Inlämningsuppgift  
Artificiell intelligens i dataspel  
2014

Sebastian Zander

a12sebza@mail.his.se

870318

Institutionen för kommunikation och information

Högskolan i Skövde

# 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 männskligt 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.

# Tekniker

## Söktekniker

### 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.

### Design

### Kod

#### <headerfilnamn>

#### <headerfilnamn>

#### <…>

#### <definitionsfilnamn>

#### <definitionsfilnamn>

#### <…>

### Körexempel

### Analys

## Evolutionära tekniker

### 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:
   1. ”turn\_left”, vilket vrider roboten åt vänster.
   2. ”turn\_right”, vilket vrider roboten åt höger.
   3. ””, vilket inte gör någonting
   4. ”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:
   1. ”wall\_left”, sant om det finns en vägg direkt till vänster om roboten.
   2. ”wall\_right”, sant om det finns en vägg direkt till höger om roboten.
   3. ”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.

### 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.

### Kod

#### 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\_;

};

#### 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\_;

};

#### 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\_;

};

#### 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\_;

};

#### 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\_;

};

#### 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() {};

};

#### 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;

};

#### 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\_;

};

#### 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, double 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\_;

};

#### 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\_change));

//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;

}

#### 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;

}

#### 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()) != nullptr)

{

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;

}

#### 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;

}

#### 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();

}

#### 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;

}

}

#### 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";

}

#### 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\_;

}

#### 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 CodeTurtle& 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);

}

### 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();

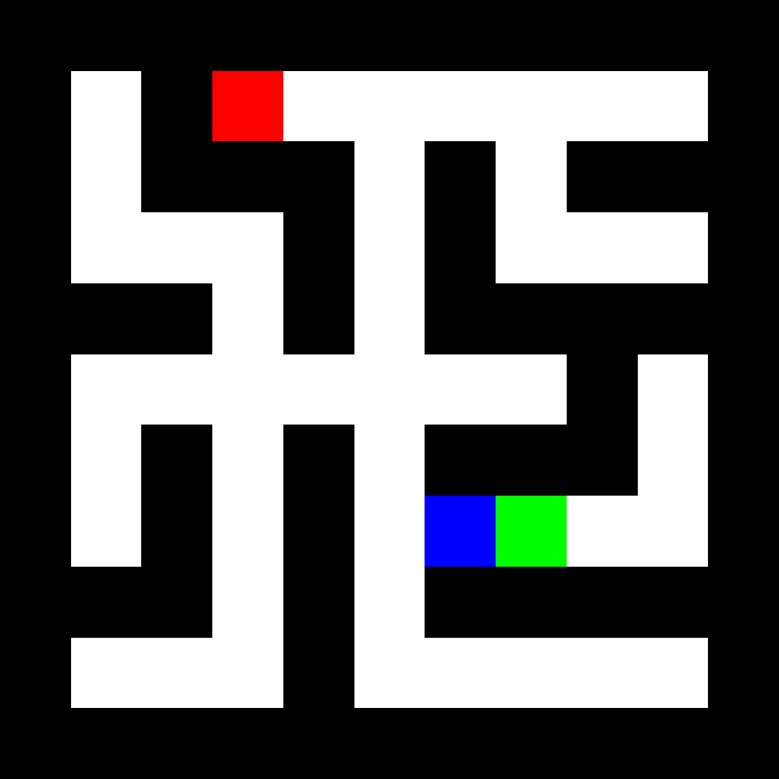
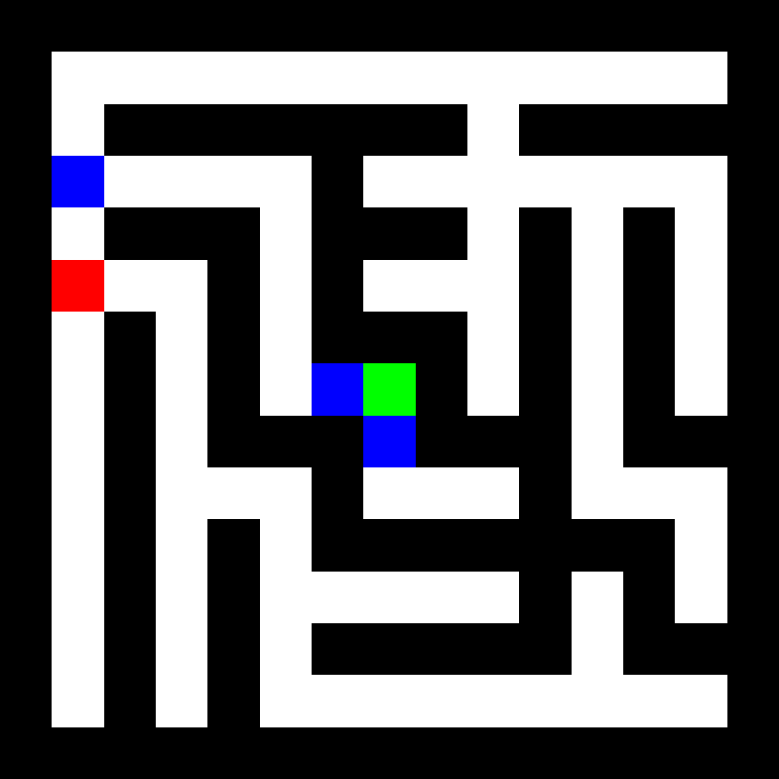
}

