Inlämningsuppgift
Artificiell intelligens i dataspel
2014

Sebastian Zander
a12sebza@mail.his.se
870318

Institutionen för kommunikation och information
Högskolan i Skövde

1 Introduktion

Uppgiften för kursen *artificiell intelligens i dataspel* går ut på att först lösa fem valfria problem med fem olika tekniker:

1. Söktekniker
2. Evolutionära tekniker
3. Konnektivistiska tekniker
4. Skriptekniker
5. Tillståndstekniker

Sedan ska en större applikation (spel eller simulering) utvecklas med en större fördjupning inom en eller flera av teknikerna.

Kursen behandlar AI i dataspel. AI i dataspel utvecklas framförallt för att återspegla mänskligt beteende.

Exempel på bra AI i dataspel är AI:n i Dwarf Fortress som är väldigt komplicerad. Exempel på dålig AI är framförallt äldre spel, där AI:n ofta är enkelspårig.

Rapporten kommer inledas med teknikerna, där varje teknik får ett eget delkapitel. Sedan kommer applikationen beskrivas. Därefter kommer en slutsats.

2 Tekniker

2.1 Söktekniker

2.1.1 Problem

Problemet som löses genom söktekniker i den här rapport är en labyrint-lösning. En sökteknik används först för att skapa labyrinten (en slumpbaserad variant av Prims algoritm används), därefter används en annan sökteknik (A*) för att lösa labyrinten från en slumpad startpunkt till en slumpad slutpunkt. Med dessa algoritmer hittas garanterat en lösning sålänge som startpunkten och slutpunkten befinner sig inom labyrinten.

2.1.2 Design

2.1.3 Kod

2.1.3.1 <**headerfilnamn**>

2.1.3.2 <**headerfilnamn**>

2.1.3.3 <…>

2.1.3.4 <**definitionsfilnamn**>

2.1.3.5 <**definitionsfilnamn**>

2.1.3.6 <…>

2.1.4 Körexempel

2.1.5 Analys

2.2 Evolutionära tekniker

2.2.1 Problem

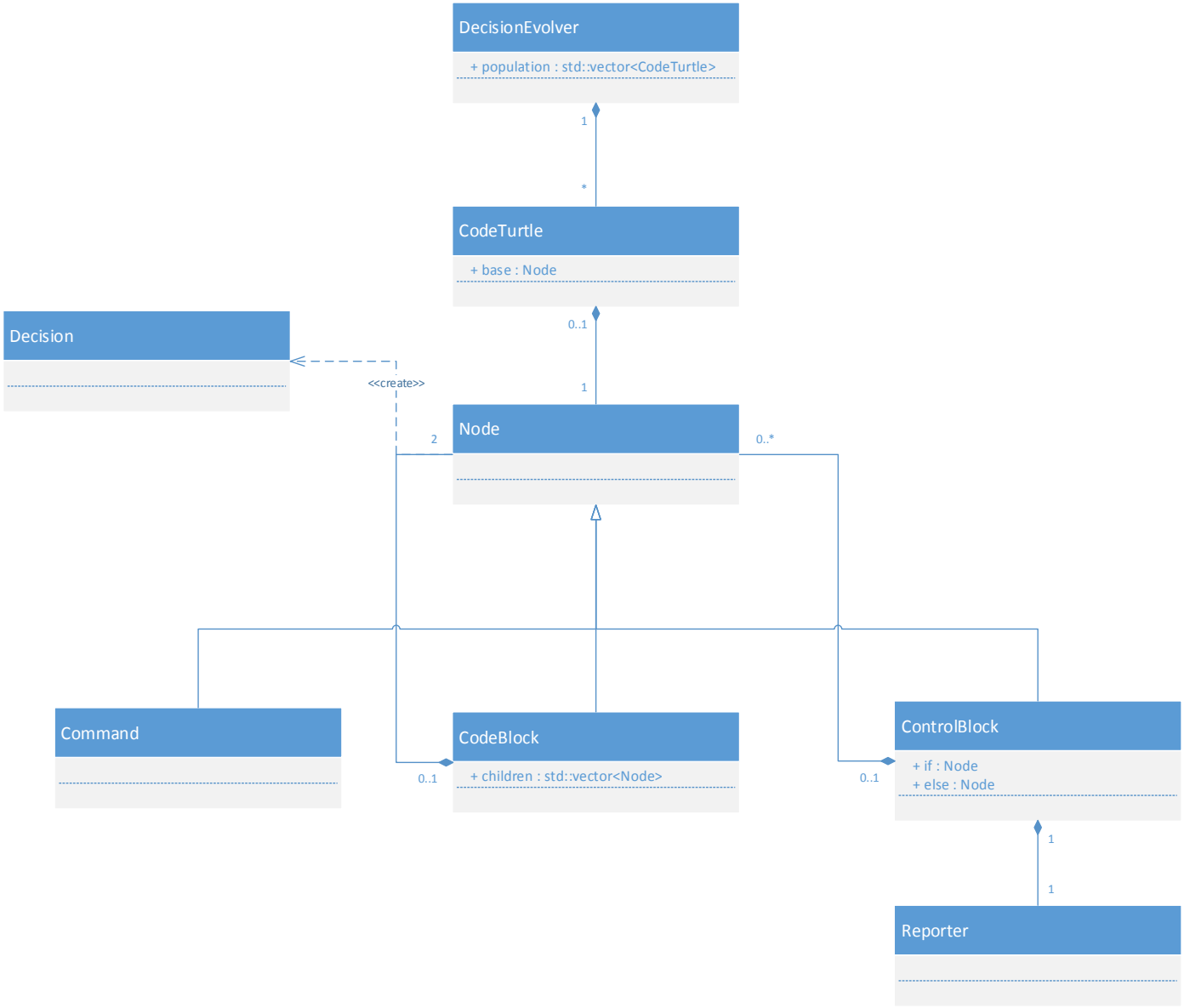
Problemet för en evolutionär teknik är även här labyrintlösning. Skapandet av labrinten görs på samma sätt som förra problemet (Söktekniker), men labyrinten löses istället evolutionärt. Den evolutionära lösningen som tas fram är ett beteende som skapas av ett enkelt syntax-träd. Varje individ i populationen är en robot som exekverar en sådan kod-snutt, som alltså representeras av ett syntax-träd. Exekveringen görs inför varje steg och låter roboten ändra riktning beroende på vilka rutor runt den som är väggar, varpå roboten går ett steg framåt. Syntaxen består av följande:

1. 4 kommandon:
    a. ”turn_left”, vilket vrider roboten åt vänster.
    b. ”turn_right”, vilket vrider roboten åt höger.
    c. ””, vilket inte gör någonting
    d. ”if else”, vilket utför en av kommando-block utifrån om ett villkor är sant eller falskt (se nedan). Med hjälp av blanka kommandon kan koden uttrycka ett ”not”-villkor, genom en blank if, etc.
2. 3 villkor:
    a. ”wall_left”, sant om det finns en vägg direkt till vänster om roboten.
    b. ”wall_right”, sant om det finns en vägg direkt till höger om roboten.
    c. ”wall_forward”, sant om det finns en vägg direkt framför roboten.

