ALGORITHMIC IMPROVEMENT FOR SOLVING MARKOV DECISION PROCESSES ON SCALABLE LARGE GRIDS

SHANNON LIN [SHANNON1@SEAS.UPENN.EDU], SOFIA PANAGOPOULOU [SOFIAPAN@SEAS.UPENN.EDU], ORESTIS SKOUTELLAS [ORESTIS@SEAS.UPENN.EDU],

ABSTRACT. The current paper presents three different approaches to speedup the Markov Decision Process when having larger scaled grids: Breadth-First Search (BFS); Divide, Conquer, Combine (DCC); and Reduced Resolution (RR). The performances of all three methods are theoretically analyzed and experimentally tested. We find that we can, in fact, significantly reduce the algorithm runtime and actual execution time while maintaining acceptable accuracy.

1. INTRODUCTION

1.1. **Project Goal.** The problem we are trying to solve is algorithmically speeding up the Markov Decision Process (MDP) while maintaining high accuracy. This is a non-trivial, meaningful endeavor because of the vast applications in robotics today and beyond. The real-world grids are usually large areas (at least room-sized) divided into millions of cells that are analyzed and determine the robot's actions. Running the MDP algorithm without having a specialized RAM is an extremely time-consuming process. Our goal is to make MDP more easily usable in the robotics sector with 3 new approaches.

1.2. **Contributions.**

Metric	Python MDP Toolbox	BFS	Divide, Conquer, Combine	Reduced Resolution
Expected Runtime Speedup	1x	400x	4x	16x
Actual Runtime Speedup	1x	20x	3x	15x
Accuracy	100%	15%	73%	47%

For each algorithm discussed, we evaluate two metrics: runtime and accuracy. Runtime encompasses both theoretical big-Oh runtime and actual execution time. Accuracy is measured against the Python library MDP toolbox where accuracy is the proportion of non-sink cells with the same policy as MDP Python toolbox policy.

2. BACKGROUND

Discounted MDPs (DMDP) are described by the tuple (S, A, P, r,), where S is the finite state space, A is the finite action space, and $0 <= \gamma <= 1$ is the discount factor. P is the collection of state-action-state transition probabilities, with each $p_a(i, j)$ specifying the probability of going to state j from state i when taking action a. r is the collection of rewards at different state-action pairs: we collect $r_a(i)$ if we are currently in state i and take action a. At each time step t, a controller takes an available action a ϵ A from their current state i and reaches the next state j with probability $p_a(i, j)$. For each action a taken, the decision maker earns an immediate reward, $r_a(i)$. The main goal in solving a DMDP is to find a deterministic policy π^* that maximizes the overall expected discounted cumulative reward.

In this paper, we summarize existing published work and implement three self-designed methods ourselves. The environment is a 2D grid cell with a perimeter 1-cell thick consisting of obstacles and 10% obstacles inside it with $\gamma = 0.9$. Once the agent reaches a sink cell, it cannot move out of it. Sink cells include all obstacle cells and the target cell. All states have reward of -1 (to incorporate the inherent cost of surviving in the environment) except for the target cell with a reward of 10 and obstacle cells with a reward of -10. Each state (s) has 4 actions (a): north, east, south, west. When the agent performs an action, it actually moves in that direction with probability 0.7, stays in the same cell with probability 0.1, and moves in the 2 other directions that is not its rival movement with probability of 0.1 each. For example, if an agent attempts to move North, it will actually do so with probability 0.7 and stay in the same cell/move East/move West with probability 0.1 (there is no chance of it going South, which is North's rival movement). The rest of the paper is organized as follows: related work, our 3 approaches with runtime/implementation/discussion, and conclusion with next steps.

3. Related Work

There are various approaches to speeding up value iteration that have been published in the past, 3 of which we summarize below. Note that not every cell is updated due to limited information given in the paper.

Metric	Toolbox	Geist & Scherrer [1]	Ho (homotopy) [2]	Ho (bisection) [2]	Sidford[3]
Runtime	$O(as^2)$	significant speed up	$O(sa \cdot logs)$	$O(sa \cdot log(sa)$	$O((\lvert S\rvert^2\lvert A\rvert + \frac{\lvert S\rvert\lvert A\rvert}{(1-\gamma)^3})log(\frac{M}{\epsilon})log(\frac{1}{\delta}))$
Accuracy	100%	N/A	100%	100%	approximate

Geist & Scherrer (2018) [1] apply the Anderson acceleration technique towards value iteration which is a method that allows speeding up the computation of fixed points. This is related to value iteration because the optimal value is the fixed point of the nonlinear Bellman operator. While the paper cites significant speed up of convergence (so assuming accuracy is maintained at 100%), it doesn't generalize into a Big-Oh runtime. Additionally, it ran experimental results on Garnet problems which are a class of randomly built MDPs with only 100 states, 4 actions, and branching factor (b) of 4. b different next states are chosen randomly and the associated probabilities are set by randomly partitioning the unit interval. Note that our approaches explore beyond 100 states.

