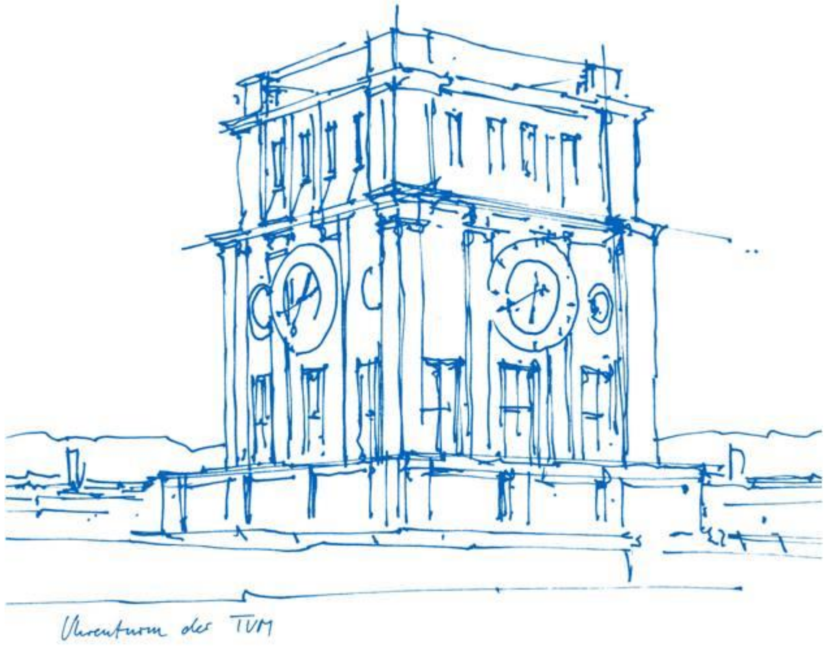
Deep Learning for Satisfiability Problems

Linus Kreitner

Technical University of Munich

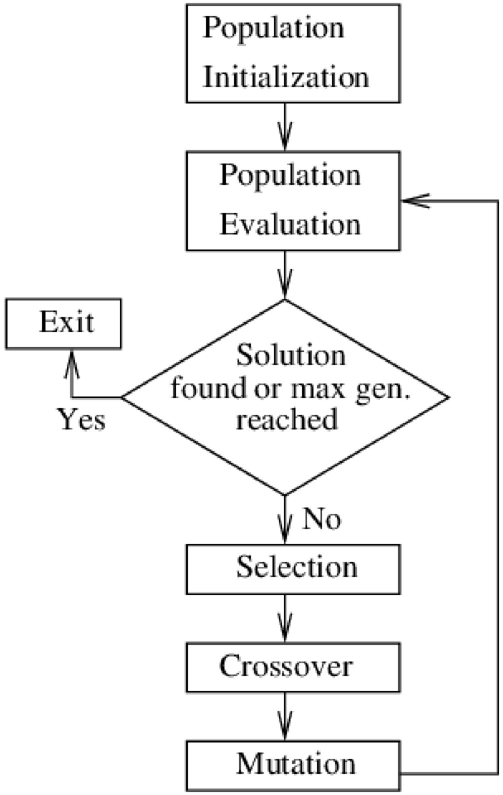
Department of Informatics

15. April 2020



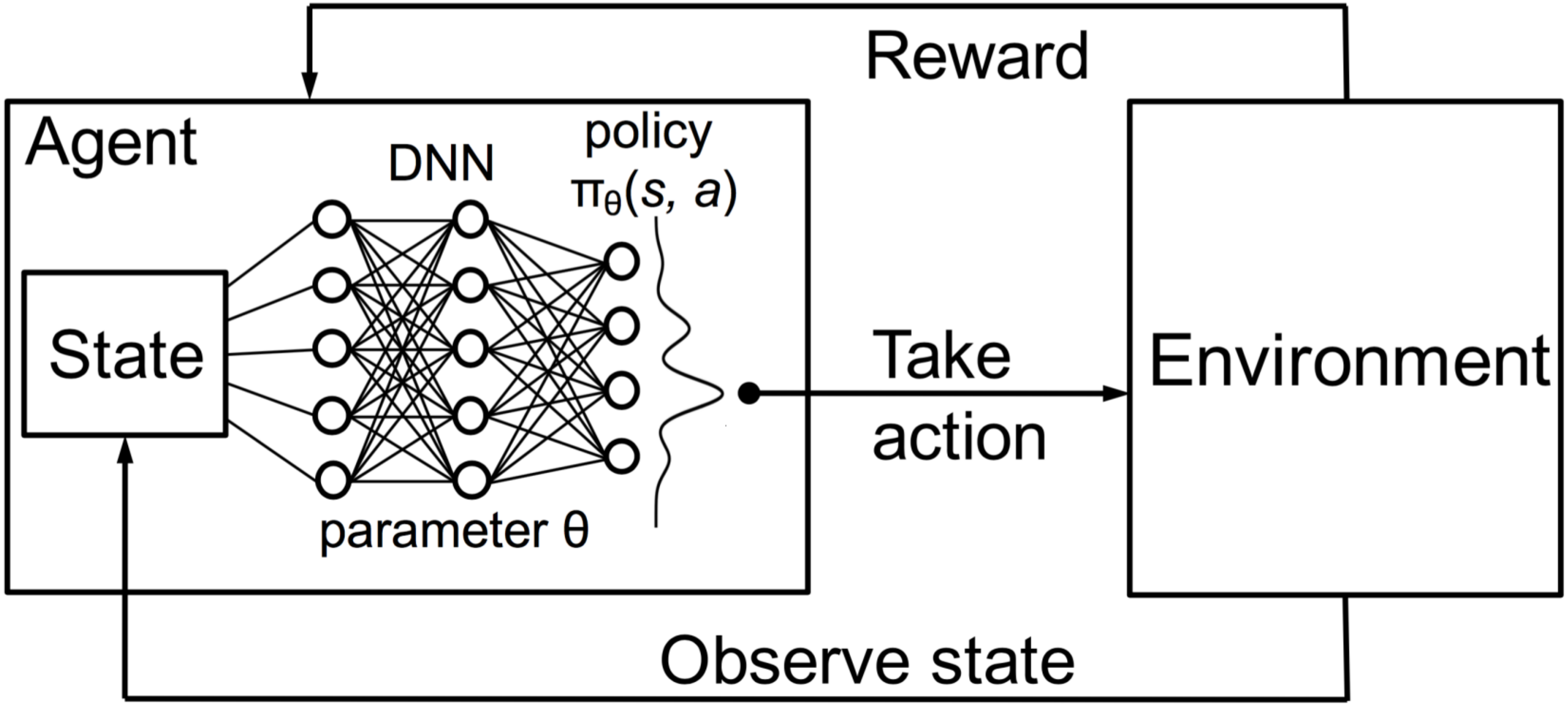
Chapter 0: Context

Def: Evolutionary Algorithm

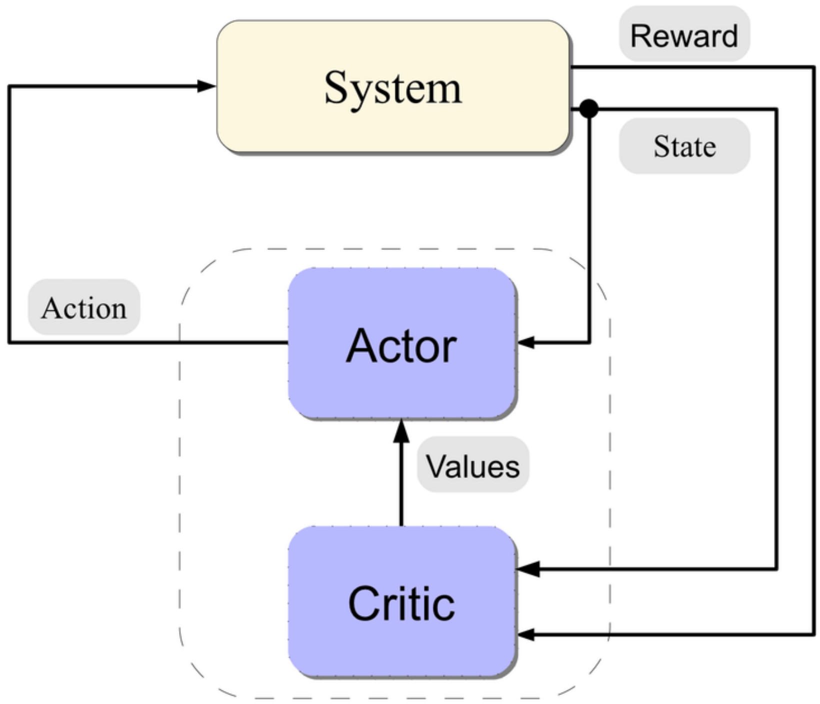


How to choose evolution parameters?

Def: Reinforcement Learning



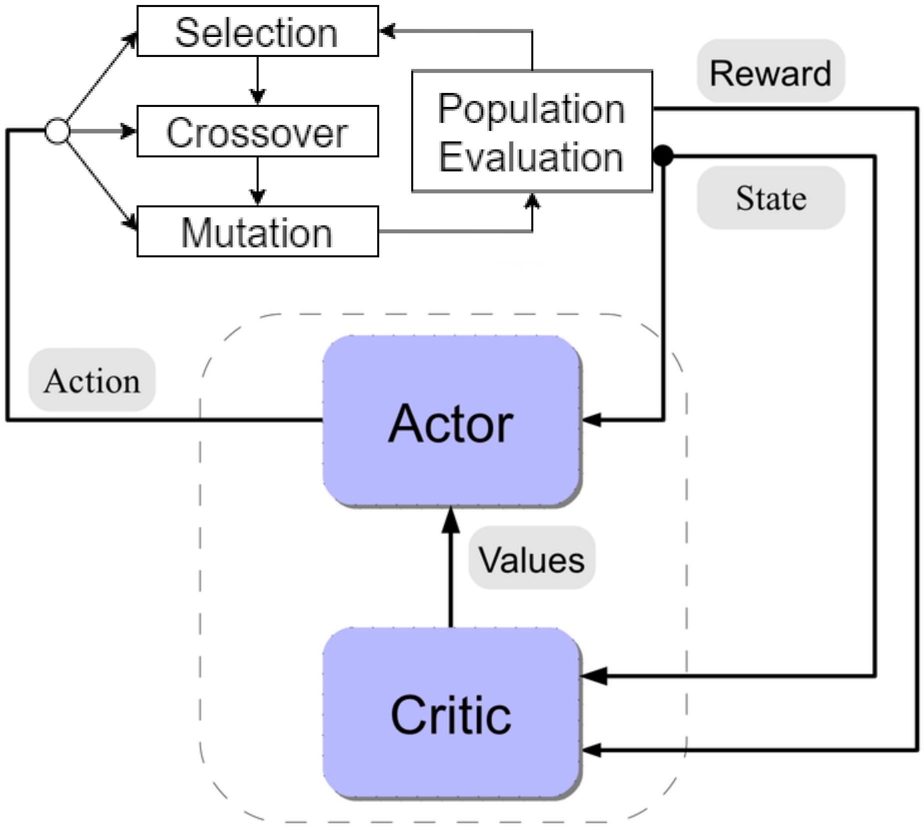
Def: Actor Critic Model



Actor: Proposes actions

Critic: Evaluates current state

Def: Learning to Evolve



Learn to dynamically adjust the evolution strategy

Chapter 1: The Task Description

Use the concept of “Learning to Evolve” on the boolean satisfiability problem and surpass state-of-the-art methods.

