Learning Generalized Policies Over Grid Environments

Abstract

1 Introduction

2 Logic Language

The Grid Domains share the set of Objects *O*, and the set of predicates *P*.

2.1 Objects

All objects in *O* belong to one sort, *cell*. Then, we have an object *o* for each cell in the grid. For instance, in a grid of size $i \times j$ the set *O* contains the object *cij*, representing the cell in the row *i* and the column *j*.

2.2 Predicates

To explain the predicates it is good to remember that each of the domains has different values contained in the grid, e.g. the domain reach-for-the-star has five possible values for each cell in the grid, which are: Empty, drawn, agent, star, right-arrow, and left-arrow. Then, let us define that for each domain there is a set of values *V* which represents the values that each cell object in the grid can take. It is also good to remember the difference between static and dynamic predicates. Then, the static predicates are always the same for any state of an instance, while a dynamic predicate will change depending on the state of the instance. *P* will contain both types of predicates, dynamic and static.

We have used the dynimic predicates to define the value of *V* that any object *o* can take at a given time. While we have used the static predicates to define the topology of the grid, i.e., the relative position of o with respect to o'. It is important to note that it is always preferable to have a static predicate rather than a dynamic one as this will considerably reduce the problem of the combinatorial explosion of symbols [some reference]. Finally, we define a set of relative directions *D*, which contains {right, left, down, up, right-up, right-down, left-down, left-up, horizontal, vertical}. Then, *D* is used to define the topology of the grid.

Then, *P* contains:

Dynamic predicates:

- *v(cij)*: Means that cell *o* has the value *v* at a certain time. Recall that v is a concrete value within the set *V*. For instance, if *v* is equalt to the value *agent*, then *agent(cij)* means that cell *cij* has the value agent.
- *player(n)*: This is predicate used for multiplayer games where $n \in \mathbb{N}$, and represents the player whose turn it is this turn. For instance, the predicate *player(0)* means that in a specific state is the turn of the player *0*.

Static predicates:

- *d(o, o')*: means that *o* has *o'* as contiguous in direction d. For example: *right(cij, cij')* means that to the adjacent right of cell *cij* is the cell *cij'*. Notice that some directions of *D* are redundan, e.g., the predicate *horizontal(cij, cij')* means that to the adjacent right o left of cell *cij* is the cell *cij'*, and could be defined combining two of the other predicates like *right(cij, cij')* $\vee$ *left(cij, cij')*. In a similar way *vertical(cij, cij')* can be defined as *up(cij, cij')* $\vee$ *down(cij, cij')*. Furthermore, some directions like *left(cij, cij')* are also redundant, since can be redefined as the invers of its opposite direction like *right(cij', cij)*. This redundancy will increase the number of possible features according to the grammar, however it will also reduce the complexity of the features.
- *same-row(o, o')*: Means that *o* is in the same row as *o'*. For example: *same-row(cij, cij')* means that cell *cij* is in the same row as cell *cij'*, the row *i*.
- *same-col(o, o')*: Means that *o* is in the same column as *o'*. For example: *same-col(cij, ci'j)* means that cell *cij* is in the same column as cell *ci'j*, the column *j*.

3 Sampling

To compute the policies our approach needs to expand the entire state space of a particular instance. However, some of the domains have a great diversity of possible situations in the full state space, and there are policies that lead to the goal that are extremely complicated to represent, because they lead to solve any situation tha has a path to some goal state. However, we want interpretable policies, which can be reasoned about and generalize to any possible situation in the environment from an initial state that satisfies certain rules, finding a way to the goal if exists. So, these overcomplicated policies

that are implicit in the full state space do not satisfy the research objectives. In the paper [Silver *et al.*, 2020] to find the policies expand a single trace from the initial state to the goal, which they try to mimic. However, if we expand the whole state space, we are trying to solve the game from any possible state within the configuration, which would be equivalent to having no concrete rules for the initial states, this is possible with our approach, but this would be a different and much more complicated problem than the one proposed by [Silver *et al.*, 2020]. To summarize, in richer domains, the fewer rules an initial state must satisfy, the more complicated is the task of finding a policy that generalizes to any initial state.

To resolve the situation, in domains with a huge number of states, e.g., chess, we do state space sampling, instead of expanding the entire state space. In the method we use we need during state expansion, for each state, to label one or more actions as recommended, and the rest as not recommended. The actions are labeled as recommended according to one or more policies at a time. To expand the state space of a given instance we perform two steps. For the first step, we follow the actions recommended by the policies for each expanded state for. In the second step, we visit each of the states reached from the nodes expanded in the first step following the non-recommended actions.

