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 lösningen av en labyrint. Som del av problemet ingår också att skapa labyrinten.

### Design

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.

### Kod

#### Astar.h

#pragma once

#include "../Maze.h"

#include "../Coord.h"

#include "../Solver.h"

#include <queue>

#include <unordered\_set>

#include <map>

class Astar: public Solver

{

public:

Astar(Coord start, Coord end);

/\* returns true if there are more steps \*/

bool Step(Maze& maze) override;

void Reset() override;

void Reset(Coord start, Coord end) override;

private:

int Heuristic(Coord a, Coord b);

int steps\_;

Coord start\_, end\_;

Coord current\_;

Coord walker\_;

class PriorityQueueCompare

{

public:

bool operator()(std::pair<int, Coord>& t1, std::pair<int, Coord>& t2)

{

return t1.first > t2.first;

}

};

std::unordered\_set<Coord> closed\_set\_;

std::unordered\_set<Coord> open\_set\_;

std::priority\_queue<std::pair<int, Coord>, std::vector<std::pair<int, Coord>>, PriorityQueueCompare> open\_queue\_;

std::unordered\_map<Coord, int> g\_;

std::unordered\_map<Coord, int> f\_;

std::unordered\_map<Coord, Coord> parent\_of\_;

std::vector<Coord> path\_;

Maze came\_from\_;

};

#### Color.h

#pragma once

#include <cmath>

#include <array>

#include <cstdint>

#include <iostream>

//To save from weird ambiguity errors there are two make functions

//rather than an overloaded constructor, the Color is stored as four

//floats internally, but can be retrieved as both bytes as floats.

//It is stored this way to allow the values to be below 0.0 and above

//1.0, for certain operations.

//Using bytes is therefore a little inaccurate.

class Color

{

public:

typedef uint8\_t byte\_t;

Color()

{

op = multiply\_op;

}

Color static white()

{

return Color::make\_from\_floats(1.f, 1.f, 1.f);

}

Color static black()

{

return Color::make\_from\_floats(0.f, 0.f, 0.f);

}

Color static red()

{

return Color::make\_from\_floats(1.f, 0.f, 0.f);

}

Color static green()

{

return Color::make\_from\_floats(0.f, 1.f, 0.f);

}

Color static blue()

{

return Color::make\_from\_floats(0.f, 0.f, 1.f);

}

Color static make\_from\_bytes(byte\_t red\_, byte\_t green\_, byte\_t blue\_, byte\_t alpha\_ = 255)

{

Color ret;

ret.Colors\_[red\_i] = to\_float(red\_);

ret.Colors\_[green\_i] = to\_float(green\_);

ret.Colors\_[blue\_i] = to\_float(blue\_);

ret.Colors\_[alpha\_i] = to\_float(alpha\_);

return ret;

}

//Range for a Color is [0-1]

Color static make\_from\_floats(float red\_, float green\_, float blue\_, float alpha\_ = 1.0f)

{

Color ret;

ret.Colors\_[red\_i] = red\_;

ret.Colors\_[green\_i] = green\_;

ret.Colors\_[blue\_i] = blue\_;

ret.Colors\_[alpha\_i] = alpha\_;

return ret;

}

Color static make\_from\_bytes(byte\_t greyscale, byte\_t alpha\_ = 255)

{

Color ret;

ret.Colors\_[red\_i] = to\_float(greyscale);

ret.Colors\_[green\_i] = to\_float(greyscale);

ret.Colors\_[blue\_i] = to\_float(greyscale);

ret.Colors\_[alpha\_i] = to\_float(alpha\_);

return ret;

}

//Range for a Color is [0-1]

Color static make\_from\_floats(float greyscale, float alpha\_ = 1.0f)

{

Color ret;

ret.Colors\_[red\_i] = greyscale;

ret.Colors\_[green\_i] = greyscale;

ret.Colors\_[blue\_i] = greyscale;

ret.Colors\_[alpha\_i] = alpha\_;

return ret;

}

//Clamps ALL the values so that they are within the range [0-1], this is

//a hard operation, and is not reversable. Returns itself for chaining.

Color& clamp()

{

for (auto& c : Colors\_)

{

c = clamp(c);

}

return \*this;

}

float get\_red() const

{

return Colors\_[red\_i];

}

float get\_green() const

{

return Colors\_[green\_i];

}

float get\_blue() const

{

return Colors\_[blue\_i];

}

float get\_alpha() const

{

return Colors\_[alpha\_i];

}

void set\_red(float red\_)

{

Colors\_[red\_i] = red\_;

}

void set\_green(float green\_)

{

Colors\_[green\_i] = green\_;

}

void set\_blue(float blue\_)

{

Colors\_[blue\_i] = blue\_;

}

void set\_alpha(float alpha\_)

{

Colors\_[alpha\_i] = alpha\_;

}

byte\_t get\_red\_byte() const

{

return to\_byte(Colors\_[red\_i]);

}

byte\_t get\_green\_byte() const

{

return to\_byte(Colors\_[green\_i]);

}

byte\_t get\_blue\_byte() const

{

return to\_byte(Colors\_[blue\_i]);

}

byte\_t get\_alpha\_byte() const

{

return to\_byte(Colors\_[alpha\_i]);

}

void set\_red\_byte(byte\_t red\_)

{

Colors\_[red\_i] = to\_float(red\_);

}

void set\_green\_byte(byte\_t green\_)

{

Colors\_[green\_i] = to\_float(green\_);

}

void set\_blue\_byte(byte\_t blue\_)

{

Colors\_[blue\_i] = to\_float(blue\_);

}

void set\_alpha\_byte(byte\_t alpha\_)

{

Colors\_[alpha\_i] = to\_float(alpha\_);

}

/\* set the operation to be performed when apply is

 \* called with this Color as an argument.

 \* this sets the operation to add.

 \*/

void add()

{

op = add\_op;

}

/\* set the operation to be performed when apply is

\* called with this Color as an argument.