Roadmap:

1. Analyse state-of-the-art SAT solving algorithms
2. Create fine tuned network architecture to extract helpful features from SAT
3. Apply Deep Reinforcement Learning + EA to enhance state-of-the-art solvers

Preliminary Work:

- Learning to Evolve (Jan's Code base)
- SAT Problem Code base (Yoav Schneider's Codebase)
  - 3-SAT Problem
  - Basic solvers
- Years of research on SAT[1]

1: See references at the end

Chapter 2: The SAT Problem

Input: CNF Function $\mathcal{F} = \bigwedge_i \bigvee_j (\neg) x_{ij}$

Output: Assignment $a \in \{T, F\}^G$ with $\mathcal{F}(a) = True$

Example: $\mathcal{F} = (x_1 + \overline{x_2})(x_2 + x_3)(\overline{x_1} + \overline{x_3}) \mapsto a = (T, T, F)$

Application areas:

- Software verification
- Constraint solving in AI
- (https://doi.org/10.1109%2FJPROC.2015.2455034)
- …

Requirements for SAT solvers :

- Can handle thousands / millions of variables / clauses
- Can handle structured and random problems
- Optimized for Execution time

Chapter 3: SAT solving approaches

1. Message Passing
   Infer correct assignment by theoretical reasoning over many steps

2. Local Search
   Iteratively improve current solution

3. Backtracking (Single / Parallel)
   Iteratively fixate variables and backtrack if conflict

Message Passing:

Basic Idea:

1. Consider SAT Problem as a Graph Network (GN) (Encoder)

   $G = (V, E)$

   $x_i^{(t-1)} \in \mathbb{R}^{D_n} \triangleq$ node features of node *i* at timestep *(t-1)*

   $e_{j,i} \in \mathbb{R}^{D_e} \triangleq$ edge features from node *j* to node *i* ($\leftrightarrow j \in \mathcal{N}(i)$)

2. Apply message passing between Nodes (Processor)

   $x_i^{(t)} = \gamma^{(k)}(x_i^{(k-1)}, \alpha_{j \in \mathcal{N}(i)}(\phi^{(k)}(x_i^{(k-1)}, e_{j,i}))),$

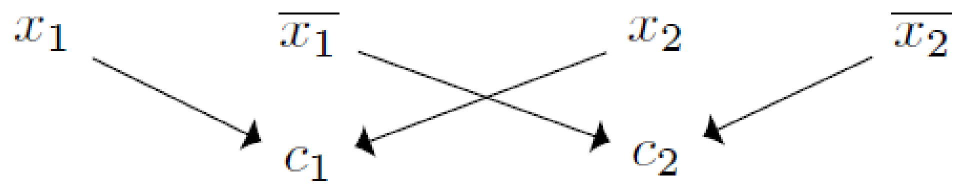
   $\alpha \triangleq$ aggregate function (e.g. sum)

   Update function γ, Message function $\phi \triangleq$ differential functions (e.g. MLP)

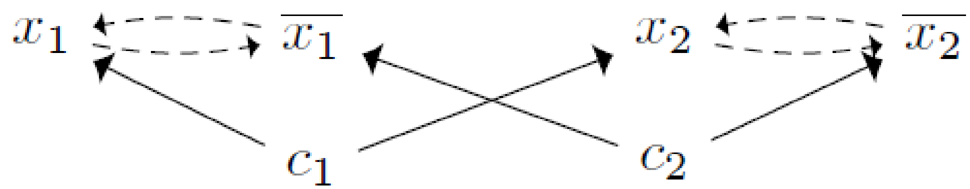
3. Use Readout function to infer required information (Decoder)

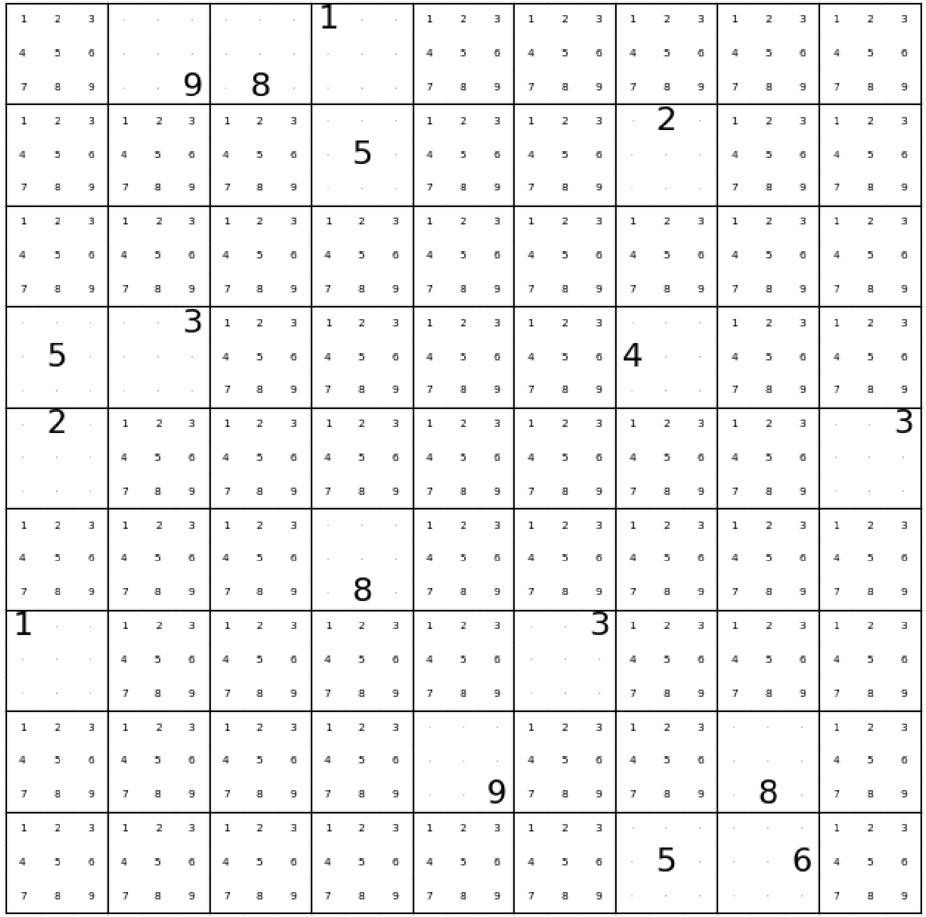
Message Passing for SAT:

1. Each clause receives message from its literals



2. Each literal receives messages from its clauses and its complement





Example: Message passing applied to Sudoku

Flaws of Message Passing

- Performance not compatible with state of the art
- Embedding size is crucial and does not scale well
- Learning an embedding requires a lot of training
- Relies on Read-out Phase

⇒ Not suited as E2E solver!

Local Search:

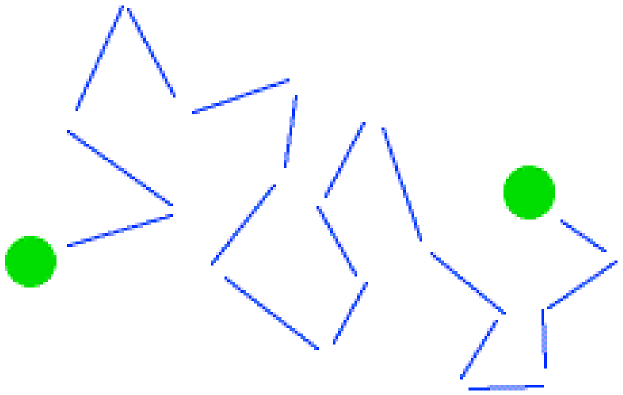
Algorithm1: Local Search:

$\boldsymbol{Input}: a \in \{False, True\}^G, CNF\ formula\ \mathcal{F}$

$\quad \boldsymbol{while}\ \mathcal{F}(a) \neq True:$

$\quad\quad \text{index} \leftarrow ChooseVariable(r))$

$\quad\quad a[index] \leftarrow not\ a[index]$



Flaws of Local Search

- Do not guarantee solution
- Prone to getting stuck in local optimum
- No learning

None of the state-of-the-art SAT solvers uses the Local Search approach!

Backtracking

Algorithm2: Backtracking

$\boldsymbol{Input}: a \in \{Unknown\}^G, CNF\ formula\ \mathcal{F}$

$\quad \boldsymbol{while}\ \mathcal{F}(a) \neq True:$

$\quad\quad \boldsymbol{if}\ \mathcal{F}(a)\ yields\ CONFLICT:$

$\quad\quad\quad \mathrm{a} \leftarrow Backtrack()$

$\quad\quad a \leftarrow AssignVariable(a)$

Assign Variables

1. Make assignment arc-consistent[1]:

$\underline{\boldsymbol{Algorithm3}: \boldsymbol{BCP}:}$

$$\textbf{While } \exists_{c \in Clauses}: c = (False, \ldots, False, x):$$

$$x \leftarrow True$$

$$\boldsymbol{if}\ \mathcal{F}(a)\ yields\ CONFLICT:$$

$$\mathrm{a} \leftarrow Backtrack()$$

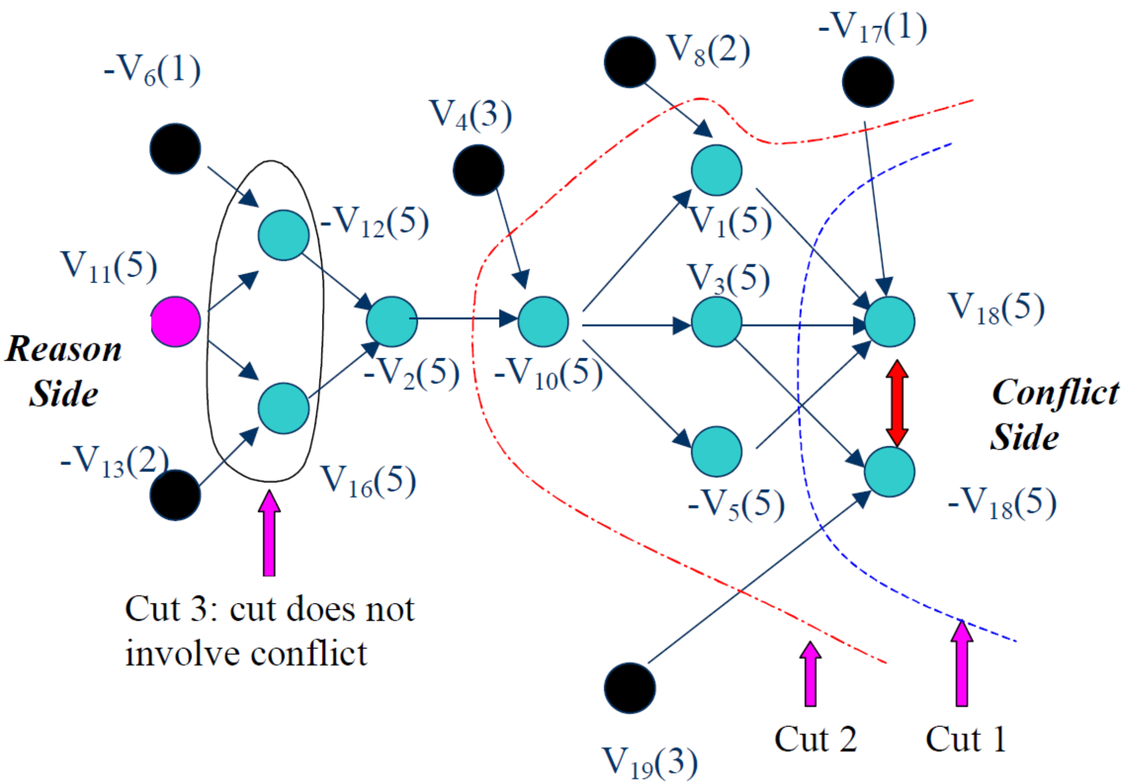
2. Make a decision:
   - Choose a variable (e.g. VSIDS)
   - Choose a polarity (e.g. Progress saving)

1: also called Unit-Propagation or Boolean Constraint Propagation (BCP)

Backtrack

1. Analyze conflict
2. Learn conflict clauses & add to clause database
3. Jump back (non-) chronological or restart

Conflict Analysis using Implication Graph



Learned Clauses:

$Cut\ 1\colon\ (\bar{V}_1 + \bar{V}_3 + V_5 + V_{17} + \bar{V}_{19})$

$Cut\ 2\colon\ (V_2 + \bar{V}_4 + \bar{V}_8 + V_{17} + \bar{V}_{19})$

$Cut\ 3\colon\ (\bar{V}_2 + V_4 + \bar{V}_{11} + V_{13})$

Look-ahead techniques:

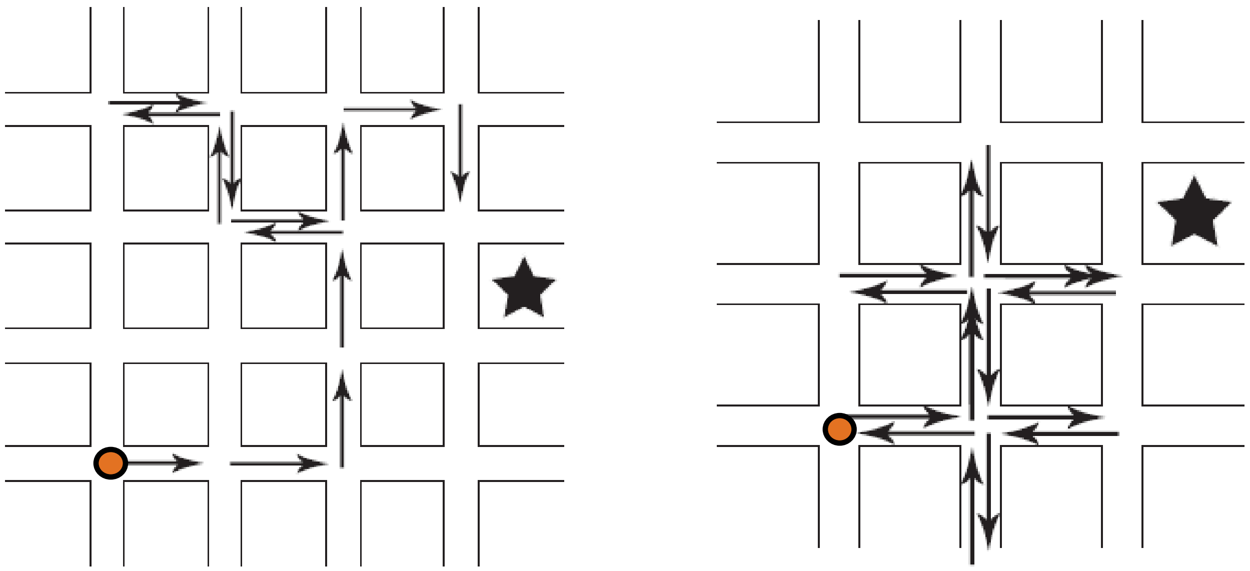


Figure 5.1. A NYC walk, conflict-driven (left) and look-ahead (right). ⋆ denotes the target.

Assign a variable and check the result instead of only relying on heuristics

Taken from DOI: 10.3233/978-1-58603-929-5-155

Flaws of Backtracking

- Necessity to decide for 1 subtree
- Decision at beginning are crucial
- Policies of variable selection, learning and restart are handcrafted
- Same policies for all problems

Parallel Backtracking

1. Cube & Conquer
   - Split problem in many sub problems
   - Balance workload on multiple instances
   - Share knowledge between individuals
2. Portfolio-based
   - Run multiple different solvers on the same problem
   - Share knowledge between solvers

→ One solver is rarely efficient for all problem types

Challenges of Parallel Solving

- Balance workload
- Sharing of knowledge
- Efficient implementation

Flaws and Strengths of current solvers

	Message Passing		Backtracking (single)			Backtracking (Parallel)	
Strengths	Good usage of individual topology	Network learns to extract deep insights	Performance	Ability to learn new clauses	Certainty of getting closer to solution	Ability to divide workload	Generate more knowledge per depth level
Flaws	Performance	Read-out function	Fixed parameters for every problem	Necessity of deciding for one subtree	Poor decisions at beginning	Balance workload requires deep insights	Complex Implementation

Combine approaches to keep strengths and bypass weaknesses

Use Reinforcement Learning to learn to guide certain heuristics in critical situations

Chapter 4: A new approach

Idea:

- Use state-of-the-art solver as “backbone”
- Guide / Replace some heuristics with NN

Implementation:

- Use parallel backtracking
- Each parallel path is an individual
- Manage population of parallel paths

https://twitter.com/DS_Stiftung/status/1171790961153904640/photo/1

Possible Network outputs:

- Branch for selected variable
- Create *b* new individuals[1] for $\{v \mid v \in Variables\}^b$
- Remove individuals $\{a \in p\}^N$ from population[2]
- Only work on specific individuals $\{a \in p\}^N$
- Restart
- Delete learned clauses[3] $\{c_1, \dots, c_N, c_{1:N} \in C\}$
- Decay of VSIDS values

1: b is the branching factor
2: $N < P$ since there must be at least one individual
3: This is considered to be the biggest weakness of the popular MiniSAT algorithm

Network outputs:

- ~~Branch for selected variable~~
- Create *b* new individuals[1] for $\{v \mid v \in Variables\}^b$

Checking both the *T* and the *F* tree of the same variable almost never results in equal workload

Possible future outputs:

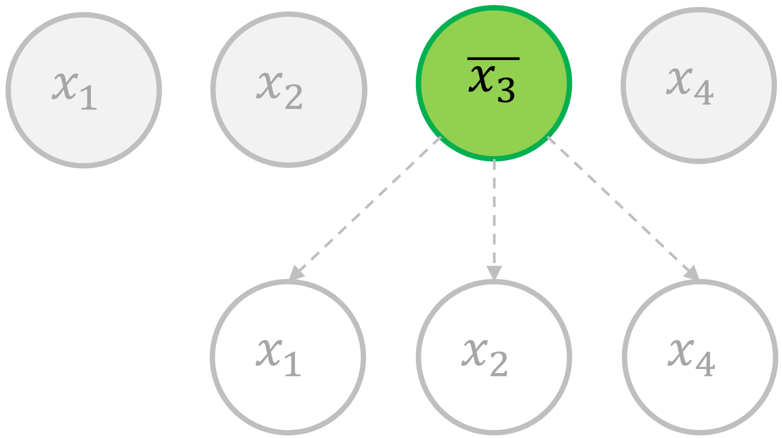
- Remove individuals $\{a \in p\}^N$ from population[2]
- Only work on specific individuals $\{a \in p\}^N$
- Restart now (Yes/No)
- Delete learned clauses $\{c_1, \ldots, c_N, c_{1:N} \in C\}$
- Decay of VSIDS values

This requires a form of fitness evaluation which is not trivial to compute!

1: b is the branching factor
2: $N < P$ since there must be at least one individual
3: This is considered to be the biggest weakness of the popular MiniSAT algorithm

New Approach:

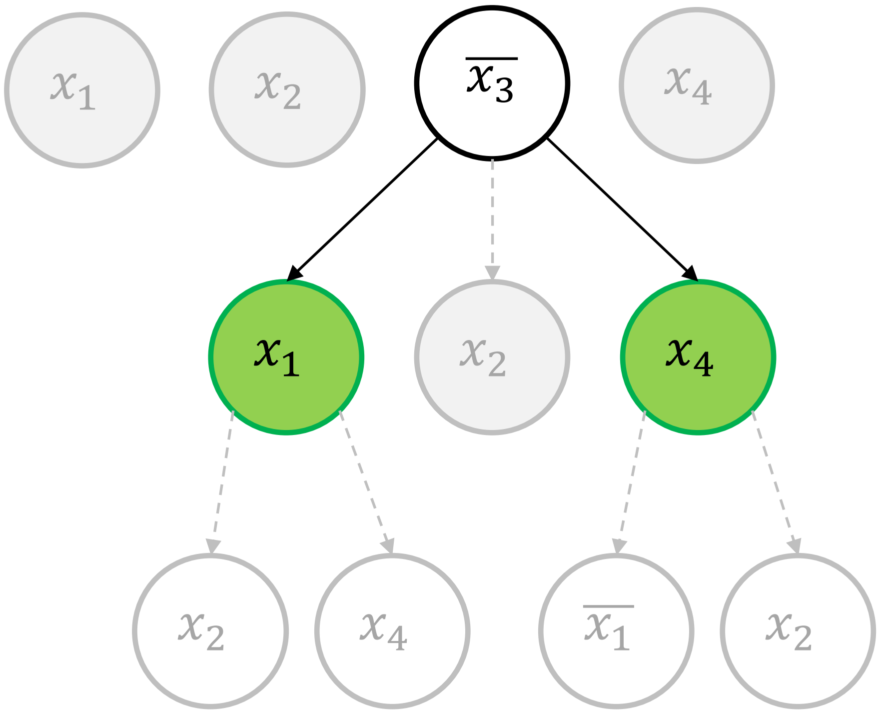
$\mathcal{F} = (x_1 + x_3)(\overline{x_2} + x4)(\overline{x_3} + x_2)$



$P = \{(U, U, F, U)\}$

New Approach:

$$\mathcal{F} = (x_1 + x_3)(\overline{x_2} + x4)(\overline{x_3} + x_2)$$

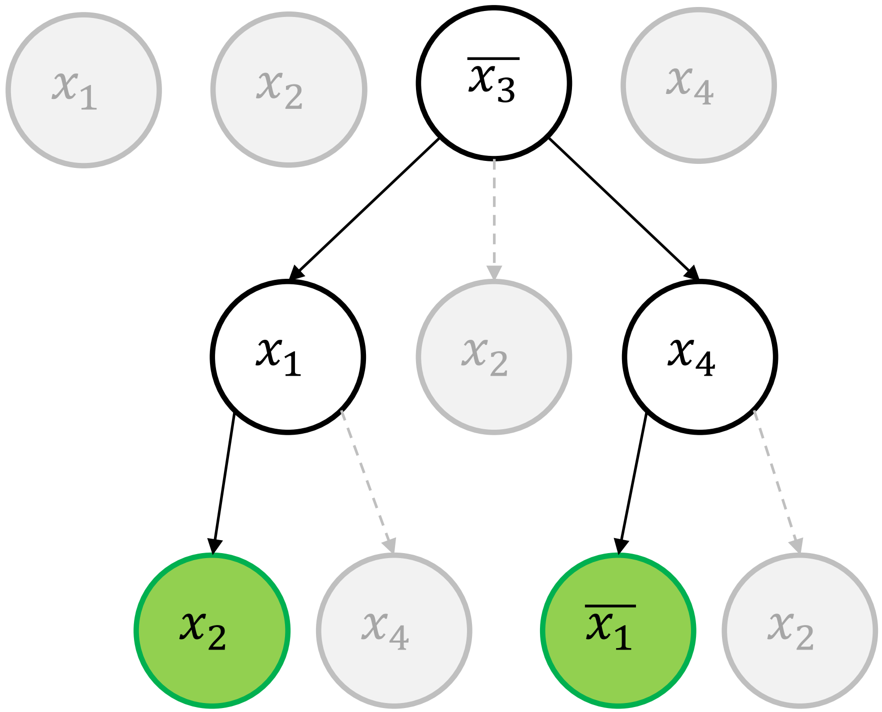


NN recommends branching

$$P = \{(T, U, F, U), (U, U, F, T)\}$$

New Approach:

$\mathcal{F} = (x_1 + x_3)(\overline{x_2} + x4)(\overline{x_3} + x_2)$



NN recommends no branching

$P = \{(T, T, F, U), (F, U, F, T)\}$

New Solving Algorithm

Algorithm4:

$\boldsymbol{Input}: p \in \left\{\{Unknown\}^G\right\}, CNF\ formula\ \mathcal{F}$

$\quad \boldsymbol{while}\ \forall_{a \in p}: \mathcal{F}(a) \neq True:$

$\quad\quad \boldsymbol{for\ all}\ a \in p:$

$\quad\quad\quad \boldsymbol{if}\ \mathcal{F}(a)\ yields\ CONFLICT:$

$\quad\quad\quad\quad \mathrm{a} \leftarrow Backtrack()$

$\quad\quad\quad \mathrm{q} \leftarrow MPN(state)$

$\quad\quad\quad \mathrm{p} \leftarrow Branch(p, q)$

$\quad\quad \boldsymbol{for\ all}\ a \in p:$

$\quad\quad\quad a \leftarrow BCP()$

Message passing network MPN generates action probabilities for all individuals to decide how to branch

Remove old Individuum a from population p and add $b \in \mathbb{N}$ new individuals

Algorithm5: Branch

$\boldsymbol{Input}: p \in \left\{\{U,T,F\}^G\right\}^P, a \in p, q \in \{(var, value)\}^{2G}$

$p \leftarrow p - a$

$\boldsymbol{for\ all}\ (var, value)\ in\ q:$

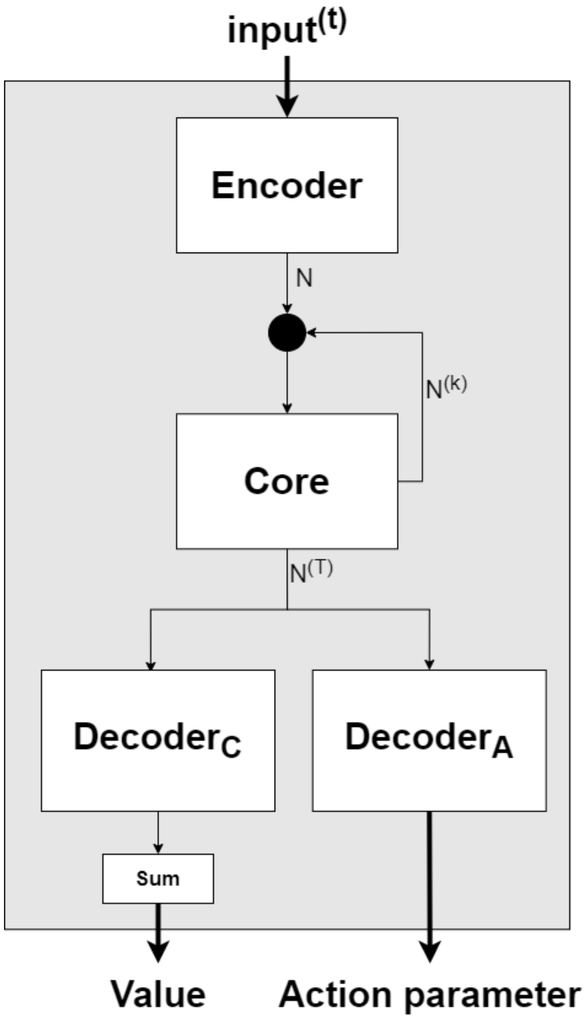
$\boldsymbol{if}\ value > 0.5:$

$a_{tmp} \leftarrow a$

$a_{tmp}[var] \leftarrow value$

$p \leftarrow p \cup a_{tmp}$

Chapter 5: Neural Network Architecture



Message Passing Network MPN:

- Uses message passing to extract deep features about current state
- Calculates next actions

➡ Network learns how to branch

$input^{(t)} \in \mathbb{R}^{PxExG}$

$0 < b < 1$

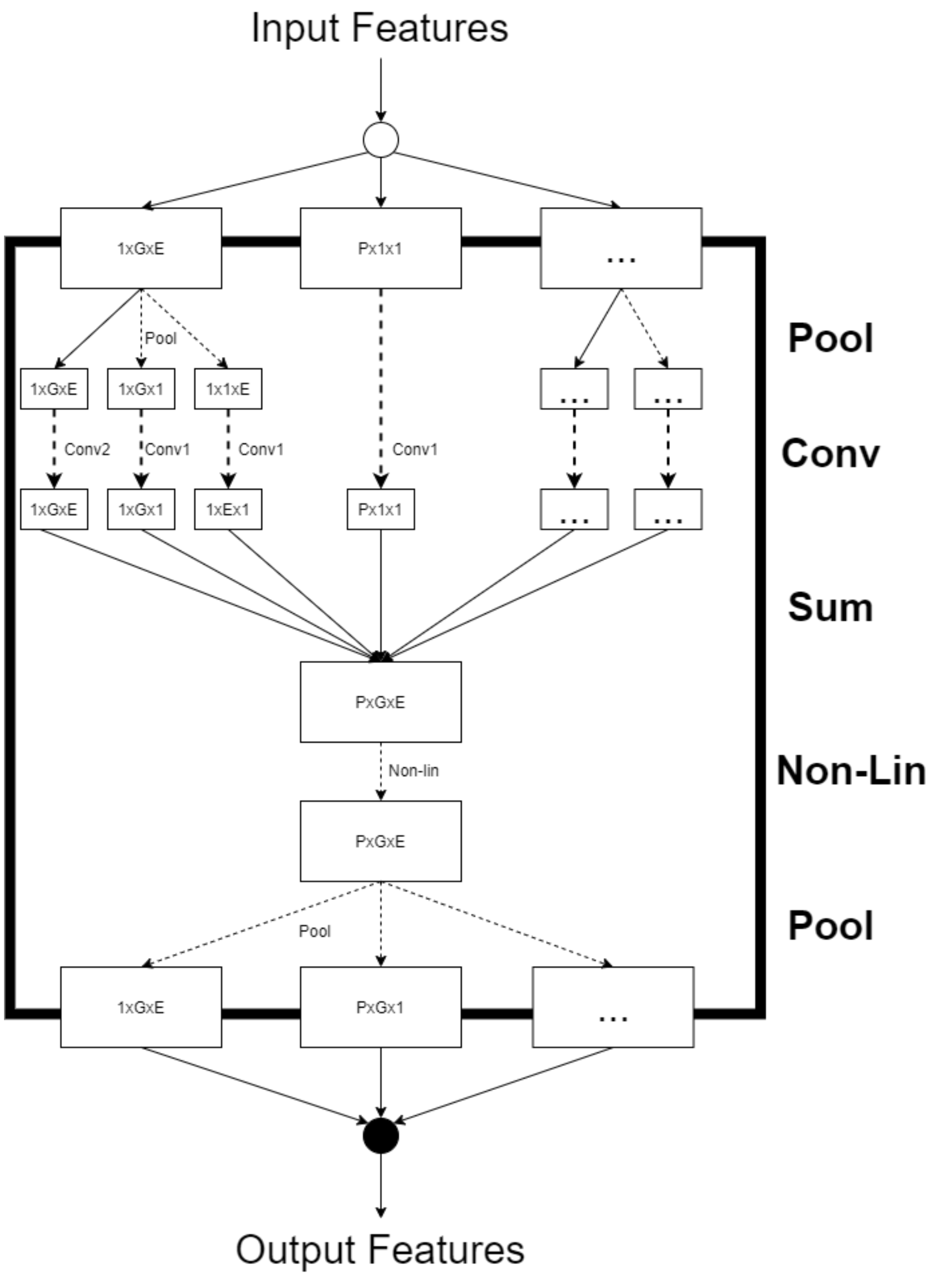
$value \in \mathbb{R}$

$action \in \mathbb{R}^{PxG}$

$\mathrm{N} \in \mathbb{R}^{PxExG}$

$Encoder, Decoder$: $\widehat{\boldsymbol{C}}$ $Layer$

$Core$: $\boldsymbol{MPC}$ $Layer$



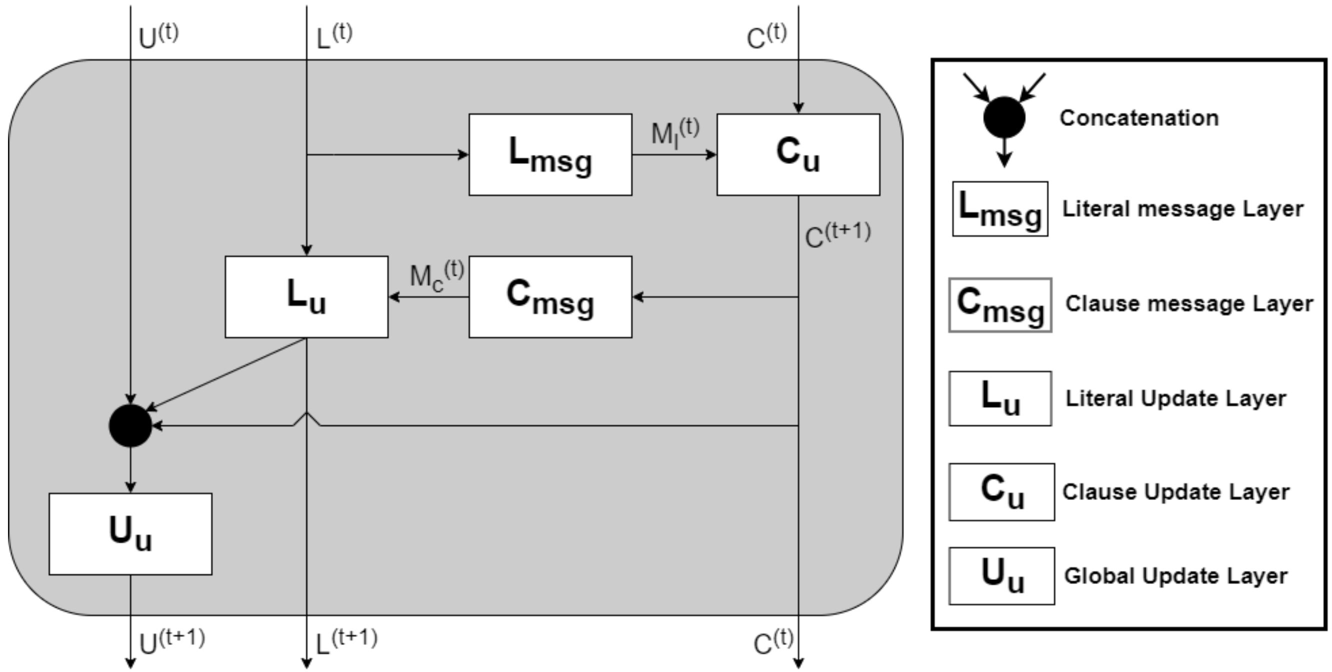
Convolution Layer $\widehat{C}^1$:

Hardwires network properties:

- Permutation Invariance[2] for all dimensions
- Memory-efficient[3] convolution

1: Note that both pooling layers are optional
2: Achieved by setting kernel-size=1 for convolution
3: No broadcasting before convolution

Message Passing Core *MPC:*



$$M_l \leftarrow L_{msg}\left(L^{(t)}\right)$$
$$C^{(t+1)} \leftarrow C_u\left(C^{(t)}, M_l A^T\right)$$
$$M_c \leftarrow C_{msg}\left(C^{(t)}\right)$$
$$L^{(t+1)} \leftarrow L_u(L^{(t)}, M_c A, Flip(L^{(t)}))$$
$$U^{(t+1)} \leftarrow U_u(U^{(t)}, L^{(t+1)}, C^{(t+1)})$$

$adjacency\ Matrix\ A \in \mathbb{R}^{Cx2V}$

$Flip(L) := swapping\ each\ row\ l\ with\ \bar{l},$

$\mathrm{L}, M_l \in \mathbb{R}^{D_l x 2V}$

$C, M_c \in \mathbb{R}^{D_c x C}$

$U \in \mathbb{R}^{D_u}$

$V := Number\ of\ variables$

$C := Number\ of\ clauses$

$D := Embedding\ size$

Possible Extension:

Problem:

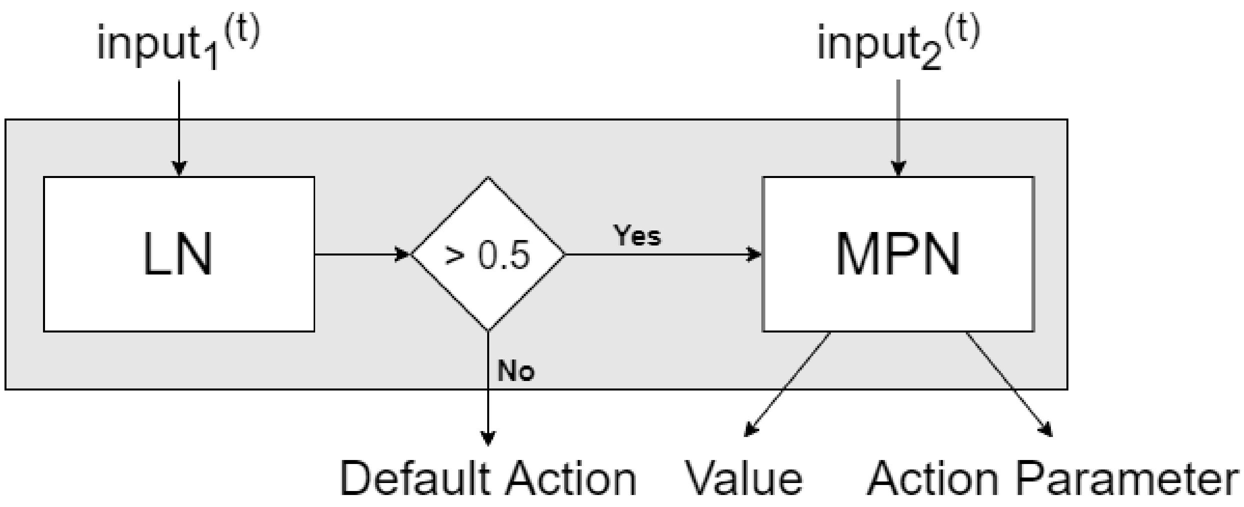
Message passing adds a lot of runtime

Idea:

Lightweight network LN perform shallow analysis to determine whether to use MPN.

Complex network MPN performs deep analysis to determine how to branch.

➡ Network learns when to use costly network MPN



LN: $\widehat{\boldsymbol{C}}$ *Layer*
with sigmoid activation

Possible Extension:

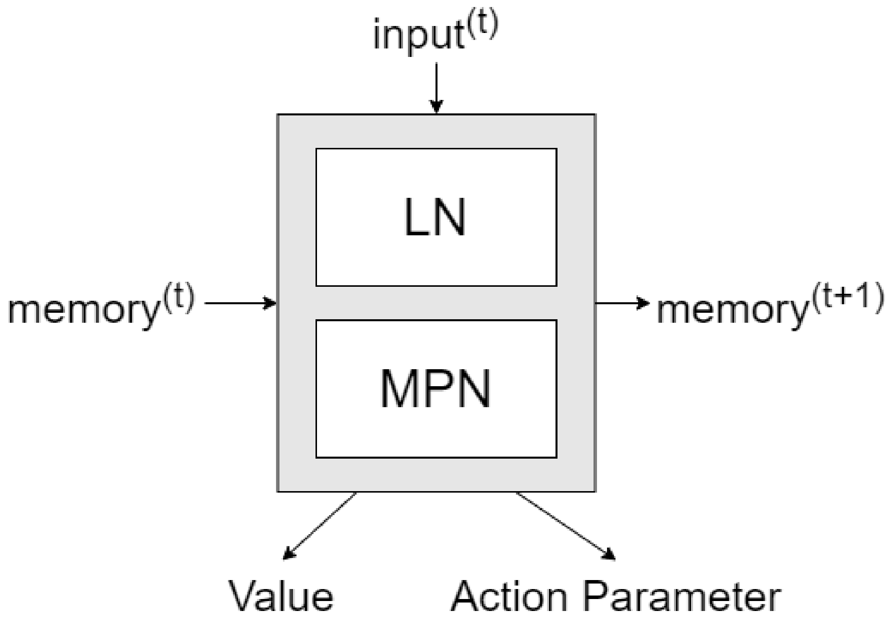
Problem:

Network can only use current state, i.e. no long-term planning

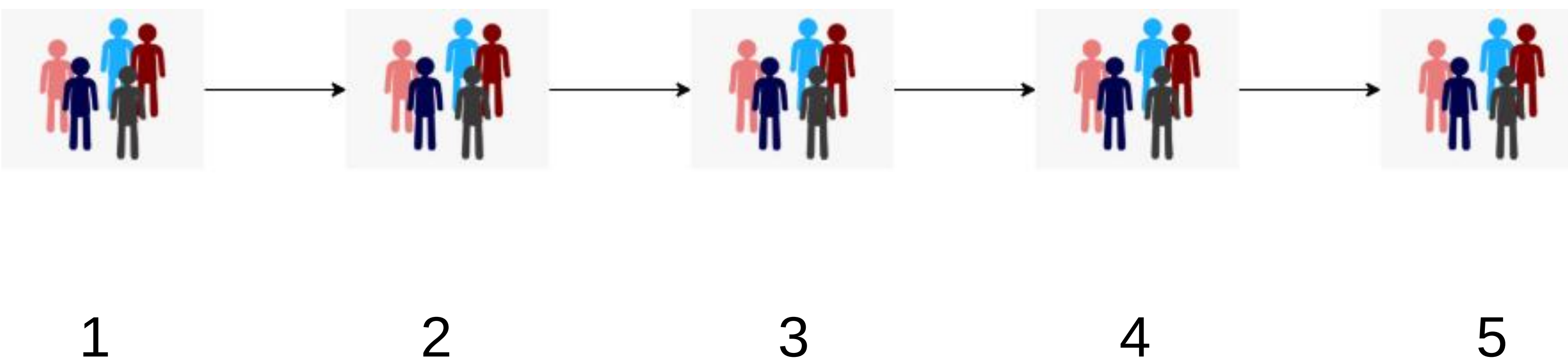
Idea:

Make whole network recurrent

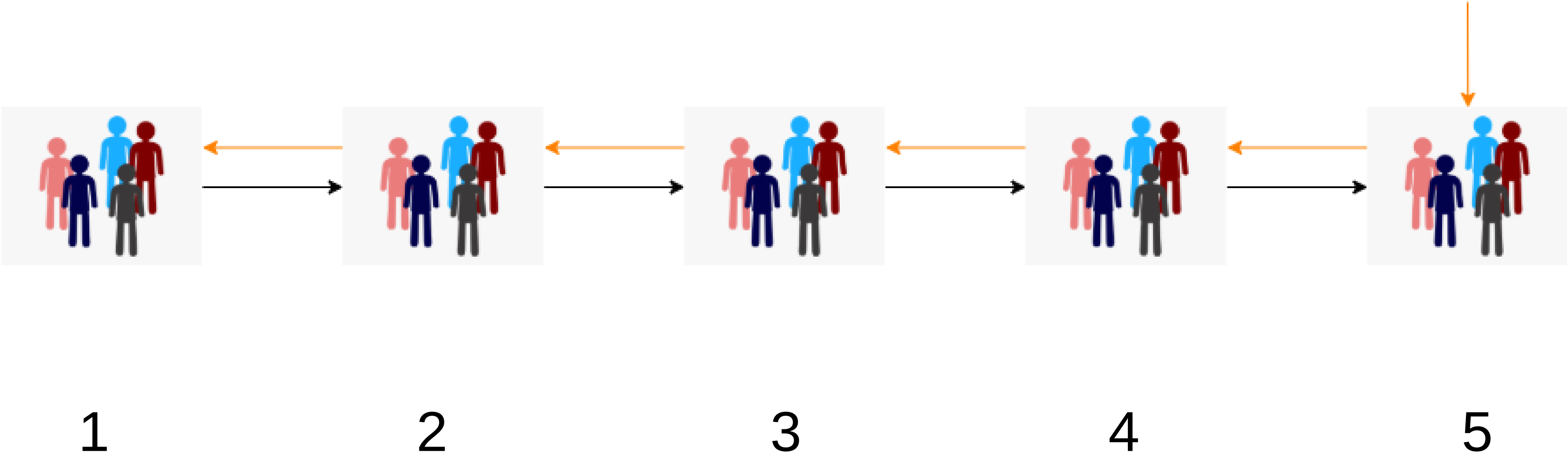
→ Can plan over multiple generations



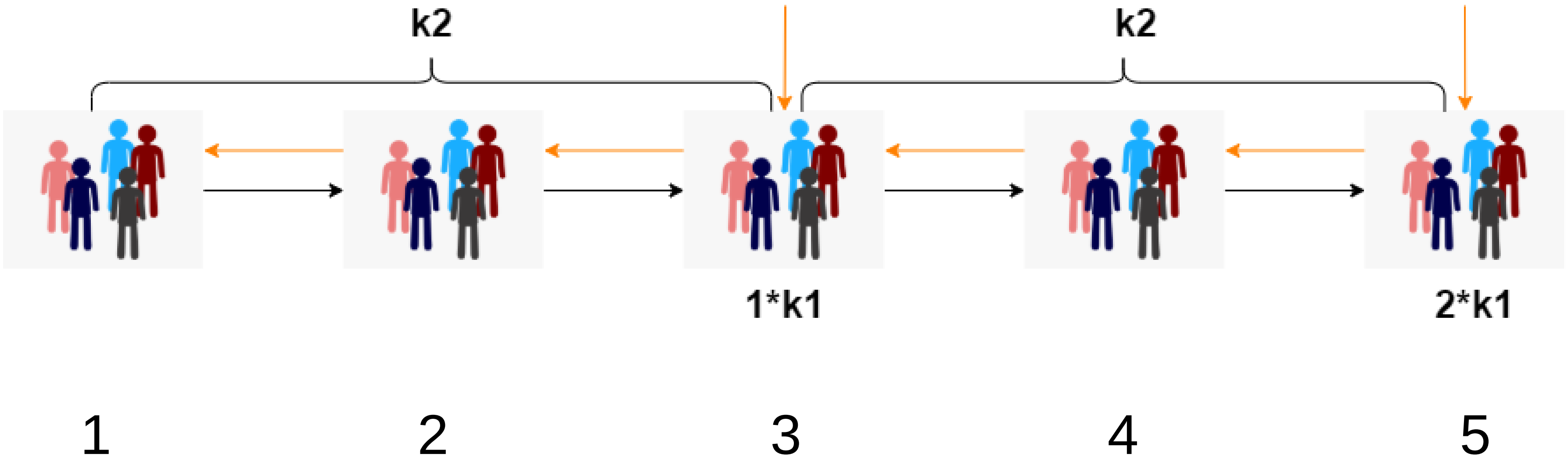
<u>Forward propagation</u>



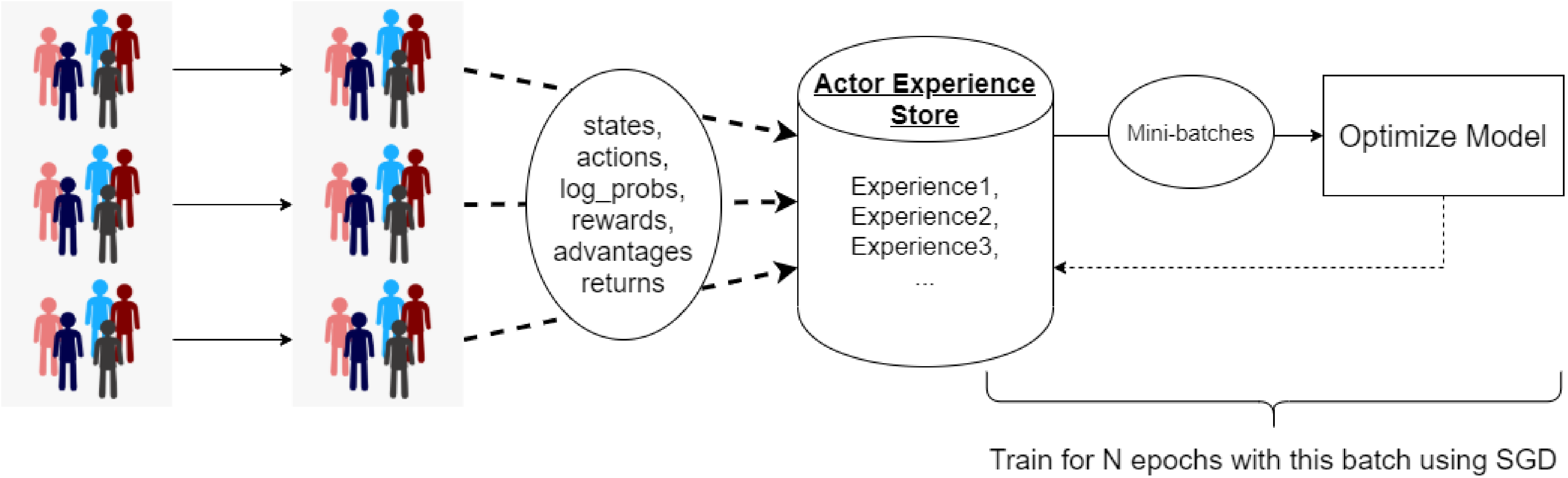
Backward propagation through time



Truncated Backward propagation through time

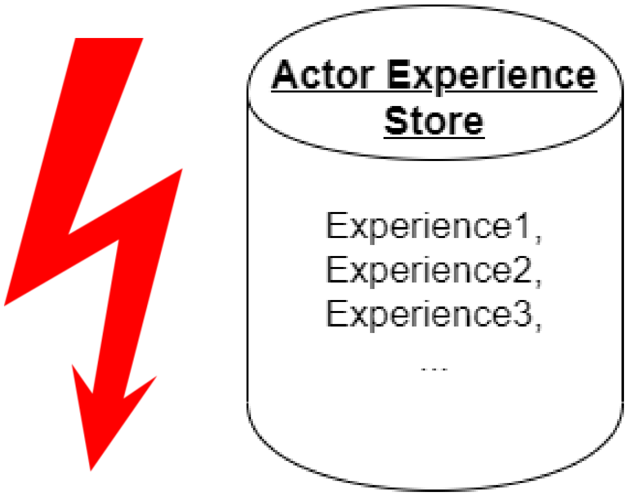


Proximal Policy Optimization (PPO)



- Prevent destructive large policy updates
- Simple Implementation (e.g. better than TRPO)
- Sample Efficiency

PPO with RNN



- Actor experience store stores full computation graph for backpropagation
- Enormous memory usage

torch.utils.checkpoint:

- Does not save intermediate activations
  - ➡ Less memory usage
- Re-computation in backward pass
  - ➡ Longer computation time (approx. +20%)

Input Features LN

• VSIDS values per Individual	Px1xG
• Depth per Individual	Px1x1
• (Optional) Memory	?

Input Features MPN

• State representation	PxExG
• VSIDS values per Individual	Px1xG
• (Optional) Memory	?

(Optional) Memory Features

• Individual properties	Px1x1
• Variable Embedding	1x1xG
• Clause Embedding	1xEx1
• …	

Other interesting features

• Original Problem instance	1xExG
• Genome per individual	Px1xG
• τ-Satisfied clauses	Px1xG
• Break, Make Values	Px1xG
• Fitness per Individual	Px1x1
• Participation per variable in clauses	1x1xE
• #Clauses, #Variables, #Generations left	1x1x1
• …	

Other interesting features

Problem Size Features:
1. **Number of clauses:** denoted c
2. **Number of variables:** denoted v
3. **Ratio:** c/v

Variable-Clause Graph Features:
4-8. **Variable nodes degree statistics:** mean, variation coefficient, min, max and entropy.
9-13. **Clause nodes degree statistics:** mean, variation coefficient, min, max and entropy.

Variable Graph Features:
14-17. **Nodes degree statistics:** mean, variation coefficient, min and max.

Balance Features:
18-20. **Ratio of positive and negative literals in each clause:** mean, variation coefficient and entropy.
21-25. **Ratio of positive and negative occurrences of each variable:** mean, variation coefficient, min, max and entropy.
26-27. **Fraction of binary and ternary clauses**

Proximity to Horn Formula:
28. **Fraction of Horn clauses**
29-33. **Number of occurrences in a Horn clause for each variable:** mean, variation coefficient, min, max and entropy.

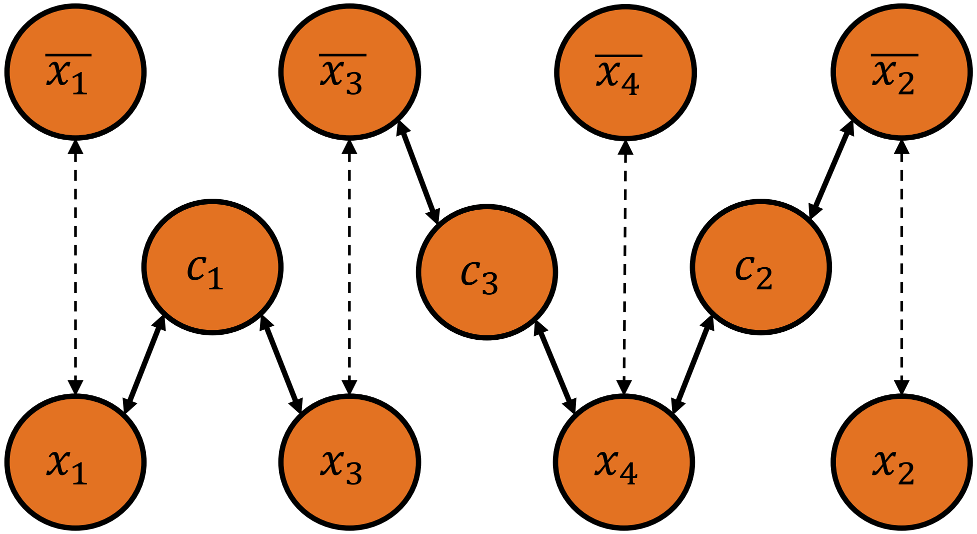
DPLL Probing Features:
34-38. **Number of unit propagations:** computed at depths 1, 4, 16, 64 and 256.
39-40. **Search space size estimate:** mean depth to contradiction, estimate of the log of number of nodes.

Local Search Probing Features:
41-44. **Number of steps to the best local minimum in a run:** mean, median, 10th and 90th percentiles for SAPS.
45. **Average improvement to best in a run:** mean improvement per step to best solution for SAPS.
46-47. **Fraction of improvement due to first local minimum:** mean for SAPS and GSAT.
48. **Coefficient of variation of the number of unsatisfied clauses in each local minimum:** mean over all runs for SAPS.

Figure 2: *The features used for building SATzilla07; these were originally introduced and described in detail by Nudelman et al. (2004a).*

State representation

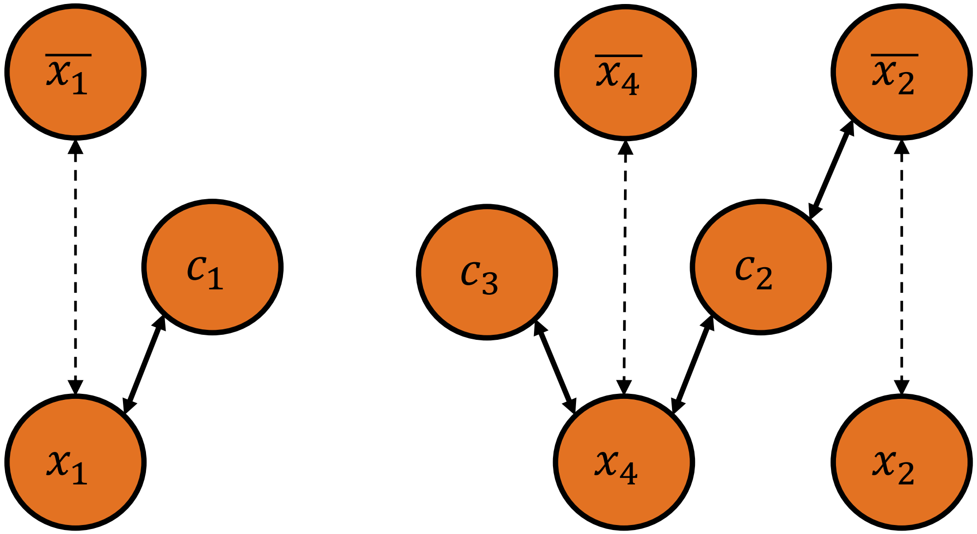
$$(x_1 + x_3)(\overline{x_2} + x_4)(x_4 + \overline{x_3})$$



State representation

$$(x_1)(\overline{x_2} + x_4)(x_4)$$

Setting a variable results in a smaller graph



State representation

$$(x_1)(\overline{x_2} + x_4)(x_4)$$

	$\boldsymbol{x_1}$	$\boldsymbol{x_2}$	$\boldsymbol{x_4}$	$\overline{\boldsymbol{x_1}}$	$\overline{\boldsymbol{x_2}}$	$\overline{\boldsymbol{x_3}}$
$\boldsymbol{c_1}$	1	0	0	0	0	0
$\boldsymbol{c_2}$	0	0	1	0	1	0
$\boldsymbol{c_3}$	0	0	1	0	0	0

Reinforcement:

	No solution	Solved	Usage of Message Passing
Reward per individual	-1	0	$-x$, $with\ Hyperparamter\ x \in \mathbb{R}^{+}$

Network learns to solve problem in as few steps as possible

Network learns to branch only if it is worth the additional punishment each step

Network learns to use costly message passing only if it is worth the additional punishment

Chapter 6: Discussion & Future Work

- Flaws of LN
  - Runtime is where our method must eventually outperform state-of-the-art solvers. Querying Network N_1 might be too expensive for every step. We could either replace it completely with a hardcoded heuristic or only query it in certain situations
  - The whole network might be too lazy to learn to branch and decides to simply rely on VSIDS. We could either train MPN separately first (to already make them both compatible) or force the network to use MP sometimes
- Runtime
  - To achieve good runtime the implementation is crucial. Modern SAT-solver have many sophisticated data structures (2WL, etc.) to help. Our solving algorithms need a similar efficient implementation. See in references for more ideas.
  - https://doi.org/10.1016/j.entcs.2004.10.020 was one of the first parallel multithreaded SAT solvers with shared memory. They reported a really bad performance because of a high number of cache misses, whereas other solvers are optimized for cache usage. We must keep that in mind for our implementation.

- Hyperparameters must be chosen to achieve both good actions and acceptable runtime
- Same Trade-off applies for pre-calculated input features

- Adapt PPO for Network
  - For PPO to be able to train the network effectively, we must be sure that PPO is able to capture the whole plan of the network, i.e. one must choose the episode length accordingly
  - This is correlated with the failure bias of the current system. The network will terminate the search after a specific amount of steps. If the solution was not found, the training process must not weight this episode heavier that if it was found early on
- Implement remaining functionality
  - Check the README of the code base for further information
- Training & Analysis
  - Training does not only show if the proposed method works but can also provide information about the used strategy. This in return could give starting points for further refinement of the architecture.

Thank you for your attention!

Chapter 7: Appendix

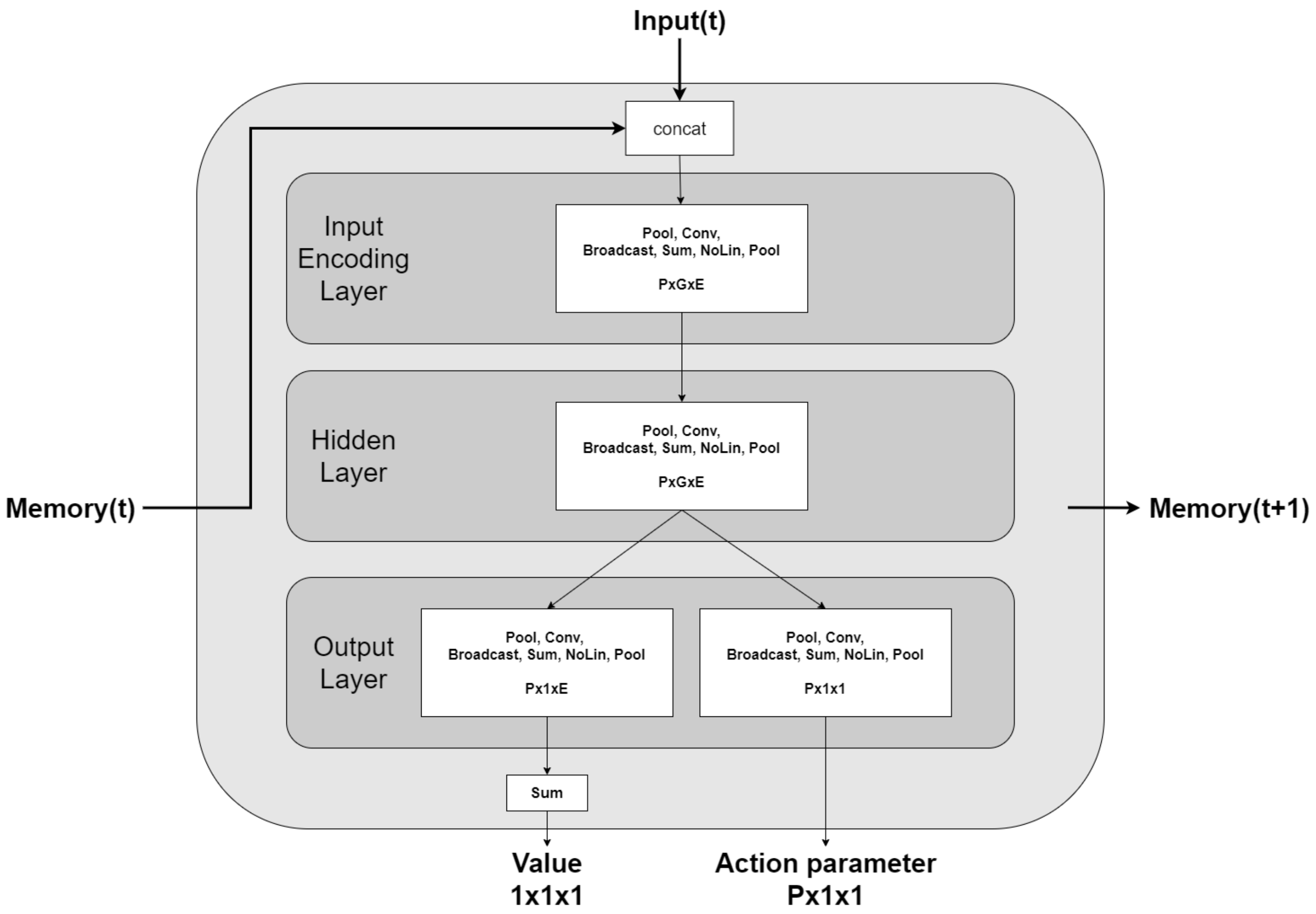
The old approach:

The old approach, started by Yoav Schneider, was using a standard genetic algorithm together with a Local Search solving approach. That means, there is a population of assignments (all variables either True or False) which is refined by local search from one generation to the next. Each individual has a fitness value that equals the total number of satisfied clauses for that assignment. The genetic algorithm uses Selection, Crossover and Mutation to generate new, (hopefully) improved individuals. For this, the genetic parameters are calculated by a neural network. Specifically, we have a NN that takes pre-calculated input features such as break values per variable per individual, etc. and outputs an action probability distribution. We then sample actions from the distribution to modify the genetic algorithm.

