# DMU Fog-Of-War Chess

Aaron Gindi, Branson Camp, Aidan Luczkow

## I. Introduction

Chess has been a problem domain that interests computing researchers since at least Charles Babbage in the 19th Century. DeepBlue and Alpha Zero more recently proved that it is an area applicable to many cutting edge techniques. Chess is a perfect information turn taking game with simple rules and great strategic depth, and so it is often held up as a benchmark for intelligence. Fog-of-War Chess (or Dark Chess) is not a perfect information game; the full state of the board is not known to either player. Ordinary techniques for turn taking games are much less applicable, leaving the game ripe for techniques that help the players make decisions under uncertainty.

## II. Problem Formulation

In our proposal, we initially conceived of this problem as a POMG. However, the complexity of this formulation proved untenable. Instead, we decided to consider a POMDP from one player's perspective. Gmytrasiewicz and Doshi claim that this approach is only justified "under the strong assumption that the other agent's behavior [can] be adequately represented implicitly within the state transition function" [1], which is a standard that we do not meet. The opposing player should be treated as a separate reasoning agent with it's own beliefs about the state. Instead, this simplification arises from necessity and time constraints. Formally, we represent Dark Chess by $< \gamma, \mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{Z}, \mathcal{T}, \mathcal{R} >$ where:

- $\gamma := 0.99$

- $\mathcal{S} :=$ the space of all possible chess positions, with a minimal number of indicators required to determine the legality of the moves - which player is to move, which player has castling rights in either direction, and an accounting of the total number of pieces the opponent still has on the board.

- $\mathcal{A} :=$ the space of all possible Dark Chess moves, which is a larger set than standard chess moves, as many previously illegal moves are now legal. Players may now move into check, castle through check, violate absolute pins - pointing out that any of these moves are illegal would leak information that a player is not supposed to be able to observe.

- $\mathcal{O} :=$ the space of all possible Dark Chess observations, which is a chess position with many squares unknown to a player.

- $\mathcal{Z} :=$ the observation probabilities, which in this case are a deterministic function. The observation that a player receives is the union of the locations of every piece that player has with the set of all possible destination squares for all of those pieces' moves.

- $\mathcal{T} :=$ the transition probability given the current state and both player actions. Similar to ordinary chess, it is a deterministic function. Dark Chess has an almost identical transition function to ordinary chess, with a few small tweaks to accommodate the slightly larger action space.

- $\mathcal{R} :=$ the reward function, +1000 if the player captures the opponent's king, -1000 if the opponent captures the player's king, and 0 in all other cases

This is an extremely rich problem, with great opportunities to explore tradeoffs in risky information gathering. In order to explore unknown parts of the state, the player needs to weigh the benefits of gained knowledge with the potential that uncovering those squares may put their resources in harm's way. The game cannot be won without exploration - victory is only achieved when the opposing king is found and subsequently captured. Therefore, any agent must balance optimism and paranoia when strategizing and making plans.

As we worked through this formulation of Dark Chess, we encountered some shortcomings. First, the state space is inadequate to fully represent the nuance of a human understanding of a position. If our player observes that their opponent had a bishop on a dark square, then it retreats into the fog, a human can remember that the bishop will always remain on that colored square. However, that information cannot be included in our state, and so the player, due to the imperfections of particle injection, will forget which color square the bishop was on. Another issue is the astonishingly large state space size. We attempted some approximate offline solutions for our POMDP formulation, and even with vastly simplified state spaces the process was resource intensive.

## III. Solution Approach

### A. Fog-Of-War Chess Interface

As chess is an incredibly popular game, there are many existing interfaces that handle the game programmatically. These interfaces include functions that keep track of the board state, allow moves to be made by each player, generate legal moves for each player, determine

when a king is in check/checkmate, and many other necessities to play the game. Unfortunately, the Fog-Of-War chess variant (or Dark Chess), does not have an existing interface. For this reason, one of the existing chess interfaces had to be modified to this variant of the game before any further work could be completed. The Python library named "python-chess" was selected as the base interface for the dark chess variant. As discussed earlier, there are many moves that are illegal in standard chess which are completely legal in dark chess. To account for these moves, the python-chess function for generating legal moves had to be modified to include all moves that may put the king into check, as well as moves that involve capturing the opponents king. Additionally, checkmate does not exist in dark chess, so this function had to be overwritten as well. Finally, modifications were made to allow for variable board sizes and to generate board visualizations that only showed the squares visible to each respective player.

