The University of Birmingham
School of Computer Science
MSc in Advanced Computer Science

Mini-Project I

The ways to improve intelligence of interacting agents

Marek Kopicki

Supervisor: Prof. Aaron Sloman

January 12, 2004

Abstract

The path planning is not a trivial problem of artificial intelligence. An agent has to find a path from one state (or position) to another whilst avoiding contact with obstacles. The configuration space used for representation of all agent states is usually continuous, which makes the problem even more complex. Skeletonisation is one of approaches, which "discretises" continuous space and reduces it to a graph search problem.

The "sheepdog" demo is a Pop-11 written computer simulation of an artificial world consisting of a dog, sheep, trees (obstacles) and a pen. The task of the dog is to drive all the sheep to the pen avoiding collisions with trees and other agents. Probabilistic roadmap and A* graph search algorithm play a major role in the current refined version of the simulation, changing an original "stimulus-response" paradigm. I will present advantages of using agent planning, and I will attempt to confront this traditional AI conceptual approach to a concept-free, perception-action architecture proposed by Rodney Brooks.

Even though the program still needs lots of improvements the overall result of the simulation is promising - the dog is able to complete the task, avoiding dynamically obstacles and changing the plan if necessary. The future version of the program might involve smarter skeletonisation procedure, more extensive use of a sim_agent toolkit, and possibly one of approximate search algorithms to tackle more complex environment.

Keywords

Multi-agent simulation, path/motion planning, skeletonisation, probabilistic roadmap, graph search algorithms, algorithm A*, collision avoidance, Pop-11, sim_agent toolkit.

Acknowledgements

I would like to take this opportunity to thank my project supervisor Prof. Aaron Sloman who painstakingly guided me during the mini-project. Without his insightful suggestions and comments as well as enlightening discussions with him, I would not have been able to complete this mini-project.

Marek Kopicki

Contents

Abstract ii

Keywords ii

Acknowledgements iii

Table of contents v

List of Figures vi

List of Tables vii

1 Preface 1

2 Why planning? 3

3 Path planning 6
- 3.1 Path planning problem . . . 6
- 3.2 Configuration space . . . 6
- 3.3 Cell decomposition . . . 6
- 3.4 Skeletonisation . . . 7
- 3.5 Graph searching . . . 8
- 3.6 Emergent problems . . . 9
- 3.7 Path optimising . . . 9
- 3.8 Local planning . . . 11
- 3.9 Obstacle avoidance . . . 12

4 The program 14
- 4.1 The SIM_AGENT toolkit . . . 14
- 4.2 Behavioural perspective . . . 15
  - 4.2.1 The dog rulesystem . . . 15
  - 4.2.2 The sheep rulesystem . . . 17
- 4.3 Functional perspective . . . 18

5 Conclusions 20

A Running the "sheepdog" simulation 22

B Source code **23**
B-1 Probabilistic roadmap routines . . . 23
B-2 Graph search algorithm . . . 24
B-3 Path area class and methods . . . 27
B-4 Path optimisation routines . . . 31
B-5 The dog rulesystem . . . 32

C Mini-project declaration **42**

D Statement of information search strategy **44**
1 Parameters of literature search . . . 44
1.1 Forms of literature . . . 44
1.2 Geographical and language coverage . . . 44
1.3 Age-Range of literature . . . 44
2 Appropriate search tools . . . 44
2.1 Engineering Index . . . 44
2.2 Science Citation Index (SCI) . . . 45
2.3 Dissertations Abstracts International (DAI) and Index to Theses . . . 45
3 Search statements . . . 45
4 Brief evaluation of the search . . . 45

Bibliography **46**

List of Figures

1.1 The “sheepdog” world 2

2.1 Traditional vertical decomposition of an intelligent system 3
2.2 Subsumption architecture 4

3.1 Random graph nodes generated by the probabilistic roadmap algorithm 7
3.2 Path topological problem 10
3.3 Path optimisation 11
3.4 Local planning 12
3.5 Path area 13

4.1 The structure of the “sheepdog” demo classes 19

List of Tables

4.1 The structure of the dog rulesystem. 16
4.2 The structure of the sheep rulesystem. 17

Chapter 1

Preface

The "sheepdog" demo is an example of computer simulation of "intelligent" interacting agents developed at the University of Birmingham. The origins of the program [13] date back to 1996 Peter Waudby's summer project, and later to 1999 Tom Carter's summer project, consecutively maintained and improved by Prof. Aaron Sloman.

The underlying purposes and appearance of the "sheepdog" has not been changed significantly since then - it is a simulation written in the Pop-11 AI language of an artificial world consisting of a dog, sheep, trees and a pen. Out of all interacting or "living" agents, the dog is supposed to display the most intelligent behaviour driving all the sheep (staying out of the pen) to the pen avoiding collisions with trees and other agents.

The dog (as well as sheep) in the original version of "sheepdog" is a kind of stimulus-response agent built on the basis of the sim_agent toolkit. Even though, he was able to complete the task successfully, a design of agents used there unveiled its limitations, which could be observed especially in more complex environment. One might argue whether more sophisticated design, within "stimulus-response" architecture, is sufficient for agents to complete their tasks. Probably so in this case - one can imagine a rather complicated procedure of following encountered walls and searching for any passages between them. Perhaps a real "intelligent" dog might be using similar tactics to the above one.

We decided, however, to endow the dog with supernatural capabilities of having a map of his world and cleverness of moving according to previously prepared plan. The new approach allows the agent to complete the task effectively and reliably.

- A plan is created for each non local move, i.e. to a destination (goal) not directly visible from a starting position (root). Created plan can be changed dynamically when already marked out path is impassable.
- A new plan is then rearranged locally only , saving in this way not only valuable resources and CPU time but also avoiding some undesirable effects, which I will point out later in the report.
- An agent following planned path optimises it "on the fly", making a motion more smooth and natural.
- Algorithms used are fairly universal and do not need too many assumptions about the objects and agents in "the sheepdog world" as it was the case before. The algorithms should work in various environments as well, with diverse shapes of agents and obstacles, and without altering most of the program code.

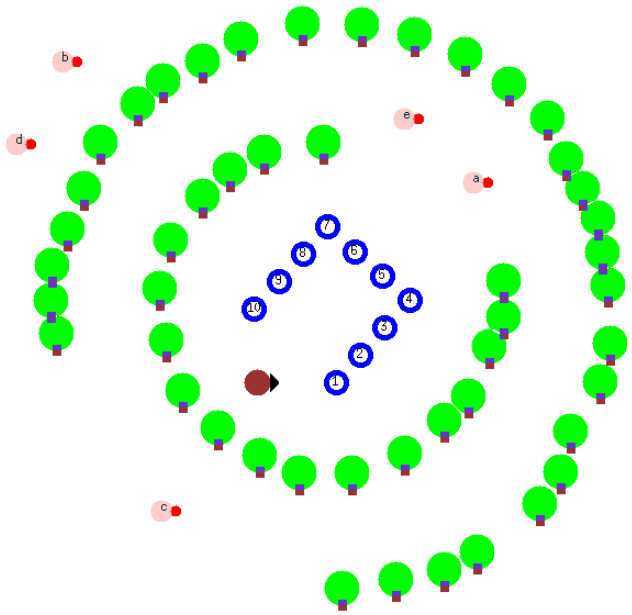

Figure 1.1: The present "sheepdog" world consists of a dog, five sheep, trees and a pen in the centre.

- Most complex and CPU demanding calculations are placed in various class members, clearing up **rulesets**[1] and **rulefamilies**, what greatly simplifies any future extensions of the system in its behavioural part in particular.

The present version of "sheepdog" demo contains many more trees as well thanks to Prof. Aaron Sloman such things as mouse-event handling and its appearance have been improved (Figure 1.1).

[1] See Chapter 4.1

Chapter 2

Why planning?

The "sheepdog" is not thought to be any guidance for replicating human level of intelligence, however, the transformation, which was done from a perception-reaction to a concept-action approach, invoked an interesting dispute between scientists a few years ago. Rodney Brooks in his paper [1] proposed a new type of architecture for an intelligent, operating in the real world robot.

His **subsumption** architecture is an example of behaviour-based approach, which was supposed to solve most of the classical AI problems, particularly related to representation of the world. The whole new concept arose in opposition to conceptual or representation-based approaches (the distinction between these two notions is not so relevant at this point) since, as they are biased by human-made abstraction causing excessive simplifications of modelled world. "The abstraction reduces the input data so that the program experiences the same perceptual world as humans" [2]. As a result a robot operating in a block or "toy world" after moving it into the real one, faces discrepancies which cannot be removed afterwards by simple means and the model used usually needs fundamental redesign.

The classical AI approach to an intelligent system prescribes functional (vertical) decomposition of the central system into subfields such as "knowledge representation", "learning" or "planning" (Figure 2.1). Brooks proposes horizontal decomposition by activity (Figure 2.2), slicing up the system into behavioural layers , responsible for avoiding obstacles, wandering, exploring etc.

Such systems or "the Creatures" are "collections of competing behaviours" and has no explicit representation even on the local level [2]. If the system is complex enough, "out of the local chaos of their interactions there emerges, in the eye of an observer, a coherent pattern of behaviour" Brooks

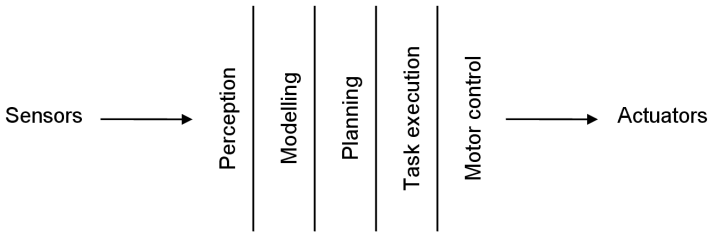

Figure 2.1: The traditional vertical decomposition of an intelligent system [1], which roughly illustrates the new design of the dog system.

