Model Minimization For Online Predictability

Sriram Gopalakrishnan,[1] **Subbarao Kambhampati,** [1]
[1] Arizona State University
sgopal28@asu.edu

Abstract

For humans in a teaming scenario, context switching between reasoning about a teammate's behavior and thinking about thier own task can slow us down, especially if the cognitive cost of predicting the teammate's actions is high. So if we can make the prediction of a robot-teammate's actions quicker, then the human can be more productive. In this paper we present an approach to constrain the actions of a robot so as to increase predictability (specifically online predictability) while keeping the plan costs of the robot within acceptable limits. Existing works on human-robot interaction do not consider the computational cost for predictability, which we consider in our approach. We approach this problem from the perspective of directed graph minimization, and we connect the concept of predictability to the out-degree of vertices. We present an algorithm to minimize graphs for predictability, and contrast this with minimization for legibility (goal inference) and optimality.

When humans are teaming together to achieve a set of tasks, the time an agent spends observing and deciphering the teammate's plan (to avoid conflicts) is time taken away from working on their own tasks. The computation for predicting the plan-of-action of a teammate maybe so high as to be infeasible. Even if it is within their ability to infer the actions of the teammate, the time and cognitive cost cost of doing so could harm the person's productivity since it takes away attention and effort from their own tasks. So in this work we focus on the problem of simplifying the behavior of an AI agent, such that the cost of predicting the coworker robot's actions is significantly reduced while keeping the robot's costs below an acceptable cutoff.

In the existing AI planning literature on human and robot interaction, the human is considered to have a separate human-model of the robot ((Chakraborti et al. 2019),(Kulkarni et al. 2019)) which represents the human's expectations of the robot . The measure of interpretable behavior is determined by how closesly the robot's actions aligns with those predicted by the human's model of the robot. Specifically, the actions should match those expected in the optimal plan in the human's model of the robot. While it seems beneficial to conform to the human's model for predictable behavior during human-AI teaming, this approach pays no heed to the cognitive effort expected of the human. The human's model need could be non-trivial, and to expect the human to infer or compute the optimal plan can be cognitively demanding. This is more so if the human is in a teaming scenario when the human has to work on their own tasks. In such cases a higher cognitive effort and divided attention to predict actions for avoiding conflicts with a robot coworker would detrimentally affect the human's own performance their task.

To help understand and motivate this work, let's consider the running example of a nursing home; and let's just consider the simpler problem of motion in this environment for communicating the concepts in our work; please note that our work is not limited to motion plans. In our nursing home example, there is a robot assisting with the care of elder citizens and works along side a full time nurse. The robot has a set of tasks it can be asked to do by the residents or nurse: (a)getting food and water to the residents from the kitchen, (b)bringing pills from the medicine cabinet to the appropriate residents The nurse handles all the important medical and human needs of the residents. The nursing home is illustrated in Figure 2, where each grid position represents a position state that the robot can be in. The nurse, the robot, and

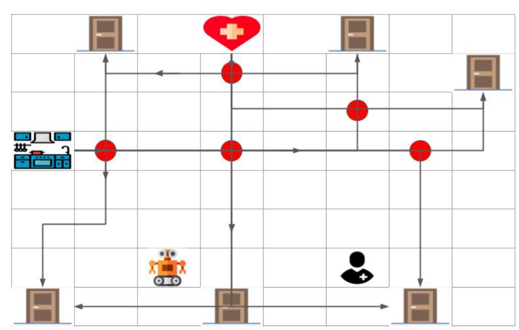


Figure 1: Nursing Home with robot state and action space pruned to preserve shortest paths to go between kitchen and medicine cabinet to all resident rooms. The grid squares that contain the path lines are the chosen states in the underlying grid graph

the senior citizens cohabit in the same environment. When

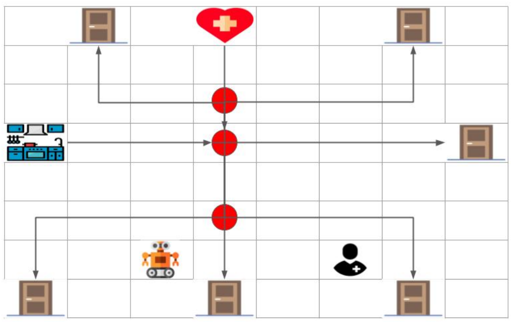


Figure 2: Nursing Home with Robot state and action space pruned to have the fewest decision nodes. The grid squares that contain the path lines are the chosen states in the underlying grid graph