Problemet kan göras mer interesant genom att använda samma population på flera labyrinter, och se om samma ”kod” kan användas för att lösa även andra labyrinter än den koden har evolverats för att lösa. Programmet startar därför med en liten labyrint. Varje gång en lösning tas fram skapas en ny labyrint som är lite större än den förra.

En robot har ett begränsat antal steg på sig att lösa uppgiften, antalet steg beräknas från antalet totala rutor i labyrinten (båda väggar och gånger) dividerat på 2. Detta ger rätt så många steg, men även en männskligt utvecklad kod kommer i värsta fall kräva att roboten går genom hela labyrinten för att hitta en lösning.

2.2.2 Design

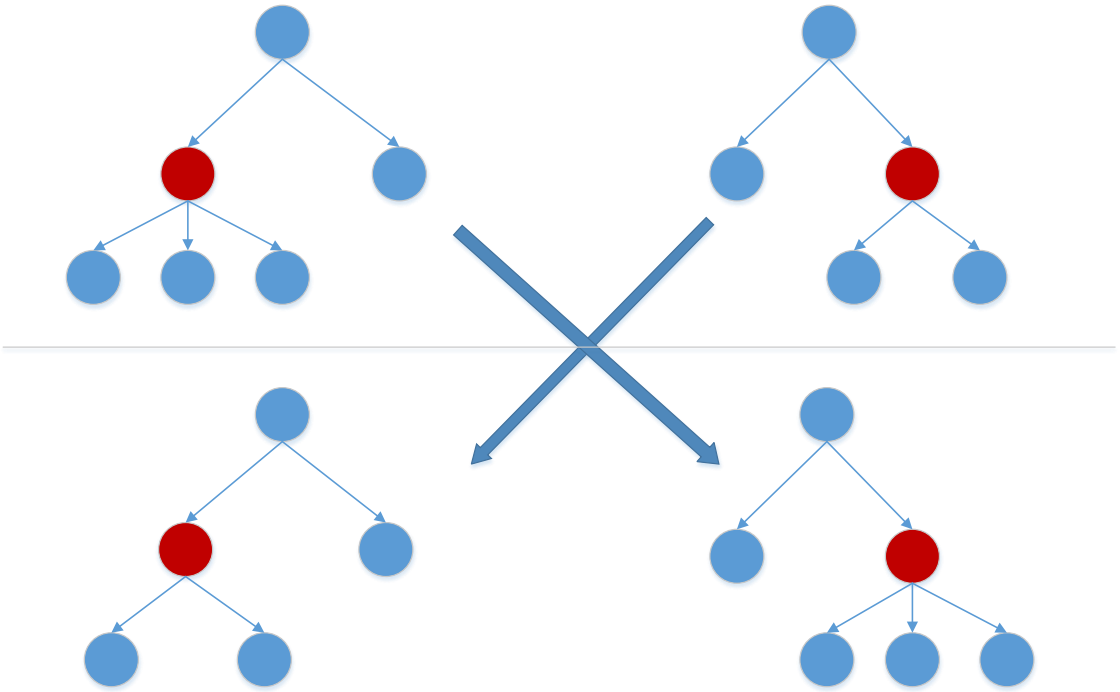


Figur 1. Klassdiagram över lösningen.

Figur 1 visar hur lösningen är designad med hjälp av ett klassdiagram. Det finns tre barnklasser till nodklassen, varav två har barnnoder. ControlBlock har en barnnod för if-blocket och en för else-blocket, medan Codeblock har ett godtyckligt antal. Rotnoden som finns i varje bot (CodeTurtle) är ingångspunkten för bottens ”kod”, vilken körs genom att kalla evaluate(). CodeBlock propagerar kallet till sina barn (i turordning). ControlBlock å andra sidan propagerar kallet till if-noden om reportern (vilket representerar vilken kontrolvillkoret) returnerar sant, i andra fall propageras kallet till else-noden. Command skapar rätt Decision beroende på vilken typ av kommando det är.

För att kunna mutera och överkorsa gener behövs ett sätt att välja en nod, vilket görs med functionen select_node(). En nod väljs ut i stort sett slumpmässigt, men chansen är större att en nod högre upp i hierarkin väljs ut. Detta görs genom att både CodeBlock och ControlBlock har en viss chans att välja ett av barnen, eller låta ett av barnen välja en nod (genom att propagera kallet). Command väljer alltid ut sig själv.

För att överkorsa gener mellan två genom (bottar) väljs två noder ut i vardera genom. Dessa två noder byter plats, vilket visas i Figur 2.



Figur 2. Överkorsning

Mutation fungerar genom att mutera den valda genen, detta görs genom att kalla mutate() på noden. ControlBlock slumpar sitt vilkor och kallar mutate() på sina två barnnoder. Command slumpar sin typ. CodeBlock slumpar om ordningen på sina noder, och kan ta bort eller lägga till noder slumpmässigt (dock begränsat till ett visst antal noder som max). Djupet på trädet måste begränsas, dels för att programmet inte ska ta för lång tid, men också för att förhindra stack overflow.

Fitnessfunktionen är: antalet steg till målet från den position botten slutar plus antalet steg botten har gått. Med andra ord så kommer en bot som kommit fram till målet ha så stor fitness som det antal steg den tog, medan en bot som inte nått fram har en fitness motsvarande avstånd till mål plus max antal tillåtna steg. En högre fitness betyder alltså att botten är mindre ”bra”. Avståndet i fitnessfunktion är manhattanavstånd.

