MICMAC MANUAL

Contents

[in package MICMAC]

- **[system]** **"micmac"**

```
- _Version:_ 0.0.2
- _Description:_ Micmac is mainly a library of graph search algorithms
    such as alpha-beta, UCT and beam search, but it also has some MCMC
    and other slightly unrelated stuff.
- _Licence:_ MIT, see COPYING.
- _Author:_ Gábor Melis <mega@retes.hu>
- _Mailto:_ [mega@retes.hu](mailto:mega@retes.hu)
- _Homepage:_ <http://melisgl.github.io/mgl-gpr>
- _Bug tracker:_ <https://github.com/melisgl/mgl-gpr/issues>
- _Source control:_ [GIT](https://github.com/melisgl/mgl-gpr.git)
- *Depends on:* alexandria, [mgl-pax][6fdb]
```

1 Introduction

1.1 Overview

MICMAC is a Common Lisp library by Gábor Melis focusing on graph search algorithms.

1.2 Links

Here is the official repository and the HTML documentation for the latest version.

2 Graph Search

- **[function]** **alpha-beta** *state &key (depth 0) alpha beta call-with-action maybe-evaluate-state list-actions record-best*

  Alpha-beta pruning for two player, zero-sum maximax (like minimax but both players maximize and the score is negated when passed between depths). Return the score of the game `state` from the point of view of the player to move at `depth` and as the second value the list of actions of the principal variant.

  `call-with-action` is a function of `(state depth action fn)`. It carries out `action` (returned by `list-actions` or `nil`) to get the state corresponding to `depth` and calls `fn` with that state. It may destructively modify `state` provided it undoes the damage after FN returns. `call-with-action` is called with `nil` as `action` for the root of the tree, in this case `state` need not be changed. `fn` returns the same kinds of values as `alpha-beta`. They may be useful for logging.

  `maybe-evaluate-state` is a function of `(state depth)`. If `state` at `depth` is a terminal node then it returns the score from the point of view of the player to move and as the second value a list of actions that lead from `state` to the position that was evaluated. The list of actions is typically empty. If we are not at a terminal node then `maybe-evaluate-state` returns `nil`.

  `list-actions` is a function of (`state depth`) and returns a non-empty list of legal candidate moves for non-terminal nodes. Actions are tried in the order `list-actions` returns them: stronger moves

  `call-with-action`, `maybe-evaluate-state` and `list-actions` are mandatory.

  `record-best`, if non-`nil`, is a function of (`depth score actions`). It is called when at `depth` a new best action is found. `actions` is a list of all the actions in the principle variant corresonding to the newly found best score. `record-best` is useful for graceful degradation in case of timeout.

  `alpha` and `beta` are typically `nil` (equivalent to -infinity, +infinity) but any real number is allowed if the range of scores can be boxed.

  See `test/test-alpha-beta.lisp` for an example.

- **[function]** **beam-search** *start-nodes &key max-depth (n-solutions 1) (beam-width (length start-nodes)) expand-node-fn expand-beam-fn score-fn upper-bound-fn solutionp-fn (finishedp-fn solutionp-fn)*

  In a graph, search for nodes that with the best scores with beam search. That is, starting from `start-nodes` perform a breadth-first search but at each depth only keep `beam-width` number of nodes with the best scores. Keep the best `n-solutions` (at most) complete solutions. Discard nodes known to be unable to get into the best `n-solutions` (due

to upper-bound-fn). Finally, return the solutions and the active nodes (the *beam*) as adjustable arrays sorted by score in descending order.

start-nodes (a sequence of elements of arbitrary type). score-fn, upper-bound-fn, solutionp-fn, finishedp-fn are all functions of one argument: the node. solutionp-fn checks whether a node represents a complete solution (i.e. some goal is reached). score-fn returns a real number that's to be maximized, it's only called for node for which solutionp-fn returned true. upper-bound-fn (if not nil) returns a real number that equal or greater than the score of all solutions reachable from that node. finishedp-fn returns true iff there is nowhere to go from the node.

expand-node-fn is also a function of a single node argument. It returns a sequence of nodes to 'one step away' from its argument node. expand-beam-fn is similar, but it takes a vector of nodes and returns all nodes one step away from any of them. It's enough provide either expand-node-fn or expand-beam-fn. The purpose of expand-beam-fn. is to allow more efficient, batch-like operations.

See test/test-beam-search.lisp for an example.

- **[function]** **parallel-beam-search** *start-node-seqs &key max-depth (n-solutions 1) beam-width expand-node-fn expand-beams-fn score-fn upper-bound-fn solutionp-fn (finishedp-fn solutionp-fn)*

  This is very much like beam-search except it solves a number of instances of the same search problem starting from different sets of nodes. The sole purpose of parallel-beam-search is to amortize the cost expand-beam-fn if possible.

  expand-beams-fn is called with sequence of beams (i.e. it's a sequence of sequence of nodes) and it must return another sequence of sequences of nodes. Each element of the returned sequence is the reachable nodes of the nodes in the corresponding element of its argument sequence.

  parallel-beam-search returns a sequence of solutions sequences, and a sequence of active node sequences.