the nurse is going about her/his activities, if the nurse is distracted by thinking about how the robot will move or act then it takes attention away from the task the nurse is doing; this could be very bad in the case of a medical emergency when divided attention could be costly. Additionally the senior residents might worry about colliding with the robot and hurt themselves, even if the robot has good collision avoidance. Simplicity and consistency of actions can make it easier for humans working in the same space as they can easily predict the robot's behavior. In our work, we will present one approach in which the robot intentionally limits it's actions so as to be predictable with lesser cognitive cost.

In the scenario we described, the human has to predict the actions of the robot without necessarily knowing all the prior states. In our example, this could (for example) be when the nurse looks up from their medical chart while walking to a senior resident's room, and sees the robot moving near the resident's room. The nurse should be able to quickly infer where the robot is going with minimal information or inference cost, and know if they might risk crossing the robot's path. This is an ”online” predictability problem; as opposed to the assumptions in existing literature in which the human has observed some prefix of actions from an initial state(Kulkarni et al. 2019). Having less information makes predictability harder. To tackle this, we take the approach of reducing or minimizing the state transition graph of the robot. The state transition graph with it's states (vertices) and actions (edges) is the model of the robots behavior. Our approach reduces the number of states that have more than one outgoing edge whilst keeping plan costs down. Our idea is that if there are sequences of states with just one outbound edge, then predictability is easy and will become familiar (memorized) over time as the same actions are always done from the same states. At every vertex(state) with more than one outgoing edge, the human has to think of what next action the robot will take. Either the human can compute/infer the remainder of the plan if the goal is known, or the robot can communicate it's next action(s) at those states. We will call these states with more than one outgoing edge as *Branching States*; these are the states at which the human has to think or get information in order to predict actions, and is where the predictability cost is incurred. Figures 1 and 2 respectively show the difference between minimizing the underlying state transition graph to preserve shortest paths, and minimizing to improve predictability. In Figure 2, there are fewer decision points (red circles), but the paths are suboptimal. We will discuss this tradeoff more in the methodology and experiments sections. In this paper we first introduce the problem of model minimization for online predictability. We then introduce metrics for this problem and present an algorithm to minimize the state transition graph to optimize for predictability for a given set of tasks. We then run experiments to show the validity of our approach, and compare the results with minimization done for optimality and legibility (goal prediction). Finally we discuss the related work from the literature on model minimization and environment design; none of which has tackled the problem of online predictability. We end with a summary of our work, and future research directions for feasible human-robot teaming.

Problem formulation

The problem of minimization for predictability is given by the tuple

$$P_{min} =< G, T, C, P > \qquad (1)$$

where

- *G* is the directed graph defined by its vertices and edges *(V,E)*. The vertices represent states, and outgoing edges represent the actions that can be taken from them.
- *T* is the set of ordered pairs of initial and goal vertices that need to remain connected in the minimized graph. For this paper, will refer to these pairs as tasks.
- *C* is the cutoff associated to each pair in *T*. The cost of the paths between each initial and goal vertex should be less than it's corresponding cutoff
- *P* is the probability of each task in *T*.More generally, this can be any weight representing importance.

The objective is to reduce the input graph *G* so as to minimize complexity and predictability metrics discussed in the following section.

Metrics for Predictability

The following metrics are used to determine how complex a model is for online predictability

- Number of Branching Vertices (*NBV*): The fewer branching vertices (states) there are , the fewer computation steps would be needed to predict the actions. States with only one outgoing edge require minimal thought, and can become memorized over time through repeated consistent observation.
- Probabilistic Predictability Cost (*PPC*) The average number of branching states that the robot will go through for a task. This metric assumes the cost at each branching state

is the same, which is not always true but it is the simplifying assumption we make for this work.

$$PPC(T, W) = \sum_{t \in T} (P(t) * \sum_{v \in S(t)} 1[deg^{+}(v) > 1]) \quad (2)$$

where *P()* returns the probability of the task, and 1[] is an identity function that returns 1 when the condition is true.

