

MiniFrame

P. Baillehache

October 28, 2018

Contents

1	Interface	2
1.1	miniframe.h	2
2	Code	10
2.1	miniframe.c	10
2.2	miniframe-inline.c	32
3	Makefile	43
4	Unit tests	44
5	Unit tests output	53
6	Examples	54
6.1	Basic example	54
6.1.1	miniframe-model.h	54
6.1.2	miniframe-model.c	56
6.1.3	miniframe-inline-model.c	61
6.2	Oware	61
6.2.1	miniframe-model.h	61
6.2.2	miniframe-model.c	64
6.2.3	miniframe-inline-model.c	71
6.2.4	main.c	71
6.2.5	Makefile	77
6.2.6	Example	78

Introduction

MiniFrame is a C library providing a framework to implement the MiniMax algorithm.

The user can define the system to which the MiniMax algorithm is apply by implementing the set of functions in files `miniframe-model.h`, `miniframe-inline-model.c` and `miniframe-model.c`.

It supports one or several actor(s) and uses a time limit to control MiniMax expansion. MiniFrame uses time prediction to maximise the number of steps computed inside the time limit and minimize the risk of overcoming this time limit.

The user can choose if MiniFrame should try to reuse previously computed worlds or recompute several times the same world if it's reachable through several transitions. If it reuses previously computed worlds MiniFrame provide the percentage of reused worlds at each step. MiniFrame also provide the time unused and the number of computed worlds at each step to allow the user to estimate performances.

A basic example is given to illustrate how to use MiniFrame, as well as the implementation for the game of Oware.

The example of the game of Oware also contains an implementation of how to combine MiniFrame with ELORank, GenAlg and NeuraNet to train a NeuraNet later used as the evaluation function of the MiniFrame.

It uses the PBErr, PBMath and GSet libraries.

1 Interface

1.1 miniframe.h

```
// ===== MINIFRAME.H =====  
  
#ifndef MINIFRAME_H  
#define MINIFRAME_H  
  
// ===== Include =====  
  
#include <stdlib.h>
```

```

#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>
#include "pberr.h"
#include "pbmath.h"
#include "gset.h"

// ===== Define =====

// Default time for expansion, in millisecond
#define MF_DEFAULTTIMEEXPANSION 100
// time_ms = clock() / MF_MILLISECTOCLOCKS
#define MF_MILLISECTOCLOCKS (CLOCKS_PER_SEC * 0.001)
// Default number of transitions per world above which the MonteCarlo
// algorithm is activated during expansion
#define MF_NBTRANSMONTECARLO 100
// Default value for pruning during expansion
#define MF_PRUNINGDELTAVAL 1000.0

// ===== Interface with the model implementation =====

#include "miniframe-model.h"

// ===== Data structure =====
typedef struct MFWorld MFWorld;
typedef struct MFTransition {
    // User defined transition
    MFModelTransition _transition;
    // Reference to the world to which this action is applied
    MFWorld* _fromWorld;
    // Reference to the reached world through this action
    // if null it means this action has not been computed
    MFWorld* _toWorld;
    // Array of forecasted POV value of this transition for each actor
    float _values[MF_NBMAXACTOR];
} MFTransition;

typedef struct MFWorld {
    // User defined status of the world
    MFModelStatus _status;
    // Set of transitions reaching this world
    GSet _sources;
    // Array of value of this world from the pov of each actor
    float _values[MF_NBMAXACTOR];
    // Array to memorize the transitions from this world instance
    MFTransition _transitions[MF_NBMAXTRANSITION];
    // Number of transitions from this world
    int _nbTransition;
    // Depth, internal variable used during expansion
    int _depth;
} MFWorld;

typedef enum MFExpansionType {
    MFExpansionTypeValue,
    MFExpansionTypeWidth
} MFExpansionType;

typedef struct MiniFrame {
    // Nb of steps
    unsigned int _nbStep;

