn (turn-left =dir)
  =imaginal>
    turn  =turn
    performance -1 ;create and save (later) in dm a chunk to mark this junction as the most recent
  =retrieval>
    performance 9999 ;set that way as a dead end
  !bind! =move *stepTurn*
  +nxt-move>
    ISA turn-left
    duration =move ;going left
  +retrieval>
    isa movement
  - direction nil)

;;;-----------------------------------------------------------------------
;;;----------------------------nowhere to go------------------------------
;;;----------------------------it's a dead end----------------------------
;;;-----------------------------------------------------------------------

(p remove-junction ;all the possible ways are dead ends, updates the previous junction
  =goal>
    isa   ext-junction
  ;all other directions have been already explored, the only chance is to follow the loop
  - north empty
  - east  empty
  - south empty
  - west  empty
    right 9999 ;even if there is a wall, instead of a dead-end, the value will be 9999
    left  9999
    front 9999
  =imaginal>
    isa   junction
```

```
    turn  =dir
  ?retrieval>
    state free
  ?retrieval>
    buffer  empty ;no match
  ==>
  +goal>
    isa   planning
    state dead-end
  +retrieval> ;the last junction before the dead-end
    isa   junction
  - performance nil ;only a valid junction
  !bind! =move (* 2 *stepTurn*)
  +nxt-move>
    ISA turn-left
    duration =move ;going back
  !bind! =side (turn-back =dir)
  =imaginal>
    turn  =side
    north nil
    east  nil
    south nil
    west  nil
    performance nil)

; (p hit ;hits the wall, stop
  ; =goal>
    ; ISA planning
  ; - state end
  ; ?nxt-touched>
    ; touched true
  ; ==>
  ; =goal>
    ; state end
  ; +nxt-move>
    ; ISA emergency-stop
  ; +nxt-visual>
    ; isa light
    ; turn off
  ; +temporal>
    ; isa clear ;reset the timer
; )

;#############################################################################################
;-------------- Ab Hier Schaetzungen ---------------------------------------------------------
;#############################################################################################

; Versuche einen passenden Abschnitt im decl. memory zu finden.
(P Ausgabe_Direkt1
  =goal>
    isa Ausgabe
    Unterstatus Suche_Direkt
  - n1  -1
  - e1  -1
```

```
  - s1  -1
  - w1  -1
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  ?retrieval>
    state free
    buffer  empty
  ?imaginal>
    state free
  ?vocal>
    state free
  ==>
  =goal>
    Unterstatus Suche_Direkt2
  !eval! (delete-duplicates *wplist* :test #'equal) ;removes the duplicated waypoints
  +retrieval>
    isa waypoint
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2)

; Weg gefunden lade die Entfernung aus dem decl. memory.
(P Ausgabe_Direkt2
  =goal>
    isa Ausgabe
    Unterstatus Suche_Direkt2
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  ?retrieval>
    state free
    - buffer  empty
  ?imaginal>
    state free
  ?vocal>
    state free
  =retrieval>
    isa waypoint
```

```
    distance  =dist
  ==>
  =goal>
    Unterstatus Suche_Direkt3
  =retrieval>
    n1  nil
    e1  nil
    s1  nil
    w1  nil
    n2  nil
    e2  nil
    s2  nil
    w2  nil
    distance nil
  +retrieval>
    isa zahl
    wert  =dist)

; Erfolgreich einen Weg gefunden. Gib diesen aus.
(P Ausgabe_Direkt3
  =goal>
    isa Ausgabe
    Unterstatus Suche_Direkt3
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  ?retrieval>
    state free
    - buffer  empty
  ?imaginal>
    state free
  ?vocal>
    state free
  =retrieval>
    isa Zahl
    Wert  =ent
    Name  =name
  ==>
  ; Hole neue Wegpunkte.
  !eval! (next-waypoint)
  !bind! =n1x (get-n1)
  !bind! =e1x (get-e1)
  !bind! =s1x (get-s1)
  !bind! =w1x (get-w1)
  !bind! =n2x (get-n2)
  !bind! =e2x (get-e2)
  !bind! =s2x (get-s2)
  !bind! =w2x (get-w2)
  =goal> ; Setze Goal auf neue Wegpunkte
```