### *B. QMDP Solver*

In order to generate the pseudo-alpha vectors necessary for a QMDP solution to the dark chess problem, every state and action must be enumerated. In this context, a state is considered to be a position of pieces on the board, disregarding extraneous information such as which player is making the current move, castling rights, and en passant captures. For this reason, the state can be represented as the board part of the FEN string. Actions are considered to be a move from one board square to another board square, ignoring which piece is making the move. In regard to enumerating these two aspects of the game, the actions are the simpler of the two.

In order to enumerate the actions, the algorithm iterates over every possible pair of starting and ending squares, and checks if moving from the starting square to the ending square is possible with one of the pieces available to the player. Since a player will always have a king, moves to adjacent squares are always accepted. Since pawns can only ever move to an adjacent square (disregarding the first move), this also covers all possible pawn moves. If the player has a rook, then the move is accepted if the starting and ending squares are on the same rank or file. If the player has a bishop, then the move is accepted if the starting and ending squares are on the same diagonal. If the player has a queen, then moves are accepted in both of these cases. Finally, if the player has a knight, then the move is accepted if the squares are separated by two squares in one direction and one square in the other. As the moves are being iterated over, a counter keeps track of the index of the current move. When a move is accepted, the move and index are stored as key-value pair in a dictionary, and the counter is increased. In this manner, every move can be efficiently mapped to its unique index.

Enumerating the possible board states requires a more complex, recursive algorithm. The algorithm takes a list of all of the white and black pieces and the board size as inputs. On the first call, the board is initialized to be empty, and the state index is initialized to zero. It then iterates over every available piece in the list of pieces, and every square on the board. If the square is not already occupied by another piece, it places the next piece in the piece list on that square. It then checks if the position has not yet been recorded and if there is at least one king on the board. If these two conditions are met, the algorithm adds the position as a key to a dictionary with the index as its value, updates the board state to include the new piece, and increments the index. If not, the position is not added to the dictionary. The algorithm then checks that there is at least one king on the board or in the list of remaining pieces. If so, the piece that was just placed on the board is removed from the list of pieces, and the function recursively calls itself with the updated board state and list of pieces. If not, the function returns the current state-index dictionary and the current index. In this manner, every possible arrangement of pieces with at least one king on the board is assigned a unique index. Any game state can be efficiently mapped to its index using the state-index dictionary.

After the states and actions have been enumerated, the final step needed before running the QMDP algorithm is to generate the state transition function. This function is modeled using a dictionary that maps tuples of the form (initial state, action, final state) to a probability between zero and one, such that the sum over all possible final states given an initial state and action is equal to one. First, the algorithm iterates over every possible initial state. For each state, a move-state list dictionary is initialized that map each of the player's moves to a list of all possible final states given that move. It then iterates over all possible legal moves for the player starting from the initial state. The state of the board is updated to account for the current move. If the game ended after this move (e.g. the player captured the opponents king), then the dictionary entry for the current move is set to only include this final state, as no other states can possibly be reached. If the game does not end after this move, the algorithm then iterates over all of the legal moves for the opponent. It then updates the board state to account for the opponents move, and appends the final state to the dictionary entry for the players move. After repeating this process for every move that the player can make from the initial state, a dictionary has been created that maps a legal player move to a list of all of the possible final board states. Now, in order to generate the probability distributions, the algorithm must iterate

over the list of legal player moves again. For each move, the dictionary that was generated in the previous step is used to extract the list of possible final states. In order to determine the probability of each one of these states, the Stockfish chess engine is used to evaluate the value of the position for the opponent. If the state has a high evaluation for the opponent, the opponent would be more likely to make the move that transitions to that state. Therefore, the probability over each of the final states is proportional to the board evaluation for the opponent; however, every final state is assigned at least some non-zero probability value, in order to account for cases where the opponent plays sub-optimally. The probability over all final states is normalized so that the sum is equal to one, and then the (initial state, action, final state) tuple is added as a key to the state transition dictionary with the probability of the final state as its value. Once this process is repeated for all initial states, the state transition dictionary is complete.