  See test/test-beam-search.lisp for an example.

2.1 UCT

[in package MICMAC.UCT]

uct Monte Carlo tree search. This is what makes current Go programs tick. And Hex programs as well, for that matter. This is a cleanup and generalization of code originally created in course of the Google AI Challenge 2010.

For now, the documentation is just a reference. See test/test-uct.lisp for an example.

- **[class]** **uct-node**

  A node in the uct tree. Roughly translates to a state in the search space. Note that the state itself is not stored explicity, but it can be recovered by 'replaying' the actions from the starting state or by customizing make-uct-node.

- **[reader]** **depth** *uct-node (:depth = 0)*
- **[accessor]** **edges** *uct-node*

  Outgoing edges.
- **[accessor]** **average-reward** *uct-node (:average-reward = 0)*

  Average reward over random playouts started from below this node. See `update-uct-statistics` and REWARD.
- **[class]** **uct-edge**

  An edge in the `uct` tree. Represents an action taken from a state. The value of an action is the value of its target state which is not quite as generic as it could be; feel free to specialize `average-reward` for the edges if that's not the case.
- **[reader]** **action** *uct-edge (:action)*

  The action represented by the edge.
- **[accessor]** **from-node** *uct-edge (:from-node)*

  The node this edge starts from.
- **[accessor]** **to-node** *uct-edge (= nil)*

  The node this edge points to if the edge has been visited or `nil`.
- **[function]** **visited-edges** *node*
- **[function]** **unvisited-edges** *node*
- **[generic-function]** **edge-score** *node edge exploration-bias*
- **[generic-function]** **select-edge** *node exploration-bias*

  Choose an action to take from a state, in other words an edge to follow from `node` in the tree. The default implementation chooses randomly from the yet unvisited edges or if there is none moves down the edge with the maximum `edge-score`. If you are thinking of customizing this, for example to make it choose the minimum at odd depths, the you may want to consider specializing REWARD or `update-uct-statistics` instead.
- **[generic-function]** **outcome->reward** *node outcome*

  Compute the reward for a node in the tree from `outcome` that is the result of a playout. This is called by the default implementation of `update-uct-statistics`. This is where one typically negates depending on the parity of `depth` in two player games.
- **[generic-function]** **update-uct-statistics** *root path outcome*

  Increment the number of visits and update the average reward in nodes and edges of `path`. By default, edges simply get their visit counter incremented while nodes also get an update to `average-reward` based on what `outcome->reward` returns.

- **[generic-function]** **make-uct-node** *parent edge parent-state*

  Create a node representing the state that `edge` leads to (from `parent`). Specialize this if you want to keep track of the state, which is not done by default as it can be expensive, especially in light of TAKE-ACTION mutating it. The default implementation simply creates an instance of the class of `parent` so that one can start from a subclass of `uct-node` and be sure that that class is going to be used for nodes below it.

- **[generic-function]** **state** *node parent edge parent-state*

  Return the state that corresponds to `node`. This is not a straightforward accessor unless `node` is customized to store it. The rest of the parameters are provided so that one can reconstruct the state by taking the action of `edge` in the `parent-state` of `parent`. It's allowed to mutate `parent-state` and return it. This function must be specialized.

- **[generic-function]** **list-edges** *node state*

  Return a list of edges representing the possible actions from `node` with `state`. This function must be customized.

- **[generic-function]** **play-out** *node state reverse-path*

  Play a random game from `node` with `state` and return the outcome that's fed into `update-uct-statistics`. The way the random game is played is referred to as 'default policy' and that's what makes or breaks `uct` search. This function must be customized.

- **[function]** **uct** *&key root fresh-root-state exploration-bias max-n-playouts*

  Starting from the `root` node, search the tree expanding it one node for each playout. Finally return the mutated `root`. `root` may be the root node of any tree, need not be a single node with no edges. `fresh-root-state` is a function that returns a fresh state corresponding to `root`. This state will be destroyed unless special care is taken in `state`.

3 Metropolis Hastings

[in package MICMAC.METROPOLIS-HASTINGS with nicknames MICMAC.MH]

Generic interface for the Metropolis-Hastings algorithm, also Metropolis Coupled MCMC.

References:

- http://en.wikipedia.org/wiki/Metropolis–Hastings_algorithm
- Markov Chain Monte Carlo and Gibbs Sampling Lecture Notes for EEB 581, version 26 April 2004 c B. Walsh 2004 http://web.mit.edu/~wingated/www/introductions/mcmc-gibbs-intro.pdf
- Geyer, C.J. (1991) Markov chain Monte Carlo maximum likelihood

For now, the documentation is just a reference. See `test/test-metropolis-hastings.lisp` for an example.

- **[class]** **mc-chain**

A simple markov chain for Metropolis Hastings. With temperature it is suitable for `mc3`.