```
    Unterstatus Suche_Direkt
    n1  =n1x
    e1  =e1x
    s1  =s1x
    w1  =w1x
    n2  =n2x
    e2  =e2x
    s2  =s2x
    w2  =w2x
  -retrieval>
  +imaginal>
    isa Antwort
    Wert  =retrieval
    Entfernung  =name
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  -imaginal>
  +vocal>
    ISA speak
    string  =name
    !output! (Wegpunkt =n1 =e1 =s1 =w1 zu =n2 =e2 =s2 =w2 habe ich =ent geschaetzt.
        Ich werde nun die distance von =n1x =e1x =s1x =w1x zu =n2x =e2x =s2x =w2x schaetzen.))

; Ist die direkte Suche nicht erfolgreich suche indirekt ueber die Karte.
(P Ausgabe_Indirekt
  =goal>
    isa Ausgabe
    Unterstatus Suche_Direkt2
  - n1  -1
  - e1  -1
  - s1  -1
  - w1  -1
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  ?retrieval>
    state error
  ?vocal>
    state free
  ==>
  =goal>
    Unterstatus Suche_Indirekt
  +retrieval> ; Hole Abschnitt der mit wp1 beginnt
```

```
    isa waypoint
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
  - n2  nil
  - e2  nil
  - s2  nil
  - w2  nil)

; Lade erste Zahl und speichere die Wegpunkte des ersten Abschnitts im imaginal.
(P Ausgabe_Kombiniere_Abschnitt
  =goal>
    isa Ausgabe
    Unterstatus Suche_Indirekt
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
  ?retrieval>
    state free
    - buffer  empty
  ?imaginal>
    state free
  =retrieval>
    isa waypoint
    distance  =ent
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  ?vocal>
    state free
  ==>
  =goal>
    Unterstatus Hole_Zahl1
  =retrieval>
    n1  nil
    e1  nil
    s1  nil
    w1  nil
    n2  nil
    e2  nil
    s2  nil
    w2  nil
    distance nil
  +retrieval>
    isa zahl
    wert  =ent  ; Lade den Zahlenchunk aus dem deklarativen Gedaechtnis
  +imaginal>
```

```
    isa waypoint
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2)

; Lade zweiten Abschnitt und speichere die entfernung aus dem retrieval in den imaginal
(P Ausgabe_Lade_Abschnitt2
  =goal>
    isa Ausgabe
    Unterstatus Hole_Zahl1
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
  ?retrieval>
    state free
    - buffer  empty
  ?imaginal>
    state free
  ?vocal>
    state free
  =retrieval> ; verfaelschte Zahl
    isa Zahl
    Wert  =ent
  =imaginal>  ; Wegpunkt 1 und mittlerer wp.
    isa waypoint
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  ==>
  =goal>
    Unterstatus check_wp
  +retrieval> ; hole 2. Abschnitt
    isa waypoint
    n1  =n2
    e1  =e2
    s1  =s2
    w1  =w2
  - n2  nil
  - e2  nil
  - s2  nil
  - w2  nil
  =imaginal>
    distance  =ent)

(P valid_wp
  =goal>
    isa Ausgabe
```

```
    Unterstatus check_wp
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
  ?retrieval>
    state free
    - buffer  empty
  ?imaginal>
    state free
    - buffer  empty
  ?vocal>
    state free
  =retrieval> ; Im retrieval der zweite Abschnitt
    isa waypoint
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  =imaginal>  ; Im Imaginal der erste Abschnitt mit schaetzfehler!
    isa waypoint  ; Mittleren Wegpunkt
  !eval! (not (and (eq =n1 =n2) (eq =e1 =e2) (eq =s1 =s2) (eq =w1 =w2))) ;the two junctions aren't the same
  ==>
  =goal>
    Unterstatus Hole_Abschnitt2
  =retrieval>
  =imaginal>)

