ECE478 Final Project

Simulator for Robot Navigation

By

Tristan Shores

Introduction

One of the main challenges in robot navigation in an unknown environment is the formulation of an effective algorithm for obstacle avoidance. Development of an effective algorithm requires repetitive and time-consuming testing in the relevant environment with the likelihood of hardware glitches skewing the test results. To avoid such problems, at least during the initial stages of algorithm development, a software simulator can be used. A simulator is also a useful tool for students seeking to explore algorithm development without having to spend weeks wrestling with hardware availability/development/reliability issues.

This paper describes a software simulator that was successfully developed by the present author to allow an end-user to:

1. Create and test custom fuzzy rules.
2. Auto-generate fuzzy rules for wall-following using a genetic algorithm.
3. Explore targeted navigation using sensor proximity data and the A* algorithm.

Each of these purposes is described in a separate section, following the simulator specification section below.

General Simulator Specifications

1. Any bitmap image with white background and non-white obstacles can be loaded as a robot maze. An example of a loaded bitmap for the first floor of the Maseeh building is shown in Figure 1.

FIGURE 1: Maseeh building floor plan.

2. The simulator overlays the user-selected maze bitmap with a grid of user-configurable density. The grid density determines the granularity of obstacle mapping and the size of the search space (i.e. number of grid cells). However, the robot moves at the granularity of pixels not grid cells, and therefore can exhibit smooth motion regardless of grid size. This is reflective of real robot motion, which is not necessarily tied to the logical mapping of search space.
3. Robot start/end points can be set by a mouse-click at any point on the bitmap image.
4. Robot speed is user adjustable; however, the time required for CPU processing of the navigational algorithm and for screen refresh limits the maximum speed of the robot.
5. The robot size (radius) is configurable by the end-user.

6. Any number of proximity sensors can be added to the robot. Each proximity sensor has a user configurable direction, beam spread, and range (as shown in Figure 2). Upon contact of a sensor beam with a non-white bitmap pixel, the grid cell overlaying the bitmap obstacle is painted black to denote a known obstacle.

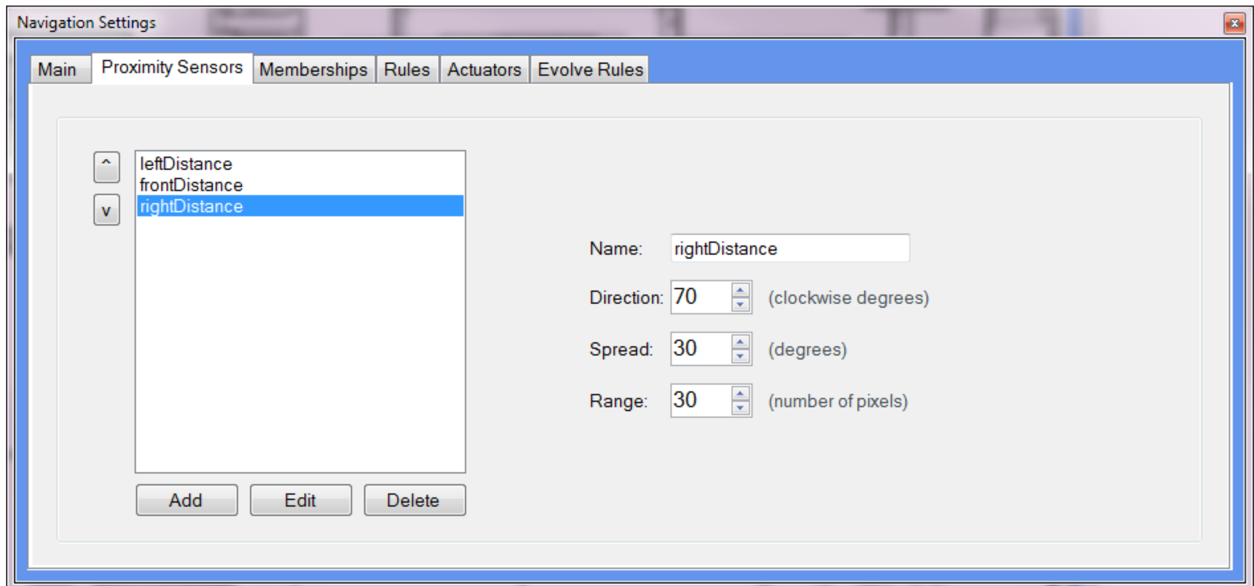


FIGURE 2: Proximity sensor configuration page.

7. All configured sensors, memberships, rules, and actuator assignments are saved to disk in between user sessions.

User Development of Custom Fuzzy Rules

The paper titled "Application of Fuzzy Logic for Robot Navigation" (see Appendix A) by the present author provides a discussion of the basic concepts of fuzzy logic, which are used in the simulator.

Any number of input/output membership functions can be dynamically created by the user. A plot area facilitates creation of user drawn plots (curved lines are smoothed using Bezier curves) with configurable crisp-axis limits, as shown in Figure 3.

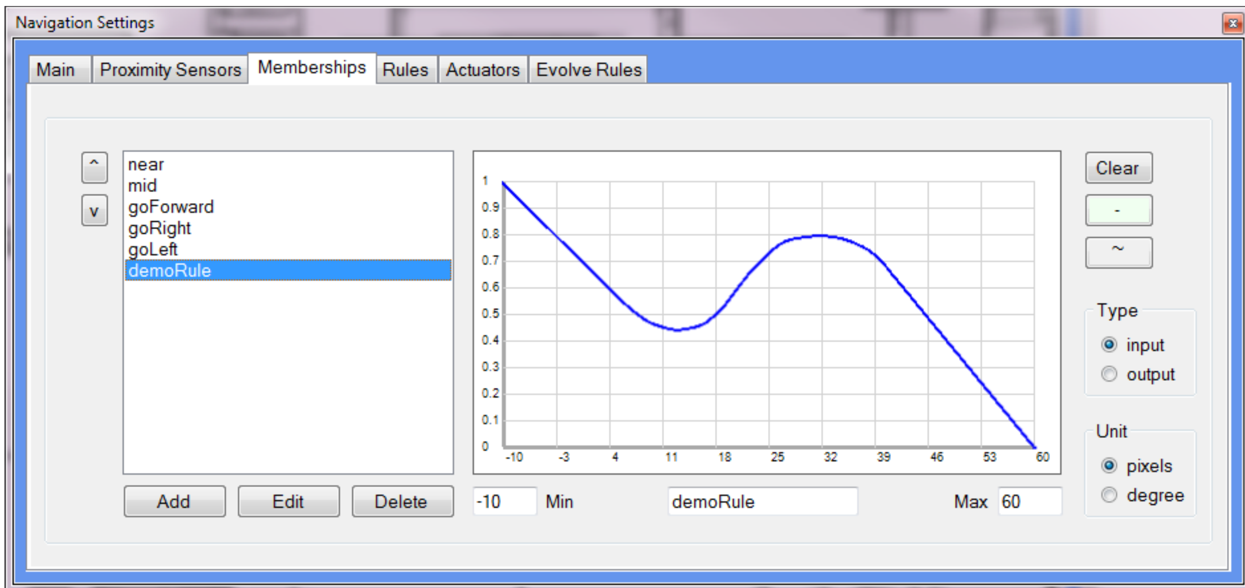


FIGURE 3: Membership function configuration page.

Rules can be dynamically created by the user. Each rule created by the user must correspond by name to one of the output membership functions. Since any number of output membership functions can be created by a user, any number of rules is possible. A valid rule statement includes operands that are either:

1. Any input membership function name paired with any proximity sensor name. The evaluation of this operation results in a decimal value between 0 and 1 inclusive (μ), which represents the degree of membership of the sensor's proximity reading in the set represented by the input membership function's name.
2. A decimal value between zero and one inclusive.

The built-in fuzzy parser will iteratively parse nested logic statements of any depth. Generic logic operators (AND, OR, NOT) and custom operators (GREATER-THAN, LESS-THAN) can be used in conjunction with the membership functions, sensor input readings, and parentheses for nesting levels, as shown in Figure 4.

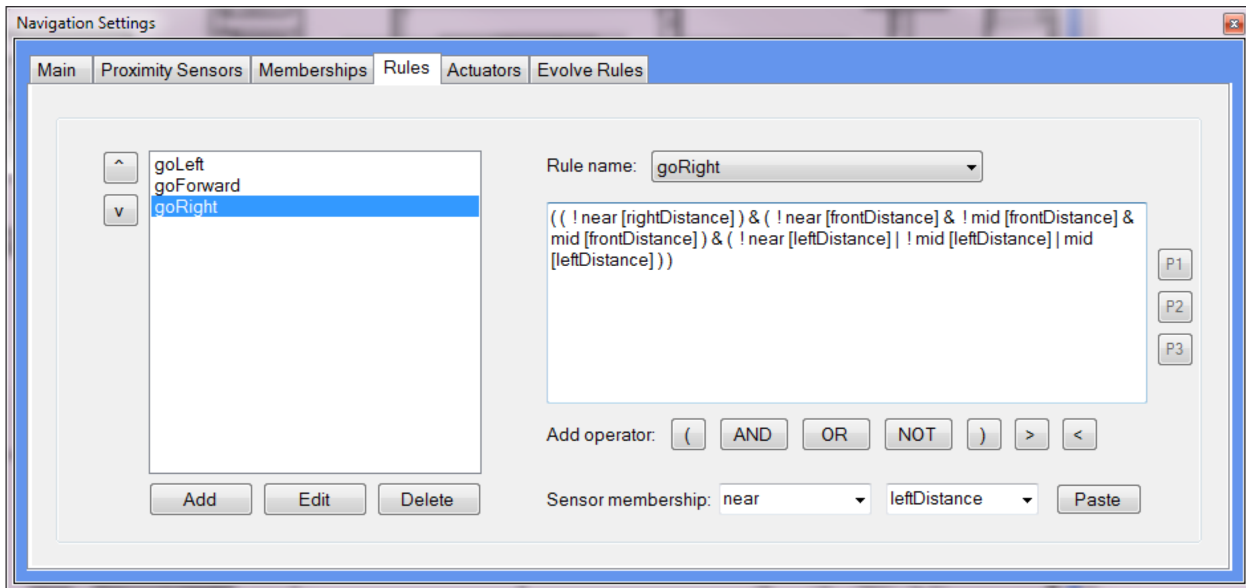


FIGURE 4: Rule configuration page.

Rule statement evaluation consists of the following steps:

1. Each operand is reduced to a decimal value between zero and one inclusive.
2. Fuzzy logic operators are applied to the operands in a deepest-first nesting order as indicated by parentheses placement. The application of logic operators to operands reduces the rule statement to a single decimal value between zero and one inclusive ($\mu_{predicate}$).
3. The crisp value for the $\mu_{predicate}$ is determined using the output membership function that corresponds to the rule's name. This crisp value is the evaluation of the rule.

Two built-in actuators provide complete 2D navigational control of the robot: robotForward and robotRotate. The crisp evaluations of all rules assigned to an actuator are summed to provide the actuator's resultant action. For example, if the robotRotate actuator evaluates to -23, the robot will rotate counter-clockwise by -23°.