```

```

// Current world instance
MFWorld* _curWorld;
// All the computed world instances, ordered by their value from the
// pov of the preempting player at the previous step
GSet _worlds;
// Time limit for expansion, in millisecond
float _maxTimeExpansion;
// Time unused during expansion, in millisecond
float _timeUnusedExpansion;
// Percent of the total available time available to search for worlds
// to expand in MFExpand(), in ]0.0, 1.0], init to 1.0
float _timeSearchWorld;
// Nb of worlds expanded during last call to MFExpand
int _nbWorldExpanded;
// Nb of worlds unexpanded during last call to MFExpand
int _nbWorldUnexpanded;
// Nb of removed world;
int _nbRemovedWorld;
// Flag to activate the reuse of previously computed same world
bool _reuseWorld;
// Percentage (in [0.0, 1.0]) of world reused during the last
// MFExpand()
float _percWorldReused;
// Time used at end of expansion (per remaining world)
float _timeEndExpansion;
// The clock considered has start during expansion
clock_t _startExpandClock;
// Maximum depth during expansion, if -1 there is no limit
int _maxDepthExp;
// Type of expansion, default is MFExpansionTypeValue
MFExpansionType _expansionType;
// Number of transitions above which the Monte Carlo algorithm is
// activated during expansion
int _nbTransMonteCarlo;
// Value for pruning during expansion
float _pruningDeltaVal;
} MiniFrame;

// ===== Functions declaration =====

// Create a new MiniFrame the initial world 'initStatus'
// The current world is initialized with a copy of 'initStatus'
// Return the new MiniFrame
MiniFrame* MiniFrameCreate(const MFModelStatus* const initStatus);

// Create a new MFWorld with a copy of the MFModelStatus 'status'
// Return the new MFWorld
MFWorld* MFWorldCreate(const MFModelStatus* const status);

// Create a new static MFTransition for the MFWorld 'world' with the
// MFModelTransition 'transition'
// Return the new MFTransition
MFTransition MFTransitionCreateStatic(const MFWorld* const world,
    const MFModelTransition* const transition);

// Free memory used by the MiniFrame 'that'
void MiniFrameFree(MiniFrame** that);

// Free memory used by the MFWorld 'that'
void MFWorldFree(MFWorld** that);

```

```

// Free memory used by properties of the MFTransition 'that'
void MFTransitionFreeStatic(MFTransition* that);

// Get the current MFWorld of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
const MFWorld* MFCurWorld(const MiniFrame* const that);

// Get the GSet of computed MFWorlds of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
const GSet* MFWorlds(const MiniFrame* const that);

// Return the number of computed worlds in the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbComputedWorld(const MiniFrame* const that);

// Return true if the expansion algorithm looks in previously
// computed worlds for same world to reuse, else false
#if BUILDMODE != 0
inline
#endif
bool MFIsWorldReusable(const MiniFrame* const that);

// Set the flag controlling if the expansion algorithm looks in
// previously computed worlds for same world to reuse to 'reuse'
#if BUILDMODE != 0
inline
#endif
void MFSetWorldReusable(MiniFrame* const that, const bool reuse);

// Add the MFWorld 'world' to the computed MFWorlds of the
// MiniFrame 'that', ordered by the world's value from the pov of
// actor 'iActor'
#if BUILDMODE != 0
inline
#endif
void MFAddWorld(MiniFrame* const that, \
    const MFWorld* const world, const int iActor);

// Get the time limit for expansion of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetMaxTimeExpansion(const MiniFrame* const that);

// Get the time unused during last expansion of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetTimeUnusedExpansion(const MiniFrame* const that);

// Get the time used to search world to expand during next expansion
// of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetTimeSearchWorld(const MiniFrame* const that);

```

```

// Get the nb of world expanded during the last expansion
// of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbWorldExpanded(const MiniFrame* const that);

// Get the nb of world unexpanded during the last expansion
// of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbWorldUnexpanded(const MiniFrame* const that);

// Get the nb of removed world during the last call to SetCurWorld
// of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbWorldRemoved(const MiniFrame* const that);

// Get the time used at end of expansion of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetTimeEndExpansion(const MiniFrame* const that);

// Get the percentage of resued world of the MiniFrame 'that' during
// the last MFEpxand()
#if BUILDMODE != 0
inline
#endif
float MFGetPercWordReused(const MiniFrame* const that);

// Get the clock considered has start during expansion
#if BUILDMODE != 0
inline
#endif
clock_t MFGetStartExpandClock(const MiniFrame* const that);

// Set the clock considered has start during expansion to 'c'
#if BUILDMODE != 0
inline
#endif
void MFSetStartExpandClock(MiniFrame* const that, clock_t c);

// Set the time limit for expansion of the MiniFrame 'that' to
// 'timeLimit', in millisecond
// The time is measured with the function clock(), see "man clock"
// for details
#if BUILDMODE != 0
inline
#endif
void MFSetMaxTimeExpansion(MiniFrame* const that, \
    const float timeLimit);

// Return the MFModelStatus of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
const MFModelStatus* MFWorldStatus(const MFWorld* const that);

