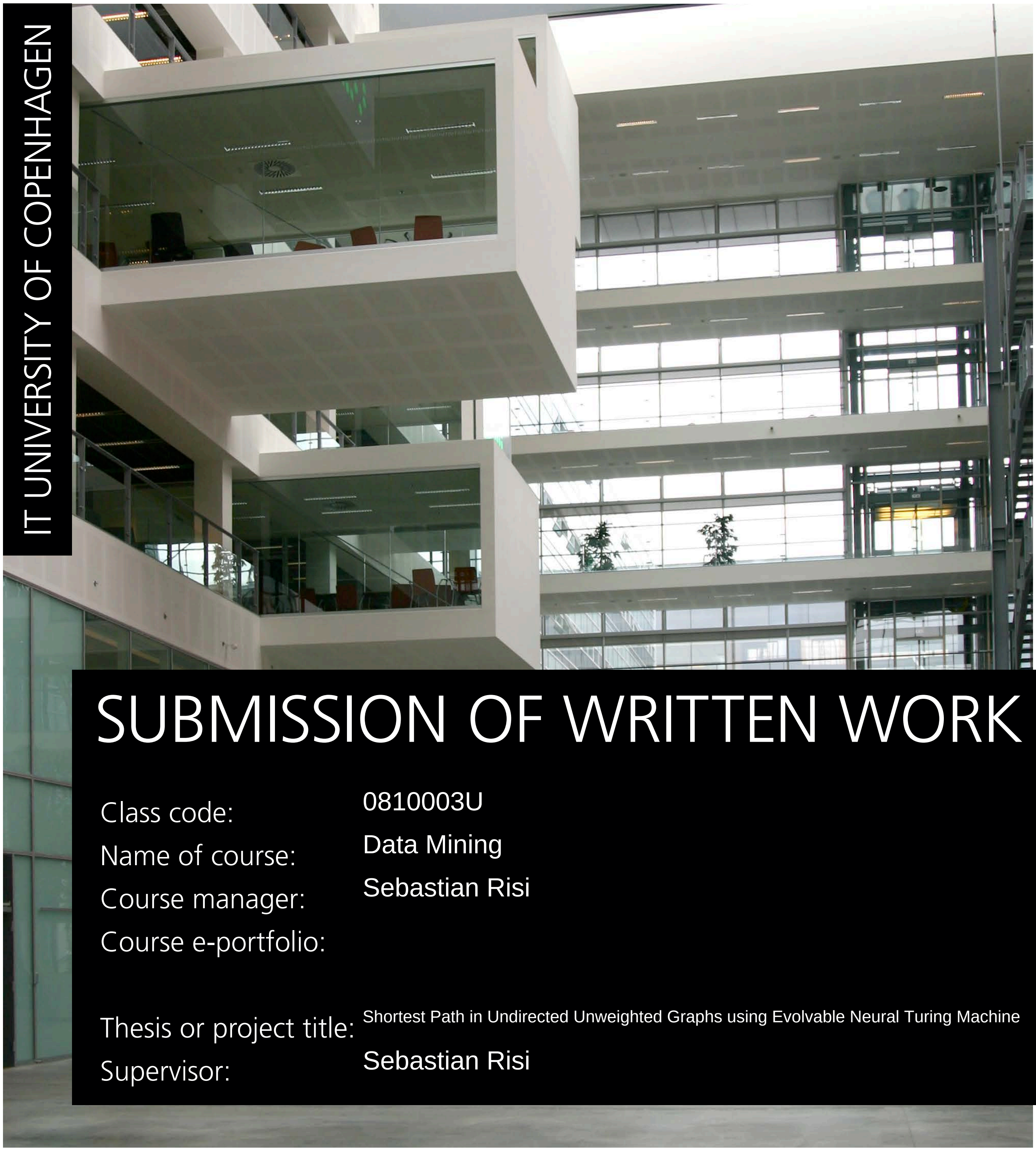


SUBMISSION OF WRITTEN WORK

Class code:	0810003U
Name of course:	Data Mining
Course manager:	Sebastian Risi
Course e-portfolio:	
Thesis or project title:	Shortest Path in Undirected Unweighted Graphs using Evolvable Neural Turing Machine
Supervisor:	Sebastian Risi

	Full Name:	Birthdate (dd/mm-yyyy):	E-mail:
1.	Anders Wind Steffensen	10/02-1993	awis@itu.dk
2.	Mikael Lindemann	17/02/1992	mlin@itu.dk
3.			@itu.dk
4.			@itu.dk
5.			@itu.dk
6.			@itu.dk
7.			@itu.dk

Shortest Path in Undirected Unweighted Graphs using Evolvable Neural Turing Machine

Anders Wind Steffensen (awis@itu.dk)
Mikael Lindemann (mlin@itu.dk)

Supervisor: Sebastian Risi (sebr@itu.dk)

May 18, 2017

Contents

1 Introduction

Finding the shortest path in a graph is one of the most well studied algorithmic graph problems. Solutions for the problem origin back to the late 1950's. Finding the shortest path has a long list of applications from transport and routing tables in networks to being an essential subroutine of many more advanced graph algorithms. In recent years neural networks has gained more and more traction in the artificial intelligence and machine learning fields. This increased focus on neural networks is both due to theoretical breakthroughs but also the more widespread adoption and advancement of graphical processing units. Neural networks are most often used for classification, finding heuristics and as the decision mechanism for artificial agents.