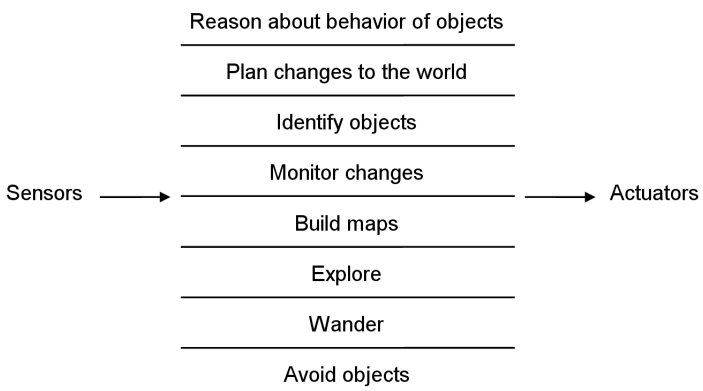

Figure 2.2: The subsumption architecture proposed by Brooks [1] decomposes a system horizontally into task achieving behaviours.

claims [2], and furthermore the complexity of the behaviour of the Creature reflects complexity of the environment rather then the Creature itself. I do not know if the complexity of behaviour, especially in the environmental context, is to be the most essential feature of human-like robot but certainly not our dog, which should display its effectiveness on completing the task rather then wander aimlessly for example.

No doubt Brooks' observations of the role of human abstraction during design of the systems for intelligent robots are worth closer investigation. Is really most of human activity concept-free or, what is more important to us, can we built an "intelligent" and useful robot mostly basing on representation-free approach?

It seems that the ability to extract any characteristic features of objects and then group or classify these objects against these features is essential for any intelligent beings (at least human-say). David Kirsh wrote an interesting article [5] responding to Brooks. He claims that "intelligent systems often frame or pose problems to themselves in a certain way, that they search through some explicit hypothesis space at times" and there is a part of intelligence which knows "how to find the structures in memory that might be helpful in a task and putting those structures to use".

Kirsh distinguishes three levels of concept organisation involved in the action management:

1. Perceptual level. An agent possesses concepts invariant across perceptual objects (e.g. the concept of a glass). It is hard to imagine the system operating with external objects without its explicit representation, a glass for example looks different from different angles.

2. Conceptual level. An agent infers about its concepts, is able to have beliefs and thoughts about them. How an intelligent robot can use a drill without explicit knowledge about functioning of drill?

3. Linguistic level. An agent knows the meaning of terms, relations between terms and concepts, can constitute logic. This level is inherent to human.

Our dog operates on the second level only; so most issues connected with perception are simply left aside. Yet, if a "brain" of an agent is designed in behavioural, representation-free fashion, it must immediately deal with complexity of the environment - the new version of "sheepdog" contains e.g. mazes of trees. Because a world model is not kept in the dog's memory, all the

information required to take a proper action must be retrieved from the environment. Moreover, "a perception-driven creature can only anticipate the future if there is evidence of the future in its present" [5]. Of course, even though it would be the case for a "pure" representation-free design, it inevitably implies complexity of such behaviour-based systems, even for the dog in the "sheepdog" simulation.

To make matter worse, let us suppose that there is no evidence in the environment that a pathway chosen by an agent is a blind alley. The agent must know how to backtrack on its way, what would be extremely difficult without any kind of the snapshots record of the followed blind alley. Those snapshots should be stored in a perspective-neutral way to avoid further problems with its recognition [5]. But, are we still thinking about a representation-free system?

The solution is somewhat simple. Before undertaking any action, an agent can prepare a plan - a kind of a model of the environment.

- Detailed information about the environment is retrieved only during preparation of a plan.
- Basically, having the plan an agent has to be checking only its feasibility (is it a blind alley?) at each step, reducing the amount of retrieved information to the minimum.
- The agent does not have to anticipate the future in such a way as before, it just has to comply with the plan.

To prepare a plan, however, there must be clearly defined goal [1], what is in contrast with behaviour-based systems where "behavioural responses are explicitly designed into the system but there are not any explicitly represented goals" [3]. Having already a concept of a plan, one can escape most of the mentioned troubles, although, as usual, there is no rose without a thorn.

[1] Unfortunately, the authors I cite in the report, use "goal" in at least two different meanings. Nevertheless this interchange should not be confusing since because their context is clear enough.

Chapter 3

Path planning

3.1 Path planning problem

As it was presented in the previous chapter, in the first step of plan preparation we have to specify the goal. The only agent making use of planning in the "sheepdog" simulation is the dog, which has to drive all the sheep to the pen. Because this goal is too general and vague, should be further subdivided into smaller parts, e.g. for each sheep separately. Thus a plan for a single sheep might have the following two stages:

1. To find a sheep (any sheep staying out of the pen).
2. To steer the chosen sheep to the pen.

Repeating above plan consecutively with each sheep allows the dog to complete the main goal. As a result we can reduce our plan to a problem of finding a passage or just path from one location to the other, called **path planning problem**.

3.2 Configuration space

We can define an agent's move as the point-to-point motion from any start position to a specified target in the **configuration space**. In general the configuration space representing an agent states like location, orientation, joint angles, etc. [7], can have many dimensions, which paradoxically simplifies move analysis. Because states other then location are not critical for any move in the "sheepdog" simulation, an agent space can be reduced to a simple 2D metric space.

The configuration space can be decomposed into two subspaces [7]: **free space** - the space attainable by moving agents, and **occupied space** - the unattainable space occupied by obstacles, the trees and the posts of the pen. Unlike the space of a "block world", the configuration space is continues, what causes additional problems. There are two major families of approaches: cell decomposition and skeletonisation. Both reduce the continuous path planning problem to a simpler one - discrete graph search.

3.3 Cell decomposition

The cell decomposition method decomposes a free space into a grid of contiguous usually square cells. An agent can move then along straight lines from cell to cell. Though the method is very

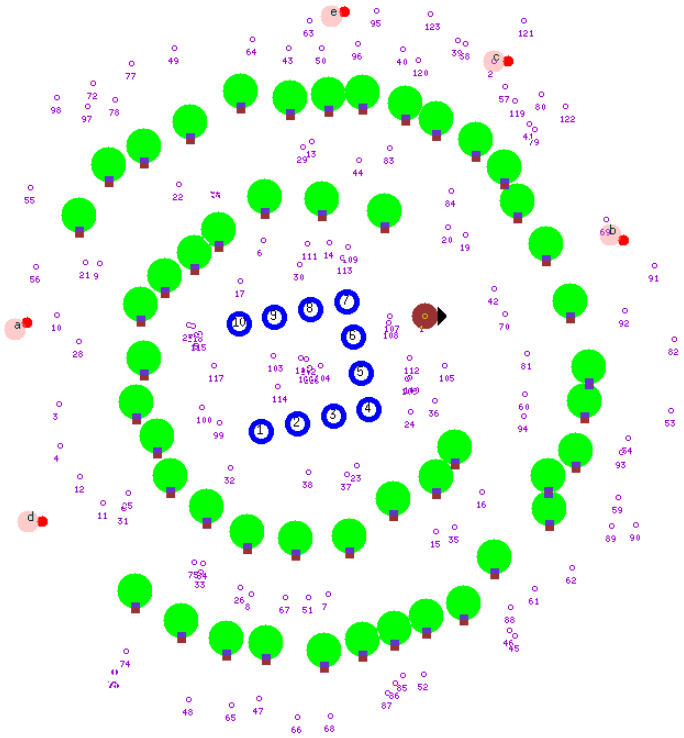

Figure 3.1: The probabilistic roadmap algorithm generates random graph nodes (violet numbered dots). The nodes are more likely generated in the nearness of obstacles but no closer and farther than certain predefined distances. Additionally, to avoid problems with tight spaces, a number of nodes are generated around the current goal (concerns sheep only) of the dog.

simple to implement it has two drawbacks [7]. Firstly, the number of grid cells grows exponentially with the number of space dimensions. Secondly, there is a problem with "half occupied" cells on the border between free and occupied space. These cells can be numbered among free and occupied cells, and the solution then will be unsound [7] (there may be no way across the cell in fact) or incomplete [7] (the free space needs further subdivision).

3.4 Skeletonisation

The methods from this group reduce the free space to a one-dimensional representation called a skeleton of the configuration space [7].

A **Voronoi graph** is a set of points equidistant to two or more obstacles in the configuration space. The Voronoi graph does not give the shortest path, but instead the created path is "secure" as it maximises the distance to the surrounding obstacles. The main weakness of the Voronoi graph method is difficulty with its calculation in high-dimensional spaces.

The **probabilistic roadmap** (see [4]) technique creates nodes of the graph in the free space with the use of random generator. The method is more general then all the previous ones, it deals better with open spaces, and more importantly it is easy to implement. Because a shape complexity of obstacles in the configuration space does not have to influence an algorithm complexity, and

a search can be carried out locally, the probabilistic roadmap tends to scale better with growing number of dimensions. It seems that the only inconvenience of this method is its incompleteness, as it does not guarantee finding a path from the root of the generated graph, to the goal.

The probabilistic roadmap is the solution employed in the "sheepdog" simulation (Figure 3.1).

3.5 Graph searching

Once we generated a set of random graph nodes, we can harness one of the graph search algorithms to find the least costly path or simply the shortest way to a goal.

The group of heuristic or informed graph search algorithms involves those proceeding searching preferentially through nodes that, "problem-specific information indicates might be on the best path to a goal" [6]. A* is an example of algorithm from this group, that guarantees finding minimal cost paths.

The problem-specific information is expressed by the function $f(n) = g(n) + h(n)$ defined for each node n of the graph, and where: $g(n)$ is a minimal cost path from the root node to n; $h(n)$ is a heuristic factor estimating the minimal cost path from n to a goal node.

I would not like to describe here a classical A* algorithm, akin to almost all AI books, but its modified version (read: faster) used in the "sheepdog" simulation. Namely, assuming that the triangle inequality on h() is satisfied, then when A* expands a node n, it has already found an optimal path to n (a proof one can find in [6]). This property allows us to use hash maps instead of Open and Closed lists [1] (see an original definition of A* in [6]).

The new algorithm A* would present as follows (C++/Java pseudo code - a Pop-11 code is attached in Appendix B-2):

```
// must be initialised with zeroes
HashMap Open, Close;
// n is the current node
int n = ROOT_NODE;
// r is the node, through which passes the best path to the current node n
int r = ROOT_NODE;

while (true) {
    // the best path to n goes through r
    Close(n) = r;
    // break the loop if the current node n is the goal
    if (n == GOAL_NODE) break;
    // then remove n from Open
    Open(n) = 0;

    // expand the current node n, iterate for all possible destination nodes j
    for ( int j = 1; j <= MAX_NODE; j++ )
        // do not expand if the destination node j is:
        // equal the expanded node n,
        // or is already on Close (see the assumption), or is not expandable
        if (j != n && Close(j) == 0 && isExpandable(n, j))
        {
            // calculate the cost of reaching j through n
```

[1] In fact, because the hash maps map node to node, and the maximal number of nodes is fixed and not very large, the most optimal solution is to use simple arrays.

```
            double c = getCost(n , j);
            // extract a node, through which passes the best path to j
            int k = Open(j);
            // redirect the best path to j if:
            // there was no path to j before, or the new path is less costly
            if (k == 0 || c < getCost(k, j)) Open(j) = n;
        }

    // find a node n with the lowest cost value
    n = findLowest(Open);
    // break the loop if Open is empty
    if (n == 0) break;
    // extract r
    r = Open(n);
}
```

A full path from the root to the goal can be extracted directly from Close by simple iteration `Close(n) -> n` until `n` points the root. The `getCost()` function employs implicitly mentioned *f()*, as well as `findLowest()` employs `getCost()`. In practise the `isExpandable()`[2] function is the real bottleneck of the algorithm since it involves an iteration over obstacles from the "sheepdog" world, roughly proportional to `MAX_NODE`.

