# Program Code

## Simulator.cs

using System;

using System.IO;

using System.Collections.Generic;

using System.Drawing;

using System.Linq;

using System.Windows.Forms;

using System.Text.RegularExpressions;

namespace TuringMachine

{

public partial class Simulator : Form

{

List<TapePanel> TapeList = new List<TapePanel>();

MachineData currentMachineData = new MachineData();

TabObj currentTab;

char defaultCaseKey = '\*';

int fullSpeedInterval = 50;

//Width of tape at which font size is changed

const int maxGraphicSize = 975;

//Constructor

public Simulator()

{

InitializeComponent();

//Load user preferences from config.ini

LoadSettings();

}

#region INPUT TABS

//Adding blank tab

public void Tab\_AddTab(object sender, EventArgs e)

{

// Instantiate new tab

TabObj tabObj = new TabObj(this)

{

Tag = new MachineData(0, 1,

new List<string>() { "" },

new List<int>() { 0 },

new List<string>() { "" })

};

//Set to active

InputWindow.SelectedTab = tabObj;

currentTab = tabObj;

currentMachineData = (MachineData)currentTab.Tag;

//Update GUI data

Tag\_SetTapes(1);

Tag\_SetTape(

new List<string>() { "" },

new List<int>() { 0 },

new List<string>() { "" }, 0);

Tag\_SetMachine(0);

}

//Adding from file

public void Tab\_AddTab(string Text, int Machine, int Tapes, List<int> point,

List<string> Tape, List<string> State, string filename)

{

//Instantiate new tab

TabObj tabObj = new TabObj(this)

{

Text = filename,

Tag = new MachineData(Machine, Tapes, Tape, point, State),

};

tabObj.GetText().Text = Text;

//Set to active

InputWindow.SelectedTab = tabObj;

currentTab = tabObj;

currentMachineData = (MachineData)currentTab.Tag;

//Update GUI data

Tag\_SetTapes(Tapes);

Tag\_SetMachine(Machine);

Tag\_SetTape(Tape, point, State, Machine);

}

//Adding by duplication

public void Tab\_AddTab(string Text, MachineData data)

{

string code = currentTab.GetText().Text;

//Set name of tab

string name = currentTab.Text;

int numberOfPreviousCopies = 0;

foreach (TabPage tabPage in InputWindow.Controls)

{

if (tabPage.Text.Contains(name))

{

numberOfPreviousCopies++;

}

}

//Instantiate duplicated tab

MachineData newData = new MachineData(data);

TabObj tabObj = new TabObj(this)

{

Text = name + "(" + (numberOfPreviousCopies).ToString() + ")",

Tag = newData

};

tabObj.GetText().Text = code;

//Set to active

InputWindow.SelectedTab = tabObj;

currentTab = tabObj;

currentMachineData = (MachineData)currentTab.Tag;

//Update GUI data

Tag\_SetMachine(data.GetMachine());

Tag\_SetTapes(data.GetTapes());

Tag\_SetTape(data.GetTape(), data.GetPointer(),

data.GetInitialState(), data.GetMachine());

}

//Deletes current tab

public void Tab\_RemoveTab(object sender, EventArgs e)

{

int position = InputWindow.SelectedIndex - 1;

//Remove from controls

InputWindow.Controls.Remove(currentTab);

//Set new tab

if (position < 0)

{

InputWindow.SelectedIndex = 0;

}

else

{

InputWindow.SelectedIndex = position;

}

//Manually remove tapes

if (InputWindow.Controls.Count == 0)

{

foreach (TapePanel tape in TapeList)

{

CloseTape(tape);

}

TapeList = new List<TapePanel>();

}

}

//Saves to path dictated by user

public void Tab\_SaveTab(object sender, EventArgs e)

{

if (InputWindow.Controls.Count > 0)

{

if (CheckMachine())

{

//Dialog config

SaveFileDialog sf = new SaveFileDialog

{

FileName = currentTab.Text,

Filter = "Text File|\*.txt",

Title = "Save Turing Machine"

};

sf.ShowDialog();

MachineData machine = currentMachineData;

if (sf.FileName != "")

{

using (StreamWriter sw = new StreamWriter(File.Create(sf.FileName)))

{

//Write machine type

sw.Write(machine.GetMachine() + Environment.NewLine);

if (machine.GetMachine() == 3 ||

machine.GetMachine() == 4 ||

machine.GetMachine() == 5)

{

//Write tape count

sw.Write(machine.GetTapes() + Environment.NewLine);

}

//Write tape data

for (int i = 0; i < machine.GetTapes(); i++)

{

if (machine.GetTape(i) == "")

{

sw.WriteLine(" ");

}

else

{

sw.WriteLine(machine.GetTape(i));

}

//Write pointer

sw.WriteLine(machine.GetPointer(i));

if (machine.GetMachine() == 4)

{

//Write initial state

sw.WriteLine(machine.GetInitialState(i));

}

}

if (machine.GetMachine() != 4)

{

//Write initial state

sw.WriteLine(machine.GetInitialState(0));

}

//Write transition functions

foreach (string line in (currentTab.GetText()).Lines)

{

sw.WriteLine(line);

}

sw.Close();

}

}

}

}

else

{

MessageBox.Show("No machines to save.");

}

}

public void Tab\_OpenTab(object sender, EventArgs e)

{

//Dialog config

OpenFileDialog sf = new OpenFileDialog

{

Filter = "Text File|\*.txt",

Title = "Open Turing Machine"

};

sf.ShowDialog();

if (sf.FileName != "")

{

try

{

string contents = "";

//Open file

StreamReader sr = new StreamReader(File.OpenRead(sf.FileName));

//Get contents

contents = sr.ReadToEnd();

List<string> lines = contents.Split(new char[] { '\r', '\n' }, StringSplitOptions.RemoveEmptyEntries).ToList<string>();

//Set machine

int Machine = Convert.ToInt32(lines[0]);

int Tapes;

List<string> Tape = new List<string>();

List<int> Pointer = new List<int>();

List<string> InitialState = new List<string>();

string Script;

switch (Machine)

{

case 0:

case 1:

case 2:

//Set tape

if (lines[1] != " ")

{

Tape.Add(lines[1]);

}

else

{

Tape.Add("");

}

//Set tapes

Tapes = 1;

//Set pointer

Pointer.Add(Convert.ToInt32(lines[2]));

//Set initial state

InitialState.Add(lines[3]);

lines.RemoveRange(0, 4);

//Set transition code

Script = "";

for (int i = 0; i < lines.Count; i++)

{

Script += lines[i] + Environment.NewLine;

}

//Create machine in program

Tab\_AddTab(Script, Machine, Tapes, Pointer, Tape,

InitialState, sf.SafeFileName.Replace(".txt", ""));

sr.Close();

break;

case 3:

case 5:

//Set tape count

Tapes = Convert.ToInt32(lines[1]);

//Set tapes and pointers

for (int i = 2; i < 2 \* Tapes + 1; i = i + 2)

{

if (lines[i] != " ")

{

Tape.Add(lines[i]);

}

else

{

Tape.Add("");

}

Pointer.Add(Convert.ToInt32(lines[i + 1]));

}

//Set initial state

InitialState.Add(lines[2 + 2 \* Tapes]);

//Set transition code

Script = "";

for (int i = 3 + 2 \* Tapes; i < lines.Count; i++)

{

Script += lines[i] + Environment.NewLine;

}

//Create machine in program

Tab\_AddTab(Script, Machine, Tapes, Pointer, Tape,

InitialState, sf.SafeFileName.Replace(".txt", ""));

sr.Close();

break;

case 4:

//Set tape count

Tapes = Convert.ToInt32(lines[1]);

//Set tapes, pointers and initial states

for (int i = 0; i < Tapes \* 3; i = i + 3)

{

if (lines[i + 2] != " ")

{

Tape.Add(lines[i + 2]);

}

else

{

Tape.Add("");

}

Pointer.Add(Convert.ToInt32(lines[i + 3]));

InitialState.Add(lines[i + 4]);

}

Script = "";

//Set transition code

for (int i = 2 + 3 \* Tapes; i < lines.Count; i++)

{

Script += lines[i] + Environment.NewLine;

}

//Create machine in program

Tab\_AddTab(Script, Machine, Tapes, Pointer, Tape,

InitialState, sf.SafeFileName.Replace(".txt", ""));

sr.Close();

break;

default:

throw new Exception();

}

}

catch (Exception)

{

MessageBox.Show("Error opening text file.");

}

}

}

//Does same task as previous, just opens through a pre-determined path

public void Tab\_OpenTab(string path)

{

if (path != "")

{

try

{

string contents = "";

StreamReader sr = new StreamReader(File.OpenRead(path + ".txt"));

contents = sr.ReadToEnd();

List<string> lines = contents.Split(new char[] { '\r', '\n' }, StringSplitOptions.RemoveEmptyEntries).ToList<string>();

int Machine = Convert.ToInt32(lines[0]);

int Tapes;

List<string> Tape = new List<string>();

List<int> Pointer = new List<int>();

List<string> InitialState = new List<string>();

string Script;

switch (Machine)

{

case 0:

case 1:

case 2:

Tape.Add(lines[1]);

Tapes = 1;

Pointer.Add(Convert.ToInt32(lines[2]));

InitialState.Add(lines[3]);

lines.RemoveRange(0, 4);

Script = "";

for (int i = 0; i < lines.Count; i++)

{

Script += lines[i] + Environment.NewLine;

}

Tab\_AddTab(Script, Machine, Tapes, Pointer, Tape,

InitialState, path.Replace(".txt", ""));

sr.Close();

break;

case 3:

case 5:

Tapes = Convert.ToInt32(lines[1]);

for (int i = 2; i < 2 \* Tapes + 1; i = i + 2)

{

if (lines[i] != " ")

{

Tape.Add(lines[i]);

}

else

{

Tape.Add("");

}

Pointer.Add(Convert.ToInt32(lines[i + 1]));

}

InitialState.Add(lines[2 + 2 \* Tapes]);

Script = "";

for (int i = 3 + 2 \* Tapes; i < lines.Count; i++)

{

Script += lines[i] + Environment.NewLine;

}

Tab\_AddTab(Script, Machine, Tapes, Pointer, Tape,

InitialState, path);

sr.Close();

break;

case 4:

Tapes = Convert.ToInt32(lines[1]);

for (int i = 0; i < Tapes \* 3; i = i + 3)

{

Tape.Add(lines[i + 2]);

Pointer.Add(Convert.ToInt32(lines[i + 3]));

InitialState.Add(lines[i + 4]);

}

Script = "";

for (int i = 2 + 3 \* Tapes; i < lines.Count; i++)

{

Script += lines[i] + Environment.NewLine;

}

Tab\_AddTab(Script, Machine, Tapes, Pointer, Tape,

InitialState, path);

sr.Close();

break;

}

}

catch (Exception)

{

MessageBox.Show("Error opening text file.");

}

}

}

//Opens example TM from Debug path

private void Tab\_OpenExample(object sender, EventArgs e)

{

string file = ((ToolStripMenuItem)sender).Text;

switch (file)

{

case "Addition":

Tab\_OpenTab("add");

break;

case "Subtraction":

Tab\_OpenTab("subtract");

break;

case "Palindrome Check":

Tab\_OpenTab("palindrome");

break;

case "Parity Calculator":

Tab\_OpenTab("paritybit");

break;

}

}

//Duplicate tab

public void Tab\_Duplicate(object sender, EventArgs e)

{

Tab\_AddTab(currentTab.GetText().Text, currentMachineData);

}

//Open rename form and rename tab

private void Tab\_Rename(object sender, EventArgs e)

{

string previousName = currentTab.Text;

RenameWindow form = new RenameWindow();

DialogResult dialogResult = form.ShowDialog();

string name = form.Controls["textBox"].Text;

if (name != "" && dialogResult == DialogResult.OK)

{

currentTab.Text = name;

}

}

//Removes any data from clipboard that is not text before pasting

public void Tab\_Paste(object sender, KeyEventArgs e)

{

if (e.Control && e.KeyCode == Keys.V)

{

if (Clipboard.ContainsText())

{

//Get Text from clipboard

string text = (string)Clipboard.GetData("Text");

//Set clipboard to text from original clipboard

Clipboard.SetText(text, TextDataFormat.Text);

}

//If no valid data, don't paste anything

else

{

e.Handled = true;

}

}

}

#endregion

#region SET UI FROM TAGS DURING CREATION

//Trims imported tapes

//Produce read/write head text for imported tapes

//Gets state list for tapes

private void Tag\_SetTape(List<string> Tape, List<int> Pointers,

List<string> States, int Machine)

{

int i = 0;

foreach (TapePanel tape in TapeList)

{

if (i < Tape.Count)

{

//Removes unnecessary spaces from tape

Tape tapeTrim = new Tape(Tape[i], Pointers[i]);

tapeTrim.TrimEdges();

tape.GetTape().Text = tapeTrim.GetTape();

//Set read/write head

tape.GetPointer().Text = FillPointerLine(Pointers[i], Tape[i].Length);

//Set drop-down box values for initial states

UI\_UpdateStates(tape.GetInitialState(), currentTab.GetText().Text);

if (Machine != 4)

{

tape.GetInitialState().Text = States[0];

}

else

{

tape.GetInitialState().Text = States[i];

}

i++;

}

}

}

//Sets the machine UI drop-down to match new machine

//Calls UI\_UpdateMachine to show/hide other relevant controls.

private void Tag\_SetMachine(int value)

{

if (InputWindow.Controls.Count > 0)

{

//Set index of ComboBox

currentMachine.SelectedIndex = value;

//Update which controls are shown by tape objects

UI\_UpdateMachine(currentMachine, null);

}

else

{

MessageBox.Show("No machine to change.");

}

}

//Instantiates the number of tapes required by the new machine

private void Tag\_SetTapes(int Tapes)

{

int count = 0;

count = TapeList.Count;

//Adds new tape objects until enough are present

while (count < Tapes)

{

if (count < Tapes)

{

AddTape();

}

count = TapeList.Count;

}

SetHeight(TapeList.Count);

}

#endregion

#region UPDATE UI POST-CREATION

#region OPTIONS PANEL FIELDS

//Shows which GUI elements need to be shown or hidden,

//based on what machine variant is chosen.

private void UI\_UpdateMachine(object sender, EventArgs e)

{

if (InputWindow.Controls.Count > 0)

{

currentMachineData.SetMachine(currentMachine.SelectedIndex);

for (int i = 0; i < currentMachineData.GetTapes(); i++)

{

if (currentMachineData.GetMachine() == 4)

{

TapeList[i].GetInitialState().Show();

TapeList[i].GetInitialStateLabel().Show();

TapeList[i].GetCurrentStateLabel().Show();

TapeList[i].GetStepLabel().Show();

steps.Hide();

currentstate.Hide();

Add.Show();

if (i == 0)

{

TapeList[i].GetCloseButton().Hide();

}

else

{

TapeList[i].GetCloseButton().Show();

}

}

else if (currentMachineData.GetMachine() == 3)

{

TapeList[i].GetCurrentStateLabel().Hide();

TapeList[i].GetStepLabel().Hide();

steps.Show();

currentstate.Show();

Add.Show();

if (i == 0)

{

TapeList[i].GetInitialState().Show();

TapeList[i].GetInitialStateLabel().Show();

TapeList[i].GetCloseButton().Hide();

}

else

{

TapeList[i].GetInitialState().Hide();