2.2.3 Kod

2.2.3.1 CodeBlock.h

```
#pragma once

#include "Decision.h"
#include <vector>
#include <memory>
#include "Node.h"

class Maze;
class Randomizer;

class CodeBlock: public Node
{
public:
        typedef std::shared_ptr<CodeBlock> ptr;
        static ptr make();

        CodeBlock();

        CodeBlock(const CodeBlock&) = delete;
        CodeBlock& operator=(const CodeBlock&) = delete;
        CodeBlock(CodeBlock&&) = delete;
```

```
        CodeBlock& operator=(CodeBlock&&) = delete;

        Decision evaluate(Maze& maze, Coord current_coord, Decision current_decision) override
;
        Node::ptr clone() override;
        Node::ptr select_node(Randomizer& random) override;
        void mutate(Randomizer& random, int depth) override;
        void replace(Node::ptr& from, Node::ptr& to) override;
        void format(Formatter& format) override;
        int max_depth() override;

private:

        std::vector<Node::ptr> children_;
};
```

2.2.3.2 Command.h

```
#pragma once

#include "Decision.h"
#include <vector>
#include <memory>
#include "Node.h"

class Maze;
class Randomizer;

class Command : public Node
{
public:
        enum Type { BLANK, TURN_LEFT, TURN_RIGHT};
        Command(Type type);

        typedef std::shared_ptr<Command> ptr;
        static ptr make(Type type);

        Command(const Command&) = delete;
        Command& operator=(const Command&) = delete;
        Command(Command&&) = delete;
        Command& operator=(Command&&) = delete;

        Decision evaluate(Maze& maze, Coord current_coord, Decision current_decision) override
;
        Node::ptr clone() override;
        Node::ptr select_node(Randomizer& random) override;
        void mutate(Randomizer& random, int depth) override;
        void replace(Node::ptr& from, Node::ptr& to) override;
        void format(Formatter& format) override;
        int max_depth() override;

private:

        Type type_;
};
```

2.2.3.3 ControlBlock.h

```
#pragma once

#include "Decision.h"
#include <vector>
```

```
#include <memory>
#include "Node.h"
#include "Reporter.h"

class Maze;
class Randomizer;

class ControlBlock : public Node
{
public:
        typedef std::shared_ptr<ControlBlock> ptr;
        static ptr make();

        ControlBlock();

        ControlBlock(const ControlBlock&) = delete;
        ControlBlock& operator=(const ControlBlock&) = delete;
        ControlBlock(ControlBlock&&) = delete;
        ControlBlock& operator=(ControlBlock&&) = delete;

        Decision evaluate(Maze& maze, Coord current_coord, Decision current_decision) override
;
        Node::ptr clone() override;
        Node::ptr select_node(Randomizer& random) override;
        void mutate(Randomizer& random, int depth) override;
        void replace(Node::ptr& from, Node::ptr& to) override;
        void format(Formatter& format) override;
        int max_depth() override;

private:

        Node::ptr if_;
        Node::ptr else_;
        Reporter::ptr reporter_;
};
```

2.2.3.4 Decision.h

```
#pragma once
#include "../Coord.h"

struct Decision
{
public:
        enum Action { TURN_LEFT = -1, TURN_RIGHT = 1 };
        enum Direction { UP, RIGHT, DOWN, LEFT };

        Decision(Direction direction);
        Decision();

        Coord forward(Coord coord);
        Decision perform(Action action);

        Direction direction() const { return direction_; }
private:
        Direction direction_;
};
```

2.2.3.5 Formatter.h

```
#pragma once
```

```
#include <sstream>

class Formatter
{
public:
        Formatter();
        void append(std::string line);
        void push_indent();
        void pop_indent();

        std::string to_string();
        void clear();
private:
        std::stringstream ss_;
        int indent_;
};
```

2.2.3.6 Node.h

```
#pragma once

#include "../Coord.h"
#include "Decision.h"
#include <vector>
#include <memory>
#include "Formatter.h"

class Maze;
class Randomizer;

class Node : std::enable_shared_from_this<Node>
{
public:
        typedef std::shared_ptr<Node> ptr;
        //returns a random node (with no child nodes)
        static ptr make_random(Randomizer& random);

        Node(const Node&) = delete;
        Node& operator=(const Node&) = delete;
        Node(Node&&) = delete;
        Node& operator=(Node&&) = delete;

        virtual Decision evaluate(Maze& maze, Coord current_coord, Decision current_decision)
= 0;
        virtual Node::ptr clone() = 0;
        virtual void mutate(Randomizer& random, int depth) = 0;
        virtual Node::ptr select_node(Randomizer& random) = 0;
        virtual void replace(Node::ptr& from, Node::ptr& to) = 0;
        virtual void format(Formatter& format) = 0;
        virtual int max_depth() = 0;

        Node::ptr get_this();
protected:
        Node() {};
};
```

2.2.3.7 Reporter.h

```
#pragma once

#include "../Coord.h"
#include <memory>
```

```
#include "Decision.h"
#include <string>

class Maze;
class Randomizer;

class Reporter
{
public:
        typedef std::unique_ptr<Reporter> ptr;
        static ptr make(Randomizer& random);

        virtual ~Reporter() {};
        virtual bool evaluate(Maze& maze, Coord current_coord, Decision decision) = 0;
        virtual ptr clone() = 0;
        virtual std::string to_string() const = 0;
protected:
        bool evaluate_helper(Maze& maze, Coord current_coord, Decision decision);
};

class WallAhead : public Reporter
{
        bool evaluate(Maze& maze, Coord current_coord, Decision decision) override;
        std::string to_string() const override;
        ptr clone() override;
};

class WallLeft : public Reporter
{
        bool evaluate(Maze& maze, Coord current_coord, Decision decision) override;
        std::string to_string() const override;
        ptr clone() override;
};

class WallRight : public Reporter
{
        bool evaluate(Maze& maze, Coord current_coord, Decision decision) override;
        std::string to_string() const override;
        ptr clone() override;
};

class TrueReporter : public Reporter
{
        bool evaluate(Maze& maze, Coord current_coord, Decision decision) override;
        std::string to_string() const override;
        ptr clone() override;
};
```

2.2.3.8 CodeTurtle.h

```
#pragma once

#include "Node.h"
#include "Decision.h"

class CodeTurtle
{
public:
        CodeTurtle();

        //returns amount of steps since latest start position reset
        int step(Maze& maze);
```

```
        void set_start_position(Coord coord);

        const Node::ptr& get_base() const;
        Node::ptr& get_base();

        Decision get_current_decision() const;
        Coord get_current_coord() const;

        void set_fitness(int fitness) { fitness_ = fitness; }
        int get_fitness() const { return fitness_; }
private:
        int steps_;
        int fitness_;
        Node::ptr base_;
        Decision current_decision_;
        Coord current_coord_;
};
```