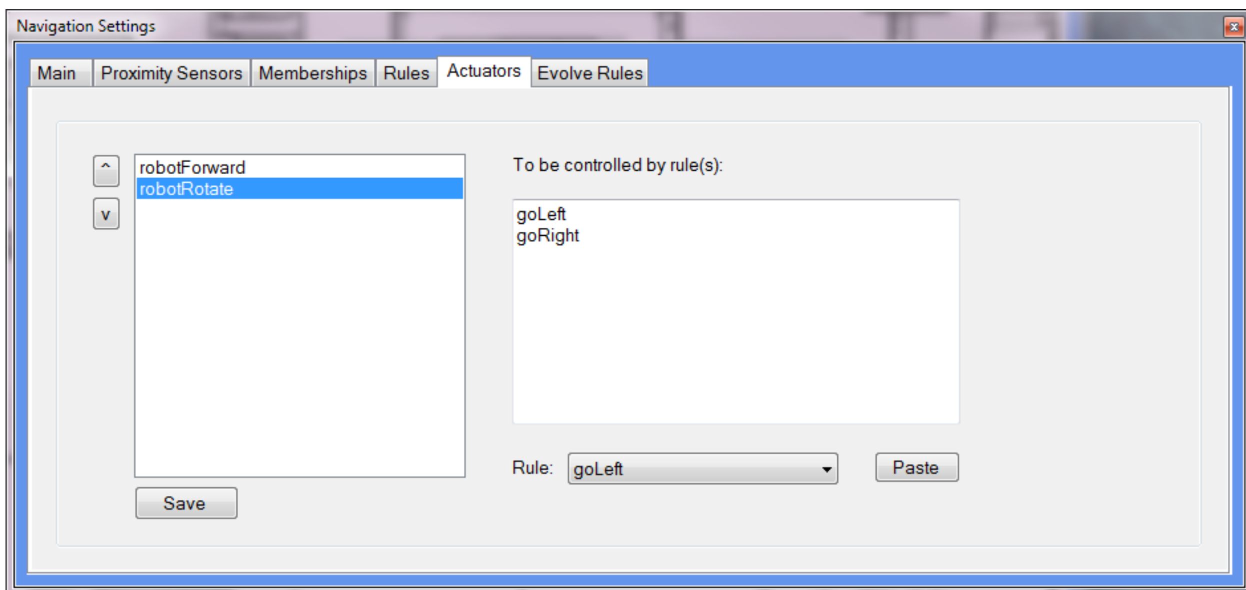


FIGURE 5: Actuator configuration page.

An example of simple wall-following behavior is shown in Figure 6. The following rules were used:

goLeft = near [leftDistance] < 0.6

goRight = near [leftDistance] > 0.7

goForward = ! near [frontDistance]

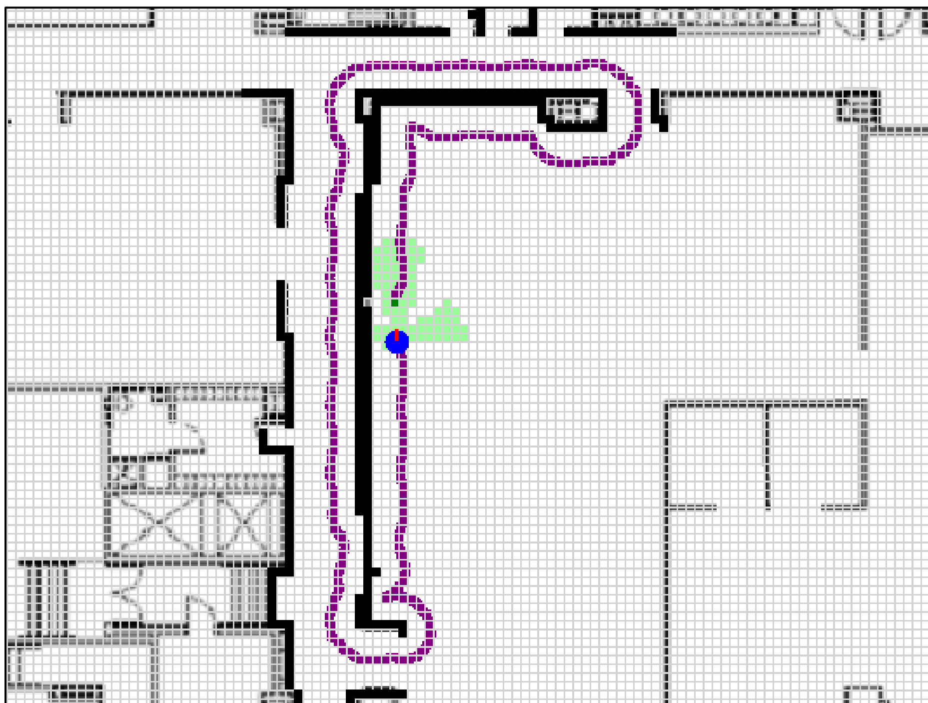


FIGURE 6: Robot navigation within simulated Maseeh building using simple wall-following rules.

Generation of Fuzzy Rules Using a Genetic Algorithm

The paper titled “Evolution of Simulated Millipede Gait” (see Appendix B) by the present author provides a discussion of a double mutation genetic algorithm, which is used in the simulator.

A built-in genetic algorithm permits evolution of fuzzy rules for wall-following behavior. The genetic pool size, number of generations, and number of robot steps performed per fitness function test are user configurable, as shown in Figure 7.

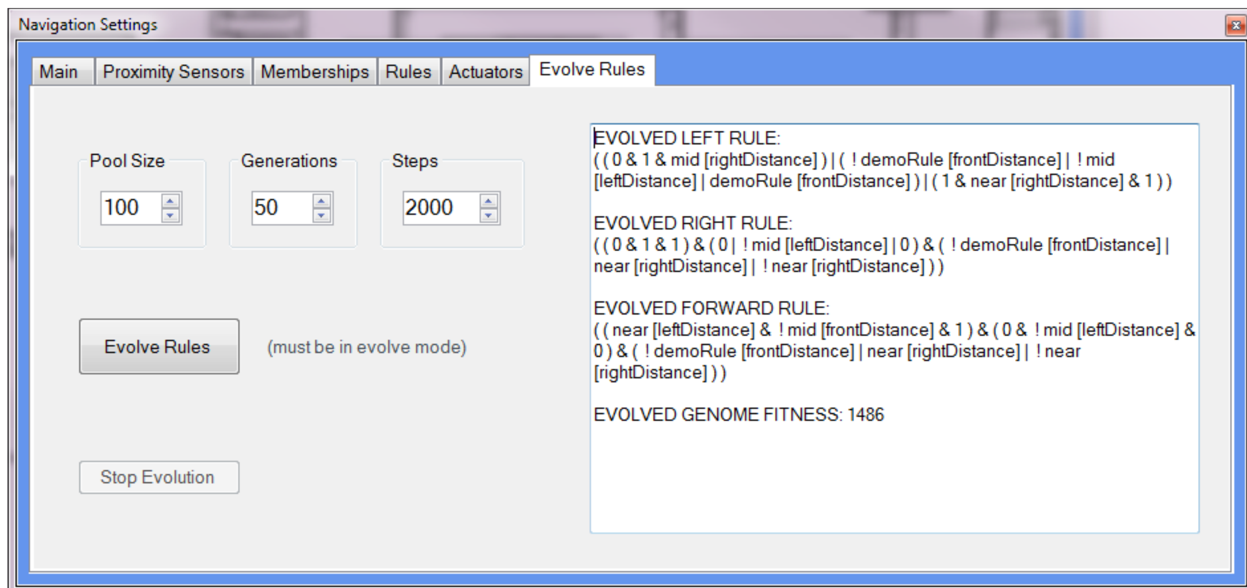


FIGURE 7: Evolution configuration page.

Double mutation was used to evolve each of the following rules: goLeft, goRight, goForward. Mutation was selected because it can mutate specific elements with a rule without invalidating rule syntax. Double mutation was selected because of its better performance over single mutation. Two random elements within a rule were selected and if those elements corresponded to a membership/sensor pair, then one of the following elements was substituted in its place:

1. A random generated membership/sensor pair
2. 0
3. 1

However, if the randomly selected elements contained an operator or parentheses, then the random location was incremented until a membership/sensor pair element was encountered. The double mutation code is shown in Figure 8.

```
private static void PerformDoubleMutation(Rule rule)
{
    //decompose rule:
    SeparateRuleString (rule);
    int[] idx = new int[2];

    //select 2 random index locations:
    idx[0] = Utility.GetRandomNumberInclusive (rule.elementList.Count - 1);
    idx[1] = Utility.GetRandomNumberInclusive (rule.elementList.Count - 1);

    //at each random index location:
    for (int i = 0; i < idx.Length; i++)
    {
        //shift index if it lands on non-mutable characters (i.e. operators/parentheses):
        while (
            rule.elementList[idx[i]] == "(" ||
            rule.elementList[idx[i]] == ")" ||
            rule.elementList[idx[i]] == "&" ||
            rule.elementList[idx[i]] == "|")
            idx[i] = ++idx[i] % (rule.elementList.Count - 1);
        //get random index:
        int randomIndex = Utility.GetRandomNumberInclusive (GeneticToolBox.operandArray.Length - 1);
        //select a random operand:
        string randomOperand = GeneticToolBox.operandArray[randomIndex];
        //replace old operand with new random operand:
        if (rule.elementList[idx[i]].Contains("[")) rule.elementList[idx[i]] = randomOperand;
    }

    //recompile rule:
    StringBuilder sb = new StringBuilder();
    foreach (string s in rule.elementList) sb.Append(s);
    rule.ruleString = sb.ToString ();
}
```

FIGURE 8: Double mutation code.

The best-evolved rule set, shown in Figure 7, simplifies to the following:

1. goLeft = ((near [rightDistance] | mid [rightDistance]) & (! near [frontDistance] & ! mid [frontDistance]))
2. goRight = ((! mid [rightDistance] | ! near [rightDistance] | near [rightDistance]) | (! mid [frontDistance] & ! near [frontDistance] & near [frontDistance]) | (near [frontDistance] & ! mid [leftDistance]))
3. goForward = 1

Each rule can be interpreted as follows:

1. Left rule: turn left if midway-close to a right-side obstacle, but only if not too-close to a front obstacle.
2. Right rule: turn hard-right whenever too-close or too-far from a right-side obstacle, or too-near or too-far from a front obstacle.
3. Forward rule: always go forward

The combination of those rules can be interpreted as follows: If midway-close to a right-side obstacle and not too-close to a front obstacle then turn left sharply. In all other situations, <u>but not that one</u>, turn right very sharply. This explains the right-wall-following behavior because the right rule dominates in all but a midway band of proximity from the right wall. This also explains the single loop of the robot when it nears an inside corner because the left rule weakens as a front obstacle grows closer, and the right rule drives the robot sharply into the corner. However, the radius of the right turn is so sharp that the robot does not collide with the wall corner, but instead loops out of the corner, subsequently connecting with the perpendicular wall away from the corner and further continuing its right-wall-following behavior. The robot's path for this evolved rule set is shown in Figure 9.

The logistics of this evolved rule set are remarkably sophisticated and non-obvious to novice fuzzy rule developers. This highlights the power of genetic algorithms to generate a sophisticated rule set for wall-following in a fraction of the time it would take to manually conceive it.