\* this sets the operation to subtract.

\*/

void subtract()

{

op = subtract\_op;

}

/\* set the operation to be performed when apply is

\* called with this Color as an argument.

\* this sets the operation to multiply.

\*/

void multiply()

{

op = multiply\_op;

}

/\* Usage:

 \* Color c = Color::white();

 \* c = apply(Color::black().subtract());

 \*

 \* Will give the same result as:

 \* Color c = Color::white();

 \* Color result = c.add(Color().subtract());

 \* c = result;

 \* Note that the second line doesn't change object c,

 \* c is modified first when result is assigned to it.

 \*/

Color apply(Color& other) const

{

return do\_operation(\*this, other, other.op);

}

Color add(const Color& c0, const Color& c1)

{

return do\_operation(c0, c1, add\_op);

}

Color subtract(const Color& c0, const Color& c1)

{

return do\_operation(c0, c1, subtract\_op);

}

Color multiply(const Color& c0, const Color& c1)

{

return do\_operation(c0, c1, multiply\_op);

}

private:

enum Color\_operator {add\_op, subtract\_op, multiply\_op};

static Color do\_operation(const Color& a, const Color& b, Color\_operator op)

{

Color ret;

switch (op)

{

case add\_op:

for (int i = 0; i < 4; ++i)

{

ret.Colors\_[i] = a.Colors\_[i] + b.Colors\_[i];

}

break;

case subtract\_op:

for (int i = 0; i < 4; ++i)

{

ret.Colors\_[i] = a.Colors\_[i] - b.Colors\_[i];

}

break;

case multiply\_op:

for (int i = 0; i < 4; ++i)

{

ret.Colors\_[i] = a.Colors\_[i] \* b.Colors\_[i];

}

break;

default:

break;

}

return ret;

}

enum Color\_index { red\_i, green\_i, blue\_i, alpha\_i };

std::array<float, 4U> Colors\_;

Color\_operator op;

static inline float to\_float(byte\_t val)

{

return static\_cast<float>(val) / 255.0;

}

//We must be careful that this doesn't just return 255 when the value

//of the float is a very small negative number, or the reverse.

//This is the problem with this approach. There are other problems, of course

static inline byte\_t to\_byte(float val)

{

return static\_cast<byte\_t>(clamp(val \* 255.0, 0.0, 255.0));;

}

static inline float clamp(float val, float min = 0.0f, float max = 1.0f)

{

return std::fmax(min, std::fmin(max, val));

}

};

#### Coord.h

#pragma once

#include <utility>

struct Coord

{

Coord(int x\_, int y\_)

:x(x\_), y(y\_)

{}

Coord() {};

int x, y;

bool operator==(const Coord& other) const

{

return other.x == x && other.y == y;

}

bool operator!=(const Coord& other) const

{

return !(\*this == other);

}

bool operator<(const Coord& other) const

{

return (x < other.x) || (x == other.x && y < other.y);

}

};

namespace std {

template <> struct hash<Coord>

{

size\_t operator()(const Coord& c) const

{

size\_t seed = std::hash<int>()(c.x);

return std::hash<int>()(c.y) + 0x9e3779b9 + (seed << 6) + (seed >> 2);

}

};

}

#### Key.h

#pragma once

struct Key

{

public:

Key()

: down(false), released(false), pressed(false), valid(false)

{}

bool down;

bool released;

bool pressed;

bool valid;

};

#### Maze.h

#pragma once

#include "Window.h"

class Maze

{

public:

struct Tile

{

Tile()

 :cost(-1)

, color(Color::black())

{

}

Tile(int cost\_, Color color\_)

:cost(cost\_)

,color(color\_)

{

}

//Any negative cost means the tile is solid/closed/occupied etc..

int cost;

Color color;

};

Maze(int w, int h);

Maze();

Maze(const Maze& other);

Maze& operator=(const Maze& other);

void Render(Window& window, int x, int y, int w, int h);

~Maze();

Tile& operator()(int x, int y);

const Tile& operator()(int x, int y) const;

const int Width() const { return w\_; }

const int Height() const { return h\_; }

bool IsBorderCoord(int x, int y) const;

private:

Tile& Get(int x, int y);

const Tile& Get(int x, int y) const;

inline bool EnsureRange(int x, int y) const { return x >= 0 && x < w\_ && y >= 0 && y < h\_; };

void InnerCopy(const Maze& other);

std::vector<Tile> maze\_;

int w\_, h\_;

SDL\_Surface\* surface\_;

};

#### PrimGenerator.h

#pragma once

#include "Random.h"

#include "Maze.h"

#include <vector>

#include "Coord.h"

class PrimGenerator

{

public:

PrimGenerator(const Coord& start);

/\* returns true if there are more steps \*/

bool Step(Maze& maze);

void Reset(Maze& maze, const Coord& start);

private:

void TryAddWalls(const Coord& coord, const Maze& maze);

void Clear(Maze& maze);

std::vector<Coord> walls\_;

int steps\_;

Randomizer random\_;

Coord start\_;

};

#### Random.h

#pragma once

/\* TO FIX:

\* The distibutions on the random number generator that uses marseene twister

\* is wrong. See STL's speech during GoingNative 2013 on rand().

\*

\*/

#include <ctime>

#include <mutex>

#include <random>

/\* A simpler random-generator, for no works on rand(), by reseeding for every number

\* with the last number. This means every Random-object is internally consistent

\* regardless of other objects.