TapeList[i].GetInitialStateLabel().Hide();

TapeList[i].GetCloseButton().Show();

}

}

else

{

TapeList[i].GetCurrentStateLabel().Hide();

TapeList[i].GetStepLabel().Hide();

steps.Show();

currentstate.Show();

if (currentMachineData.GetMachine() == 5)

{

if (currentMachineData.GetTapes() != 3)

{

if (currentMachineData.GetTapes() < 3)

{

while (currentMachineData.GetTapes() < 3)

{

AddTape(sender, e);

}

}

else

{

while (currentMachineData.GetTapes() > 3)

{

CloseTape(TapeList[TapeList.Count - 1]);

currentMachineData.CloseTape(TapeList.Count - 1);

}

}

}

TapeList[i].GetCloseButton().Hide();

if (i == 0)

{

TapeList[0].GetInitialState().Show();

TapeList[0].GetInitialStateLabel().Show();

}

else

{

TapeList[i].GetInitialState().Hide();

TapeList[i].GetInitialStateLabel().Hide();

}

}

else

{

if (currentMachineData.GetTapes() > 1)

{

if (i == 0)

{

TapeList[0].GetInitialState().Show();

TapeList[0].GetInitialStateLabel().Show();

}

else

{

TapeList[i].GetInitialState().Hide();

TapeList[i].GetInitialStateLabel().Hide();

}

TapeList[i].GetCloseButton().Show();

}

else

{

TapeList[i].GetInitialState().Show();

TapeList[i].GetInitialStateLabel().Show();

TapeList[i].GetCloseButton().Hide();

}

}

Add.Hide();

}

}

if (currentMachineData.GetMachine() == 5)

{

generateuniversal.Show();

}

else

{

generateuniversal.Hide();

}

}

else

{

if (currentMachine.SelectedIndex != -1)

{

MessageBox.Show("No machine to change.");

currentMachine.SelectedIndex = -1;

}

}

}

//Updates time interval for run-timer.

private void UI\_SetSpeed(object sender, EventArgs e)

{

if (runTimer.Enabled)

{

runTimer.Stop();

runTimer.Interval = speedSlider.Value;

speedLabel.Text = speedSlider.Value.ToString();

runTimer.Start();

}

else

{

runTimer.Interval = speedSlider.Value;

speedLabel.Text = speedSlider.Value.ToString();

}

}

#endregion

#region TAPE PANEL FIELDS

//Update initial state ComboBox when changing machines

private void UI\_UpdateStates(ComboBox cb, string Text)

{

List<string> States = new List<string>();

string[] Lines = Text.Split(new char[] { '\r', '\n' }, StringSplitOptions.RemoveEmptyEntries);

foreach (string line in Lines)

{

string[] words = line.Split(' ');

if (!States.Contains(words[0]))

{

if (words[0].Length >= 2)

{

if (words[0][0] != '/' && words[0][1] != '/')

{

States.Add(words[0]);

}

}

else

{

States.Add(words[0]);

}

}

}

while (cb.Items.Count > 0)

{

cb.Items.RemoveAt(0);

}

foreach (string str in States)

{

cb.Items.Add(str);

}

if (!cb.Items.Contains(cb.Text))

{

cb.Text = "";

}

}

//Update initial state ComboBox when writing transition code

private void UI\_UpdateStates(object sender, KeyEventArgs e)

{

if (InputWindow.Controls.Count != 0)

{

List<string> States = new List<string>();

string[] Lines = currentTab.GetText().Lines;

foreach (string line in Lines)

{

if (line != "")

{

string[] words = line.Split(' ');

if (!States.Contains(words[0]))

{

if (words[0].Length >= 2)

{

if (words[0][0] != '/' && words[0][1] != '/')

{

States.Add(words[0]);

}

}

else

{

States.Add(words[0]);

}

}

}

}

foreach (TapePanel tape in TapeList)

{

ComboBox cb = (ComboBox)tape.GetInitialState();

while (cb.Items.Count > 0)

{

cb.Items.RemoveAt(0);

}

foreach (string str in States)

{

cb.Items.Add(str);

}

int matchingIndex = TapeList.FindIndex(i => i == tape);

cb.Text = (currentMachineData).GetInitialState(matchingIndex);

if (!cb.Items.Contains(cb.Text))

{

cb.Text = "";

}

}

}

}

//Fix tape font size when editing tape

public void UI\_UpdateTape(object sender, EventArgs e)

{

if (InputWindow.Controls.Count > 0)

{

string font = "Consolas";

TextBox text = (TextBox)sender;

TextBox pointer = ((TapePanel)text.Parent).GetPointer();

float initialFontSize = text.Font.Size;

PaintEventArgs pe = new PaintEventArgs(text.CreateGraphics(), text.DisplayRectangle);

SizeF tapeLength = pe.Graphics.MeasureString(text.Text, text.Font);

if (tapeLength.Width > maxGraphicSize)

{

while (tapeLength.Width > maxGraphicSize)

{

tapeLength = pe.Graphics.MeasureString(text.Text, text.Font);

if (tapeLength.Width > maxGraphicSize)

{

initialFontSize -= 0.1f;

text.Font = new Font(font, initialFontSize);

pointer.Font = new Font(font, initialFontSize);

}

}

}

else if (tapeLength.Width < maxGraphicSize && initialFontSize < 15.0f)

{

while (tapeLength.Width < maxGraphicSize)

{

tapeLength = pe.Graphics.MeasureString(text.Text, new Font(font, initialFontSize + 0.1f));

if (tapeLength.Width < maxGraphicSize && initialFontSize + 0.1f <= 15.0f)

{

initialFontSize += 0.1f;

text.Font = new Font(font, initialFontSize);

pointer.Font = new Font(font, initialFontSize);

if (initialFontSize == 15.0f)

{

pointer.Font = new Font(font, 15.0f);

break;

}

}

}

}

int matchingIndex = TapeList.FindIndex(i => i == (Panel)((TextBox)sender).Parent);

if (!runTimer.Enabled)

{

if (!Machine.isCompiled())

{

currentMachineData.SetTape(TapeList[matchingIndex].GetTape().Text, matchingIndex);

UI\_UpdatePointer(TapeList[matchingIndex], currentMachineData.GetPointer(matchingIndex), 0);

}

}

}

}

//Update tag value of initial state

public void UI\_UpdateInitialState(object sender, EventArgs e)

{

if (!runTimer.Enabled)

{

int matchingIndex = TapeList.FindIndex(i => i == (Panel)((ComboBox)sender).Parent);

if (currentMachineData.GetMachine() == 4)

{

currentMachineData.SetInitialState((string)((ComboBox)sender).SelectedItem, matchingIndex);

}

else

{

currentMachineData.SetInitialState((string)((ComboBox)sender).SelectedItem, 0);

}

}

}

//Update read/write head when changing machines or using arrow buttons

private void UI\_UpdatePointer(TapePanel tape, int Pointer, int increment)

{

TextBox pointerTB = tape.GetPointer();

if (InputWindow.Controls.Count > 0)

{

if (!runTimer.Enabled)

{

pointerTB.Text = "";

Pointer = Pointer + increment;

pointerTB.Text = FillPointerLine(Pointer, tape.GetTape().Text.Length);

}

}

}

//Move read/write head right

public void UI\_PointerAdd(object sender, EventArgs e)

{

int index = TapeList.FindIndex(i => i == ((Button)sender).Parent);

UI\_UpdatePointer((TapePanel)((Button)sender).Parent, currentMachineData.GetPointer(index), 1);

currentMachineData.SetPointer(currentMachineData.GetPointer(index) + 1, index);

}

//Move read/write head left

public void UI\_PointerSubtract(object sender, EventArgs e)

{

int index = TapeList.FindIndex(i => i == ((Button)sender).Parent);

UI\_UpdatePointer((TapePanel)((Button)sender).Parent, currentMachineData.GetPointer(index), -1);

currentMachineData.SetPointer(currentMachineData.GetPointer(index) - 1, index);

}

#endregion

//Changes UI controls to match newly selected machine

private void ChangeMachine(object sender, EventArgs e)

{

if (InputWindow.Controls.Count != 0)

{

currentTab = (TabObj)InputWindow.SelectedTab;

currentMachineData = (MachineData)currentTab.Tag;

//Reset tapes

foreach (TapePanel tape in TapeList)

{

CloseTape(tape);

}

TapeList.RemoveRange(0, TapeList.Count);

//Add new tapes

for (int i = 0; i < currentMachineData.GetTapes(); i++)

{

AddTape();

}

//Refresh height

SetHeight(TapeList.Count);

//Update UI settings

Tag\_SetMachine(currentMachineData.GetMachine());

for (int i = 0; i < currentMachineData.GetTapes(); i++)

{

TapeList[i].GetTape().Text = currentMachineData.GetTape(i);

UI\_UpdatePointer(TapeList[i], currentMachineData.GetPointer(i), 0);

ComboBox initialState = TapeList[i].GetInitialState();

UI\_UpdateStates(initialState, currentTab.GetText().Text);

UI\_UpdateMachine(sender, e);

if (currentMachineData.GetMachine() == 3)

{

initialState.Text = currentMachineData.GetInitialState(0);

//UI\_SetStates(initialState,

//currentMachineData.GetInitialState(0));

}

else

{

initialState.Text = currentMachineData.GetInitialState(i);

//UI\_SetStates(initialState,

//currentMachineData.GetInitialState(i));

}

}

}

}

private int SetHeight(int Tapes)

{

int h;

Height = 500 + 75 \* (Tapes - 1);

h = Height;

return h;

}

#endregion

#region TAPE INSTANCING

//Remove tape without removing data

private void CloseTape(TapePanel item)

{

foreach (Control i in item.Controls)

{

item.Controls.Remove(i);

}

item.Parent.Controls.Remove(item);

}

//Remove tape and its data from tag

public void CloseTape(object sender, EventArgs e)

{

if (TapeList.Count != 1)

{

Control sen = (Control)sender;

int matchingIndex = 0;

Control item = sen.Parent;

matchingIndex = TapeList.FindIndex(i => i == (TapePanel)item);

foreach (Control i in item.Controls)

{

item.Controls.Remove(i);

}

item.Parent.Controls.Remove(item);

TapeList.Remove((TapePanel)item);

currentMachineData.CloseTape(matchingIndex);

SetHeight(TapeList.Count);

for (int i = 0; i < TapeList.Count; i++)

{

TapeList[i].Location = new Point(12, Height - 473 - 75 \* i);

}

if (TapeList.Count == 1)

{

TapeList[0].GetCloseButton().Hide();

TapeList[0].GetInitialState().Show();

TapeList[0].GetInitialStateLabel().Show();

}

}

else

{

MessageBox.Show("Cannot delete last tape.");

}

}

//Add new tape with blank data

private void AddTape(object sender, EventArgs e)

{

TapePanel tapePanel = new TapePanel(this)

{

Location = new Point(12, Height - 473 - 75 \* TapeList.Count)

};

Controls.Add(tapePanel);

TapeList.Add(tapePanel);

SetHeight(TapeList.Count);

currentMachineData.AddTape();

UI\_UpdateStates(sender, null);

UI\_UpdateMachine(currentMachine, null);

}

//Add new tape to take existing data

private void AddTape()

{

TapePanel tapePanel = new TapePanel(this)

{

Location = new Point(12, Height - 473 - 75 \* TapeList.Count)

};

Controls.Add(tapePanel);

TapeList.Add(tapePanel);

}

#endregion

#region RUNTIME

//Compile machine and lock controls

private void StartMachine(object sender, EventArgs e)

{

if (!runTimer.Enabled)

{

//Disable controls

DisableControls(this);

//Enable controls needed

foreach (TapePanel tape in TapeList)

{

EnableControls(tape.GetTape());

EnableControls(tape.GetPointer());

EnableControls(tape.GetCurrentStateLabel());

EnableControls(tape.GetStepLabel());

(tape.GetTape()).ReadOnly = true;

}

EnableControls(pause);

EnableControls(debug);

EnableControls(currentstate);

EnableControls(steps);

EnableControls(speedL);

EnableControls(speedLabel);

if (!isFullSpeed.Checked)

{

EnableControls(speedSlider);

}

string Script = currentTab.GetText().Text;

//Compile machine

if (!Machine.isCompiled())

{

if (steps.Text == "STEPS")

{

currentMachineData.SetPointers();

}

Machine.Compile(Script, ref currentMachineData, ref TapeList, defaultCaseKey);

}

}

}

//Returns true/false if machine can run

private bool CheckMachine()

{

//Make sure tape does not have illegal characters

foreach (TapePanel tape in TapeList)

{

if (tape.GetTape().Text.Contains(defaultCaseKey))

{

MessageBox.Show("Error: Tape cannot contain the default case character.");

return false;

}

}

//Make sure Universal Tape is valid

if (currentMachineData.GetMachine() == 5)

{

//Define regex for tapes

string instructionTape = TapeList[0].GetTape().Text;

string instructionTapeSyntax = @"^(\w\S\S(r|l|s)\w)(\|(\w\S\S(r|l|\-)\w))\*$";

if (!Regex.IsMatch(instructionTape, instructionTapeSyntax))

{

MessageBox.Show("Error: Instruction Tape in incorrect format.");

return false;

}

//if (TapeList[1].GetTape().Text.Length != 1)

{

//MessageBox.Show("Error: State Tape in incorrect format.");

//return false;

}

}

//Check initial states

if (currentMachineData.GetMachine() == 4)

{

foreach (TapePanel tape in TapeList)

{

if (tape.GetInitialState().Text == "")

{

MessageBox.Show("Error: No initial state selected.");

return false;

}

}

}

else

{

if (TapeList[0].GetInitialState().Text == "")

{

MessageBox.Show("Error: No initial state selected.");

return false;

}

if (currentMachineData.GetMachine() != 3 &&

currentMachineData.GetMachine() != 5)

{

if (TapeList.Count != 1)

{

MessageBox.Show("Error: Cannot use more than one tape \r\nin a single tape Turing machine.");

}

}

}

string[] lines = (currentTab.GetText()).Lines;

int Tapes = currentMachineData.GetTapes();

//Using to check deterministic machines for non-deterministic code

Dictionary<string, List<string>> claimedReadKeys = new Dictionary<string, List<string>>();

//Define regular expressions for verification

string lineSyntax = @"^(\S)+\s\S\s\S\s(r|l|\_)\s(\S)+$";

string keySyntax = @"^\S$";

string moveSyntax = @"^(r|l|\_)$";

string readOnlyLineSyntax = @"^(\S)+\s\S\s(r|l|\_)\s(\S)+$";

string multiTapeLineSyntax = @"^(\S)+\s\S{" + Tapes + @"}\s\S{" + Tapes + @"}\s(r|l|\_){" + Tapes + @"}\s(\S)+$";

string multiTapeKeySyntax = @"^\S{" + Tapes + @"}$";

string multiTapeMoveSyntax = @"^(r|l|\_){" + Tapes + @"}$";

for (int i = 0; i < lines.Length; i++)

{

if (lines[i] != "")

{

if (lines[i][0] != '/' && lines[i][1] != '/')

{

if (lines.Count(j => j == lines[i]) > 1)

{

//Cannot have duplicate lines

MessageBox.Show("Error: Repeating lines. \r\nLine " + (i + 1).ToString());

return false;

}

string[] words = lines[i].Split(' ');

if (currentMachineData.GetMachine() != 1)