(P invalid_wp
  =goal>
    isa Ausgabe
    Unterstatus check_wp
    n1  =n2
    e1  =e2
    s1  =s2
    w1  =w2
  ?retrieval>
    state free
    - buffer  empty
  ?imaginal>
    state free
    - buffer  empty
  ?vocal>
    state free
  =retrieval> ; Im retrieval der zweite Abschnitt
    isa waypoint
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
```

```
  =imaginal>  ; Im Imaginal der erste Abschnitt mit schaetzfehler!
    isa waypoint  ; Mittleren Wegpunkt
    distance  =ent
  ==>
  =goal>
    Unterstatus Hole_Zahl1 ;try again
  !eval! (reduce_activ =n1 =e1 =s1 =w1 =n2 =e2 =s2 =w2) ;reduce the activation of this chunk, so next time
     it won't be retrieved
  =retrieval>
    n1  nil
    e1  nil
    s1  nil
    w1  nil
    n2  nil
    e2  nil
    s2  nil
    w2  nil
    distance nil
  +retrieval>
    isa zahl
    wert  =ent
  =imaginal>)

; Hole die Entfernung des zweiten Abschnits (Schaetzfehler durch das decl. memory).
(P Ausgabe_Lade_Entfernung_Abschnitt2
  =goal>
    isa Ausgabe
    Unterstatus Hole_Abschnitt2
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
  ?retrieval>
    state free
    - buffer  empty
  ?imaginal>
    state free
    - buffer  empty
  ?vocal>
    state free
  =retrieval> ; Im retrieval der zweite Abschnitt
    isa waypoint
    distance  =ent_neuer_abschnitt
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  =imaginal>  ; Im Imaginal der erste Abschnitt mit schaetzfehler!
    isa waypoint
    distance  =ent_bisher ; Entfernung mit schaetzfehler
    n2  =n_mitte
    e2  =e_mitte
    s2  =s_mitte
    w2  =w_mitte  ; Mittleren Wegpunkt
```

```
  ==>
  =goal>
    Unterstatus Addiere_Zahlen
  =retrieval>
    n1  nil
    e1  nil
    s1  nil
    w1  nil
    n2  nil
    e2  nil
    s2  nil
    w2  nil
    distance nil
  +retrieval>
    isa zahl
    wert  =ent_neuer_abschnitt  ; Hole die verfaelschte zahl aus dem deklarativen Memory
  =imaginal>
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2)

; Addiere die beiden Entfernungen und kehre zurueck zur direkten suche, da jetzt im dekl. memory neue chunks
      sind!
(P Ausgabe_Addiere_Abschnitt2
  =goal>
    isa Ausgabe
    Unterstatus Addiere_Zahlen
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
  ?retrieval>
    state free
    - buffer  empty
  ?imaginal>
    state free
    - buffer  empty
  ?vocal>
    state free
  =retrieval>
    isa Zahl
    Wert  =ent_abschnitt2
  =imaginal>
    isa waypoint
    distance  =ent_abschnitt1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2

  !bind!  =ent  (+ =ent_abschnitt1 =ent_abschnitt2) ; Addiere die beiden Zahlen zusammen.
  ==>
  =goal>  ; Gehe zurueck zur direkten Suche.
```

```
    Unterstatus Addiere_Zahl_Hole_Zahl
  +retrieval>
    isa Zahl
    Wert  =ent
  =imaginal>
  !output!  (Entfernung des neuen Chunks von =n1 =e1 =s1 =w1 zu =n2 =e2 =s2 =w2 ist =ent))