One of the most criticized problems of neural networks and machine learning in general is that agents are inherently single purpose and problems different from what is trained on is often not handled very well.

With the introduction of neural networks with memory more domains should be feasibly solvable. In [2] a differentiable neural computer was shown to be able to find the shortest path in the London underground by using memory to store variables and information about the graph. In [3] an evolutionary version of the neural Turing machine was introduced. Evolutionary neural networks does not require every part of the problem to be differentiable and allows reward based learning.

In this report we want to examine how well these evolutionary neural Turing machines can handle the shortest path problem on undirected, unweighted graphs. Both acyclic and recurrent neural networks will be examined.

The source code of the experimentation can be found at
www.github.com/Awia00/Data-Mining.

2 Background

2.1 Graphs

A *graph* $G(V, E)$ is data-structure which consists of a set of *vertices* V and a set of *edges* E where an edge is a pair of vertices (u, v) where $u, v \in V$. An *undirected graph* implies that if edge (u, v) exists then (v, u) exists in the graph. In an *unweighted graph* there is no cost distinction between edges. Two vertices are *adjacent* if they share an edge e. A *path* P is a ordered set of edges such that for every two consecutive edges e and f, $e = (u, v)$, $f = (v, w)$. To vertices u, v are *connected* if there exist a path between u and v. The term *graph* will be used to denote undirected unweighted graphs in the remainder of the report.

In a graph the length of a path P is $|P|$. A shortest path between two vertices u, v is a path P connecting u, v of minimum length. The Breadth First Search

(BFS) algorithm can find the shortest path in time $\mathcal{O}(n+m)$ where $n = |V|$ and $m = |E|$. BFS uses a queue to store vertices in the fringe and for each vertex a pointer to the vertex that comes before it on the shortest path.

2.2 Neural Networks

A *neural network* is an approximation method for n dimensional continuous functions. Neural networks consists of a ordered set of *layers* $L_1..L_n$ where L_1 is the *input layer*, $L_2..L_{n-1}$ are *hidden layers*, and L_n is the *output layer*. Each layer L_i consist of a number of *neurons*.

The *topology* of a neural network describes the number of layers, the number of neurons in each layer, and how neurons are connected. In an *acyclic* topology each neuron is connected to other neurons only in the layers before and after its own layer. In a *recurrent* topology neurons are allowed to be connected to itself, other neurons in the same layer, or neurons of any other layer in the network.

Neurons can send signals to other neurons it is connected to and the strength of a signal is specified by a *signal function*. A neuron is activated if the sum of the signals sent to it reaches a threshold. The signal function is based on a *formula*, a *weight* and a *bias*.

A *one hot encoding* of a nominal data point, is a conversion of the data point into a bit array A with the length n, where n is the number of possible nominal values. Each of the nominal values are then assigned a label $i \in \{0..n-1\}$. For a nominal value with label i only $A[i]$ is 1, all other elements in A is 0. This encoding is often used for neural networks, as each element of A can be used as the input for a single neuron in the input layer.

2.3 Neuroevolution

Neuroevolution is a method for training the weights and in some cases the topology of a neural network based on evolutionary algorithms. Instead of using supervised learning, neuroevolution uses reward based learning where a fitness function describes how well the network does in a specific instance. In contrast to gradient descent which minimizes errors, neuroevolution uses biologically inspired selection methods such as *mutation* and *reproduction* to optimize the fitness of a *population* of networks. The best performing network in a population is declared the *champion*.

2.3.1 NeuroEvolution of Augmenting Topologies (NEAT)

NEAT is introduced by Stanley and Miikkulainen in [5] as a new method for neuroevolution. In NEAT a neural network is called a *phenotype*. NEAT uses the concept of *genes* to both describe neurons, and connections between them. A *genome* consists of genes where some of them might be deactivated. The active genes of a genome corresponds to a phenotype.

To improve over previous methods of neuroevolution NEAT uses: Innovation numbering, speciation, and incremental growth from minimal structure.

Innovation numbering uniquely labels a gene to distinguish or compare genes across genomes. The technique is used both in reproduction and when grouping the genomes into *species*.