```

```

// Expand the MiniFrame 'that' until it reaches its time limit or can't
// expand anymore
void MFExpand(MiniFrame* that);

// Return the forecasted value of the MFWorld 'that' for the
// actor 'iActor'.
// This is the best value of the transitions from this world,
// or the value of this world if it has no transition.
float MFWorldGetForecastValue(const MFWorld* const that,
    const int iActor);

// Get the number of transition for the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
int MFWorldGetNbTrans(const MFWorld* const that);

// Get the number of expandable transition for the MFWorld 'that'
int MFWorldGetNbTransExpandable(const MFWorld* const that);

// Get the MFWorld which the MFTransition 'that' is leading to
#if BUILDMODE != 0
inline
#endif
const MFWorld* MFTransitionToWorld(const MFTransition* const that);

// Set the MFWorld to which the MFTransition 'that' is leading to
// 'world'
#if BUILDMODE != 0
inline
#endif
void MFTransitionSetToWorld(MFTransition* const that,
    MFWorld* const world);

// Get the MFWorld which the MFTransition 'that' is coming from
#if BUILDMODE != 0
inline
#endif
const MFWorld* MFTransitionFromWorld(const MFTransition* const that);

// Return true if the MFTransition 'that' is expandable, i.e. its
// 'toWorld' is null, else return false
bool MFTransitionIsExpandable(const MFTransition* const that);

// Get the 'iTrans' MFTransition of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
const MFTransition* MFWorldTransition(const MFWorld* const that,
    const int iTrans);

// Get the set of MFTransition reaching the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
const GSet* MFWorldSources(const MFWorld* const that);

// Return the array of values of the MFWorld 'that' for each actor
#if BUILDMODE != 0
inline
#endif

```

```

const float* MFWorldValues(const MFWorld* const that);

// Compute the MFModelState resulting from the 'iTrans' MFTransition
// of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
MFModelState MFWorldComputeTransition(const MFWorld* const that,
    const int iTrans);

// Get the forecast value of the MFWorld 'that' for the actor 'iActor'
float MFWorldGetForecastValue(const MFWorld* const that, int iActor);

// Set the value of the MFTransition 'that' for the actor 'iActor' to
// 'val'
#if BUILDMODE != 0
inline
#endif
void MFTransitionSetValue(MFTransition* const that, const int iActor,
    const float val);

// Return the value of the MFTransition 'that' for the
// actor 'iActor'.
#if BUILDMODE != 0
inline
#endif
float MFTransitionGetValue(const MFTransition* const that,
    const int iActor);

// Return the value of the MFWorld 'that' for the
// actor 'iActor'.
#if BUILDMODE != 0
inline
#endif
float MFWorldGetValue(const MFWorld* const that, const int iActor);

// Get the best MFModelTransition for the 'iActor'-th actor in the
// current MFWorld of the MiniFrame 'that'
// Return an undefined MFTransition if the currenrt world has no
// transition
const MFModelTransition* MFBestTransition(
    const MiniFrame* const that, const int iActor);

// Print the MFWorld 'that' on the stream 'stream'
void MFWorldPrint(const MFWorld* const that, FILE* const stream);

// Print the MFTransition 'that' on the stream 'stream'
void MFTransitionPrint(const MFTransition* const that,
    FILE* const stream);

// Print the MFWorld 'that' and its MFTransition on the stream 'stream'
void MFWorldTransPrintln(const MFWorld* const that,
    FILE* const stream);

// Set the current world of the MiniFrame 'that' to match the
// MFModelState 'status'
// If the world is in computed worlds reuse it, else create a new one
void MFSetCurWorld(MiniFrame* const that,
    const MFModelState* const world);

// Print the best forecasted story from the MFWorld 'that' for the
// actor 'iActor' on the stream 'stream'

```



```

void MFWorldPrintBestStoryln(const MFWorld* const that, const int iActor,
    FILE* const stream);

// Set the values of the MFWorld 'that' to 'values'
void MFWorldSetValues(MFWorld* const that, const float* const values);

// Return the max depth during expansion for the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetMaxDepthExp(const MiniFrame* const that);

// Set the max depth during expansion for the MiniFrame 'that' to 'depth'
// If depth is less than -1 it is converted to -1
#if BUILDMODE != 0
inline
#endif
void MFSetMaxDepthExp(MiniFrame* const that, const int depth);

// Return the type of expansion for the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
MFExpansionType MFGetExpansionType(const MiniFrame* const that);

// Set the type expansion for the MiniFrame 'that' to 'type'
#if BUILDMODE != 0
inline
#endif
void MFSetExpansionType(MiniFrame* const that, const MFExpansionType type);

// Set the nb of transitio to activate MonteCarlo during expansion
// for the MiniFrame 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void MFSetNbTransMonteCarlo(MiniFrame* const that, const int nb);

// Set the nb of transitions to activate MonteCarlo during expansion
// for the MiniFrame 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void MFSetNbTransMonteCarlo(MiniFrame* const that, const int nb);

// Get the nb of transitions to activate MonteCarlo during expansion
// for the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbTransMonteCarlo(const MiniFrame* const that);

// Return true if the MFTransition is expanded, false else
#if BUILDMODE != 0
inline
#endif
bool MFTransitionIsExpanded(const MFTransition* const that);

// Set the pruning threshold during expansion for the MiniFrame 'that'
// to 'val'
#if BUILDMODE != 0
inline