- Weighted Ratio *NV/NBV*: Tells us how often a person has to think about decisions at branching states (vertices). It gives the average number of states in a sequence that don't require information or additional thought to predict. This is the only metric in our work in which a higher value is better

$$NV/NBV(V, S, T, P) = \frac{\sum_{t \in T} \sum_{v \in S(t)} P(t)}{\sum_{t \in T} \sum_{v \in S(t)} P(t) * 1[deg^{+}(v) > 1]} \quad (3)$$

where *S()* returns the shortest path for the task in the graph being evaluated, and $deg^{+}(.)$ returns the outdegree of the input vertex.

Among these metrics, *PPC* and the ratio of *NV/NBV* are central to evaluating online predictability of a model. *NV/NBV* tells us that if the human were to randomly observe the robot in a state, on average how many steps can the human predict or rollout before needing to think or ask for information about the robot's plan, i.e. incur a prediction cost. Sometimes the human may incur no teammate prediction cost for the course of their current task, and this is ideal. Let's take the example shown in Figure 2. If the nurse was moving to the middle row at the bottom and the robot was on the path to the left of that door, then the human knows with a glance that the robot will only ever move towards door, since the robot has restricted it's actions accordingly. This can be predicted with just the current state, no other information or computation. The human knows they can move without conflict to the middle door to complete her/his task without any further thought to the robot. The *PPC* metric is the expected predictability cost assuming the human observes from the start of a task and wants to know the entire plan. It quantifies the average number of computation steps, or alternatively pieces of information the human needs to know the full plan. The information here is the edge taken at a decision vertex. The human can either ask the robot for the information (like "Which way are you turning"), or the human can heuristically evaluate successor states at the decision vertex to infer the next action; the latter would either require goal knowledge and a heurisitic, or a policy function that can tell us the next action. Either way, a prediction cost is incurred at the branching states.

Methodology

Our problem of graph minimization for predictability is one of constrained optimization. We need to find a graph that minimizes the *PPC* cost such that all the required pairs of initial and goal states in *T* are connected, and there is a path between them that is under the corresponding cutoff specified by *C*. To find such a minimized graph, we used a hill-climbing search approach. The two key challenges in using this approach is finding a good set of paths for each pair of initial and goal states, and how to score (compare) different graph candidates. Our approach is shown in Algorithm 1. The number of possible paths between the initial and goal states for an arbitrary graph can explode combinatorially based on the number of edges and graph-connectivity. Paths cannot be evaluated independently of others; it is the combination of all paths in the graph that determine it's PPC score. The combinatorial number of cases makes this a challenging search problem. Tackling it with a hill climbing search requires choosing a good population of paths as it largely determines the outcome of the search. In this regard, one set of helpful constraints are the cutoffs or restrictions on the path costs for the various tasks. This helps constrain our search space of paths to consider. This is shown in the function *BUILD_CANDIDATES()* in Algorithm 1.

The other part the search process is evaluating and comparing successor candidates to determine what direction to go. From a given set of paths that satisfy all initial and goal state pairings in *T*, we consider all single path replacements, i.e. replacing one of the paths for a particular task. Then we compare the score of all the replacements, and if it is lower than the current score, we take that step. If not, we stop the search process. Finally, we use a fixed number of random restarts(10) from random initial candidates to repeat the hill climbing process, and use the best result. With such an algorithm, more candidates, and more random restarts can give better results, at the expense of compututation and time; some of the computation steps can be effectively parallelized such as when choosing and evaluating the possible successor states. We will make the code available after the review of this paper

The minimized graph for legibility is obtained by a similar hill climbing approach, except the metric that is minimized is the Probabilistic Legibility Cost (*PLC*). It considers the number of steps in the path before the minimum of the shortest paths to each of the goals correctly identifies the true goal. The prediction should not change thereafter. This metric is in accordance with existing work on plan legibility (Kulkarni et al. 2019), as it determines the target goal based on the costs to each of the goals. *PLC* is formally defined as follows:

$$\begin{gathered} PLC(S, T, G, P) = \sum_{t \in T} P(t) * \\ \arg\min_{i \in indices(S[t])} (1[\arg\min_{g \in G} (cost(S[(S(t)[x], g)]) = goal(t))]) \\ s.t. \quad \forall j \in \{len(S(t) > j > i\} | \\ \arg\min_{g \in G} (cost(S[(S(t)[x], g)]) = goal(t)), \\ \nexists g' \in G \setminus goal(t) | \\ cost(S[(S(t)[x], g')]) <= cost(S[(S(t)[x], goal(t))]) \end{gathered} \quad (4)$$

