Integrating MDPs into Gujarat Simulation

Miquel Ramírez

April 18, 2012

Summary

This document describes how Monte–Carlo Tree Search techniques have been introduced into the Gujarat Simulation in order to control hunter gatherer agents. New concepts introduced into the framework will be introduced, along with the new classes and interfaces which encode these concepts and the changes made in existing code in order to accomodate them into the existing system. Implementation notes will be offered which will cover those aspects critical for performance and soundness. The last section in this document contains a To–Do list, which should be taken as the roadmap for future steps.

Contents

1 Monte–Carlo Tree Search and MDPs

Given the complexity of the Gujarat agent–based simulation, we have discarded out of hand model–based approaches to solve MDPs, as the ones discussed in the classical Optimal Foraging Theory literature. Rather than attempting at computing a value function V, and therefore a policy π_V, out of the MDP model M, either with *model–based* algorithms such as Value Iteration, we have aimed at obtaining an on–line *action selection* mechanism, which *implicitly* defines a policy π for MDP M.

These family of algorithms, collectively known as *simulation–based*, do not require a full model of the MDP M but it rather suffices that the algorithm has access to a simulation of M which allows it to *sample* the state space and the outcomes of actions. In the context of the Gujarat simulation, this amounts to provide an interface so that relevant parts of the agent–based simulation are made available to a component simulating the MDP M. So far, this has proved to be a relatively easy task.

2 New Concepts, Classes & Interfaces Introduced

We have taken Blai Bonet's `libmdp` library concepts and implementation of algorithms off–the–shelf and integrated it into the Gujarat simulation, including suitably defined implementations of the interfaces in `libmdp` for the concepts of STATE and MODEL and creating a *proxy* interface AGENTCONTROLLER that allows to abstract away the details of how agents in the simulation choose actions.

When this mechanism relies on action selection algorithms for MDPs, the concrete controller amounts to wrap together the MDP model object, which is sort of an *observer* on GUJARATAGENT objects, with the selected simulation–based MDP algorithm. In the current implementation we are using the implementation of UCT available on `libmdp`.

Finally, a class encapsulating the parameters which can be tweaked from the simulation XML configuration document, has been introduced. This allows to put some order into the the sorry mess that the GUJARATCONFIG class is becoming as we progress.

A general overview of the relationships and interaction of the new components between them and with already existing components such as GUJARATAGENT or GUJARATWORLD is depicted in Figure 2.

2.1 MDP Configuration

This class implementation resides on the files:

- `HunterGathererMDPConfig.hxx`
- `HunterGathererMDPConfig.cxx`

For now, we are exposing four parameters:

- *Number of* FORAGE *actions* – This allows to further restrict the number of FORAGE actions to be considered beyond the restriction inherent in the number of sectors which model the Hunter–Gatherer agent perception of