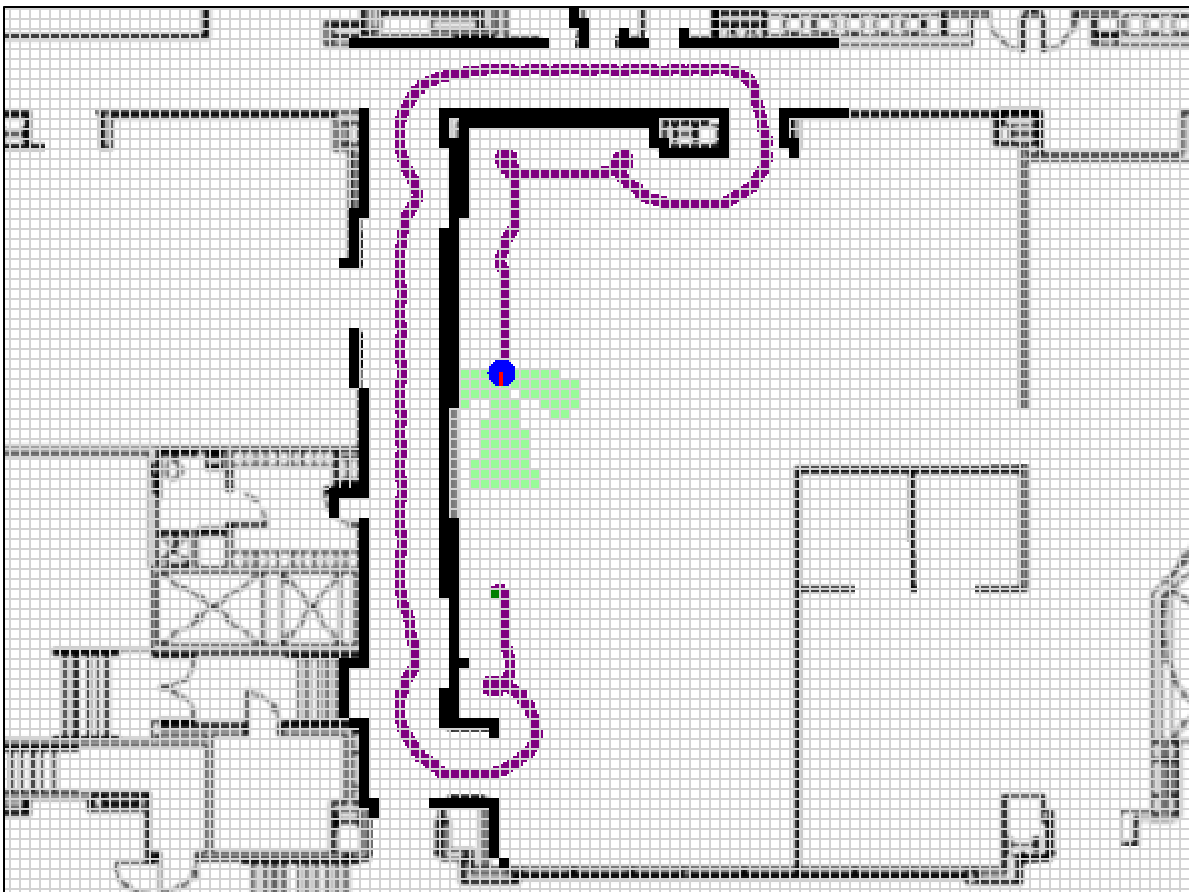


FIGURE 9: Robot wall-following navigation using best-evolved rule set.

Targeted Navigation Using Sensor Proximity Data and A* Algorithm

The paper titled "Shortest Path through Maze" (see Appendix C) by the present author provides an extensive discussion of the A* search algorithm, which is used in the simulator.

Targeted navigation features a built-in navigation strategy, using the A* algorithm, which demonstrates successful obstacle avoidance, backtracking, and shortest distance approximation to a specified end point in the maze.

Prior to each step through the maze, the robot examines the sensor data to determine whether an obstacle is detected. If so, it is added to a known-obstacle list (and colored black in the simulator). Prior to discovering an obstacle via its sensors, the robot is completely unaware of the existence of the obstacle. This is illustrated in Figure 10, where the robot's shortest path projection based on the A* algorithm (shown in orange) bends away from known obstacles, but cuts straight through unknown obstacles. Note that the purple path indicates the robot's previous path, the pale blue areas indicate the closed-list search nodes, and the medium blue band marks the open-list search nodes.

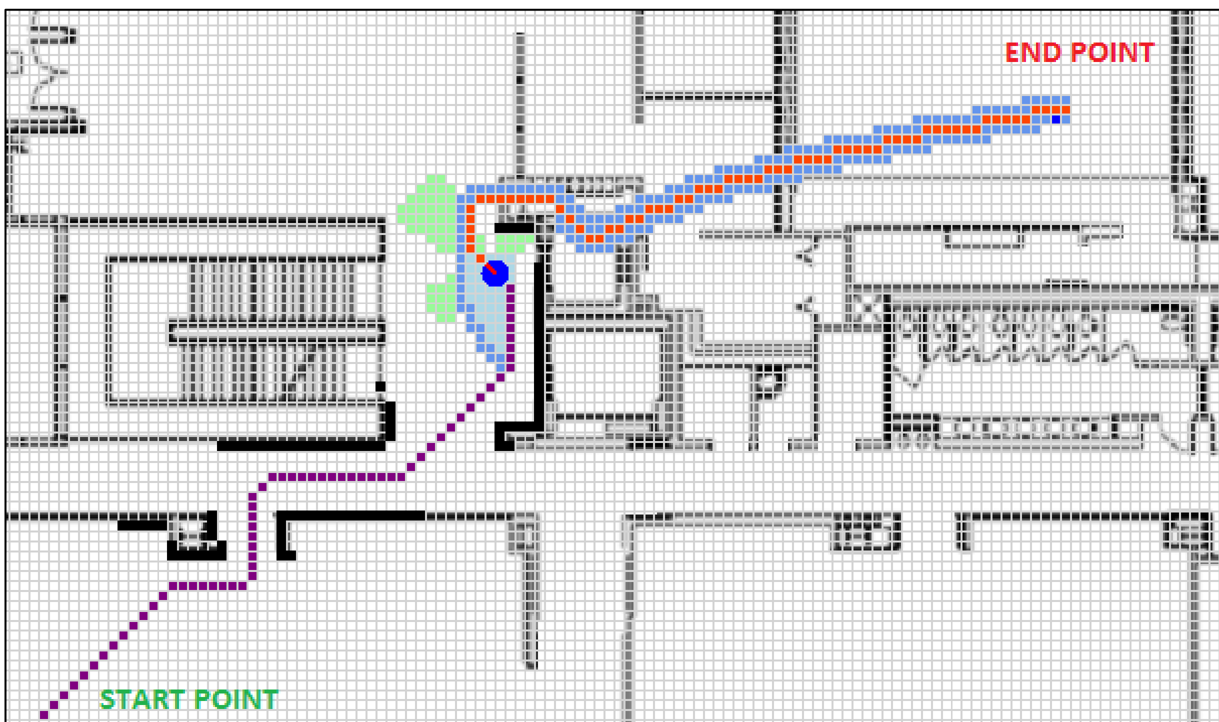


FIGURE 10: An example of targeted navigation using the A* algorithm.

The robot takes its next step along the path routed by the A* algorithm. If a new obstacle is detected by any sensor because of that step, the robot will halt and rerun the A*algorithm from its new position with its updated

known-obstacle-list. Thus, the robot advances rapidly through open space, but has to slow down in obstacle-rich environments in order to rerun the A* algorithm each time the known-obstacle-list changes.

Robot path backtracking is shown in Figure 11. Note that the backtracking behavior exhibited by the dead-end purple paths is NOT backtracking using saved open-list nodes, because the open-list is cleared between each run of the A* algorithm from a different start position. This is a necessity because the open-list is only valid for a given set of known obstacles. However, true backtracking using the open-list is implemented within each run of the A* algorithm in order to generate the next shortest path approximation to the end point.

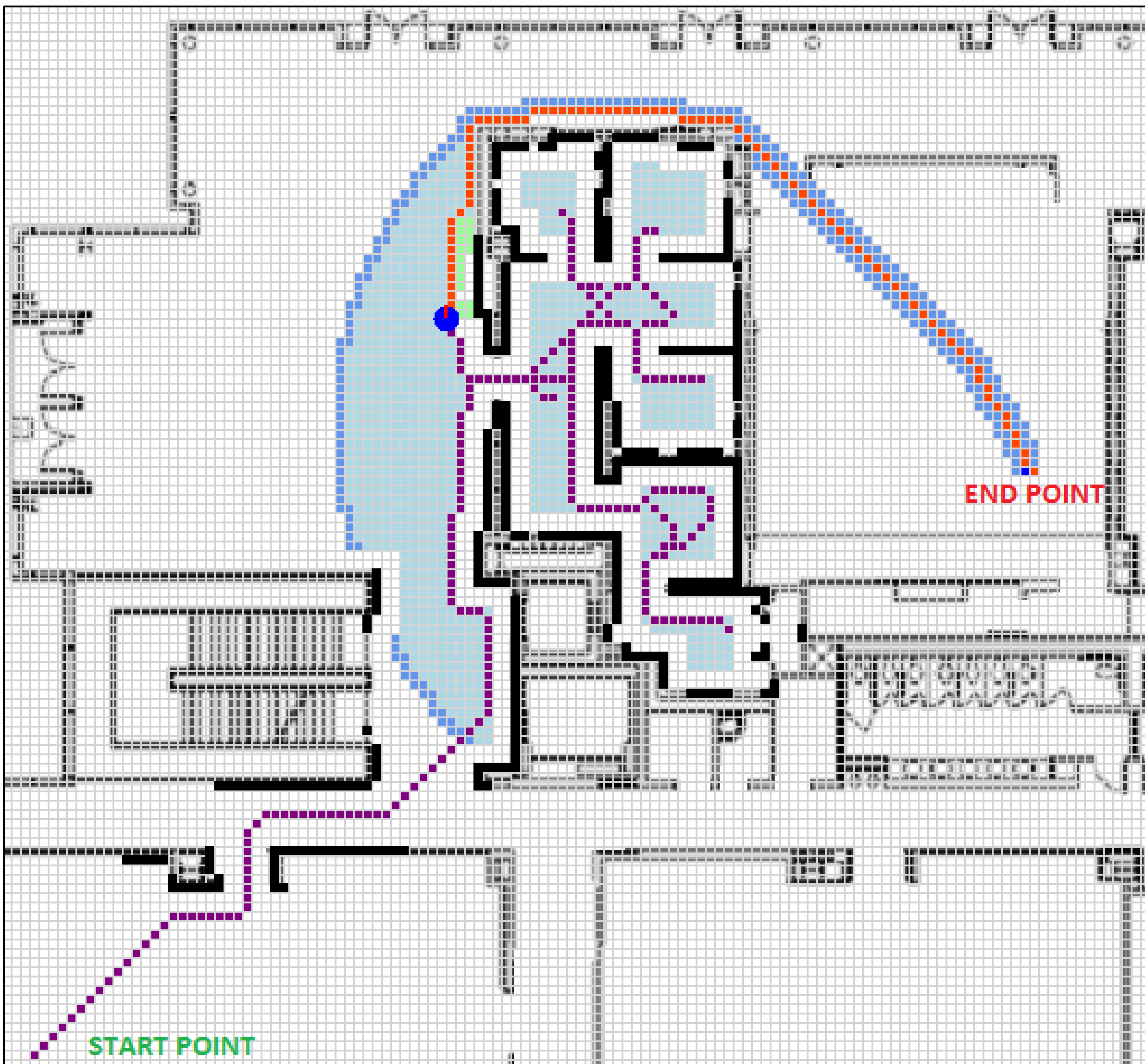


FIGURE 11: Robot path backtracking.

Conclusion

The simulator for robot navigation successfully demonstrates the applicability of fuzzy logic, genetic algorithms, and search strategies to robot navigation in an unknown environment.

The simulator would be significantly enhanced through incorporation of Kalman filters, neural nets, and image processing through directionally-sensitive embedded images in the maze (i.e. if the robot faces a doorway at a specified grid location with a specified orientation, then a pre-specified embedded image will be fed into the robot's camera for image-analysis).