where *G* is the set of goal vertices from the initial and

Algorithm 1 Graph Minimization For Predictability By Hill Climbing Search

```
function PREDICTABILITY_MINIMIZ(G = (V, E), tasks, cutoffMap, weightsMap, restartCount, maxPopul)
                                ▷ The connection to the problem definition is tasks = T, cutoffMap = C, weightsMap = W
    taskCandidatesMap = BUILD_CANDIDATES(G, tasks, cutoffMap, maxPopul)
    minimized_graph = HC_SEARCH(G, tasks, taskCandidatesMap, cutoffMap, weightsMap, restartCount)
return minimized_graph
end function
----------------------------------------------------------------------------------------------------
function BUILD_CANDIDATES(G = (V, E), tasks, cutoff, maxPopul)
    taskCandidatesMap = {}                                        ▷ maps each (i, g) ∈ I × G to a set of paths
    for (i, g) ∈ tasks do
        copyG = COPY(G)                                           ▷ make an copy of the graph to edit
        for i = 0 to maxPopul do
            shortestPath = GET_SHORTEST_PATH(copyG, i, g)
            if LENGTH(G, shortestPath)< cutoffMap[(i, g)] then    ▷ We measure the cost in the original graph
                taskCandidatesMap[(i, g)] = taskCandidatesMap[(i, g)] ∪ shortestPath
                for (v_1, v_2 ∈ shortestPath do
                    copyG[v_1][v_2]∗ = 2                          ▷ double path edges' weights to get a diverse set of paths
                end for
            end if
        end for
    end for
    return taskCandidatesMap
end function
----------------------------------------------------------------------------------------------------
function HC_SEARCH(G, tasks, taskCandidatesMap, cutoffMap, weightsMap, restartCount)
    bestMinScore = 0
    bestPathsMap = {}                                             ▷ tracks the best paths for each task t = (i, g)
    for i = 0 to restartCount do                                  ▷ Random restarts to improve search
        roundBestMinScore = 0
                                                                  ▷ Initialize with a random starting point
        currPathMap = {t : RANDOM_CHOICE(taskCandidatesMap[t]) | ∀t ∈ tasks}
        while True do
            prevScore = roundBestMinScore
            for t ∈ tasks do
                newScore, newPathMap = REPLACE_SINGLE_PATH(
                        t, G, tasks, currPathsMap, taskCandidatesMap, cutoffMap, weightsMap)
                if newScore < roundBestMinScore then
                    roundBestMinScore = newScore
                    currPathMap = newPathMap
                end if
            end for
            if prevScore == roundBestMinScore then
                break                                             ▷ Terminate current round when no improvement is seen
            end if
        end while
    end for
    if roundBestMinScore < bestMinScore then
        bestMinScore = roundBestMinScore
        bestPathsMap = currPathsMap
    end if
    minimizedGraph = BUILD_GRAPH_FROM_PATHS(bestPathsMap)        ▷ returns a graph with all the nodes
                                                                  ▷ and edges from the given paths
    return minimizedGraph
end function
```

Algorithm 2 Replacing paths to reduce the number of branching vertices

function REPLACE_SINGLE_PATH($t, G, tasks, currPathMap, taskCandidatesMap, cutoffMap, weightsMap$)
  $currBestPath = currTaskToPathMap[t]$
  ▷ Any new option has to beat the current option to be selected
  $minScore$ = COUNT_BRANCHING_NODES($currPathMap, weightsMap$)
  $chosenPathCost$ = ADD_EDGE_WEIGHTS($currPathMap[t]$)
  $editablePathMap$ = COPY($taskCandidatesMap$)
  ▷ The above is a data structure to hold edits to the original set of paths between initial and goal vertices
  for $altPath$ in $taskCandidatesMap[t]$ **do**
    $editablePathMap[t] = altPath$
    $newScore$ = COUNT_BRANCHING_NODES($editablePathMap, weightsMap$)
    $newCost$ = ADD_EDGE_WEIGHTS($altPath$)
    if $newScore < minScore$ **then**
      $minScore = newScore$
      $chosenPathCost = newCost$
      $currBestPath = altPath$
    end if
    if $newScore == minScore \;\&\; newCost < chosenPathCost$ **then**
      $minScore = newScore$
      $chosenPathCost = newCost$
      $currBestPath = altPath$
    end if
    $editablePathMap$ = COPY($taskCandidatesMap$) ▷ reset this data structure
  end for
  $editablePathMap[t] = currBestPath$
  return ($minScore, editablePathMap$)
end function