{

if (words.Length > 5)

{

MessageBox.Show("Error: Too many fields in transition function. \r\nLine: "

+ (i + 1).ToString());

return false;

}

else if (words.Length < 5)

{

MessageBox.Show("Error: Too few fields in transition function. \r\nLine: "

+ (i + 1).ToString());

return false;

}

}

else

{

if (words.Length > 4)

{

MessageBox.Show("Error: Too many fields in transition function. \r\nLine: "

+ (i + 1).ToString());

return false;

}

else if (words.Length < 4)

{

MessageBox.Show("Error: Too few fields in transition function. \r\nLine: "

+ (i + 1).ToString());

return false;

}

}

if (!claimedReadKeys.ContainsKey(words[0]))

{

claimedReadKeys.Add(words[0], new List<string>() { words[1] });

}

else

{

if (claimedReadKeys[words[0]].Contains(words[1]))

{

if (currentMachineData.GetMachine() != 2)

{

//Cannot use non-deterministic state transitions in deterministic machines

MessageBox.Show("Cannot have multiple functions for the same state and read key in this machine. Line: "

+ (i + 1).ToString());

return false;

}

}

else

{

claimedReadKeys[words[0]].Add(words[1]);

}

}

try

{

switch (currentMachineData.GetMachine())

{

case 0:

case 2:

case 4:

if (!Regex.IsMatch(lines[i], lineSyntax))

{

//Issue with read/write key

if (!Regex.IsMatch(words[1], keySyntax) ||

!Regex.IsMatch(words[2], keySyntax))

{

MessageBox.Show("Error: Key error. (must be of length 1 in this machine.)\r\n" +

"Line: "

+ (i + 1).ToString());

}

//Issue with move key

else if (!Regex.IsMatch(words[3], moveSyntax))

{

MessageBox.Show("Error: Move key error. (must be 'r', 'l', or '\_')\r\n" +

"Line: "

+ (i + 1).ToString());

}

//Unkown error. Probably a space after the final state.

else

{

MessageBox.Show("Syntax Error.\r\n" +

"Make sure there is no space after your final state.\r\n" +

"Line: "

+ (i + 1).ToString());

}

return false;

}

break;

//For read-only machines

case 1:

if (!Regex.IsMatch(lines[i], readOnlyLineSyntax))

{

if (!Regex.IsMatch(words[1], keySyntax))

{

MessageBox.Show("Error: Key error. (must be of length 1 in this machine.)\r\n" +

"Line: "

+ (i + 1).ToString());

}

else if (!Regex.IsMatch(words[2], moveSyntax))

{

MessageBox.Show("Error: Move key error. (must be 'r', 'l', or '\_')\r\n" +

"Line: "

+ (i + 1).ToString());

}

else

{

MessageBox.Show("Syntax Error.\r\n" +

"Make sure there is no space after your final state.\r\n" +

"Line: "

+ (i + 1).ToString());

}

return false;

}

break;

//For multi-tape machines and universal machines

case 3:

case 5:

//Generic errors

if (!Regex.IsMatch(lines[i], multiTapeLineSyntax))

{

if (!Regex.IsMatch(words[1], multiTapeKeySyntax) ||

!Regex.IsMatch(words[2], multiTapeKeySyntax))

{

MessageBox.Show("Error: Key error.\r\n" +

"Line: "

+ (i + 1).ToString());

}

else if (!Regex.IsMatch(words[3], multiTapeMoveSyntax))

{

MessageBox.Show("Error: Move key error. (must be 'r', 'l', or '\_')\r\n" +

"Line: "

+ (i).ToString());

}

else

{

MessageBox.Show("Syntax Error.\r\n" +

"Make sure there is no space after your final state.\r\n" +

"Line: "

+ (i + 1).ToString());

}

return false;

}

break;

}

}

//Error not determinable

catch (Exception)

{

MessageBox.Show("Error. Please check your machine.");

}

}

}

}

return true;

}

//Enable ability to reset machine

private void PauseMachine(object sender, EventArgs e)

{

EnableControls(run);

EnableControls(isFullSpeed);

EnableControls(reset);

EnableControls(step);

DisableControls(pause);

runTimer.Enabled = false;

fastTimer.Enabled = false;

}

//Reset all values

private void StopMachine(object sender, EventArgs e)

{

//Re-enables controls

EnableAllControls(this);

if (isFullSpeed.Checked)

{

DisableControls(speedSlider);

DisableControls(speedLabel);

}

DisableControls(reset);

DisableControls(pause);

runTimer.Enabled = false;

currentstate.Text = "CURRENTSTATE";

steps.Text = "STEPS";

//Reset labels

foreach (TapePanel tape in TapeList)

{

tape.GetCurrentStateLabel().Text = "CURRENTSTATE";

tape.GetStepLabel().Text = "STEPS";

}

//Decompile machine

Machine.DeCompile();

//Reset tape values

for (int i = 0; i < TapeList.Count; i++)

{

(TapeList[i].GetTape()).ReadOnly = false;

(TapeList[i].GetTape()).Text = currentMachineData.GetTape(i);

currentMachineData.ResetPointers();

UI\_UpdatePointer(TapeList[i], currentMachineData.GetPointer(i), 0);

}

}

//Allow machine to be reset, write to debug

private void HaltMachine(object sender, EventArgs e)

{

DisableControls(run);

DisableControls(step);

try

{

steps.Text = (Convert.ToInt32(steps.Text) - 1).ToString();

}

catch (Exception) { }

EnableControls(reset);

DisableControls(pause);

//Output information

if (isDebugBox.Checked)

{

debug.Text += "MACHINE HALTED" + Environment.NewLine;

debug.Text += "==============================" + Environment.NewLine;

debug.Text += "OUTPUTS:" + Environment.NewLine;

if (currentMachineData.GetMachine() == 3)

{

for (int i = 0; i < TapeList.Count; i++)

{

debug.Text += (i + 1).ToString() + ": ";

if (i != TapeList.Count - 1)

{

debug.Text += TapeList[i].GetTape().Text + Environment.NewLine;

}

else

{

debug.Text += TapeList[i].GetTape().Text + "," + Environment.NewLine;

}

}

debug.Text += currentstate.Text + " in " + steps.Text + " steps." + Environment.NewLine;

}

else if (currentMachineData.GetMachine() == 4)

{

for (int i = 0; i < TapeList.Count; i++)

{

debug.Text += (i + 1).ToString() + ": ";

debug.Text += TapeList[i].GetTape().Text + ", ";

debug.Text += TapeList[i].GetCurrentStateLabel().Text + " in ";

debug.Text += TapeList[i].GetStepLabel().Text + " steps. " + Environment.NewLine;

}

}

else if (currentMachineData.GetMachine() == 5)

{

debug.Text += TapeList[2].GetTape().Text + ", ";

debug.Text += TapeList[1].GetTape().Text + " in ";

debug.Text += steps.Text + " steps. " + Environment.NewLine;

}

else

{

debug.Text += TapeList[0].GetTape().Text + ", ";

debug.Text += currentstate.Text + " in ";

debug.Text += steps.Text + " steps. " + Environment.NewLine;

}

debug.Text += "==============================" + Environment.NewLine;

}

}

//Perform one instruction in machine

private void StepMachine(object sender, EventArgs e)

{

//Set steps and initial state

if (currentMachineData.GetMachine() == 4)

{

for (int i = 0; i < TapeList.Count; i++)

{

if (TapeList[i].GetCurrentStateLabel().Text == "CURRENTSTATE")

{

TapeList[i].GetCurrentStateLabel().Text = currentMachineData.GetInitialState(i);

}

if (TapeList[i].GetStepLabel().Text == "STEPS")

{

TapeList[i].GetStepLabel().Text = "1";

}

else

{

int NewSteps = Convert.ToInt32(TapeList[i].GetStepLabel().Text);

NewSteps++;

TapeList[i].GetStepLabel().Text = NewSteps.ToString();

}

}

}

else

{

if (currentstate.Text == "CURRENTSTATE")

{

currentstate.Text = currentMachineData.GetInitialState(0);

}

if (steps.Text == "STEPS")

{

steps.Text = "1";

}

else

{

int NewSteps = Convert.ToInt32(steps.Text);

NewSteps++;

steps.Text = NewSteps.ToString();

}

}

//Perform instruction

bool IsHalted = false;

bool[] IsTapeHalted = new bool[TapeList.Count];

Machine.Tick();

Machine.Update(ref TapeList, ref currentMachineData, ref currentstate, ref IsHalted, ref IsTapeHalted);

//Halt individual tapes

for (int i = 0; i < TapeList.Count; i++)

{

if (IsTapeHalted[i])

{

int NewSteps = Convert.ToInt32(TapeList[i].GetStepLabel().Text);

NewSteps--;

TapeList[i].GetStepLabel().Text = NewSteps.ToString();

}

}

//Halt machine

if (IsHalted)

{

runTimer.Enabled = false;

HaltMachine(sender, e);

}

}

//Called every 10ms by fastTimer

//Runs machine many (user defined) times before updating UI

//Shorter compute time at cost of smooth transition

private void FastStepMachine(object sender, EventArgs e)

{

for (int i = 0; i < fullSpeedInterval; i++)

{

if (Machine.isHalted())

{

fastTimer.Enabled = false;

}

else

{

StepMachine(sender, e);

}

}

}

//Resets labels and start machine

private void RunMachine(object sender, EventArgs e)

{

if (InputWindow.Controls.Count != 0)

{

if (!Machine.isHalted())

{

if (!Machine.isCompiled())

{

if (CheckMachine())

{

//Reset labels before running

if (currentMachineData.GetMachine() == 4)

{

foreach (TapePanel tape in TapeList)

{

if (tape.GetCurrentStateLabel().Text != "CURRENTSTATE")

{

currentstate.Text = "CURRENTSTATE";

}

if (tape.GetStepLabel().Text != "STEPS")

{

if (tape.GetStepLabel().Text == "1")

{

tape.GetStepLabel().Text = "STEPS";

}

}

}

}

else

{

if (currentstate.Text != "CURRENTSTATE")

{

currentstate.Text = "CURRENTSTATE";

}

if (steps.Text != "STEPS")

{

if (steps.Text == "1")

{

steps.Text = "STEPS";

}

}

}

}

else

{

return;

}

}

//Set more values

StartMachine(sender, e);

if (!isFullSpeed.Checked)

{

//Start run-timer for machine

runTimer.Enabled = true;

UI\_SetSpeed(sender, e);

}

else

{

fastTimer.Enabled = true;

}

}

}

else

{

MessageBox.Show("No machines open to run.");

}

}

//Start machine to step only once

private void StepStartMachine(object sender, EventArgs e)

{

if (InputWindow.Controls.Count != 0)

{

if (!Machine.isHalted())

{

if (!Machine.isCompiled())

{

//Compile machine

StartMachine(sender, e);

EnableControls(run);

EnableControls(isFullSpeed);

EnableControls(step);

EnableControls(reset);

DisableControls(pause);

}

//Perform instruction without enabling run-timer

StepMachine(sender, e);

}

}

else

{

MessageBox.Show("No machines open to step.");

}

}

#endregion

#region UTILITY

//Recursively disable all children controls

private void DisableControls(Control C)

{

foreach (Control c in C.Controls)

{

DisableControls(c);

}

C.Enabled = false;

}

//Recursively enable this control and all above it

private void EnableControls(Control C)

{

if (C != null)

{

C.Enabled = true;

EnableControls(C.Parent);

}

}

//Enable only the control C

private void EnableControl(Control C)

{

if (C != null)

{

C.Enabled = true;

}

}

//Recursively enable all children controls

private void EnableAllControls(Control C)

{

foreach (Control c in C.Controls)

{

EnableAllControls(c);

}

C.Enabled = true;

}

//Clears the contents of the debug text box

private void ClearBox(object sender, EventArgs e)

{

debug.Text = "";

}

//Disables speed slider and label when enabling full speed mode

//Enables speed slider and label when disabling full speed mode

private void ToggleSpeed(object sender, EventArgs e)

{

if (((CheckBox)sender).Checked)

{

DisableControls(speedSlider);

DisableControls(speedLabel);

}

else

{

EnableControls(speedSlider);

EnableControls(speedLabel);

}

}

//Creates text for pointer textbox

public static string FillPointerLine(int Pointer, int TapeLength)

{

string Line = "";

if (Pointer < 0)

{

for (int i = Pointer; i < Math.Abs(Pointer) + TapeLength; i++)

{

if (i == Pointer)

{

Line += "^";

}

else

{

Line += " ";

}

}

}

else if (Pointer >= TapeLength)

{

for (int i = -(Pointer - TapeLength) - 1; i <= Pointer; i++)

{

if (i == Pointer)

{

Line += "^";

}

else

{

Line += " ";

}

}

}

else

{

for (int i = 0; i < TapeLength; i++)

{

if (i == Pointer)

{

Line += "^";

}

else

{

Line += " ";

}

}

}

return Line;

}

//Update which line is selected in RichTextBox

public void SetLineNumberLabel(object sender, EventArgs e)

{

RichTextBox Sender = (RichTextBox)sender;

int lineNumber = Sender.GetLineFromCharIndex(Sender.SelectionStart);

LineNumber.Text = "Line: " + (1 + lineNumber);

}

//Creates code to run UTM

private void GenerateUniversalCode(object sender, EventArgs e)

{

string tape = TapeList[0].GetTape().Text;

//Get each instruction

string[] fields = tape.Split('|');

//Create code to run instructions

List<string> code = UniversalMachine.GenerateFunctions(fields, defaultCaseKey);

currentTab.GetText().Text = "";

//Add to program

foreach (string line in code)

{

currentTab.GetText().Text += line + Environment.NewLine;

}

UI\_UpdateStates((ComboBox)TapeList[0].GetInitialState(), currentTab.GetText().Text);

((ComboBox)TapeList[0].GetInitialState()).SelectedText = "q0";

currentMachineData.SetInitialState("q0", 0);

}

//Converts DTM into UTM tape

private void ConvertMachine(object sender, EventArgs e)

{

int machineType = currentMachineData.GetMachine();

if (machineType == 0 || machineType == 1)

{

if (CheckMachine())

{

RichTextBox code = currentTab.GetText();

List<string> Tapes = UniversalMachine.ConvertMachine(ref currentMachineData, TapeList, ref code, defaultCaseKey);

if (Tapes != null)

{

currentMachineData.SetMachine(5);

Tag\_SetMachine(5);

UI\_UpdateMachine(sender, e);

TapeList[0].GetTape().Text = Tapes[0];

TapeList[1].GetTape().Text = Tapes[1];

TapeList[2].GetTape().Text = Tapes[2];

}

}

}

else

{

MessageBox.Show("Machine must be a Deterministic or Read-Only Machine to be converted to a UTM.");

}

}

//Reads config file and sets values

private void LoadSettings()

{

string textBlock = "";

try

{

using (StreamReader sr = new StreamReader("config.ini"))

{

textBlock = sr.ReadToEnd();

sr.Close();

}

string[] settings = textBlock.Split(new char[] { '\r', '\n' },

StringSplitOptions.RemoveEmptyEntries);

speedSlider.Maximum = Convert.ToInt32(settings[0]);

speedSlider.Minimum = Convert.ToInt32(settings[1]);

defaultCaseKey = settings[2][0];

fullSpeedInterval = Convert.ToInt32(settings[3]);

speedSlider.Value = speedSlider.Minimum;

