1 Randomly Generated Models

1.1 Why Use Randomly Generated Models

To make a quantitative assertion about the performance of the algorithms, we need first a broad spectrum of models we can test the algorithms on. At the moment we are aware of 12 case studies, giving rise to 17 distinct reachability problems for stochastic games. These problems have often parameters that can be adjusted, resulting in infinetely many graphs. However, graphs derived from the same problem will have similar graph structures.

Clearly, we need more distinct models to provide a solid algorithm comparison. Creating handcrafted reachability problems is useful to make comparisons on edge-case situations and show off the weaknesses of categories of algorithms (like the haddad-monmege model does for value iteration), but contributes only very little to the overall coverage of possible reachability-problems for stochastic games.

To tackle this issue we introduce randomly generated models.

1.2 Constraining the Random-Generation Process

While it is possible to completely randomize every property of a model, this procedure would yield graphs with undesirable properties. A model is **undesirable** if there are states to which there is no path from the initial state since these states are just overly complex sinks. To this end we create easily modifiable constraints that guide the random-generation process into creating desirable models. The constraints are fed to a generator that uses guidelines on how to create the models aswell as simple parameters like the maximum number of actions a state may have. Ideally, we would like to have guidelines that allow us to sample uniformly from the space of all possible models fulfilling our constraints. Note that this requires a natural upper bound on the number of states and a finite set of transition probabilities, since otherwise the model space becomes infinite. Additionally, at this moment I have no real idea how to do this. We use instead an iterative generation process with more specific guidelines. While the approach we use cannot sample the whole probability space uniformly, it is easy to implement and modify.

1.3 Our take on creating models as random as possible

To generate random, desirable models we use the following procedure

1. Add a new state s' to the graph
2. Chose at least one s that was created before s'. Add an action to s that has a random probability greater than 0 of reaching s'.
3. While the probability of the action is not 1, extend the action by a transition to a state not yet reached by the action.

This ensures that the graph is connected and there are no states to which there is no path from the initial state. Note that since s is created before s' there are no end components. To provide an option of generating end components, we iterate from the last state to the first and randomly decide on whether they should have additional actions leading to randomly selected states. Thus, end components may appear but are not guaranteed to exist.

This procedure can create every model in the desirable model space since states may have arbitrary many actions with arbitrary transition probabilities to every state. <- Maybe this is worth a proof? I feel like this is true, but Im not 100% sure yet. However, generating models this way does not sample the space uniformly, since the number of transitions per action is sampled non-uniformly. Thus, actions tend to have rather few transitions. Also, states with smaller indices tend to have more actions than states with higher indices.

Fixing Properties

In some cases, it is interesting to compare algorithms on models that have several properties in common. By fixing a property we have more insight on the structure of the model and can easier analyse why one algorithm is superior to another in the given scenario. If for example model A has more states, more and bigger MECs than model B, it is hard to relate the performance difference of an algorithm on model A and B to a specific property. [What I want to say: Fixing more feature-dimensions leaves fewer feature dimensions that can vary -> ideally leaving only one feature-dimension with variance. This way we can precisely oberserve how changes in that feature-dimension affect performance of the algorithms]

While we can fix properties like the number of states in our generation-procedure there are others we cannot control like the number or size of MECs or at least not in an easy way.

If we are interested in fixing such properties, we use other procedures that restrict the desirable model space further but provide control over certain parameters. Procedures we have come up with are:

- **RandomTree** A treelike graph-structure where the initial-state is the root. In this procedure we can easily set the number of actions every state has.
- **RandomMEC** Create m subgraphs of desired size randomly. Force them to be MECs by inserting actions accordingly. Sort them, assign to each MEC a representative. Now we have m representatives which we can connect similar to our random-procedure. Can handle size and number of MECs
- **RandomSEC** Create m subgraphs by our random-procedure of desired size randomly. Per subgraph find each SEC and connect them, resulting in one SEC per subgraph. Sort the subgraphs, assign to each a representative. Now we have m representatives which we can connect similar to our random-procedure. Can handle size and number of SECs

The different guidelines are helpful because we will see in the experimental section that their structural differences have a huge impact on the performance of our algorithms. With just a purely random procedure this claim is harder to visualize.

1.4 Adding Prefix-Graphs / Configurations

This is rather an implementation thing than an abstract thing, so maybe I should not put it into this section? Since the purpose of random generated models is to same from a space as big a possible, we have per se few insights on the structure of the model, even after fixing properties. Thus, this method is not very well suited if one wants to analyse behaviour of an algorithm on very specialized models. Traditionally, this is where one would build handcrafted models. However, handcrafting parameterizable prism files allows only for very restrictive structural changes in the models. Therefor, we have implemented the option to create models at runtime. These models can also be prepended to other models to evaluate how much impact a certain structure has if added to a model.

2 Algorithm Analysis

To facilitate the algorithm performance-analysis, we track statistics about the algorithm like the time it requires to solve the problem, the iterations needed in case we use a value-iteration-based algorithm and the value it has computed for correctness checks. Furthermore, we track features of the model to relate algorithm-performance to model properties. The features we track are [...].

Lastly, we use various data mining tools to visualize, simplify and analyse the collected data.

[HOW MANY IMPLEMENTATION DETAILS SHOULD THERE BE?]

3 Implementing Random-Generated Graphs

3.1 Implementation vs Theory

Here we say how our implemention works and what the differences are in comparison to the theoretical graph generation we introduced in Chapter 1.

3.2 A manual on how to use our implementation

About one page long. Mainly a documentation page.

4 Conducting the Analysis / Results

Regarding runtime a lot of stuff may change. Iterations wont change as long as we use the same deterministic algorithms, so its a more stable metric.

Various studies we have made, knit into a nice purple string and story. Here might appear:

- TOP was not as good as we have hoped
- SI with Linear Programming for MDPs is good and we dont know why
- Hopefully some rules of thumb on when to use which algorithm that could be a precursor to portfolio solving

Maybe it would also be cool to show preformance differences in real case studies in comparison to the random generated models and make conclusions Ideally: We did previously not have enough models to see behaviour XYZ but now we can.

5 Conclusion

We have created an arbitrarily big or small benchmarking-suite for stochastic games that could be profitable for the whole stochastic game community. With the random models and our analysis tool we cool derive some interesting property of algorithm X and found out Y about algorithm Z.