goal pairs in *T*, *S()* returns the shortest path (in the graph being evaluated) for a given pair of initial and goal state. The path returned can be indexed to get the vertex at that step. *cost()* gives the cost for a specified path, and *goal()* returns the goal vertex associated with an input task. The *PLC* score tells us (on average) how many steps an observer has to wait after the initial state before the goal can be inferred from the shortest path costs. Finally, to minimize the graph to preserve optimal paths, we simply discard all nodes and edges not on the shortest paths for the given tasks in the original (full) graph. We now compare the graphs minimized for predictability, legibility, and optimality in a set of grid graphs with varying and random number of nodes and edges dropped.

Experiments

In line with our running example of a nursing home, we ran experiments on multiple grid graphs with a proportion of nodes and edges randomly removed. The removed nodes represent obstacles in the space. Since edges are removed too, not all paths are bidirectional. This could reflect path restrictions such as in narrow corridors or directions in which doors open outward (could hit a resident).

Each experimental grid is made by starting with a full grid of size 25×25 (see fig 3a) with all edges bidirectional, and then randomly removing nodes and edges whilst maintaining connectivity between the remaining nodes. The graphs used in these experiments will be part of the codebase, as well as the graph generation code. The random graphs and tasks in our experiments can be consistently regenerated; we used a fixed random seed set to 1. We will also store the graphs in the code base. The code will be publicly available.

We varied the ratio of the number of dropped nodes between $[0.0, 0.3]$ in increments of 0.1, and the same for the ratio of dropped edges. This gives us 16 random grid-graphs of different connectivity. For each grid-graph we chose 10 random tasks, defined by randomly chosen initial and goal states. All tasks had equal probability set to $1/|T|$. For all hill climbing searches, we kept the max population ($maxPopul$) for paths per task to 10, and the number of random restarts ($restartCount$) to 5. We tried higher values for both these parameters upto 20, and found that the improvement in results was not significant beyond our chosen values for our experimental graphs. For larger graphs increasing these parameters would help the search, at the cost of computation time. We will parallelize the search process before releasing the code to help with this.

The paths added to the search space for each task, were no more than 3 times the optimal cost, and this was the $cutoff$ parameter in the hill climbing search. We chose 3 as a reasonable upper bound on suboptimality. Higher suboptimal cutoff did not improve the results for our experiments. One reason for this could be because very long paths were more likely to intersect with other paths for other tasks during hill climbing search.

We ran the experiments on an Intel® Core™ i7-6700 CPU, with 32GB of RAM, running Ubuntu 16.04. All graph operations were done with networkx python library (Hag-

Edge Drop Ratio / Node Drop Ratio	0.0	0.1	0.2	0.3
0.0	6, 1.2, 13.3 9, 1.7, 11.7 3, 0.9, 22.4	25, 6.0, 3.5 13, 2.7, 9.3 3, 0.7, 36.0	17, 3.6, 4.7 11, 2.1, 8.7 2, 0.5, 44.0	20, 4.5, 4.2 11, 2.4, 10.0 4, 1.0, 22.5
0.1	9, 1.9, 7.2 5, 1.0, 17.3 1, 0.2, 76.9	15, 3.2, 5.9 10, 2.1, 10.4 3, 0.7, 32.8	15, 3.5, 4.3 14, 3.1, 5.9 6, 1.3, 14.8	14, 3.1, 6.6 9, 1.9, 12.3 6, 1.4, 16.2
0.2	14, 3.0, 5.2 7, 1.8, 9.9 6, 1.5, 13.2	10, 2.3, 7.5 11, 2.5, 7.3 3, 0.7, 29.1	28, 7.4, 3.1 18, 4.1, 6.4 11, 3.5, 7.8	22, 4.8, 4.9 17, 4.4, 5.9 9, 3.1, 10.1
0.3	35, 11.9, 2.5 66, 17.6, 1.9 7, 3.0, 14.5	20, 5.1, 4.0 16, 4.1, 5.5 5, 1.8, 13.1	15, 4.0, 6.1 11, 2.7, 9.7 6, 1.7, 18.8	20, 6.3, 6.5 14, 4.4, 9.3 11, 4.2, 10.4