UI\_SetSpeed(null, null);

}

catch (Exception)

{

SaveSettings();

}

}

private void SaveSettings()

{

using (StreamWriter sw = new StreamWriter(File.Create("config.ini")))

{

sw.WriteLine(speedSlider.Maximum);

sw.WriteLine(speedSlider.Minimum);

sw.WriteLine(defaultCaseKey);

sw.WriteLine(fullSpeedInterval);

sw.Close();

}

}

//Close program

private void QuitForm(object sender, EventArgs e)

{

Close();

}

#endregion

#region OPEN FORMS

//Open new diagram form

private void CreateDiagram(object sender, EventArgs e)

{

if (currentMachineData.GetMachine() != 2 &&

currentMachineData.GetMachine() != 5)

{

if (InputWindow.Controls.Count > 0)

{

if (!runTimer.Enabled)

{

if (CheckMachine())

{

FiniteStateMachine diagram;

diagram = new FiniteStateMachine(currentTab.GetText().Text,

TapeList[0].GetInitialState().Text,

currentTab.Text);

diagram.Show();

}

}

}

else

{

MessageBox.Show("No machine to convert to diagram.");

}

}

else

{

MessageBox.Show("Cannot display this machine variant.");

}

}

//Edit preferences

private void OpenSettings(object sender, EventArgs e)

{

OptionsMenu optionsMenu;

optionsMenu = new OptionsMenu(speedSlider.Maximum, speedSlider.Minimum,

defaultCaseKey, fullSpeedInterval);

DialogResult dialogResult = optionsMenu.ShowDialog();

//Set values collected

if (dialogResult == DialogResult.OK)

{

int sliderValue = speedSlider.Value;

speedSlider.Maximum = optionsMenu.MaxSpeed;

speedSlider.Minimum = optionsMenu.MinSpeed;

if (sliderValue < speedSlider.Minimum || sliderValue > speedSlider.Maximum)

{

speedSlider.Value = speedSlider.Minimum;

UI\_SetSpeed(null, null);

}

defaultCaseKey = optionsMenu.defaultKey;

fullSpeedInterval = optionsMenu.fullSpeedInterval;

}

//Save preferences to config.ini

SaveSettings();

}

//Open information form

private void OpenInfo(object sender, EventArgs e)

{

InfoWindow info = new InfoWindow();

info.Show();

}

#endregion

}

}

## MachineData.cs

using System.Collections.Generic;

namespace TuringMachine

{

public class MachineData

{

List<int> Pointer;

List<int> InitialPointer;

List<string> Tape;

List<string> InitialState;

int Machine;

int Tapes;

//Create blank object

public MachineData() { }

//Make out of individual parameters

public MachineData(int machine, int tapes, List<string> tape, List<int> pointer, List<string> initialState)

{

Pointer = new List<int>(pointer);

InitialPointer = new List<int>(pointer);

Tape = tape;

InitialState = initialState;

Machine = machine;

Tapes = tapes;

}

//Create copy of existing object

public MachineData(MachineData machineData)

{

Pointer = new List<int>(machineData.GetPointer());

InitialPointer = new List<int>(machineData.GetPointer());

Tape = new List<string>(machineData.GetTape());

InitialState = new List<string>(machineData.GetInitialState());

Machine = machineData.GetMachine();

Tapes = machineData.GetTapes();

}

#region GET

public int GetPointer(int index)

{

return Pointer[index];

}

public List<int> GetPointer()

{

return Pointer;

}

public string GetTape(int index)

{

return Tape[index];

}

public List<string> GetTape()

{

return Tape;

}

public string GetInitialState(int index)

{

if(Machine != 4)

{

return InitialState[0];

}

return InitialState[index];

}

public List<string> GetInitialState()

{

return InitialState;

}

public int GetMachine()

{

return Machine;

}

public int GetTapes()

{

return Tapes;

}

#endregion

#region SET

public void SetPointer(int value, int index)

{

Pointer[index] = value;

}

public void SetTape(string value, int index)

{

Tape[index] = value;

}

public void SetInitialState(string value, int index)

{

InitialState[index] = value;

}

public void SetMachine(int value)

{

Machine = value;

}

public void ResetPointers()

{

Pointer = new List<int>(InitialPointer);

}

public void SetPointers()

{

InitialPointer = new List<int>(Pointer);

}

//Add new default fields to each list

public void AddTape()

{

Pointer.Add(0);

InitialPointer.Add(0);

Tape.Add("");

InitialState.Add("");

Tapes++;

}

public void SetTapes(int value)

{

Tapes = value;

}

//Remove information from each list at specified index

public void CloseTape(int index)

{

Pointer.RemoveAt(index);

InitialPointer.RemoveAt(index);

Tape.RemoveAt(index);

InitialState.RemoveAt(index);

Tapes--;

}

#endregion

}

}

## TabObj.cs

using System;

using System.Windows.Forms;

using System.Drawing;

namespace TuringMachine

{

public class TabObj : TabPage

{

public TabObj(Control parent)

{

Parent = parent.Controls["InputWindow"];

RichTextBox tb = new RichTextBox

{

Font = new Font("Consolas", 13F),

Location = new Point(0, 0),

Multiline = true,

Name = "codeBox",

ScrollBars = RichTextBoxScrollBars.Vertical,

Size = new Size(505, 301),

TabIndex = 0,

BorderStyle = BorderStyle.FixedSingle

};

tb.SelectionChanged += new EventHandler(((Simulator)parent).SetLineNumberLabel);

tb.KeyDown += new KeyEventHandler(((Simulator)parent).Tab\_Paste);

Controls.Add(tb);

Location = new Point(4, 22);

Name = "tabPage" + Parent.Controls.Count;

Padding = new Padding(3);

Size = new Size(443, 299);

Text = "machine" + Parent.Controls.Count;

UseVisualStyleBackColor = true;

}

public RichTextBox GetText()

{

return (RichTextBox)Controls["codeBox"];

}

}

}

## TapeObj.cs

using System;

using System.Windows.Forms;

using System.Drawing;

namespace TuringMachine

{

public class TapePanel : Panel

{

public TapePanel(Control parent)

{

//Move read/write head

Button pointRight = new Button();

Button pointLeft = new Button();

//Delete tape

Button close = new Button();

//Tape and read/write head

TextBox point = new TextBox();

TextBox Tape = new TextBox();

//Initial state drop-down box

ComboBox iState = new ComboBox();

Label iStateLabel = new Label();

//Labels

Label currentState = new Label();

Label steps = new Label();

#region SET VALUES

Parent = parent;

Anchor = AnchorStyles.Bottom;

Controls.Add(pointRight);

Controls.Add(pointLeft);

Controls.Add(close);

Controls.Add(point);

Controls.Add(Tape);

Controls.Add(iStateLabel);

Controls.Add(iState);

Controls.Add(currentState);

Controls.Add(steps);

Name = "tapePanel" + 1.ToString();

Size = new Size(1100, 70);

Location = new Point(0, 27);

TabStop = false;

Tape.BorderStyle = BorderStyle.None;

Tape.BackColor = Color.White;

Tape.Anchor = AnchorStyles.None;

Tape.Font = new Font("Consolas", 15F);

Tape.Location = new Point(0, 3);

Tape.Name = "tape";

Tape.Size = new Size(1100, 24);

Tape.TabStop = false;

Tape.TextAlign = HorizontalAlignment.Center;

Tape.TextChanged += new EventHandler(((Simulator)Parent).UI\_UpdateTape);

point.BorderStyle = BorderStyle.None;

point.BackColor = Color.White;

point.Anchor = AnchorStyles.None;

point.Font = new Font("Consolas", 15F);

point.ForeColor = SystemColors.WindowText;

point.Location = new Point(0, 25);

point.Margin = new Padding(3, 0, 3, 0);

point.Name = "pointer";

point.ReadOnly = true;

point.Size = new Size(1100, 24);

point.TabStop = false;

point.Text = "^";

point.TextAlign = HorizontalAlignment.Center;

pointRight.AutoSize = true;

pointRight.AutoSizeMode = AutoSizeMode.GrowAndShrink;

pointRight.Location = new Point(553, 45);

pointRight.Name = "pointerRight";

pointRight.Size = new Size(40, 23);

pointRight.TabStop = false;

pointRight.Text = ">>>>";

pointRight.UseVisualStyleBackColor = true;

pointRight.Click += new EventHandler(((Simulator)Parent).UI\_PointerAdd);

pointLeft.AutoSize = true;

pointLeft.AutoSizeMode = AutoSizeMode.GrowAndShrink;

pointLeft.Location = new Point(507, 45);

pointLeft.Name = "pointerLeft";

pointLeft.Size = new Size(40, 23);

pointLeft.TabStop = false;

pointLeft.Text = "<<<<";

pointLeft.UseVisualStyleBackColor = true;

pointLeft.Click += new EventHandler(((Simulator)Parent).UI\_PointerSubtract);

close.AutoSize = true;

close.AutoSizeMode = AutoSizeMode.GrowAndShrink;

close.Location = new Point(831, 44);

close.Name = "Close";

close.Size = new Size(71, 23);

close.TabStop = false;

close.Text = "Close Tape";

close.UseVisualStyleBackColor = true;

close.Visible = true;

close.Click += new EventHandler(((Simulator)Parent).CloseTape);

iState.Font = new Font("Microsoft Sans Serif", 10F);

iState.FormattingEnabled = true;

iState.Location = new Point(723, 44);

iState.Name = "initialState";

iState.Size = new Size(97, 24);

iState.TabStop = false;

iState.SelectedIndexChanged += new EventHandler(((Simulator)Parent).UI\_UpdateInitialState);

iStateLabel.AutoSize = true;

iStateLabel.Font = new Font("Microsoft Sans Serif", 9F);

iStateLabel.Location = new Point(647, 47);

iStateLabel.Margin = new Padding(3);

iStateLabel.Name = "initialStateLabel";

iStateLabel.Size = new Size(70, 15);

iStateLabel.TabStop = false;

iStateLabel.Text = "Initial State:";

currentState.AutoSize = false;

currentState.Name = "currentState";

currentState.Text = "CURRENTSTATE";

currentState.Font = new Font("Consolas", 12F);

currentState.Location = new Point(240, 35);

currentState.BorderStyle = BorderStyle.FixedSingle;

currentState.Size = new Size(150, 35);

currentState.TextAlign = ContentAlignment.MiddleCenter;

steps.AutoSize = false;

steps.Name = "steps";

steps.Text = "STEPS";

steps.Font = new Font("Consolas", 12F);

steps.Location = new Point(146, 35);

steps.BorderStyle = BorderStyle.FixedSingle;

steps.Size = new Size(89, 35);

steps.TextAlign = ContentAlignment.MiddleCenter;

currentState.BringToFront();

steps.BringToFront();

iState.BringToFront();

iStateLabel.BringToFront();

close.BringToFront();

#endregion

}

public ComboBox GetInitialState()

{

return (ComboBox)Controls["initialState"];

}

public TextBox GetTape()

{

return (TextBox)Controls["tape"];

}

public TextBox GetPointer()

{

return (TextBox)Controls["pointer"];

}

public Label GetCurrentStateLabel()

{

return (Label)Controls["currentState"];

}

public Label GetInitialStateLabel()

{

return (Label)Controls["initialStateLabel"];

}

public Label GetStepLabel()

{

return (Label)Controls["steps"];

}

public Button GetCloseButton()

{

return (Button)Controls["close"];

}

}

}

## Machine.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Windows.Forms;

namespace TuringMachine

{

static class Machine

{

public static char defaultCaseCharacter;

static int machine = -1;

static Tape tape;

static MultiTape multiTape;

static List<Tape> multiTrack;

static string currentState;

static List<string> currentStates;

static bool IsHalted;

static bool[] IsTapeHalted;

static List<StandardTransition> Commands;

static List<ReadOnlyTransition> Commands1;

static List<MultiTapeTransition> Commands3;

static string OverrideString;

static string[] OverrideArray;

static List<string> OverrideSequences;

//Checks if data has been added to class for use

public static bool isCompiled()

{

if (machine == -1)

{

return false;

}

return true;

}

//Checks if machine is already halted, and if it can be run/stepped without resetting

public static bool isHalted()

{

if (isCompiled())

{

if (IsHalted)

{

return true;

}

}

return false;

}

//Adds data to variables associated with machine to be used

public static void Compile(string Text, ref MachineData machineData, ref List<TapePanel> Tapes, char defaultCase)

{

machine = machineData.GetMachine();

tape = null;

currentState = "";

IsHalted = false;

IsTapeHalted = new bool[Tapes.Count];

OverrideString = "";

OverrideArray = new string[Tapes.Count];

OverrideSequences = new List<string>();

defaultCaseCharacter = defaultCase;

List<string> Script = Text.Split(new char[] { '\r', '\n' }, StringSplitOptions.RemoveEmptyEntries).ToList<string>();

//Data for Multi-Tape Machines and UTMs

if (machine == 3 || machine == 5)

{

List<int> Pointers = machineData.GetPointer();

List<string> MultiTape = new List<string>();

for (int i = 0; i < Tapes.Count; i++)

{

MultiTape.Add(Tapes[i].GetTape().Text);

}

currentState = machineData.GetInitialState(0);

multiTape = new MultiTape(MultiTape, Pointers);

}

//Data for Multi-Track Machines

else if (machine == 4)

{

List<int> Pointers = machineData.GetPointer();

List<string> MultiTape = machineData.GetTape();

multiTrack = new List<Tape>();

for (int i = 0; i < machineData.GetTapes(); i++)

{

multiTrack.Add(new Tape(MultiTape[i], Pointers[i]));

}

currentStates = new List<string>(machineData.GetInitialState());

currentState = machineData.GetInitialState(0);

}

//Data for other machines

else

{

int Pointer = machineData.GetPointer(0);

string Tape = Tapes[0].GetTape().Text;

tape = new Tape(Tape, Pointer);

currentState = machineData.GetInitialState(0);

}

//Generate transition functions and override characters

switch (machine)

{

case 0:

case 2:

case 4:

Commands = new List<StandardTransition>();

for (int i = 0; i < Script.Count; i++)

{

if (Script[i][0] != '/' && Script[i][1] != '/')

{

Commands.Add(new StandardTransition(Script[i]));

}

}

if(machine == 4)

{

OverrideArray = GetOverrideArray(Commands, Tapes.Count);

}

else

{

OverrideString = GetOverrideChars(Commands);

}

break;

case 1:

Commands1 = new List<ReadOnlyTransition>();

for (int i = 0; i < Script.Count; i++)

{

if (Script[i][0] != '/' && Script[i][1] != '/')

{

Commands1.Add(new ReadOnlyTransition(Script[i]));

}

}

OverrideString = GetOverrideChars(Commands1);

break;

case 3:

case 5:

Commands3 = new List<MultiTapeTransition>();

for (int i = 0; i < Script.Count; i++)

{

if (Script[i][0] != '/' && Script[i][1] != '/')

{

Commands3.Add(new MultiTapeTransition(Script[i], machineData.GetTapes()));

}

}

OverrideSequences = GetOverrideSequences(Commands3, Tapes.Count);

break;

}

}

//Perform instruction

public static void Tick()

{

switch (machine)

{

case 0:

DeterministicMachine.Main(ref tape, Commands, ref currentState, ref IsHalted, OverrideString);

