How to Reproduce Results on State Aggregation Combined with Options

Kamil Ciosek
Centre for Computational Statistics and Machine Learning
University College London
Gower Street, WC1E 6BT London

Abstract

This note summarizes the steps that have to be taken to reproduce our results on combining value iteration with options with state aggregation. It describes the MATLAB source code used to obtain results discussed in our paper, as well as gives the output of the software on three separate computers. It also explains how to extend our code to work with other domains.

1 Hardware and software requirements

We recommend running this software on a computer with at least 2GB of RAM and a processor running at at least 2Ghz. The computer has to be running a recent version of MATLAB (the earliest one we tested was R2012a). The Java heap space allocated to MATLAB must be at least 256MB. In recent versions of MATLAB, this can be set by clicking the 'Preferences' button on the 'Home' tab, then navigating to 'General' and then 'Java Heap memory'. Please note that the default amount that comes when you install MATLAB will typically be less, so you have to change this. We do not support any version of Octave, because it currently does not share MATLAB's support of classes and other object-oriented constructs.

2 How to run the software

The entry point to this software is the file `run.m`, which is a script that runs all the experiment referred to in our paper. You can run the software by unpacking the supplied zip archive, changing the current directory to the OptionsAggregation folder and then typing run in the MATLAB console. Please note that executing this file to the end may take some time (a few hours).

3 Structure of the source code

There are three classes that contain methods that provide a high-level access to the described algorithms, one for each problem domain. These are `TaxiRunner`, `HanoiRunner` and `EightPuzzleRunner`. The methods in these classes then call other functions to finally compute the value function.

There are two pieces of code responsible for the iteration proper. These are the function `iterateAll`, which iterates the algorithm and the class `OptimizePolicyAndTerminationNestedIteration`, which contains the code for the particular kind of two-stage iteration employed in our paper.

The code for generating the subgoal features is contained in classes whose name ends with `Features`. These classes have a method for generating the `Phi` matrix and the `G` vector. Comments in these classes explain the particular kind of approximation employed. For the eight-puzzle problem, there are as many as 15 possible subgoals, which are mapped to the appropriate classes in the function `EightPuzzleRunner.generateSubgoal`. In figure 1, we graphically show all subgoals referred to in this function.

4 How to use our algorithm in a new domain

Before we begin, we note that the current version of our software only supports discrete domains where the number of states is manageable (we feel about 200000 states is the maximum that can be currently handled in the deterministic setting, and about 400 states in the completely non-deterministic setting – i.e. where the transition matrices are completely full). Therefore the new domain has to meet these criteria.