Ho et al (2018) [2] contribute 2 efficient and exact algorithms for computing Bellman updates that significantly improve runtime for solving robust MDPs which assume that the transition probabilities and/or rewards are uncertain and can take on any plausible value from a so-called ambiguity set. The first is the homotopy approach which starts from the worst response then traces it back to nature's response with the increasing ambiguity set size. The second is the bisection approach (on top of homotopy) that uses s-rectangular ambiguity sets that assume a weaker nature that must commit to a realization of transition probabilities before observing the decision-maker's actions.

Sidford et al (2019) [3] develop a class of new value iteration-based algorithms that employ variance reduction to achieve improved running times for solving discounted MDPs. Their algorithms leverage further insights to ensure that our algorithms make monotonic progress towards the optimal value. Overall, they provide the first nearly linear convergent, nearly linear time algorithms that can solve MDPs with high precision even when the discount factor depends polynomially (with a small exponent) on the number of states and actions.

4. Approach

We discuss 3 of our approaches below with BFS as the weakest baseline, Divide/Conquer/Combine as the best option for increasing execution time while maintaining high accuracy, and Reduced Resolution as the best option for speeding up exectuion time without much heed to accuracy.

4.1. **Approach: Shortest path with Breadth-First Search (BFS).**

(1) Compute shortest path from each cell to the target cell using BFS.
(2) Ideal policy from each cell is the policy used to achieve the shortest path.

4.1.1. *Reasoning.* Intuitively, this approach will not achieve a high accuracy because BFS doesn't take into account uncertain actions, ones that don't actually produce the control that it was given. However, we utilized this approach as a baseline first approach because of the runtime improvement.

4.1.2. *Runtime.* The runtime of BFS is O(m+n) where m is the number of edges and n the number of nodes. In this problem, m=a*s and n = s so the runtime is O(as+s).

4.1.3. *Implementation.* This approach is solved by the classical BFS implementation with a double ended queue (deque). BFS involves traversing all the nodes in a given layer then moving to the next layer. We can easily track the distance from the target cell as each node in the same layer has the same distance. The policy is recorded simultaneously by tracking the relative motion from the current node to the next node within the shortest path.

4.2. **Approach: Divide, Conquer, Combine (DCC).**

(1) **Divide** in M equal-sized grids, by splitting each grid axis by S.
(2) **Conquer** by executing MDP in each of the M sub-grids, starting from the one with the reward, then moving to the adjacent ones progressing gradually to all.
(3) **Combine** the M sub-grid solutions and concatenate them to form the original big grid.

4.2.1. *Reasoning.* Inspiration was drawn from the reduction in sorting runtime using mergesort, which splits up progressively in two until only one node is left. Why wouldn't the same apply here, to an extent, since as long as you can portray the reward accurately and you start from the reward, we should be fairly accurate on our projection. We did not break down more than one level in splitting, for simplicity sake, yet it's worth exploring such aspect in later discussion.

4.2.2. *Runtime.* By splitting each axis by S are creating M new maps, where $M = S^2$ in our implementation. Each sub-grid has $\frac{1}{S}$ the length of the original one, that is $N' = \frac{N}{S}$. Thus, our updated runtime is $O(Ma(N')^4) = O(Ma\frac{N^4}{S^4}) = O(S^2\frac{1}{S^4}aN^4) = O(S^2aN^4) = O(MaN^4)$ showing us that we'll speedup our runtime by a factor equal to the amount of sub-maps we create. Let us demonstrate that with one simple example: If our original grid is $(20, 20)$ with 4 actions, that is $N = 20$ and $a = 4$, then it's original runtime will be $a * N^4 = 640,000$. With our approach, we divide it in 4 sub-grids with size $(10, 10)$, that is $M = 4$ and $N' = 10$, thus producing a runtime of $M * a * (N')^4 = 160,000$. This is a 4x speedup of runtime.

Therefore, we expect an approximate Mx speedup using this approach.