\*/

class Randomizer

{

public:

//default seeds the generator with time()

Randomizer();

Randomizer(unsigned int seed);

void Reseed();

void Reseed(unsigned int seed);

//returns a random int from 0 up to max (inclusive)

unsigned int NextInt(unsigned int max);

//returns a random int from min to max (inclusive)

int NextInt(int min, int max);

//returns a random double from min to max (inclusive)

double NextDouble(double min, double max);

//returns a random bool, where trueFactor is a value between 0.0 and 1.0,

//the higher the factor he higher the chance for true. With the current

//implementation the random double seems to be scewed towards 0.5

bool NextBool(double true\_factor);

//just returns the next rand()

unsigned int NextRand();

private:

std::mt19937 engine\_;

};

class Noise

{

public:

double Generate(double x);

double Generate(double x, double y);

double Generate(double x, double y, double z);

private:

double FindNoise(int x);

double FindNoise(int x, int y);

double FindNoise(int x, int y, int z);

double Interpolate(double y1, double y2, double mu);

int floor(double f);

int ceil(double f);

//wraps negative ints to postive.

int WrapAround(int i);

static const double PI;

};

class PerlinNoise

{

public:

PerlinNoise(double frequency, int octaves, double amplitude, double lacunarity)

:frequency\_(frequency), lacunarity\_(lacunarity), amplitude\_(amplitude), octaves\_(octaves)

{}

void FetFrequency(double frequency) { frequency\_ = frequency; }

void SetOctaves(int octaves) { octaves\_ = octaves; }

void SetLacunarity(double lacunarity) { lacunarity\_ = lacunarity; }

void SetAmplitude(double amplitude) { amplitude\_ = amplitude; }

double Generate(double x);

double Generate(double x, double y);

double Generate(double x, double y, double z);

private:

template<typename returnType>

returnType round(double f)

{

if (f >= 0x1e23) return static\_cast<returnType>(f);

if (f > 0.f)

return static\_cast<returnType>(static\_cast<int>(f + 0.49999997f));

else

{

try

{

f = abs(f);

return static\_cast<returnType>(-static\_cast<int>(f + 0.49999997f));

}

catch (...)

{

return 0;

}

}

}

Noise noise;

double frequency\_;

double lacunarity\_;

double amplitude\_;

int octaves\_;

 };

#### Solver.h

#pragma once

#include "Coord.h"

class Maze;

class Solver

{

public:

virtual ~Solver() {}

virtual bool Step(Maze& maze) = 0;

virtual void Reset() = 0;

virtual void Reset(Coord start, Coord end) = 0;

};

#### Timer.h

#include <chrono>

class Timer

{

public:

void Start()

{

start\_ = std::chrono::steady\_clock::now();

}

int ElapsedMilliseconds()

{

auto now = std::chrono::steady\_clock::now();

auto diff = now - start\_;

millisecs\_t duration(std::chrono::duration\_cast<millisecs\_t>(diff));

return duration.count();

}

private:

typedef std::chrono::duration<int, std::milli> millisecs\_t;

std::chrono::steady\_clock::time\_point start\_;

};

#### Window.h

#pragma once

#include <SDL.h>

#include <memory>

#include <string>

#include <unordered\_map>

#include "Key.h"

#include "Color.h"

class Window

{

public:

Window(const std::string& title, int w, int h);

~Window();

Window(const Window&) = delete;

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

bool Open() const;

void PollEvents();

void Display();

void Clear();

void PrintScreen(const std::string& file);

void RenderRectangle(int x, int y, int w, int h, Color color);

void MapKey(SDL\_Keycode sym, std::string str);

const Key& GetKey(std::string mapping);

SDL\_Surface\* NewSurface(int w, int h);

void DestroySurface(SDL\_Surface\* surface);

void Exit();

void SetPixel(SDL\_Surface \*surface, int x, int y, Uint32 pixel);

Uint32 ToPixel(Color color);

void RenderSurface(SDL\_Surface\* surface, int x, int y, int w, int h);

private:

void DrawRectangle(SDL\_Surface \*surface, int w, int h, Uint32 pixel);

bool saveScreenshotBMP(std::string filepath, SDL\_Window\* SDLWindow, SDL\_Renderer\* SDLRenderer);

void HandleEvent(SDL\_Event&);

int width\_;

int height\_;

SDL\_Window\* win\_;

SDL\_Renderer\* renderer\_;

bool open\_;

std::unordered\_map<std::string, Key> keys\_;

std::unordered\_map<SDL\_Keycode, std::string> keymaps\_;

};

#### Astar.cpp

#include "Astar.h"

Astar::Astar(Coord start, Coord end)

:start\_(start)

,end\_(end)

,current\_(start)

,steps\_(0)

{

}

bool Astar::Step(Maze& maze)

{

if (steps\_ == 0)

{

came\_from\_ = Maze(maze.Width(), maze.Height());

maze(start\_.x, start\_.y).color = Color::green();

maze(end\_.x, end\_.y).color = Color::red();

Reset();

}

++steps\_;

bool done = false;

if (open\_queue\_.size() != 0)

{

current\_ = open\_queue\_.top().second;

if (current\_ != end\_)

{

open\_queue\_.pop();

open\_set\_.erase(current\_);

closed\_set\_.insert(current\_);

Coord neighbours[] =

{

{ current\_.x - 1, current\_.y }, { current\_.x + 1, current\_.y },

{ current\_.x, current\_.y - 1 }, { current\_.x, current\_.y + 1 }

};

for (auto& n : neighbours)

{

if (closed\_set\_.count(n) == 0 && !maze.IsBorderCoord(n.x, n.y) && maze(n.x, n.y).cost >= 0)

{

int tentative\_g = g\_[current\_] + maze(current\_.x, current\_.y).cost;

if (open\_set\_.count(n) == 0 || tentative\_g < g\_[n])

{

came\_from\_(n.x, n.y).cost = 1;

if (n != end\_)

{

maze(n.x, n.y).color = Color::blue();

}

if (open\_set\_.count(n) == 0)

{

parent\_of\_.insert(std::make\_pair(n, current\_));

g\_.insert(std::make\_pair(n, tentative\_g));

f\_.insert(std::make\_pair(n, Heuristic(n, end\_)));

open\_set\_.insert(n);

open\_queue\_.push(std::make\_pair(Heuristic(n, end\_), n));

}

else

{

parent\_of\_[n] = current\_;

g\_[n] = tentative\_g;

f\_[n] = Heuristic(n, end\_);