The notion of species is used to protect innovative but unoptimized structures in new genomes. The phenotype of a genome with a newly introduced gene, might perform worse due to unoptimized weights, even if the new gene is useful for further optimization. Instead of having all genomes compete against each other, they are divided into species, where only genomes inside a specie compete against each other. This allows the innovative mutations to last some generations, so they can prove, or disprove, their worth.

2.4 Evolvable Neural Turing Machine (ENTM)

The *Neural Turing Machine* (NTM, [1]) introduces a fixed size memory bank which allows the network to store and recall information in each activation. This allows the network to describe variables and therefore to approximate algorithms.

In [3] Greve et. al. introduce the *Evolvable Neural Turing Machine* as an extension of the NTM with two main differences.

First of all ENTM does not require the network to read through the entire memory bank at each time-step which entails that the memory bank can be theoretically unlimited. Second, ENTM uses NEAT to optimize fitness where the NTM uses gradient descent.

The ENTM addresses the memory with the following four instructions:

1. Write - Writes the content of the write vector to the current position in the memory bank.
2. Content Jump - Moves the read/write head to the memory bank position that is most similar to the write vector.
3. Shift - The read/write head can be shifted one position to the left or right, or it can be kept at the current position.
4. Read - The content of the current position is used as input for the next artificial neural network in the next time-step.

Greve et. al. show that the ENTM is able to solve simple algorithmic tasks such as the copy task, and the continuous T-maze task.

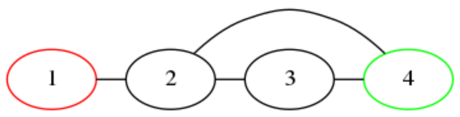
3 Experiment

The problem definition is as follows: Given a graph G, a source vertex s, and a goal vertex t, find the shortest path from s to t in G.

We have restricted the problem to instances where a path between s and t exists.

3.1 Instances

In order to feed the instance into the network, we use an encoding of the adjacency matrix of the graph, where each edge has the value 1, and each non-edge has the value 0. The source and goal vertices are one-hot encoded. See Figure 1 for an example. The output of the ENTM is interpreted as a one-hot encoding of the next vertex on the path to the goal. After each step the source vertex is updated to the previously outputted vertex in the new input to the ENTM.



(a)

	1	**2**	**3**	**4**
1	0	1	0	0
2	1	0	1	1
3	0	1	0	1
4	0	1	1	0

(b)

0	1	0	0	1	0	1	1	0	1	0	1	0	1	1	0	1	0	0	0	0	0	0	1

(c)

Figure 1: (a) Shows a simple graph, (b) its adjacency matrix and (c) an encoding of the entire instance where $s = 1$ and $t = 4$. The array contains the flattened adjacency table, the one-hot encoding of s, and the one-hot encoding of t

The instances are generated in a manner which tries to avoid over fitting and support general solutions. This is done by using randomly generated graphs with certain restrictions. The graph must contain a path of a given length to ensure a certain level of complexity. This level of complexity filters out optimal strategies on non general case graphs such as graphs with paths of length 1 where the optimal solution is outputting the goal vertex. Generating these graphs is done by picking two random vertices s and t and randomly pick vertices that forms a path from s to t. The remaining vertices are then connected to the set of explored vertices randomly. This ensures that the ENTM cannot just follow edges but need to explore branches of the graph. The generated graphs are of fixed size to ensure neural network compatibility.

3.2 Fitness

The fitness function is defined as a summation of scores for the output of each step on the shortest path. The function $DistTo(u)$ is defined to return the shortest distance from vertex u to the goal vertex. The vertex *current* describes the vertex which was given to the network and *next* denotes the network's choice of the next vertex along the path to the goal. The following scores are given in each step:

1 point: if $DistTo(next) > DistTo(current)$

2 points: if $DistTo(next) = DistTo(current)$

4 points: if $DistTo(next) < DistTo(current)$

In the case that one step resulted in a move where no edge exists the overall score is set to 0, and no further steps are performed.

Note that the instance generation defined above does not create any cycles and therefore the score 2 will never be given.

The motivation for these scores is to reward neural networks that understand the underlying graph problem more than a network that chooses arbitrarily.

The final score is normalized by the following formula $\frac{score}{DistTo(source)*4}$ where the denominator is the maximum score. Note that $DistTo(source)$ is the maximum number of moves allowed.

3.3 The Experiments

To assess and measure how well the shortest path problem is solved, four different types of experiments are performed. The first experiment uses the NEAT framework to train and build neural networks with recurrent connections. The next experiment uses the ENTM extension of the NEAT framework using recurrent neural networks. The memory is unlimited (within the bounds of the host machine), the write vector size is 10 and the default jump mechanism introduced in [4] is used. The two experiments are repeated on acyclic topologies.