3.6 Emergent problems

Basically, the most important of limitations derive from the probabilistic roadmap algorithm. Firstly, as mentioned before, it does not guarantee that a path from a root of a generated graph, to a goal can be found at all. This, unfortunately, can be checked only by carrying out a graph searching. Secondly, even if the path to the goal exists, it is almost never the shortest one, but irregular and angular. Thirdly, the path found could be topologically different for each run of the algorithm, what was depicted in Figure 3.2.

The simplest way of handling the first shortcoming is just to repeat a procedure of generating a graph, as long as a graph search algorithm finds a path. However, the second and the third shortcomings are not so easy to fix.

It is convenient to make the explicit distinction between path planning (plan preparation) and path following (plan execution). Path planning involves **deliberative** mechanisms of creation of a global general plan, which usually require a lot of CPU time. On the other hand, path following involves **reactive** mechanisms, which should allow an agent to react quickly to unexpected changes in the world, as well as to improve the agent's general plan because detailed information was not available during its creation. Thus, the deliberative part of the dog's "brain" searches for a global plan, and then the reactive part deals with *path optimising* and *"local" planning*, which are the topics of the next two chapters.

3.7 Path optimising

None of the paths shown in Figure 3.2 would be satisfactory for a real dog, to say nothing about any robot. A trivial solution of the problem is to increase a number of nodes in a graph, but unluckily it would result in a rapid growth (strongly dependent of the function *f()*) of amount of

[2] An algorithm which lies behind `isExpandable()` function is described in Chapter 3.9.

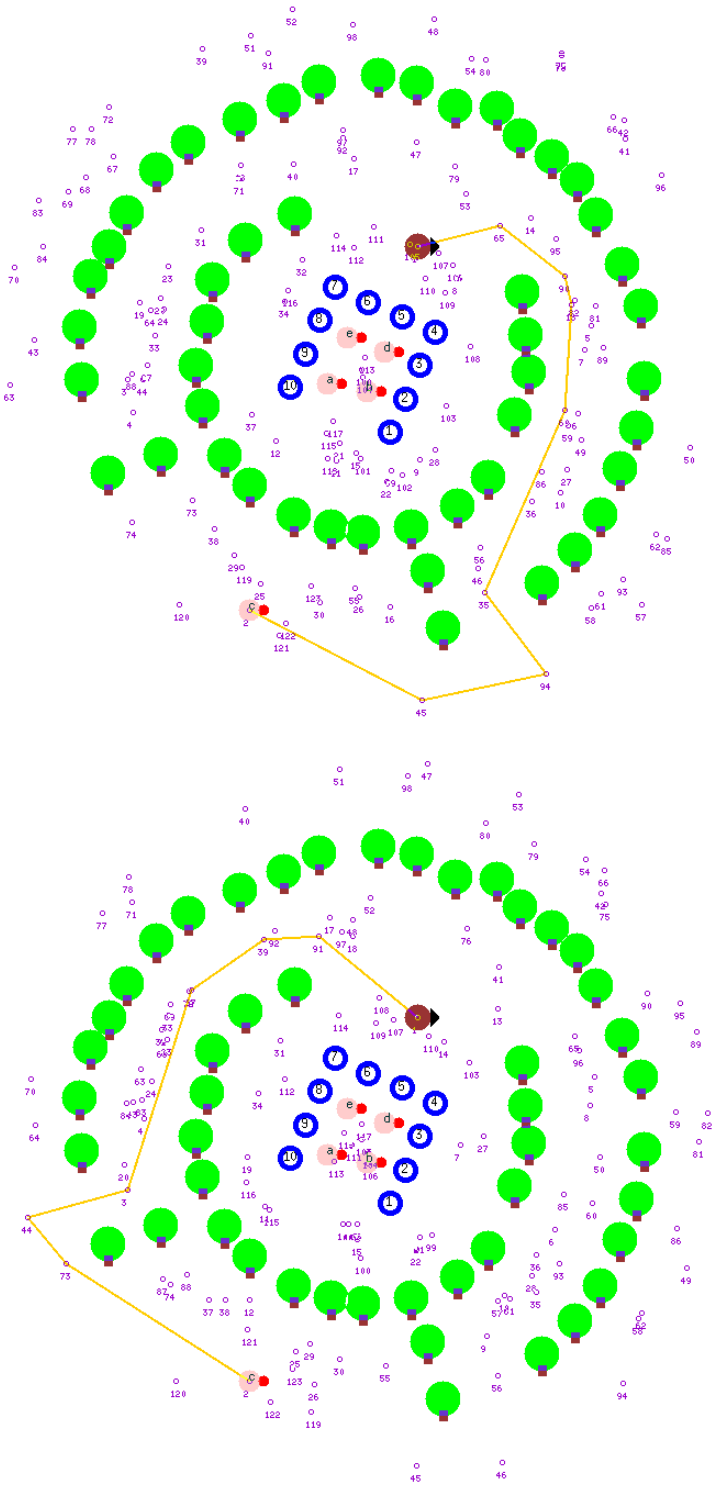

Figure 3.2: Two depicted configurations of nodes of the search graph reveal a serious weakness of the approach used. Because of the probabilistic nature of the algorithm it may find topologically different paths for each run. Intuitively, the two paths are topologically equal if it is possible to reduce them by "tightening" into one and the same path.

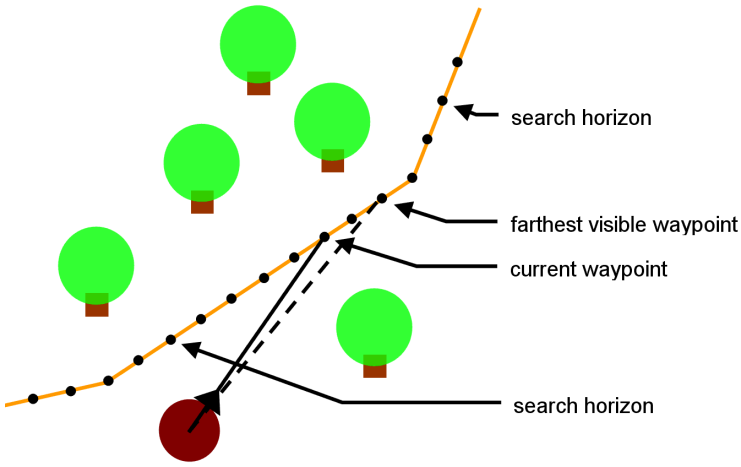

Figure 3.3: An agent always heads for any waypoint. This is an iterative process: the farthest visible waypoint becomes the current one in the next time step.

calculations. Another approach is an optimisation of a path "on the fly", so that a motion seems smooth and natural.

The idea is not to strictly follow a path[3], but instead to head towards it. Furthermore, because a path is directed, an agent can search for the farthest or closest to a goal, but still visible, path segment, and then head for it. Let a **waypoint** be a point defined in a configuration space used for motion navigation. One can describe then a path as a set of waypoints - initially a set of nodes of a graph defining a path to a goal (for instance see paths in Figure 3.2). We can thicken this initial set of waypoints adding newly generated ones, and as a result avoid again a continuity problem. An agent can head now for the farthest visible waypoint on a path to a goal[4] (Figure 3.3). Since an agent world is dynamical, the search for a waypoint is carried out starting from a current one in both "directions" and in a certain "search horizon" (Figure 3.3).

3.8 Local planning

As it is clearly visible in Figure 3.2, the probabilistic roadmap algorithm not only does not assure finding shortest paths (because of their irregular shape) to the goal, but also paths found could be topologically different. That is not all - this problem implies another one. A moving agent may come across an obstacle and then because it needs a new plan, it may create a new and topologically different path. This not only may lengthen the way to the goal, but in the worst case it can lead to a kind of oscillations between obstacles, if new paths go interchangeably in opposite directions[5]. Though local planning does not help solving the first problem, it removes the second one, and what is more important saves resources and CPU time as a local graph is usually much smaller.

The dog creates a new plan if the distance to a waypoint he has been heading for, became smaller then a certain threshold value. It usually means that a current path has been blocked by

[3]Steering a sheep exactly along a path is even impossible, due to its unpredictable behaviour.

[4]In reality, an agent always heads for any waypoint, what also simplifies collision detection by reducing it to searching for any visible waypoint and controlling the distance to this waypoint. This technique, however, might be difficult to apply to the real robot.

[5]Of course, this situation might happen only for obstacles periodically blocking each newly created path.

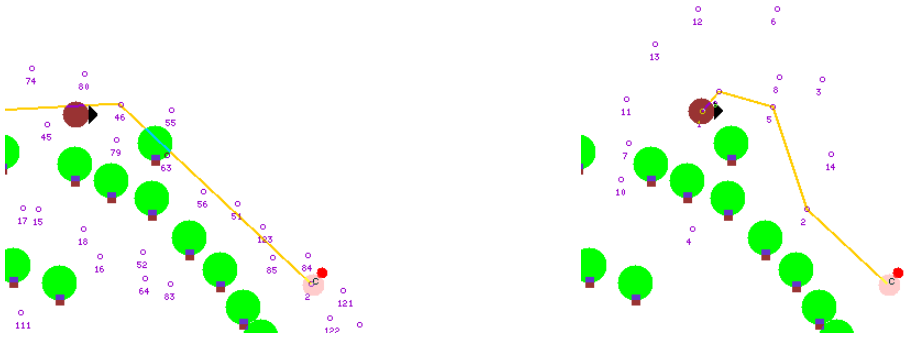

Figure 3.4: If an obstacle block a path, the dog generates a local path to go around this blocked area. A generated local graph and a merging waypoint all lie within a dog visual range.

an obstacle (Figure 3.4) - the dog cannot move farther, since he has already approached to the farthest (the closest to the goal) visible waypoint.