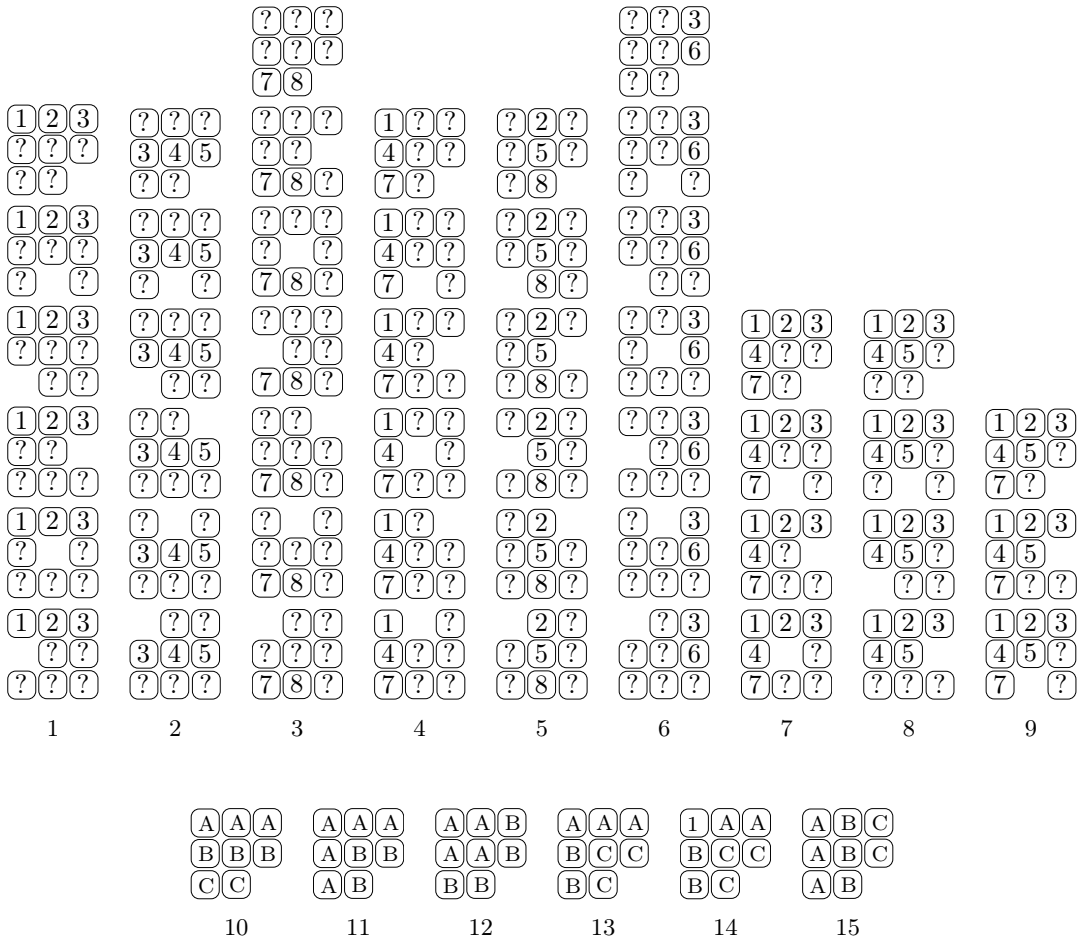

Figure 1: Subgoals used in the 8-puzzle domain

The first thing one needs to do is to write a function (or a class) that is capable of generating action models for the new domain. Our system stores actions in cell arrays, so that for instance in the Hanoi problem the `As` vertical cell array has three elements, each a model of size $(n+1) \times (n+1)$ where n is the number of states in the MDP. The function `packModel` can be used to convert a vector of expected rewards and a transition matrix into a valid model. That being done, it should be easy to implement value iteration by following steps entirely analogous to the function `TaxiRunner.vi`.

The next step is to add state abstraction with options. We will describe the case where we have only one subgoal. In this case, we need two things. First, we need a function or a class to generate the aggregation matrix `Phi`, where the rows correspond to original system states and the columns to aggregate states. Then, we also need a function to generate the subgoal `G` in terms of the aggregate states. Once we have these things, it is easy to write code that first solves for the subgoal and then solves the main goal using the subgoal model, based on code in the function `EightPuzzleRunner.optionsAggregation`.

5 Results of three independent runs

The tables below summarize the results obtained by running the `run.m` script on three separate computers. We begin with the TAXI domain. The 'Iterations' column refers to the iterations solving the main subgoal (we do not give the number of iterations required to solve the subgoals). The time columns refer to the total time necessary to compute the value function, which includes the generation and solution of subgoals. All times are given in seconds. We note the difference between value iteration and plain value iteration – the first one uses matrix models representing the current policy, whereas the plain version only stores the current value function. The number of iterations for the two may be different because of different initialization (model value iteration is initialized with the first action, plain value iteration is initialized with a zero value function).

Problem	Iterations	Time (1)	Time (2)	Time (3)
Deterministic Problem				
Value Iteration (models)	22	11.64	18.36	21.06
Value Iteration (plain)	22	6.43	9.79	9.91
Options	14	78.20	121.06	136.70
Aggregation	19	11.73	18.12	20.11
Options + Aggregation (models)	7	6.55	10.42	11.22
Options + Aggregation (plain)	7	4.57	7.08	7.30
Approx. aggregation (models)	28	4.83	8.29	9.36
Approx. aggregation (plain)	28	2.94	4.92	5.17
Non-deterministic Problem				
Value Iteration (models)	30	47.80	67.40	60.19
Value Iteration (plain)	30	8.30	12.36	12.69
Options	18	256.04	351.73	342.48
Aggregation	28	26.04	38.65	38.02
Options + Aggregation (models)	7	6.78	10.83	11.20
Options + Aggregation (plain)	7	4.83	7.60	7.52
Approx. aggregation (models)	31	5.23	8.82	10.06
Approx. aggregation (plain)	30	3.08	5.22	5.29

The next table gives the results in the Towers of Hanoi domain (with 8 disks).

Problem	Iterations	Time (1)	Time (2)	Time (3)
Deterministic Problem				
Value Iteration (mat. models)	256	51.65	81.49	93.83
Value Iteration (plain)	257	23.45	35.75	39.47
Options with Aggregation	4	11.57	18.97	19.61
Non-deterministic Problem				
Value Iteration (mat. models)	296	357.52	335.32	413.99
Value Iteration (plain)	297	27.31	41.28	45.32
Options with Aggregation	10	21.71	34.64	36.57

Next, we show results for the (deterministic) eight-puzzle domain.

Problem	Iterations	Time (1)	Time (2)	Time (3)
Value Iteration (mat. models)	32	221.20	341.89	395.97
Value Iteration (plain)	33	100.19	155.02	166.87
Options (subgoals 1,2,3)	24	162.52	249.84	269.91
Options (subgoals 1–6)	22	232.39	353.24	386.50
Options (subgoals 1–8)	18	602.71	875.35	917.75
Options (subgoals 7,8)	20	462.91	665.36	696.36
Options (subgoal 7)	26	282.31	411.19	433.51
Options (subgoal 8)	28	299.90	439.42	459.67
Options (subgoal 9)	5	1940.35	1910.67	2422.08
Options (subgoal 10)	25	109.51	169.22	182.50
Options (subgoal 11)	32	130.55	202.01	217.57
Options (subgoal 12)	29	119.33	235.65	198.39
Options (subgoal 13)	32	136.80	210.06	226.88
Options (subgoal 14)	32	153.25	236.72	253.97
Options (subgoal 15)	26	113.65	180.25	189.01
Options (use 1,4 to learn 7, use 1 to learn 8)	20	720.20	1005.39	1033.77
Options (subgoal 10, horizon 9)	25	85.94	131.71	146.08