All experiments uses the exact same configuration for NEAT and runs on the same instances to make them comparable. Each configuration uses a population of 500 divided into 30 species. Each phenotype is tested on 25 unique graphs per generation. All graphs have 10 vertices and 9 edges, of which 5 are on the path between the source and the goal. An example configuration can be seen in Appendix A. The appendix also contains the changes between the different experiments.

4 Results

The results of the experiments can be seen on Figures 2, 3, 4 and 5. The x axis shows the generation, and the y axis shows the champion fitness achieved for that generation in the run.

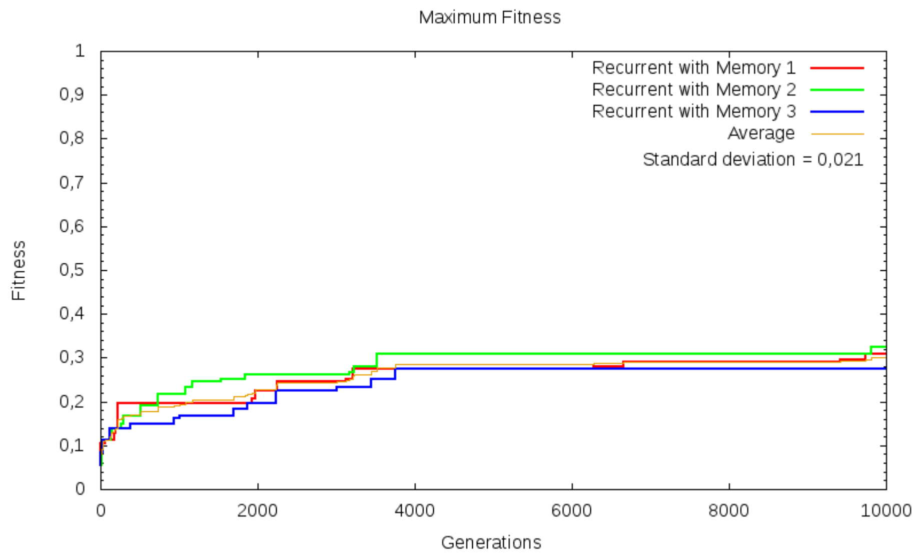


Figure 2: Graph of the results when using recurrent networks with enabled memory bank

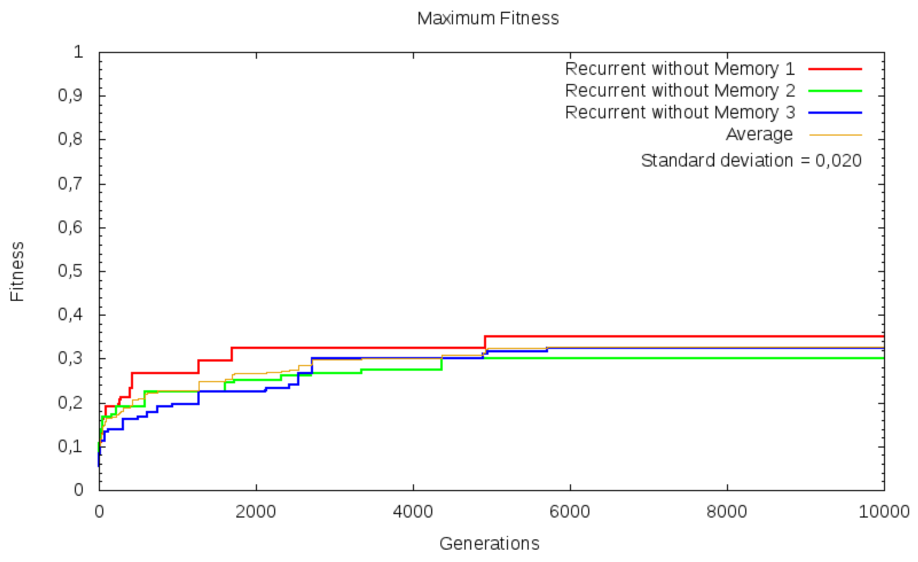


Figure 3: Graph of the results when using recurrent networks with disabled memory bank.