Using several policies at a time is definitely unsound, but there is no problem because we want to sample the states, and do not need to have explicit traces that lead to the goal. Nevertheless, as we will see from the results, if we use a single policy for sampling, i.e., the trivial case, the approach is able to imitate this policy or find another one within the expanded state space.

4 Results

In this section we will show, for each domain, the features and policies computed, as well as the results archived by them.

4.1 Two Pile Nim

Task Description

There are two columns (piles) of matchsticks and empty cells. Clicking on a matchstick cell changes all cells above and including the clicked cell to empty; clicking on a empty cell has no effect. After each matchstick cell click, a second player takes a turn, selecting another matchstick cell. The second player is modeled as part of the environment transition and plays optimally. When there are multiple optimal moves, one is selected randomly. The objective is to remove the last matchstick cell.

In this domain V is {token, empty}.

Features:

(a) $|\mathit{player}.0| \equiv$ Is the turn of the player 0, i.e., our turn.

(b) $|(\exists \mathit{leftdown.token}) \sqcap \mathit{empty}| \equiv$ Number of cells that have value empty and at the adjacent left-down have a cell with value token

(c) $|(\exists \mathit{right.token}) \sqcap \mathit{empty}| \equiv$ Number of cells that have value empty and at the adjacent right have a cell with value token

Policy:

1. $(a > 0) \wedge (b = 0) \wedge (c > 0) \mapsto \{a, b, c\downarrow\}, \{a, b, c\}$
2. $(a > 0) \wedge (b > 0) \wedge (c = 0) \mapsto \{a, b\downarrow, c\uparrow\}$

Discussion

The policy has been computed in a completely unsupervised way, expanding the full state space. Two training instances have been used, and it manages to solve all test instances. Then, the computed policy manages to solve all the test instances provided by [Silver *et al.*, 2020], and furthermore any possible configuration of the environment on any size of the grid. We can manually reason that it will meet any possible configuration of the environment in which the game starts in a situation where winning is possible, since the initial part of the policy clauses are general enough to encompass these situations, and map to one of the two states describing the single best play. It is important to note that the policy of player one, i.e., our adversary, is deterministic.

4.2 Checkmate Tactic

Task Description

Task Description This task is inspired by a common checkmating pattern in Chess. Note that only three pieces are involved in this game (two kings and a white queen) and that the board size may be H × W for any H, W, rather than the standard 8×8. Initial states in this game feature the black king somewhere on the boundary of the board, the white king two cells adjacent in the direction away from the boundary, and the white queen attacking the cell in between the two kings. Clicking on a white piece (queen or king) selects that piece for movement on the next action. Note that a selected piece is a distinct value from a non-selected piece. Subsequently clicking on an empty cell moves the selected piece to that cell if that move is legal. All other actions have no effect.

If the action results in a checkmate, the game is over and won; otherwise, the black king makes a random legal move. In this domain V is {empty, black-king, white-king, white-queen, highlighted-white-queen, highlighted-white-king, highlighted-black-king}.

Features:

(a) $|(\forall \mathit{left\text{-}down.empty}) \sqcap \mathit{white\text{-}queen}| \equiv$ Number of cells that have value white-queen and at the adjacent left-down have a cell with value empty

(b) $|(\forall \mathit{same\text{-}row.black\text{-}king}) \sqcap \mathit{white\text{-}queen}| \equiv$ Number of cells that have value white-queen and at the same row have a cell with value black-king

(c) $|(\forall \mathit{right\text{-}down.empty}) \sqcap \mathit{white\text{-}queen}| \equiv$ Number of cells that have value white-queen and at the adjacent right-down have a cell with value empty

(d) $|(\forall \mathit{right\text{-}up.empty}) \sqcap \mathit{white\text{-}queen}| \equiv$ Number of cells that have value white-queen and at the adjacent right-up have a cell with value empty

(e) $|(\forall \mathit{left\text{-}up.empty}) \sqcap \mathit{white\text{-}queen}| \equiv$ Number of cells that have value white-queen and at the adjacent left-up have a cell with value empty

(f) $|(\forall$*same-col.white-queen*$) \sqcap$ *black-king*$|$ $\equiv$ Number of cells that has value black-king and at the same column have a cell with value white-queen

Policy:

1. $(f = 0) \wedge (b = 0) \wedge (a = 0) \wedge (e = 0) \wedge (c = 0) \wedge (d = 0) \mapsto \{f, b\uparrow, a\uparrow, e\uparrow, c\uparrow, d\uparrow\}\{f, b, a, e\uparrow, c\uparrow, d\}$ $\{f\uparrow, b, a\uparrow, e\uparrow, c\uparrow, d\uparrow\}\{f, b, a\uparrow, e, c, d\uparrow\}$ $\{f, b, a\uparrow, e, c\uparrow, d\}$
2. $(f = 0) \wedge (b = 0) \wedge (a > 0) \wedge (e > 0) \wedge (c > 0) \wedge (d > 0) \mapsto \{f, b, a\downarrow, e\downarrow, c\downarrow, d\downarrow\}$

Discussion

The policy has been computed in a semi-unsupervised way, since a policy has been used to sample the state space. The teach policy consists of selecting the white queen and placing it between the two kings in such a way that the black king cannot escape. To compute the policy, eight instances 4×4 and eight instances 5×5 have been used. The computed policy manages to solve all the test instances provided by [Silver *et al.*, 2020], and furthermore any possible configuration of these three pieces on any size of the board. We can manually reason that it will solve any possible configuration of the environment in which the game starts in a situation where winning is possible, since the initial part of the policy clauses are general enough to encompass these situations, and map to select the whit queen and move it to any position between the two kings, which is the best move possible. It is important to note that the policy of player one, i.e., our adversary, is deterministic.

4.3 Reach For The Star

Task Description

In this task, a robot must move to a cell with a yellow star. Left and right arrow keys cause the robot to move. Clicking on an empty cell creates a dynamic brown block. Gravity is always on, so brown objects fall until they are supported by another brown block. If the robot is adjacent to a brown block and the cell above the brown block is empty, the robot will move on top of the brown block when the corresponding arrow key is clicked. (In other words, the robot can only climb one block, not two or more.).

In this domain *V* is {empty, drawn, agent, star, left-arrow, right-arrow}.

Features:

(a) $|\forall$*same-row.empty*$|$ $\equiv$ Number of cells in an all-empty-cells row.

(b) $|\forall$*up***.empty*$|$ $\equiv$ Number of cells with all cells above being empty.

(c) $|\forall$*left***.empty*$|$ $\equiv$ Number of cells with all cells to the left being empty.

(d) $|\forall$*right***.empty*$|$ $\equiv$ Number of cells with all cells to the right being empty.

(e) $|(\exists$*left.drawn*$) \sqcap$ *drawn*$|$ $\equiv$ Number of drawn cells with a drawn cell on their left.

Policy:

1. $(e > 0) \wedge (c > 0) \wedge (d > 0) \wedge (b > 0) \wedge (a > 0) \mapsto \{e, c\uparrow, d, b, a\}, \{e, c, d\downarrow, b\uparrow, a\downarrow\}, \{e, c\downarrow, d, b\uparrow, a\downarrow\}, \{e\uparrow, c, d\downarrow, b\downarrow, a\downarrow\}, \{e, c\downarrow, d\downarrow, b\uparrow, a\downarrow\}, \{e, c\uparrow, d, b\uparrow, a\uparrow\}, \{e, c, d\uparrow, b\uparrow, a\uparrow\}, \{e, c, d\uparrow, b, a\}, \{e, c\uparrow, d\downarrow, b\downarrow, a\}, \{e, c\downarrow, d, b, a\downarrow\}, \{e, c\downarrow, d\uparrow, b\downarrow, a\}, \{e\uparrow, c\downarrow, d, b\downarrow, a\downarrow\}, \{e, c, d\downarrow, b, a\downarrow\}$

Discussion

The policy has been computed in a semi-unsupervised way, since a policy has been used to sample the state space. Only four instances 7×9 have been used to compute the policy. The computed policy manages to solve all the test instances provided by [Silver *et al.*, 2020], and also any possible solvable configuration at any board size. We can manually reason that it will solve any possible configuration of the environment in which the game starts in a situation where winning is possible.

4.4 Chase

Task Description

This task features a stick figure agent, a rabbit adversary, walls, and four arrow keys. At each time step, the adversary randomly chooses a move (up, down, left, or right) that increases its distance from the agent. Clicking an arrow key moves the agent in the corresponding direction. Clicking a gray wall has no effect other than advancing time. Clicking an empty cell creates a new (blue) wall. The agent and adversary cannot move through gray or blue walls. The objective is to "catch" the adversary, that is, move the agent into the same cell. It is not possible to catch the adversary without creating a new wall; the adversary will always be able to move away before capture.