}

}

}

}

}

}

if (current\_ == end\_)

{

if (walker\_ != start\_)

{

walker\_ = parent\_of\_[walker\_];

if (walker\_ != end\_ && walker\_ != start\_)

{

path\_.push\_back(walker\_);

maze(walker\_.x, walker\_.y).color = Color::make\_from\_bytes(127, 127, 255);

}

}

else if (!path\_.empty())

{

if (steps\_ % 5 != 0)

{

return true;

}

Coord p = path\_.back();

maze(p.x, p.y).color = Color::make\_from\_bytes(127, 127, 255);

path\_.pop\_back();

if (!path\_.empty())

{

Coord p = path\_.back();

maze(p.x, p.y).color = Color::make\_from\_bytes(0, 255, 255);

}

}

else

{

return false;

}

}

return !done;

}

void Astar::Reset(Coord start, Coord end)

{

start\_ = start;

end\_ = end;

Reset();

}

void Astar::Reset()

{

for (int i = 0; i < came\_from\_.Height(); i++)

{

for (int j = 0; j < came\_from\_.Width(); ++j)

{

came\_from\_(j, i).cost = -1;

}

}

current\_ = start\_;

walker\_ = end\_;

steps\_ = 0;

g\_.clear();

f\_.clear();

path\_.clear();

while (!open\_queue\_.empty())

{

open\_queue\_.pop();

}

closed\_set\_.clear();

open\_set\_.clear();

g\_.insert(std::make\_pair(start\_, 0));

f\_.insert(std::make\_pair(start\_, Heuristic(start\_, end\_)));

open\_queue\_.push(std::make\_pair(Heuristic(start\_, end\_), start\_));

open\_set\_.insert(start\_);

parent\_of\_.clear();

}

int Astar::Heuristic(Coord a, Coord b)

{

//Manhattan distance

return std::abs((a.x - b.x)) + std::abs((a.y - b.y));

#### Maze.cpp

#include "Maze.h"

#include <algorithm>

#include <cassert>

Maze::Maze(int w, int h)

:w\_(w)

,h\_(h)

,maze\_(w \* h)

,surface\_(nullptr)

{

}

Maze::Maze()

:w\_(0)

, h\_(0)

,surface\_(nullptr)

{}

Maze::~Maze()

{

if (surface\_ != nullptr)

{

SDL\_FreeSurface(surface\_);

}

}

Maze::Maze(const Maze& other)

{

InnerCopy(other);

}

Maze& Maze::operator=(const Maze& other)

{

if (&other != this)

{

InnerCopy(other);

}

return \*this;

}

void Maze::InnerCopy(const Maze& maze)

{

w\_ = maze.w\_;

h\_ = maze.h\_;;

std::swap(maze\_, std::vector<Tile>(h\_ \* w\_));

surface\_ = nullptr;

}

void Maze::Render(Window& window, const int x, const int y, const int w, const int h)

{

if (surface\_ == nullptr)

{

surface\_ = window.NewSurface(w\_, h\_);

}

int tile\_w = w / w\_;

int tile\_h = h / h\_;

for (int i = 0; i < h\_; ++i)

{

for (int j = 0; j < w\_; ++j)

{

window.SetPixel(surface\_, j, i, window.ToPixel(Get(j, i).color));

}

}

window.RenderSurface(surface\_, x, y, w, h);

}

Maze::Tile& Maze::operator()(int x, int y)

{

return Get(x, y);

}

const Maze::Tile& Maze::operator()(int x, int y) const

{

return Get(x, y);

}

Maze::Tile& Maze::Get(int x, int y)

{

EnsureRange(x, y);

return maze\_[y \* w\_ + x];

}

const Maze::Tile& Maze::Get(int x, int y) const

{

EnsureRange(x, y);;

return maze\_[y \* w\_ + x];

}

bool Maze::IsBorderCoord(const int x, const int y) const

{

bool ret = (x <= 0 || y <= 0 || x >= w\_- 1 || y >= h\_ - 1);

return ret;

}

#### PrimGenerator.cpp

#include "PrimGenerator.h"

#include <cassert>

#include <random>

PrimGenerator::PrimGenerator(const Coord& start)

:steps\_(0)

,start\_(start.x + (1 - (start.x % 2)), start.x + (1 - (start.x % 2)))

{

std::random\_device rd;

random\_.Reseed(rd());

}

/\* returns true if there are more steps \*/

bool PrimGenerator::Step(Maze& maze)

{

if (steps\_ == 0)

{

Clear(maze);

TryAddWalls(start\_, maze);

maze(start\_.x, start\_.y).cost = 1;

maze(start\_.x, start\_.y).color = Color::white();

}

if (walls\_.size() == 0)

{

return false;

}

while (walls\_.size() != 0)

{

int rand = random\_.NextInt(0, walls\_.size() - 1);

Coord wall = walls\_[rand];

walls\_[rand] = walls\_.back();

walls\_.pop\_back();

auto horizontal = [this, &maze](const Coord& coord)

{

return(maze(coord.x, coord.y - 1).cost == -1 || maze(coord.x, coord.y + 1).cost == -1);

};

auto vertical = [this, &maze](const Coord& coord)

{

return(maze(coord.x - 1, coord.y).cost == -1 || maze(coord.x + 1, coord.y).cost == -1);

};

if (!vertical(wall) && !horizontal(wall))

{

continue;

}

Coord node;

bool vert = wall.x % 2 == 0;

if (vert && vertical(wall))

{

if (maze(wall.x - 1, wall.y).cost == -1)

{

node = Coord(wall.x - 1, wall.y);

}

else

{

node = Coord(wall.x + 1, wall.y);

}

}

else if (!vert && horizontal(wall))

{

if (maze(wall.x, wall.y - 1).cost == -1)

{

node = Coord(wall.x, wall.y - 1);

}

else

{

node = Coord(wall.x, wall.y + 1);