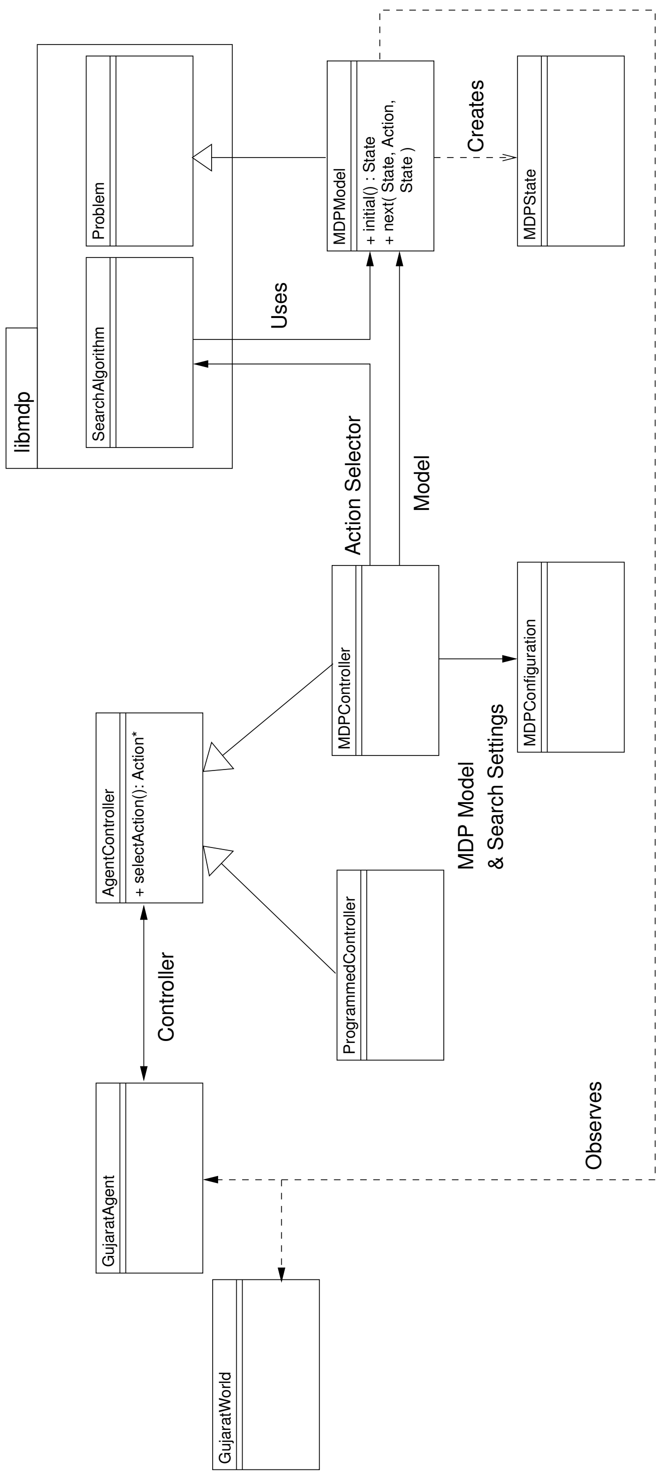

Figure 1: Overview of the integration between the Gujarat simulation and `libmdp`.

her environment. Since we are already ranking these FORAGE actions, I doubt that every action is actually required to be evaluated. This should allow us to restrict the branching factor of the state space and avoid performing unnecessary computation. This value **cannot** be bigger than the number of sectors specified in the Hunter–Gatherer configuration.

- *Number of* MOVEHOME *actions* – Similarly to the above, the ranking we are doing on SETTLEMENTAREAS opens up the possibility of more aggressive simplification. The value of this parameter is not constrained by any other parameters in the XML configuration. If this value is higher than the actual number of SETTLEMENTAREAS found to be relevant for an agent during the simulation, no additional "virtual" actions are generated.
- *Do Nothing Is Allowed* – While we are pretty confident on the FORAGE and MOVEHOME actions, the issue raised by M. Madella regarding hunter–gatherers – in the real world – not foraging or moving around all the time is valid. However, we have very little meaning attached to this action. We don't really have any *incentives* in the simulation for the agent to do this, that is, to do nothing at all. Perhaps this could model other activities such as mending hunting equipment. But it also sort of overlaps with the yet–to–be–introduced *social* activities we have in our draft design document.
- *Horizon* – The number of time steps in the MDP M. This refers to the notion of the agent deliberating, by means of projecting itself and the outcome of its future actions forward in time. In principle, the bigger this horizon T is, the *better* informed will be the decisions the agent makes. However, this comes at a hefty computational cost. On the other hand, this parameter can also model cognitive limitations inherent to human decision–making.

2.2 MDP State

This class implementation resides on the files:

- `HunterGathererMDPState.hxx`
- `HunterGatehrerMDPState.cxx`

This is a subclass of `libmdp` STATE interface. Initial states s_0 are a *view* on the Gujarat simulation state when an action is requested from an agent at simulation step t. Three datums are extracted from the simulation state:

- The *resources* dynamic raster state at simulation step t.
- The location on the map of the agent.
- The amount of calories the agent has *on hand* at simulation step t.