The simulator is a professionally written robust C# application compatible with any Windows environment (native, virtual, 32-bit, or 64-bit). For release licenses, the author may be contacted at tristan@electronic.io.

APPENDIX A

APPLICATION OF FUZZY LOGIC FOR ROBOT NAVIGATION

Introduction

Boolean logic applied to robotic navigation permits only two potential outcomes for each robot decision. In contrast, fuzzy logic allows unlimited outcomes for each robot decision.

For example, a fuzzy logic rule might state: "**turn-right** if **front-proximity** is **close** and **left-proximity** is **close** and **right-proximity** is **far**", where:

- **front-proximity**, **left-proximity**, and **right-proximity** are sensor readings.
- **close** and **far** are input membership functions that flexibly translate a specific proximity (crisp value) in a defined sensor reading range to a decimal value in the range of 0 to 1 (degree of membership).
- **turn-right** is an output membership function that flexibly translates a decimal value in the range of 0 to 1 (degree of membership) into a clockwise rotation in the range of 0 - 90° (crisp value).
- AND is a fuzzy logic operator that typically asserts the minimum of the operands.

If the evaluation of all conditions is satisfied to any extent, then the robot will execute a clockwise turn in the range of 0 to 90°, with the exact rotation dependent on the degree to which conditions are satisfied. In this way, a single fuzzy logic rule can generate smooth robot navigation.

An example of a fuzzy logic system used to solve a robot navigational decision (degree of robot rotation) is given in Fig. 1.

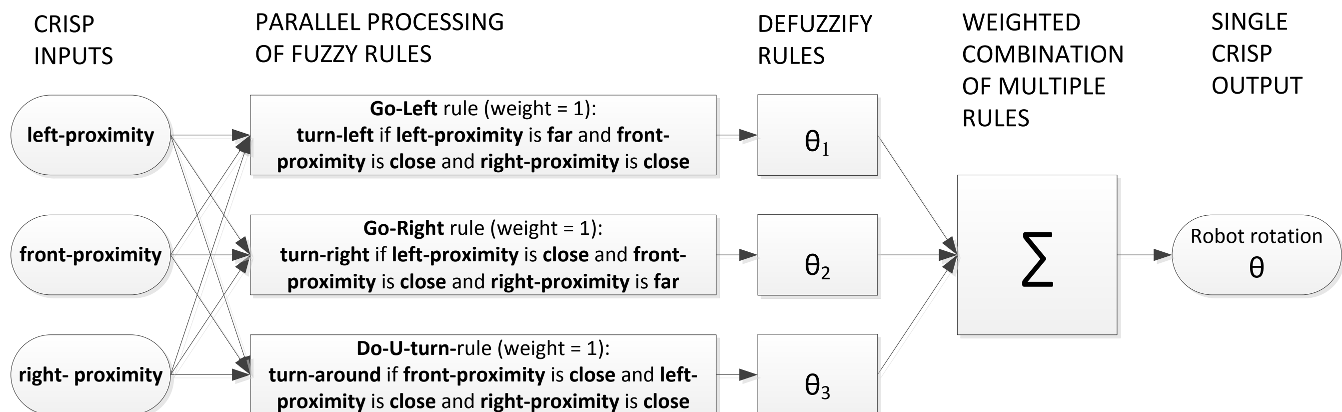


Figure 1. Triple input, triple rule, single output fuzzy system.

Crisp Inputs

A crisp value is a real non-scaled value, such as a robot's unobstructed front distance in feet. The system shown in Fig. 1 has three crisp inputs that provide the unobstructed distance on a robot's left, front, and right sides.

Parallel Processing of Fuzzy Rules

A fuzzy rule consists of one or more conditions that when combined comprise the rule's predicate which determines the rule's consequence. For example, the predicate of the rule **Go-Right** is "if **front-proximity** is **close** and **left-proximity** is **close** and **right-proximity** is **far**", and the consequence of the rule is **turn-right**. Each condition independently evaluates to a degree of membership μ in a set specified by that condition. For example, "**left-proximity** is **far**" evaluates to **left-proximity**'s degree of membership μ_{far} in the set **far**.

μ is in the range of 0 to 1, where a μ of 0 indicates that a condition was not satisfied to any extent, a μ of 1 indicates full satisfaction of a condition, and a fractional μ indicates the partial degree to which a condition was satisfied.

In the example shown in Fig. 1, each rule has three conditions that individually need to be evaluated. For example, the **Go-Left** rule requires evaluation of "**left-proximity** is **far**", "**front-proximity** is **close**", and "**right-proximity** is **far**". The terms **close** and **far** represent fuzzy sets to which **left-proximity**, **front-proximity**, and **right-proximity** may belong to some degree. The exact extent to which **left-proximity**, **front-proximity**, and **right-proximity** belong to the **close** and **far** sets is evaluated using the input membership function for each set.

The set **close** interprets an input parameter (representing proximity) according to the input membership function shown in Figure 2. For example, for a proximity of 8ft, the function outputs a μ_{close} value of 0.2. This indicates that this proximity value is a marginal member of the set.

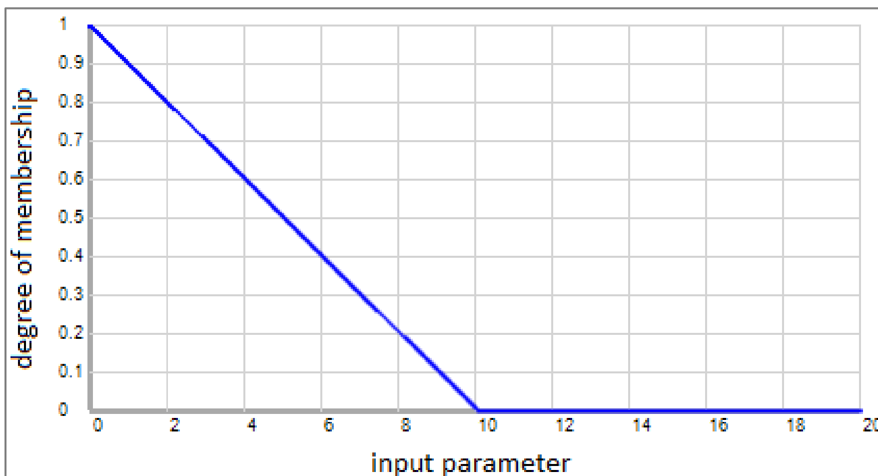


Figure 2. Input membership function for the **close** set.

The set **far** interprets an input parameter (representing proximity) according to the input membership function shown in Figure 3. For example, for a proximity of 8ft, the function outputs a μ_{far} value of 0.8. This indicates that this proximity value is a strong member of the set.

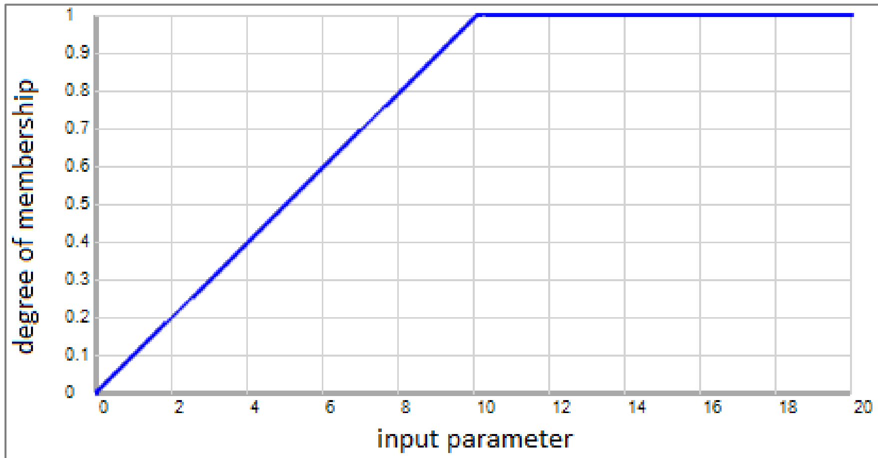


Figure 3. Input membership function for the **far** set.

It is significant that proximity values in the non-inclusive range 0 to 10ft have partial membership in both the **close** and **far** fuzzy sets, as will be discussed later.

Fuzzy logic operators determine the way in which multiple conditions will combine to yield an overall degree of membership for the rule's predicate. This degree of membership applies to the set represented by the rule's consequence. For example, the predicate of the **Go-Right** rule evaluates to a degree of membership $\mu_{predicate}$ in the **turn-right** set.

Common fuzzy logic operators are AND, OR and, NOT, where:

- AND typically asserts the minimum of the operands.
- OR typically asserts the maximum of the operands.
- NOT inverts the associated operand (i.e. evaluates to 1 minus the operand value).

The use of fuzzy logic operators to determine $\mu_{predicate}$ for each of the **Go-Left**, **Go-Right**, and **Do-U-Turn** rules is shown in Figs. 4A, 4B, and 4C respectively. The three rules are processed in parallel to yield 3 separate $\mu_{predicate}$ values.

<table>
<tr><th>Crisp input names</th><th>Crisp input values</th><th>membership in close set μ_{close}</th><th>membership in far set μ_{far}</th><th>left-proximity is far AND front-proximity is close AND right-proximity is close</th><th>$\mu_{predicate}$</th></tr>
<tr><td>left-proximity</td><td>8</td><td>0.2</td><td>0.8</td><td rowspan="3">min (0.8, 0.8, 0.5)</td><td rowspan="3">0.5</td></tr>
<tr><td>front-proximity</td><td>2</td><td>0.8</td><td>0.2</td></tr>
<tr><td>right-proximity</td><td>5</td><td>0.5</td><td>0.5</td></tr>
</table>

Figure 4A. Application of the fuzzy logic AND operator to the **Go-Left** rule.

<table>
<tr><th>Crisp input names</th><th>Crisp input values</th><th>membership in close set μ_{close}</th><th>membership in far set μ_{far}</th><th>left-proximity is close AND front-proximity is close AND right-proximity is far</th><th>$\mu_{predicate}$</th></tr>
<tr><td>left-proximity</td><td>8</td><td>0.2</td><td>0.8</td><td rowspan="3">min (0.2, 0.8, 0.5)</td><td rowspan="3">0.2</td></tr>
<tr><td>front-proximity</td><td>2</td><td>0.8</td><td>0.2</td></tr>
<tr><td>right-proximity</td><td>5</td><td>0.5</td><td>0.5</td></tr>
</table>

Figure 4B. Application of the fuzzy logic AND operator to the **Go-Right** rule.

<table>
<tr><th>Crisp input names</th><th>Crisp input values</th><th>membership in close set μ_{close}</th><th>membership in far set μ_{far}</th><th>left-proximity is close AND front-proximity is close AND right-proximity is close</th><th>$\mu_{predicate}$</th></tr>
<tr><td>left-proximity</td><td>8</td><td>0.2</td><td>0.8</td><td rowspan="3">min (0.2, 0.8, 0.5)</td><td rowspan="3">0.2</td></tr>
<tr><td>front-proximity</td><td>2</td><td>0.8</td><td>0.2</td></tr>
<tr><td>right-proximity</td><td>5</td><td>0.5</td><td>0.5</td></tr>
</table>

Figure 4C. Application of the fuzzy logic AND operator to the **Do-U-Turn** rule.