OverrideString = GetOverrideChars(Commands);

break;

case 1:

ReadOnlyMachine.Main(ref tape, Commands1, ref currentState, ref IsHalted, OverrideString);

OverrideString = GetOverrideChars(Commands1);

break;

case 2:

NonDeterministicMachine.Main(ref tape, Commands, ref currentState, ref IsHalted, OverrideString);

OverrideString = GetOverrideChars(Commands);

break;

case 3:

case 5:

MultiTapeMachine.Main(ref multiTape, Commands3, ref currentState, ref IsHalted, OverrideSequences);

OverrideSequences = GetOverrideSequences(Commands3, multiTape.GetTapes());

break;

case 4:

MultiTrackMachine.Main(ref multiTrack, Commands, ref currentStates, ref IsHalted, ref IsTapeHalted, OverrideArray);

OverrideArray = GetOverrideArray(Commands, multiTrack.Count);

break;

}

return;

}

//Update UI and MachineData values,

//performed after each tick.

public static void Update(ref List<TapePanel> Tapes, ref MachineData machineData,

ref Label CurrentState, ref bool isHalted, ref bool[] isTapeHalted)

{

switch (machine)

{

case 0:

case 1:

case 2:

tape.TrimEdges();

//Set underscores to blank characters

//Update tape in UI

Tapes[0].GetTape().Text = tape.GetTape().Replace('\_', ' ');

//Update pointer in UI and MachineData

machineData.SetPointer(tape.GetPointer(), 0);

Tapes[0].GetPointer().Text = Simulator.FillPointerLine(tape.GetPointer(),Tapes[0].GetTape().Text.Length);

//Update state in UI

CurrentState.Text = currentState;

isHalted = IsHalted;

break;

case 3:

case 5:

for (int i = 0; i < multiTape.GetTapes(); i++)

{

multiTape.TrimEdges(i);

Tapes[i].GetTape().Text = multiTape.GetTape(i).Replace('\_', ' ');

machineData.SetPointer(multiTape.GetPointer(i), i);

Tapes[i].GetPointer().Text = Simulator.FillPointerLine(multiTape.GetPointer(i), Tapes[i].GetTape().Text.Length);

}

CurrentState.Text = currentState;

isHalted = IsHalted;

break;

case 4:

for (int i = 0; i < multiTrack.Count; i++)

{

multiTrack[i].TrimEdges();

Tapes[i].GetTape().Text = multiTrack[i].GetTape().Replace('\_', ' ');

machineData.SetPointer(multiTrack[i].GetPointer(), i);

Tapes[i].GetPointer().Text = Simulator.FillPointerLine(multiTrack[i].GetPointer(), Tapes[i].GetTape().Text.Length);

Tapes[i].GetCurrentStateLabel().Text = currentStates[i];

//Indicate if individual tape is halted

isTapeHalted[i] = IsTapeHalted[i];

}

isHalted = IsHalted;

break;

}

}

//Sets variables to null when stopping machine.

//Must be done for isCompiled() to work.

public static void DeCompile()

{

machine = -1;

}

//Get Override Characters for current state

public static string GetOverrideChars(List<StandardTransition> Script)

{

string overrideTemp = "";

bool containsOverride = false;

foreach(StandardTransition line in Script)

{

//Only get characters from current state functions

if (line.GetInitialState() == currentState)

{

if (line.GetReadKey() != defaultCaseCharacter)

{

overrideTemp += line.GetReadKey();

}

else

{

//If default case function is present, keep override chars found.

containsOverride = true;

}

}

}

//If no default case, no need for overrides

if (!containsOverride)

{

overrideTemp = "";

}

return overrideTemp;

}

public static string GetOverrideChars(List<ReadOnlyTransition> Script)

{

string overrideTemp = "";

bool containsOverride = false;

foreach (ReadOnlyTransition line in Script)

{

if (line.GetInitialState() == currentState)

{

if (line.GetReadKey() != defaultCaseCharacter)

{

overrideTemp += line.GetReadKey();

}

else

{

containsOverride = true;

}

}

}

if (!containsOverride)

{

overrideTemp = "";

}

return overrideTemp;

}

//Get Override Characters for each current state

public static string[] GetOverrideArray(List<StandardTransition> Script, int tapeCount)

{

string[] OverrideCharArray = new string[tapeCount];

//Get characters for each state

for (int i = 0; i < tapeCount; i++)

{

if (!IsTapeHalted[i])

{

string overrideTemp = "";

bool containsOverride = false;

foreach (StandardTransition line in Script)

{

//Only get characters from current state functions

if (line.GetInitialState() == currentStates[i])

{

if (line.GetReadKey() != defaultCaseCharacter)

{

overrideTemp += line.GetReadKey();

}

else

{

//If default case function is present, keep override chars found.

containsOverride = true;

}

}

}

//If no default case, no need for overrides

if (!containsOverride)

{

OverrideCharArray[i] = "";

}

else

{

OverrideCharArray[i] = overrideTemp;

}

}

}

return OverrideCharArray;

}

//Get Sequences that can override default case function

public static List<string> GetOverrideSequences(List<MultiTapeTransition> Script, int tapeCount)

{

List<string> OverrideSequencesTemp = new List<string>();

//Do for each transition

foreach (MultiTapeTransition line in Script)

{

//Only perform on functions of the current state

if (line.GetInitialState() == currentState)

{

bool isOverride = false;

//Check if function is default case function

for (int i = 0; i < tapeCount; i++)

{

//If all chars are default case, isOverride will end as true.

if (line.GetReadKey(i) == defaultCaseCharacter)

{

isOverride = true;

}

else

{

isOverride = false;

break;

}

}

//If not the default case function,

if (!isOverride)

{

//add to the list of override sequences

string sequence = "";

for(int i = 0; i < line.GetReadKeys().Count; i++)

{

sequence += line.GetReadKey(i);

}

OverrideSequencesTemp.Add(sequence);

}

}

}

return OverrideSequencesTemp;

}

}

}

## Machines

### DeterministicMachine.cs

using System.Collections.Generic;

using System.Linq;

namespace TuringMachine

{

class DeterministicMachine

{

public static void Main(ref Tape Tape, List<StandardTransition> Commands,

ref string CurrentState, ref bool IsHalted, string OverrideString)

{

bool IsStepCompleted = false;

foreach (StandardTransition command in Commands)

{

if (!IsStepCompleted)

{

//If transition's state matches current state

if (command.GetInitialState() == CurrentState)

{

Step(command, OverrideString, ref Tape, ref CurrentState, ref IsStepCompleted);

}

}

else

{

break;

}

}

//Halt machine

if (!IsStepCompleted)

{

IsHalted = true;

}

else

{

IsHalted = false;

}

}

public static void Step(StandardTransition code, string OverrideString, ref Tape Tape,

ref string CurrentState, ref bool IsStepCompleted)

{

bool IsMatch = false;

//If pointer right of bounds (will be blank character)

if (Tape.GetPointer() >= Tape.GetLength())

{

if (code.GetReadKey() == Machine.defaultCaseCharacter)

{

if (OverrideString.Contains("\_"))

{

IsMatch = false;

}

else

{

IsMatch = true;

}

}

else if (code.GetReadKey() == '\_')

{

IsMatch = true;

}

}

//If pointer left of bounds (will be blank character)

else if (Tape.GetPointer() < 0)

{

if (code.GetReadKey() == Machine.defaultCaseCharacter)

{

if (OverrideString.Contains("\_"))

{

IsMatch = false;

}

else

{

IsMatch = true;

}

}

else if (code.GetReadKey() == '\_')

{

IsMatch = true;

}

}

//Pointer in bounds

else

{

//Is match

if (code.GetReadKey() == Tape.GetKey())

{

IsMatch = true;

}

//If default char can be used

else if (code.GetReadKey() == Machine.defaultCaseCharacter)

{

if (OverrideString.Contains(Tape.GetKey()))

{

IsMatch = false;

}

else

{

IsMatch = true;

}

}

else

{

IsMatch = false;

}

}

//Writes and moves tape, set new state

if (IsMatch)

{

WriteTape(code.GetWriteKey(), ref Tape);

MoveTape(code.GetMoveKey(), ref Tape);

CurrentState = code.GetFinalState();

IsStepCompleted = true;

}

else

{

return;

}

}

public static void WriteTape(char WriteKey, ref Tape Tape)

{

//Add to beginning

if (Tape.GetPointer() < 0)

{

if(WriteKey != Machine.defaultCaseCharacter)

{

Tape.AddFront(WriteKey);

}

}

//Add to end

else if (Tape.GetPointer() >= Tape.GetLength())

{

if (WriteKey != Machine.defaultCaseCharacter)

{

Tape.AddBack(WriteKey);

}

}

//Set at pointer

else

{

if (WriteKey != Machine.defaultCaseCharacter)

{

Tape.AddMiddle(WriteKey);

}

}

}

public static void MoveTape(char MoveKey, ref Tape Tape)

{

switch (MoveKey)

{

//Move left

case 'l':

if (Tape.GetPointer() > 0)

{

Tape.ChangePointer(-1);

}

else if(Tape.GetLength() == 0)

{

Tape.AddFront(' ');

Tape.AddFront(' ');

}

else

{

Tape.AddFront('\_');

}

break;

//Move right

case 'r':

if(Tape.GetLength() == 0)

{

Tape.AddBack(' ');

}

else if(Tape.GetPointer() >= Tape.GetLength())

{

Tape.AddBack(' ');

}

Tape.ChangePointer(1);

break;

//Don't move

case '\_':

break;

}

}

}

}

### ReadOnlyMachine.cs

using System;

using System.Collections.Generic;

using System.Linq;

namespace TuringMachine

{

class ReadOnlyMachine

{

//Same algorithm as DTM,

//Exception: WriteTape() method is not used

//See DTM implementation for comments on base algorithm

public static void Main(ref Tape Tape, List<ReadOnlyTransition> Commands, ref string CurrentState, ref bool IsHalted, string OverrideString)

{

bool IsStepCompleted = false;

foreach (ReadOnlyTransition command in Commands)

{

if (!IsStepCompleted)

{

if (command.GetInitialState() == CurrentState)

{

Step(command, ref Tape, ref CurrentState, ref IsStepCompleted, OverrideString);

}

}

}

if (!IsStepCompleted)

{

IsHalted = true;

}

else

{

IsHalted = false;

}

return;

}

public static void Step(ReadOnlyTransition code, ref Tape Tape, ref string CurrentState, ref Boolean IsStepCompleted, string OverrideString)

{

bool IsMatch = false;

//If pointer right of bounds (will be blank character)

if (Tape.GetPointer() >= Tape.GetLength())

{

if (code.GetReadKey() == Machine.defaultCaseCharacter)

{

if (OverrideString.Contains("\_"))

{

IsMatch = false;

}

else

{

IsMatch = true;

}

}

else if (code.GetReadKey() == '\_')

{

IsMatch = true;

}

}

//If pointer left of bounds (will be blank character)

else if (Tape.GetPointer() < 0)

{

if (code.GetReadKey() == Machine.defaultCaseCharacter)

{

if (OverrideString.Contains("\_"))

{

IsMatch = false;

}

else

{

IsMatch = true;

}

}

else if (code.GetReadKey() == '\_')

{

IsMatch = true;

}

}

//Pointer in bounds

else

{

//Is match

if (code.GetReadKey() == Tape.GetKey())

{

IsMatch = true;

}

//If default char can be used

else if (code.GetReadKey() == Machine.defaultCaseCharacter)

{

if (OverrideString.Contains(Tape.GetKey()))

{

IsMatch = false;

}

else

{

IsMatch = true;

}

}

else

{

IsMatch = false;

}

}

if (IsMatch)

{

MoveTape(code.GetMoveKey(), ref Tape);

CurrentState = code.GetFinalState();

IsStepCompleted = true;

}

else

{

return;

}

}

public static void MoveTape(char MoveKey, ref Tape Tape)

{

switch (MoveKey)

{

case 'l':

if (Tape.GetPointer() > 0)

{

Tape.ChangePointer(-1);

}

else if (Tape.GetLength() == 0)

{

Tape.AddFront(' ');

Tape.AddFront(' ');

}

else

{

Tape.AddFront('\_');

}

break;

case 'r':

Tape.ChangePointer(1);

break;

case '\_':

break;

}

}

}

}

### NonDeterministicMachine.cs

using System;

using System.Collections.Generic;

using System.Linq;

namespace TuringMachine

{

class NonDeterministicMachine

{

//Same algorithm as DTM,

//Exception: Beginning of Step() method

//See DTM implementation for comments on base algorithm

public static void Main(ref Tape Tape, List<StandardTransition> Commands, ref string CurrentState, ref bool IsHalted, string OverrideString)

{

bool IsStepCompleted = false;

Step(Commands, ref Tape, ref CurrentState, ref IsStepCompleted, OverrideString);

if (!IsStepCompleted)

{

IsHalted = true;

}

else

{

IsHalted = false;

}

}

public static void Step(List<StandardTransition> commands, ref Tape Tape, ref string CurrentState, ref bool IsStepCompleted, string OverrideString)

{

//Get read key from tape

char ReadKey;

try

{

ReadKey = Tape.GetKey();

}

catch (ArgumentOutOfRangeException)

{

ReadKey = '\_';

}

List<StandardTransition> MatchingCommands = new List<StandardTransition>();

StandardTransition code;

Random rng = new Random();

int random = 0;

bool IsMatch = false;

//Get commands with same state and read key

foreach (StandardTransition command in commands)

{

if (command.GetInitialState() == CurrentState)

{

if (command.GetReadKey() == ReadKey)

{

MatchingCommands.Add(command);

}

else if(command.GetReadKey() == Machine.defaultCaseCharacter && !OverrideString.Contains(ReadKey))

{

MatchingCommands.Add(command);

}

}

}

//Get random command from list of matching commands

if (MatchingCommands.Count > 0)

{

IsMatch = true;

random = rng.Next(0, MatchingCommands.Count);

code = MatchingCommands[random];

}

else

{

IsMatch = false;

return;

}

if (IsMatch)

{

WriteTape(code.GetWriteKey(), ref Tape);

MoveTape(code.GetMoveKey(), ref Tape);

CurrentState = code.GetFinalState();

IsStepCompleted = true;

}

else

{

return;

}

}

public static void WriteTape(char WriteKey, ref Tape Tape)

{

if (Tape.GetPointer() < 0)

{

if (WriteKey != Machine.defaultCaseCharacter)

{

Tape.AddFront(WriteKey);

}

}

else if (Tape.GetPointer() >= Tape.GetLength())

{

if (WriteKey != Machine.defaultCaseCharacter)

{

Tape.AddBack(WriteKey);

}

}

else

{

if (WriteKey != Machine.defaultCaseCharacter)

{

Tape.AddMiddle(WriteKey);

}

}

}

public static void MoveTape(char MoveKey, ref Tape Tape)

{

switch (MoveKey)

{

case 'l':

if (Tape.GetPointer() > 0)

{

Tape.ChangePointer(-1);

}

else if (Tape.GetLength() == 0)

{

Tape.AddFront(' ');

Tape.AddFront(' ');

}

else

{

Tape.AddFront('\_');

}

break;

case 'r':

Tape.ChangePointer(1);

break;

case '\_':

break;

}

}

}

}

### MultiTapeMachine.cs

using System.Collections.Generic;

using System.Linq;

using System.Text.RegularExpressions;

namespace TuringMachine

{

class MultiTapeMachine

{

//Same algorithm as DTM,

//Exception: Step() algorithm repeated for each tape.