2.2.3.9 DecisionEvolver.h

```
#pragma once

#include "CodeTurtle.h"
#include "../Random.h"
#include "../Coord.h"
#include "../Solver.h"
#include <utility>

class DecisionEvolver : public Solver
{
public:
        DecisionEvolver();
        DecisionEvolver(Coord start, Coord end, int max_turtle_steps, int population_size, dou
ble combination_probability, double mutation_probability);

        bool Step(Maze& maze) override;
        void Reset() override;
        void Reset(Coord start, Coord end) override;
        void DecisionEvolver::NewMaze(Coord start, Coord end, int max_steps);

private:
        void crossover(CodeTurtle&, CodeTurtle&);
        void mutate(CodeTurtle& n);

        int calculate_fitness(CodeTurtle& g, Maze& maze);
        void calculate_fitness(Maze& maze);
        void sort_by_fitness();

        void evolve(Maze& maze);

        int distance(Coord a, Coord b);

        Randomizer rand_;
        std::vector<CodeTurtle> population_;
        CodeTurtle& select();

        int population_size_;
        float combination_p_, mutation_p_;
        Coord start_, end_;
        int steps_, max_turtle_steps_;
};
```

2.2.3.10 CodeBlock.cpp

```
#include "CodeBlock.h"
#include "../Random.h"
#include <algorithm>

CodeBlock::ptr CodeBlock::make()
{
	return std::make_shared<CodeBlock>();
}

CodeBlock::CodeBlock()
{

}

Decision CodeBlock::evaluate(Maze& maze, Coord current_coord, Decision current_decision)
{
	for (Node::ptr n : children_)
	{
		current_decision = n->evaluate(maze, current_coord, current_decision);
	}
	return current_decision;
}

Node::ptr CodeBlock::select_node(Randomizer& random)
{
	//select a child at random

	int index = random.NextInt(0, children_.size());

	//Only select self if there are no children or once in every lenght of child.
	if (index == 0 || children_.size() == 0)
	{
		return get_this();
	}

	Node::ptr s = children_[index - 1];
	//select that child half the time, select a childs child half the time.
	if (random.NextInt(0, 2) == 0)
	{
		return s;
	}
	else
	{
		return s->select_node(random);
	}
}

Node::ptr CodeBlock::clone()
{
	CodeBlock::ptr copy = std::make_shared<CodeBlock>();
	for (Node::ptr n : children_)
	{
		copy->children_.push_back(std::move(n->clone()));
	}
	return std::move(copy);
}

void CodeBlock::mutate(Randomizer& random, int depth)
{
	int size = children_.size();
```

```
        if (size != 0)
        {
                int size_change = random.NextInt(-2 - (2 - children_.size()), 2 - (2 -
 children_.size()));

                auto rand = [this, &random, size](int i){return random.NextInt(0, i - 1); };

                //shuffle nodes
                try
                {
                        std::random_shuffle(children_.begin(), children_.end(), rand);
                }
                catch (...)
                {
                        throw;
                }

        int rem = std::min(static_cast<int>(children_.size()), std::max(0, rand(size) + size_c
hange));
                //erase a random amount of nodes
                children_.erase(children_.begin() + rem, children_.end());
                //mutate some of the remaining children
                for (auto& child : children_)
                {
                        if (random.NextInt(0, 1) == 0)
                        {
                                if (random.NextInt(0, 1) == 0)
                                {
                                        child = Node::make_random(random);
                                }

                                if (depth > 0)
                                {
                                        child->mutate(random, depth - 1);
                                }
                        }
                }
                //add a random amount of nodes, such that on average there are as many
                //nodes left as there were nodes to start with.
                for (int i = 0; i < rand(size) + size_change; ++i)
                {
                        Node::ptr c = Node::make_random(random);
                        if (depth > 0)
                        {
                                c->mutate(random, depth - 1);
                        }
                        children_.push_back(c);
                }

        }
        else
        {
                int size = random.NextInt(0, 2);
                for (int i = 0; i < size; ++i)
                {
                        //if no nodes, just add a random node.
                        Node::ptr c = Node::make_random(random);
                        if (depth > 0)
                        {
                                c->mutate(random, depth - 1);
                        }
```

```
                        children_.push_back(c);
                }
        }
}

void CodeBlock::replace(Node::ptr& from, Node::ptr& to)
{
        //can't replace self
        if (this != from.get())
        {
                for (Node::ptr& c : children_)
                {
                        if (c == from)
                        {
                                c = to;
                                break;
                        }
                        c->replace(from, to);
                }
        }
}

void CodeBlock::format(Formatter& format)
{
        for (Node::ptr& n : children_)
        {
                n->format(format);
        }
}

int CodeBlock::max_depth()
{
        int max = 0;
        for (Node::ptr& n : children_)
        {
                max = std::max(n->max_depth(), max);
        }
        return max + 1;
}
```

2.2.3.11 Command.cpp

```
#include "Command.h"
#include "../Random.h"
#include <algorithm>

Command::ptr Command::make(Command::Type type)
{
        return std::make_shared<Command>(type);
}


Command::Command(Command::Type type)
{
        type_ = type;
}

Decision Command::evaluate(Maze& maze, Coord current_coord, Decision current_decision)
{
        switch (type_)
        {
        case Command::BLANK:
```

```
                break;
        case Command::TURN_LEFT:
                current_decision.perform(Decision::TURN_LEFT);
                break;
        case Command::TURN_RIGHT:
                current_decision.perform(Decision::TURN_RIGHT);
                break;
        default:
                break;
        }
        return current_decision;
}

Node::ptr Command::select_node(Randomizer& random)
{
        return get_this();
}

Node::ptr Command::clone()
{
        return std::move(Command::make(type_));
}

void Command::mutate(Randomizer& random, int depth)
{
        int blank = random.NextInt(0, 1);
        type_ = static_cast<Command::Type>(random.NextInt(blank, 2));
}

void Command::replace(Node::ptr& from, Node::ptr& to)
{

}

void Command::format(Formatter& format)
{
        switch (type_)
        {
        case Command::BLANK:
                break;
        case Command::TURN_LEFT:
                format.append("turn_left");
                break;
        case Command::TURN_RIGHT:
                format.append("turn_right");
                break;
        default:
                break;
        }
}

int Command::max_depth()
{
        return 0;
}
```