Table 1: Predictability metrics for grid size 25x25 with various combinations of node and edge drop ratios. In each entry, from left to right are the metrics $NB, PPC, NV/NBV$. From top to bottom are the minimization by (1)optimality (2) legibility, and (3)predictability

Edge Drop Ratio / Node Drop Ratio	0.0	0.1	0.2	0.3
0.0	6.9 4.9 12.2	13.9 4.9 13.1	9.3 4.2 11.6	10.0 5.8 9.4
0.1	6.0 1.8 4.0	12.3 5.1 13.5	9.1 6.3 11.4	9.6 5.6 13.5
0.2	6.9 3.5 9.2	9.0 7.6 10.4	14.3 12.0 16.9	15.1 12.2 14.7
0.3	23.7 19.4 33.4	13.6 10.2 15.7	16.7 13.4 17.6	29.1 28.8 30.9

Table 2: Probabilistic Legibility Cost (PLC) for grid size 25x25 with various combinations of node and edge drop ratios. The order of each entry is minimization by (1)optimality (2) legibility, and (3)predictability

berg, Schult, and Swart 2008), all other operations were done with native python libraries.We minimized each graph for optimality, legibility, and predictability. For each minimized graph we computed the metrics *PLC*, *PPC*, and Weighted *NDV/NV*. The results are in Figures 1,2.

Results and Discussion

Results

In Table 1 we see the results for the predictability metrics; each entry has 3 rows (from top to bottom) corresponding to minimization for optimality, legibility and predictability. Each row itself has 3 values for the metrics (from left to right) *NB, PPC, NV/NBV*. Of these metrics *NV/NBV* is the only metric in which higher is better.

As one might expect, when the ratio of nodes and edges dropped are lower, there are more paths that satisfy the path cost cutoff which can be used in the search, and so the predictability metrics are best in these graphs. As we approach the node and edge drop ratios of $0.3, 0.3$ the predictability metrics worsen (unsurprisingly). There are fewer disparate paths within the cutoff, leading to more branching vertices, and the search is harder to minimize predictability. Simply put a greater diversity in graph connectivity makes the search process easier.

One way we could have improved the search process is a more granular search when the hill climbing search hits a local minimum. We could drop portions of the existing paths around branching vertices and look for alternate edges to complete the paths without branching vertices. Another way is to consider replacing more than one path at a time. These changes would increase the computation cost and time, and is a decision made based on the computing resources available.

As for legibility metrics, if we look at the same minimized graphs for their legibility scores in Table 2, the middle value in each entry is always the lowest and corresponds to the minimization by legibility. While the metrics for legibility and predictability are not diametrically oppposed (neither is optimality), optimizing for one often comes at a cost to the other; the legibility score for the model optimized for predictability is considerably worse than the model for legibility and vice versa. We can glean this by comparing the values in Table 1 and 2

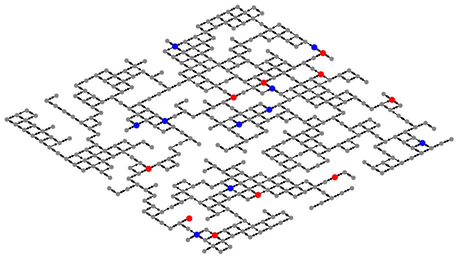
It must be noted that for predictability, the best solution, if path costs was not a concern, would be one long winding path through all the initial and goal states like Hamiltonian circuit through a subset of nodes (assuming one exists). There are no branching vertices in such a path, and by our PPC metric the predictability cost would be zero. This may go against the human's expectations, which we will discuss in the next section.

Discussion on Minimization For Predictability

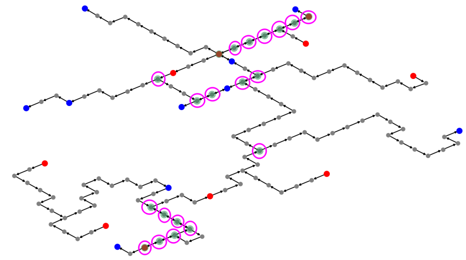
Our work focuses on reducing the number of branching vertices to reduce the steps requiring computation for predicting actions. However, there is another important aspect to consider for predictability. That other aspect is the human's prior expectations. We would posit that the more a path conforms to their existing expectations, the easier it would be to become familiar with the minimized model. One way we could factor in prior expectations is to only choose paths in our search that line up with human expectations.