The reason for these datums to be extracted is that these are the elements of the simulation state which are *directly* modified by the three actions we are considering for Hunter–Gatherer agents, namely, FORAGE, MOVEHOME and

DoNothing. States s resulting from applying any action a, can be formally defined as follows

$$\langle R, \vec{l}, C, d \rangle$$

where R is the resource raster resulting from applying any actions, $\vec{l}$ is the location of the agent, C is the number of calories the agent has *on hand* and d is the associated time index. While $\vec{l}$ and C are very light objects – primitive types or collections of such – the resources raster is a potentially huge collection of values[1]. This involved introducing the class IncrementalRaster, a subclass of Raster which is discussed on Section 4.1. Another important issue is that of *hashing* states, important to detect "duplicate" states, that it, states resulting from actions a and b at time step d which result in the same value for R, $\vec{l}$ and C. Hashing is discussed in Section 4.2.

States are *terminal states* whenever the value of the time index equals that of the horizon, $d = T$. For now we are leaving out the possibility of the agent dying or emigrating during the next T simulation time steps[2].

I have decided to store the actions applicable to an state s in the State object itself as it is the most compact way to deal with state spaces where states may have very different numbers and kinds of actions being applicable. This adds some complexity to the implementation of actions (see Section 3.1) and that of the State class itself, since copy constructors need to be implemented carefully and make sure that copies of State objects keep valid pointers to action objects at all times.

2.3 MDP Model

The implementation of the class encapsulating the notion of Mdp model or *problem* can be found on this two files:

- `HunterGathererMDPModel.hxx`
- `HunterGathererMDPModel.cxx`

This class has two roles. On the one hand, it fulfills the *Observer* pattern, by associating itself with one of the Hunter–Gatherer agents currently active in the simulation. On the other hand, it fulfills the *Facade* pattern, providing the action selection algorithm with access to states and the transition function. We will devote some space to discuss successor state generation.

Whenever an action is executed on a State object, representing an Mdp state s, the `next()` method is invoked in order to obtain a second State object, which corresponds to one state $s' \in F(s,a)$, where $F(\cdot)$ is a non–deterministic transition function. In our case, $F(\cdot)$ is implemented on top of the same elements as actions are *actually* executed on simulation states. This function is not deterministic since the Forage action effects include *chance* outcomes, such as the actual amount of biomass retrieved from cells in a given Sector.

Generating s' consists of three steps, which are implemented in different places:

[1] Right now we are working with a "toy" 400x400 grid, which is already rather big.

[2] Actually this is an application of the "optimism in the face of uncertainty" heuristic. Note that heuristics can be either used *implicitly* during modeling, that is, the modeller "hardcodes" the heuristic, or made *explicit* by analyzing the structure of some given model so it can be exploited computationally.

1. The State class, which has the responsability of properly initializing s' attributes in terms of the attributes of s.

2. Concrete Action class, which modifies s' attributes as necessary to account for the effect of the action.

3. The Model class, which access the Hunter–Gatherer agent object state in order to access static data – such as SettlementAreas and Sectors – in order to build and attach to s' the action instances corresponding to the set of available actions to do in s'.

Another important, and underdeveloped, part of the implementation of the Model class is that of the `cost()` method. `libmdp`, rather than working on the "maximizing reward" formulation for Mdps, works on that of "minimizing costs". Both formulations are equivalent, and each of them are more intuitive in different settings. In the setting of the Gujarat Simulation, this involves doing a linear transformation on actual rewards, so minimizing costs corresponds exactly with maximizing reward. Right now, the definition of costs of actions $c(a, s)$ is rather simple:

$$c(a, s) = K - C(s) + t(a)$$

where $C(s)$ is the amount of on–hand calories in state s, $t(a)$ is the "time" needed to execute the actions, and K is an arbitrarily big constant, so that reaching states s with very low on–hand calories have associated very high costs. Both K and $C(s)$ need to be calibrated (they go together) so that we can keep the number of states low and their magnitudes are in line with the numbers we actually use in the simulation.

2.4 Agent Controllers

The AgentController interface can be found on the following file:

- `AgentController.hxx`

Classes implementing AgentController encapsulate the code we used to have inside `evaluateIntraSeasonalActions` and the like. Introducing this proxy between GujaratAgent and the actual action selection mechanisms allows us to decouple the latter from the former, and offers to us a greater degree of flexibility when it comes to defining the control mechanisms that generate the agent behavior, for instance, several alternative hand–coded policies, or generic action selection algorithms working on top of alternative Mdp formulations of the simulation.