}

}

else

{

continue;

}

if (maze.IsBorderCoord(node.x, node.y))

{

continue;

}

maze(node.x, node.y).cost = 1;

maze(node.x, node.y).color = Color::white();

maze(wall.x, wall.y).cost = 1;

maze(wall.x, wall.y).color = Color::white();

TryAddWalls(node, maze);

break;

}

++steps\_;

return walls\_.size() != 0;

}

void PrimGenerator::Clear(Maze& maze)

{

for (int i = 0; i < maze.Height(); ++i)

{

for (int j = 0; j < maze.Width(); ++j)

{

auto& tile = maze(j, i);

tile.color = Color::black();

tile.cost = -1;

}

}

walls\_.clear();

}

void PrimGenerator::Reset(Maze& maze, const Coord& start)

{

start\_ = start;

steps\_ = 0;

}

void PrimGenerator::TryAddWalls(const Coord& coord, const Maze& maze)

{

Coord coords[4] =

{

{ coord.x - 1, coord.y }, { coord.x + 1, coord.y },

{ coord.x, coord.y - 1}, { coord.x, coord.y + 1 }

};

for (int i = 0; i < 4; ++i)

{

if (!maze.IsBorderCoord(coords[i].x, coords[i].y) && std::find(walls\_.begin(), walls\_.end(), coords[i]) == walls\_.end() && (maze(coords[i].x, coords[i].y).cost == -1))

{

walls\_.push\_back(coords[i]);

}

}

}

#### Random.cpp

#include "Random.h"

#include <cmath>

#include <limits>

#include <chrono>

const double Noise::PI = 3.14159265359;

Randomizer::Randomizer()

{

Reseed();

}

Randomizer::Randomizer(unsigned int seed)

{

Reseed(seed);

}

/\* Reseeds with current time. We can't seed with 0 until Microsoft fixes bug

\* Written: Sebastian Zander 2013-06-05 19:18

\*/

void Randomizer::Reseed()

{

unsigned int now = time(0);

engine\_.seed(now == 0 ? 1 : now);

}

/\* Note: We can't seed with 0 until Microsoft fixes bug, so if 0 is input,

\* it will seed with 1 instead

\* Written: Sebastian Zander 2013-06-05 19:18

\*/

void Randomizer::Reseed(unsigned int seed)

{

engine\_.seed(seed == 0 ? 1 : seed);

}

unsigned int Randomizer::NextInt(unsigned int max)

{

if (max == 0) return 0;

std::uniform\_int\_distribution<unsigned int> dist(0, max);

return dist(engine\_);

}

int Randomizer::NextInt(int min, int max)

{

if (max - min <= 0) return min;

std::uniform\_int\_distribution<int> dist(min, max);

return dist(engine\_);

}

double Randomizer::NextDouble(double min, double max)

{

if (max - min <= 0) return min;

std::uniform\_real\_distribution<double> dist(min, max);

return dist(engine\_);

}

//this Generates a double between 0 and 1, and if that double is lower than trueFactor, true is returned

bool Randomizer::NextBool(double true\_factor)

{

return NextDouble(0.f, 1.f) <= true\_factor;

}

unsigned int Randomizer::NextRand()

{

return engine\_();

}

double Noise::FindNoise(int x)

{

x = WrapAround(x);

int n = (x << 13) ^ x;

return (double)(1.0 - ((n \* (n \* n \* 15731 + 789221) + 1376312589) & 0x7fffffff) / 1073741824.0);

}

double Noise::FindNoise(int x, int y)

{

x = WrapAround(x);

y = WrapAround(y);

int n = x \* 331 + y \* 337;

n = (n << 13) ^ n;

return (double)(1.0 - ((n \* (n \* n \* 15731 + 789221) + 1376312589) & 0x7fffffff) / 1073741824.0);

}

double Noise::FindNoise(int x, int y, int z)

{

x = WrapAround(x);

y = WrapAround(y);

z = WrapAround(z);

//I'm guessing these are primes?

int n = x \* 331 + y \* 337 + z \* 339;

n = (n << 13) ^ n;

return (double)(1.0 - ((n \* (n \* n \* 15731 + 789221) + 1376312589) & 0x7fffffff) / 1073741824.0);

}

int Noise::WrapAround(int i)

{

int max = std::numeric\_limits<int>::max();

if (i < 0)

return max + i;

return i;

}

double Noise::Generate(double x)

{

double x1 = FindNoise(ceil(x));

double x2 = FindNoise(floor(x));

double mux = x - floorf(x);

double xNoise = Interpolate(x1, x2, mux);

return xNoise;

}

double Noise::Generate(double x, double y)

{

double s, t, u, v;

s = FindNoise(floor(x), floor(y));

t = FindNoise(ceil(x), floor(y));

u = FindNoise(floor(x), ceil(y));//Get the surrounding pixels to calculate the transition.

v = FindNoise(ceil(x), ceil(y));

double y1 = Interpolate(s, t, x - floorf(x));//Interpolate between the values.

double y2 = Interpolate(u, v, x - floorf(x));//Here we use x-floorx, to get 1st dimension. Don't mind the x-floorx thingie, it's part of the cosine formula.

return Interpolate(y1, y2, y - floorf(y));//Here we use y-floory, to get the 2nd dimension.

}

double Noise::Generate(double x, double y, double z)

{

double s, t, u, v, y1, y2;

//first plane

s = FindNoise(floor(x), floor(y), floor(z));

t = FindNoise(ceil(x), floor(y), floor(z));

u = FindNoise(floor(x), ceil(y), floor(z));//Get the surrounding pixels to calculate the transition.

v = FindNoise(ceil(x), ceil(y), floor(z));

y1 = Interpolate(s, t, x - floorf(x));//Interpolate between the values.

y2 = Interpolate(u, v, x - floorf(x));//Here we use x-floorx, to get 1st dimension. Don't mind the x-floorx thingie, it's part of the cosine formula.

double plane1 = Interpolate(y1, y2, y - floorf(y));

//second plane

s = FindNoise(floor(x), floor(y), ceil(z));

t = FindNoise(ceil(x), floor(y), ceil(z));

u = FindNoise(floor(x), ceil(y), ceil(z));//Get the surrounding pixels to calculate the transition.

v = FindNoise(ceil(x), ceil(y), ceil(z));

y1 = Interpolate(s, t, x - floorf(x));//Interpolate between the values.

y2 = Interpolate(u, v, x - floorf(x));//Here we use x-floorx, to get 1st dimension. Don't mind the x-floorx thingie, it's part of the cosine formula.

double plane2 = Interpolate(y1, y2, y - floorf(y));

return Interpolate(plane1, plane2, z - floorf(z));

}

double Noise::Interpolate(double y1, double y2, double mu)

{

double mu2 = (1 - cos(mu \* PI)) / 2;

return (y1 \* (1 - mu2) + y2 \* mu2);

}

int Noise::floor(double f)

{

return round<int>(floorf(f));

}

int Noise::ceil(double f)

{

return round<int>(ceilf(f));

}

double PerlinNoise::Generate(double x)

{

//properties of one octave (changing each loop)

double t = 0.0f;

double freq = frequency\_;

for (int k = 0; k < octaves\_; k++)

{

t += noise.Generate(x \* freq) \* amplitude\_;

amplitude\_ \*= lacunarity\_;

freq \*= 2;

}

return t;

}

double PerlinNoise::Generate(double x, double y)

{

//properties of one octave (changing each loop)

double t = 0.0f;

double freq = frequency\_;

for (int k = 0; k < octaves\_; k++)

{

t += noise.Generate(x \* freq, y \* freq) \* amplitude\_;

amplitude\_ \*= lacunarity\_;

freq \*= 2;

}

return t;

}

double PerlinNoise::Generate(double x, double y, double z)

{

//properties of one octave (changing each loop)

double t = 0.0f;

double freq = frequency\_;

for (int k = 0; k < octaves\_; k++)

{

t += noise.Generate(x \* freq, y \* freq, z \* freq) \* amplitude\_;

amplitude\_ \*= lacunarity\_;

freq \*= 2;

}

return t;

}

#### Main.cpp

#include <SDL.h>

#include "Window.h"

#include "Timer.h"

#include "Maze.h"

#include "PrimGenerator.h"

#include "Coord.h"

#include "Astar/Astar.h"

int astar\_main();

Coord RandomCoord(Randomizer& r, Maze& maze)

{

Coord ret;

ret = Coord(r.NextInt(1, maze.Width() - 3), r.NextInt(1, maze.Height() - 3));

while (ret.x % 2 == 0 ||ret.y % 2 == 0)

ret = Coord(r.NextInt(1, maze.Width() - 3), r.NextInt(1, maze.Height() - 3));

return ret;

}

int main(int argc, char \*argv[])

{

return astar\_main();

}

int astar\_main()

{

Window win("Maze solver!", 800, 800);

std::random\_device rd;

Randomizer rand(rd());

Coord start, end;

Maze maze(800, 800);

PrimGenerator generator(RandomCoord(rand, maze));

start = RandomCoord(rand, maze);

end = RandomCoord(rand, maze);

while (start == end)

end = RandomCoord(rand, maze);

Astar solver(start, end);

long long step = 0;

system("mkdir screens");

while (win.Open())

{

win.PollEvents();

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);

}

}

}

maze.Render(win, 0, 0, 800, 800);

win.Display();

step++;

win.Clear();

}

return 0;

}

### 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();

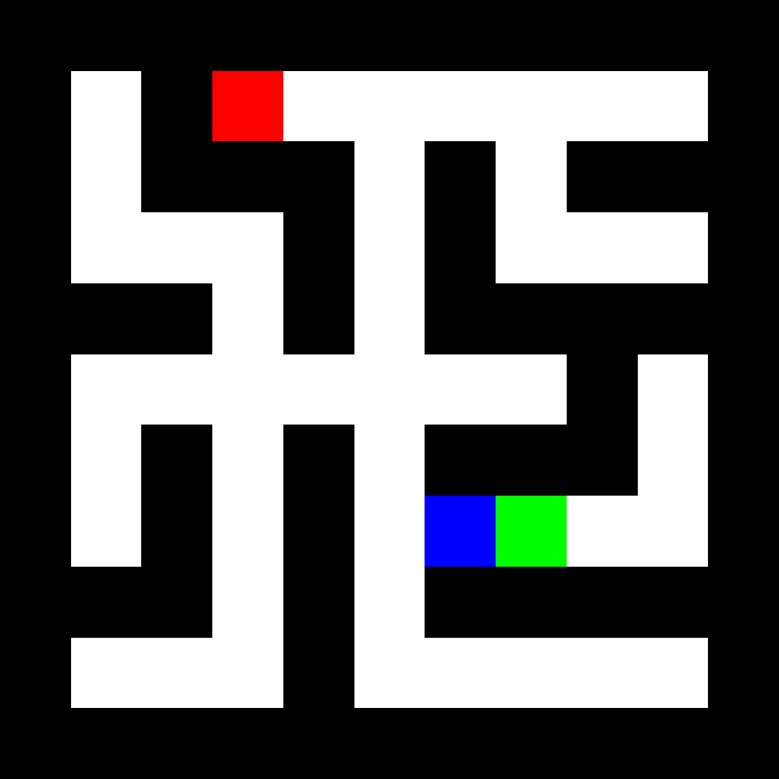
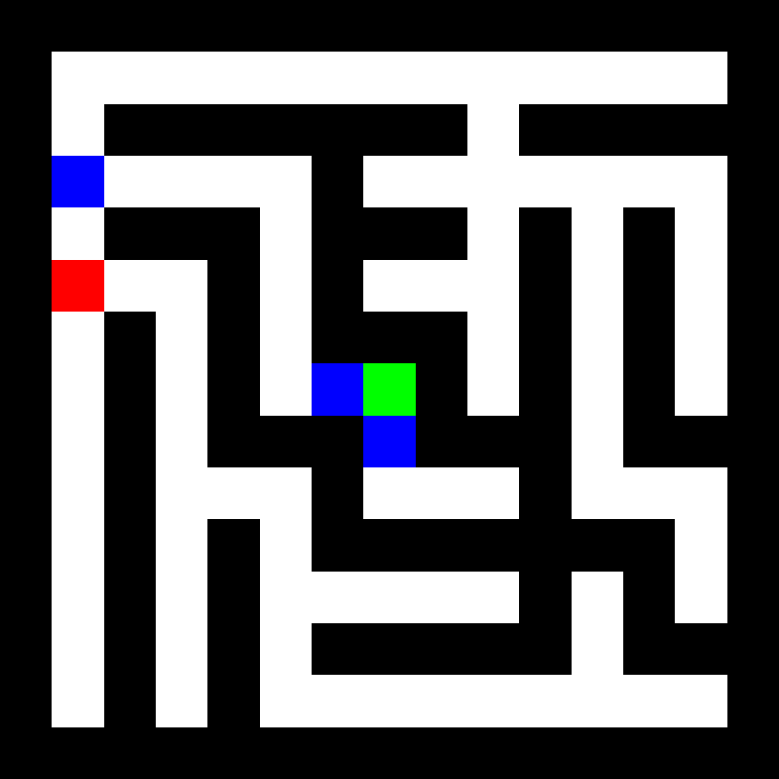
}

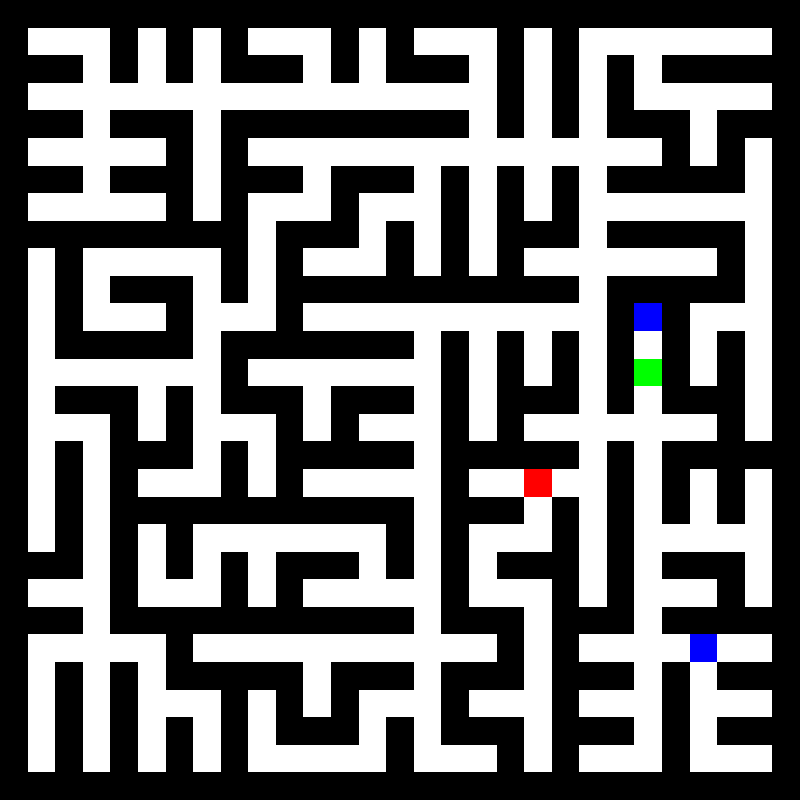
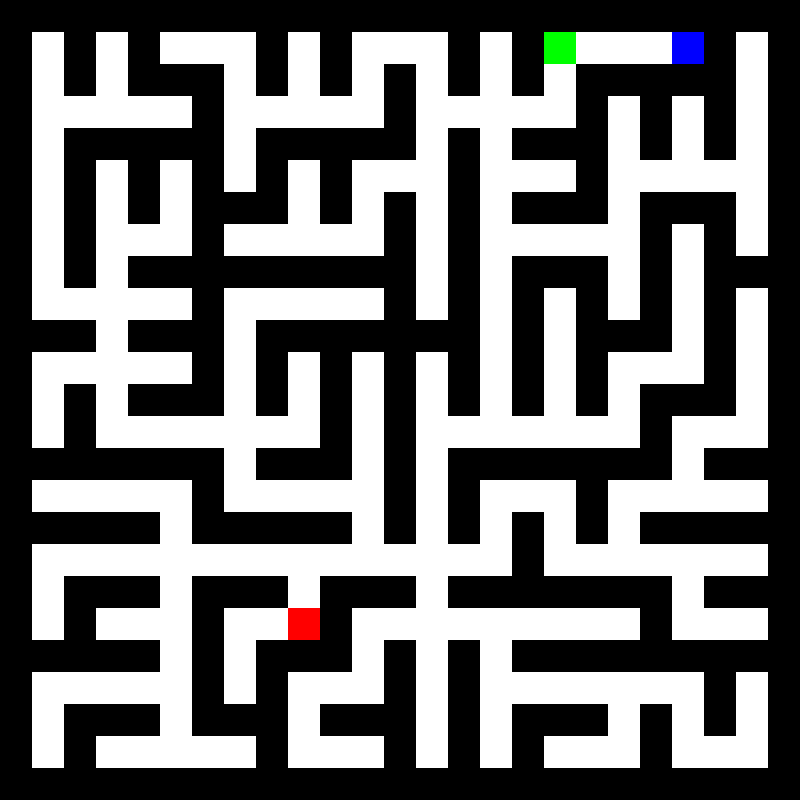
return 0;

}

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

### Problem

### Design

### Kod

#### <headerfilnamn>

#### <headerfilnamn>

#### <…>

#### <definitionsfilnamn>

#### <definitionsfilnamn>

#### <…>

### Körexempel

### Analys

## Skripttekniker

### Problem

I ett begränsat spelfält, låt robotar spela ett enkelt tafatt-spel. Olika robotar ska kunna ha olika beteenden och det ska gå enkelt att lägga till och ändra beteenden utan att kompilera om programmet.

### Design

### Kod

#### <headerfilnamn>

#### <headerfilnamn>

#### <…>

#### <definitionsfilnamn>

#### <definitionsfilnamn>

#### <…>

### Körexempel

### Analys

## Tillståndstekniker

### Problem

Problemet som ska lösas är att en mus i ett begränsat spelfält dels ska leta efter och äta ost om den är hungrig men också undvika katter, som jagar musen om den kommer för nära.

### Design

För att lösa problemet används en stackbaserad tillståndsmaskin. Eftersom att musen ska kunna sättas i flyende tillstånd oavsett vilket annat tillstånd den befinner sig i och samtidigt kunna gå tillbaka till samma tillstånd igen behöver tillståndsmaskinen vara stackbaserad. Figuren nedan illustrerar denna tillståndsmaskin.

### 

Figur 3. Diagram för musens tillståndståndsmaskin.

### Kod

#### <headerfilnamn>

#### <headerfilnamn>

#### <…>

#### <definitionsfilnamn>

#### <definitionsfilnamn>

#### <…>

### Körexempel

### Analys

# Applikation

## Problem

I den här rapporten är problemet för applikationen att skapa en AI som kan spela tetris.

I klassiskt testris kontrollerar spelare en fallande spelpjäs, en s.k. tetromino, av olika former. Det finns 7 olika former vilka är namngedda efter hur de ser ut:



Figur 4. Från vänster till höger: I, J, L, O, S, T och Z.

När pjäser landar fastnar dem på banan och en ny pjäs faller. Vilket pjäs som faller är slumpmässigt, men spelare kan alltid se vilket pjäs som kommer falla efter den pjäs spelaren för tillfället kontrollerar. Banan består av celler, 10 på bredden och 20 på höjden. När spelare har fyllt en rad rensas den raden och alla rader ovanför faller ner en rad. Spelare kan som max rensa 4 rader per fallande pjäs, detta eftersom att den längsta pjäsen (I) är 4 rader lång. Spelaren får mer poäng för att rensa fler rader åt gången. Var tionde rad ökas nivån på spelet och pjäser faller snabbare. Nivån påverkar också hur många poäng spelaren får för radrensning:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Nivå | 1 rad | 2 rader | 3 rader | 4 rader |
| *n* | 40 \* (n + 1) | 100 \* (n + 1) | 300 \* (n + 1) | 1200 \* (n + 1) |

## Design

Designen är två-fald. Dels ska tetris-spelet byggas, och dels ska en AI som kan lösa den tetris-implementationen byggas.

För att AI:n ska kunna spela spelet kan den ta reda på vilken pjäs som faller för tillfället, samt se framtida pjäser (det finns ingen gräns på hur många pjäser som går att få tag på annat än för minnet). AI:n kan styra pjäsen genom att skicka in händelser till tetris-implementationen.

Varje gång en ny tetromino faller söker AI:n igenom alla möjliga tillstånd som orskar ett lås (att tetrominon ”fastnar”) på tetris-planen. Den söker därefter igenom alla möjliga tillstånd för nästa tetromino, och bygger på så sätt upp ett träd av möjliga tillstånd som är n djupt, där n är hur många tetrominon som spelet förhandsvisar plus den tetromino som faller på spelplanen. I teorin kan n vara hur djupt som helst, men är i den här lösningen begränsad till 2: den som fallar, och en som förhandsvisas.

Från detta har AI:n ett antal möjliga tillstånd. Av dessa tillstånd väljer AI:n ut en som ska användas, vilket beror på ett antal evalueringsfunktioner som på något sätt analyserar resultatet av detta tillståndet på spelplanen. Resultatet av dessa funktioner är ett realtal, och alla resultat adderas. Det tillstånd som ger lägst värde är det som väljs ut av AI:n. AI:n kan sedan visualisera detta resultat, genom att flytta tetiminon till rätt position.

Varje evalueringsfunktion har en vikt associerad till sig. Dessa vikter kan tas bland annat skrivas för hand eller så kan de tas fram med hjälp av en evolutionär teknik. I den här lösningen tas de fram med hjälp av en evolutionär teknik. Den populationen sparas också till en fil, så att programmet alltid kan fortsätta den evolutionära processesen, vilket gör det enklare att köra processen under en längre tid och på så sätt – förhoppningsvis – få bättre resultat. Detta betyder också att AI:n aldrig blir klar, den blir bara bättre och bättre –även detta en förhoppning.

[VILKA ÄR EVALUERINGSFUNKTIONERA!?]

Generna i den evolutionära teknik som används är helt enkelt vikterna. Mutation av genomet görs genom att slumpmässigt variera vikterna, detta görs genom att låta den nya vikten få ett slumpat värde i ett interval run det gamla värdet (detta för att ge en gräns för hur stora eller små värdena kan bli, så att värdena inte kan bli godtyckligt höga/låga). Vid överkorsning skyfflas dessa vikter mellan två olika individer. En AI kan skapas med dessa vikter och spela ett spel. När varje individ i populationen har spelat ett spel används dess poäng som fitnessfunktion och en ny population skapas från den gamla.

## Kod

### <headerfilnamn>

### <headerfilnamn>

### <…>

### <definitionsfilnamn>

### <definitionsfilnamn>

### <…>

## Körexempel

## Analys

Att använda en evolutionär algoritm för att bestämma vikterna för evalueringsfunktionen ger fördelen att dessa vikter inte behöver skrivas in manuellt, något som hade varit svårt att göra då det borde vara svårt, om inte omöjligt, att veta vilka vikter som vore optimala.

# Slutsats