2.2.3.12 ControlBlock.cpp

```
#include "ControlBlock.h"
#include "../Random.h"
#include <algorithm>
#include "Command.h"
```

```
ControlBlock::ptr ControlBlock::make()
{
	return std::make_shared<ControlBlock>();
}


ControlBlock::ControlBlock()
{
	reporter_ = std::make_unique<TrueReporter>();
	if_ = Command::make(Command::BLANK);
	else_ = Command::make(Command::BLANK);
}

Decision ControlBlock::evaluate(Maze& maze, Coord current_coord, Decision current_decision)
{
	if (reporter_->evaluate(maze, current_coord, current_decision))
	{
		return if_->evaluate(maze, current_coord, current_decision);
	}
	else
	{
		return else_->evaluate(maze, current_coord, current_decision);
	}

}

Node::ptr ControlBlock::select_node(Randomizer& random)
{
	int r = random.NextInt(0, 2);
	if (r == 0)
	{
		return get_this();
	}
	else if (r == 1)
	{
		return if_;
	}
	else
	{
		return else_;
	}
}

Node::ptr ControlBlock::clone()
{
	ControlBlock::ptr copy = std::make_shared<ControlBlock>();
	copy->reporter_ = reporter_->clone();
	copy->if_ = if_->clone();
	copy->else_ = else_->clone();
	return std::move(copy);
}

void ControlBlock::mutate(Randomizer& random, int depth)
{
	//half the time randomize the reporter
	if (random.NextInt(0, 1) == 0 || dynamic_cast<TrueReporter*>(reporter_.get()) != nullp
tr)
	{
		reporter_ = Reporter::make(random);
	}
```

```
        //half the time randomize the if
        if (random.NextInt(0, 1) == 0)
        {
                if (random.NextInt(0, 1) == 0)
                {
                        if_ = Node::make_random(random);
                }
                if (depth > 0)
                {
                        if_->mutate(random, depth - 1);
                }
        }
        //half the time randomize the else
        if (random.NextInt(0, 1) == 0)
        {
                //half the time randomize a type
                if (random.NextInt(0, 1) == 0)
                {
                        else_ = Node::make_random(random);
                }
                if (depth > 0)
                {
                        else_->mutate(random, depth);
                }
        }
}

void ControlBlock::replace(Node::ptr& from, Node::ptr& to)
{
        //can't replace self
        if (this != from.get())
        {
                if (if_ == from)
                {
                        if_ = to;
                }
                else if (else_ == from)
                {
                        else_ = to;
                }
                else
                {
                        if_->replace(from, to);
                        else_->replace(from, to);
                }

        }
}

void ControlBlock::format(Formatter& format)
{
        format.append("if (" + reporter_->to_string() + "?)");
        format.push_indent();
        if_->format(format);
        format.pop_indent();
        format.append("else");
        format.push_indent();
        else_->format(format);
        format.pop_indent();
}
```

```
int ControlBlock::max_depth()
{
	return std::max(else_->max_depth(), if_->max_depth()) + 1;
}
```

2.2.3.13 Decision.cpp

```
#include "Decision.h"

Decision::Decision(Decision::Direction direction)
{
	direction_ = direction;
}

Decision::Decision()
{
	direction_ = Decision::UP;
}

Coord Decision::forward(Coord coord)
{
	switch (direction_)
	{
	case Decision::UP:
		return Coord(coord.x, coord.y - 1);
		break;
	case Decision::RIGHT:
		return Coord(coord.x + 1, coord.y);
		break;
	case Decision::DOWN:
		return Coord(coord.x, coord.y + 1);
		break;
	case Decision::LEFT:
		return Coord(coord.x - 1, coord.y);
		break;
	default:
		return coord;
		break;
	}
}

Decision Decision::perform(Decision::Action action)
{
	int n = static_cast<int>(direction_)+static_cast<int>(action);
	if (n < 0)
	{
		n = 3;
	}
	direction_ = static_cast<Direction>(n % 4);
	return *this;
}
```

2.2.3.14 Formatter.cpp

```
#include "Formatter.h"
#include <exception>

Formatter::Formatter()
	:indent_(0)
{}

void Formatter::append(std::string line)
```

```
{
        for (int i = 0; i < indent_; ++i)
        {
                ss_ << "    ";
        }
        ss_ << line << "\n";
}

void Formatter::push_indent()
{
        indent_++;
}

void Formatter::pop_indent()
{
        if (indent_ == 0)
        {
                throw std::out_of_range("Too many pops");
        }
        indent_--;
}

std::string Formatter::to_string()
{
        return ss_.str();
}

void Formatter::clear()
{
        ss_.clear();
}
```

2.2.3.15 Node.cpp

```
#include "Node.h"
#include "../Random.h"
#include "CodeBlock.h"
#include "ControlBlock.h"
#include "Command.h"

Node::ptr Node::get_this()
{
        return shared_from_this();
}

Node::ptr Node::make_random(Randomizer& random)
{
        int r = random.NextInt(0, 5);
        switch (r)
        {
        case 0:
                return ControlBlock::make();
        case 1:
                return CodeBlock::make();
        case 2:
                return CodeBlock::make();
        case 3:
                return Command::make(Command::BLANK);
        case 4:
                return Command::make(Command::BLANK);
        case 5:
                return Command::make(Command::BLANK);
```

```
        default:

                //supress warning
                return nullptr;
        }
}
```

2.2.3.16 Reporter.cpp

```
#include "Reporter.h"
#include "../Random.h"
#include "../Maze.h"

Reporter::ptr Reporter::make(Randomizer& random)
{
        int r = random.NextInt(0, 2);
        switch (r)
        {
        case 0:
                return std::move(std::make_unique<WallAhead>());
        case 1:
                return std::move(std::make_unique<WallLeft>());
        case 2:
                return std::move(std::make_unique<WallRight>());
        default:
                return nullptr;
        }
}

Reporter::ptr WallAhead::clone()
{
        return std::move(std::make_unique<WallAhead>());
}

Reporter::ptr WallLeft::clone()
{
        return std::move(std::make_unique<WallLeft>());
}

Reporter::ptr WallRight::clone()
{
        return std::move(std::make_unique<WallRight>());
}

Reporter::ptr TrueReporter::clone()
{
        return std::move(std::make_unique<TrueReporter>());
}

bool Reporter::evaluate_helper(Maze& maze, Coord current_coord, Decision decision)
{
        auto dir = decision.direction();
        switch (dir)
        {
        case Decision::UP:
                return maze(current_coord.x, current_coord.y - 1).cost == -1;
        case Decision::RIGHT:
                return maze(current_coord.x + 1, current_coord.y).cost == -1;
        case Decision::DOWN:
                return maze(current_coord.x, current_coord.y + 1).cost == -1;
        case Decision::LEFT:
                return maze(current_coord.x - 1, current_coord.y).cost == -1;
```

```
        default:
                return false;
        }
}

bool WallAhead::evaluate(Maze& maze, Coord current_coord, Decision decision)
{
        return evaluate_helper(maze, current_coord, decision);
}

bool WallLeft::evaluate(Maze& maze, Coord current_coord, Decision decision)
{
        decision.perform(Decision::TURN_LEFT);
        return evaluate_helper(maze, current_coord, decision);
}

bool WallRight::evaluate(Maze& maze, Coord current_coord, Decision decision)
{
        decision.perform(Decision::TURN_RIGHT);
        return evaluate_helper(maze, current_coord, decision);
}

bool TrueReporter::evaluate(Maze& maze, Coord current_coord, Decision decision)
{
        return true;
}

std::string WallAhead::to_string() const
{
        return "wall_ahead";
}

std::string WallLeft::to_string() const
{
        return "wall_left";
}

std::string WallRight::to_string() const
{
        return "wall_right";
}

std::string TrueReporter::to_string() const
{
        return "true";
}
```