4.2.3. *Implementation.* For simplicity, we used $S = 2$, meaning splitting an (N, N) grid in $M = 4$ sub-grids, each $(\frac{N}{2}, \frac{N}{2})$. Running MDP in the sub-grid where the reward is, is pretty straightforward. However, for the 2nd, 3rd and 4th sub-grid, there is no reward, so we need to create one. We do that by appending the adjacent 1st grid's reward output row or column normalized to the adjacent 2 grids (see Appendix Figure 8). We run MDP on these two and then take their adjacent reward output normalized and place it on the last 4th sub-grid to be able to run MDP on the last one. This methodology is critical (and can also be further improved), so that we accurately portray where the reward is higher on the walls from grid to grid. We also use sufficient padding to create artificial walls which we then remove.

4.3. **Approach: Reduced resolution.**

4.3.1. *Reasoning.* As we are aiming for a significant runtime decrease, another possible trade-off is the grid resolution in the MDP algorithm. Below, specific details of this study are provided.

4.3.2. *Runtime.* By reducing the resolution of the grid by C, we cut up the grid in (C, C) sub-grids which are now one cell each (using average or other methodology to set a value). The new lower-resolution grid has $\frac{1}{S}$ the length of the original one, that is $N' = \frac{N}{C}$. Thus, our updated runtime is $O(a(N')^4) = O(a\frac{N^4}{C^4})$ showing us that we'll speedup our runtime by a factor equal to the amount of sub-maps "quartic-ed" we create. Let us demonstrate that with one simple example: If our original grid is $(20, 20)$ with 4 actions, that is $N = 20$ and $a = 4$, then it's original runtime will be $a * N^4 = 640,000$. With our approach, we transform every 4 cells into 1 and end up with a grid of size $(10, 10)$, that is $N' = 10$, thus producing a runtime of $a * (N')^4 = 40,000$. This is a 16x (which is 2^4) speedup of runtime.

Therefore, we expect an approximate C^4x speedup using this approach.

4.3.3. *Implementation.* For simplicity, we used $C = 4$, meaning creating groups of 4 cells each, resulting in a new map of size $(N/2, N/2)$. The reward of each cell is calculated by taking the average of the initial 4 cells. Based on this number, we determine whether the new cell is an obstacle (reward < -5 meaning 2 or more of the 4 initial cells were obstacles), the target (if one of the 4 initial cells was the initial target), or an open accessible cell (all other numbers). Additionally, we add walls to the new grid to create a standalone environment (see Appendix Figure 7). The rest of the process matches our reference MDP code, including the generation of the probability and transition matrices, as well as running the MDP algorithm through the python toolbox. Finally, we expand the resulting matrices to (N, N) dimensions to be able to compare them with the reference results.

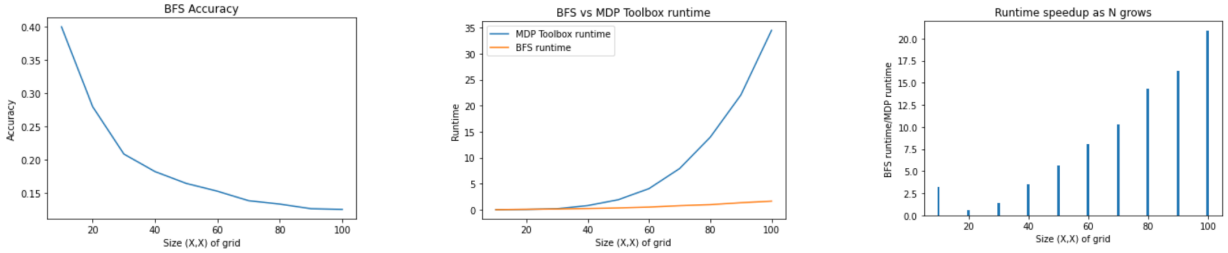
4.4. **Success Metric: Policy Accuracy.**

Our chosen method of assessing success is the comparison of the resulting policy matrices. In all approaches, we utilize Python's out of box MDP library solution as the gold standard comparison. Our metric is the proportion of non-obstacle cells suggesting the same policy (action).

5. Experimental Results

Each metric figure (accuracy/runtime) are the result of averaging 5 iterations for each grid size to avoid any outlier results due to randomness.