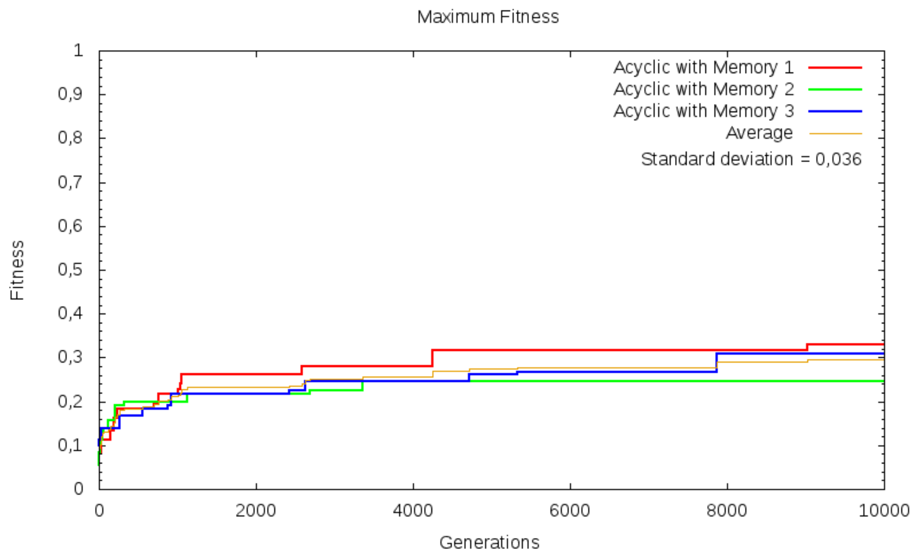


Figure 4: Graph of the results when using acyclic networks with enabled memory bank.

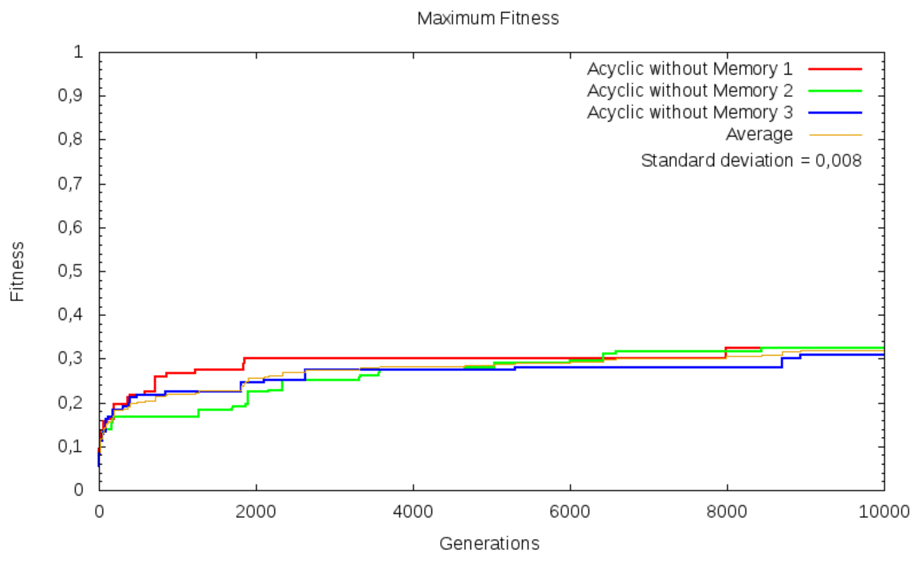


Figure 5: Graph of the results when using acyclic networks with disabled memory bank.

To compare the different experiments their averages are shown in Figure 6. At generation 10.000 the averages show that both kinds of topologies find a better performing neural network when the Turing-functionality is turned off by a small margin of around 2%.