; Wenn die Zahlen zu gro
(P Ausgabe_Addiere_Abschnitt_Zu_Gross
  =goal>
    isa Ausgabe
    Unterstatus Addiere_Zahl_Hole_Zahl
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
  ?imaginal>
    state free
    - buffer  empty
  ?retrieval>
    state error
  =imaginal>
    isa waypoint
    distance  =ent_abschnitt1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  ?vocal>
    state free
  ==>
  =goal>  ; Gehe zurueck zur direkten Suche.
    Unterstatus Suche_Indirekt
  +vocal>
    ISA speak
    string  "Och␣ne!␣Ich␣weigere␣mich␣mit␣solchen␣Zahlen␣zu␣Arbeiten!"
  -retrieval>
  -imaginal>)

; Speichere den neuen Abschnitt, und suche erneut einen direkten Weg!
(P Addiere_Zahl_Hole_Zahl
  =goal>
    isa Ausgabe
    Unterstatus Addiere_Zahl_Hole_Zahl
  ?retrieval>
    state free
    - buffer  empty
  =retrieval>
    isa Zahl
    wert  =ent
  ?imaginal>
    state free
    - buffer  empty
  =imaginal>
```

```
    isa waypoint
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  ==>
  =goal>
    Unterstatus Suche_Direkt
  =imaginal>
    distance  =ent
  -retrieval>
  -imaginal>)

; Wenn sich das model nicht mehr erinnert, oder wenn das model den Weg nicht gelaufen ist, hole naechste zu
     schaetzende distance
(P Nicht_Gefunden
  =goal>
    isa Ausgabe
    Unterstatus Suche_Indirekt
    n1  =n1
    e1  =e1
    s1  =s1
    w1  =w1
    n2  =n2
    e2  =e2
    s2  =s2
    w2  =w2
  ?vocal>
    state free
  ?retrieval>
    state error
  ==>
  ; Hole neue Wegpunkte.
  !eval! (next-waypoint)
  !bind! =n1x (get-n1)
  !bind! =e1x (get-e1)
  !bind! =s1x (get-s1)
  !bind! =w1x (get-w1)
  !bind! =n2x (get-n2)
  !bind! =e2x (get-e2)
  !bind! =s2x (get-s2)
  !bind! =w2x (get-w2)
  =goal> ; Setze Goal auf neue Wegpunkte
    Unterstatus Suche_Direkt
    n1  =n1x
    e1  =e1x
    s1  =s1x
    w1  =w1x
    n2  =n2x
    e2  =e2x
    s2  =s2x
    w2  =w2x
  -retrieval>
  +vocal>
```

```
    ISA speak
    string  "Haeh!␣Wei␣ich␣nicht!"
  !output! (Wegpunkt =n1 =e1 =s1 =w1 zu =n2 =e2 =s2 =w2 habe ich nicht gefunden.
      Ich werde nun die Entfernung von =n1x =e1x =s1x =w1x zu =n2x =e2x =s2x =w2x schaetzen.) )

; Wenn der Versuch zuende hoere auf!
(P Ende
  =goal>
    isa Ausgabe
    Unterstatus Suche_Direkt
    n1  -1
    e1  -1
    s1  -1
    w1  -1
  ?vocal>
    state free
  ==>
  +goal>
    isa planning
    state move
  +retrieval> ;filling the buffer for the next production
    isa   movement
  - direction nil
  +vocal>
    ISA speak
    string  "Das␣war␣Anstrengend!␣Ich␣nehm␣dann␣die␣Versuchspersonenstunden!"
  !Output!  ( The End )
  !eval!  (setq *wplist* '((-2 -2 -2 -2)))
  !eval! (setq *wp1index* 0)
  !eval! (setq *wp2index* 1))


(set-similarities
  (Eins Zwei      0)
  (Zwei Drei      0)
  (Drei Vier      0)
  (Vier Fuenf     0)
  (Fuenf Sechs    0)
  (Sechs Sieben     0)
  (Sieben Acht    0)
  (Acht Neun      0)
  (Neun Zehn      0)
  (Zehn Elf       0)
  (Elf Zwoelf     0)
  (Zwoelf Dreizehn  0)
  (Dreizehn Vierzehn  0))