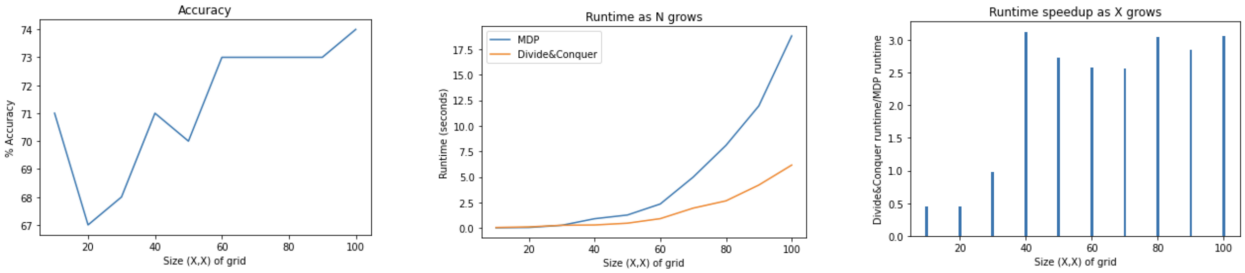
5.1. **Approach: Breadth-First Search BFS.** In alignment with expectations, BFS generates solutions much faster than the Python MDP toolbox solution with lower accuracy as seen in Figure 1 below. Generally, the larger the grid, the larger the difference in execution time which supports the goal of this project to implement faster algorithms to solve MDP. However, at best, BFS is not even half accurate and the accuracy decreases as the size of the grid increases. These empirical results support our hypotheses and we move next to highlight promising results from the next 2 approaches.



(A) Accuracy (B) Runtime (C) Speedup

FIGURE 1. Benchmark comparison and speedup with BFS

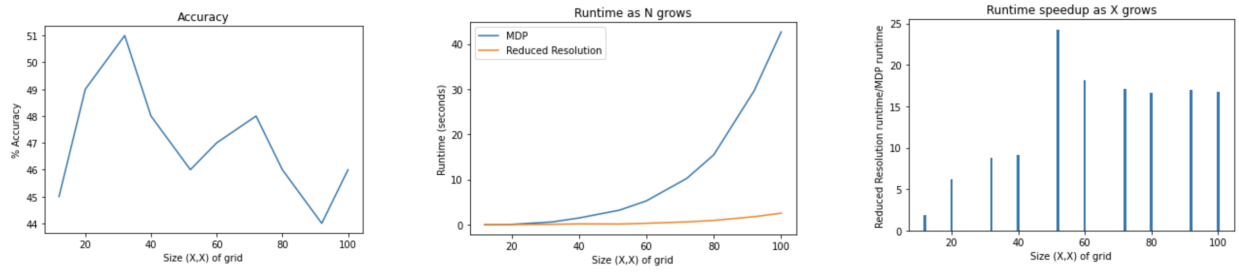
5.2. **Approach: Divide, Conquer, Combine (DCC).** Despite the same grid size limitations as mentioned above (due to Google Colaboratory RAM capacity when running the MDP Python library toolbox solution), our results are notable. First, we notice that the accuracy is robust (see Figure 2 below): consistently above 70%. While our approach's benefit is not visible in smaller $(20, 20)$ grids, as N grows, we find an average speedup of 3x (compared to the expected 4x). This validates our hypothesis that we can speed up MDP significantly.



(A) Accuracy (B) Runtime (C) Speedup

FIGURE 2. Benchmark comparison and speedup with Divide, Conquer, Combine

5.3. **Approach: Reduced Resolution.** First, note that the accuracy doesn't converge to a value as size increases, so no general conclusions can be drawn. However, the Figure 3 below shows significant improvement of up to 25x speedup as N grows, beating the theoretical calculation of 16x.



(A) Accuracy (B) Runtime (C) Speedup

FIGURE 3. Benchmark comparison and speedup with Reduced Resolution

6. DISCUSSION

The **BFS** approach matched all expectations. No further work would be spent on this approach except to test on different map edge cases. However, we do not expect much improvement in accuracy even in the best case scenario and continue to expect fast execution time even in the worst case scenario because of the nature of BFS in solving simply the shortest path without heed of any probabilities.

Extensions of **Divide, Conquer, Combine approach** analysis can include dividing into more layers (like our inspiration of mergesort) and measuring the effect of splitting in three or in four, instead of just splitting in half as we do now.

Extension of **Reduced Resolution approach** analysis can include reducing resolution even further and measuring the impact on accuracy and effect of the % of blockage elements inside the grid since all of our grid map tests are with 10% obstacles within the perimeter. We could also develop a specific equation calculating the ideal number of grouped cells based on the size of the initial environment, so that the trade-off between accuracy and resolution is optimal. Finally, more robust and enlightening results could be highlighted with significantly larger grids.