A procedure used tries to find a local path to a certain "merging" waypoint lying by the edge of an agent visual range. A complete path to a goal is then a concatenation of the local path and the old path made in the merging waypoint (Figure 3.4). Since this simple procedure does not assure finding a local path[6], in case of failure it is being repeated a few times before eventual switching to a global planner.

3.9 Obstacle avoidance

Wherever an agent moves to, it always heads for some previously chosen waypoint lying on the path to the goal[7]. The central role here belongs to a function which checks if there is a passage from one waypoint to any other - the `isExpandable()` function used in A* algorithm. The `isExpandable()` also hides implementation details of objects of the "sheepdog" world, is an interface between high and low level code (see Appendix B-3).

The question of existence of a passage between any two waypoints is an equivalent of finding a certain obstacle-free area that connects those waypoints. Since assumed that all agents in the "sheepdog" (including obstacles) are circular, and they are entirely characterised by their coordinates and radius, the problem can be decomposed as follows:

1. Create a path area connecting a start waypoint and a destination waypoint given their coordinates, a radius of an agent and a radius of an obstacle (Figure B-3).

2. Search for any obstacles (their coordinates) contained in those area.

The above procedure should return false if it finds any obstacle within created path area, or true otherwise.

[6] A merging waypoint may be obscured itself, for instance.

[7] When the dog steers a sheep, a plan is actually created only for the driven sheep. The dog then tries to keep the sheep on a path by changing his position behind the sheep, going to a certain waypoint (which in fact does not lie on the path).

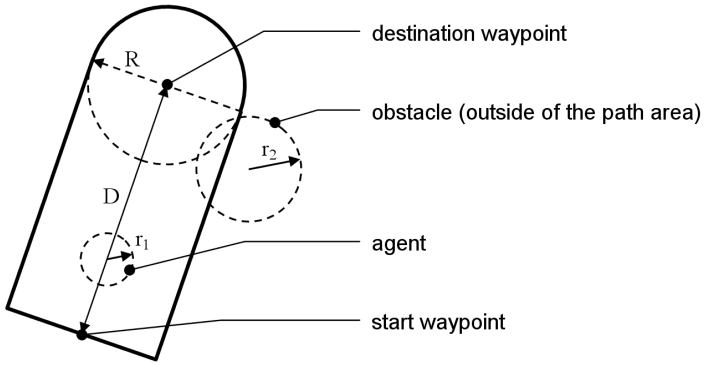

Figure 3.5: The path area the connecting start and destination waypoint. The shape and size of this area is entirely determined by the distance between waypoints (D), the radius of an agent ($r1$) and obstacles ($r2$), where $R = r1 + r2$.

Chapter 4

The program

Pop-11 is neither strictly an object oriented, nor procedural programming language. Additionally, it has some other unique features like rule-based mechanisms, what make an analysis of a Pop-11 written program in some way incomplete and unclear rather than more difficult.

Because of multi-aspectuality of Pop-11 written programs, I would like to describe the "sheepdog" demo from two different perspectives. The rule-based toolkit called SIM_AGENT, allows us to concentrate on a "behavioural" aspect of agents in the demo, and on the contrary, the classes used reveal more details of a "functional" aspect.

4.1 The SIM_AGENT toolkit

The SIM_AGENT toolkit supports both symbolic (rule-based) and sub-symbolic (e.g. neural network) mechanisms [11]. Essentially it is designed to explore various human-like capabilities framed in architectures, which go beyond those actually employed in the "sheepdog" demo. The following features are characteristic to the toolkit [11]:

- "minimal ontological commitment" allows exploring many agent architectures using neural networks, implementing multi-layered sensory interpretation, etc.
- Each agent is represented by an active object (that changes its state spontaneously), a passive object (that does not change its state at all, or only in response to any active object), or an object composed of other objects (modularity).
- The distinction between an external and internal behaviour of agents, determined by the Poprulebase condition-action rules system.
- Use of classes and inheritance that simplify a construction of complex objects. All objects must be inherited from the `sim_agent` base class.
- Rapid prototyping via Pop-11 incremental compilation.

A computer simulation implemented in Pop-11 with using the SIM_AGENT toolkit, is managed by a scheduler (a function `sim_scheduler()`), that allocates a time-slice to all objects from a list of managed objects (agents). Each "run" of a single object usually consists of three stages [11] [9] at which an agent does:

1. Sense its environment.

2. Interpret sensed data and messages, manipulate their internal states.

3. Produce an action and send messages influencing any other agents.

In fact, the scheduler runs twice each time-slice to avoid an influence of ordering of the object list - it has to separate an action execution.

The core of the SIM_AGENT toolkit is the Poprulebase package supporting "forward chaining system for specifying and running sets of condition-action rules" [10]. Thus, each active object (agent) has an associated hierarchical structure of condition-action rules communicating via internal agent database.

One can distinguish four levels in the Poprulebase hierarchy [10][8]:

1. The base element of the hierarchy is a single condition-action rule that is "fired" when a specified condition is satisfied.

2. The ruleset consists of an ordered set of rules that are sequentially tested for its "firing" condition.

3. The rulefamily consists of a set of rulesets, amongst which at each run only one can be runnable.

4. The topmost element is the rulesystem - an ordered collection of rulesets and rulefamilies.

It clearly follows that the rulesystem is this element, which decides of behaviour of an agent.

4.2 Behavioural perspective

4.2.1 The dog rulesystem

A structure of the dog rulesystem is shown in Table 4.1.

Apart from a short moment before an initialisation, the dog can perform one of two activities. Once the "Find" is activated, he chooses a current sheep - the sheep that is to be driven to the pen. After finding the current sheep, the "Steer" is activated. The whole cycle is repeated when the current sheep has been successfully driven to the pen.

The `rulesetDogPerception` is run at the beginning of each time-slice.

- The `watchObjects` rule senses other agents including obstacles.
- The `watchCurrentSheep` rule collects the most important information about the current sheep.
- The `checkGoal` checks if the main goal has been attained (all sheep in the pen) to eventually stop the simulation.
- The `updateTimer` updates a timer which is an indicator of a path problem. The path problem occurs when the dog or the driven sheep is not able to follow its path.

The `rulesetDogBehaviour` decides of a current activity of the dog.

- The `init` rule is fired once only, just after start of the demo, and initialises an agent database. It also switches an activity to "Find".
- The `find` rule controls "Find" activity by permanent checking all the conditions, which are necessary to switch the current activity to the "Steer".

Rulefamilies	Rulesets	Rules
	rulesetDogPerception	watchObjects watchCurrentSheep checkGoal updateTimer
	rulesetDogBehaviour	init find steer
rulefamilyDogActivity	rulesetDogPlan	problem plan switch
	rulesetDogFind	switch main problem
	rulesetDogSteer	switch main problem
	rulesetDogExec	exec default

Table 4.1: The structure of the dog rulesystem.

- The `steer` rule controls "Steer" activity by permanent checking all the conditions, which are necessary to switch the current activity to the "Find".

Once no plan is prepared, the `rulesetDogPlan` is activated. The ruleset is a member of the `rulefamilyDogActivity`.

- The `problem` rule is fired if a path problem occurs. It calculates a plan index - the index of a "merging" waypoint used for merging an old global plan and the new local one.
- The `plan` rule establishes which sheep is to be found and driven to the pen. It then generates a new local or global plan depending on the value of the plan index.
- Finally, the `switch` rule switches an active rulefamily to a rulefamily corresponding to a current activity.

The `rulesetDogFind` corresponds to "Find" activity.

- The `switch` rule activates the `rulesetDogSteer` if the "find" rule from the `rulesetDogBehaviour` has set a suitable flag.
- The `main` rule controls the dog movement, and sets a timer-counter flag if a path problem occurred.
- The `problem` rule activates the `rulesetDogPlan` if a path problem flag is set.

The `rulesetDogSteer` corresponds to "Steer" activity. The ruleset is nearly a twin of the `rulesetDogFind`.

- The `switch` rule activates the `rulesetDogFind` if the "steer" rule from the `rulesetDogBehaviour` has set a suitable flag.

Rulefamilies	Rulesets	Rules
	sheep_perception_rules	see_obstacle spy_one spy_more alert
	sheep_obstacle_rules	gather normalise activity
	sheep_instinct_rules	eat flee lonely contented
sheep_social_rules	avoidance	avoidance limit1 flipper1
	center	centre limit2 flipper2
	imitate	imitate limit3 flipper3
	sheep_resolve_behaviour_rules	resolve_flocking_and_fleeing resolve_movement_and_eating resolve_movement_and_idling resolve_movement_and_constraints
	sheep_action_rules	default_wander_action default_idle_action eat_action positive_action

Table 4.2: The structure of the sheep rulesystem.

- The `main` rule controls the dog movement, as well as indirectly a movement of the driven sheep. It sets a timer-counter value as a flag indicating a path problem.
- The `problem` rule activates the `rulesetDogPlan` if a path problem flag is set.

Finally, the `rulesetDogExec` collects and executes all passed commands by means of the `exec` rule, and otherwise if there are no any commands awaiting execution, the `default` rule is fired.

4.2.2 The sheep rulesystem

Since, as a sheep "design" is not the main subject of this report, I will only describe it in short. A structure of the sheep rulesystem is shown on the Table 4.2.

The table clearly points up a complexity of a typical "behavioural" design, though the old design of the dog would be a better example. The sheep displays the following instincts:

- Avoiding obstacles is one of the very base ones.

- Fleeing is an effect of presence of a dog. Its intensity depends not only on a distance to a dog, but on a dog speed and "barking level" as well.
- Eating may stop wandering if a sheep is hungry enough.
- Wandering intensifies only in absence of other sheep in neighbourhood.
- Flocking, on the contrary to wandering, intensifies only in presence of other sheep in neighbourhood.
- Imitating encourages a sheep to follow its closest friend.

The `sheep_resolve_behaviour_rules` ruleset resolves ultimately all those frequently conflicting instincts.

4.3 Functional perspective

Before I get to the point I would like remark on a few atypical features of Pop-11 language. First of all, it is difficult to say if a certain function is a method of any class since in the class declaration there are only declarations of components (variables), so one can add a method to any class from any library without modifying it! Moreover, there is nothing like a static or final class member, what makes grouping of methods operating on the same resources a bit tricky. All these features harmonize well with a concept of Pop-11 flexibility, though one has to be careful when writing programs in Pop-11 language[1].