2.4.1 Programmed Controllers

The random action selection policy we used to have inside the Hunter–Gatherer class `evaluateIntraSeasonalActions` method is now encapsulated by the class implemented in the following two files:

- `HunterGathererProgrammedController.hxx`

- `HunterGathererProgrammedController.cxx`

2.4.2 MDP Controller

The agent controller that results from applying a generic action–selection algorithm on our first MDP formulation implementation can be found in the following two files:

- `HunterGathererMDPController.hxx`
- `HunterGathererMDPController.cxx`

Instances of this class aggregate three kinds of objects:

- MODEL instances which are associated with one single agent.
- A `libmdp` *base policy* (currently fixed to the random policy).
- A copy of the configuration parameters relevant for MDP–based controllers.

The `selectAction()` method is called by `evaluateIntraSeasonalActions` in order to retrieve the best action according to the generic action–selection algorithm used (currently fixed to UCT). This method implementation is relatively simple and consists of:

1. *Resetting* the MODEL instance, which amounts to pulling from the simulation state the datums necessary to construct the initial state s_0 of the MDP.
2. Creating a temporary UCT policy object, π_{uct}, initialized according to the parameters in the configuration.
3. Invoking the policy object on the initial state to obtain the action to be executed.
4. Copying the action[3].

The copy of the best action is then returned to the controlled Hunter–Gatherer agent, so it can be forwarded to GUJARATWORLD action execution scheduler.

3 Changes in Existing Interfaces

Eisting classes and interfaces have required substantial changes or the addition of new features, and the nature of these changes and new features is discussed in this Section.

[3]The owners of action objects created during the process of analyzing the MDP model are the STATE instances. Therefore, once we are finished with selecting an action, we need to copy it, since any action objects previously created will be disposed of along with their owning states.

3.1 Changes in Action Interface and Sub–Classes

The Action interface has been changed so that:

- A new `execute()` overload has been added, to account for executing the action on Mdp states rather than simulation states (via GujaratAgent instances).
- Implementation of concrete actions has been changed to ensure that the least possible amount of code is duplicated between the two versions of the `execute()` method.
- A new `copy()` method is required from classes implementing the Action interface. This method creates a perfect duplicate of the concrete Action object it is invoked on.
- A new class implementing the DoNothing action has been added.

The Forage action has been changed so that:

- Now Sector objects might or might not be owned by the Forage class. Sector instance ownership is required for Forage actions required during the action–selection algorithm execution.
- New `protected` methods have been added to encapsulate the common behavior between both `execute()` versions. There is still some duplicate behavior, though, between both flavors of `execute()`, which should be easy to remove.

The MoveHome action has been changed so that:

- Now action generation and execution is totally separated (it wasn't previously, to my surprise).
- A new overload for `generatePossibleActions` has been added, to generate the MoveHome actions available to Mdp states. Note that we still need a reference to the GujaratAgent instance in order to access the (static) SettlementArea objects.

3.2 Changes in GujaratAgent Sub–Classes and GujaratWorld

- The Mdp controller configuration is now properly loaded (and there's some degree of reasonable error–handling).
- Some pieces of mis-placed functionality such as computation of consumed calories, actual returns from biomass, etc. are now in a more meaningful place.
- The `updateKnowledge()` method is now split into two versions: one which modifies the sectors attribute, the other operates on a vector of Sector objects (and is meant to be used by Model sub–classes).

3.3 Changes in Pandora

3.3.1 Intervention on the Raster Class Hierarchy

The most important change in this class hierarchy is the introduction of IncrementalRaster, implemented on files:

- `IncrementalRaster.hxx`
- `IncrementalRaster.cxx`

IncrementalRaster is a *virtual* raster: it does not represent a raster, but rather the *changes* done on some *DynamicRaster*[4]. By overwriting `setValue()` and `getValue()` is quite easy to keep changes into an auxiliary data structure[5], and reading values which have been changed from it.

The reason for introducing IncrementalRaster is that concrete Mdp State objects have the *resources* raster as one of its attributes. Since many state objects might be generated by the action selection algorithm in order to evaluate actions, it was critical to keep low the cost of generating successors. The straight-forward approach would have consisted in copying – verbatim – a DynamicRaster, which involves a hefty computational burden.