```

```

#endif
void MFSetPruningDeltaVal(MiniFrame* const that, const float val);

// Get the pruning threshold during expansion for the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetPruningDeltaVal(const MiniFrame* const that);

// ===== Inliner =====

#if BUILDMODE != 0
#include "miniframe-inline.c"
#endif

#endif

```

2 Code

2.1 miniframe.c

```

// ===== MINIFRAME.C =====

// ===== Include =====

#include "miniframe.h"
#if BUILDMODE == 0
#include "miniframe-inline.c"
#endif

// ===== Functions declaration =====

// Return true if the MFWorld 'that' should be pruned during search for
// worlds to expand when reaching it through transition 'trans',
// false else
bool MFWorldIsPrunedDuringExpansion(const MFWorld* const that,
    const MiniFrame* const mf, const MFTransition* const trans);

// Get the set of worlds to be expanded for the MiniFrame 'that'
// Prune worlds which have a value lower than a threhsold compare to
// their best brother
// Stop searching for world if clock() >= clockLimit
// Will return at least one world even if clockLimit == current clock
// The MiniFrame must have at least one world in its set of computed
// worlds
// Force the current world to the end of the returned set to ensure
// it will be the first to be expanded
GSet MFGetWorldsToExpand(MiniFrame* const that,
    const clock_t clockLimit);
void MFGetWorldsToExpandRec(MiniFrame* const that,
    MFWorld* const world, GSet* set, const clock_t clockLimit,
    int depth, GSet* setVisited);

// Return true if the MFWorld 'that' has at least one transition to be
// expanded
bool MFWorldIsExpandable(const MFWorld* const that);

```

```

// Search in computed worlds of the MiniFrame 'that' if there is
// one with same status as the MFModelState 'status'
// If there is one return it, if not return null
MFWorld* MFSearchWorld(const MiniFrame* const that,
    const MFModelState* const status);

// Set the MFWorld 'toWorld' has the result of the 'iTrans' transition
// of the world 'that'
// Update the value of the transition
void MFWorldSetTransitionToWorld(
    MFWorld* const that, const int iTrans, MFWorld* const toWorld);

// Update backward the forecast values for actor 'iActor' for each
// transitions leading to the MFWorld 'world' in the MiniFrame 'that'
// Use a penalty growing with each recursive call to
// MFUpdateForecastValues to give priority to fastest convergence to
// best solution
// Avoid infinite loop due to reuse of computed worlds
void MFUpdateForecastValues(MiniFrame* const that,
    const MFWorld* const world, int delayPenalty, GSet* const setWorld,
    int iActor);

// Update the values of the MFTransition 'that' for actor 'iActor' with
// 'val'
// Return true if the value has been updated, else false
bool MFTransitionUpdateValue(MFTransition* const that, const int iActor,
    const float val);

// Pop a MFTransition from the sources of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
MFTransition* MFWorldPopSource(MFWorld* const that);

// Remove the MFTransition 'source' from the sources of the
// MFWorld 'that'
void MFWorldRemoveSource(MFWorld* const that,
    const MFTransition* const source);

// Get the best MFModelTransition for the 'iActor'-th actor in the
// MFWorld 'that'
// Return NULL if the world has no transition
const MFModelTransition* MFWorldBestTransition(
    const MFWorld* const that, const int iActor);

// Free the memory used by the disposable worlds in the computed worlds
// of the MiniFrame 'that'
void MFFreeDisposableWorld(MiniFrame* const that);

// ===== Functions implementation =====

// Create a new MiniFrame the initial world 'initStatus'
// The current world is initialized with a copy of 'initStatus'
// Return the new MiniFrame
MiniFrame* MiniFrameCreate(const MFModelState* const initStatus) {
#if BUILDMODE == 0
    if (initStatus == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'initStatus' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
}