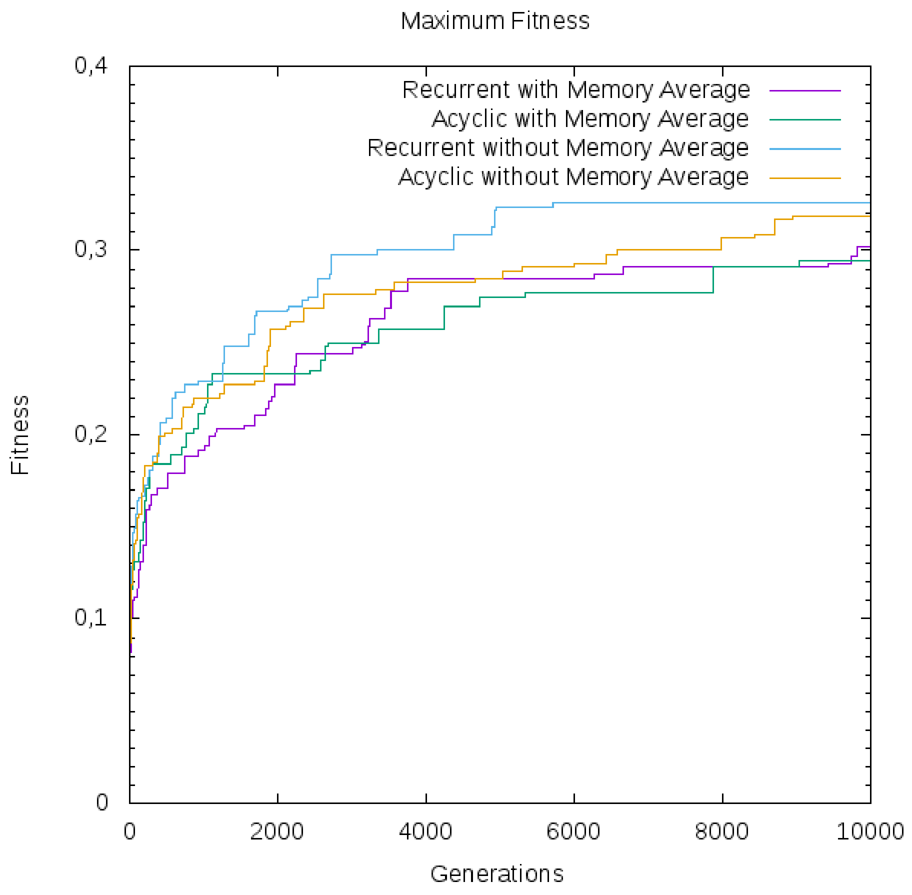


Figure 6: Averages over three runs of all configurations. Note that the y-axis has been truncated to make it easier to distinguish the different runs.

5 Analysis

Recurrent and acyclic neural networks as well as Turing enabled and disabled networks never (in 10.000 generations) achieve higher fitness than approximately 0.35. To put the value into perspective the result is compared to three suboptimal strategies.[1] A strategy where moves are picked randomly including non-edges will result in an average fitness of 0.000126. A strategy which randomly picks legal moves will get an average fitness of 0.67 and a strategy which always moves away from the goal will get a fitness of 0.25. The span between these numbers are great, and it is clear that as soon as a strategy learns to not pick non-edges the fitness greatly increase. The observation is that the champion must have some understanding of the graph but only enough so that it most of the time does not pick non-edges. The strategy of always moving away from the goal is in some sense just as hard a problem as the shortest path to the goal, therefore it is not likely that the champion uses this strategy.

To further analyse the champions we examine the behaviour of the champion on a collection of graphs. The champions seem to pick a lot of non edges but in a lot of instances the weights of the output contains high values for multiple decisions. In Table 1 an action of one of the champions with memory can be seen. It decides to move to vertex 3 which is faulty and results in score 0 on the entire instance. But it is interesting to see that the only valid move of the instance also has a high score which only differs in the 5th decimal. This supports the indication that the network in general has higher chances of picking correct moves than a totally uniform random strategy.

Move:	decision	score for decision
0	0.5	0
1	0.5	0
2	0.5	0
3	0.99999	0
4	0.5	0
5	0.5	0
6	0.99392	0
7	0.99994	4
8	0.5	0
9	0.5	0

Table 1: The output array of a network. The network is on vertex 1 goal is 0. Should move to 7 as it is the only possible edge. Chooses to move to 3 which is connected to 0.

The champions seem to be able to handle some of the graphs better than others.

[1]We have included an Excel sheet to the submission of the report, where the calculations can be seen. The file is named Strategies.xlsx.

In a lot of instances the network picks non edges, while in others, the network gains a very high fitness. This results in a comparatively high fitness in average, but indicates that the ENTM has not been able to generalize and might be over fitted on some of the graphs. To overcome this problem training the networks with more than 25 instances would be beneficial.

One possible reason for the bad results, might be that the encoding of the problem is not feasible for the ENTM. If we compare our encoding to that of the Copy Task from [3], they made sure to signal the network when starting a new assignment and when asking it to return the content back. Also, they supplied the input in multiple time-steps rather than giving the entire sequence all at once. Preliminary experiments on this type of encoding however, have not indicated substantial improvements over the current configuration, but further research has to be done to conclude anything on the subject.

The shortest path problem differ from the Copy task in that the memory bank should not only be used to store content as provided, it should be used to model some structure that would allow for the planning problem to be performed. Since this task has to be learned by a part of the neural network, it seems to be far more advanced, compared to the copy task. Because every iteration requires an output of the network, the ENTM is not allowed to process the input more than once. Since the memory bank is only read from at the beginning of a step and written to at the end of a step, the network cannot use the memory for preliminary results. Further experiments might try to feed the input in multiple steps, and train the network to signal when it is ready to answer on the shortest path problem. Another experiment could change the ENTM implementation to allow the network to read and write memory multiple times during a step.

6 Conclusion

We were unsuccessful in training an Evolutionary Neural Turing Machine to find the shortest path in a undirected unweighted graph. On graphs with 10 vertices and 9 edges, where the shortest path between the source and target has a length of 5, only a fitness value of ≈ 0.35 was achieved. This result outperforms a strategy of randomly picking moves by a large factor but is only half as good as randomly picking legal moves. The network was not able to fully learn how to recognize edges in the graph but indicated higher values for the correct move than the average case. The difference between the neural Turing machines and the regular neural networks was insignificant and in average the networks without memory achieved a higher fitness.

There is a lot of possible extensions and areas left unexplored which could provide more interesting results. Allowing the network to read and write memory multiple times in one activation or changing the structure of experiments might allow the networks to better plan ahead. Also, the ENTM might need a few intermediate activations in order to update the memory bank, before it is possible for it to give the correct result in subsequent time-steps.