Figure 4.1 depicts a structure of classes of the "sheepdog" demo. The classes drawn up with dashed line are a part of Pop-11 libraries, so they are not a part of the program.

All classes can be split up into two general groups:

- Agent classes are mostly those derived from the `sim_agent` class - a skeleton of agent.
- Helper classes are those used for visualisation (`VisualObj`), path planning (`Waypoint`, `Path`), and written as a support of agent classes (`Pen`).

The `VisualObj` is a very simple class representing visual aspects of a single search graph vertex or an edge of path. All instances of `VisualObj` class are collected in the Dog class, as well as the methods which initialise it.

The `Pen` is a class that is a kind of a "centre" of the demo. All agents, including posts (the `Post` class) comprising the pen, are being transformed and located with respect to an initial data stored in this class.

The `Waypoint` class contains parameters characterising a single waypoint or just a node, like coordinates and a cost value. Most of the methods using this class are members of the `Animal` class.

Members of a bit more complex `Path` class define and operate on a path area, which was described in one of the previous chapters. The `create()` method initialises an `Path` object, the methods `contains1()` and `contains2()` test if a given point is contained within created path area.

The `Animal` is the base class for all (active) agents using path planning. Though it has no components (sheep do not use planning yet), it contains key methods used in path planning:

- The *probabilistic roadmap methods* `graphGenerateInit()`, `graphGenerate()` and `graphGenerateList()` are used for generation a random graph (see Appendix B-1).

[1] There are much more such unusual features, for example free access to all class components or no type control.

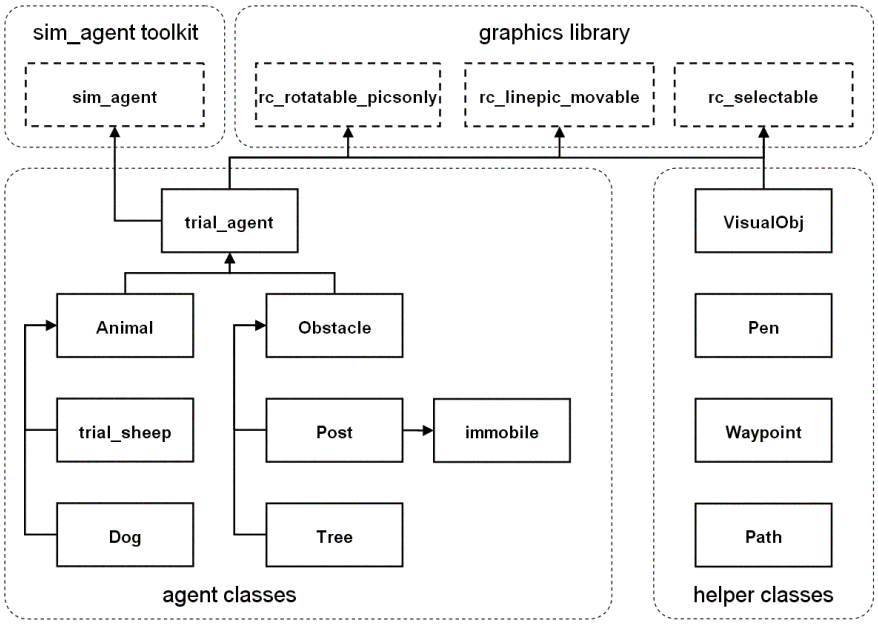

Figure 4.1: The structure of the "sheepdog" demo classes. The dashed rectangles are classes being a part of Pop-11 libraries.

- The *graph heuristic function* `graphHeuristicFunc()` and the *graph search function* `graphSearchA()` (see Appendix B-2).
- The methods `waypointTransform()` and `waypointFindNext()`, which operate on waypoints and are used for *path optimisation* (see Appendix B-4).
- The *visualisation methods* `verticesShow()`, `egdesShow()` and `directionShow()` are used for displaying graph vertices (nodes), path edges and a current movement direction.

Finally, the `Dog` class contains mostly utility methods like those used for initialising components (e.g. `setupDog()`), sensing objects (`collectObjects()`), moving (`moveDog()`) etc. Of course, above description is far from being complete. More detailed code documentation can be found in the source file.

Chapter 5

Conclusions

The new "sheepdog" demo is thought to be a framework of a multi-agent simulation rather than its final version. All the classes making up the "brain" of the dog can be easily applied to any other agent. Moreover, they are designed to work on fairly separate functional levels, so it would be possible, for instance, to construct a completely different agent rule system. Many aspects of agent behaviour can be modified just through program constants, which are all collected in one place, clearing up in this manner the code from any "magic" numbers. The intelligence of the dog can be improved by extending a visible range - the main parameter responsible for a "range" of a local plan (see Chapter 3.8). Extending the search horizon (see Chapter 3.7) may help the dog to deal better with path optimisation. Changing the number of generated nodes of a search graph can minimise amount of computation for a specified environment, and so on.

The overall result of the simulation is promising - the dog is able to complete the task, avoiding dynamically obstacles and changing the plan if the path has been blocked. The program, however, still needs lots of enhancements. First of all, it would be interesting to relax the assumptions about the rounded obstacle and agent shapes, making the simulation more realistic. Secondly, the dog in the current design may simply "jump" over a driven sheep in order to keep the sheep on a path. Instead, the dog would create a special mini-plan to get on the sheep back, moving carefully not to startle it. Thirdly, an algorithm that controls movement in the nearness of obstacles, may too often call a path planner, even if the dog only momentarily loses sight of the driven sheep. A possible solution to this problem can be a slight modification of the path area shape (see Figure B-3), although a detailed analysis of this problem could be a topic of a whole chapter. Finally, since the SIM_AGENT toolkit allows coexisting diverse agent architectures [1] (see Chapter 4.1), there is a large room to manoeuvre in redesigning of the dog rulesystem, by supporting it (see Chapter 4.2.1) with new rules and rulefamilies, what can better adjust the system to the specified tasks.

Apart from the mentioned possible improvements, one can imagine more fundamental ones. The very first improvement should probably involve smarter skeletonisation procedure, mainly because the current algorithm generates plenty of redundant graph nodes, which are never used as waypoints. Then, to find a path to a goal, instead of algorithm A* one can harness for example an anytime algorithm - the one from the group of approximate search algorithms. Anytime algorithms are especially valuable in case of limited computational resources, since they can be stopped at anytime producing an useful result - an approximate, not always reliable, but path to a goal. From the other hand, if we take into consideration the kinematic properties of real material objects (e.g. robots) we face a problem of feasibility of a created plan. The path can be just too wavy to be passable for a moving car. An interesting algorithm proposed by Steven M. La Valle [12], attempts

[1] A good example can be a reactive architecture of the sheep and a deliberative one of the dog

to connect randomly generated “pairs of nearby configurations”. Because produced random graphs are usually similar to spread trees, the technique or rather so created data structure is called Rapidly-exploring Random Trees (RRT).

Appendix A

Running the "sheepdog" simulation

To run the "sheepdog" demo, follow the instructions below.

1. Install the POPLOG software development environment on Linux or other UNIX platform (for details see http://www.cs.bham.ac.uk/research/poplog/poplog.info.html).
2. Open the program source file in the XVed editor (some basic information can be found at http://www.cs.bham.ac.uk/research/poplog/doc/popxvedhelp/xved).
3. Type `<ENTER> l1 <RETURN>`, and then `<ESC> D` on the printed line `run_sheep(20000)`.

Appendix B

Source code

Because the program source file is rather long, I have only collected a few key functions and methods. The complete source code is currently accessible at `~msc35msk/work/Mini-project1/sim_sheepdog.p`, and should be soon available on the Birmingham University web pages.

B-1 Probabilistic roadmap routines

```
;;; The probabilistic roadmap method generating an initial configuration of
;;; nodes (waypoints) indexed IDX_ROOT and IDX_GOAL.
define :method graphGenerateInit(
    this:Animal,
    x1, y1,
    x2, y2,
    wpL) -> wpNum;

    lvars wp;

    wpL(IDX_ROOT) -> wp;
    round(x1), round(y1) -> (wp.x, wp.y);
    wpL(IDX_GOAL) -> wp;
    round(x2), round(y2) -> (wp.x, wp.y);
    2 -> wpNum;
enddefine;

;;; The probabilistic roadmap method that generates nCount of random nodes
;;; (waypoints) and stores them in array wpL starting from index wpIdx + 1.
;;; All random nodes are uniformly distributed within (rMin, rMin + rWidth)
;;; circular area avoiding obstacles from oList.
;;; The method returns the total number of random nodes in wpL.
define :method graphGenerate(
    this:Animal,
    oList,
    obj,
    nCount, rMin, rWidth,
    wpL, wpIdx) -> wpNum;
```

```
    lconstant MAX_TRIALS = 100;
    lvars wp, j;

    wpIdx -> wpNum;
    repeat nCount times
        wpNum + 1 -> wpNum;
        wpL(wpNum) -> wp;

        for 1 -> j step j + 1 -> j till j > MAX_TRIALS do
            generatePoint(rMin, rWidth) -> (wp.x, wp.y);
            intof(wp.x) + obj.rc_picx, intof(wp.y) + obj.rc_picy -> (wp.x, wp.y);

            if not(collisionCheck(oList, getCoords(wp), rMin)) then
                quitloop(1);
            endif;
        endfor;
    endrepeat;
enddefine;

;;; The probabilistic roadmap method that generates nCount of random nodes
;;; (waypoints) for each obstacle from oList, and stores them in array wpL
;;; starting from index wpIdx + 1.
;;; All random nodes are uniformly distributed within (rMin, rMin + rWidth)
;;; circular area avoiding obstacles from oList.
;;; The method returns the total number of random nodes in wpL.
define :method graphGenerateList(
    this:Animal,
    oList,
    nCount, rMin, rWidth,
    wpL, wpIdx) -> wpNum;

    lvars o;

    wpIdx -> wpNum;
    for o in oList do
        graphGenerate(
            this,
            oList,
            o,
            nCount, rMin, rWidth,
            wpL, wpNum) -> wpNum;
    endfor;
enddefine;
```