```

```

// Allocate memory
MiniFrame *that = PBErrMalloc(MiniFrameErr, sizeof(MiniFrame));
// Set properties
that->_nbStep = 0;
MFSetMaxTimeExpansion(that, MF_DEFAULTTIMEEXPANSION);
that->_curWorld = MFWorldCreate(initStatus);
that->_worlds = GSetCreateStatic();
MFAddWorld(that, MFCurWorld(that), MFModelStatusGetSente(initStatus));
that->_timeSearchWorld = MF_DEFAULTTIMEEXPANSION;
that->_nbWorldExpanded = 0;
that->_nbWorldUnexpanded = 0;
that->_nbRemovedWorld = 0;
that->_timeUnusedExpansion = 0.0;
that->_reuseWorld = false;
that->_percWorldReused = 0.0;
that->_startExpandClock = 0;
that->_maxDepthExp = -1;
that->_expansionType = MFExpansionTypeValue;
that->_nbTransMonteCarlo = MF_NBTRANSMONTECARLO;
that->_pruningDeltaVal = MF_PRUNINGDELTAVAL;
// Estimate the time used at end of expansion which is the time
// used to flush a gset
GSet set = GSetCreateStatic();
int nb = 100;
float timeFlush = 0.0;
do {
    for (int i = nb; i--;)
        GSetPush(&set, NULL);
    clock_t timeStart = clock();
    GSetFlush(&set);
    clock_t timeEnd = clock();
    timeFlush = ((double)(timeEnd - timeStart)) / MF_MILLISECTOCLOCKS;
} while (timeFlush < 0.0);
that->_timeEndExpansion = timeFlush / (float)nb;
// Return the new MiniFrame
return that;
}

// Create a new MFWorld with a copy of the MFModelStatus 'status'
// Return the new MFWorld
MFWorld* MFWorldCreate(const MFModelStatus* const status) {
#ifdef BUILDMODE == 0
    if (status == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'status' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    // Allocate memory
    MFWorld *that = PBErrMalloc(MiniFrameErr, sizeof(MFWorld));
    // Set the status
    MFModelStatusCopy(status, &(that->_status));
    // Initialise the set of transitions reaching this world
    that->_sources = GSetCreateStatic();
    // Set the values
    float values[MF_NBMAXACTOR] = {0.0};
    MFModelStatusGetValues(status, values);
    MFWorldSetValues(that, values);
    // Set the possible transitions from this world
    MFModelTransition transitions[MF_NBMAXTRANSITION];
    MFModelStatusGetTrans(status, transitions, &(that->_nbTransition));
    for (int iTrans = that->_nbTransition; iTrans--;)

```

```

        that->_transitions[iTrans] =
            MFTransitionCreateStatic(that, transitions + iTrans);
    // Return the new MFWorld
    return that;
}

// Create a new static MFTransition for the MFWorld 'world' with the
// MFModelTransition 'transition'
// Return the new MFTransition
MFTransition MFTransitionCreateStatic(const MFWorld* const world,
    const MFModelTransition* const transition) {
#ifdef BUILDMODE == 0
    if (world == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'world' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    // Declare a variable to memorize the new action
    MFTransition that;
    // Set properties
    that._transition = *transition;
    that._fromWorld = (MFWorld*)world;
    that._toWorld = NULL;
    for (int iActor = MF_NBMAXACTOR; iActor--;)
        that._values[iActor] = 0.0;
    // Return the new MFTransition
    return that;
}

// Free memory used by the MiniFrame 'that'
void MiniFrameFree(MiniFrame** that) {
    // Check argument
    if (that == NULL || *that == NULL) return;
    // Free memory
    while(GSetNbElem(&((*that)->_worlds)) > 0) {
        MFWorld* world = GSetPop(&((*that)->_worlds));
        MFWorldFree(&world);
    }
    free(*that);
    *that = NULL;
}

// Free memory used by the MFWorld 'that'
void MFWorldFree(MFWorld** that) {
    // Check argument
    if (that == NULL || *that == NULL) return;
    // Free memory
    GSetFlush(&((*that)->_sources));
    MFModelStatusFreeStatic(&((*that)->_status));
    for (int iAct = (*that)->_nbTransition; iAct--;) {
        if ((*that)->_transitions[iAct]._toWorld != NULL)
            MFTransitionFreeStatic((*that)->_transitions + iAct);
    }
    free(*that);
    *that = NULL;
}

// Free memory used by properties of the MFTransition 'that'
void MFTransitionFreeStatic(MFTransition* that) {
    // Check argument
    if (that == NULL) return;