Defuzzify Rules

In the example shown of Fig. 1, the **Go-Left** rule is defuzzified to yield a crisp value that is determined by the degree of membership of its $\mu_{predicate}$ in the **turn-left** set. Similarly, the **Go-Right** rule is defuzzified to yield a crisp value that is determined by the degree of membership of its $\mu_{predicate}$ in the **turn-right** set. Finally, the **Do-U-Turn** rule is defuzzified to yield a crisp value that is determined by the degree of membership of its $\mu_{predicate}$ in the **turn-around** set. The crisp value for a given $\mu_{predicate}$ is determined using the output membership function for the set to which the degree of membership applies.

The output membership function shown in Figure 5, specifies a crisp value for each degree of membership $\mu_{predicate}$ in the **turn-left** set. For example, given a $\mu_{predicate}$ of 0.5, the function outputs a crisp value of -45°. This crisp value is a middle member of the set.

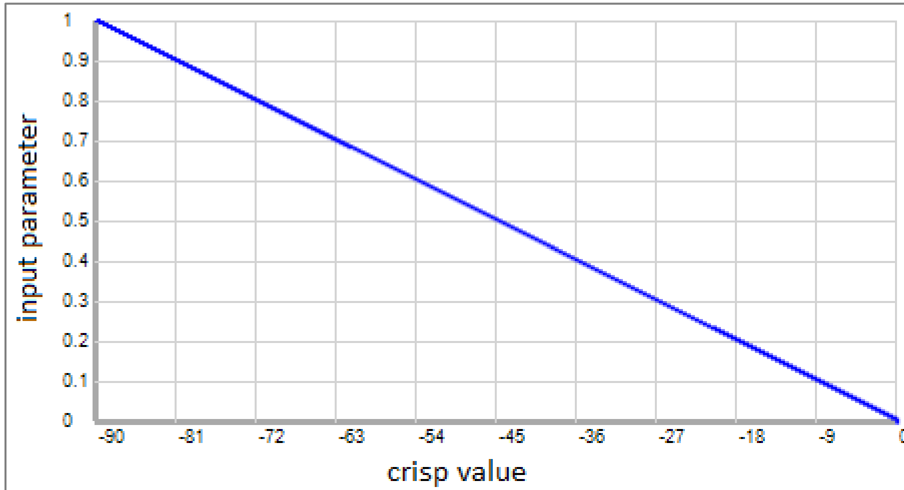


Figure 5. Output membership function for the **turn-left** set.

The output membership function shown in Figure 6, specifies a crisp value for each degree of membership $\mu_{predicate}$ in the **turn-right** set. For example, given a $\mu_{predicate}$ of 0.2, the function outputs a crisp value of +18°. This $\mu_{predicate}$ value is a marginal member of the set.

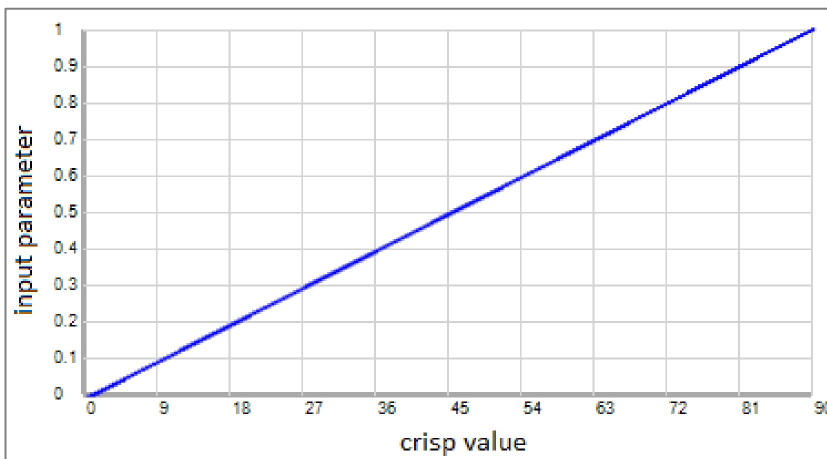


Figure 6. Output membership function for the **turn-right** set.

The output membership function shown in Figure 7, specifies a crisp value for each degree of membership $\mu_{predicate}$ in the **turn-around** set. For example, given a $\mu_{predicate}$ of 0.2, the function outputs a crisp value of 0°. This $\mu_{predicate}$ value is a non-member of the set.

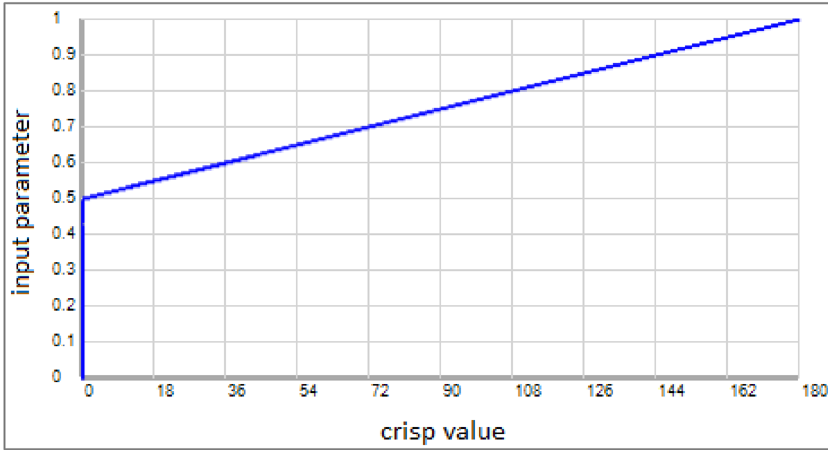


Figure 7. Output membership function for the **turn-around** set.

In the case of a single rule fuzzy system the resulting crisp value found from the output membership function would be the single crisp output of the fuzzy system. However, in the example shown of Fig. 1, the three rules are processed in parallel and need to be combined in order to generate the single crisp output of the fuzzy system.

Weighted Combination of Multiple Fuzzy Rules

In the example shown of Fig. 1, each fuzzy rule weighting is unity and therefore a multiplication factor is not applied to the crisp value of any rule.

The combination of multiple crisp values can occur in various ways. In the case of the example shown in Fig. 1, a straight forward summation is used, as shown in Figure 8.

Crisp input names	Crisp input values	Rule name	μ of the predicate	crisp value	∑	robot rotation
leftProximity	8	Go Left Rule	0.5	-45°	-27°	turn 27° to the left
frontProximity	2	Go Right Rule	0.2	+18°		
rightProximity	5	Do U-Turn Rule	0.2	0°		

Figure 8. Combination of three rules to generate a single crisp output.

Another method for combination of fuzzy rules truncates each output membership function plot at the $\mu_{predicate}$ for that plot, aggregates the truncated plot areas, and then calculates the centroid of the aggregated area. The crisp value of the centroid (x-coordinate) is then used as the single crisp output.

Discussion

The parallel processing of rules potentially allows multiple rules to influence the output crisp value, depending on each rule's crisp evaluation and weighting. This helps avoid sharp switching between opposing rules which could result in non-smooth crisp output transitions.

However, to harness the benefits of parallel rule processing, the fuzzy system designer must ensure overlapping transition domains within the membership functions of diametrically opposed sets. For example, proximity values in the non-inclusive range 0 to 10ft have partial membership in both the **close** and **far** sets. This is shown in Fig. 9, which is a plot of superimposed input membership functions for the **close** and **far** sets.

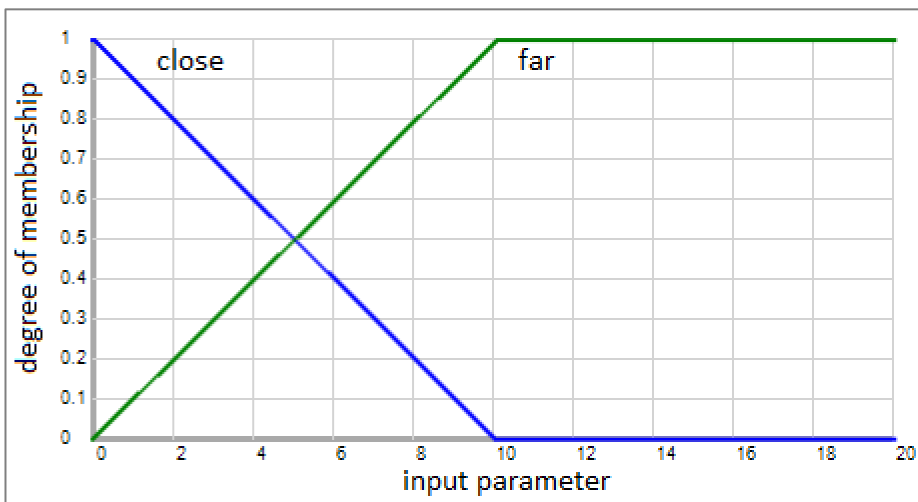


Figure 9. Overlapping transition domains for the input membership functions for the **close** and **far** sets.

Note that the input membership function for the **far** set is equivalent to the input membership function of a NOT **close** set. Thus, the rule "**turn-right** if **front-proximity** is **close** and **left-proximity** is **close** and **right-proximity** is **far**" could equivalently have been phrased "**turn-right** if **front-proximity** is **close** and **left-proximity** is **close** and **right-proximity** is NOT **close**".

Conclusion

The application of fuzzy logic to robot navigation permits vastly more complex and fine grained robot actuation than could normally be achieved through the use of Boolean logic. The challenge with fuzzy logic is to derive a meaningful set of rules that meets the intended goal of robot navigation in a particular environment. Rule generation is sufficiently complex and time consuming to warrant development via a neural network or genetic algorithm. Ideally, rule generation would occur in a simulated environment, where time and mechanical issues are less likely to impede rule development.

APPENDIX B

EVOLUTION OF SIMULATED MILLIPEDE GAIT

Introduction

The relative performance of crossover, mutation, double crossover, and double mutation genetic algorithms in generating an optimal cyclic forward gait for a simulated millipede was explored. A graphical software program written in C# simulated the motion of a millipede of varying physical characteristics (e.g. spine length, leg number, leg size, leg spacing). Matlab was used to create 3D plots that represent the performance of the tested genetic algorithms.

Millipede Leg Position Genome

A snapshot of a millipede's legs placement provides an angle (θ) for each leg. A 0° leg angle describes a leg perpendicular to the spine. A rotational limit of $\pm$ 20° was selected to avoid crossed legs. Forward leg rotation is represented by a positive angle. Each leg angle was expressed as a 5-bit binary string, termed a leg position gene. The concatenation of all leg position genes in fixed leg order results in a binary string, termed a genome. For example, if a millipede has 6 legs, a 30-bit binary string captures the millipede's leg positions. The initial/reset leg-state of a millipede is a 30-bit binary string of zeros (all legs are perpendicular to the spine i.e. straight out).

Millipede Motion

Millipede motion occurs by cyclic execution of the following two steps:

1. Rotational movements of each leg of the millipede (i.e. represented by a genome). The spine does not move, and is grounded.
2. A body motion (forward/backward/rotation), which resets each leg to perpendicular.

Step 1 represents the millipede reaching with its legs, and step 2 represents the resulting motion of the millipede’s body as it straightens out its legs.