2.2.3.17 CodeTurtle.cpp

```
#include "CodeTurtle.h"
#include "Command.h"
#include "../Maze.h"

CodeTurtle::CodeTurtle()
{
        base_ = Command::make(Command::BLANK);
        steps_ = 0;
}

int CodeTurtle::step(Maze& maze)
{
        current_decision_ = base_->evaluate(maze, current_coord_, current_decision_);
```

```
        Coord next = current_decision_.forward(current_coord_);

        if (maze(next.x, next.y).cost != -1)
        {
                current_coord_ = next;
        }
        ++steps_;
        return steps_;
}

void CodeTurtle::set_start_position(Coord coord)
{
        current_decision_ = Decision();
        steps_ = 0;
        current_coord_ = coord;
}

const Node::ptr& CodeTurtle::get_base() const
{
        return base_;
}

Node::ptr& CodeTurtle::get_base()
{
        return base_;
}

Decision CodeTurtle::get_current_decision() const
{
        return current_decision_;
}

Coord CodeTurtle::get_current_coord() const
{
        return current_coord_;
}
```

2.2.3.18 DecisionEvolver.cpp

```
#include "DecisionEvolver.h"
#include "../Maze.h"

DecisionEvolver::DecisionEvolver()
        :population_size_(0)
        ,combination_p_(0.0f)
        ,mutation_p_(0.0f)
        ,steps_(0)
        ,max_turtle_steps_(0)
{
}

DecisionEvolver::DecisionEvolver(Coord start, Coord end, int max_turtle_steps, int population_
size, double combination_probability, double mutation_probability)
        :population_size_(population_size)
        , combination_p_(combination_probability)
        , mutation_p_(mutation_probability)
        , steps_(0)
        , start_(start)
        , end_(end)
        , max_turtle_steps_(max_turtle_steps)
{
```

```
}

bool DecisionEvolver::Step(Maze& maze)
{
	if (steps_ == 0)
	{
		for (int i = 0; i < population_size_; ++i)
		{
			population_.push_back(CodeTurtle());

			CodeTurtle& turtle = population_[i];
			turtle.get_base() = Node::make_random(rand_);
			turtle.get_base()->mutate(rand_, 5);
		}
	}

	/*Formatter format;

	format.append("Population");
	format.push_indent();

	for (int i = 0; i < population_.size(); ++i)
	{

	format.append(std::to_string(i) + "(" + std::to_string(population_[i].get_fitness()) +
 "):");
		format.push_indent();
		population_[i].get_base()->format(format);
		format.pop_indent();
	}
	format.pop_indent();

	std::cout << format.to_string() << std::endl;*/


	if (steps_ % 2 == 0)
	{
		evolve(maze);

		/*for (auto& bot : population_)
		{
			std::cout << bot.get_fitness() << ", ";
		}
		std::cout << std::endl;*/
	}
	else
	{
		for (int i = 0; i < maze.Height(); ++i)
		{
			for (int j = 0; j < maze.Width(); ++j)
			{
				if (maze(j, i).cost != -1)
				{
					maze(j, i).color = Color::white();
				}
			}
		}

		for (auto& bot : population_)
		{
			bot.set_start_position(start_);
```

```
                        while (bot.step(maze) < max_turtle_steps_)
                        {
                                if (bot.get_current_coord() == end_)
                                {
                                        Formatter format;

        format.append("Good bot! (" + std::to_string(bot.get_fitness()) + ")");
                                        format.push_indent();

                                        format.push_indent();
                                        bot.get_base()->format(format);
                                        format.pop_indent();

                                        format.pop_indent();
                                        std::cout << format.to_string() << std::endl;
                                        return false;
                                }
                        }

                        Coord c = bot.get_current_coord();
                        maze(c.x, c.y).color = Color::blue();
                }
                maze(start_.x, start_.y).color = Color::green();
                maze(end_.x, end_.y).color = Color::red();

        }
        steps_++;

        return true;
}

void DecisionEvolver::evolve(Maze& maze)
{
        calculate_fitness(maze);
        sort_by_fitness();

        std::vector<CodeTurtle> child_population;
        while (child_population.size() < population_size_)
        {
                Formatter format;

                CodeTurtle& c0 = select();
                CodeTurtle& c1 = select();

                if (rand_.NextBool(combination_p_))
                {
                        crossover(c0, c1);
                }
                if (rand_.NextBool(mutation_p_))
                {
                        mutate(c0);
                }
                if (rand_.NextBool(mutation_p_))
                {
                        mutate(c1);
                }

                child_population.push_back(c0);
                child_population.push_back(c1);
        }
        population_ = child_population;
```

```
        calculate_fitness(maze);
}

void DecisionEvolver::Reset()
{
        steps_ = 0;
        population_.clear();
}

void DecisionEvolver::NewMaze(Coord start, Coord end, int max_steps)
{
        start_ = start;
        end_ = end;
        max_turtle_steps_ = max_steps;
}

void DecisionEvolver::Reset(Coord start, Coord end)
{
        start_ = start;
        end_ = end;
        Reset();
}

CodeTurtle& DecisionEvolver::select()
{
        //tournament selection
        auto rind = [this]() { return rand_.NextInt(0, population_.size() - 1); };

        int best = rind();

        for (int i = 0; i < population_.size() - 1; ++i)
        {
                int curr = rind();
                if (population_[curr].get_fitness() < population_[best].get_fitness())
                {
                        best = curr;
                }
        }
        return population_[best];
}

void DecisionEvolver::sort_by_fitness()
{
        std::sort(population_.begin(), population_.end(), [&](const CodeTurtle& g0, const Code
Turtle& g1) {return g0.get_fitness() < g1.get_fitness(); });
}

int DecisionEvolver::calculate_fitness(CodeTurtle& g, Maze& maze)
{
        g.set_start_position(start_);
        for (int i = 0; i < max_turtle_steps_; ++i)
        {
                if (g.get_current_coord() == end_)
                {
                        return i;
                }
                g.step(maze);
        }
        while (g.step(maze) <= max_turtle_steps_)
                ;
```

```
        return max_turtle_steps_ + distance(g.get_current_coord(), end_);
}

void DecisionEvolver::calculate_fitness(Maze& maze)
{
        for (CodeTurtle& gene : population_)
        {
                int fitness = calculate_fitness(gene, maze);
                gene.set_fitness(fitness);
        }
}

void DecisionEvolver::crossover(CodeTurtle& t0, CodeTurtle& t1)
{
        Node::ptr b0 = t0.get_base();
        Node::ptr b1 = t1.get_base();
        Node::ptr s0 = b0->select_node(rand_);
        Node::ptr s1 = b1->select_node(rand_);
        if (b0 == s0)
        {
                b0 = b1->clone();
        }
        else
        {
                int max_d = b0->max_depth();
                int select_d = max_d - s1->max_depth();
                //Force the code to a certain depth
                if (max_d - select_d >= 5)
                {
                        Node::ptr n = Node::make_random(rand_);
                        n->mutate(rand_, std::max(0, std::min(2, 5 - select_d)));
                        b0->replace(s0, n);
                }
                else
                {
                        b0->replace(s0, s1->clone());
                }
        }
        t0.get_base() = b0;

        if (b1 == s1)
        {
                b1 = b0->clone();
        }
        else
        {
                int max_d = b1->max_depth();
                int select_d = max_d - s0->max_depth();
                //Force the code to a certain depth
                if (max_d - select_d > 5)
                {
                        Node::ptr n = Node::make_random(rand_);
                        n->mutate(rand_, std::max(0, std::min(2, 5 - select_d)));
                        b1->replace(s0, n);
                }
                else
                {
                        b1->replace(s1, s0->clone());
                }
        }
```

```
        t1.get_base() = b1;
}

void DecisionEvolver::mutate(CodeTurtle& t)
{
        Node::ptr s = t.get_base()->select_node(rand_);
        Node::ptr n = s->clone();

        int max_d = t.get_base()->max_depth();
        int select_d = max_d - s->max_depth();

        if (rand_.NextInt(0, 1) == 1 || max_d - select_d >= 5)
        {
                n = Node::make_random(rand_);
        }

        n->mutate(rand_, std::max(0, std::min(2, 5 - select_d)));
        if (s == t.get_base())
        {
                t.get_base() = n;
        }
        else
        {
                t.get_base()->replace(s, n);
        }
}

int DecisionEvolver::distance(Coord a, Coord b)
{
        return abs(a.x - b.x) + abs(a.y - b.y);
}
```