```

```

    // Free memory
    MFModelTransitionFreeStatic(&(that->_transition));
}

// Expand the MiniFrame 'that' until it reaches its time limit or can't
// expand anymore
void MFExpand(MiniFrame* that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
    // Declare a variable to memorize the time at beginning of the whole
    // expansion process
    clock_t clockStart = MFGetStartExpandClock(that);
    // Declare a variable to memorize the maximum time used for one
    // step of expansion
    double maxTimeOneStep = 0.0;
    // Free the disposable worlds
    MFFreeDisposableWorld(that);
    // Create the set of world instances to be expanded, ordered by
    // world's value from the point of view of the preempting actor
    // for each world
    // The time available for this step is limited to avoid spending
    // time to search for worlds to expand and finally not having time
    // to compute them
    clock_t clockLimit = clockStart +
        that->_timeSearchWorld * MF_MILLISECTOCLOCKS;
    GSet worldsToExpand = MFGetWorldsToExpand(that, clockLimit);
    // Memorize the number of worlds to expand
    int nbWorldToExpand = GSetNbElem(&worldsToExpand);
    // Declare a variable to memorize the time spend expanding
    double timeUsed =
        ((double)(clock() - clockStart)) / MF_MILLISECTOCLOCKS;
    // Declare a variable to memorize the number of reused worlds
    int nbReusedWorld = 0;
    // Declare a variable to memorize the number of worlds to expand added
    // to the original set
    int nbWorldToExpandPost = 0;
    // Loop until we have time for one more step of expansion or there
    // is no world to expand
    // Take care of clock() wrapping around
    while (timeUsed >= 0.0 &&
        timeUsed + maxTimeOneStep +
        MFGetTimeEndExpansion(that) * GSetNbElem(&worldsToExpand) <
        MFGetMaxTimeExpansion(that) &&
        GSetNbElem(&worldsToExpand) > 0) {
        // Declare a variable to memorize the time at the beginning of one
        // step of expansion
        clock_t clockStartLoop = clock();
        // Drop the world to expand with highest value
        MFWorld* worldToExpand = GSetDrop(&worldsToExpand);
        // Get the sente for this world
        int sente = MFModelStatusGetSente(MFWorldStatus(worldToExpand));
        // Get the number of expandable transition
        int nbTransExpandable = MFWorldGetNbTransExpandable(worldToExpand);
        // Get the threshold for expansion to activate montecarlo when
        // there are too many transitions
        float thresholdMonteCarlo =
            (float)MFGetNbTransMonteCarlo(that) / (float)nbTransExpandable;
    }
}

```

```

// For each transitions from the expanded world and until we have
// time available
// Take care of clock() wrapping around
for (int iTrans = 0; iTrans < MFWorldGetNbTrans(worldToExpand) &&
    timeUsed >= 0.0 &&
    timeUsed + maxTimeOneStep +
    MFGetTimeEndExpansion(that) * GSetNbElem(&worldsToExpand) <
    MFGetMaxTimeExpansion(that);
    ++iTrans) {
    // If this transition has not been computed
    const MFTransition* const trans =
        MFWorldTransition(worldToExpand, iTrans);
    if (MFTransitionIsExpandable(trans) &&
        rnd() < thresholdMonteCarlo) {
        // Expand through this transition
        MFModelState status =
            MFWorldComputeTransition(worldToExpand, iTrans);
        // If the resulting status has not already been computed
        MFWorld* sameWorld = MFSearchWorld(that, &status);
        if (sameWorld == NULL) {
            // Create a MFWorld for the new status
            MFWorld* expandedWorld = MFWorldCreate(&status);
            // Add the world to the set of computed world
            MFAddWorld(that, expandedWorld, sente);
            // Set the expanded world as the result of the transition
            MFWorldSetTransitionToWorld(
                worldToExpand, iTrans, expandedWorld);
            // If it's not an end status world and we haven't reached
            // the expansion limit
            if ((that->_maxDepthExp < 0 ||
                worldToExpand->_depth < that->_maxDepthExp) &&
                !MFModelStateIsEnd(MFWorldStatus(expandedWorld))) {
                // Add the world to the set of worlds to expand
                ++nbWorldToExpand;
                expandedWorld->_depth = worldToExpand->_depth + 1;
                if (MFGetExpansionType(that) == MFExpansionTypeValue) {
                    float value = MFWorldGetValue(expandedWorld, sente);
                    GSetAddSort(&worldsToExpand, expandedWorld, value);
                } else if (MFGetExpansionType(that) == MFExpansionTypeWidth) {
                    GSetPush(&worldsToExpand, expandedWorld);
                }
                ++nbWorldToExpandPost;
            }
        } else {
            // Increment the number of reused world
            ++nbReusedWorld;
            // Set the already computed one as the result of the
            // transition
            MFWorldSetTransitionToWorld(worldToExpand, iTrans, sameWorld);
        }
    }
    // Update the total time used from beginning of expansion
    timeUsed =
        ((double)(clock() - clockStart)) / MF_MILLISECTOCLOCKS;
}
// For each actor
for (int iActor =
    MFModelStateGetNbActor(MFWorldStatus(worldToExpand));
    iActor--;) {
    // Update backward the forecast values for each transitions
    // leading to the expanded world according to its new transitions
    GSet setWorld = GSetCreateStatic();