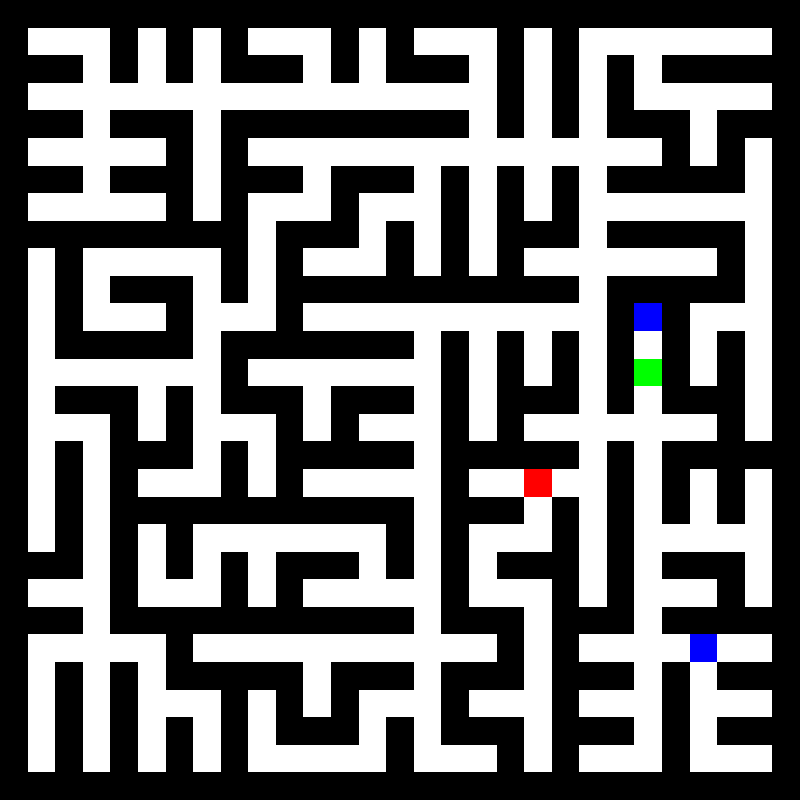
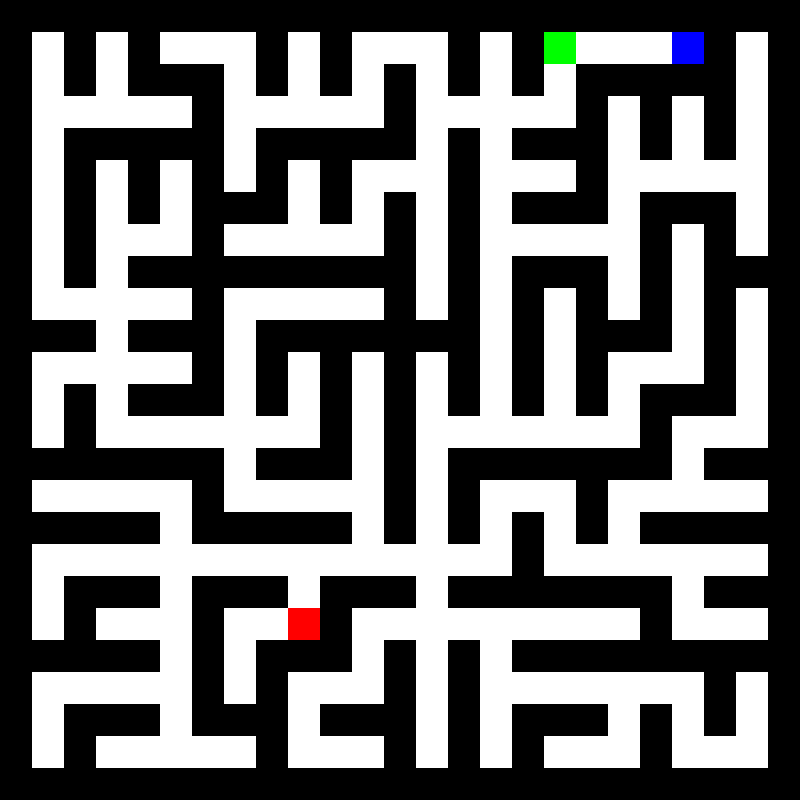
return 0;

}

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.

#### 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.

#### 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!

### Analys

## Konnektionistiska tekniker

## Skripttekniker

## Tillståndstekniker

# Applikation

## Problem

## Design

## Kod

### <headerfilnamn>

### <headerfilnamn>

### <…>

### <definitionsfilnamn>

### <definitionsfilnamn>

### <…>

## Körexempel

## Analys

# Slutsats