//All tapes must be match with the tape for instruction to be accepted and excecuted

//See DTM implementation for comments on base algorithm

public static void Main(ref MultiTape Tape, List<MultiTapeTransition> Commands, ref string CurrentState, ref bool IsHalted, List<string> OverrideSequences)

{

bool IsStepCompleted = false;

foreach (MultiTapeTransition command in Commands)

{

if (!IsStepCompleted)

{

if (command.GetInitialState() == CurrentState)

{

Step(command, ref Tape, ref CurrentState, ref IsStepCompleted, OverrideSequences);

}

}

else

{

break;

}

}

if (!IsStepCompleted)

{

IsHalted = true;

}

else

{

IsHalted = false;

}

}

public static void Step(MultiTapeTransition code, ref MultiTape Tape, ref string CurrentState, ref bool IsStepCompleted, List<string> OverrideSequences)

{

//Get currently read string from tapes

string Read = "";

for(int i = 0; i < Tape.GetTapes(); i++)

{

Read += Tape.GetKey(i);

}

//Make OverrideSequences empty list if null

//Removes NullReferenceException issue

if (OverrideSequences == null)

{

OverrideSequences = new List<string>();

}

//Get how many times default case appears in command

int priority = 0;

string readKeys = "";

for(int i = 0; i < Read.Length; i++)

{

readKeys += code.GetReadKey(i);

}

bool IsMatch = false;

for (int i = 0; i < Read.Length; i++)

{

//Accept default character or tape contents

if (readKeys[i] == Read[i])

{

IsMatch = true;

}

else

{

if (readKeys[i] != Machine.defaultCaseCharacter)

{

IsMatch = false;

break;

}

else

{

IsMatch = true;

}

}

}

priority = readKeys.Count(i => i == Machine.defaultCaseCharacter);

//Get highest priority from override sequences

int overridePriority = Read.Length;

string overrideSequence = "";

foreach(string Override in OverrideSequences)

{

if (Override != readKeys)

{

//Get priority / number of default characters

int tempOverridePriority = Override.Count(i => i == Machine.defaultCaseCharacter);

//If more fully defined

if (tempOverridePriority < overridePriority)

{

//Check sequence matches tape

bool doesOverrideMatch = false;

for (int i = 0; i < Read.Length; i++)

{

//Accept default character or tape contents

if (Override[i] == Read[i])

{

doesOverrideMatch = true;

}

else

{

if (Override[i] != Machine.defaultCaseCharacter)

{

doesOverrideMatch = false;

break;

}

else

{

doesOverrideMatch = true;

}

}

}

//If it matches, make new highest priority override

if (doesOverrideMatch)

{

overridePriority = tempOverridePriority;

overrideSequence = Override;

}

}

}

}

//If a more fully defined function exists

if(overridePriority < priority)

{

return;

}

if (IsMatch)

{

WriteTapes(code.GetWriteKeys(), ref Tape);

MoveTapes(code.GetMoveKeys(), ref Tape);

CurrentState = code.GetFinalState();

IsStepCompleted = true;

}

}

public static void WriteTapes(List<char> WriteKeys, ref MultiTape Tape)

{

for (int i = 0; i < Tape.GetTapes(); i++)

{

if (Tape.GetPointer(i) < 0)

{

if (WriteKeys[i] != Machine.defaultCaseCharacter)

{

Tape.AddFront(WriteKeys[i], i);

}

}

else if (Tape.GetPointer(i) >= Tape.GetLength(i))

{

if (WriteKeys[i] != Machine.defaultCaseCharacter)

{

Tape.AddBack(WriteKeys[i], i);

}

}

else

{

if (WriteKeys[i] != Machine.defaultCaseCharacter)

{

Tape.AddMiddle(WriteKeys[i], i);

}

}

}

}

public static void MoveTapes(List<char> MoveKeys, ref MultiTape Tape)

{

for (int i = 0; i < Tape.GetTapes(); i++)

{

switch (MoveKeys[i])

{

case 'l':

if (Tape.GetPointer(i) > 0)

{

Tape.ChangePointer(-1, i);

}

else if(Tape.GetLength(i) == 0)

{

Tape.AddFront('\_', i);

Tape.AddFront('\_', i);

}

else

{

Tape.AddFront('\_', i);

}

break;

case 'r':

if(Tape.GetLength(i) == 0)

{

Tape.AddBack(' ', i);

}

else if(Tape.GetPointer(i) >= Tape.GetLength(i))

{

Tape.AddBack('\_', i);

}

Tape.ChangePointer(1, i);

break;

case '\_':

break;

}

}

}

}

}

### MultiTrackMachine.cs

using System.Collections.Generic;

namespace TuringMachine

{

class MultiTrackMachine

{

//Same algorithm as DTM,

//Exception: Algorithm repeated for each tape in main() method.

//See DTM implementation for comments on base algorithm

public static void Main(ref List<Tape> Tape, List<StandardTransition> Commands, ref List<string> CurrentStates, ref bool IsAllHalted, ref bool[] IsTapeHalted, string[] OverrideArray)

{

bool[] IsStepCompleted = new bool[Tape.Count];

//Loop algorithm for each tape

//Tapes work independently, eg: one can halt while others continue

for (int i = 0; i < Tape.Count; i++)

{

if (!IsTapeHalted[i])

{

foreach (StandardTransition Command in Commands)

{

if (!IsStepCompleted[i])

{

if (Command.GetInitialState() == CurrentStates[i])

{

Tape CurrentTape = Tape[i];

string currentState = CurrentStates[i];

Step(Command, ref CurrentTape, ref currentState, ref IsStepCompleted[i], OverrideArray[i]);

CurrentStates[i] = currentState;

}

}

}

if (!IsStepCompleted[i])

{

IsTapeHalted[i] = true;

}

else

{

IsTapeHalted[i] = false;

}

IsStepCompleted[i] = false;

}

}

IsAllHalted = IsAllHaltedCheck(IsTapeHalted);

}

public static void Step(StandardTransition code, ref Tape Tape, ref string CurrentState, ref bool IsStepCompleted, string OverrideString)

{

bool IsMatch = false;

//If pointer right of bounds (will be blank character)

if (Tape.GetPointer() >= Tape.GetLength())

{

if (code.GetReadKey() == Machine.defaultCaseCharacter)

{

if (OverrideString.Contains("\_"))

{

IsMatch = false;

}

else

{

IsMatch = true;

}

}

else if (code.GetReadKey() == '\_')

{

IsMatch = true;

}

}

//If pointer left of bounds (will be blank character)

else if (Tape.GetPointer() < 0)

{

if (code.GetReadKey() == Machine.defaultCaseCharacter)

{

if (OverrideString.Contains("\_"))

{

IsMatch = false;

}

else

{

IsMatch = true;

}

}

else if (code.GetReadKey() == '\_')

{

IsMatch = true;

}

}

//Pointer in bounds

else

{

//Is match

if (code.GetReadKey() == Tape.GetKey())

{

IsMatch = true;

}

//If default char can be used

else if (code.GetReadKey() == Machine.defaultCaseCharacter)

{

if (OverrideString.Contains(Tape.GetKey().ToString()))

{

IsMatch = false;

}

else

{

IsMatch = true;

}

}

else

{

IsMatch = false;

}

}

if (IsMatch)

{

WriteTape(code.GetWriteKey(), ref Tape);

MoveTape(code.GetMoveKey(), ref Tape);

CurrentState = code.GetFinalState();

IsStepCompleted = true;

}

else

{

return;

}

//Console.ReadLine();

}

public static void WriteTape(char WriteKey, ref Tape Tape)

{

if (Tape.GetPointer() < 0)

{

if (WriteKey != Machine.defaultCaseCharacter)

{

Tape.AddFront(WriteKey);

}

}

else if (Tape.GetPointer() >= Tape.GetLength())

{

if (WriteKey != Machine.defaultCaseCharacter)

{

Tape.AddBack(WriteKey);

}

}

else

{

if (WriteKey != Machine.defaultCaseCharacter)

{

Tape.AddMiddle(WriteKey);

}

}

}

public static void MoveTape(char MoveKey, ref Tape Tape)

{

switch (MoveKey)

{

case 'l':

if (Tape.GetPointer() > 0)

{

Tape.ChangePointer(-1);

}

else if (Tape.GetLength() == 0)

{

Tape.AddFront(' ');

Tape.AddFront(' ');

}

else

{

Tape.AddFront('\_');

}

break;

case 'r':

Tape.ChangePointer(1);

break;

case '\_':

break;

}

}

public static bool IsAllHaltedCheck(bool[] IsHalted)

{

foreach (bool halt in IsHalted)

{

if (!halt)

{

return false;

}

}

return true;

}

}

}

## Tapes

### Tape.cs

using System.Collections.Generic;

namespace TuringMachine

{

public class Tape

{

#region ATTRIBUTES

List<char> Contents = new List<char>();

int Pointer;

#endregion

#region CONSTRUCTORS

//Add blank tape

public Tape()

{

Contents = new List<char>();

}

//Tape with known contents

public Tape(string Tape, int Pointer)

{

Contents = new List<char>();

foreach (char key in Tape)

{

Contents.Add(key);

}

this.Pointer = Pointer;

}

#endregion

#region GET

//Get whole tape

public string GetTape()

{

string tape = "";

foreach (char c in Contents)

{

tape += c;

}

return tape;

}

//Get character from tape

public char GetKey()

{

return Contents[Pointer];

}

public int GetLength()

{

return Contents.Count;

}

public int GetPointer()

{

return Pointer;

}

#endregion

#region SET

//Add at beginning of tape

public void AddFront(char Input)

{

if (Contents.Count == 1)

{

Contents.Add(Contents[0]);

}

else if(Contents.Count == 0)

{

Contents.Add(Input);

}

else

{

Contents.Add(Contents[Contents.Count - 1]);

}

for (int i = Contents.Count - 1; i > 0; i--)

{

Contents[i] = Contents[i - 1];

}

Contents[0] = Input;

}

//Add at pointer value of tape

public void AddMiddle(char Input)

{

Contents[Pointer] = Input;

}

//Add to end of tape

public void AddBack(char Input)

{

Contents.Add(Input);

}

//Remove unnecessary spaces from ends of tape

public void TrimEdges()

{

if (Contents.Count != 0)

{

int IndexOfLeftBound = 0;

int IndexOfRightBound = 0;

for (int i = 0; i < Contents.Count; i++)

{

if (Contents[i] != '\_')

{

//Assign first item that is not blank

IndexOfLeftBound = i;

break;

}

}

for (int i = Contents.Count - 1; i >= 0; i--)

{

if (Contents[i] != '\_')

{

//Assign first item that is not blank

IndexOfRightBound = i;

break;

}

}

//If pointer is after left bound

//Delete all blank spaces to left of tape

if (Pointer >= IndexOfLeftBound)

{

Contents.RemoveRange(0, IndexOfLeftBound);

Pointer -= IndexOfLeftBound;

}

//If pointer is before right bound

//Delete all blank spaces to right of tape

if (Pointer <= IndexOfRightBound && Contents.Count - IndexOfRightBound - 1 > 0)

{

Contents.RemoveRange(IndexOfRightBound + 1, Contents.Count - IndexOfRightBound - 1);

}

}

}

//Move pointer by +/- 1

public void ChangePointer(int Value)

{

Pointer += Value;

}

#endregion

}

}

### MultiTape.cs

using System;

using System.Collections.Generic;

namespace TuringMachine

{

public class MultiTape

{

#region ATTRIBUTES

List<List<char>> Contents;

readonly List<int> Pointers;

#endregion

#region CONSTRUCTORS

//Add blank tape

public MultiTape()

{

Contents = new List<List<char>>();

}

//Add tape with known values

public MultiTape(List<string> Tape, List<int> Pointers)

{

if (Tape.Count != Pointers.Count)

{

return;

}

Contents = new List<List<char>>();

for (int i = 0; i < Tape.Count; i++)

{

Contents.Add(new List<char>());

foreach (char key in Tape[i])

{

Contents[i].Add(key);

}

}

this.Pointers = new List<int>();

for (int i = 0; i < Tape.Count; i++)

{

this.Pointers.Add(Pointers[i]);

}

}

#endregion

#region GET

//Get whole tape

public string GetTape(int Tape)

{

string tape = "";

foreach (char c in Contents[Tape])

{

tape += c;

}

return tape;

}

//Get character from tape

public char GetKey(int Tape)

{

try

{

return Contents[Tape][Pointers[Tape]];

}

catch(Exception)

{

return '\_';

}

}

public int GetTapes()

{

return Contents.Count;

}

public int GetLength(int Tape)

{

return Contents[Tape].Count;

}

public int GetPointer(int Tape)

{

return Pointers[Tape];

}

public List<int> GetPointers()

{

return Pointers;

}

#endregion

#region SET

//Add to beginning of specified tape

public void AddFront(char Input, int Tape)

{

if (Contents[Tape].Count == 1)

{

Contents[Tape].Add(Contents[Tape][0]);

}

else if(Contents[Tape].Count == 0)

{

Contents[Tape].Add(Input);

}

else

{

Contents[Tape].Add(Contents[Tape][Contents[Tape].Count - 1]);

}

for (int i = Contents[Tape].Count - 1; i > 0; i--)

{

Contents[Tape][i] = Contents[Tape][i - 1];

}

Contents[Tape][0] = Input;

}

//Add at current position on tape

public void AddMiddle(char Input, int Tape)

{

Contents[Tape][Pointers[Tape]] = Input;

}

//Add to end of specified tape

public void AddBack(char Input, int Tape)

{

Contents[Tape].Add(Input);

}

//Removes unnecessary spaces from tape

public void TrimEdges(int Tape)

{

if (Contents[Tape].Count != 0)

{

int IndexOfLeftBound = 0;

int IndexOfRightBound = 0;

for (int i = 0; i < Contents[Tape].Count; i++)

{

if (Contents[Tape][i] != '\_')

{

//Assign first item that is not blank

IndexOfLeftBound = i;

break;

}

}

for (int i = Contents[Tape].Count - 1; i >= 0; i--)

{

if (Contents[Tape][i] != '\_')

{

//Assign first item that is not blank

IndexOfRightBound = i;

break;

}

}

//If pointer is after left bound

//Delete all blank spaces to left of tape

if (Pointers[Tape] >= IndexOfLeftBound)

{

Contents[Tape].RemoveRange(0, IndexOfLeftBound);

Pointers[Tape] -= IndexOfLeftBound;

}

//If pointer is before right bound

//Delete all blank spaces to right of tape

if (Pointers[Tape] <= IndexOfRightBound && Contents[Tape].Count - IndexOfRightBound - 1 > 0)

{

Contents[Tape].RemoveRange(IndexOfRightBound + 1, Contents[Tape].Count - IndexOfRightBound - 1);

}

}

}

//Changes pointer by +/- 1

public void ChangePointer(int Value, int Tape)

{

Pointers[Tape] += Value;

}

#endregion

}

}

## Transition Functions

### ReadOnlyTransition.cs

namespace TuringMachine

{

public class ReadOnlyTransition

{

#region ATTRIBUTES

protected string InitialState;

protected char ReadKey;

protected char MoveKey;

protected string FinalState;

#endregion

#region CONSTRUCTORS

public ReadOnlyTransition()