After all of the states and actions are enumerated and the state transition dictionary is complete, the QMDP algorithm is ready to run. This process follows the standard QMDP algorithm very closely, with only slight modifications for efficiency given that almost every move is illegal from a given state. First, a pseudo-alpha vector (from here forward referred to simply as alpha vectors) is generated for every move in the list of enumerated moves, with an entry of zero for every state in the list of enumerated states. The algorithm then iterates over every move in the move list and every state in the state list. Given the state, a list of legal player moves is generated. If the current move is not one of the legal moves from the current state, the algorithm sets the alpha vector value for the current move-state pair to a very low value (in this case, -10,000 was selected). In the rare case that the move is legal from the current state, the algorithm updates the board state to reflect the current move. If the game ends after the move is made, then the alpha vector value for the current action-state pair is set to a high reward (1000 in this specific implementation), as the player would win the game by making this move in this state. If the game did not end, an expected reward value is initialized to zero, and the algorithm generates a list of all possible moves that the opponent can make. Since moves are deterministic, this is equivalent to iterating over the possible next states given the player's first move. If the game ends after the opponent's move is played (i.e. the opponent wins the game) then the "maximum alpha vector" value is set to a very low reward (in this case, -1000); however, if the game does not end, all of the previous alpha vectors are iterated over to determine the true maximum alpha vector value at the next state. The maximum alpha vector value is then multiplied by the probability of transitioning to the next state (given by the state transition dictionary) and added to the expected reward value. Finally, once all of the possible next states have been accounted for, the alpha vector for the current move at the index of the current state is set equal to the expected reward multiplied by a discount factor (in this case 0.99 was used). No reward is added to the value, as rewards are only assigned if the game is won or lost, in order to avoid reward shaping. Once every player move has been iterated over, a new set of alpha vectors has been generated, and the process repeats with the new set of alpha vectors. The entire process repeats for a predetermined number of iterations, and then the set of alpha vectors is returned. With the alpha vectors generated, the best move from a given belief state can be determined by taking the dot product of the belief state with each alpha vector. The alpha vector that yields the maximum dot product will correspond with the best move.

### *C. Particle Filter for Belief Updates*

In order to use QMDP effectively, we need an accurate belief of which state(s) we are most likely in. In a standard chess game, you wouldn't need a belief because you can see the full state. In Fog of War chess, your observations are limited: you can only see where your own pieces are and the squares within each of your pieces' line of sight (where they can legally move). We felt that a rejection particle filter was the best option for updating beliefs for Fog of War chess. Since the initial state is known to both players, the particle filter starts by creating $N$ particles set to the initial board state. For each subsequent full turn (after the player and enemy makes a move), the particle filter updates in four steps (note: the steps describe white's particle filter but can be easily modified for black):

1) **Advance Particles**: Each particle attempts to make white's move. Then, each particle makes a random move for black.
2) **Optional Particle Generation**: If the particle filter has no particles left or has below an acceptable threshold, regenerate up to $N$ particles (see section C).
3) **Particle Rejection**: Each particle that doesn't match the current observation is discarded. For example, if white sees a black bishop on B2, every particle without a black bishop on B2 is discarded.
4) **Extract a Belief**: A count is assigned to each state based on how many particles are in the state. The counts are then normalized to sum to 1 and are then returned as the new belief.

### *D. Particle Generation*

One of the largest problems facing particle filters, rejection particle filters in particular, is particle depletion

[2]. In this problem domain, resampling particles is incredibly challenging. The number of potential states is huge, growing exponentially with each time step. Furthermore, the observation that the player receives is likely to be the same over consecutive turns, leading to less confidence in any particular belief. Finally, when a particle is rejected, there may be a state that is very similar and critical (leading to a sudden negative reward) - this is one of the major perils surrounding particle depletion in this particular domain.

The particle generator is capable of generating particles in two vastly different ways. First, it is able to uniformly sample from the state space, generating chess positions that while perhaps not actually plausible are usually legal and are consistent with the most recent observation. The function ensures that pawns are placed in legal locations, but gives few other assurances. Such sampling from a uniform distribution over the state space "reduces the accuracy" of the beliefs represented by the particle filter [2].

The other sampling process is simulation based - it approximates the last known state and randomly transitions the opponent's pieces to a place where they are consistent with the current observation. This may mistakenly reintroduce previously rejected particles, but states can be reached through many different sequences of actions in this domain. Those rejections may have introduced a blind spot, which this injection may correct.