2.2.4 Körexempel

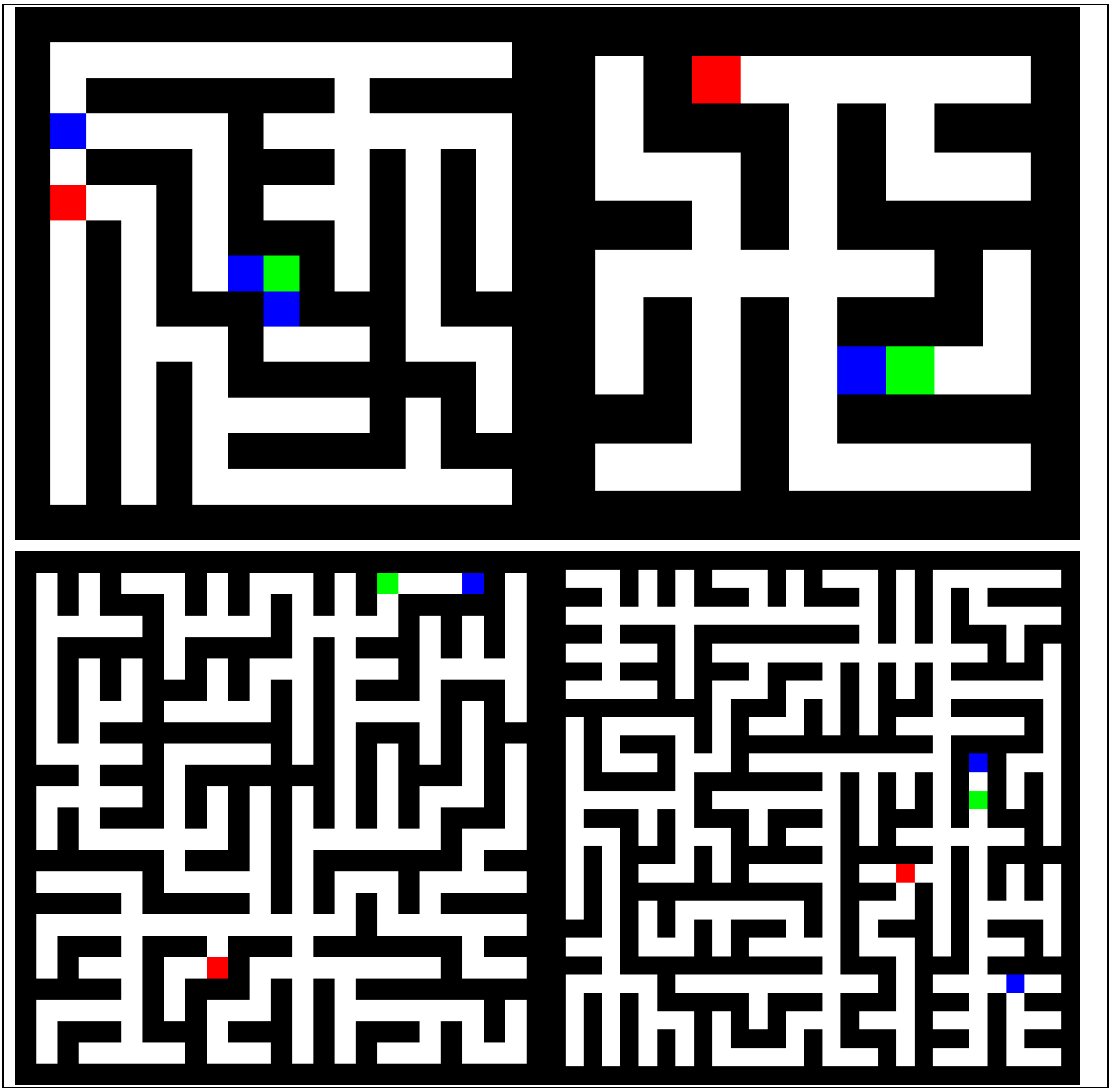
Körexemplet baseras på följande funktion:

```
int decision_based_main()
{
	Window win("Maze solver!", 800, 800);

	std::random_device rd;
	Randomizer rand(rd());

	Coord start, end;
	Maze maze(11, 11);
	PrimGenerator generator(RandomCoord(rand, maze));

	start = RandomCoord(rand, maze);
	end = RandomCoord(rand, maze);
	while (start == end)
		end = RandomCoord(rand, maze);

	DecisionEvolver solver(start, end, maze.Width() * maze.Height(), 100, 0.5f, 0.01f);
	//Astar solver(start, end);
	long long step = 0;
	system("mkdir screens");
	while (win.Open())
	{
		win.PollEvents();
		while (generator.Step(maze))
			;

		if (!generator.Step(maze))
		{
			if (!solver.Step(maze))
			{
				std::cout << "SOLVED!" << std::endl;
				maze = Maze(maze.Width() + 2, maze.Height() + 2);
				generator.Reset(maze, RandomCoord(rand, maze));

				start = RandomCoord(rand, maze);
				end = RandomCoord(rand, maze);
				while (end == start)
				{
					end = RandomCoord(rand, maze);
				}
				solver.NewMaze(start, end, maze.Width() * maze.Height() / 2);
			}
		}

		maze.Render(win, 0, 0, 800, 800);

		win.Display();
		win.PrintScreen("screens/screen_" + std::to_string(step) + ".bmp");
		step++;
		win.Clear();

	}
	return 0;
}
```