Another part of the teaming problem outside our scope, is how to communicate the minimized model to the human. This can be done using environment cues, explicit communication, and also repetition over time. There would be an initial cost associated to the human learning this model. Keeping the branching nodes to a minimum also ensures that many state action sequences will be repeated consistently, helping with memorization by rote repetition.

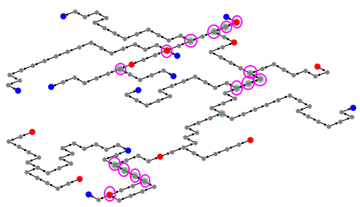
Lastly, Even if the goal is not known, the observer might be able to predict a couple of steps ahead since there are a reduced number of branching states. The higher the measure of *NV/NBV*, the more states on average that the human can quickly look ahead without knowing the goal. This maybe sufficient to avoid conflicts for the current task that the human is working on. The complete plan may not be necessary for the current task, and the action predictions can be



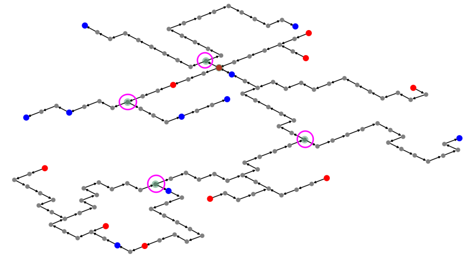
(a) Graph before minimization.



(b) Graph after minimization to preserve optimal path costs



(c) Graph after minimization for legibility.



(d) Graph after minimization for predictability.

Figure 3: Graphs corresponding to the experiment with 25x25 grid with the ratio of nodes and edges dropped equalling $(0.3, 0.1)$ respectively. Initial and Goal nodes are colored red and blue respectively. Branching nodes are highlighted in a purple circle

computed *as-needed*. The human may not know of future conflicts until they start the next task or if the tasks change. Consider this scenario in our running example: A resident in the nursing home may suddenly have a medical emergency when the nurse is performing another task, like checking and securing the premises. Now the nurse has to switch priorities to a different task, and so any effort spent thinking too far ahead for the robot teammate's actions is wasted. To our knowledge, no other work has explicitly discussed this notion of *as-needed* predictions in the context of predictability in plans. The metric *NV/NBV* tells us how inexpensive such computations can be for a teammate using a given model.

Related work

The are three related sets of literature, which can be categorized into network optimization, model minimization, and most importantly environmental design for interpretable behavior. The idea of pruning or selecting nodes and edges for optimizing some metric(s) has a vast existing literature, and is especially useful optimizing message routing networks and chip layout design. Variants to the Steiner Tree problem (for undirected graphs) have been used for modeling the problem of computing layouts for VLSI(chip) design (Grötschel, Martin, and Weismantel 1997), or more generally for problems relating to any network connectivity/routing. The directed graph analogue of this problem is closer to ours, and also has applications in network routing; specifically the problem of Directed Steiner Tree with Limited Diffusing nodes (DSTLD); it has applications in multicast packet routing (Watel et al. 2014). Solutions to DSTLD optimizes for the total edge-weight cost of the tree, and limits the number of branching states (which they call diffusing nodes) but only from a single source state. They do not consider the problem of optimizing the number of branching states across across the paths between multiple source and sink nodes. They also do not consider any cost cutoffs on each of the paths, only that the total tree cost from one root (source) node is minimal.

There has also been work done on minimizing state-transition graphs for preserving optimality. Model minimization work has been done to preserve optimality in Markov Decision Processes (MDP) (Dean and Givan 1997), as well as minimization with STRIPS style planning (Givan and Dean 1997). However, these works pay no heed to complexity of the resultant graph with respect to predictability.