While it might well be the case that the number of entries in the IncrementalRaster becomes the same as that of cells than that of the DynamicRaster it is based upon, this depends on the length of the planning horizon T and the actual biomass values generated by the simulation, and should be a rare occurrence.

4 Notes

4.1 Exposing the Simulation State to Agent Controllers

Applying state–of–the–art planning algorithms into "real world" systems, and the Gujarat Simulation is quite "real world" from the perspective of Ai research, usually involves accessing and working on complex data structures. Even harder to deal with is the fact that the domains of the fields of such data structures is not explicitly *bounded* to some reasonable size.

A naïve approach to define the notion of *state* in the context of an object–oriented application would be to use those same data structures as the "meat and potatoes" of *state* objects. However, this approach fails as soon as such data structures get too far from the propositional representations usually used for clarity and simplicity when developing planning algorithms, which certainly is the most common situation.

For the Gujarat Simulation that means that the Mdp models we use need to use more abstract data structures[6]. For now, we only have a case study, that of the Hunter Gatherer Mdp model State class, but I do think that a more comprenhensive solution can be found, so that the simulation can expose with

[4]The Raster class should be renamed some day

[5]Currently a `std::map`, I'm not too happy with the $O(n \log n)$ asymptotic cost of accessing elements in the worst case, but it will do for the time being.

[6]And it could be argued that programmed agents would also benefit from more abstract data structures, to avoid both tight coupling with code prone to change, and to avoid excessive specifity.

a uniform interface relevant parts of its internal state to agent controllers to process.

4.2 Hashing

A quite delicate issue when writing the concrete State class for the Hunter–Gatherer Mdp model was that of how to compute a robust hashing function. Most planning algorithms need to keep track of states which have already been visited, either because of efficiency or because the need to update the values associated to state and action pairs.

This could have been addressed, rather naïvely, by computing the hash function directly upon all raster's cells values. However, this would have been a huge waste of computation, since the rasters associated to states s potentially differ very little from the one inside the initial state s_0. Keeping track of changes with respect a base raster allows us to compute a good hash function efficiently, by limiting the amount of data to be processed.

4.3 The Importance of Constness

While this might sound as nit–picking, the `const` keyword is quite important for the sake of efficiency. It provides us with a very powerful tool to help the compiler to generate very efficient code, since it gives it a good hint about how to proceed with uploading and downloading data from main memory to the cache, and from the cache to the register bank. Besides that, proper usage of `const` forces us to "put the data into the right object" and encourages the use of accessors to get a hold of object attributes[7].

5 TO-DO list

This is a quite miscellaneous list, which includes stuff both related or not to the Mdp integration, and that I have been spotting during this work:

- Calibrate calories consumed by simulation time step with calories as obtained from biomass exploitation. Check that the agent calories reserve is in the right units and always the same attribute is being used.
- SettlementAreas implementation needs to be revised. Looks to me as too messy and departs slightly from the coding standards used elsewhere.
- "Frame" effects – the passage of time, calories consumption of agent's people, etc. – are now implemented multiple times across concrete actions. A way to encapsulate these, and to relate them to the Mdp model, is needed.
- Resource raster values range is far too detailed. A "compressed" representation of calories intakes and availabilities would make much easier for the planning algorithm to explore quicker the state space. For instance,

[7]A rather problematic issue appears when attributes are accessed "on the rocks": I have had some trouble replacing attributes by computations derived from the value of the attribute. The temptation to duplicate code is very strong sometimes, but there's really no excuse for laziness :-)

we could use a representation which works on "levels", each level corresponding to the caloric requirements of the agent at the simulation time step t.

- Start some "realistic" experiments to see how things work out. Especially important is to see that the agents have a decent change of surviving for some time. A protocol for generating these simulations should be discussed. Automating this, so we can effectively assess changes in behavior, would be a plus (and something required for the Validation Work Package).

- Write a decent programmed policy. The random programmed policy we have now is just, unsurprisingly, very poor. Mass extinction should be the exception, not the rule.

- The ability to pass the XML configuration file as an argument to the simulation is a very useful feature which is surprisingly still missing.

- A comprenhensive event logging system is sorely needed. The current policy of dumping messages on `std::cout` is totally useless as debugging messages from many different modules get mixed up.