The transformation of the legs rotations into various millipede body motions for a millipede with N pairs of legs of length L is determined as follows:

The forward motion of each leg (δf) is given by: $Lsin\theta$

Linear motion of left side of millipede (ΔF_{left}) is given by: $\frac{\sum_1^N \delta f_{left}}{N}$

Forward motion of right side of millipede (ΔF_{right}) is given by: $\frac{\sum_{1}^{N} \delta f_{right}}{N}$

Forward motion of the millipede occurs if: $\Delta F_{left} > 0 \; and \; \Delta F_{right} > 0$

Backward motion of the millipede occurs if: $\Delta F_{left} < 0 \; and \; \Delta F_{right} < 0$

Clockwise motion of the millipede occurs if: $\Delta F_{left} > 0 \; and \; \Delta F_{right} < 0$

Counter-clockwise motion of the millipede occurs if: $\Delta F_{left} < 0 \; and \; \Delta F_{right} > 0$

If the conditions for backward, forward, clockwise, or counter-clockwise motion are satisfied:

The magnitude of forward motion is given by: $\text{Minimum}(\Delta F_{left}, \Delta F_{right})$

The magnitude of backward motion is given by: $\text{Maximum}(\Delta F_{left}, \Delta F_{right})$

The magnitude of clockwise motion is given by: $\Delta F_{left} - \Delta F_{right}$

The magnitude of counter-clockwise motion is given by: $\Delta F_{right} - \Delta F_{left}$

Millipede Fitness

Millipede fitness was arbitrarily evaluated as the magnitude of forward motion. The fittest leg position genome has a +20° rotation for each leg, since a millipede that reaches forward with its legs to the greatest extent will move forward the furthest when its legs are reset. E.g. for a 6 legged millipede, the optimal gait was represented by the genome: "10100 10100 10100 10100 10100 10100" (gaps for visual clarity).

Note that a different fitness criteria could have just as readily been selected (e.g. backwards or rotational motion).

Evolutionary Process

The following steps describe the generalized evolutionary process:

1. Random rotational leg motions (genes) were generated for a millipede and the resulting genome placed in a genome pool. The process was repeated until a genome pool of N_{pool} genomes was created. The genome pool was then evolved over N_{gens} generations using a particular genetic algorithm. The genome from the resulting genome pool with the highest genome fitness was recorded. This entire step was repeated 100 times and the average of the recorded highest genome fitness values was determined for the given N_{pool} and N_{gens}.

2. Step 1 repeated for every combination of N_{pool} (odd value) andN_{gens} (even value) in the domain: $3 \leq N_{pool} \leq 80$ and $2 \leq N_{gens} \leq 80$. The average highest genome fitness value for each unique combination of N_{pool} and N_{gens} was then plotted using Matlab.

3. Steps 1 & 2 were performed for each of the genetic algorithms:
    a. Random (no genetic algorithm applied to each generation).
    b. Crossover
    c. Double crossover
    d. Mutation
    e. Double mutation

<u>Description of Genetic Algorithms</u>

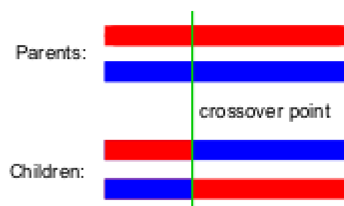
The application of each genetic algorithm followed the form:

1. Remove the parent genome with the lowest fitness from the pool. This results is a pool size of: $N_{pool} - 1$. This is an even number due to the odd values for pool sizes that were used.
2. Make a copy of the parent genome with the highest fitness.
3. Evolve each parent genome (in the case of mutation or double-mutation), or pair of genomes (in the case of crossover or double-crossover), using the given genetic algorithm. This does not change the pool size, which is: $N_{pool} - 1$.
4. Add in the parent genome with the highest fitness. This restores the pool size to: N_{pool}.

The particular evolution action on each genome is described below. In all evolutionary strategies, if the decimal value of any evolved gene in a child genome was outside the range $\pm$ 20, it was truncated to fall within that range.

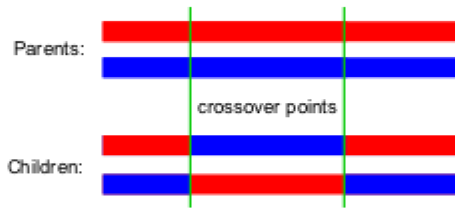
<u>Crossover</u>

Two parent genomes were combined to create two child genomes as shown below.



Double crossover

Two parent genomes were combined to create two child genomes as shown below:



Mutation

A bit in the genome bit string was inverted at a random position in the bit string.

Double Mutation

Two bits in the genome bit string were inverted at two random positions in the bit string.

Results

The fitness value for each pool size and number of evolution generations are plotted below for each evolutionary strategy. The highest fitness value possible (fastest forward motion) was normalized to a value of 10.

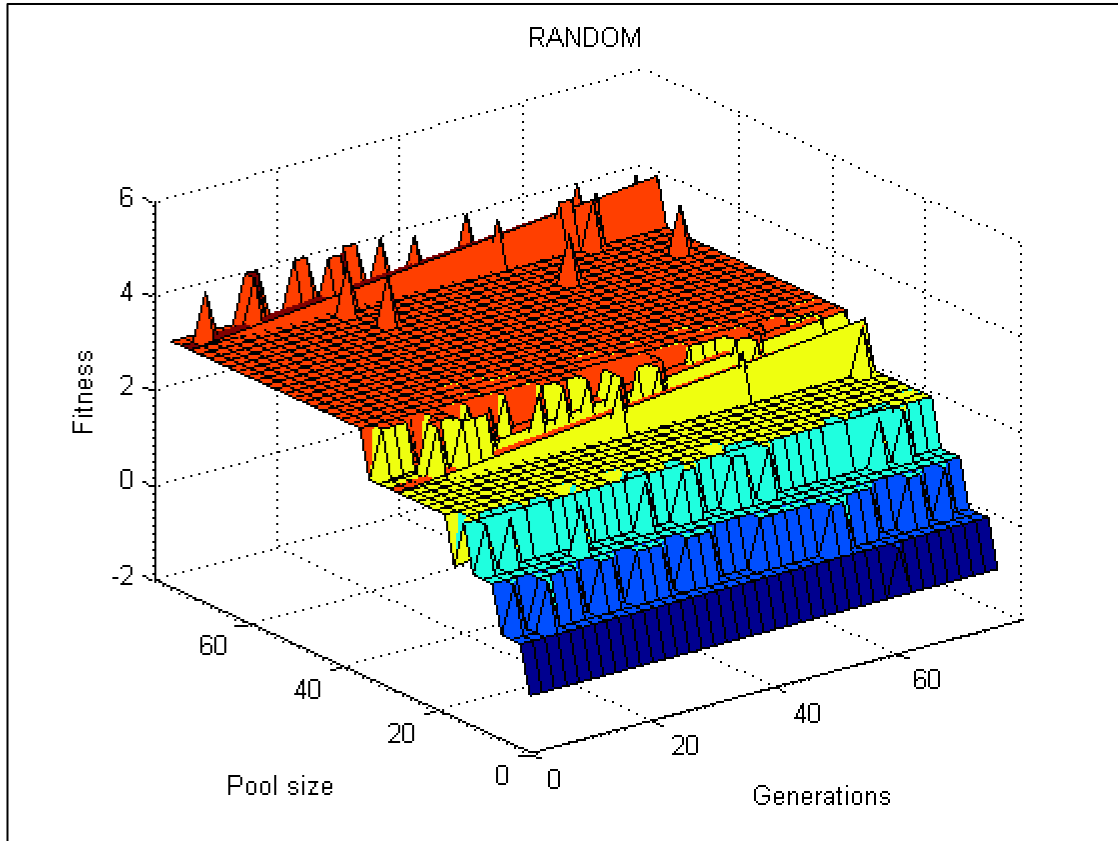
The minimum pool size and number of generations required to achieve the highest fitness value for each evolutionary strategy is summarized in the table below:

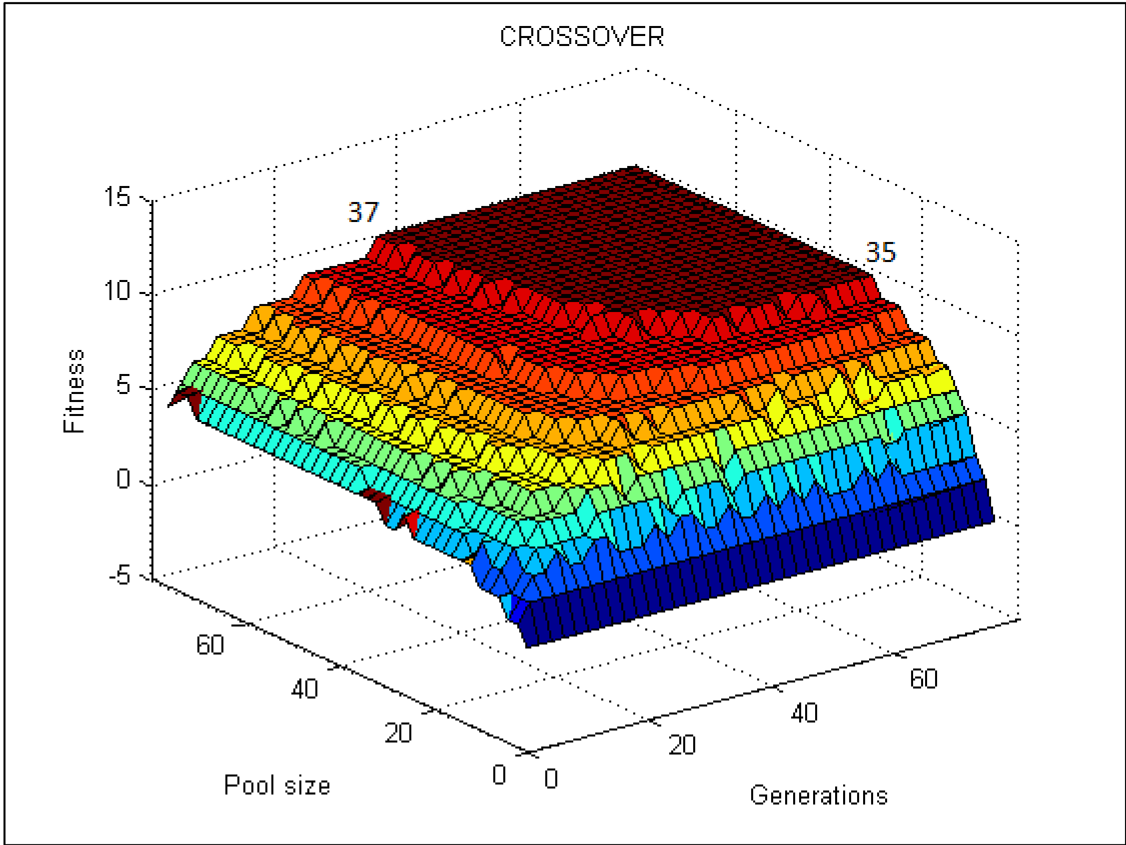
6-Legged Millipede	Random	Crossover	Mutation	Double Crossover	Double Mutation
Pool size	none	35	10	10	8
Generations	none	37	19	10	8