B-2 Graph search algorithm

```
;;; The simplified heuristic function f(n) = g(n) + h(n), where h(n) = 0.
;;; Here f() calculates the relative cost of a move from (x1, y1) to (x2, y2).
```

```
;;; The function must satisfy the triangle inequality.
define :method graphHeuristicFunc(
    this:Animal,
    x1, y1,
    x2, y2) -> c;

    ;;; simplified
    round(sim_distance_from(x1, y1, x2, y2)) -> c;
enddefine;

;;; A modified version of A* graph search algorithm, where:
;;; oList is an obstacle list,
;;; dA and dO are suitably an agent and a maximal obstacle size,
;;; wpL1 is an initial search graph, containing wpNum1 nodes (waypoints), and
;;; wpL2 is a result set of wpNum2 waypoints, which constitutes
;;; an outcoming path. If wpNum2 is equal 0 no path has been found.
define :method graphSearchA(
    this:Animal,
    oList,
    dA, dO,
    wpL1, wpNum1,
    wpL2) -> wpNum2;

    lvars procedure
        C        = newanyarray([1 ^wpNum1 1 ^wpNum1], initshortvec, subscrshortvec),
        Open    = newanyarray([1 ^wpNum1], initshortvec, subscrshortvec),
        Close    = newanyarray([1 ^wpNum1], initshortvec, subscrshortvec);
    lvars
        rCurrent, nCurrent, cCurrent, j, c, k, wpn, wpj;

    ;;; C, Open and Close must be initialised with zeroes.
    ;;; nCurrent is the current node.
    ;;; rCurrent is the node, through which passes the best path to nCurrent.
    ;;; cCurrent is a cost of reaching nCurrent.

    IDX_ROOT ->> rCurrent -> nCurrent;
    0 -> cCurrent;
    while true do
        ;;; The best path to nCurrent goes through rCurrent.
        rCurrent -> Close(nCurrent);

        ;;; Break the loop if nCurrent is the goal.
        if nCurrent = IDX_GOAL then
            quitloop(1);
        endif;

        ;;; then remove nCurrent from Open.
        0 -> Open(nCurrent);
```

```
;;; Expand the current node nCurrent,
;;; iterate for all possible destination nodes j
for 1 -> j step j + 1 -> j till j > wpNum1 do
    ;;; Do not expand if the destination node j is:
    ;;; equal the expanded node n, or is already on Close.
    if nCurrent /= j and Close(j) = 0 then
        wpL1(nCurrent) -> wpn;
        wpL1(j) -> wpj;

        ;;; Check if j is expandable.
        if isExpandable(
            this,
            oList,
            getCoords(wpn),
            getCoords(wpj),
            dA, dO)
        then
            ;;; Calculate the cost of reaching j through nCurrent.
            graphHeuristicFunc(
                this,
                getCoords(wpn),
                getCoords(wpj)) + cCurrent -> c;

            ;;; Extract a node, through which passes the best path to j.
            Open(j) ->    k;
            ;;; Redirect the best path to j if there was no
            ;;; path to j before, or the new path is less costly.
            if k = 0 or C(k, j) > c then
                nCurrent -> Open(j);
            endif;

            c
        else

            INF
        endif -> C(nCurrent, j);
    endif;
endfor;

;;; Find a node nCurrent with the lowest cost value.
0 -> nCurrent;
INF -> cCurrent;
for 1 -> j step j + 1 -> j till j > wpNum1 do
    Open(j) ->    k;
    if k > 0 then
        C(k, j) -> c;
        if c < cCurrent then
            c, k, j -> (cCurrent, rCurrent, nCurrent);
        endif;
```

```
            endif;
        endfor;

        ;;; Break the loop if Open is empty.
        if nCurrent = 0 then
            quitloop(1);
        endif;
    endwhile;

    ;;; Build a outcoming path as an array of wpNum2 waypoints by
    ;;; simple iteration Close(n) -> n until n points the root.
    0 -> wpNum2;
    if nCurrent > 0 then
        for 1 -> j step j + 1 -> j till j > waypointMax do
            Close(nCurrent) -> rCurrent;
            wpL1(nCurrent) -> wpn;
            wpL2(j) -> wpj;
            getCoords(wpn), C(rCurrent, nCurrent) -> (wpj.x, wpj.y, wpj.cost);
            if nCurrent = IDX_ROOT then
                j -> wpNum2;
                quitloop(1);
            endif;
            rCurrent -> nCurrent;
        endfor;
    endif;
enddefine;
```

B-3 Path area class and methods

```
;;; Represents a path area between two waypoints. The path area consists of
;;; two parts: a rectangular one defined by 4 linear equations, and
;;; a circular one defined by a pair of coordinates and a square of radius.
define :class Path;
    ;;; Parameters of 4 equations of a path area edges: fn(x) = an*x + bn
    slot a1 = 0;
    slot b1 = 0;
    slot a2 = 0;
    slot b2 = 0;
    slot a3 = 0;
    slot b3 = 0;
    slot a4 = 0;
    slot b4 = 0;
    ;;; Coordinates of a destination waypoint and
    ;;; a square of radius of a circular part
    slot x  = 0;
    slot y  = 0;
    slot rr = 0;
    ;;; True if the path area is an "inclined" rectangle
```

```
    slot isInclined = false;
enddefine;


;;; Creates a path area defined by coordinates of two waypoints and parameter r,
;;; where r is a "width" of a path or a radius of a circular part of a path.
define :method create(
    p:Path,
    x1, y1,
    x2, y2,
    r);

    lvars dx, dy, tmp;

    ;;; {x1, y1, x2, y2} must be integers
    round(x1), round(y1), round(x2), round(y2) -> (x1, y1, x2, y2);
    ;;; Store coordinates of a destination waypoint and
    ;;; a square of a "width" of a path
    x2, y2, r*r -> (p.x, p.y, p.rr);

    ;;; Perform a transformation of an area
    ;;; There can be 4 cases of non inclined rectangular path, and
    ;;; 4 cases of inclined one

    if x1 = x2 then
        ;;; Non inclined path, cases #1-2

        if y1 > y2 then y1, y2 -> (y2, y1); endif;
        y1     ->> p.b1 -> p.b4;
        y2     ->> p.b2 -> p.b3;
        x1 - r ->> p.a1 -> p.a2;
        x1 + r ->> p.a3 -> p.a4;

        false
    elseif y1 = y2 then
        ;;; Non inclined path, case #3-4

        if x1 > x2 then x1, x2 -> (x2, x1); endif;
        x1     ->> p.a1 -> p.a2;
        x2     ->> p.a3 -> p.a4;
        y1 - r ->> p.b1 -> p.b4;
        y1 + r ->> p.b2 -> p.b3;

        false
    else
        ;;; Inclined path, cases #1-4
        ;;; Determine the parameters of equation fn(x) = an*x + bn
        ;;; for four edges (a1, b1) ... (a4, b4)
```

```
        x2 - x1 -> dx; ;;; cannot be 0
        y2 - y1 -> dy; ;;; cannot be 0
        sqrt(dx*dx + dy*dy) -> tmp;
        r*dx/tmp -> dx;
        r*dy/tmp -> dy;

        x1 - dy, y1 + dx,
        x2 - dy, y2 + dx,
        x2 + dy, y2 - dx,
        x1 + dy, y1 - dx ->
            if dx > 0 then
                if dy > 0 then
                    (p.a1, p.b1, p.a2, p.b2, p.a3, p.b3, p.a4, p.b4)
                else
                    (p.a2, p.b2, p.a3, p.b3, p.a4, p.b4, p.a1, p.b1)
                endif
            else
                if dy > 0 then
                    (p.a4, p.b4, p.a1, p.b1, p.a2, p.b2, p.a3, p.b3)
                else
                    (p.a3, p.b3, p.a4, p.b4, p.a1, p.b1, p.a2, p.b2)
                endif
            endif;

        ;;; A divisor nor tmp will never be equal 0
        realof(p.b2 - p.b1)/(p.a2 - p.a1) -> tmp;

        ;;; fn(x) = an*x + bn
        tmp   , p.b1 - tmp*p.a1 -> (p.a1, p.b1);
        -1/tmp, p.b2 + p.a2/tmp -> (p.a2, p.b2);
        tmp   , p.b3 - tmp*p.a3 -> (p.a3, p.b3);
        -1/tmp, p.b4 + p.a4/tmp -> (p.a4, p.b4);

        true
    endif -> p.isInclined;
enddefine;

;;; Checks if the rectangular part of the path contains a point (x, y)
define :method contains1(
    p:Path,
    x, y) -> bool;

    if p.isInclined then
        y < p.a1*x + p.b1 and y < p.a2*x + p.b2 and y > p.a3*x + p.b3 and y > p.a4*x + p.b4
    else
        x > p.a1 and x < p.a3 and y > p.b1 and y < p.b3
    endif -> bool;
enddefine;
```

```
;;; Checks if the circular part of the path contains a point (x2, y2)
define :method isInArea2(
    p:Path,
    x2, y2) -> bool;

    x2 - p.x, y2 - p.y -> (x2, y2);
    x2*x2 + y2*y2 < p.rr -> bool;
enddefine;

;;; Checks if both the rectangular and the circular part of the path
;;; contains a point (x, y). This is the most frequently called function, mostly
;;; by graph search rutines. As it is a bottleneck of the system, the method
;;; must be as fast as possible.
define :method contains2(
    p:Path,
    x, y) -> bool;

    if p.isInclined then
        y < p.a1*x + p.b1 and y < p.a2*x + p.b2 and y > p.a3*x + p.b3 and y > p.a4*x + p.b4 or
        isInArea2(p, x, y)
    else
        x > p.a1 and x < p.a3 and y > p.b1 and y < p.b3 or
        isInArea2(p, x, y)
    endif -> bool;
enddefine;

;;; Checks if there is a passage between waypoints (x1, y1) and (x2, y2).
;;; oList is an obstacle list.
;;; dA and dO are suitably an agent and a maximal obstacle size.
define :method isExpandable(
    this:Animal,
    oList,
    x1, y1,
    x2, y2,
    dA, dO) -> bool;

    lconstant p1 = instance Path;
    endinstance;
    lconstant p2 = instance Path;
    endinstance;
    lvars o;

    ;;; To speed up calculations a search is carried out for a path,
    ;;; which has the "largest" possible size, defined by a maximal obstacle size.
    ;;; If any obstacle lies within this path (p1), a new "smaller" one (p2) is
    ;;; created basing on a real size of the found obstacle.

    create(p1, x1, y1, x2, y2, dA + dO);
```

```
    true -> bool;
    for o in oList do
        if contains2(p1, getCoords(o)) then
            create(p2, x1, y1, x2, y2, dA + o.trial_size);
            if contains2(p2, getCoords(o)) then
                ;;; It means that obstacle o lies within p2.
                ;;; A path from (x1, y1) to (x2, y2) is not passable
                false -> bool;
                return();
            endif;
        endif;
    endfor;
enddefine;
```