```

```

        MFUpdateForecastValues(that, worldToExpand, 0, &setWorld, iActor);
        GSetFlush(&setWorld);
    }
    // Declare a variable to memorize the time at the end of one
    // step of expansion
    clock_t clockEndLoop = clock();
    // Calculate the time for this step
    double timeOneStep =
        ((double)(clockEndLoop - clockStartLoop)) / MF_MILLISECTOCLOCKS;
    // Update max time used by one step
    if (maxTimeOneStep < timeOneStep)
        maxTimeOneStep = timeOneStep;
    // Update the total time used from beginning of expansion
    timeUsed =
        ((double)(clockEndLoop - clockStart)) / MF_MILLISECTOCLOCKS;
}
// Memorize the remaining number of worlds to expand
int nbRemainingWorldToExpand =
    MAX(0, GSetNbElem(&worldsToExpand) - nbWorldToExpandPost);
// Update the total time used from beginning of expansion
timeUsed = ((double)(clock() - clockStart)) / MF_MILLISECTOCLOCKS;
// Update the percentage of time allocated to searching for worlds
// to expand
// If there was worlds to expand
if (nbWorldToExpand > 0) {
    // If we could expand all the worlds
    if (nbRemainingWorldToExpand == 0) {
        that->_timeSearchWorld *=
            MFGetMaxTimeExpansion(that) / timeUsed;
        if (that->_timeSearchWorld > MFGetMaxTimeExpansion(that))
            that->_timeSearchWorld = MFGetMaxTimeExpansion(that);
    }
    // Else, we had not enough time to expand all the worlds
    } else {
        that->_timeSearchWorld *=
            (float)nbRemainingWorldToExpand / (float)nbWorldToExpand;
    }
} else {
    that->_timeSearchWorld =
        MAX(0, MFGetMaxTimeExpansion(that) - timeUsed);
}
// Empty the list of worlds to expand
GSetFlush(&worldsToExpand);
// Update the total time used from beginning of expansion
timeUsed = ((double)(clock() - clockStart)) / MF_MILLISECTOCLOCKS;
// Take care of clock() wrapping around
if (timeUsed < 0.0)
    timeUsed = MFGetMaxTimeExpansion(that);
// Telemetry for debugging
that->_timeUnusedExpansion = MFGetMaxTimeExpansion(that) - timeUsed;
that->_nbWorldExpanded =
    nbWorldToExpand - nbRemainingWorldToExpand + nbReusedWorld;
that->_nbWorldUnexpanded = nbRemainingWorldToExpand;
if (that->_nbWorldExpanded > 0)
    that->_percWorldReused =
        (float)nbReusedWorld / (float)(that->_nbWorldExpanded);
else
    that->_percWorldReused = 0.0;
}

// Get the set of worlds to be expanded (having at least one transition
// whose _toWorld is null) for the MiniFrame 'that'
// Stop searching for world if clock() >= clockLimit

```



```

// Will return at least one world even if clockLimit == current clock
// The MiniFrame must have at least one world in its set of computed
// worlds
// Force the current world to the end of the returned set to ensure
// it will be the first to be expanded
GSet MFGetWorldsToExpandOld(MiniFrame* const that,
    const clock_t clockLimit) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (GSetNbElem(MFWorlds(that)) == 0) {
        MiniFrameErr->_type = PBErrTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "The MiniFrame has no computed world");
        PBErrCatch(MiniFrameErr);
    }
#endif
    // Free the disposable worlds
    MFFreeDisposableWorld(that);
    // Declare the set to memorize worlds to expand
    GSet set = GSetCreateStatic();
    // Loop through the computed worlds
    GSetIterForward iter = GSetIterForwardCreateStatic(MFWorlds(that));
    do {
        MFWorld* world = GSetIterGet(&iter);
        // If this world has transition to expand
        if (world != MFCurWorld(that) && MFWorldIsExpandable(world)) {
            // Add this world to the result set ordered by the value
            world->_depth = 0;
            if (MFGetExpansionType(that) == MFExpansionTypeValue) {
                int sente = MFModelStatusGetSente(MFWorldStatus(world));
                float value = MFWorldGetForecastValue(world, sente);
                GSetAddSort(&set, world, value);
            } else if (MFGetExpansionType(that) == MFExpansionTypeWidth) {
                GSetPush(&set, world);
            }
        }
    } while (GSetIterStep(&iter) && clock() < clockLimit);
    // Add the current world
    if (MFWorldIsExpandable(MFCurWorld(that))) {
        that->_curWorld->_depth = 0;
        GSetAppend(&set, MFCurWorld(that));
    }
    // Return the set of worlds to expand
    return set;
}

// Return true if the MFWorld 'that' should be pruned during search for
// worlds to expand when reaching it through transition 'trans',
// false else
bool MFWorldIsPrunedDuringExpansion(const MFWorld* const that,
    const MiniFrame* const mf, const MFTransition* const trans) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (mf == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
    }