7 References

[1] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[2] Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwińska, Sergio Gómez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John Agapiou, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.

[3] Rasmus Boll Greve, Emil Juul Jacobsen, and Sebastian Risi. Evolving neural turing machines for reward-based learning. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference*, pages 117–124. ACM, 2016.

[4] Benno Lüders, Mikkel Schläger, Aleksandra Korach, and Sebastian Risi. Continual and one-shot learning through neural networks with dynamic external memory. In *European Conference on the Applications of Evolutionary Computation*, pages 886–901. Springer, 2017.

[5] Kenneth O Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

A SharpNEAT Configuration Files

A.1 Configuration with memory and recurrent networks

```
<?xml version="1.0" encoding="utf-8" ?>
<Config>
  <!-- The name of the experiment -->
  <Name>Shortest Path Task With Paths only (10,5) with memory</Name
      >
  <!-- The fully qualified name of the experiment class -->
  <ExperimentClass>NeatBFS.Experiments.
      SingleStepShortestPathTaskExperiment</ExperimentClass>
  <!-- Experiment description -->
  <Description>
    Shortest path task experiment with turing controller.
  </Description>
  <!-- Comment about the specific experiment setup. NO LINE BREAKS!
       -->
  <Comment>10 vertices, 5 edges</Comment>
  <!-- How many experiments will be performed (serially) -->
  <ExperimentCount>3</ExperimentCount>
  <!-- The maximum number of generations that the experiment will
      run before terminating if the maximum fitness has not been
      achieved. -1 for unlimited -->
  <MaxGenerations>10000</MaxGenerations>
  <!-- NEAT -->
  <PopulationSize>500</PopulationSize>
  <!-- The generation interval between writing the status to the
      log -->
  <LogInterval>10</LogInterval>
  <!-- Wether or not to use parallel or serial evaluation. Parallel
       is faster, but serial can be necessary for debugging -->
  <MultiThreading>true</MultiThreading>
  <!-- Maximum number of threads if multithreading is enabled. -1
      defaults to CPU core count -->
  <MaxDegreeOfParallelism>-1</MaxDegreeOfParallelism>
  <ShortestPathTaskParams>
    <Iterations>25</Iterations>
    <Graph type="PathWithRandom" vertices="10" seed="978654321"
        minpath="5" symmetricinstances="true" />
  </ShortestPathTaskParams>
  <!-- Turing Machine Parameters-->
  <TuringMachineParams>
    <!-- If set to false, the Turing machine will always return a 0
         vector -->
    <Enabled>true</Enabled>
    <!-- Maximum memory size (-1 for unlimited) -->
    <N>-1</N>
    <!-- Write vector size -->
    <M>10</M>
    <!-- If M > 1 this will initialize the values of the turing
        machine with a gradient from 0 to 1. Default = false -->
    <InitalizeWithGradient>false</InitalizeWithGradient>
    <!-- The turing machine vectors can have an initial values
        instead of a gradient. Default = 0-->
    <InitalValue>0.1</InitalValue>
    <!-- Minimum similarity of the turing data vector to consider
        this location for a content jump -->
    <MinSimilarityToJump>0.5</MinSimilarityToJump>
    <!-- Number of Read/Write heads-->
    <Heads>1</Heads>
    <!-- Number of shift inputs -->
    <ShiftLength>3</ShiftLength>
    <!-- "Single" or "Multiple". Single has only a single input
        value, multiple has 3 for [-1, 0, 1] -->
    <ShiftMode>Multiple</ShiftMode>
    <!-- A extra memory location is maintained with the initial
        values at the end of the tape. If the machine write to this
         location, a new one is created.
                This can be used to give the tm an option to always
                    have a place to jump to create new memories.
                   Default = false -->
    <UseMemoryExpandLocation>true</UseMemoryExpandLocation>
    <!-- Similarity threshold to check if a write operation changed
         the values at a location Default = 0.9 -->
    <DidWriteThreshold>0.9</DidWriteThreshold>
    <!-- "Interpolate" or "Overwrite".
        Interpolate: Turing machine will interpolate between the
            write vector and the content already stored at the
            position.
        Overwrite: Turing machine will overwrite if output is >=
            0.5, otherwise it will not write. -->
    <WriteMode>Interpolate</WriteMode>
  </TuringMachineParams>
  <!-- Novelty Search Parameters has been omitted as it is disabled
      -->
  <!-- MultiObjective has been omitted as it is disabled -->
  <!-- The network activation scheme to use -->
  <Activation>
    <!-- "Acyclic", "CyclicFixedIters" or "CyclicRelax"
         Acyclic:           Does not support recurrent connections.
              Network is fully activated each activation.
         CyclicFixedIters:  Each iteration will iterate signals one
              step from input -> output.
         CyclicRelax:       Activate the network until all node
             output has remained unchanged between iterations