(goal-focus goal)
(spp recognise-goal :u 20)
(spp dead-end :u 0)
);define-model
```

References

[ABB+04] J.R. Anderson, D. Bothell, M.D. Byrne, S. Douglass, C. Lebiere, and Y. Qin. An integrated theory of the mind. *Psychological Review*, 111(4):1036–1060, 2004.

[Act] Act-R. http://act-r.psy.cmu.edu. 15

[And03] M.L. Anderson. Embodied cognition: A field guide. *Artificial intelligence*, 149(1):91–130, 2003. 17

[BBM+99] R.A. Brooks, C. Breazeal, M. Marjanović, B. Scassellati, and M.M. Williamson. The cog project: Building a humanoid robot. In *Computation for metaphors, analogy, and agents*, pages 52–87. Springer-Verlag, 1999. 17

[BHW09] S.J. Buechner, C. Hölscher, and J.M. Wiener. Search strategies and their success in a virtual maze. In *Proceedings of the 31st Annual Conference of the Cognitive Science Society*, pages 1066–1071, 2009. 3, 17, 18, 22, 45, 73, 74, 76

[BK91] J. Borenstein and Y. Koren. The vector field histogram and fast obstacle-avoidance for mobile robots. *IEEE Journal of Robotics and Automation*, 7:278–288, 1991. 18

[Bot04] D. Bothell. ACT-R 6.0 reference manual. Technical report, 2004. 77, 78

[BSB+08] Cynthia Breazeal, Michael Siegel, Matt Berlin, Jesse Gray, Roderic A. Grupen, Patrick Deegan, Jeff Weber, Kailas Narendran, and John McBean. Mobile, dexterous, social robots for mobile manipulation and human-robot interaction. In *SIGGRAPH New Tech Demos*, page 27. ACM, 2008. 19

[CC02] N.L. Cassimatis and N.L. Cassimatis. Polyscheme: A cognitive architecture for integrating multiple representation and inference schemes. 2002. 6

[CG99] Andy Clark and Rick Grush. Towards a Cognitive Robotics. *Adaptive Behavior*, 7(1):5–16, 1999. 17

[DK+11] F. Dietrich, J. Kan, et al. ACT-Rientierung. http://sourceforge.net/projects/act-rientierung/, 2011. 59

[Hir07] T. Hiraishi. Nxt controller in lisp. http://super.para.media.kyoto-u.ac.jp/~tasuku/index-e.html, 2007. 10

[HT10] A.M. Harrison and J.G. Trafton. Cognition for action: an architectural account for "grounded interaction". In *Proceedings of the 32nd Annual Conference of the Cognitive Science Society*, pages 246–251, 2010. 19

[KN00] Jonathan B. Knudsen and Markus L. Noga. *Das inoffizielle Handbuch für LEGO® MINDSTORMS™ Roboter*. O'Reilly, Beijing, Cambridge, Farnham, Köln, Paris, Sebastopol, Taipei, Tokyo, 1. Aufl. edition, 2000. engl. Original: The Unofficial Guide to LEGO® MINDSTORMS™ Robots, 1999 – deutsche Übersetzung: Matthias Kalle Dalheimer. 15

[Leg] Lego Mindstorms. http://mindstorms.lego.com. 15

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Prentice Hall, 2nd international edition edition, 2003. 11, 15, 18, 31, 32, 46, 57, 82

[ST+04] Donald Sofge, J. Gregory Trafton, et al. Human-robot collaboration and cognition with an autonomous mobile robot. *Intelligent autonomous systems 8*, page 80, 2004. 6, 18

[Tol48] E.C. Tolman. Cognitive maps in rats and men. *Psychological review*, 55(4):189, 1948. 15

[Tra09] J.G. Trafton. An embodied model of infant gaze-following. Technical report, DTIC Document, 2009. 19