B-4 Path optimisation routines

```
;;; The key path optimisation method. Given wpL1 array of
;;; waypoints of length wpNum1 which constitutes an initial path, the method
;;; adds new waypoints to this path by making wpL1 more "dense".
;;; An outcoming set of waypoints of size wpNum2 is stored in wpL1 starting
;;; from index wpIdx2 + 1. In other words to the old "dense" path wpL2 is
;;; concatenated a new one generated from wpL1, but only if wpIdx2 is greater than 1.
;;; The element wpL1(wpIdx2) is then a "merging" waypoint.
define :method waypointTransform(
    this:Animal,
    wpL1, wpNum1,
    wpL2, wpIdx2) -> wpNum2;

    lvars
        dc, j, wp1, wp2, costDelta, x1, y1, c1, x2, y2, c2, dx, dy;

    ;;; Check if there is enough free space (waypoints) in wpL2.
    if wpNum1 = 0 or wpNum1 > waypointMax - wpIdx2 + 1 then
        0 -> wpNum2;
        return();
    endif;
    wpIdx2 -> wpNum2;

    ;;; root: j = wpNum1, cost = cost(wpL1(wpNum1)) = minimum
    ;;; goal: j = 1, cost = cost(wpL1(1)) = maximum

    wpL1(1) -> wp1;
    getCoords(wp1), wp1.cost -> (x1, y1, c1);

    wpL2(wpNum2) -> wp2;
    if wpNum2 > 1 then wp2.cost - wp1.cost else 0 endif -> costDelta;
    x1, y1, c1 -> (wp2.x, wp2.y, wp2.cost);
    true -> wp2.isNode;
```

```
    ;;; If wpIdx2 > 1 then all cost values from the remaining old path wpL2
    ;;; must be "rescaled" in order to avoid potentially harmful discontinuities
    ;;; in the "merging" waypoint.
    if costDelta < 0 then
        for 1 -> j step j + 1 -> j till j >= wpNum2 do
            (wpL2(j)).cost - costDelta -> (wpL2(j)).cost;
        endfor;
    endif;

    ;;; Find a delta cost value which determines the distance between waypoints.
    ;;; The value cannot be smaller than waypointCostDelta.
    max(realof(c1)/(waypointMax - wpNum1 - wpNum2), waypointCostDelta) -> dc;

    ;;; Create a new "dense" set of waypoints basing on wpL1, and add them to wpL2.
    for 2 -> j  step j + 1 -> j till j > wpNum1 do
        wpL1(j) -> wp1;
        getCoords(wp1), wp1.cost -> (x2, y2, c2);

        dc*(x2 - x1)/(c2 - c1), dc*(y2 - y1)/(c2 - c1) -> (dx, dy);
        ;;; Calculate the cost value of a new waypoint
        c1 - dc -> c1;
        ;;; Keep adding to wpL2 all newly generated waypoints lying "before"
        ;;; (in a cost sense) a next waypoint (x2, y2, c2) from wpL1.
        while c1 > c2 do
            wpNum2 + 1 -> wpNum2;
            wpL2(wpNum2) -> wp2;
            x1 - dx, y1 - dy -> (x1, y1);
            round(x1), round(y1), round(c1) -> (wp2.x, wp2.y, wp2.cost);
            false -> wp2.isNode;

            c1 - dc -> c1;
        endwhile;

        ;;; The waypoints from wpL1 which constitutes an initial path are
        ;;; always being added to wpL2.
        wpNum2 + 1 -> wpNum2;
        wpL2(wpNum2) -> wp2;
        x2, y2, c2 -> (wp2.x, wp2.y, wp2.cost);
        true -> wp2.isNode;

        x2, y2, c2 -> (x1, y1, c1);
    endfor;
enddefine;
```

B-5 The dog rulesystem

```
/*
-- Rulesets and rulefamily for dog
```

```
*/

define :ruleset rulesetDogPerception;
    [DLOCAL [prb_allrules = true]];
    [LVARS [this = sim_myself][name = this.sim_name]];

    ;;; Sense all obstacles, including these local ones within visual range.
    ;;; Find all sheep staying out of the pen.
    RULE watchObjects
    ==>
        [POP11
            collectObjects(this, all_agents);
        ]

    RULE watchCurrentSheep
        [WHERE this.sheepCurrent]
    ==>
        [POP11
            lvars x, y, d;

            this.sheepCurrentX, this.sheepCurrentY -> (x, y);
            getCoords(this.sheepCurrent) -> (this.sheepCurrentX, this.sheepCurrentY);
            if this.sheepCurrentVisible then
                sim_distance_from(x, y, this.sheepCurrentX, this.sheepCurrentY)
            else
                0
            endif -> this.sheepCurrentSpeed;

            getDistance(
                this,
                this.sheepCurrent
            ) -> this.sheepCurrentDist;

            this.sheepCurrentDist < this.visualRange and
            isExpandable(
                this,
                this.obstacleLocalListDog,
                getCoords(this),
                getCoords(this.sheepCurrent),
                searchDA, searchDO
            ) -> this.sheepCurrentVisible;
        ]

    RULE checkGoal
        [WHERE this.sheepOutOfPenList = []]
    ==>
        [SAY ?name 'is very happy to complete all the tasks']
        [POP11
            sim_stop_scheduler();
```

```
        ]

    RULE updateTimer
    ==>
        [POP11
            this.timer + 1 -> this.timer;
        ]
enddefine;


define :ruleset rulesetDogBehaviour;
    [LVARS [this = sim_myself]];

    RULE init
        [NOT initBehaviour]
    ==>
        [initBehaviour]
        [LVARS index done activity]
        [POP11
            1, false -> (index, done);
            "rulesetDogFind" -> activity;
        ]
        [behaviour plan ?index ?done]
        [behaviour ?activity]

    RULE find
        [behaviour ?activity][->> item1]
        [WHERE activity = "rulesetDogFind"]
        [behaviour plan ?index ?done][->> item2]
        [WHERE done = true]
    ==>
        [POP11
            if this.sheepCurrentVisible and this.sheepCurrentDist < dogSheepDistMax then
                printf(
                    'Behaviour find, current sheep: visible = %P distance = %P\n',
                    [%this.sheepCurrentVisible, this.sheepCurrentDist%]);

                1, false -> (index, done);
                "rulesetDogSteer" -> activity;
            else
                0 -> this.bark;
            endif;
        ]
        [DEL ?item1 ?item2]
        [behaviour ?activity]
        [behaviour plan ?index ?done]

    RULE steer
        [behaviour ?activity][->> item1]
```

```
        [WHERE activity = "rulesetDogSteer"]
        [behaviour plan ?index ?done][->> item2]
        [WHERE done = true]
    ==>
        [POP11
            lvars b;
            isInPen(this.sheepPen, getCoords(this.sheepCurrent)) -> b;
            if not(this.sheepCurrentVisible) or b then
                printf(
                    'Behaviour steer, current sheep: visible = %P is in pen = %P\n',
                    [%this.sheepCurrentVisible, b%]);

                1, false -> (index, done);
                "rulesetDogFind" -> activity;
                if b then
                    false ->> this.sheepCurrent -> this.sheepCurrentVisible;
                endif;
            elseif this.sheepCurrentSpeed then
                if this.sheepCurrentSpeed < dogSheepSteerSpeed then
                    if this.bark < dogBarkMax then this.bark + 1 else dogBarkMax endif
                else
                    if this.bark > 0 then this.bark - 1 else 0 endif
                endif -> this.bark;
            endif;
        ]
        [DEL ?item1 ?item2]
        [behaviour ?activity]
        [behaviour plan ?index ?done]
enddefine;


define :ruleset rulesetDogPlan;
    [DLOCAL [prb_allrules = true]];
    [LVARS [this = sim_myself][name = this.sim_name]];

    RULE problem
        [WHERE this.pathProblemTm > 0]
        [behaviour plan ?index ?done][->> item1]
    ==>
        [POP11
            if this.timer > this.pathProblemTm + dogPlanLocalTrials + 1 then
                0 -> this.pathProblemTm;
                1
            else
                waypointPlanIndex(this, this.visualRange - searchDAPath)
            endif -> index;

            false -> done;
        ]
```

```
        [DEL ?item1]
        [behaviour plan ?index ?done]
        [SAY ?name 'needs a new plan ...']

    RULE plan
        [behaviour plan ?index ?done][->> item1]
        [WHERE done = false]
        [behaviour ?activity]
    ==>
        [SAY 'searching for the plan ( index =' ?index ') ...']
        [POP11
            lvars range, isLocal, goal, list;

            if activity = "rulesetDogFind" then
                if this.sheepCurrent = false then
                    selectSheep(this) -> this.sheepCurrent;
                endif;
                this, this.sheepCurrent, this.obstacleLocalListDog
            else
                this.sheepCurrent, this.sheepPen, this.obstacleLocalListSheep
            endif -> (this.currentRoot, this.currentGoal, list);

            this.visualRange - searchDAPath -> range;
            getDistance(this.currentRoot, this.currentGoal) < range and
            this.pathProblemTm > 0 or
            index > 1 -> isLocal;

            if not(isLocal) then
                this.obstacleList -> list;
            endif;

            if index > 1 then
                (this.wpPList)(index)
            else
                this.currentGoal
            endif -> goal;

;;;[isLocal ^isLocal problemTm (%this.timer% %this.pathProblemTm%)]=>

            graphGenerateInit(
                this,
                getCoords(this.currentRoot),
                getCoords(goal),
                this.wpGList
            ) -> this.wpGNum;

            if isLocal then
                graphGenerate(
                    this,
```

```
        list,
        this.currentRoot,
        waypointGenLocalNum*length(list), waypointORLocalMin, range,
        this.wpGList, this.wpGNum
    ) -> this.wpGNum;
else
    graphGenerateList(
        this,
        list,
        waypointGenNum, waypointORMin, waypointORWidth,
        this.wpGList, this.wpGNum
    ) -> this.wpGNum;
    /*graphGenerate(
        this,
        list,
        pen,
        waypointGenNum*length(list),
        waypointORLocalMin,
        min(sheep_window_xsize, sheep_window_ysize)/2,
        this.wpGList, this.wpGNum
    ) -> this.wpGNum;*/
    graphGenerate(
        this,
        list,
        this.sheepCurrent,
        waypointGenSheepNum, waypointORLocalMin, waypointORWidth,
        this.wpGList, this.wpGNum
    ) -> this.wpGNum;
endif;

routeFind(
    this,
    list,
    this.wpGList, this.wpGNum,
    this.wpPList, index
) -> this.wpPNum;

this.wpPNum > 0 -> done;

if done then
    this.wpPNum -> this.wpIdx;
    0 -> this.pathProblemTm;

    verticesShow(
        this,
        sheep_win,
        showVertices,
        this.wpGList, this.wpGNum);
    egdesShow(
```

```
                    this,
                    sheep_win,
                    showEdges,
                    this.wpPList, this.wpPNum);
            endif;
        ]
        [DEL ?item1]
        [behaviour plan ?index ?done]

    RULE switch
        [behaviour plan ?index ?done]
        [WHERE done = true]
        [behaviour ?activity]
    ==>
        [POP11
            0 -> this.timer;
        ]
        [SAY 'switching to' ?activity]
        [RESTORERULESET ?activity]
enddefine;

define :ruleset rulesetDogFind;
    [LVARS [this = sim_myself]];

    RULE switch
        [behaviour ?activity]
        [WHERE activity /= "rulesetDogFind"]
    ==>
        [RESTORERULESET rulesetDogPlan]

    RULE main
        [WHERE this.pathProblemTm = 0]
    ==>
        [LVARS x y s]
        [POP11
            lvars wp, d;

            waypointFind(
                this,
                this,
                this.obstacleLocalListDog) -> (wp, d);

            if d < dogFindDistMin then
                printf('Find path problem: distance = %P\n', [%d%]);
                this.timer -> this.pathProblemTm;
            endif;

            getCoords(wp), speedFind(this) -> (x, y, s);
        ]
```

```
        [exec ?x ?y ?s]

    RULE problem
        [WHERE this.pathProblemTm > 0]
    ==>
        [RESTORERULESET rulesetDogPlan]
enddefine;

define :ruleset rulesetDogSteer;
    [LVARS [this = sim_myself]];

    RULE switch
        [behaviour ?activity]
        [WHERE activity /= "rulesetDogSteer"]
    ==>
        [RESTORERULESET rulesetDogPlan]

    RULE main
        [WHERE this.pathProblemTm = 0]
    ==>
        [LVARS x y s]
        [POP11
            lvars wp, d, l, xS, yS;

            waypointFind(
                this,
                this.sheepCurrent,
                this.obstacleLocalListSheep) -> (wp, d);

            if d < dogSteerDistMin then
                printf('Steer path problem: distance = %P\n', [%d%]);
                this.timer -> this.pathProblemTm;
            endif;

            if d > 0 then
                dogSheepDist/d -> l;
                getCoords(wp) -> (x, y);
                getCoords(this.sheepCurrent) -> (xS, yS);
                xS + l*(xS - x), yS + l*(yS - y) -> (x, y);

                getNearest(
                    this,
                    this.obstacleLocalListDog,
                    xS, yS,
                    x, y,
                    searchDA, searchDO) -> l;
                min(dogSheepDist, max(0, l - dogObstacleDistMin))/dogSheepDist -> l;

                xS + l*(x - xS), yS + l*(y - yS), dogSpeedMax
```

```
            else
                0, 0, 0
            endif -> (x, y, s);
        ]
        [exec ?x ?y ?s]

    RULE problem
        [WHERE this.pathProblemTm > 0]
    ==>
        [RESTORERULESET rulesetDogPlan]
enddefine;

define :rulefamily rulefamilyDogActivity;

    ruleset: rulesetDogPlan
    ruleset: rulesetDogFind
    ruleset: rulesetDogSteer
enddefine;


define :ruleset rulesetDogExec;
    [LVARS [this = sim_myself]];

    RULE exec
        [exec ?x ?y ?s][->> item1]
        [behaviour ?activity]
    ==>
        [POP11
            moveDog(this, x, y, s);

            directionShow(
                this,
                sheep_win,
                showDirection,
                getCoords(if activity = "rulesetDogFind" then this else this.sheepCurrent endif),
                getCoords((this.wpPList)(this.wpIdx)));
        ]
        [DEL ?item1]

    RULE default
    ==>
        [POP11
            0 -> this.speed;
        ]
enddefine;


/*
-- Rulesystem for dog
```

```
*/

define :rulesystem rulesystemDog;
    [DLOCAL [prb_allrules = false]];

    debug = false;
    cycle_limit = 1;

    include: rulesetDogPerception
    include: rulesetDogBehaviour
    include: rulefamilyDogActivity
    include: rulesetDogExec
enddefine;
```