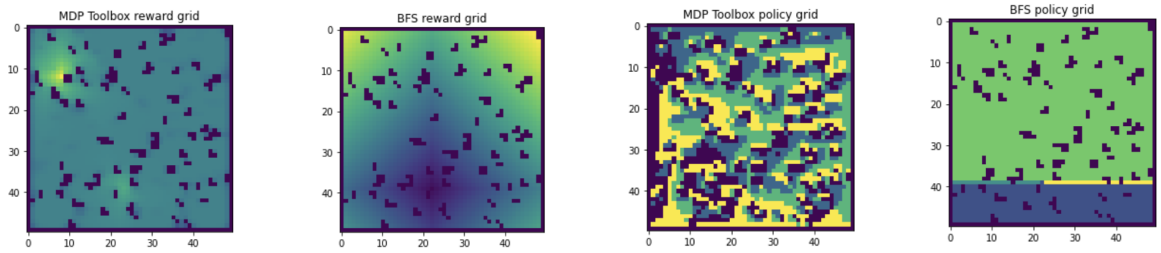
Other approaches, except the three ideated during our initial brainstorming, can include a combination of existing previous (see Background Section) algorithms and ours. Bringing the best or the averages of two or more different approaches and creating an ensemble model would potentially be optimal to reducing the runtime of MDP.

Another extension would potentially be using a GPU or higher RAM capacity to scale up even further to $(1000x1000)$ or larger grids, since the maximum we could reach using Google Collaboratory Pro with 12GB or RAM was $(100, 100)$. Given more time, we would have attempted to replicate the competing existing literature approaches and identify clear pros and cons to each approach, as well as ours. There appears to be no common way shared across papers to identify accuracy or error, which we found quite surprising to be able to measure one against the other. One of our notable contribution to the existing literature is straight-forward benchmarking and mentioning the tradeoff between accuracy and runtime speedup savings.

7. CONCLUSION

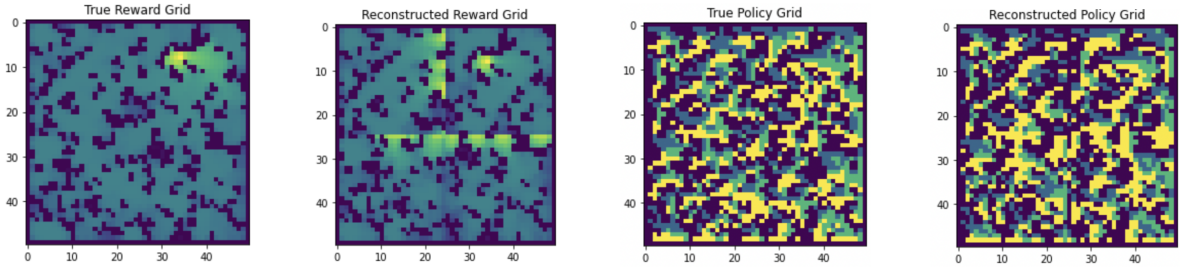
Our unique approaches show promising results towards more quickly solving large-scale Markov Decision Processes with high accuracy. This paper discussed in-depth implementation, evaluation, and limitations with next steps. We see a strong future with these approaches that are easily implemented and referenced below.

8. APPENDIX



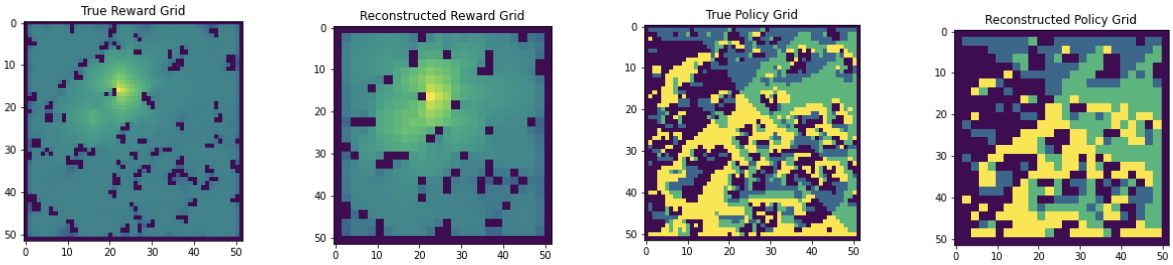
(A) Reference (B) BFS (C) Reference (D) BFS