Another set of related literature, is that on environment design. In a recent paper that combined much of the existing work into one cohesive framework (Kulkarni et al. 2019), the authors define legibility, predictability, and explicability under the same framework. The general theme by which our work on predictability distinguishes itself from this prior work is as follows: The existing literature ((Kulkarni et al. 2019)(Keren, Gal, and Karpas 2019),(Kulkarni et al. 2020),(Chakraborti et al. 2019)) requires (for quantifying predictability) observing a prefix of actions, not just the current state; Additionally the human would have to compute an optimal plan completion (using their separate human model of the robot) for comparison with the current actions of the robot to predict it's behavior. When the human is in a teaming scenario, both these expectations for predictability detract from the human acting on their own tasks. We do not require the prefix to be observed just from the current state (online predictability), and we try to make the path computation simpler with fewer branching nodes. Without this there could be many alternate plans to consider and multiple computations steps at each branching node.

A different approach in the literature to predictability is through plan libraries (Mirsky et al. 2019). This line of work optimizes a set of plans to reduce the number of observations needed to uniquely identify a plan.There is an implicit assumption that the observer (human) catches these observations which means they cannot just predict from the current state (not online). Additionally the plan library is small enough for the human to quickly connect it to the correct plan. We think this approach can work well if the plans are sufficiently different so as to make plan matching easier. We approach the problem of predictability from a different perspective of directed graphs and branching vertices, and so distinguish ourselves

Conclusion and Future Directions

In this paper we work on the problem of computational cost for predictability, specifically online predictability, with the motivation of making it easier for humans to team with robots. We treat the robot's domain model as a directed graph, and connect the computation cost for predictability to branching vertices (states) in the graph. Then we present a hill climbing search approach to minimize graphs for predictability, and compare this with minimization to preserve optimality, and minimization to improve legibility.

This was the first foray into model minimization for online predictability and it's computation cost. There are many problems left to address such as incorporating prior expectations of the human, redundant plans, large factored state spaces and action types (STRIPS style planning), as well as the cognitive cost of different computations at each branching state. The last problem requires considering the cognitive effort (cost) of the rule or heuristic computation that the human must use to determine the next action. Alternatively, the human can just incur a communication cost and ask the robot. This trade off between communication and computation for predictability is an interesting avenue we hope to pursue. Lastly, human trials become essential when comparing cognitive costs of different computations and will be part of subsequent work.

References

Chakraborti, T.; Kulkarni, A.; Sreedharan, S.; Smith, D. E.; and Kambhampati, S. 2019. Explicability? legibility? predictability? transparency? privacy? security? the emerging landscape of interpretable agent behavior. In *Proceedings of the international conference on automated planning and scheduling*, volume 29, 86–96.

Dean, T.; and Givan, R. 1997. Model minimization in Markov decision processes. In *AAAI/IAAI*, 106–111.

Givan, R.; and Dean, T. 1997. Model minimization, regression, and propositional STRIPS planning. In *IJCAI*, 1163–1168. Citeseer.

Grötschel, M.; Martin, A.; and Weismantel, R. 1997. The Steiner tree packing problem in VLSI design. *Mathematical Programming* 78(2): 265–281.

Hagberg, A. A.; Schult, D. A.; and Swart, P. J. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In Varoquaux, G.; Vaught, T.; and Millman, J., eds., *Proceedings of the 7th Python in Science Conference*, 11 – 15. Pasadena, CA USA.

Keren, S.; Gal, A.; and Karpas, E. 2019. Goal recognition design in deterministic environments. *Journal of Artificial Intelligence Research* 65: 209–269.

Kulkarni, A.; Sreedharan, S.; Keren, S.; Chakraborti, T.; Smith, D.; and Kambhampati, S. 2020. Designing Environments Conducive to Interpretable Robot Behavior. *arXiv preprint arXiv:2007.00820* .

Kulkarni, A.; Sreedharan, S.; Keren, S.; Chakraborti, T.; Smith, D. E.; and Kambhampati, S. 2019. Design for Interpretability. In *ICAPS Workshop on Explainable AI Planning (XAIP)*.

Mirsky, R.; Gal, K.; Stern, R.; and Kalech, M. 2019. Goal and plan recognition design for plan libraries. *ACM Transactions on Intelligent Systems and Technology (TIST)* 10(2): 1–23.

Watel, D.; Weisser, M.-A.; Bentz, C.; and Barth, D. 2014. Directed Steiner tree with branching constraint. In *International Computing and Combinatorics Conference*, 263–275. Springer.