Other hypothesized methods for sampling the state space involve conditioning on the observation history, i.e. more in depth and sophisticated simulation based sampling, and conditioning on the current belief, i.e. employing strategies similar to SIRPF [3]. These could not be pursued due to time constraints.

## IV. Results

### A. QMDP Solver

In order to demonstrate the effectiveness of the QMDP solver without an associated belief updater, games were simulated assuming that the player always had access to a perfect belief state. This translates to the QMDP algorithm playing classical chess, where the board is fully observable. The results of the QMDP algorithm with access to the full board state over 100 simulated games are presented in Table I, in the rows where the White Policy is set to "full-state". These games were played on a 4x4 board where the QMDP algorithm (White) has a king and a queen, while the opponent (Black) only has a king. Additionally, the games include the cases where the opponent always selects a random move (Black Policy = random), and when the opponent plays intelligent moves using the QMDP algorithm (Black Policy = full-state). In both cases, the QMDP algorithm playing the white pieces was able to win 100% of the time, verifying the algorithms efficacy in this simplified case.

In order to provide an example of more complicated play, a 4x4 game is shown below in Figures 1 through 9 where the QMDP algorithm plays the white pieces, and one of the authors plays the black pieces. The white pieces include a king and a queen, while the black pieces include a king and a bishop. In order to save space on the document, every image except for the initial and final states corresponds to a move by white and the subsequent move by black. The standard algebraic notation for the moves played by each player are included in the figure labels. By Figure 7, the QMDP algorithm has forced the black player to give up its bishop. In Figure 8, once the bishop has been removed, the black king has no choice except to move into a capture, loosing the game. This game demonstrates that the QMDP algorithm can exhibit more advanced playing behaviors over fully observable board states.