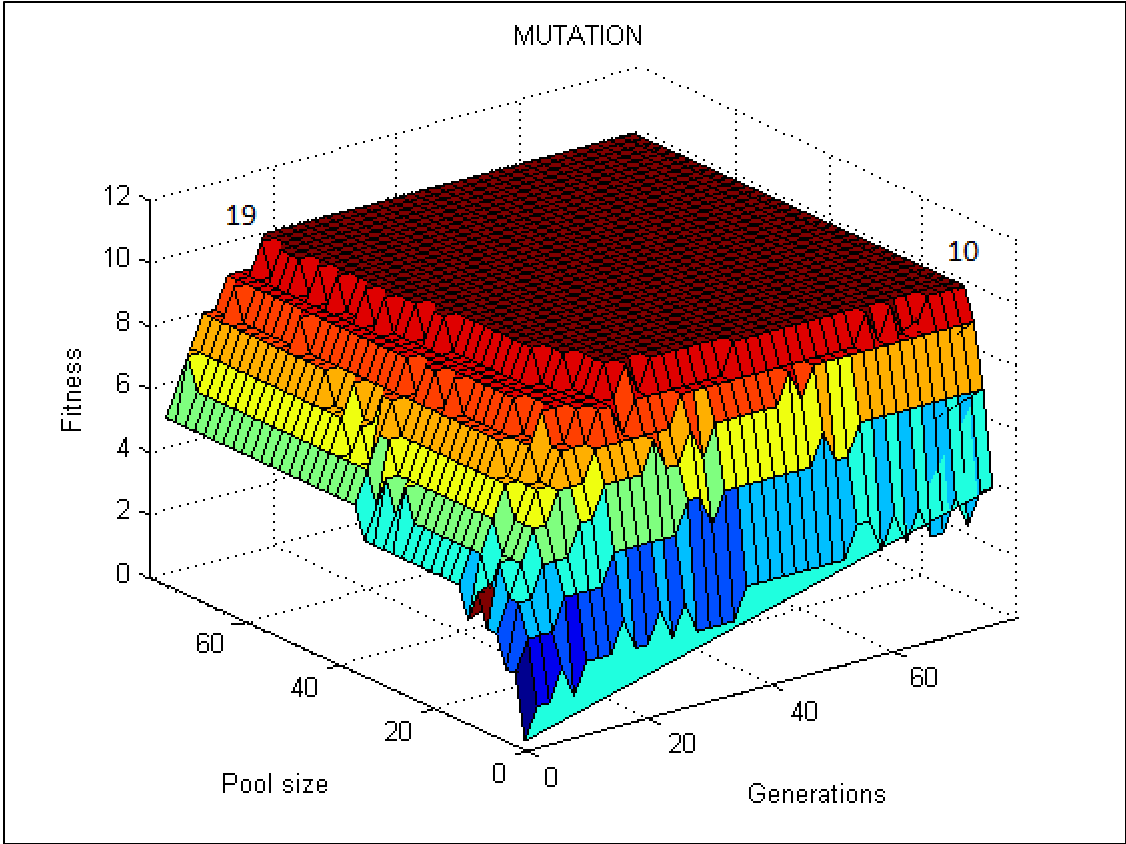
The most effective evolutionary strategy was double mutation, which required the smallest genome pool size (8) and least number of generations (8) to achieve optimal fitness.

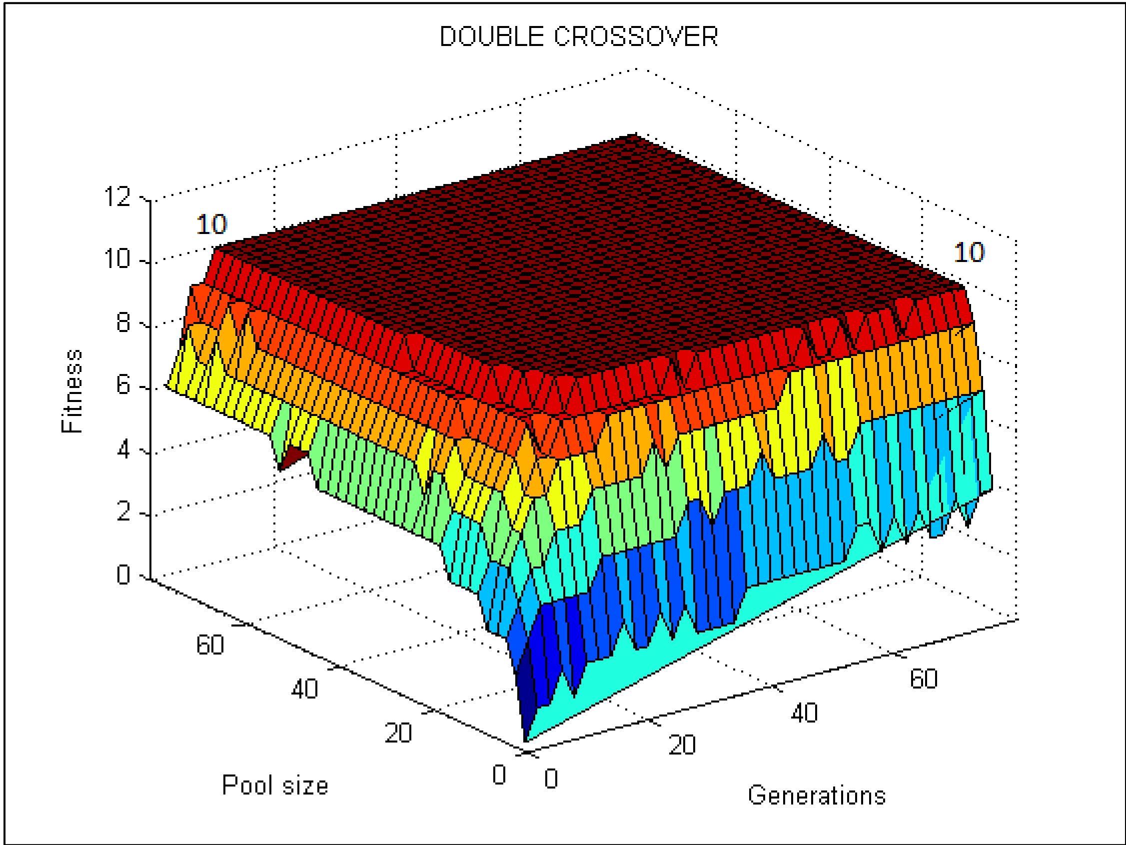
The least effective evolutionary strategy was random evolution, which never reached optimal fitness within the pool-size and generations domain.

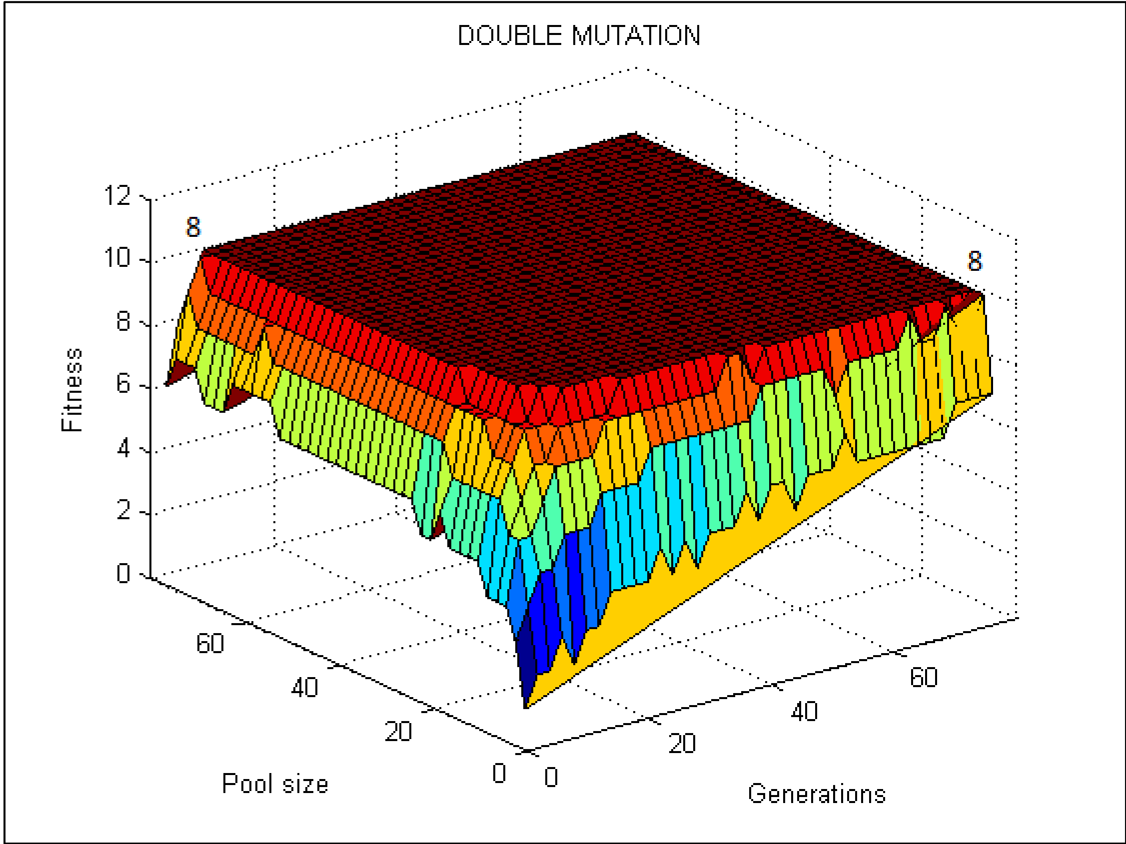
Matlab Plots For 6-Leg Millipede




CROSSOVER
Fitness
15
10
5
0
-5
37
35
60
40
20
0
Pool size
0
20
40
60
Generations


MUTATION
Fitness
12
10
8
6
4
2
0
19
10
60
40
20
0
Pool size
0
20
40
60
Generations


DOUBLE CROSSOVER
Fitness
12
10
8
6
4
2
0
10
10
60
40
20
0
Pool size
0
20
40
60
Generations


DOUBLE MUTATION
Fitness
12
10
8
6
4
2
0
8
8
60
40
20
0
Pool size
0
20
40
60
Generations

APPENDIX C

SHORTEST PATH THROUGH MAZE

Introduction

A path finding program was implemented to explore the performance of the Dijkstra (a simplified version) and A* search algorithms. Unlike breadth-first, depth-first, and iterative depth-first searches, the Dijkstra and A* searches are active search strategies (i.e. employ heuristics). At any current node, possible next nodes (child/candidate nodes) are all the adjacent nodes that are not off-map or coincident with maze walls. Dijkstra and A* search algorithms use different heuristics to select the next node in the search path.

Dijkstra Algorithm

Dijkstra's algorithm selects a next node from a candidate set of possible next nodes based on a candidate node's distance from the search start point. The candidate node with the shortest linear distance from the start point is selected. Thus in an open space search, nodes in all the different directions are explored uniformly in an expanding circular manner. This search strategy is algebraically expressed as:
$f(n) = g(n)$. $g(n)$ measures the unweighted negative distance of any node n to the start node, and $f(n)$ is the fitness function. The shortest distance is less negative and therefore fitter.

The program code to implement a simplified Dijkstra algorithm is shown below.

```
private void DijkstraSearch(object sender, DoWorkEventArgs e)
{
    while (openList.Count != 0)
    {
        BackgroundWorker wkr = sender as BackgroundWorker;
        if (Cancelled(wkr, e) == true) break;

        //get a next location to evaluate:
        MyPoint currentPoint = openList.ElementAt(0);   //initially gets start point.
        openList.RemoveAt(0);   //delete item from list.
        closedList.Add(currentPoint);   //add retrieved point to closed list.

        //check whether next location is the end point:
        if (TestForEndPoint(currentPoint) == true)
        {
            //store history of previous nodes that led to current node (end point):
            foreach (Point p in currentPoint.historyList) resultList.Add(new MyPoint(p.X, p.Y));
            //update GUI:
            DrawBitmap();
            break;
        }

        //get the next set of locations adjacent to the current location:
        List<MyPoint> nextPointList = GetAdjacentPoints(currentPoint);
        //add next set of locations to the open list if they have not been previously added:
        openList = openList.Union(nextPointList).Except(closedList).ToList();
        //sort open list by distance from start point.
        openList.Sort(MyPoint.SortByStartDistance);
        //update GUI:
        DrawBitmap();   //terminate search.

        Thread.Sleep(searchSpeed);  //control search speed.
    }
}
```

A* Algorithm

The A* algorithm selects a next node from a candidate set of possible next nodes based on a candidate node's distance from both the search start and end points. The candidate node with the shortest linear distance from the start point plus the *weighted* shortest linear distance from the endpoint is selected. A weight > 1 (3 is ideal) is required in order to keep the search more focused on the endpoint than the start point. This search strategy is algebraically expressed as:

$f(n) = g(n) + h(n)$. $g(n)$ is the unweighted the negative distance of any node n to the start node, $h(n)$ is the weighted negative distance of any node n to the end node, and $f(n)$ is the fitness function.