Varje gång en lösning hittas skapas labyrinten om med en storlek större (två tiles), men populationen förblir densamma. DecisionEvolver skriver ut koden för den ”vinnande” botten till konsolen varje gång.

Utskriftterna och ett antal skärmdumpar demonstrerar körexemplet. Även om funktionen kan köras i oändligheten kommer det här körexemplet avbrytas efter bara ett antal hittade lösningar. En observation är att väldigt ofta löser samma bot flera labyrinter i rad.

2.2.4.1 Skärmdumpar



I bilderna ovan är väggar svarta och korrider vita. Målet är rött och startpunkten är grön. Det blå punkterna representerar en eller flera bottars slutposition.

2.2.4.2 Konsolutskrifter

```
Good bot! (34)
        if (wall_right?)
```

```
            if (wall_ahead?)
                turn_right
            else
        else
            if (wall_right?)
                if (wall_right?)
                    if (wall_right?)
                        if (wall_right?)
                            turn_left
                        else
                            turn_left
                    else
                        turn_right
                else
                    if (wall_right?)
                        if (wall_right?)
                            turn_left
                        else
                            turn_left
                    else
                        turn_right
            else
                turn_right

SOLVED!
Good bot! (128)
        if (wall_right?)
            turn_left
        else
            if (wall_right?)
                if (wall_right?)
                    if (wall_right?)
                        turn_left
                    else
                        turn_left
                else
                    turn_right
            else
                if (wall_right?)
                    if (wall_right?)
                        turn_left
                    else
                        turn_left
                else
                    turn_right

SOLVED!
Good bot! (102)
        if (wall_right?)
            if (wall_ahead?)
                turn_left
            else
        else
            turn_right

SOLVED!
Good bot! (114)
        if (wall_right?)
            if (wall_ahead?)
                turn_left
            else
```

```
        else
            if (wall_right?)
                if (wall_right?)
                    if (wall_left?)
                    else
                        turn_left
                else
                    if (wall_left?)
                        turn_right
                    else
            else
                if (wall_right?)
                    if (wall_right?)
                        if (wall_left?)
                        else
                            turn_left
                    else
                        if (wall_left?)
                            turn_right
                        else
                else
                    if (wall_right?)
                        if (wall_left?)
                        else
                            turn_left
                    else
                        if (wall_left?)
                            turn_right
                        else

SOLVED!
Good bot! (102)
        if (wall_right?)
            if (wall_ahead?)
                turn_left
            else
        else
            turn_right

SOLVED!
Good bot! (120)
        if (wall_right?)
            if (wall_right?)
                if (wall_right?)
                    if (wall_right?)
                        if (wall_left?)
                        else
                            turn_left
                    else
                        if (wall_left?)
                            turn_right
                        else
                else
                    if (wall_right?)
                        if (wall_right?)
                            if (wall_left?)
                            else
                                turn_left
                        else
                            if (wall_left?)
                                turn_right
```

```
                            else
                    else
                        if (wall_right?)
                            if (wall_left?)
                            else
                                turn_left
                        else
                            if (wall_left?)
                                turn_right
                            else
            else
                if (wall_right?)
                    if (wall_left?)
                    else
                        turn_left
                else
                    if (wall_left?)
                        turn_right
                    else
        else
            turn_right

SOLVED!
Good bot! (69)
        if (wall_right?)
            if (wall_right?)
                turn_left
            else
                if (wall_right?)
                else
        else
            if (wall_right?)
            else
                turn_right

SOLVED!
Good bot! (281)
        turn_left
        turn_right
        if (wall_left?)
        else
            turn_left
        turn_right
        if (wall_ahead?)
        else
        turn_left
        turn_right
        if (wall_left?)
        else
            turn_left
        turn_right
        if (wall_ahead?)
        else
        turn_left

SOLVED!
Good bot! (186)
        if (wall_right?)
        else
            turn_right
        turn_left
```

```
        turn_left
        turn_right
        turn_right
        if (wall_ahead?)
        else
        turn_left
        if (wall_right?)
        else
            turn_right

SOLVED!
Good bot! (40)
        if (wall_ahead?)
            turn_right
        else
            if (wall_ahead?)
                turn_left
            else
        turn_right
        turn_left
        if (wall_ahead?)
            turn_left
        else
        turn_right
        turn_left
        if (wall_ahead?)
            turn_right
        else
        turn_right
        if (wall_right?)
        else
            turn_right
        turn_right
        if (wall_right?)
        else
            turn_right
        turn_left
        if (wall_ahead?)
            turn_right
        else
            if (wall_ahead?)
                turn_left
            else
        turn_left
        turn_right

SOLVED!
```

2.2.5 Analys

2.3 Konnektionistiska tekniker

2.4 Skripttekniker

2.5 Tillståndstekniker

3 Applikation

3.1 Problem

3.2 Design

3.3 Kod

3.3.1 <headerfilnamn>

3.3.2 <headerfilnamn>

3.3.3 <…>

3.3.4 <definitionsfilnamn>

3.3.5 <definitionsfilnamn>

3.3.6 <…>

3.4 Körexempel

3.5 Analys

4 Slutsats