```

```

        sprintf(MiniFrameErr->_msg, "'mf' is null");
        PBErriCatch(MiniFrameErr);
    }
    if (trans == NULL) {
        MiniFrameErr->_type = PBErriTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'trans' is null");
        PBErriCatch(MiniFrameErr);
    }
    if (MFTransitionToWorld(trans) != world) {
        MiniFrameErr->_type = PBErriTypeInvalidArg;
        sprintf(MiniFrameErr->_msg,
            "The transition doesn't reach the world");
        PBErriCatch(MiniFrameErr);
    }
    if (MFTransitionFromWorld(trans) == NULL) {
        MiniFrameErr->_type = PBErriTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "The transition has no origin");
        PBErriCatch(MiniFrameErr);
    }
}
#endif
// Declare a variable to memorize the result
bool ret = false;
// Get the origin world of the transition
const MFWorld* const fatherWorld = MFTransitionFromWorld(trans);
// Get the sente of the father world
int sente = MFModelStatusGetSente(MFWorldStatus(fatherWorld));
// Declare a variable to memorize the maximum value of brothers world
float max = 0.0;
const MFWorld* bestBrother = NULL;
// Loop on transitions from the father world
for (int iTrans = MFWorldGetNbTrans(fatherWorld); iTrans--;) {
    // Get the origin of the current transition
    const MFWorld* const brother =
        MFTransitionToWorld(MFWorldTransition(fatherWorld, iTrans));
    if (brother != that && brother != NULL) {
        // Get the value of the brother
        const float val = MFWorldGetForecastValue(brother, sente);
        // Update the maximum if necessary
        if (bestBrother == NULL || max < val) {
            bestBrother = brother;
            max = val;
        }
    }
}
// If there is at least one brother
if (bestBrother != NULL) {
    // Get the value of the world in argument
    float val = MFWorldGetForecastValue(that, sente);
    // If the pruning constraint is true
    if (val < max - MFGetPruningDeltaVal(mf))
        // Update the result
        ret = true;
}
// Return the result
return ret;
}

// Get the set of worlds to be expanded for the MiniFrame 'that'
// Prune worlds which have a value lower than a threhsold compare to
// their best brother
// Stop searching for world if clock() >= clockLimit
// Will return at least one world even if clockLimit == current clock

```

```

// The MiniFrame must have at least one world in its set of computed
// worlds
// Force the current world to the end of the returned set to ensure
// it will be the first to be expanded
GSet MFGetWorldsToExpand(MiniFrame* const that,
    const clock_t clockLimit) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
    if (GSetNbElem(MFWorlds(that)) == 0) {
        MiniFrameErr->_type = PErrTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "The MiniFrame has no computed world");
        PErrCatch(MiniFrameErr);
    }
#endif
    // Create the result set
    GSet set = GSetCreateStatic();
    // Create the visited set
    GSet setVisited = GSetCreateStatic();
    // Get the current world
    MFWorld* const world = (MFWorld*)MFCurWorld(that);
    // Start the recursion
    MFGetWorldsToExpandRec(that, world, &set, clockLimit, 0, &setVisited);
    // Add the current world of the MiniFrame at the end of the set
    if (MFWorldIsExpandable(world)) {
        world->_depth = 0;
        GSetAppend(&set, world);
    }
    // Free memory
    GSetFlush(&setVisited);
    // Return the set
    return set;
}

void MFGetWorldsToExpandRec(MiniFrame* const that,
    MFWorld* const world, GSet* set, const clock_t clockLimit,
    int depth, GSet* setVisited) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
    if (GSetNbElem(MFWorlds(that)) == 0) {
        MiniFrameErr->_type = PErrTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "The MiniFrame has no computed world");
        PErrCatch(MiniFrameErr);
    }
#endif
    // