Finding Optimal Paths Using Networks Without Learning—Unifying Classical Approaches

Tomas Kulvicius, Sebastian Herzog, Minija Tamosiunaite, and Florentin Wörgötter

***Abstract*— Trajectory or path planning is a fundamental issue in a wide variety of applications. In this article, we show that it is possible to solve path planning on a maze for multiple start point and endpoint highly efficiently with a novel configuration of multilayer networks that use only weighted pooling operations, for which no network training is needed. These networks create solutions, which are identical to those from classical algorithms such as breadth-first search (BFS), Dijkstra's algorithm, or TD(0). Different from competing approaches, very large mazes containing almost one billion nodes with dense obstacle configuration and several thousand importance-weighted path endpoints can this way be solved quickly in a single pass on parallel hardware.**

***Index Terms*— Deep multilayer network, multiagent systems, multisource shortest paths.**

I. Introduction

PATH planning is a prevalent problem that exists, for example, in traffic control to optimize traffic flow patterns or in the gaming industry. Possibly the best examples of path planning problems can be found in robotics. For example, consider the problem of a mobile robot that has to navigate inside a building from a start point to a target location. Doing this requires that the machine avoids walls and obstacles that it finds doorways and does not fall downstairs. The goal of path or motion planning is to encode these constraints in some kind of appropriate representation framework and use this as an input to the path-planning algorithm to generate commands for the robot for creating an error-free movement trajectory from start point to endpoint. The situation can get more complicated if input information is incomplete or if there are multiple agents (robots) that are supposed to find the shortest path each to the target nearest to them (multisource–multitarget problem). Many of these problems can be solved by finding paths on a grid-based map representation, where the path constraints (e.g., obstacles) are presented as forbidden grid points. Other more general algorithms use (weighted) graphs as representation framework and substantial work have been done since around 1950 to address these kinds of problems.

Manuscript received 14 December 2020; revised 9 April 2021; accepted 6 June 2021. Date of publication 25 June 2021; date of current version 1 December 2022. This work was supported by the European Community's H2020 Programme [Future and Emerging Technologies (FET)] (Plan4Act) under Grant 732266. *(Corresponding author: Tomas Kulvicius.)*

Tomas Kulvicius is with the Department of Computational Neuroscience, University of Göttingen, 37073 Göttingen, Germany, and also with the University Medical Center Göttingen, Child and Adolescent Psychiatry and Psychotherapy, 37075 Göttingen, Germany (e-mail: tomas.kulvicius@uni-goettingen.de).

Sebastian Herzog and Florentin Wörgötter are with the Department of Computational Neuroscience, University of Göttingen, 37073 Göttingen, Germany.

Minija Tamosiunaite is with the Department of Computational Neuroscience, University of Göttingen, 37073 Göttingen, Germany, and also with the Faculty of Computer Science, Vytautas Magnus University, Kaunas, Lithuania.

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TNNLS.2021.3089023.

Digital Object Identifier 10.1109/TNNLS.2021.3089023

II. Related Work

A. State of the Art

The most common classical approaches for path planning are breadth-first search (BFS) [1], [2] as well as the Dijkstra [3], and Bellman–Ford [4], [5] algorithms. These algorithms perform well on graphs and grid-based structures and can solve the so-called single-source shortest path (SSSP) problem; however, they do not scale well with increased dimensions (e.g., nD space, $n > 2$) and path lengths.

Some heuristic, goal-oriented search algorithms, such as greedy BFS (Greedy-BFS, [6], [7]), A* algorithm [8], and its variants (see [9]–[12]), exist, which are faster than the abovementioned algorithms, and however, they can only find solutions for single-source single-target path-finding problems. Furthermore, Greedy-BFS will not always find optimal solutions.

Another class of common approaches, especially in robotics, is sampling-based methods, such as the rapidly exploring random tree (RRT) algorithm (see [13]), and it is variants (see [14]–[16]). RRT-based methods are well suited for continuous spaces, and however, these methods do not perform so well on grids or in complex environments like mazes compared to Dijkstra or A* [17], [18]. Also, the paths found by RRT-based methods are usually not optimal and converge to an optimal solution only asymptotically as the number of samples grows to infinity. Furthermore, these methods require parameter tuning to obtain optimal performance, whereas the above-discussed classical and heuristic search algorithms are parameter-free methods.

The third class of algorithms is based on artificial neural networks. Some early approaches attempted to perform shortest paths search in graphs using Hopfield networks [19], [20], and however, these methods can only solve relatively small graphs (below 100 nodes) and solutions are not always optimal or networks even fail to find solutions, especially if graphs are getting bigger. Later on, other bio-inspired neural networks, which work on grid structures, were proposed [21]–[26]. In these approaches, the environment is represented by a network with inhibitory (to represent obstacles) and excitatory (to represent free spaces) neurons that are

topographically arranged on a grid. Activity in the network is propagated from the source neuron to the neighboring cells and so on. After activity propagation within the whole network is finished, paths are then reconstructed by the following activity gradients. The drawback of these approaches is that they are only suited for small-scale environments (e.g., grids of size less than 1000×1000) since activity in the network very quickly decays to zero (see [24], [26]). Some other approaches first learn an environment representation and then find solutions using reinforcement learning (see [27], [28]). These approaches do not provide optimal solutions in terms of some distance metric but rather an approximate path from source to destination.

Recently, also deep learning approaches utilizing deep multilayer perceptrons (DMLPs; see [29]), deep reinforcement learning approaches [28], [30], [31], fully convolutional networks [32]–[34], long short-term memory (LSTM) networks [18], and graph neural networks [35] have been proposed for solving path-finding problem, too. However, these approaches require learning and, thus, they need a lot of training data. In addition, these approaches will not always return optimal solutions, and in a substantial fraction of cases, they cannot even guarantee that a solution will be found at all (e.g., in cases that are quite different from the ones used to train models).

A general way to address the path-finding problem for many applications is to first create a gradient field that captures the topology of the underlying environment, including start point and endpoint(s) for one or multiple paths. After having achieved this, one can then compute optimal paths following some desired metric (e.g., Manhattan or octile) within that field. Some of the above-discussed algorithms (e.g., bio-inspired networks [24]–[26]) follow this approach and some classical approaches can be used, too, for example, BFS [1], [2], Dijkstra's algorithm [3], modern variants of them [36]–[38], or reinforcement learning approaches such as TD-learning [39]. One central advantage of this is that multisource multitarget path-finding problems can be directly addressed this way.

Recently, a new and very powerful parallel approach, which also does not require learning has been proposed in [40]. This method can generate Euclidean-optimal maps for multiple weighted sources and the authors of this study have shown in quite an exhaustive comparison that it outperforms many other parallel (GPU-based) approaches [2], [41]–[44].

B. Contributions

In this study, we are addressing the problem of how to efficiently create specific activation maps, which have gradients that are optimal for path planning under different metrics and we will show that this can be achieved by a novel class of multilayer neural networks that do not require learning. To this end, we present a set of novel multilayer networks, which consists of only transformed pooling layers (T-POOLs). We show that in this way, we can—on a grid—identically reproduce BFS, but also Dijkstra's algorithm as well as TD(0), known from reinforcement learning [39]. Some approaches exist for neural implementation of Dijkstra's algorithm using the Hopfield network [45] or spiking neural network [46], and however, we are not aware of other neural network architecture, which could emulate different classical algorithms using the same network architecture. In fact, our approach relates to the approach that utilizes distance transforms to generate gradient maps and perform path search [47]. Different from this approach, we use only one filter and perform only one pass compared to several different filters and two passes, which makes our algorithm computationally more efficient.

T-POOL networks do not require training data such that learning is not necessary and this approach can process very large environments on standard GPUs in short time. We demonstrate that we outperform the two fastest—as far as we know—GPU-based algorithms, namely GPU-BFS [2] and OpenGL Shaders [40], in different situations.[1]

III. Methods

A. Network Architecture

T-POOL uses always the same general structure to emulate four different algorithms, BFS, Dijkstra, TD(0), and the activity propagation AP-Net [24], as shown in Fig. 1. Specific definitions for the generic version of the different algorithms are given in Section III.

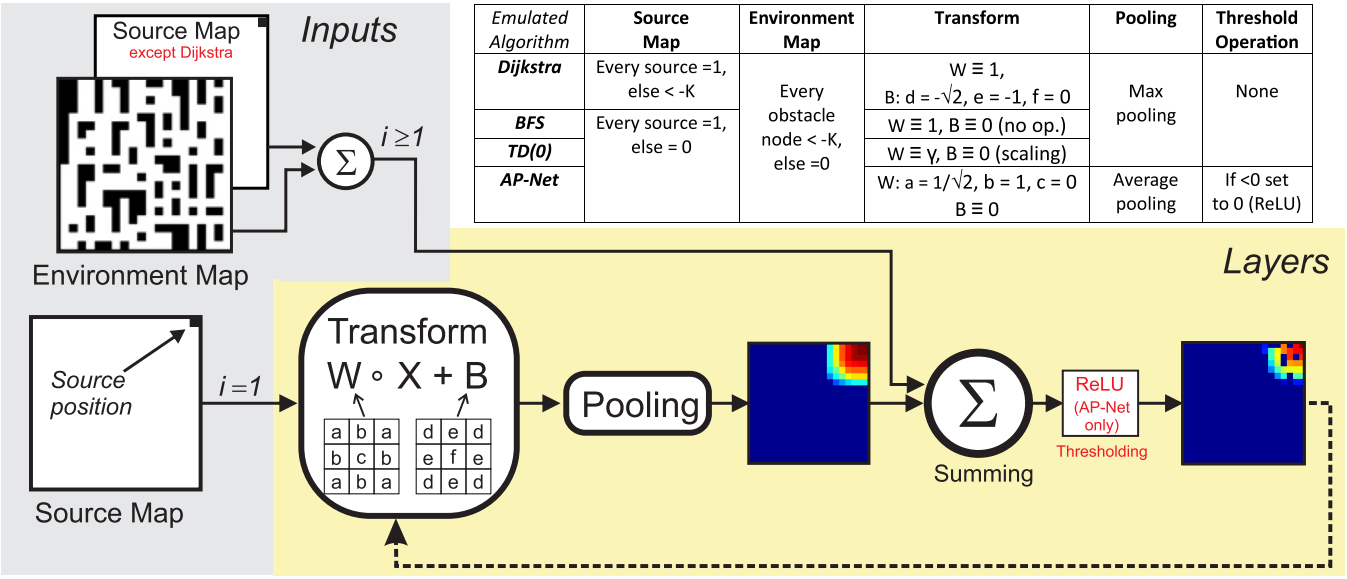
T-POOL needs as inputs two binary images of size $m \times n$: a source map I_s, with $s \geq 1$ sources, and an environment map I_e (gray box in Fig. 1). Environment maps are for all emulations defined in the same way by setting all nodes to $I_e(i, j) = -K$, where an obstacle exists, and to zero else. The value of K can be any integer larger than $2L$, where L is the total number of layers. This setting works for all emulations, where one can show that the actual upper bounds for $-K$ for BFS and Dijkstra are $-L$ and $-L\sqrt{2}$, respectively, but this is not of practical relevance.

To understand this choice, we first need to discuss how the algorithm works, and then, we can come back to the issue why $K = 2L$ is sufficient.

Every T-POOL network consists of L identical layers l_i (see Fig. 1). Networks use different types of transformations with weighting W and biasing B filters of size 3×3. By this, every 3×3 map-patch X is transformed as: $W \circ X + B$, where $\circ$ represents componentwise multiplication (Hadamard product).

Note that the values $a, b, \ldots, e, f$ for the transform (see Fig. 1) are constants and do not have to be learned. In Dijkstra's and the AP-Net algorithm, these values correspond to the costs of horizontal/vertical as well as diagonal moves, i.e., to 1 and $\sqrt{2}$. Note here that, Ap-Net operates multiplicative, and hence, we need to set $(a, b, c) = (\sqrt{2}, 1, 0)$. Different from this, Dijkstra operates subtractive, for which we have to set $(d, e, f) = (-\sqrt{2}, -1, 0)$, instead. In case of BFS, all moves have a uniform cost of 1. Numerically, this corresponds to a no-operation, and in terms of our mathematical transform, this is represented by setting of $(a, b, c) = (1, 1, 0)$. In case of TD(0), these values correspond to the discount factor γ of the generic TD(0) algorithm with $(a, b, c) = (\gamma, \gamma, \gamma)$.

[1] A preprint of an earlier version of this article has been uploaded to https://arxiv.org/abs/2004.00540



Emulated Algorithm	**Source Map**	**Environment Map**	**Transform**	**Pooling**	**Threshold Operation**
Dijkstra	Every source =1, else < -K	Every obstacle node < -K, else =0	W ≡ 1, B: d = -√2, e = -1, f = 0	Max pooling	None
BFS	Every source =1, else = 0		W ≡ 1, B ≡ 0 (no op.)		
TD(0)			W ≡ γ, B ≡ 0 (scaling)		
AP-Net			W: a = 1/√2, b = 1, c = 0 B ≡ 0	Average pooling	If <0 set to 0 (ReLU)

Fig. 1. Network architecture and algorithmic process. All T-POOL networks consist of L identical layers l_i. All networks need a source map and an environment map (with obstacles, black) as inputs. More than one source can exist. Transformation follows: $W \circ X + B$, where X is a 3×3 map-patch and W and B are weighing and biasing filters of the same size respectively. With $\circ$, we denote componentwise multiplication (Hadamard product). Transformation and pooling have a stride 1×1 and zero padding (no subsampling). All filters take a shape as shown in the figure, where coefficients $a, b, \ldots, f$ depend on the desired emulation (see the table). The identity sign ($\equiv$) indicates that all coefficients are set to the same value. Here, for graphical reasons, the colored maps show the results after several iterations. AP-Net in addition needs a thresholding (ReLu) operation, and for Dijkstra, the source map must not be additively reentered.

This is then followed by a pooling step, which operates on the same 3×3 kernel size. We specifically use such a size for transformation and pooling in order to spread activation only to the nearest grid cells (similar to the wavefront expansion algorithm [48]). Otherwise, in case of larger filters, the spread could propagate also to grid cells, which are separated by obstacles.

Note that the transformation for BFS represents a "no-operation": map patches are not altered at all and only max pooling is then performed. This makes T-Pool-BFS very fast (see the following). For TD(0), we note that the transformation is a uniform scaling by γ, which is the discount factor of conventional TD-algorithm (see Section III) Thus, remarkably, TD(0) represents itself as an iterative-multiplicatively scaled version of BFS.

Naturally, the environment map has to be entered into every layer, where—except for Dijkstra—the source map first has to be summed with the environment map, an operation that has to be performed only once, before entering. For Dijkstra, the activation in the map spread using solely a subtractive process, and this makes it necessary to initialize the map outside the sources with negative values to prevent propagation from irrelevant locations via max pooling.

For AP-Net, an additional thresholding operation (ReLU) is needed, too.

These steps are repeated for all layers until the last layer, which produces the output activation map O of size $m \times n$ as an output.

The transformation used for the T-POOL variants of BFS, Dijkstra, and TD(0) ensures that these emulations are mathematically identical to their generic versions and all properties of these algorithms concerning path finding hold then too. This will be further discussed in the appendix.

A graphical visualization of the process of activation map generation using T-POOL-BFS is shown in Fig. 2. Here, we used a maze of size 9×9 and placed one source in the middle. We show the obstacle map (black grid cells denote obstacles), the input activity (source map), and the activity of each max pooling layer. As we can see, after the first layer (see output l_1) activity is propagated from the source only to its neighboring grid cells (except at obstacles) that obtain the values of 1, while the activity at the source cell is increased by one to a value of 2. From layer to layer, activity in the network grows and propagates to grid cells increasingly distant from the source. In this particular example, map generation is complete after nine layers.

This explains now also the choice of setting values at the obstacles to $-K$; values in our T-POOL algorithm increase from layer to layer, as shown in Fig. 2. Thus, in order to always neglect values at the obstacles when performing pooling, we have to ensure that these values remain ≤ 0 until the end of a run as values at nonobstacle grid cells is always positive. For the example of BFS in Fig. 2 and a network with L layers, the maximal value for any grid cell after all iterations is given by its initial value plus L. Hence, if grid cells at obstacles are set to $-K$ with—for BFS—$K = L$, then we assure that values at the obstacles will always be ≤ 0. For other emulations, different values of K can be used, where $K = 2L$ will always suffice.

T-POOL does not have any tunable variables and also the number of layers L, which is the only existing free parameter, can be unequivocally determined. It is identical to

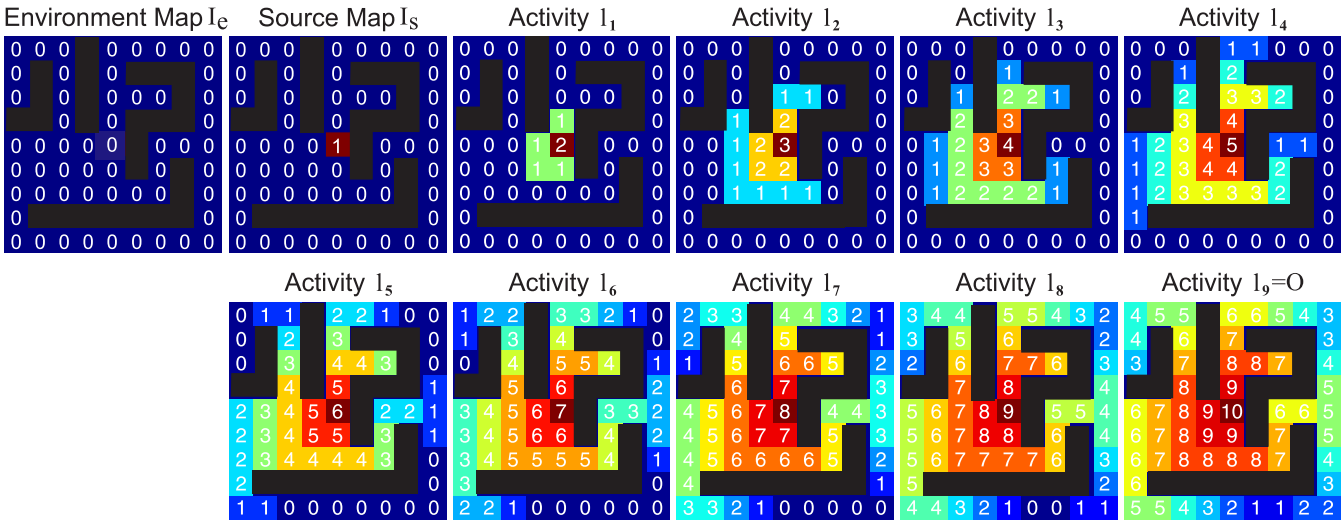


Fig. 2. Illustration of the generation of an activation map using T-POOL-BFS. Black grid cells in the environment map stand for cell values of $-L$, $L = 9$, and represent obstacles. The brown cell with value 1 in the source map denotes the source location. Activation at each layer is shown, and black cells therein represent negative numbers that persist at the obstacle locations throughout all layers. The process ends at layer 9 with a complete output map O.

the maximal path length in an environment, which depends on the location of the source(s), the size of the environment, and the distribution of obstacles. The theoretically existing longest path in any $n \times m$ grid is given as

$$\max(n, m) \times \text{int}\left(\frac{\min(n, m) + 1}{2}\right) + \text{int}\left(\frac{\min(n, m)}{2}\right) \quad (1)$$

which is a path that meanders back and forth between two interleaved comb-like obstacle rows. In a square grid, this number can be approximated by $(n^2/2)$ and this number would have to be matched by L. From our experiments with real maps, we found, however, that this is a highly unrealistic situation and usually $1.5n < L < 2n$ suffices. As an alternative, the algorithm can also be run recursively by adding layer after layer until the activity map does not contain any of the values anymore that had been initialized for the first iteration (except at the obstacles). Thus, L can be set using this procedure. Note, however, that this increases run time (see Appendix C) such that setting L using an estimate is usually preferable, because even if one sets L a bit too high, increase of run time is very small (see the results on run-time comparisons in the following).

B. Algorithmic Complexity

Assuming a grid with $N = n \times m$ nodes and the number of layers L, the run-time complexity of the T-POOL algorithm is

$$O(N, L) = N\,L. \quad (2)$$

Furthermore, assuming that all neurons in one layer can be processed in parallel, e.g., on the GPU in time N/p with p processing units, this leads to an expected run time

$$O_e(N, L, p) = \frac{N}{p}\,L. \quad (3)$$

C. Definitions for the Different Generic Algorithms

1) BFS: For serial CPU implementation, we used a grid representation with octile neighborhood and a first-in-first-out (FIFO) queue, whereas for parallel GPU implementation, an adjacency list was used for encoding the grid. Note that the generic GPU-BFS implementation is based on the method described in [2] using the implementation of available at https://github.com/rafalk342/bfs-cuda.

2) Dijkstra: As in the generic BFS CPU implementation, we used a grid representation with octile neighborhood. In this case, we used an improved Dijkstra's algorithm for path search on grids [49] with a simple queue instead of priority queue where each time the node with the minimum cost was selected from the queue.

3) TD(0): This algorithm had been developed in the context of reinforcement learning [50]. Given a state space (here: grid) with states s (grid cells), the value v of each state is updated iteratively using

$$v(s_t) \leftarrow v(s_t) + \alpha\,[r_{t+1} + \gamma\,v(s_{t+1}) - v(s_t)] \quad (4)$$

where r is a numerical reward, α is the learning rate, which does not affect the final values of v, and γ is the discount factor. Here, we used $r = 1$, $\alpha = 0.1$, and $\gamma = 0.95$.

Updates are achieved by the reinforcement learning agent traveling many times through the state space using some predefined action policy until it reaches a reward. Here, we used an octile *greedy* action policy (north, east, west, south, and all diagonals) without exploration where a random state was selected in cases of $v = 0$.

Values v are equated in T-POOL to the activation values of the grid cells. The rewards r correspond to the values of the source cells. Under these conditions, T-POOL will identically emulate TD(0), where TD(0) converges only asymptotically and is, thus, very slow even on a small grid.

4) AP-Net: We used the shunting model mentioned in [24] where neurons in the network are topologically ordered and

correspond to the grid cells. Each neuron has lateral connections to its closest eight neighbors on the octile grid. The dynamics of the ith neuron at the time t is defined by

$$\frac{dx_i}{dt} = -Ax_i + (B - x_i)\left([I_i]^+ + \sum_{j=1}^{k} w_{ij}[x_j]^+\right) - (D + x_i[I_i]^-) \quad (5)$$

where $k = 8$ is the number of neighboring neurons. $I_i = E$ if there is a source, $I_i = -E$ if there is an obstacle, and $I_i = 0$ anywhere else. Here, $[a]^- = \max\{-a, 0\}$ and $[a]^+ = \max\{a, 0\}$. In this study, we used $dt = 0.01$, $A = 60$, $B = D = 1$, and $E = 100$. For more details on parameter analysis, please refer to [24].

5) OpenGL Shaders: OpenGL shaders [40] utilize GPU rasterization in order to propagate optimal costs on polygonal 2-D environments. This algorithm is based on cone rasterization from sources and obstacle vertices, which partitions an environment into the regions that share the same parent point along the shortest path to the closest source. This method can generate optimal path maps for multiple sources. We provide also an in-depth analysis that shows that their approach outperforms many other GPU-based approaches (see [2], [41]–[44], see discussion in [40]). While it can create the Euclidean-optimal paths, it scales unfavorably with increasing number of obstacles (see Section IV-A). We have in our study used, for comparison, the same profiles as in [40], shown in Appendix B, where L indicates the number of layers used for T-POOL-BFS on these profiles.

D. Path Reconstruction

A path from any given target location to the source (or closest source) can be found from the generated activity map O by following the activity gradient. Hence, we start from the chosen target location (start point) and select a neighboring grid cell with maximum value and repeat this until reaching the source. Note that there can be cases of more than one neighboring cell with maximal value. In such a case, we chose the next cell randomly as all choices lead to equally long paths.

Path reconstruction for BFS and TD(0) was done using paths that follow a Manhattan topography, thus allowing traversing only for north, east, west, and south. For all other algorithms, octile-topography was used, which allows also moving diagonally. The reason for this is that BFS and TD(0) consider vertical/horizontal and diagonal transitions equally (which leads to rectangular distance pattern), whereas Dijkstra's algorithm and AP-Net consider different costs for vertical/horizontal (cost $= 1$) and diagonal (cost $= \sqrt{2}$) transitions (which leads to octile [Dijkstra] or approximately Euclidean [AP-Net] distance pattern). To obtain the shortest distance in terms of path length (with respect to the Manhattan distance), but not in terms of the number of steps, here, we considered a Manhattan topography for BFS and TD(0). If one would use octile topography for path reconstruction, then one would obtain the shortest path in terms of number of steps but not in terms of octile distance.

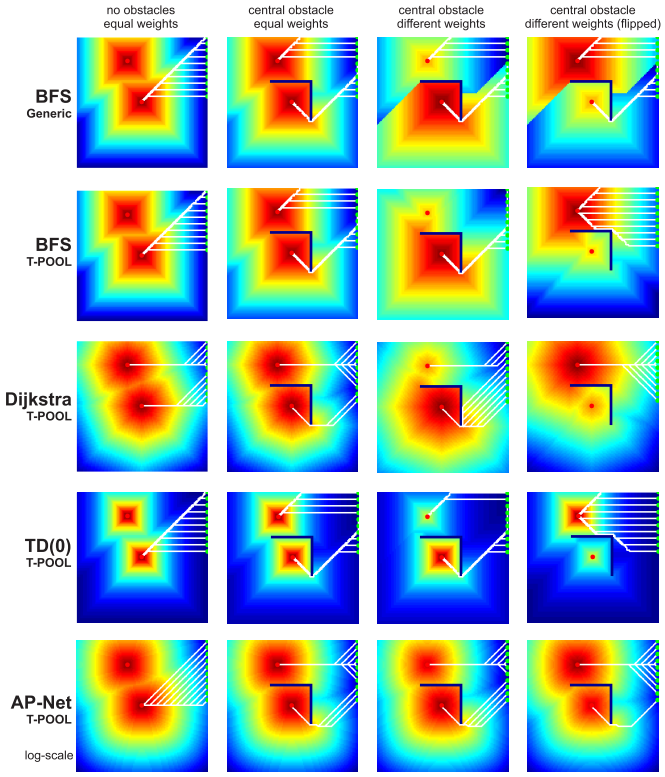


Fig. 3. Comparison of different algorithms in a square grid with $n = L = 100$ for T-POOL. Except for BFS, we only show results for T-POOL because they are identical to the generic versions of the investigated algorithms. Parameters: for BFS and Dijkstra, weighted sources (columns 3 and 4) differ by 25 (1 and 26). Other than that, BFS and Dijkstra are parameter-free. The generic AP-Net parameters were $dt = 0.01$, $A = 60$, $T = 100$ (steps), unweighted sources $= 100$, and weighted sources: 100 and 10^-5, which has to be a number a bit larger than zero. In case of T-POOL-AP-Net, source values were either 1 or 10^-7. For TD(0), we have $\gamma = 0.95$, weighted sources with values 1 and 0.6. Colors encode activity, blue: small values and red: large values. Activity maps for AP-Net are plotted on a log scale all others on a linear scale.

E. Hardware and Implementation Details

For all cases, except for Fig. 3, we used parallel GPU implementations with an NVIDIA GTX 1080 Ti on a PC with Intel Xeon Silver 4114 CPU (2.2 GHz). Parallel T-POOL-BFS/Dijkstra was implemented using Tensorflow and Keras API, whereas generic, parallel BFS was implemented using C++. All simple grids (Fig. 3) were calculated in MATLAB on the same PC.

IV. Results

Fig. 3 shows the activity maps obtained by the different algorithms using two sources with equal or different weights, with or without a central obstacle. The white lines show reconstructed paths from different starting points (green). Paths were generated using the corresponding reconstruction metrics (Manhattan for BFS and TD(0), and octile for all others).

Only for BFS, we show the comparison between a generic implementation and T-POOL-BFS because, only in this case, there is a difference. For all other algorithms, there are no differences between their generic version and T-POOL.

Generic BFS addresses weighted sources in a way that leads to clashes between the two gradients, an undesired effect that is not found in any of the other algorithms. These clashes, which are large value differences at the border where gradients from the two sources meet, arise due to nature of the BFS algorithm. BFS explores states from both sources in parallel and stops whenever all states on the grid are explored once. The values in the BFS gradient map correspond to the number of states (transitions) from the closest source. In case of equally weighted sources, the values at the border, where two gradients meet, will be the same or differ by 1 (depending on whether the number of steps between two sources is even or odd) since we get the same number of transitions to one or the other source, which consequently leads to a smooth gradient transition at the border. In case of weighted sources, the gradients will meet at the same place but the values at the border, where two gradients meet, will be different due to unequal initialization of source values. For our example, with sources of 1 and 26, at the border value, differences will, thus, be 25 or 26 (depending on whether the number of steps between two sources is even or odd). All the other algorithms will explore and/or update states multiple times, which consequently leads to smooth transitions also in case of weighted sources, and however, these algorithms are more complex and slower than BFS.

For generic TD(0), an octile greedy action policy is assumed as this leads to identical results by T-POOL when using the transformation, as shown in Fig. 1. Note that the gradient decreases exponentially for TD(0) and linearly for BFS.

For AP-Net, we had to use logarithmic scaling for the color code. The activity in AP-Net drops extremely fast when moving away from a source (see also [24]). This leads to the fact that grid of more than 500×500 nodes cannot be solved anymore because of numerical zeroing.

A. Run Time

Run-time evaluations are difficult because they are governed by the hardware, and algorithmic performance also strongly depends on the definition of the problem. From an extended literature search, we found that the parallel implementation of BFS mentioned in [2] and the OpenGL Shaders algorithm from [40] are substantially faster than other algorithms for generation of activity maps. All run-time measurements shown next use T-POOL-BFS.

Fig. 4(a) shows that OpenGL Shaders is the fastest algorithm for few obstacles but scales unfavorably when the number of vertices (corners) from the obstacles in an environment increases. T-POOL-BFS remains slower than BFS, even on the faster NVIDIA 1080 Ti GPU, but for more than 64 rectangular obstacles (256 vertices), it outperforms OpenGL Shaders. Note that in our case, we had set the source in the bottom-left corner, which is the worst case with respect to computation time and T-POOL run-time scales down by a factor of two if the source were located in the middle (best case).

The power of T-POOL-BFS becomes visible, however, when considering multiple weighted sources on different grid sizes [see Fig. 4(c)]. Generic BFS, which is very fast for unweighted sources (green), cannot address this in parallel (see also Fig. 3). It needs to be rerun for every source and all results combined using a max operation (blue). This leads to the situation that T-POOL-BFS (black) remains faster for many cases.

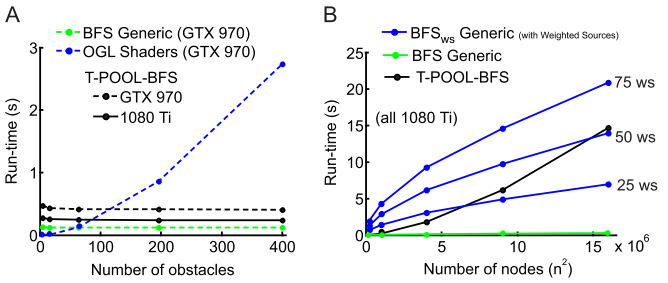


Fig. 4. Run-time comparison for different, highly parallel algorithms. (a) Run-time comparison to OpenGL Shaders and generic BFS for different numbers of obstacles and using different GPUs. Data for OpenGL Shaders were taken from [40], and BFS and T-POOL-BFS were tested on the same grids as in this publication (replotted in Fig. 9) with $n = 1000$ and $L = 1176$, 1088, 1044, 1026, and 1018 for 4, 16, 64, 196, and 400 obstacles, respectively. The source was in the bottom-left corner (worst case). (b) Comparison with the generic BFS algorithm on square grids with one source or multiple weighted sources (ws). In case of T-POOL-BFS, we used $L = n$.

To measure the influence of different settings on the run time of T-POOL, in the following, we define the linear grid size as n and consider here square grids with n^2 grid cells ("nodes"). We used empty maps without obstacles because, for T-POOL, the number of operations does not depend on the number of obstacles and the number of sources.

Panel A in Fig. 5 shows that run time increases linearly against the number of nodes n^2 when keeping the number of layers constant. Similarly, linear growth is also observed when keeping the grid size n constant and increasing L [see Fig. 5(b)].

From above, it is clear that more layers are needed when the grid gets bigger because this leads to longer paths for which the network has to be increased. Because T-POOL-BFS traverses with an octile pattern for activity map building, the shortest possible longest traverse that has to be covered is $0.5n$ (empty square grid, source in the middle). In panel C, we thus consider the number of nodes n^2 together with a changeable number of layers L and we let L depend on the grid size for approximating the fact that in larger grids, paths are longer. As expected from panels A and B, these curves now linearly follow n^3 with slopes that increase for increasing L also in a linear manner.

The above run-time estimates hold as long as T-POOL-BFS runs essentially in forward mode without (too many) iterations i. Note, however, that the overhead of having to pass information iteratively back to the start of a new batch of layers will remain tiny if this happens just a few times.

B. Large Maps and Limitations

The simple algorithmic structure of T-POOL-BFS allows addressing very large systems in one forward pass. Current limits on the NVIDIA GTX 1080 Ti are $n_{\max} = 30\,000$ (equivalent to 900 000 000 nodes) and $L_{\max} = 6000$, where

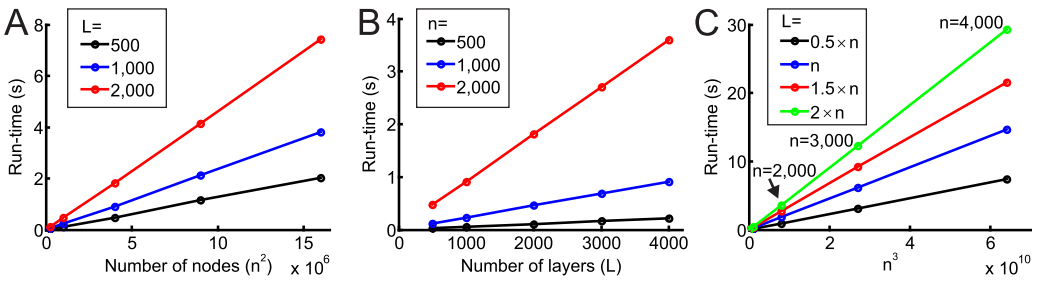


Fig. 5. Run-time evaluation of T-POOL-BFS. (a) Run time versus different numbers of nodes n^2 (linear grid size: $n = [500; 1000; 2000; 3000; 4000]$ for a fixed number of layers L. (b) Run time versus different numbers of layers ($L = [500; 1000; 2000; 3000; 4000]$) for a fixed linear grid size n. (c) Run time versus n^3 ($n = [500; 1000; 2000; 3000; 4000]$) and different numbers of layers ($L = [0.5 \times n, n, 1.5 \times n, 2 \times n]$).

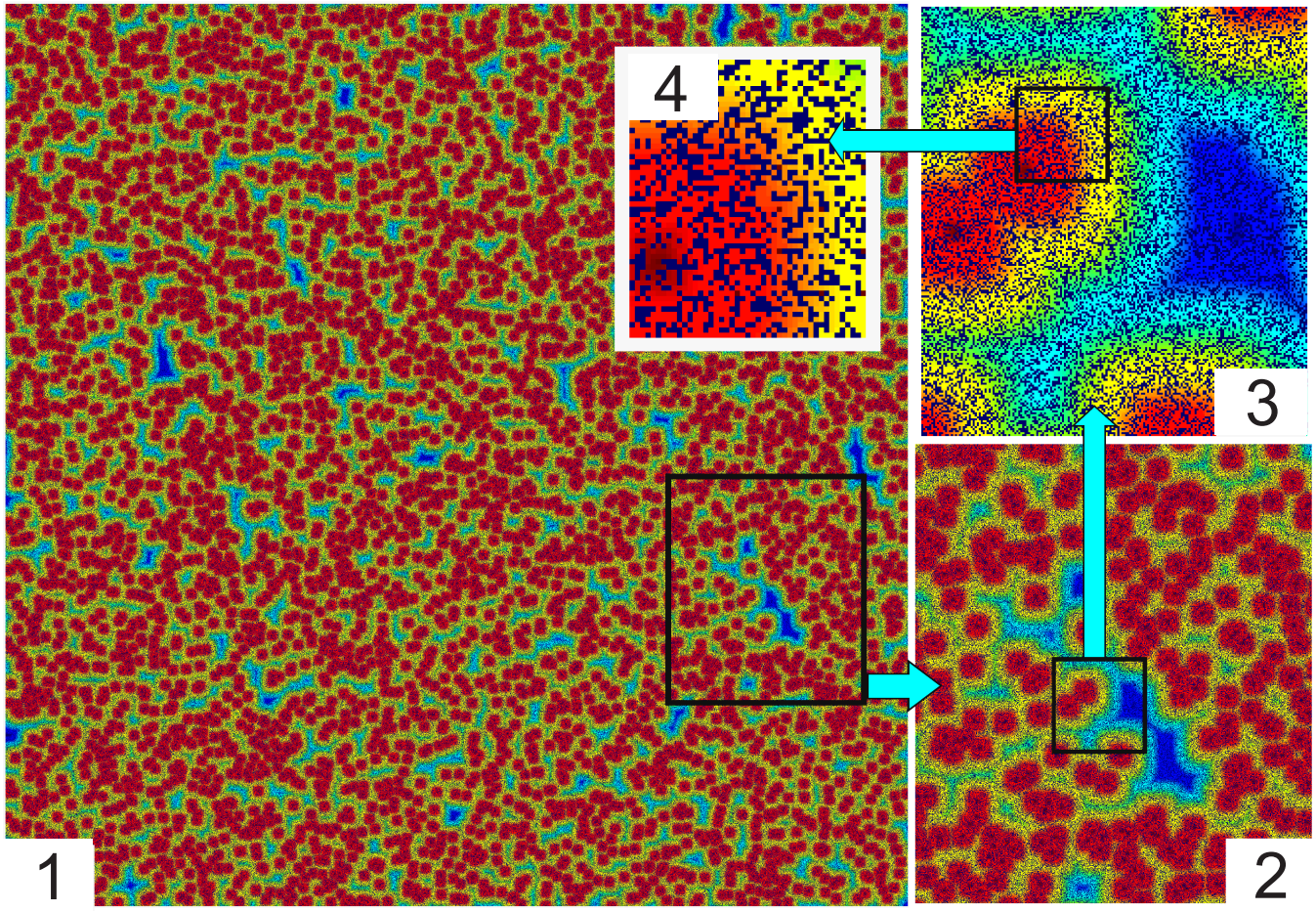


Fig. 6. Example of a huge square grid with $n = 26\,000$ corresponding to 676 000 000 nodes. Panel 1 shows the full grid with 5000 sources and panels 2–4 show magnifications to make the obstacles visible (panel 4).

$n_{\max}$ is the maximal linear grid size and $L_{\max}$ is the maximal number of layers possible for running T-POOL-BFS in one forward pass. Note that the number of required layers L will decrease and so does the run time, as soon as more than one source exists. As mentioned above, from our experiments with real maps, we found that usually, $1.5n < L < 2n$ suffices if there is a single source. Thus, if we assume that $L = 2n$, then we can run systems with $n = 3000$ still in one forward pass using $L_{\max} = 6000$. Under the same assumption ($L = 2n$), we would need $L = 60\,000$ for the maximal possible grid with $n_{\max} = 30\,000$. This would still only require $i = 10$ iterations leading to a negligible temporal overhead by this recursion such that the extrapolation of the curves in Fig. 5 still holds.

Fig. 6 shows an example of a grid with $n = 26\,000$ and 5000 sources and very many obstacles. To compute this, we needed 650 layers and a run time of only 111.7 s.

V. Application on a Real Map

The analysis above has shown that this set of methods can be applied on any path-finding problem on a grid. Hence, here, we show only one more specific example (see Fig. 7). We applied our method to a multisource–multitarget task testing our network on a real taxi scenario in Berlin, where there

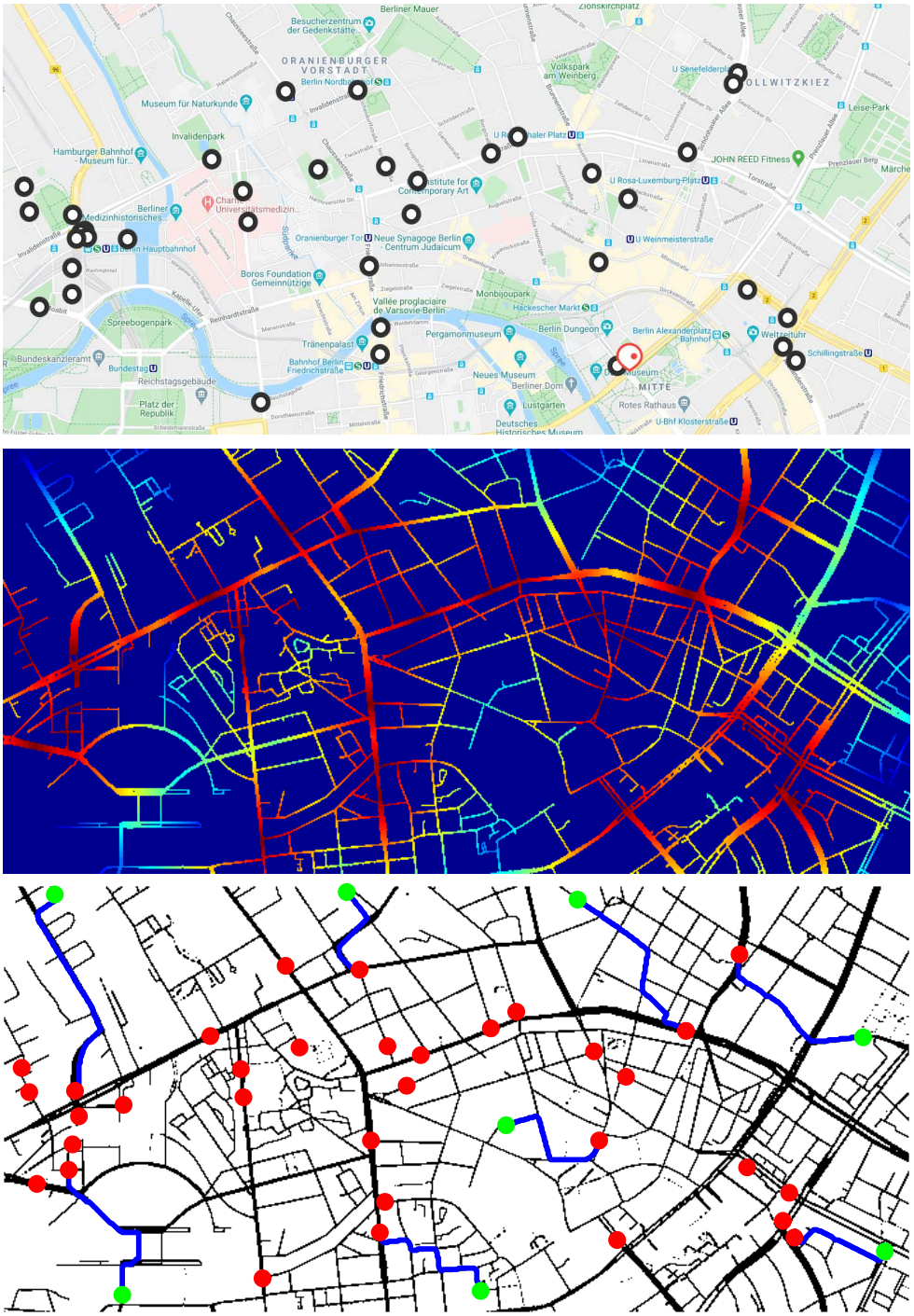


Fig. 7. Multisource–multitarget path-finding problem in a real taxi scenario. Top: section of a Berlin city map (332×709) with disks showing taxi positions obtained on March 3, 2020 at 2:19 P.M., which is a screenshot from the online application available at https://www.taxi.de. Middle: multisource activity map using T-POOL-BFS (160 layers). Bottom: shortest paths (blue lines) from customers (green dots) to taxis (red dots) are shown. Customer positions were defined manually.

are multiple taxi cabs and multiple customers. The task was to find the closest taxi cab for each customer. To be realistic as possible, we obtained taxi cab positions in the area of Berlin close to the train station at a specific moment in time using an online taxi application provided by https://www.taxi.de. The streets were extracted automatically using our own written program. Positions of the taxi cabs were set as sources (in total 33, taxi cabs that were very close to each other were marked with one source) and an activity map was generated using T-POOL-BFS. Finally, eight customer positions were defined manually and the optimal paths were reconstructed from the activity map for each customer as described above. The results are shown in Fig. 7. In the top panel, we show taxi locations in Berlin close to the main train station from an internet-taxi-app at one given point in time. In the middle and the bottom panel, we show the resulting activity map

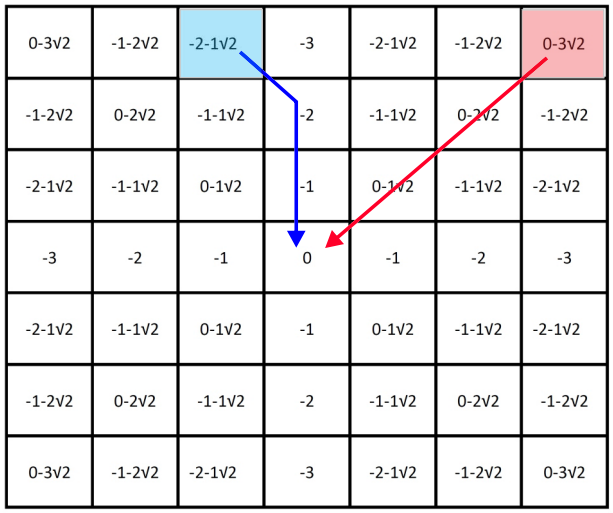


Fig. 8. Activity map for T-POOL-Dijkstra for one source in the middle and three layers. Two paths are shown (blue and red). The blue path requires two steps of length 1 and one step of length $\sqrt{2}$, whereas the red path requires three steps of length $\sqrt{2}$ to get to the source. These values are found at the start locations marked by the red and blue grid cells.

for 33 sources and the reconstructed shortest paths for eight customers to the closest taxi cabs, respectively. We used a network with 160 layers and it took only 14.1 ms to generate the activity map (resolution 332×709). The reconstructed paths (blue trajectories) show that the closest taxi cabs and the shortest paths were found for all eight customers.

VI. Conclusion

Thus, here, we have presented a framework, which allowed us to identically reproduce the behavior of several classical algorithms [BFS, Dijkstra, and TD(0)] by using a set of multilayer networks without learning. In this way, we could efficiently address the problem of creating specific activation maps, which are optimal for path planning under different metrics. We have shown that the gradients within these maps allow for path finding from multiple start to multiple end points. The fastest variant of this group of networks (T-POOL-BFS) outperforms competing approaches when addressing multiple weighted sources. Such problems are common, for example, when considering a group of salesmen who wish to arrive at more probable buyers. Also, the proposed approach is suitable for the animation of multiple agents moving toward different goals in a virtual environment. In addition to this, T-POOL networks scale favorably with an increasing number of sources because the maximally needed number of layers L depends on the maximal path length, which gets shorter the more sources exist. The simplicity and efficiency of this new class of networks may, thus, lend itself also to applications that combine T-POOL nets with other network structures.

Appendix A
On the Equivalence Between the Algorithms

Proof of the identity between the different algorithms [except TD(0)] rests on the argument of their propagation rules and specifically on the propagation diagrams that arise from them. In Fig. 8, we show the analytical values of the activity map from a single source for $L = 3$ using T-POOL-Dijkstra, which is more complex than BFS.

The source has been initialized with zero to make results better interpretable. Results are given as $-p - q\sqrt{2}$, with integers $p, q \geq 0$. Propagation will continue in the same way when using more layers.

For every grid cell, the values in Fig. 8 correspond exactly to the distance to the source under an octile metric identical to the results from classical Dijkstra. Thus, this propagation rule and its iteratively calculated propagation diagram are identical to Dijkstra. Ap-Net operates in the same way as Dijkstra but uses a multiplicative transform. The same derivation as in Fig. 8 could, thus, be performed for Ap-Net, too (not shown). BFS uses an even simpler propagation rule (all values in the transform are "1") and the corresponding propagation diagram is, thus, trivial. It has a continuous one-by-one drop (as also shown in Fig. 2). Hence, the gradient fields for these three cases are identically reproduced by T-POOL. Due to this, also the paths calculated from them are identical between the original algorithms and their T-POOL emulations.

TABLE I

Mean Absolute Error Over All Grid Cells Between T-POOL-TD and Generic TD(0) for Different Discount Factor γ and Equal or Different Rewards r. Network Parameters: $n = 100$, $L = 400$, Obstacle as in Fig. 3. TD(0) Parameters: 500 000 Learning Episodes and Learning Rate $\alpha = 1.0$. All Maps Were Normalized Between 0 and 1

	$r_1 = 1.0$ $r_2 = 1.0$	$r_1 = 0.8$ $r_2 = 0.6$	$r_1 = 0.6$ $r_2 = 0.8$
$\gamma = 0.95$	8.8846×10^{-17}	1.8115×10^{-10}	1.7868×10^{-10}
$\gamma = 0.90$	1.4578×10^{-17}	3.4777×10^{-17}	2.8423×10^{-17}
$\gamma = 0.85$	2.3479×10^{-17}	1.8680×10^{-17}	1.3512×10^{-17}

For TD(0), it is not possible to easily arrive at a rigorous analytical argument because the update of the value table of classical TD(0) follows a stochastic process requiring very many iterations until convergence. We have, therefore, in Table I numerically compared the results from classical TD(0) with those obtained by T-POOL-TD for a square grid with $n = 100$ with two sources and L-shape obstacle (see Fig. 3) for three different discount factors γ and three different rewards r. Note that rewards in TD(0) correspond to the weights of the sources in T-POOL-TD. In all nine cases, the mean absolute error over all grid cells between the maps of TD(0) and T-POOL-TD is below 10^{-9}, which demonstrates that T-POOL-TD can very precisely reproduce generic TD(0).

As noted above, TD(0) requires many episodes until final convergence. In Table II, the mean absolute errors over all grid points between value/activity maps of TD(0) and T-POOL-TD(0) on a 100×100 grid with two equally weighted sources and no obstacles are given. As results demonstrate, TD(0) requires more than 200 000 episodes to achieve an error below 10^{-3} (maps were normalized between 0 and 1) with a maximal learning rate $\alpha = 1.0$. After 500 000 episodes, the error is below 10^{-15} and can be neglected, and however, for 5^5 episodes, it takes on average 83.33 s with SD = 9.56 s

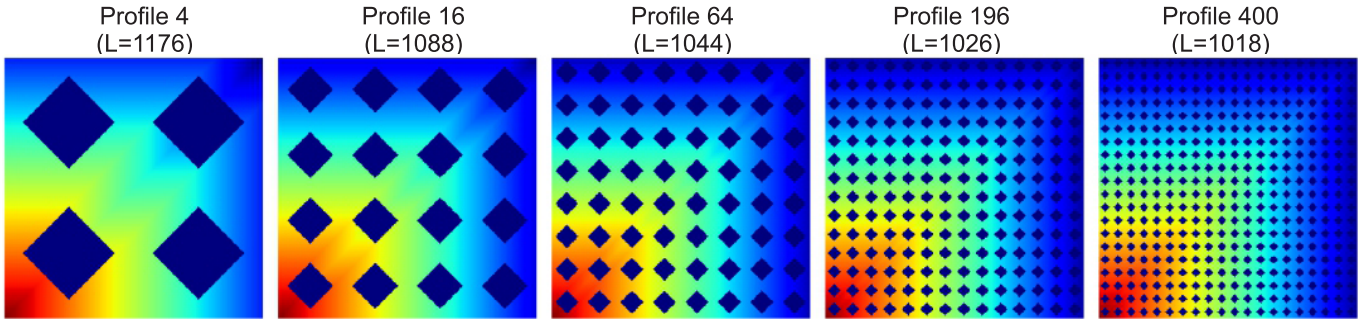


Fig. 9. Obstacle profiles replotted from [40] used here, too, for the comparison between T-POOL-BFS and OpenGL Shaders. L is the number of layers needed for T-POOL-BFS.

TABLE II
MEAN ABSOLUTE ERROR OVER ALL GRID CELLS BETWEEN T-POOL-TD AND GENERIC TD(0) VERSUS NUMBER OF EPISODES FOR TD(0). LEARNING WITH $r_1 = r_2 = 1.0$, NO OBSTACLE. PARAMETERS AS IN TABLE I

Number of episodes	Mean absolute error
100,000	0.0234
200,000	0.0034
300,000	1.4217×10^{-5}
500,000	1.0474×10^{-16}

TABLE III
RUN-TIME COMPARISON OF DIFFERENT IMPLEMENTATIONS. THE GRID SIZE OF 2000×2000 WAS USED IN BOTH CASES

Method	Run-time (s)	Ratio
T-POOL-BFS, one-pass	1.8177	-
T-POOL-BFS, recursive	68.2679	37.56

to compute the TD(0) map, whereas for our T-POOL-TD, it takes on average 3.26 s with SD = 0.046 s (MATLAB CPU implementation on an Intel Core i7-1165G7, 2.80-GHz processor).

APPENDIX B
OBSTACLE PROFILES FOR OPENGL SHADERS

In Fig. 9, we replot the profiles used to compare T-POOL-BFS to the OpenGL Shaders algorithm from [40].

APPENDIX C
RUN-TIME EVALUATION FOR ITERATIVE VERSUS MULTILAYER IMPLEMENTATION

We also compared the run time of T-POOL-BFS (as in Fig. 4 for two extreme cases, where in one case, we ran T-POOL with 2000 max-pooling layers for one iteration (one forward pass), and in the other case, we ran this as single-layer network for 2000 iterations recursively, i.e., the output was used as the next input. The results are shown in Table III where we can see that the one-forward-pass architecture is almost 40 times faster than the recursive one.

REFERENCES

[1] E. F. Moore, "The shortest path through a maze," in *Proc. Int. Symp. Theory Switching*, 1959, pp. 285–292.
[2] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 117–128, Sep. 2012.
[3] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Math.*, vol. 1, no. 1, pp. 269–271, Dec. 1959.
[4] R. Bellman, "On a routing problem," *Quart. Appl. Math.*, vol. 16, no. 1, pp. 87–90, 1958.
[5] L. R. Ford, "Network flow theory," Rand Corp., Santa Monica, CA, USA, Tech. Rep. P-923, 1956.
[6] J. Pearl, *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Reading, MA, USA: Addison-Wesley, 1984.
[7] S. Russell and P. Norvig, "Artificial intelligence: A modern approach," Tech. Rep., 2002.
[8] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. SSC-4, no. 2, pp. 100–107, Jul. 1968.
[9] R. E. Korf, "Depth-first iterative-deepening: An optimal admissible tree search," *Artif. Intell.*, vol. 27, no. 1, pp. 97–109, 1985.
[10] S. Koenig, M. Likhachev, and D. Furcy, "Lifelong planning A*," *Artif. Intell.*, vol. 155, nos. 1–2, pp. 93–146, 2004.
[11] X. Sun, S. Koenig, and W. Yeoh, "Generalized adaptive A," in *Proc. 7th Int. J. Conf. Auto. Agents Multiagent Syst. (AAMAS)*, vol. 1, 2008, pp. 469–476.
[12] D. D. Harabor and A. Grastien, "Online graph pruning for pathfinding on grid maps," in *Proc. AAAI*, 2011, pp. 1114–1119.
[13] S. M. LaValle, "Rapidly-exploring random trees: A new tool for path planning," Tech. Rep., 1998.
[14] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *Int. J. Robot. Res.*, vol. 30, no. 7, pp. 846–894, Jun. 2011.
[15] F. Islam, J. Nasir, U. Malik, Y. Ayaz, and O. Hasan, "RRT*-smart: Rapid convergence implementation of RRT* towards optimal solution," in *Proc. IEEE Int. Conf. Mechatronics Autom.*, Aug. 2012, pp. 1651–1656.
[16] J. D. Gammell, T. D. Barfoot, and S. S. Srinivasa, "Batch informed trees (BIT*): Informed asymptotically optimal anytime search," *Int. J. Robot. Res.*, vol. 39, no. 5, pp. 543–567, Apr. 2020.
[17] L. Knispel and R. Matousek, "A performance comparison of rapidly-exploring random tree and Dijkstra's algorithm for holonomic robot path planning," Inst. Autom. Comput. Sci., Fac. Mech. Eng., Brno Univ. Technol., Brno, Czechia, Tech. Rep., 2013.
[18] M. J. Bency, A. H. Qureshi, and M. C. Yip, "Neural path planning: Fixed time, near-optimal path generation via oracle imitation," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, 2019, pp. 3965–3972, doi: 10.1109/IROS40897.2019.8968089.
[19] D.-C. Park and S.-E. Choi, "A neural network based multi-destination routing algorithm for communication network," in *Proc. IEEE Int. Joint Conf. Neural Netw., IEEE World Congr. Comput. Intell.*, vol. 2, May 1998, pp. 1673–1678.
[20] F. Araujo, B. Ribeiro, and L. Rodrigues, "A neural network for shortest path computation," *IEEE Trans. Neural Netw.*, vol. 12, no. 5, pp. 1067–1073, Sep. 2001.
[21] R. Glasius, A. Komoda, and S. C. A. M. Gielen, "Neural network dynamics for path planning and obstacle avoidance," *Neural Netw.*, vol. 8, no. 1, pp. 125–133, Jan. 1995.
[22] R. Glasius, A. Komoda, and S. C. A. M. Gielen, "A biologically inspired neural net for trajectory formation and obstacle avoidance," *Biol. Cybern.*, vol. 74, no. 6, pp. 511–520, Jun. 1996.

[23] N. Bin, C. Xiong, Z. Liming, and X. Wendong, "Recurrent neural network for robot path planning," in *Proc. Int. Conf. Parallel Distrib. Computing: Appl. Technol.*, 2004, pp. 188–191.
[24] S. X. Yang and M. Meng, "Neural network approaches to dynamic collision-free trajectory generation," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 31, no. 3, pp. 302–318, Jun. 2001.
[25] H. Qu, S. X. Yang, A. R. Willms, and Z. Yi, "Real-time robot path planning based on a modified pulse-coupled neural network model," *IEEE Trans. Neural Netw.*, vol. 20, no. 11, pp. 1724–1739, Nov. 2009.
[26] J. Ni, L. Wu, P. Shi, and S. X. Yang, "A dynamic bioinspired neural network based real-time path planning method for autonomous underwater vehicles," *Comput. Intell. Neurosci.*, vol. 2017, pp. 1–16, Feb. 2017.
[27] E. Rueckert, D. Kappel, D. Tanneberg, D. Pecevski, and J. Peters, "Recurrent spiking networks solve planning tasks," *Sci. Rep.*, vol. 6, no. 1, p. 21142, Aug. 2016.
[28] A. Banino *et al.*, "Vector-based navigation using grid-like representations in artificial agents," *Nature*, vol. 557, no. 7705, pp. 429–433, May 2018.
[29] A. H. Qureshi, A. Simeonov, M. J. Bency, and M. C. Yip, "Motion planning networks," in *Proc. Int. Conf. Robot. Automat. (ICRA)*, 2019, pp. 2118–2124, doi: 10.1109/ICRA.2019.8793889.
[30] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sep. 2017, pp. 31–36.
[31] A. I. Panov, K. S. Yakovlev, and R. Suvorov, "Grid path planning with deep reinforcement learning: Preliminary results," *Procedia Comput. Sci.*, vol. 123, pp. 347–353, Jan. 2018.
[32] N. Perez-Higueras, F. Caballero, and L. Merino, "Learning human-aware path planning with fully convolutional networks," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2018, pp. 1–6.
[33] Y. Ariki and T. Narihira, "Fully convolutional search heuristic learning for rapid path planners," 2019, *arXiv:1908.03343*. [Online]. Available: http://arxiv.org/abs/1908.03343
[34] T. Kulvicius, S. Herzog, T. Lüddecke, M. Tamosiunaite, and F. Wörgötter, "One-shot multi-path planning using fully convolutional networks in a comparison to other algorithms," *Frontiers Neurorobotics*, vol. 14, p. 115, Jan. 2021.
[35] P. Veličković, R. Ying, M. Padovano, R. Hadsell, and C. Blundell, "Neural execution of graph algorithms," 2019, *arXiv:1910.10593*. [Online]. Available: http://arxiv.org/abs/1910.10593
[36] I. Zarembo *et al.*, "Pathfinding algorithm efficiency analysis in 2D grid," in *Proc. 9th Int. Sci. Practical Conf.*, vol. 1, 2013, pp. 46–50.
[37] Z. A. Algfoor, M. S. Sunar, and H. Kolivand, "A comprehensive study on pathfinding techniques for robotics and video games," *Int. J. Comput. Games Technol.*, vol. 2015, pp. 1–11, Apr. 2015.
[38] L. Dhulipala, G. E. Blelloch, and J. Shun, "Theoretically efficient parallel graph algorithms can be fast and scalable," in *Proc. 30th Symp. Parallelism Algorithms Archit.*, Jul. 2018, pp. 393–404.
[39] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Mach. Learn.*, vol. 3, no. 1, pp. 9–44, Aug. 1988.
[40] R. Farias and M. Kallmann, "Optimal path maps on the GPU," *IEEE Trans. Vis. Comput. Graphics*, vol. 26, no. 9, pp. 2863–2874, Mar. 2019.
[41] L. Luo, M. Wong, and W.-M. Hwu, "An effective GPU implementation of breadth-first search," in *Proc. 47th Design Autom. Conf. (DAC)*, 2010, pp. 52–55.
[42] E. Wynters, "Constructing shortest path maps in parallel on GPUs," in *Proc. 28th Annu. Spring Conf. PA Comput. Inf. Sci. Educators*, 2013.
[43] M. Kapadia, F. Garcia, C. D. Boatright, and N. I. Badler, "Dynamic search on the GPU," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Nov. 2013, pp. 3332–3337.
[44] F. M. Garcia, M. Kapadia, and N. I. Badler, "GPU-based dynamic search on adaptive resolution grids," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2014, pp. 1631–1638.
[45] E. Mérida-Casermeiro, J. Muñoz-Pérez, and R. Benítez-Rochel, "Neural implementation of Dijkstra's algorithm," in *Proc. Int. Work-Conf. Artif. Neural Netw.*, 2003, pp. 342–349.
[46] J. B. Aimone, O. Parekh, C. A. Phillips, A. Pinar, W. Severa, and H. Xu, "Dynamic programming with spiking neural computing," in *Proc. Int. Conf. Neuromorphic Syst.*, Jul. 2019, pp. 1–9.
[47] J. C. Elizondo-Leal, E. F. Parra-González, and J. G. Ramírez-Torres, "The exact Euclidean distance transform: A new algorithm for universal path planning," *Int. J. Adv. Robotic Syst.*, vol. 10, no. 6, p. 266, Jun. 2013.
[48] H. M. Choset, S. Hutchinson, K. M. Lynch, G. Kantor, W. Burgard, L. E. Kavraki, and S. Thrun, *Principles of Robot Motion: Theory, Algorithms, and Implementation*. Cambridge, MA, USA: MIT Press, 2005.
[49] L. Wenzheng, L. Junjun, and Y. Shunli, "An improved Dijkstra's algorithm for shortest path planning on 2D grid maps," in *Proc. IEEE 9th Int. Conf. Electron. Inf. Emergency Commun. (ICEIEC)*, Jul. 2019, pp. 438–441.
[50] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, vol. 135. Cambridge, MA, USA: MIT Press, 1998.

Tomas Kulvicius received the Ph.D. degree in computer science from the University of Göttingen, Göttingen, Germany, in 2010. In his Ph.D. thesis, he investigated the development of receptive fields in closed-loop learning systems.

From 2010 to 2015, he was a Researcher at the University of Göttingen, where he worked on trajectory generation and motion control for robotic manipulators. From 2015 to 2017, he was appointed as an Assistant Professor at the Centre for Bio Robotics, University of Southern Denmark, Odense, Denmark. He is currently a Research Assistant at the University of Göttingen. His research interests include modeling of closed-loop behavioral systems, robotics, artificial intelligence, machine learning algorithms, movement generation and trajectory planning, action recognition and prediction, movement analysis, and diagnostics.

Sebastian Herzog studied computational neuroscience and mathematics at the University of Göttingen, Göttingen, Germany.

His current research interests include robotics, artificial intelligence, machine learning, nonlinear dynamics, self-organization, computational fluid dynamics, quantum mechanics, information theory, and data assimilation.

Minija Tamosiunaite received the Ph.D. degree in informatics from Vytautas Magnus University, Kaunas, Lithuania, in 1997.

She currently works as a Senior Researcher at the Bernstein Center for Computational Neuroscience, Third Institute of Physics, University of Göttingen, Göttingen, Germany. Her research interests include machine learning, biological signal analysis, and application of learning methods in robotics.

Florentin Wörgötter has studied biology and mathematics at the University of Düsseldorf, Düsseldorf, Germany. He received the Ph.D. degree from the University of Duisburg–Essen, Duisburg, Germany, in 1988, for work on the visual cortex.

From 1988 to 1990, he was engaged in computational studies at the California Institute of Technology, Pasadena, CA, USA. From 1990 to 2000, he was a Researcher at the University of Bochum, Bochum, Germany, where he was investigating the experimental and computational neuroscience of the visual system. From 2000 to 2005, he was a Professor of computational neuroscience with the Department of Psychology, University of Stirling, Stirling, U.K., where his interests strongly turned toward learning in neurons. Since July 2005, he has been the Head of the Bernstein Center for Computational Neuroscience, Department of Computational Neuroscience, Third Institute of Physics, University of Göttingen, Göttingen, Germany. His current research interests include information processing in closed-loop perception-action systems, sensory processing, motor control, and learning/plasticity, which are tested in different robotic implementations.