Appendix C

Mini-project declaration

MSc in Advanced Computer Science

First semester mini-project

This form is to be used to declare your choice of mini-project in the first semester of the course. Please complete this form, obtain the signature of your supervisor and post in the appropriate assessed work pigeon hole.

Deadline: 16.00 hrs, 23rd October 2003

Name:	Marek Kopicki
Student number:	564641

Mini-project title:	The ways to improve intelligence of interacting agents
Mini-project supervisor:	Prof. Aaron Sloman

The following questions should be answered in conjunction with a reading of the course book.

Aim of mini-project	To write a refined version of "sheepdog" simulation demo program with improved and more smart agents.

Objectives to be achieved	1. Revise and deepen knowledge of AI areas like knowledge representation, reasoning, planning and communication between agents. 2. Learn Pop-11 language. 3. Deepen knowledge of evolutionary and neural computation areas.

Project management skills Briefly explain how you will devise a management plan to allow your supervisor to evaluate your progress	1. Work out the original version of the "sheepdog" demo. 2. Propose an general framework of a new and improved version of a program. 3. Gradually write the new program simultaneously introducing new features. 4. Demonstrate final version of the new program. 5. During the semester discuss key issues and emerging problems as they come.

Systematic literature skills Briefly explain how you will find previous relevant work	My main source of knowledge will be books, journals and also Internet. I will use keywords like Pop-11, prolog, artificial intelligence, predicate calculus, interacting agents etc. and suggestion from supervisor.

Communication skills What communication skills will you practise during this mini-project?	I will practise all types of communication skills, especially writing one.

Signed (student) *M. Kopicki*

Date: 20.10.2003

Signed (supervisor) *A. Sloman*

Date: 20.10.2003

Appendix D

Statement of information search strategy

1 Parameters of literature search

Essentially, the literature search was carried out in two separate steps. Working on the program I was searching for Pop-11 language and libraries manuals, various path planning techniques and graph searching algorithms. During writing the mini-project report, however, I focused mainly on more general architectural problems of agent design. My supervisor Prof. Aaron Sloman also provided me with some papers.

1.1 Forms of literature

The categories explored maximally were (in order of usage, not relevance): www pages, a range of manuals, journal articles, technical reports, conference papers, books, theses books.

1.2 Geographical and language coverage

A vast majority of work comes from the North America and Europe and is written in English. Beside my native Polish language I am able to read papers written in French and Russian.

1.3 Age-Range of literature

The search for probabilistic roadmap planning (the key algorithm of the simulation) was carried out mostly from 1995 onwards.

2 Appropriate search tools

2.1 Engineering Index

The Compendex http://www.engineeringvillage.org is a huge interdisciplinary engineering database with 7.5 million records referencing 5,000 engineering journals and conference materials. Abstracts are available for most publications.

2.2 Science Citation Index (SCI)

I used this for finding journal articles and technical reports. Cited reference search was also performed using SCI. In SCI keywords can be used to find some papers to start with, further cited reference search can be used. The CiteSeer http://citeseer.nj.nec.com/cs is the largest and free public citation index of scientific literature. Additionally, the ISI Web of Knowledge Service http://wos.mimas.ac.uk can be used as a support.

2.3 Dissertations Abstracts International (DAI) and Index to Theses

DAI http://wwwlib.umi.com/dissertations/gateway can be used to retrieve North American theses. The Index to Theses http://www.theses.com can be used to retrieve theses accepted for higher degrees by universities in Great Britain and Ireland.

3 Search statements

The search statements were based on following groups of keywords: multi-agent simulation, path planning, motion planning, skeletonisation, probabilistic roadmap, graph search, sim_agent toolkit, etc. Additionally, the above keywords were combined together with AND and OR boolean operators, and also with usage of wildcard characters (depending on a search engine).

4 Brief evaluation of the search

Working on the program, using various keywords for searching in Science Citation Index and Google, I retrieved the following items (including subsequent cited reference search).

- Dozens of web pages
- 15 manuals
- 12 journal articles
- 5 technical reports
- 2 conference papers

During writing the mini-project report I finally decided to attach the following relevant sources.

- 4 journal articles
- 4 manuals
- 2 technical reports
- 1 conference papers

Bibliography

[1] R. A. Brooks. A robust layered control system for a mobile robot. *IEEE Journal of Robotics and Automation*, RA-2:14–23, 1986.

[2] R. A. Brooks. Intelligence without representation. *Artificial Intelligence*, 47:139–159, 1991.

[3] R. A. Brooks. From earwigs to humans. *Robotics and Autonomous Systems*, 20:291–304, 1997.

[4] L. Kavraki, P. Svestka, J.-C. Latombe, and M. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. Technical Report STAN-CS-TR-94-1519, Department of Computer Science, Stanford University, 1994.

[5] D. Kirsh. Today the earwig, tomorrow man? *Artificial Intelligence*, 47:161–184, 1991.

[6] N. J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann, 1998.

[7] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, second edition, 2003.

[8] A. Sloman. *RULESYSTEMS help file*. School of Computer Science, the University of Birmingham, http://www.cs.bham.ac.uk/research/poplog/newkit/prb/help/rulesystems, Jun 1996.

[9] A. Sloman. *SIM_AGENT help file*. School of Computer Science, the University of Birmingham, http://www.cs.bham.ac.uk/research/poplog/sim/help/sim_agent, Jun 1996.

[10] A. Sloman. *POPRULEBASE help file*. School of Computer Science, the University of Birmingham, http://www.cs.bham.ac.uk/research/poplog/newkit/prb/help/poprulebase, Aug 2000.

[11] A. Sloman and R. Poli. Sim_agent: A toolkit for exploring agent designs. In M. Wooldridge, J. Mueller, and M. Tambe, editors, *Intelligent Agents Vol II (ATAL-95)*, pages 392–407. Springer-Verlag, 1996.

[12] S. M. L. Valle. Rapidly-exploring random trees: A new tool for path planning. Technical Report TR 98-11, Computer Science Dept., Iowa State University, 1998.

[13] P. Waudby and T. Carter. *TEACH SIM_SHEEPDOG.P*. School of Computer Science, the University of Birmingham, http://www.cs.bham.ac.uk/research/poplog/newkit/sim/teach/sim_sheepdog.p, Sep 1999.