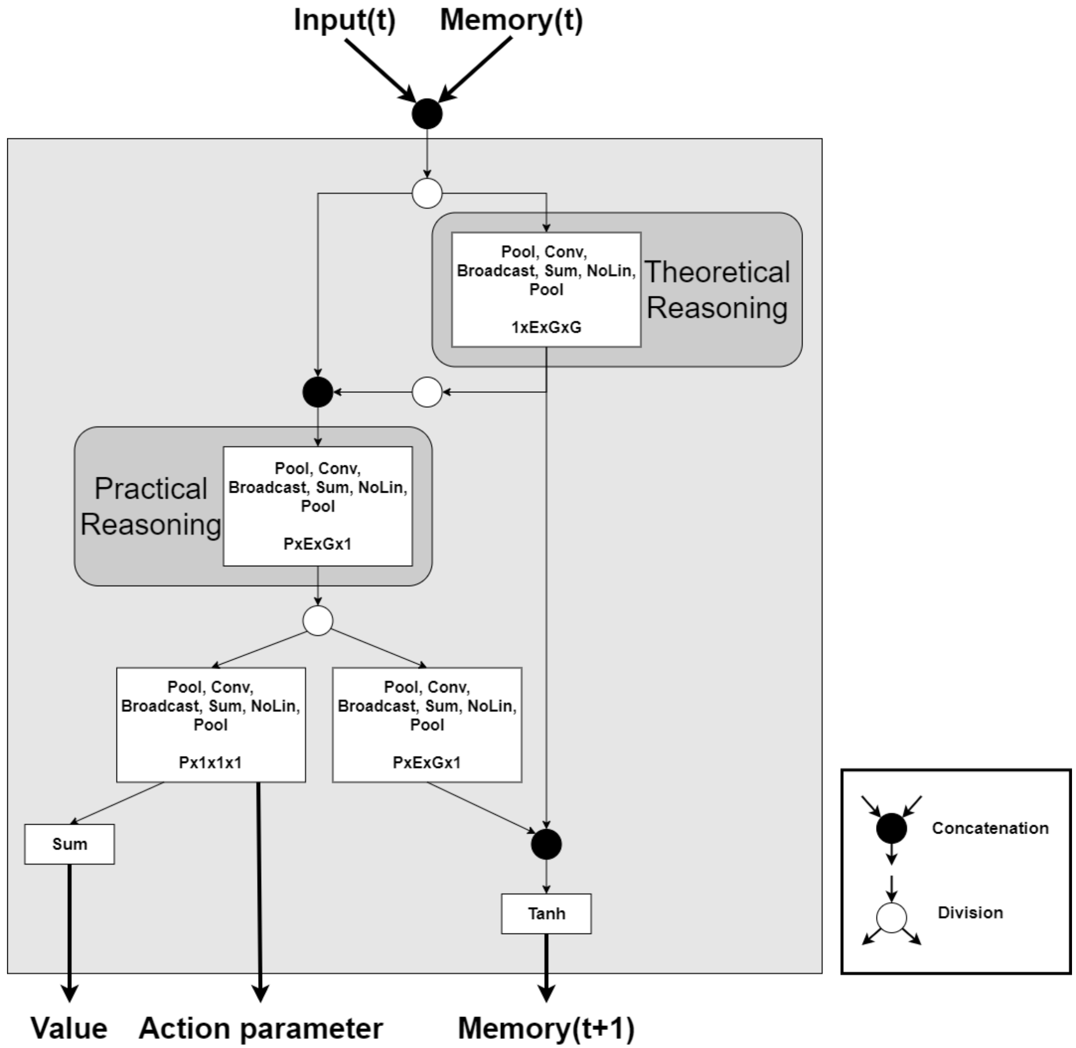
The aim by Yoav Schneider was to surpass the standard genetic algorithm with fixed parameters (similar to this approach https://doi.org/10.1162/EVCO_a_00078). Indeed, there was an improvement w.r.t. the number generations needed to solve a given problem.

Improvements on the old approach:

In WS19, Linus Kreitner (that’s me) followed the future work section of Yoav Schneider to improve the current approach. For this, more relevant input features, such as the problem instance itself, where given to the network. Furthermore, the whole architecture that was previously copied from “Learning to Evolve”, was changed. The goal was to create a network that is specifically tailored for the Boolean Satisfiability Problem. In the following you can see a few of the tested ideas.



The novel (Pool, Conv, Sum, Non-lin, Pool) block is used to enforce permutation equivariance along all dimension, as well as a more memory efficient way of convolution. The network was also transformed into an RNN to pass on information over multiple generations.



This network tries to infer deep features about the problem statement in the theoretical reasoning block over many generations. It has a GxG matrix to store variable-to-variable features. This is similar to what message passing does, only in a less structured way. The practical building block is used to process the previously extracted deep features together with population specific stats to create a plan how to behave next.

The flaws of the old approach:

As mentioned, the goal was to surpass the standard genetic algorithm with fixed parameters. However, it does not really matter if the dynamic evolution strategy was better than the static one, since using genetic algorithms for local search or even local search at all is not competitive with sate-of-the-art solving strategies. As discussed, none of the best solving algorithms works with local search. Moreover, in the SAT community the runtime of a solver is far more interesting than the number of steps needed. For instance, in SAT competitions there is a timeout after 5000 seconds of runtime. So the calculation of complex input features, as well as a complex network architecture is not suitable if we want to make a serious contribution to current research.

The old approach also suffered from the flaws of the local search:

- No learning
- No certain progress towards the solution (local optimum, circles, etc.)
- Number of satisfied clauses does not always indicate if it is close to a solution

Based on these considerations the new backtracking approach was developed.

References / Helpful Literature

Reinforcement Learning:

- Jan Schuchardt and Vladimir Golkov and Daniel Cremers. 2019. Learning to Evolve. https://arxiv.org/abs/1905.03389
- Richard S. Sutton and Andrew G. Barto. 2018. Reinforcement Learning: An Introduction. A Bradford Book, Cambridge, MA, USA. https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
- Frédéric Lardeux, Frédéric Saubion, Jin-Kao Hao. 2006. GASAT: A Genetic Local Search Algorithm for the Satisfiability Problem. https://doi.org/10.1162/evco.2006.14.2.223
- B. Li and Y. Zhang, "A hybrid genetic algorithm to solve 3-SAT problem," *2016 12th International Conference on Natural Computation, Fuzzy Systems and Knowledge Discovery (ICNC-FSKD)*, Changsha, 2016, pp. 476-480. DOI: 10.1109/FSKD.2016.7603220

Message Passing:

- M. Gori, G. Monfardini and F. Scarselli, "A new model for learning in graph domains," *Proceedings. 2005 IEEE International Joint Conference on Neural Networks, 2005.*, Montreal, Que., 2005, pp. 729-734 vol. 2.
- Peter Battaglia et al. 2018. Relational inductive biases, deep learning, and graph networks. https://arxiv.org/abs/1806.01261
- Rasmus Berg Palm and Ulrich Paquet and Ole Winther. 2017. Recurrent Relational Networks. https://arxiv.org/abs/1711.08028

- Daniel Selsam and Matthew Lamm and Benedikt Bünz and Percy Liang and Leonardo de Moura and David L. Dill. 2018. Learning a SAT Solver from Single-Bit Supervision https://arxiv.org/abs/1802.03685
- Vitaly Kurin and Saad Godil and Shimon Whiteson and Bryan Catanzaro. 2019. Improving SAT Solver Heuristics with Graph Networks and Reinforcement Learning. https://arxiv.org/abs/1909.11830

Local Search

- Sixue Liu. 2015. An Efficient Implementation for WalkSAT. https://arxiv.org/abs/1510.07217
- Noureddine Bouhmala, A Multilevel Memetic Algorithm for Large SAT-Encoded Problems. 2012. Evolutionary Computation, 2012, Vol.20(4), pp.641-664. DOI: https://doi.org/10.1162/EVCO_a_00078
- S. Cai, C. Luo and K. Su, "Improving WalkSAT By Effective Tie-Breaking and Efficient Implementation," in *The Computer Journal*, vol. 58, no. 11, pp. 2864-2875, Nov. 2015. https://ieeexplore.ieee.org/document/8213089

Backtracking:

- Lintao, Zhang & Madigan, C.F. & Moskewicz, M.H. & Malik, Suhani. (2001). Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers. 279-285. DOI: 10.1109/ICCAD.2001.968634. https://doi.org/10.1109/ICCAD.2001.968634
- Niklas Een, Niklas Sörensson, 2005. MiniSat – A Solver with Conflict-Clause Minimization. http://minisat.se/downloads/MiniSat_v1.13_short.pdf
- J. P. Marques Silva and K. A. Sakallah, "GRASP-A new search algorithm for satisfiability," *Proceedings of International Conference on Computer Aided Design*, San Jose, CA, USA, 1996, pp. 220-227 https://link.springer.com/chapter/10.1007/978-1-4615-0292-0_7

- Gil Lederman and Markus N. Rabe and Edward A. Lee and Sanjit A. Seshia. 2018. Learning Heuristics for Quantified Boolean Formulas through Deep Reinforcement Learning. https://arxiv.org/abs/1807.08058
- Heule, Marijn & Maaren, Hans. (2009). Look-ahead based SAT solvers. Frontiers in Artificial Intelligence and Applications. 185. 10.3233/978-1-58603-929-5-155. DOI: 10.3233/978-1-58603-929-5-155
- Daniel Selsam and Nikolaj Bjørner. 2019. Guiding High-Performance SAT Solvers with Unsat-Core Predictions. https://arxiv.org/abs/1903.04671

Parallel SAT solvers:

- Heule M.J.H., Kullmann O., Wieringa S., Biere A. 2012. Cube and Conquer: Guiding CDCL SAT Solvers by Lookaheads. In: Eder K., Lourenço J., Shehory O. (eds) Hardware and Software: Verification and Testing. HVC 2011. Lecture Notes in Computer Science, vol 7261. Springer, Berlin, Heidelberg. https://link.springer.com/chapter/10.1007/978-3-642-34188-5_8
- Xu, L. and Hutter, F. and Hoos, H. H. and Leyton-Brown, K. 2008. SATzilla: Portfolio-based Algorithm Selection for SAT. Journal of Artificial Intelligence Research. https://arxiv.org/abs/1111.2249
- Hamadi, Youssef, Jabbour, Said, and Sais, Lakhdar. ‘ManySAT: a Parallel SAT Solver’. 1 Jan. 2010 : 245 – 262. https://content.iospress.com/articles/journal-on-satisfiability-boolean-modeling-and-computation/sat190070

Proximal Policy Optimization

- John Schulman and Filip Wolski and Prafulla Dhariwal and Alec Radford and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms https://arxiv.org/abs/1707.06347
- Arxiv Insights. 2018. Policy Gradient methods and Proximal Policy Optimization (PPO): diving into Deep RL!. YouTube. https://www.youtube.com/watch?v=5P7I-xPq8u8

Efficient Implementation:

- Le Frioux L., Baarir S., Sopena J., Kordon F. 2017. PaInleSS: A Framework for Parallel SAT Solving. In: Gaspers S., Walsh T. (eds) Theory and Applications of Satisfiability Testing – SAT 2017. SAT 2017. Lecture Notes in Computer Science, vol 10491. Springer, Cham https://link-springer-com.eaccess.ub.tum.de/chapter/10.1007%2F978-3-319-66263-3_15
- Osama, M., & Wijs, A. (2019). Parallel SAT simplification on GPU architectures. In L. Zhang, & T. Vojnar (Eds.), Tools and Algorithms for the Construction and Analysis of Systems - 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Proceedings (pp. 21-40). (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics); Vol. 11427 LNCS). Cham: Springer. https://doi.org/10.1007/978-3-030-17462-0_2
- Yulik Feldman and Nachum Dershowitz and Ziyad Hanna. 2005. Parallel Multithreaded Satisfiability Solver: Design and Implementation. Electronic Notes in Theoretical Computer Science, Vol.128(3). https://doi.org/10.1016/j.entcs.2004.10.020