- **[accessor]** **temperature** *mc-chain (:temperature = 1.0d0)*

  The PROBABILITY-RATIO of samples is raised to the power of 1 / `temperature` before calculating the acceptance probability. This effectively flattens the peaks if `temperature` > 1 which makes it easier for the chain to traverse deep valleys.

- **[reader]** **state** *mc-chain (:state)*

  This is the current sample where the chain is.

- **[function]** **jump-to-sample** *chain jump &key (result-sample (state chain))*

  From the current state of `chain` make `jump` (from the current distribution of `chain`) and return the sample where we landed. Reuse `result-sample` when possible.

- **[generic-function]** **jump-to-sample*** *chain jump result-sample*

  This function is called by `jump-to-sample`. It is where `jump-to-sample` behaviour shall be customized.

- **[generic-function]** **prepare-jump-distribution** *chain*

  Prepare for sampling from the F(X) = Q(SAMPLE->X) distribution. Called by `random-jump`. The around method ensures that nothing is done unless there was a state change.

- **[generic-function]** **random-jump** *chain*

  Sample a jump from the current distribution of jumps that was computed by `prepare-jump-distribution`.

- **[generic-function]** **log-probability-ratio** *chain sample1 sample2*

  Return P(`sample1`)/P(`sample2`). It's in the log domain to avoid overflows and the ratio part is because that it may allow computational shortcuts as opposed to calculating unnormalized probabilities separately.

- **[generic-function]** **log-probability-ratio-to-jump-target** *chain jump target*

  Return P(`target`)/P(`state`) where `jump` is from the current state of `chain` to `target` sample. This can be specialized for speed. The default implementation just falls back on `log-probability-ratio`.

- **[generic-function]** **log-jump-probability-ratio** *chain jump target*

  Return Q(TARGET->STATE) / Q(STATE->TARGET) where Q is the jump distribution and `jump` is from the current `state` of `chain` to `target` sample.

- **[generic-function]** **acceptance-probability** *chain jump candidate*

  Calculate the acceptance probability of `candidate` to which `jump` leads from the current `state` of `chain`.

- **[generic-function]** **accept-jump** *chain jump candidate*

Called when `chain` accepts `jump` to `candidate`.

- **[generic-function]** **reject-jump** *chain jump candidate*

  Called when `chain` rejects `jump` to `candidate`. It does nothing by default, it's just a convenience for debugging.

- **[generic-function]** **maybe-jump** *chain jump candidate acceptance-probability*

  Randomly accept or reject `jump` to `candidate` from the current state of `chain` with `acceptance-probability`.

- **[generic-function]** **jump** *chain*

  Take a step on the markov chain. Return a boolean indicating whether the proposed jump was accepted.

- **[class]** **mc3-chain** *mc-chain*

  High probability island separated by low valley make the chain poorly mixing. `mc3-chain` has a number of `hot-chains` that have state probabilities similar to that of the main chain but less jagged. Often it suffices to set the temperatures of the `hot-chains` higher use the very same base probability distribution.

- **[generic-function]** **accept-swap-chain-states** *mc3 chain1 chain2*

  Swap the states of `chain1` and `chain2` of `mc3`.

- **[generic-function]** **reject-swap-chain-states** *mc3 chain1 chain2*

  Called when the swap of states of `chain1` and `chain2` is rejected. It does nothing by default, it's just a convenience for debugging.

- **[generic-function]** **maybe-swap-chain-states** *mc3 chain1 chain2 acceptance-probability*

  Swap of states of `chain1` and `chain2` of `mc3` with `acceptance-probability`.

- **[generic-function]** **jump-between-chains** *mc3*

  Choose two chains randomly and swap their states with `mc3` acceptance probability.

- **[class]** **enumerating-chain** *mc-chain*

  A simple abstract chain subclass that explicitly enumerates the probabilities of the distribution.

- **[class]** **tracing-chain**

  Mix this in with your chain to have it print trace of acceptances/rejections.

4 Game Theory

[in package MICMAC.GAME-THEORY]

- **[function]** **find-nash-equilibrium** *payoff &key (n-iterations 100)*

```
Find a Nash equilibrium of a zero-sum game represented by `payoff`
matrix (a 2d matrix or a nested list). `payoff` is from the point of
view of the row player: the player who choses column wants to
minimize, the row player wants to maximize. The first value returned
is a vector of unnormalized probabilities assigned to each action of
the row player, the second value is the same for the column player
and the third is the expected payoff of the row player in the nash
equilibrium represented by the oddment vectors.
```

5 Indices

Referrer definition type abbreviations:

- *f*: for definitions in the function namespace (macros, compiler macros and also methods)
- *t*: DEFTYPEs, classes, conditions, structs
- *d*: documentation sections and glossary terms
- *l*: definitions of definition types
- *s*: ASDF systems
- *p*: packages
- *n*: named readtables
- *v*: special variables and constants
- *r*: restarts
- *?*: other

5.1 Function and Macro Index

5.2 Type Index

5.3 Misc Index