{

}

//Parse parameter to get attributes (validation already done)

public ReadOnlyTransition(string Line)

{

string[] strings = Line.Split(' ');

InitialState = strings[0];

ReadKey = strings[1][0];

MoveKey = strings[2][0];

FinalState = strings[3];

}

#endregion

#region GET

//Return private attributes

public string GetInitialState()

{

return InitialState;

}

public char GetReadKey()

{

return ReadKey;

}

public char GetMoveKey()

{

return MoveKey;

}

public string GetFinalState()

{

return FinalState;

}

#endregion

}

}

### StandardTransition.cs

using System.Collections.Generic;

namespace TuringMachine

{

public class StandardTransition : ReadOnlyTransition

{

protected char WriteKey;

public StandardTransition()

{

}

//Parse parameter to get attributes (validation already done)

public StandardTransition(string Line)

{

string[] strings = Line.Split(' ');

InitialState = strings[0];

ReadKey = strings[1][0];

WriteKey = strings[2][0];

MoveKey = strings[3][0];

FinalState = strings[4];

}

public char GetWriteKey()

{

return WriteKey;

}

}

}

### MultiTapeTransition.cs

using System.Collections.Generic;

using System.Linq;

namespace TuringMachine

{

public class MultiTapeTransition : StandardTransition

{

#region ATTRIBUTES

new List<char> ReadKey;

new List<char> WriteKey;

new List<char> MoveKey;

#endregion

#region CONSTRUCTORS

public MultiTapeTransition()

{

}

//Parse parameter to get attributes (validation already done)

public MultiTapeTransition(string Line, int Tapes)

{

string[] strings = Line.Split(' ');

InitialState = strings[0];

ReadKey = strings[1].ToCharArray().ToList();

WriteKey = strings[2].ToCharArray().ToList();

MoveKey = strings[3].ToCharArray().ToList();

FinalState = strings[strings.Length - 1];

}

#endregion

#region GET

public char GetReadKey(int Tape)

{

return ReadKey[Tape];

}

public List<char> GetReadKeys()

{

return ReadKey;

}

public char GetWriteKey(int Tape)

{

return WriteKey[Tape];

}

public List<char> GetWriteKeys()

{

return WriteKey;

}

public char GetMoveKey(int Tape)

{

return MoveKey[Tape];

}

public List<char> GetMoveKeys()

{

return MoveKey;

}

#endregion

}

}

## FiniteStateMachine.cs

using System;

using System.Collections.Generic;

using System.Drawing;

using System.Windows.Forms;

namespace TuringMachine

{

public partial class FiniteStateMachine : Form

{

string code = "";

public string initialState { get; } = "";

Graph graph;

const int spacing = 200;

public FiniteStateMachine(string code, string initialState, string name)

{

InitializeComponent();

Text = "Graphical Representation - " + name;

this.code = code;

this.initialState = initialState;

VerticalScroll.Visible = true;

}

//Generate diagram on loading

private void Generate(object sender, EventArgs e)

{

try

{

graph = new Graph(code);

if (graph.statesHalting.Contains(initialState))

{

MessageBox.Show("Error: Initial state cannot be a halting state.");

Close();

}

//Load nodes

Spawn(initialState, "");

//Add edges

EdgeCanvas edgeCanvas = new EdgeCanvas(graph);

Controls.Add(edgeCanvas);

//Set scroll 'cheat' object

int highest = 0;

int lowest = 0;

foreach (Control control in Controls)

{

if (control is Node)

{

if (control.Location.Y > highest)

{

highest = control.Location.Y;

}

if (control.Location.Y < lowest)

{

lowest = control.Location.Y;

}

}

}

vScrollBar1.Location = new Point(-50, (highest - lowest) / 2);

vScrollBar1.Size = new Size(2, highest - lowest);

}

catch (Exception)

{

MessageBox.Show("Error generating FSM.");

Close();

}

}

//Depth-first loading of nodes

private void Spawn(string state, string parent)

{

if (!Controls.ContainsKey(state))

{

//Initial state case

if (parent == "")

{

Controls.Add(new Node(state));

Controls[state].Location = new Point(50, Height / 2 - 64);

}

//Spawn relative to parent node

else

{

Controls.Add(new Node(state));

Point parentLocation = Controls[parent].Location;

List<string> parentNeighbours = new List<string>(graph.neighbours[parent]);

parentNeighbours.Remove(parent);

if (parentNeighbours.Count > 1)

{

int maxHeight = (parentNeighbours.Count - 1) / 2;

int IndexInList = parentNeighbours.FindIndex(i => i == state);

int X = parentLocation.X + spacing;

int Y;

if (parentNeighbours.Count % 2 == 0)

{

Y = parentLocation.Y + spacing \* IndexInList - spacing / 2 - spacing \* maxHeight;

}

else

{

Y = parentLocation.Y + spacing \* IndexInList - spacing \* maxHeight;

}

Controls[state].Location = new Point(X, Y);

}

else

{

int X = parentLocation.X + spacing;

int Y = parentLocation.Y;

Controls[state].Location = new Point(X, Y);

}

}

}

//If state is halting

if (graph.statesHalting.Contains(Controls[Controls.Count - 1].Name))

{

((Node)Controls[Controls.Count - 1]).SetHalting();

}

//Spawn each neighbour node + their children

if (graph.neighbours.ContainsKey(state))

{

foreach (string neighbour in graph.neighbours[state])

{

if (!Controls.ContainsKey(neighbour))

{

Spawn(neighbour, state);

}

}

}

}

//Refresh edge background

private void RefreshEdges(object sender, EventArgs e)

{

foreach(Control control in Controls)

{

if(control is EdgeCanvas)

{

control.Refresh();

}

}

}

}

}

## Graphics

### Graph.cs

using System;

using System.Collections.Generic;

using System.Text.RegularExpressions;

namespace TuringMachine

{

public class Graph

{

public Dictionary<string, Dictionary<string, string>> graph { get; } =

new Dictionary<string, Dictionary<string, string>>();

public Dictionary<string, List<string>> neighbours { get; } = new Dictionary<string, List<string>>();

public List<string> statesHalting { get; } = new List<string>();

bool isReadOnly = false;

public Graph(string text)

{

string[] lines = text.Split(new char[] { '\r', '\n' }, StringSplitOptions.RemoveEmptyEntries);

List<string> stateList = new List<string>();

for (int i = 0; i < lines.Length; i++)

{

if (lines[i][0] != '/' && lines[i][1] != '/')

{

string[] fields = lines[i].Split(' ');

if(i == 0)

{

if(fields.Length == 5)

{

isReadOnly = false;

}

else

{

isReadOnly = true;

}

}

//Add new pair in graph dictionary

if (!graph.ContainsKey(fields[0]))

{

graph.Add(fields[0], new Dictionary<string, string>());

}

//Add state to state list

if (!stateList.Contains(fields[0]))

{

stateList.Add(fields[0]);

}

//Add new pair in neighbour dictionary

if (!neighbours.ContainsKey(fields[0]))

{

neighbours.Add(fields[0], new List<string>());

}

if (neighbours.ContainsKey(fields[0]))

{

if (!neighbours[fields[0]].Contains(fields[fields.Length - 1]))

{

neighbours[fields[0]].Add(fields[fields.Length - 1]);

}

}

//Create information for state transition

string state = fields[0];

string hstate = fields[fields.Length - 1];

string terms = "";

//For read-only machines

if (fields.Length == 4)

{

terms += fields[1] + " | ";

if (fields[2] == "r")

{

terms += ">";

}

else if (fields[2] == "l")

{

terms += "<";

}

else

{

terms += "-";

}

}

//For all other machines

else

{

terms += fields[1] + " | ";

terms += fields[2] + " ";

foreach (char c in fields[3])

{

if (c == 'r')

{

terms += ">";

}

else if (c == 'l')

{

terms += "<";

}

else

{

terms += "-";

}

}

}

try

{

graph[state].Add(fields[fields.Length - 1], terms);

}

//If two transitions from the same state lead to the same state

catch (Exception)

{

string newLine;

if (fields.Length == 5)

{

newLine = ReassembleLine(graph[state][fields[fields.Length - 1]], fields[1], fields[2], fields[3], false);

}

else

{

newLine = ReassembleLine(graph[state][fields[fields.Length - 1]], fields[1], fields[2], fields[2], true);

}

graph[state][fields[fields.Length - 1]] = newLine;

}

}

}

//Find what states are halting

List<string> haltingStateList = new List<string>();

for (int i = 0; i < lines.Length; i++)

{

if (lines[i][0] != '/' && lines[i][1] != '/')

{

string[] fields = lines[i].Split(' ');

if (!stateList.Contains(fields[fields.Length - 1]))

{

haltingStateList.Add(fields[fields.Length - 1]);

}

}

}

statesHalting = haltingStateList;

Dictionary<string, Dictionary<string, string>> newGraph = new Dictionary<string, Dictionary<string, string>>();

foreach (KeyValuePair<string, Dictionary<string, string>> connection in graph)

{

Dictionary<string, string> newConnection = new Dictionary<string, string>();

foreach (KeyValuePair<string, string> edge in connection.Value)

{

bool isSimplified = false;

//Check if all values the same

int i = 0;

string previous = "";

List<string> ReadKeys = GetReadKeys(edge.Value, ref i);

i++;

List<string> WriteKeys = new List<string>();

if (!isReadOnly)

{

WriteKeys = GetWriteKeys(edge.Value, ref i);

previous = WriteKeys[0];

bool simplifyWriteKeys = false;

for (int j = 1; j < WriteKeys.Count; j++)

{

if (WriteKeys[j] != previous)

{

simplifyWriteKeys = false;

break;

}

else

{

simplifyWriteKeys = true;

}

}

if (simplifyWriteKeys)

{

string writeKey = WriteKeys[0];

WriteKeys = new List<string>();

WriteKeys.Add(writeKey);

isSimplified = true;

}

}

List<string> MoveKeys = GetMoveKeys(edge.Value, ref i);

previous = MoveKeys[0];

bool simplifyMoveKeys = false;

for (i = 1; i < MoveKeys.Count; i++)

{

if (MoveKeys[i] != previous)

{

simplifyMoveKeys = false;

break;

}

else

{

simplifyMoveKeys = true;

}

}

if (simplifyMoveKeys)

{

string moveKey = MoveKeys[0];

MoveKeys = new List<string>();

MoveKeys.Add(moveKey);

}

//Check if all values stay the same

if (!isSimplified && !isReadOnly)

{

bool isMatch = false;

for (i = 0; i < ReadKeys.Count; i++)

{

if (ReadKeys[i] == WriteKeys[i])

{

isMatch = true;

}

else if (Regex.IsMatch(WriteKeys[i], @"(\\*)+"))

{

isMatch = true;

}

else

{

isMatch = false;

}

}

if (isMatch)

{

WriteKeys = new List<string>();

}

}

newConnection.Add(edge.Key, AssembleLine(ReadKeys, WriteKeys, MoveKeys, isReadOnly));

}

newGraph.Add(connection.Key, newConnection);

}

graph = newGraph;

}

//Recreate line if multiple transitions go to the same state from the same state

string ReassembleLine(string existingLine, string readKey, string writeKey, string moveKey, bool isReadOnly)

{

int i = 0;

List<string> ReadKeys = GetReadKeys(existingLine, ref i);

i++;

List<string> WriteKeys = new List<string>();

if (!isReadOnly)

{

WriteKeys = GetWriteKeys(existingLine, ref i);

}

List<string> MoveKeys = GetMoveKeys(existingLine, ref i);

//Add new fields to lists

ReadKeys.Add(readKey);

WriteKeys.Add(writeKey);

string convertedMoveKey = "";

foreach (char c in moveKey)

{

if (c == 'r')

{

convertedMoveKey += ">";

}

else if (c == 'l')

{

convertedMoveKey += "<";

}

else

{

convertedMoveKey += "-";

}

}

MoveKeys.Add(convertedMoveKey);

return AssembleLine(ReadKeys, WriteKeys, MoveKeys, isReadOnly);

}

List<string> GetReadKeys(string line, ref int i)

{

List<string> ReadKeys = new List<string>();

string readKey = "";

i = 0;

while (line[i] != '|')

{

if (line[i] != ' ' && line[i] != ',')

{

readKey += line[i];

}

else

{

ReadKeys.Add(readKey);

readKey = "";

}

i++;

}

return ReadKeys;

}

List<string> GetWriteKeys(string line, ref int i)

{

List<string> WriteKeys = new List<string>();

string writeKey = "";

while (line[i] != '>' && line[i] != '<' && line[i] != '-')

{

if (line[i] != ' ' && line[i] != ',')

{

writeKey += line[i];

}

else

{

if (writeKey != "")

{

WriteKeys.Add(writeKey);

}

writeKey = "";

}

i++;

}

return WriteKeys;

}

List<string> GetMoveKeys(string line, ref int i)

{

List<string> MoveKeys = new List<string>();

string moveKey = "";

while (i <= line.Length)

{

if (i == line.Length)

{

if (moveKey != "")

{

MoveKeys.Add(moveKey);

}

moveKey = "";

}

else if (line[i] != ' ' && line[i] != ',')

{

moveKey += line[i];

}

else

{

if (moveKey != "")

{

MoveKeys.Add(moveKey);

}

moveKey = "";

}

i++;

}

return MoveKeys;

}

string AssembleLine(List<string> ReadKeys, List<string> WriteKeys, List<string> MoveKeys, bool isReadOnly)

{

string newLine = "";

for (int j = 0; j < ReadKeys.Count; j++)

{

newLine += ReadKeys[j];

if (j != ReadKeys.Count - 1)

{

newLine += ",";

}

else

{

newLine += " | ";

}

}

if (!isReadOnly)

{

for (int k = 0; k < WriteKeys.Count; k++)

{

newLine += WriteKeys[k];

if (k != WriteKeys.Count - 1)

{

newLine += ",";

}

else

{

newLine += " ";

}

}

}

for (int l = 0; l < MoveKeys.Count; l++)

{

newLine += MoveKeys[l];

if (l != MoveKeys.Count - 1)

{

newLine += ",";

}

}

return newLine;

}

}

}

### Node.cs

using System.Windows.Forms;

using System.Drawing;

using System.IO;

namespace TuringMachine

{

public class Node : PictureBox

{

public Node(string state)

{

Width = 128;

Height = 128;

SizeMode = PictureBoxSizeMode.StretchImage;

Anchor = AnchorStyles.Left;

BackColor = Color.White;

SendToBack();

Label stateLabel = new Label()

{

Text = state,

Font = new Font("Consolas", 15.0f),

TextAlign = ContentAlignment.MiddleCenter,

Anchor = AnchorStyles.Top | AnchorStyles.Left,

AutoSize = true

};

Name = state;

Controls.Add(stateLabel);

stateLabel.Location = new Point((Width - stateLabel.Width) / 2, (Height - stateLabel.Height) / 2);

string path = Path.GetDirectoryName(Application.StartupPath) + "\\Debug\\state\_circle.png";

Image = new Bitmap(path);

}

public void SetHalting()

{

string path = Path.GetDirectoryName(Application.StartupPath) + "\\Debug\\state\_circle\_halting.png";

Image = new Bitmap(path);

}

}

}

### EdgeCanvas.cs

using System;

using System.Drawing;

using System.Windows.Forms;

using System.Collections.Generic;

namespace TuringMachine

{

public partial class EdgeCanvas : Panel

{

Graph graph;

public EdgeCanvas()