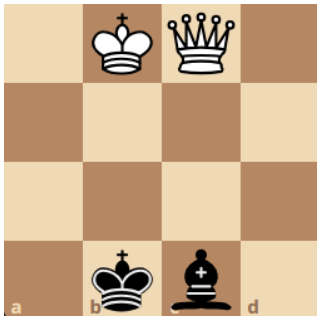


Fig. 1. Initial State

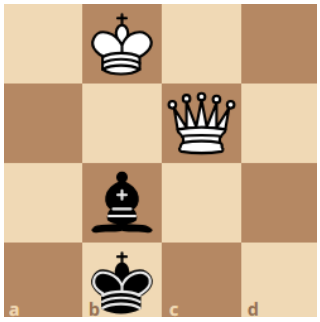


Fig. 2. White Move: Qc3, Black Move: bb2

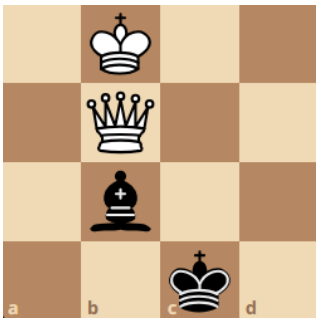

Fig. 3. White Move: Qb3, Black Move: kc1

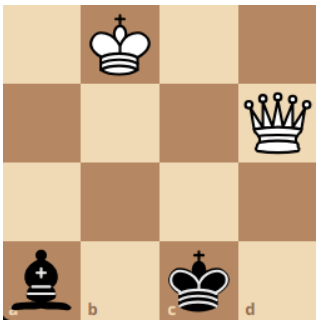

Fig. 4. White Move: Qd3, Black Move: ba1

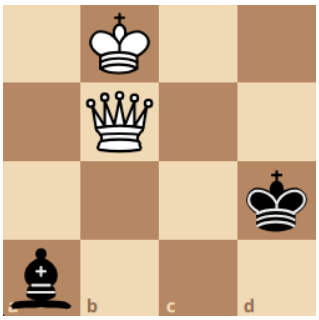

Fig. 5. White Move: Qb3, Black Move: kd2

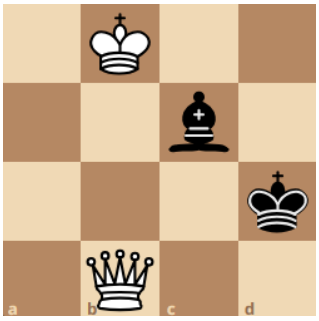

Fig. 6. White Move: Qc1, Black Move: bc3+

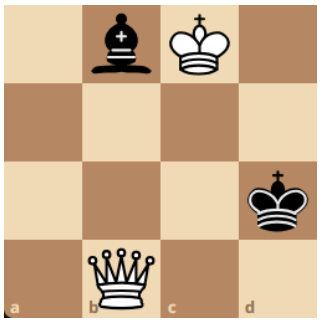

Fig. 7. White Move: Kc4, Black Move: bb4

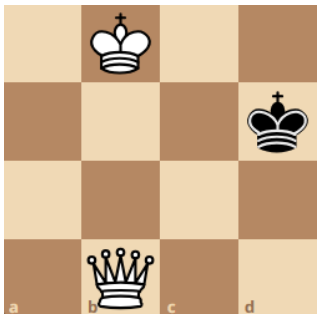

Fig. 8. White Move: Kb4, Black Move: kd3

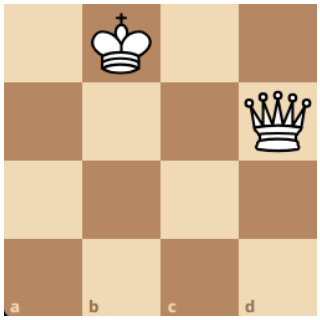

Fig. 9. White Move: Qd3++, Game Over

### *B. Particle Generation*

Particles for evaluation were generated ten at a time based on initial positions sampled randomly from ordinary chess games played online at lichess.org. Particle quality was gauged in both plausibility and evaluation difference. To measure plausibility, the domain specific tool Natch was employed. Natch is capable of determining with relatively high but not perfect accuracy whether or not a given chess position is reachable from the initial position within a given number of moves. The simulation or ”traced” particle generation has erratic plausibility due to observability challenges, but decreases generally over the number of moves. The uniform particle generation starts out wildly implausible but the plausibility increases slightly over time. Qualitatively, these particles are more

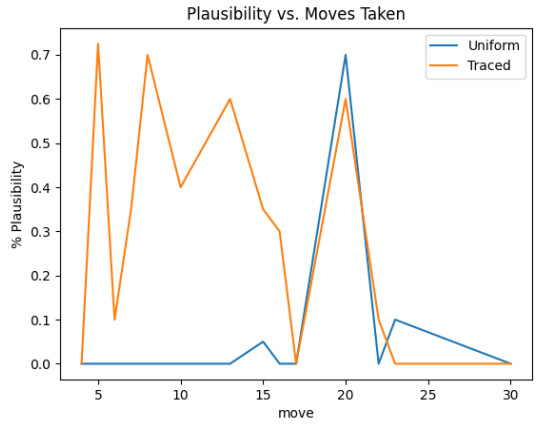


Fig. 10. Plausibility vs Moves Taken

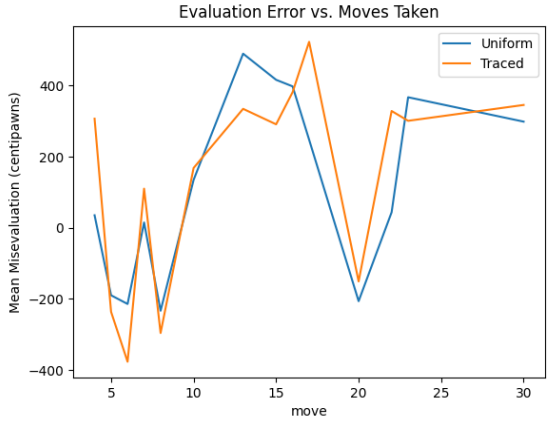


Fig. 11. Evaluation Error vs Moves Taken

plausible when the uncertainty in the observation of the initial state is small and a greater number moves allows for more unconventional positions to be reached.

Evaluation of each particle was performed using Stockfish, and compared to the evaluation of the initial position each particle was based on. The two methods performed remarkably similarly, both tending toward optimism, i.e. evaluating the position as better for whichever player's turn it is.

### C. Monte Carlo Simulations

While our QMDP solutions looked good after we manually played against it, we wanted a more robust way to quantify the effectiveness of these algorithms. In order to do this, we performed a series of Monte Carlo simulations where each simulation was done on 100 randomized initial 4x4 board states.