             within the threshold,
                            or until the maximum iteration count is
                                 reached. -->
    <Scheme>CyclicRelax</Scheme>
    <!-- CyclicFixedIters only -->
    <Iters>3</Iters>
    <!-- CyclicRelax only -->
    <Threshold>0.1</Threshold>
    <!-- CyclicRelax only -->
    <MaxIters>5</MaxIters>
  </Activation>
  <!-- Complexity Regulation Parameters -->
  <ComplexityRegulation>
    <!-- "Absolute" or "Relative"
       Absolute: Defines an absolute ceiling on complexity.
       Relative: Defines a relative ceiling on complexity. E.g.
           relative to the complexity at the end of the most recent
```

```
            simplification phase. -->
    <ComplexityRegulationStrategy>Relative</
        ComplexityRegulationStrategy>
    <!-- The complexity ceiling. When complexity reaches this
        threshold, the algorithm will switch from complexifying to
        simplification -->
    <ComplexityThreshold>25</ComplexityThreshold>
  </ComplexityRegulation>
  <!-- Evolution Algorithm Parameters -->
  <EAParams>
    <!-- Number of species in the population. Default: 10 -->
    <SpecieCount>30</SpecieCount>
    <!-- We sort specie genomes by fitness and keep the top N%, the
         other genomes are removed to make way for the offspring.
        Default: 0.2  -->
    <ElitismProportion>0.2</ElitismProportion>
    <!-- We sort specie genomes by fitness and select parent
        genomes for producing offspring from the top N%.
         Selection is performed prior to elitism being applied,
             therefore selecting from more genomes than will be
             made elite is possible. Default: 0.2 -->
    <SelectionProportion>0.2</SelectionProportion>
    <!-- The proportion of offspring to be produced from asexual
        reproduction (mutation). Default: 0.5 -->
    <OffspringAsexualProportion>0.5</OffspringAsexualProportion>
    <!-- The proportion of offspring to be produced from sexual
        reproduction. Default: 0.5 -->
    <OffspringSexualProportion>0.5</OffspringSexualProportion>
    <!-- The proportion of sexual reproductions that will use
        genomes from different species. Default: 0.01 -->
    <InterspeciesMatingProportion>0.01</
        InterspeciesMatingProportion>
  </EAParams>
  <!-- Genome Parameters -->
  <GenomeParams>
    <!-- The connection weight range to use in NEAT genomes. E.g. a
         value of 5 defines a weight range of -5 to 5.
         The weight range is strictly enforced - e.g. when creating
              new connections and mutating existing ones. Default:
             5.0 -->
    <ConnectionWeightRange>5.0</ConnectionWeightRange>
    <!-- A proportion that specifies the number of interconnections
         to make between input and output neurons in an initial
        random population.
         This is a proportion of the total number of possible
             interconnections. Default: 0.05 -->
    <InitialInterconnectionsProportion>0.05</
        InitialInterconnectionsProportion>
    <!-- The probability that all excess and disjoint genes are
        copied into an offspring genome during sexual reproduction.
         Currently the execss/disjoint genes are copied in an all
             or nothing strategy. Default: 0.1 -->
    <DisjointExcessGenesRecombinedProbability>0.1</
        DisjointExcessGenesRecombinedProbability>
    <!-- The probability that a genome mutation operates on genome
        connection weights. Default: 0.988 -->
    <ConnectionWeightMutationProbability>0.988</
        ConnectionWeightMutationProbability>
    <!-- The probability that a genome mutation is an 'add node'
        mutation. Default: 0.001 -->
    <AddNodeMutationProbability>0.001</AddNodeMutationProbability>
    <!-- The probability that a genome mutation is an 'add
        connection' mutation. Default: 0.01 -->
    <AddConnectionMutationProbability>0.01</
        AddConnectionMutationProbability>
    <!-- The probability that a genome mutation is a 'delete
        connection' mutation. Default: 0.001 -->
    <DeleteConnectionMutationProbability>0.001</
        DeleteConnectionMutationProbability>
  </GenomeParams>
</Config>
```

A.2 Configuration without memory

Changes:

```
  <TuringMachineParams>
    <!-- If set to false, the Turing machine will always return a 0
        vector -->
    <Enabled>false</Enabled>
```

A.3 Configurations with acyclic networks

Changes:

```
  <Activation>
    <!-- "Acyclic", "CyclicFixedIters" or "CyclicRelax"
          Acyclic:           Does not support recurrent connections.
              Network is fully activated each activation.
          CyclicFixedIters:  Each iteration will iterate signals one
              step from input -> output.
          CyclicRelax:       Activate the network until all node
            output has remained unchanged between iterations
            within the threshold,
                             or until the maximum iteration count is
                                 reached. -->
    <Scheme>Acyclic</Scheme>
```