The program code to implement the A* algorithm is shown below.

```
private void SearchAStar(object sender, DoWorkEventArgs e)
{
    while (openList.Count != 0)
    {
        BackgroundWorker wkr = sender as BackgroundWorker;
        if (Cancelled(wkr, e) == true) break;

        //get a next location to evaluate:
        MyPoint currentPoint = openList.ElementAt(0);    //initially gets start point.
        openList.RemoveAt(0);    //delete item from list.
        closedList.Add(currentPoint);    //add retrieved point to closed list.

        //check whether next location is the end point:
        if (TestForEndPoint(currentPoint) == true)
        {
            //store history of previous nodes that led to current node (end point):
            foreach (Point p in currentPoint.historyList) resultList.Add(new MyPoint(p.X, p.Y));
            //update GUI:
            DrawBitmap();
            break;  //terminate search.
        }

        //get the next set of locations adjacent to the current location:
        List<MyPoint> nextPointList = GetAdjacentPoints(currentPoint);
        //add next set of locations to the open list if they have not been previously added:
        openList = openList.Union(nextPointList).Except(closedList).ToList();
        //sort open list by distance from start point.
        openList.Sort(MyPoint.SortByStartEndDistance);
        //update GUI:
        DrawBitmap();

        Thread.Sleep(searchSpeed);  //control search speed.
    }
}
```

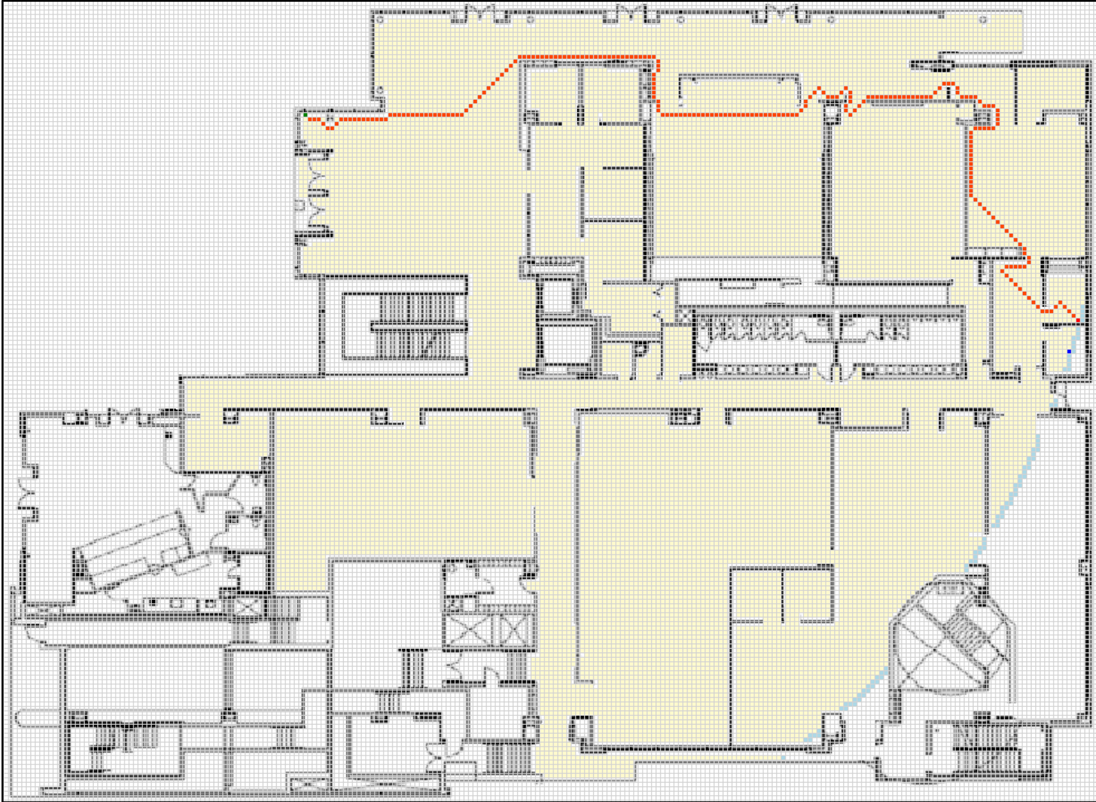
Maze Design

The Maseeh building 1st floor and Cramer Hall 1st floor plans were used to auto-generate mazes. Sealed areas include stairwells, bathrooms, utility areas, and outdoors.

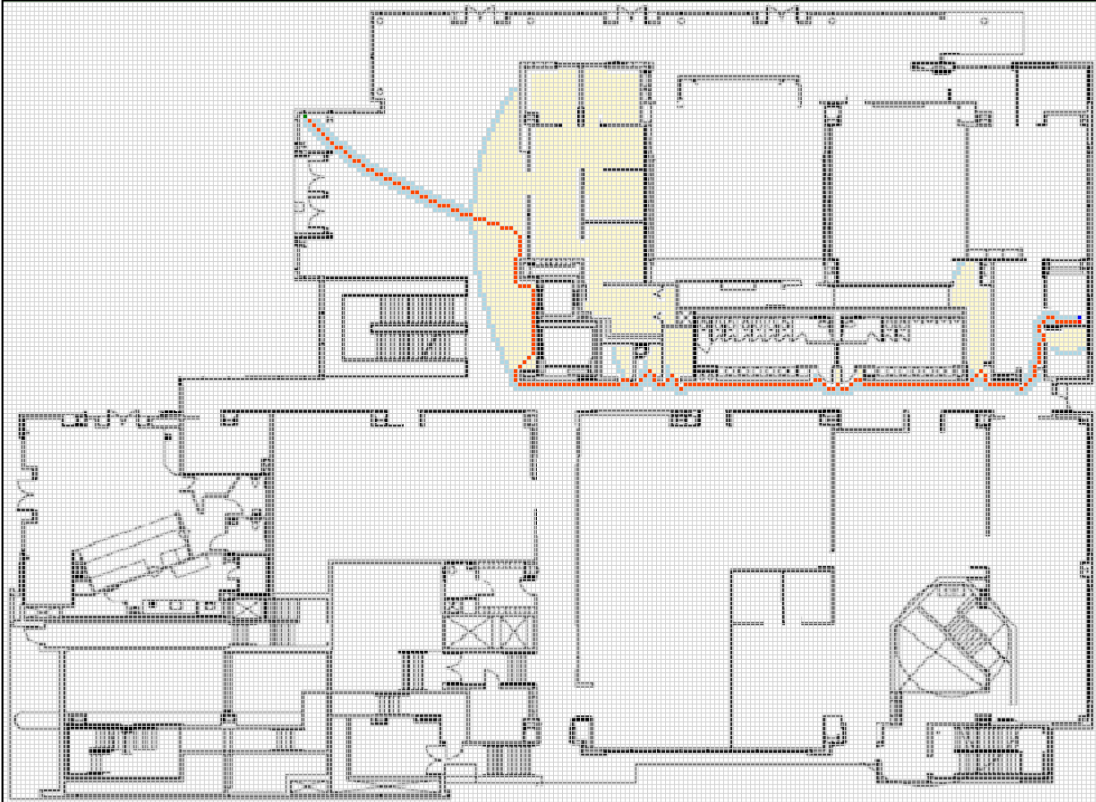
Search Results

Example searches using Dijkstra's algorithm and the A* algorithm are shown below.

Simulated Maseeh Building 1st Floor, navigated using Dijkstra's algorithm:



Simulated Maseeh Building 1st Floor, navigated using A* algorithm:



Simulated Cramer Hall 1st Floor, navigated using Dijkstra's algorithm:

Simulated Cramer Hall 1st Floor, navigated using A* algorithm:

Yellow areas indicate previously traversed nodes (closed list). Light blue areas indicate non-traversed potential candidate nodes (open list). The dark orange line indicates the discovered path from start to finish points (result list).

Following is an analysis of the search results for the Maseeh Building 1st Floor search:

MASEEH BLDG	Total Searchable Nodes (N_S)	Total Traversed Nodes (N_T)	Search Path Node Length	Ratio N_S/N_T
Dijkstra	20228	18514	233	91.5%
A*	20228	2031	220	10.0%

Following is an analysis of the search results for the Cramer Hall 1st Floor search:

CRAMER HALL	Total Searchable Nodes (N_S)	Total Traversed Nodes (N_T)	Search Path Node Length	Ratio N_S/N_T
Dijkstra	11947	11135	322	93.2%
A*	11947	1364	311	11.4%

Conclusion

Both the A* and Dijkstra algorithms discovered the endpoint, however the A* algorithm did it with less searching and with far more efficiency and speed. The ratio of the actual-searched space vs. total searchable space is a good indication of each algorithm's effectiveness. The A* algorithm is only implementable if the distance to the endpoint can be determined. This extra information allows the A* algorithm to be a more informed and therefore be a superior algorithm.

Simulations with a denser obstacle dense environment (first floor areas filled with obstacles) significantly reduced the performance gap between the two algorithms. This can mainly be attributed to the reduced total traversed nodes for the Dijkstra algorithm.

In this application, every search path was determined by heuristics, except in rare cases where two child nodes had equal suitability. Heuristics had a very limited role in the generation of child nodes, just to disallow child nodes off-map or coincident with maze walls. Dijkstra and A* search strategies are both active search strategies in contrast to breadth-first, depth-first, and iterative depth-first searches.

REFERENCES

A* search algorithm. (n.d.). In *Wikipedia*. Retrieved November 4, 2012, from http://en.wikipedia.org/wiki/A*_search_algorithm

Braunl, T. (2008). *Embedded Robotics.* Berlin, Heidelberg: Springer-Verlag.

Dijkstra's algorithm. (n.d.). In *Wikipedia*. Retrieved November 4, 2012, from http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

Fuzzy Control System. (n.d.). In Wikipedia. Retrieved November 25, 2012, from http://en.wikipedia.org/wiki/Fuzzy_control_system

Fuzzy Inference Process. (n.d.) In MathWorks. Retrieved November 25, 2012, from http://www.mathworks.com/help/fuzzy/fuzzy-inference-process.html#FP346

Li W. (1994). Fuzzy-Logic-Based Reactive Behavior Control of an Autonomous Mobile System in Unknown Environments. Engng Applic. Artif. Intell. Vol. 7, No. 5, pp. 521-531

Luger, G. (2009). *Artificial Intelligence.* Boston, MA: Pearson.

Portland State University. (2010). *First Floor Plan Engineering Building*. Retrieved November 5, 2012, from http://www.fap.pdx.edu/floorplans/docs/EB_f1317916752.pdf

Russell, S. and Norvig, P. (2010). *Artificial Intelligence: a modern approach.* Upper Saddle River, NJ: Pearson.

Wolfer, J. and George, C. (2006). *Fuzzy Logic Control For Robot Maze Traversal: An Undergraduate Case Study.* World Congress on Computer Science, Engineering and Technology Education.