FIGURE 4. Policies and Reward grid result of 50x50 with BFS



(A) Reference (B) DCC (C) Reference (D) DCC

FIGURE 5. Policies and Reward grid result of 50x50 with Divide, Conquer, Combine



(A) Reference (B) RR (C) Reference (D) RR

FIGURE 6. Policies and Reward grid result of 52x52 with Reduced Resolution

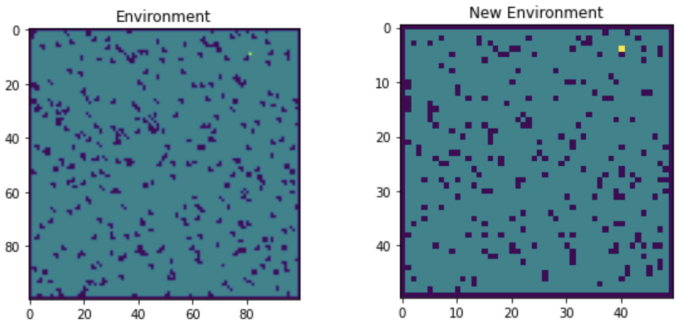
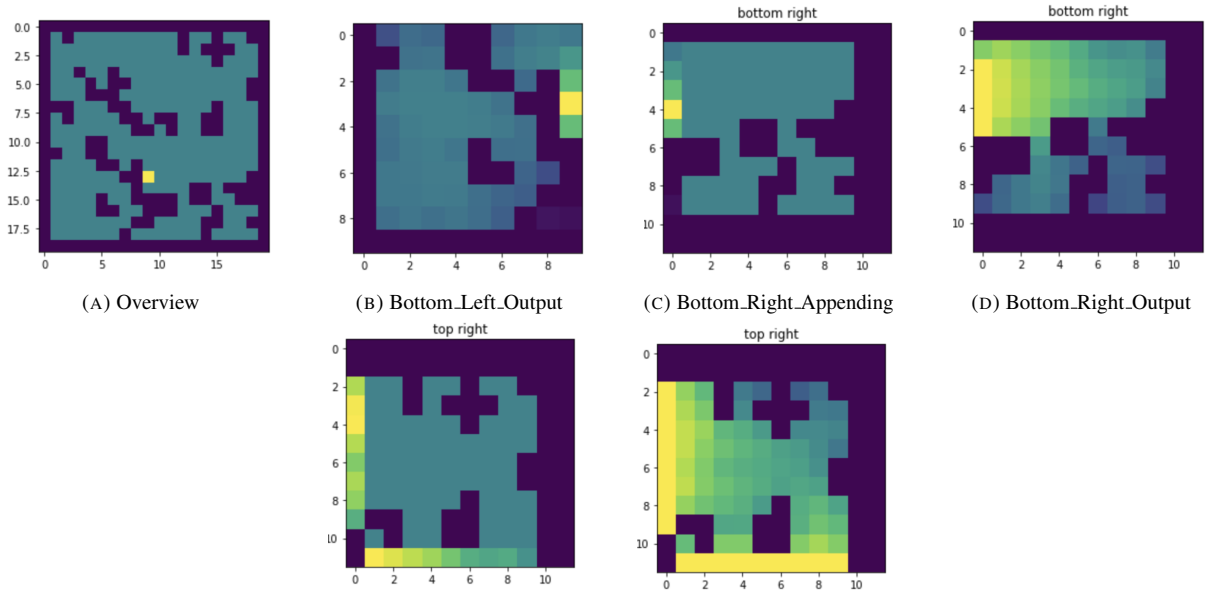


FIGURE 7. Reducing the resolution of the environment



(A) Overview (B) Bottom_Left_Output (C) Bottom_Right_Appending (D) Bottom_Right_Output

(E) Top_Right_Appending (F) Top_Right_Output

FIGURE 8. Implementation Details of Divide, Conquer, Combine

BFS Implementation
Divide, Conquer, Combine Implementation
Reduced Resolution Implementation

REFERENCES

[1] Matthieu Geist and Bruno Scherrer. Anderson acceleration for reinforcement learning, 2018.
[2] Chin Pang Ho, Marek Petrik, and Wolfram Wiesemann. Fast Bellman updates for robust MDPs. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1979–1988. PMLR, 10–15 Jul 2018.
[3] Aaron Sidford, Mengdi Wang, Xian Wu, and Yinyu Ye. Variance reduced value iteration and faster algorithms for solving markov decision processes, 2017.