{

InitializeComponent();

}

public EdgeCanvas(Graph graph)

{

this.graph = graph;

InitializeComponent();

}

//Create graphics object and define pen, load draw method

private void PaintEvent(object sender, PaintEventArgs e)

{

Graphics g = e.Graphics;

Pen p = new Pen(Color.Black, 8)

{

EndCap = System.Drawing.Drawing2D.LineCap.ArrowAnchor

};

FiniteStateMachine parent = (FiniteStateMachine)((EdgeCanvas)sender).Parent;

int x = parent.HorizontalScroll.Value;

int y = parent.VerticalScroll.Value;

//Create edges and labels

DrawEdge(g, p, parent, x, y);

}

private void DrawEdge(Graphics g, Pen p, FiniteStateMachine parent, int xOffset, int yOffset)

{

//Offset initial point by scroll amount

Point initial = parent.Controls[parent.initialState].Location;

initial.Offset(xOffset, yOffset);

initial = new Point(initial.X - 15, initial.Y + 64);

g.DrawLine(p, new Point(0, initial.Y), initial);

foreach (KeyValuePair<string, Dictionary<string, string>> transitionFunction in graph.graph)

{

string state = transitionFunction.Key;

Dictionary<string, string> info = transitionFunction.Value;

foreach (KeyValuePair<string, string> link in info)

{

string finalState = link.Key;

//Offset points by how much form is scrolled by

Point sender = parent.Controls[state].Location;

sender.Offset(xOffset, yOffset);

Point destination = parent.Controls[finalState].Location;

destination.Offset(xOffset, yOffset);

Point spawnPoint;

int dY = destination.Y - sender.Y;

int dX = destination.X - sender.X;

sender = new Point(sender.X + 64, sender.Y + 64);

//Make sure arrow points to centre of node

if (dX != 0)

{

int xDisplacement = 64;

double yDisplacement = xDisplacement \* ((double)dY / (double)dX);

destination = new Point(destination.X - 10 \* (dX / Math.Abs(dX)), destination.Y + 64 - Convert.ToInt32(yDisplacement));

}

else

{

destination = new Point(destination.X - 10, destination.Y + 64);

}

//Find which way to offset points

int sign = -1;

if (sender.Y < initial.Y)

{

sign = 1;

}

//Straight Line

if (dX > 0)

{

if (destination.Y < initial.Y)

{

sign = 1;

}

spawnPoint = new Point((sender.X + destination.X) / 2, (sender.Y + destination.Y) / 2);

spawnPoint = new Point(spawnPoint.X + 32, spawnPoint.Y);

if(dY != 0)

{

spawnPoint = new Point(spawnPoint.X + 32, spawnPoint.Y);

}

else

{

spawnPoint = new Point(spawnPoint.X, spawnPoint.Y - 32 \* sign);

}

g.DrawLine(p, sender, destination);

}

//Looping Line

else if(dX == 0 && dY == 0)

{

//Points for bezier curve

Point p0 = new Point(sender.X + 16, sender.Y - 70 \* sign);

Point p1 = new Point(sender.X + 16, sender.Y - 150 \* sign);

Point p2 = new Point(destination.X + 48, p1.Y);

Point p3 = new Point(destination.X + 48, destination.Y - 70 \* sign);

spawnPoint = new Point((p1.X + p2.X) / 2, (p2.Y));

g.DrawBezier(p, p0, p1, p2, p3);

}

//Arcing Line

else

{

//Points for bezier curve

Point p1 = new Point(sender.X, sender.Y - 300 \* sign);

Point p2 = new Point(destination.X + 64, p1.Y - 25 \* sign);

Point p3 = new Point(destination.X + 64, destination.Y - 100 \* sign);

spawnPoint = new Point((p1.X + p2.X) / 2, (p2.Y + 48 \* sign));

g.DrawBezier(p, sender, p1, p2, p3);

}

//Add Label on first drawing

//Don't if re-drawing

if (!parent.Controls.ContainsKey(state + finalState + graph.graph[state][finalState]))

{

parent.Controls.Add(new TransitionTag(state, finalState, link.Value, spawnPoint));

Control label = parent.Controls[parent.Controls.Count - 1];

label.BringToFront();

label.Location = new Point(label.Location.X - label.Width / 2, label.Location.Y - label.Height / 2);

}

}

}

}

}

}

### Label.cs

using System.Windows.Forms;

using System.Drawing;

namespace TuringMachine

{

class TransitionTag : Label

{

public TransitionTag(string state, string finalstate, string info, Point location)

{

Anchor = AnchorStyles.Left;

AutoSize = true;

BackColor = Color.White;

Font = new Font("Consolas", 11, FontStyle.Bold);

Name = state + finalstate + info;

Text = info;

Location = location;

}

private void InitializeComponent()

{

this.SuspendLayout();

//

// TransitionTag

//

this.BackColor = System.Drawing.Color.White;

this.ResumeLayout(false);

}

}

}

## UniversalMachine.cs

using System.Collections.Generic;

using System.Linq;

using System.Windows.Forms;

namespace TuringMachine

{

public static class UniversalMachine

{

//Generate code for UTM to emulate DTM

public static List<string> GenerateFunctions(string[] fields, char defaultCaseCharacter)

{

List<string> code = new List<string>();

List<char> stateList = new List<char>();

List<char> haltingStateList = new List<char>();

List<char> alphabet = new List<char>();

//Gather lists of states and characters

for (int i = 0; i < fields.Length; i++)

{

List<char> function = new List<char> {fields[i][0], fields[i][1], fields[i][2], fields[i][3], fields[i][4]};

//Only add initial states

if (!stateList.Contains(function[0]))

{

stateList.Add(function[0]);

}

//Add read and write keys

if (!alphabet.Contains(function[1]))

{

alphabet.Add(function[1]);

}

if (!alphabet.Contains(function[2]))

{

alphabet.Add(function[2]);

}

}

//Gather list of halting states

for (int i = 0; i < fields.Length; i++)

{

//Only add states that do not have any instructions

if (!stateList.Contains(fields[i][4]))

{

haltingStateList.Add(fields[i][4]);

}

}

#region TRANSITION FUNCTIONS

foreach (char state in stateList)

{

//Get current state from State Tape (Tape 1)

code.Add("q0 " + defaultCaseCharacter + state + defaultCaseCharacter + " " + defaultCaseCharacter + state + defaultCaseCharacter + " \_\_\_ q" + state);

code.Add("q" + state + " " + defaultCaseCharacter + defaultCaseCharacter + "\_ " + defaultCaseCharacter + defaultCaseCharacter + "\_ \_\_\_ q" + state + "-");

foreach (char key in alphabet)

{

//Get read key from Data Tape (Tape 2)

code.Add("q" + state + " " + defaultCaseCharacter + defaultCaseCharacter + key + " " + defaultCaseCharacter + defaultCaseCharacter + key + " \_\_\_ q" + state + key);

//Check if Instruction Tape (Tape 0) has correct state

code.Add("q" + state + key + " " + state + defaultCaseCharacter + defaultCaseCharacter + " " + state + defaultCaseCharacter + defaultCaseCharacter + " r\_\_ q" + state + key + "acceptstate");

//If not, go to next instruction

code.Add("q" + state + key + " " + defaultCaseCharacter + defaultCaseCharacter + defaultCaseCharacter + " " + defaultCaseCharacter + defaultCaseCharacter + defaultCaseCharacter + " r\_\_ q" + state + key + "reject");

//Go forward to next instruction

code.Add("q" + state + key + "reject |" + defaultCaseCharacter + defaultCaseCharacter + " |" + defaultCaseCharacter + defaultCaseCharacter + " r\_\_ q" + state + key);

code.Add("q" + state + key + "reject " + defaultCaseCharacter + defaultCaseCharacter + defaultCaseCharacter + " " + defaultCaseCharacter + defaultCaseCharacter + defaultCaseCharacter + " r\_\_ q" + state + key + "reject");

//If ending in a non-halting/non-accepting state

code.Add("q" + state + key + "reject \_" + defaultCaseCharacter + defaultCaseCharacter + " \_" + defaultCaseCharacter + defaultCaseCharacter + " \_\_\_ qhalt\_reject");

//Check if Instruction Tape (Tape 0) has correct read key

code.Add("q" + state + key + "acceptstate " + key + defaultCaseCharacter + defaultCaseCharacter + " " + key + defaultCaseCharacter + defaultCaseCharacter + " r\_\_ qwrite");

//If not, go to next instruction

code.Add("q" + state + key + "acceptstate " + defaultCaseCharacter + defaultCaseCharacter + defaultCaseCharacter + " " + defaultCaseCharacter + defaultCaseCharacter + defaultCaseCharacter + " r\_\_ q" + state + key + "reject");

}

//Set new state at end to State Tape (Tape 1)

code.Add("qstate " + state + defaultCaseCharacter + defaultCaseCharacter + " " + state + state + defaultCaseCharacter + " l\_\_ qreset");

}

//If ending in a halting/accepting state

foreach (char haltingState in haltingStateList)

{

code.Add("qstate " + haltingState + defaultCaseCharacter + defaultCaseCharacter + " " + haltingState + haltingState + defaultCaseCharacter + " r\_\_ qhalt\_accept");

}

//Write to Data Tape (Tape 2) variations

foreach (char key in alphabet)

{

if (key == '-')

{

code.Add("qwrite " + key + defaultCaseCharacter + defaultCaseCharacter + " " + key + defaultCaseCharacter + "\_ r\_\_ qmove");

}

else

{

code.Add("qwrite " + key + defaultCaseCharacter + defaultCaseCharacter + " " + key + defaultCaseCharacter + key + " r\_\_ qmove");

}

}

//Move Data Tape (Tape 2) variations

code.Add("qmove r" + defaultCaseCharacter + defaultCaseCharacter + " r" + defaultCaseCharacter + defaultCaseCharacter + " r\_r qstate");

code.Add("qmove l" + defaultCaseCharacter + defaultCaseCharacter + " l" + defaultCaseCharacter + defaultCaseCharacter + " r\_l qstate");

code.Add("qmove -" + defaultCaseCharacter + defaultCaseCharacter + " -" + defaultCaseCharacter + defaultCaseCharacter + " r\_\_ qstate");

//Reset after instruction

code.Add("qreset " + defaultCaseCharacter + defaultCaseCharacter + defaultCaseCharacter + " " + defaultCaseCharacter + defaultCaseCharacter + defaultCaseCharacter + " l\_\_ qreset");

code.Add("qreset \_" + defaultCaseCharacter + defaultCaseCharacter + " \_" + defaultCaseCharacter + defaultCaseCharacter + " r\_\_ q0");

#endregion

code = code.Distinct().ToList();

return code;

}

public static List<string> ConvertMachine(ref MachineData currentMachineData, List<TapePanel> TapeList, ref RichTextBox code, char defaultCaseCharacter)

{

List<string> Tapes = new List<string>();

//Dictionary to link DTM state to single-character UTM state

char state = 'a';

Dictionary<string, char> stateDictionary = new Dictionary<string, char>();

//Saves for tape 3 in UTM

string oldTape = TapeList[0].GetTape().Text;

string newTape = "";

List<string> lines = code.Lines.ToList();

lines.RemoveAll(i => i == "");

for (int i = 0; i < lines.Count; i++)

{

if (lines[i][0] != '/' && lines[i][1] != '/')

{

string[] fields = lines[i].Split(' ');

if (fields[1][0] == defaultCaseCharacter)

{

MessageBox.Show("Cannot use the default case character in UTM tapes.");

return null;

}

if (fields.Length == 5)

{

if (fields[2] == defaultCaseCharacter.ToString())

{

fields[2] = fields[1];

}

}

if (lines[i] != "")

{

//Collects states

if (!stateDictionary.ContainsKey(fields[0]))

{

if (fields[0].Length == 1)

{

stateDictionary.Add(fields[0], fields[0][0]);

}

else

{

stateDictionary.Add(fields[0], state);

state = (char)(state + 1);

}

}

if (!stateDictionary.ContainsKey(fields[fields.Length - 1]))

{

if (fields[0].Length == 1)

{

stateDictionary.Add(fields[fields.Length - 1], fields[fields.Length - 1][0]);

}

else

{

stateDictionary.Add(fields[fields.Length - 1], state);

state = (char)(state + 1);

}

}

if (state > 'z')

{

state = 'A';

}

//Add instruction with simplified states

string field;

if (currentMachineData.GetMachine() == 0)

{

field = stateDictionary[fields[0]] + fields[1] + fields[2] + fields[3] + stateDictionary[fields[4]];

}

else

{

field = stateDictionary[fields[0]] + fields[1] + fields[1] + fields[2] + stateDictionary[fields[3]];

}

newTape += field;

if (i != lines.Count - 1)

{

newTape += "|";

}

}

}

}

newTape = newTape.Replace('\_', '-');

if (stateDictionary.Count > 52)

{

MessageBox.Show("Too many states to convert into a UTM. Limit is 52.");

return null;

}

//Empty code box

code.Text = "";

//Get machine to Universal

//Set tapes

Tapes.Add(newTape);

Tapes.Add("" + stateDictionary[currentMachineData.GetInitialState(0)]);

Tapes.Add(oldTape);

return Tapes;

}

}

}

## Settings.cs

using System;

using System.Windows.Forms;

namespace TuringMachine

{

public partial class OptionsMenu : Form

{

public int MaxSpeed = 0;

public int MinSpeed = 0;

public char defaultKey = '\*';

public int fullSpeedInterval = 50;

public OptionsMenu(int MaxSpeed, int MinSpeed, char defaultKey, int fullSpeedInterval)

{

InitializeComponent();

speedMax.Value = MaxSpeed;

speedMin.Value = MinSpeed;

defaultKeyBox.Text = defaultKey.ToString();

fullSpeedIntervalBox.Value = fullSpeedInterval;

}

//Get data to return to main form

private void CollectData(object sender, EventArgs e)

{

if (ValidateOptions())

{

MaxSpeed = (int)speedMax.Value;

MinSpeed = (int)speedMin.Value;

defaultKey = defaultKeyBox.Text[0];

fullSpeedInterval = (int)fullSpeedIntervalBox.Value;

DialogResult = DialogResult.OK;

Close();

}

else

{

DialogResult = DialogResult.None;

}

}

//Make sure data entered is valid

private bool ValidateOptions()

{

if (speedMax.Value < speedMin.Value)

{

MessageBox.Show("Cannot have a smaller maximum speed value than minimum value.");

return false;

}

else if(speedMax.Value == speedMin.Value)

{

MessageBox.Show("Two speeds cannot be equal.");

return false;

}

if(defaultKeyBox.Text.Length == 0)

{

MessageBox.Show("Default case character not specified.");

return false;

}

else if(defaultKeyBox.Text == " " || defaultKeyBox.Text == "\_")

{

MessageBox.Show("Default case character cannot be the blank character.");

return false;

}

else

{

return true;

}

}

}

}

## InfoWindow.cs

using System;

using System.Windows.Forms;

namespace TuringMachine

{

public partial class InfoWindow : Form

{

public InfoWindow()

{

InitializeComponent();

}

private void Close(object sender, EventArgs e)

{

Close();

}

}

}

## RenameWindow.cs

using System.Windows.Forms;

namespace TuringMachine

{

public partial class RenameWindow : Form

{

public RenameWindow()

{

InitializeComponent();

}

}

}