First, we investigated the simple case of a 4x4 board containing a white king, a white queen, and a black queen. In each game, the initial board state was randomized such that the black king was not initially in check. We simulated different combinations of policies for black and white: a random policy where the player picks a random move regardless of state, a full-state QMDP where the player is given a perfect observation every time, and the particle filter QMDP where the player gets a partial observation according to the rules of Fog of War chess. The results of the Monte Carlo are shown in Table I. This shows that both the full-state algorithm and particle filter QMDP were very successful, with both leading to white nearly always winning. Another Monte Carlo simulation was done on our more complex board, KQkb, where black also has a bishop. Again, for each game the board was randomized, placing each of the four pieces on the 4x4 board such that the black king wasn't in check. The results are shown in Table II. The extra bishop was no problem for white.

## V. Conclusion

Our findings suggest that QMDP with a rejection particle filter was able to reach near-optimal performance in a simplified 4x4 Fog of War chess board. While the particle filter is versatile to increasing the board size, QMDP struggles with this because calculating the optimal alpha vectors increases exponentially in time and space. In the future, we would like to combine our advanced particle filter with a tree-based method like MCTS to be able to tackle full-size Fog of War chess games.

## VI. Contributions and Release

Aidan implemented QMDP from scratch and wrote the sections discussing the QMDP implementation and results. Branson wrote the belief updater and particle filter. Aaron adapted the python_chess library and wrote a particle generator. The QMDP algorithm and particle filters were written from scratch in python without off-the-shelf libraries. Stockfish, an off-the-shelf chess engine, was used to help generate the transition function, $\mathcal{T}$. The off-the-shelf library Natch was used to verify plausible states in the advanced particle generator, which was otherwise written from scratch. The authors grant permission for this report to be posted publicly.

| Board | White Policy | Black Policy | White Wins | W Mean Turns | $\sigma$ Turns | Black Wins | B Mean Turns | $\sigma$ Turns | Draws |
|---|---|---|---|---|---|---|---|---|---|
| KQk | random | random | **59** | 9.508 | 9.294 | **41** | 10.171 | 9.013 | **0** |
| | particle-filter | random | **100** | 2.08 | 0.272 | **0** | 0 | 0 | **0** |
| | full-state | random | **100** | 2.11 | 0.373 | **0** | 0 | 0 | **0** |
| | random | full-state | **2** | 2.5 | 0.707 | **98** | 4.429 | 2.632 | **0** |
| | particle-filter | full-state | **97** | 2.773 | 1.906 | **2** | 23 | 19.79 | **1** |
| | full-state | full-state | **100** | 2.73 | 1.108 | **0** | 0 | 0 | **0** |

TABLE I

PERFORMANCE COMPARISON BETWEEN DIFFERENT WHITE AND BLACK POLICIES ON THE KQK BOARD.

| Board | White Policy | Black Policy | White Wins | W Mean Turns | $\sigma$ Turns | Black Wins | B Mean Turns | $\sigma$ Turns | Draws |
|---|---|---|---|---|---|---|---|---|---|
| KQkb | random | random | **54** | 10.685 | 8.529 | **46** | 9.022 | 8.538 | **0** |
| | particle-filter | random | **100** | 2.34 | 0.639 | **0** | 0 | 0 | **0** |
| | full-state | random | **100** | 2.38 | 0.663 | **0** | 0 | 0 | **0** |

TABLE II

PERFORMANCE COMPARISON BETWEEN DIFFERENT WHITE AND BLACK POLICIES ON THE KQKB BOARD.

## REFERENCES


[1] Doshi, P., & Gmytrasiewicz, P. J. "A particle filtering based approach to approximating interactive pomdps." AAAI. (2005).

[2] M. J. Kochenderfer, T. A. Wheeler, and K. H. Wray, Algorithms for Decision Making. MIT Press, (2022).

[3] Lee, J., Ahmed, N. R., Wray, K. H., & Sunberg, Z. N. "Rao-Blackwellized POMDP Planning." arXiv preprint arXiv:2409.16392 (2024).

[4] Manzo, A., Ciancarini, P. "Enhancing Stockfish: A Chess Engine Tailored for Training Human Players" (2023). In: Ciancarini, P., Di Iorio, A., Hlavacs, H., Poggi, F. (eds) Entertainment Computing – ICEC 2023. ICEC 2023. Lecture Notes in Computer Science, vol 14455. Springer, Singapore. https://doi.org/10.1007/978 − 981 − 99 − 8248 − 6_23.

[5] Li, X., Zhong, H., & Brandeau, M. L. (2022). Quantile Markov Decision Processes. Operations research, 70(3), 1428–1447. https://doi.org/10.1287/opre.2021.2123