In this domain *V* is {empty, target, agent, drawn, wall, up-arrow, down-arrow, left-arrow, right-arrow}.

Features:

(a) $|empty| \equiv$ Number of empty cells.

(b) $|(\exists$*same-row.drawn*$) \sqcap$ *target*$|$ $\equiv$ Whether there is a blocked cell in the rabbit's row.

(c) $|(\exists$*horizontal.*$(\exists$*same-col.drawn*$)) \sqcap$ *target*$|$ $\equiv$ Whether there is some blocked cell in one of the two columns that are adjacent to the rabbit's column.

(d) $Dist(\exists$*same-col.agent*; *horizontal*; $\exists$*same-col.target*$) \equiv$ Absolute distance between the agent's column and the rabbit's column.

(e) $Dist(\exists$*same-row.agent*; *vertical*; $\exists$*same-row.target*$) \equiv$ Absolute distance between the agent's row and the rabbit's row.

Policy:

1. $(c = 0) \wedge (b = 0) \wedge (d > 0) \wedge (e > 0) \wedge (a > 0) \mapsto \{c, b, d, e\uparrow, a\}\{c, b, d\uparrow, e, a\}$ $\{c\uparrow, b\uparrow, d, e\uparrow, a\downarrow\}\{c\uparrow, b\uparrow, d, e, a\downarrow\}$
2. $(c = 0) \wedge (b = 0) \wedge (d = 0) \wedge (e > 0) \wedge (a > 0) \mapsto \{c, b, d\uparrow, e\uparrow, a\uparrow\}\{c\uparrow, b\uparrow, d, e, a\downarrow\}$

3. $(c > 0) \wedge (b > 0) \wedge (d = 0) \wedge (e > 0) \wedge (a > 0) \mapsto \{c\downarrow, b\downarrow, d\uparrow, e\uparrow, a\uparrow\}\{c, b, d, e\downarrow, a\}$
4. $(c > 0) \wedge (b > 0) \wedge (d > 0) \wedge (e > 0) \wedge (a > 0) \mapsto \{c, b, d\downarrow, e, a\}$
5. $(c = 0) \wedge (b = 0) \wedge (d > 0) \wedge (e = 0) \wedge (a > 0) \mapsto \{c, b, d\uparrow, e\uparrow, a\uparrow\}$

Discussion

The policy has been computed in a semi-unsupervised way, since a policy has been used to sample the state space.Twenty-nine instances 4×6 and four instances 6×7 have been used to compute the policy.Twenty-nine instances 4×6 and four instances 6×7 have been used to compute the policy. The computed policy manages to solve all the test instances provided by [Silver *et al.*, 2020], however it solves $\sim 95\%$ of all possible solvable configurations and at any board size. The last $\sim 5\%$ of the instances can be solved if we do a small loockahead of depth 2, and check not only if the transition from the current state to a successor is good, but also the transition from the current state to the successor of the successor, and from the successor to the successor of the successor. We can manually reason that it will solve any possible configuration of the environment in which the game starts in a situation where winning is possible.

4.5 Stop The Fall

Task Description

This task involves a parachuter, gray static blocks, red "fire", and a green button that turns on gravity and causes the parachuter and blocks to fall. Clicking an empty cell creates a blue static block. The game is won when gravity is turned on and the parachuter falls to rest without touching (being immediately adjacent to) fire.

In this domain *V* is {empty, falling, red, static, advance, drawn}.

Features:

(a) $|drawn| \equiv$ Number of cells that have value drawn

(b) $Dist(falling; down; empty) \equiv$ Distance between the cell that have value falling and the first cell that have value empty rolling down

Policy:

1. $(b > 0) \wedge (a > 0) \mapsto \{b\uparrow, a\uparrow\}\{b, a\}$
2. $(b > 0) \wedge (a = 0) \mapsto \{b\uparrow, a\uparrow\}$

Discussion

The policy has been computed in a semi-unsupervised manner, since a policy has been used to sample the state space. Only two training instances have been used to compute the policy. The computed policy manages to solve all the test instances provided by [Silver *et al.*, 2020], and also any possible solvable configuration at any board size. We can manually reason that it will solve any possible configuration of the environment in which the game starts in a situation where winning is possible.

References

[Silver *et al.*, 2020] Tom Silver, Kelsey R. Allen, Alex K. Lew, Leslie Pack Kaelbling, and Josh Tenenbaum. Few-shot bayesian imitation learning with logical program policies. In *AAAI*, 2020.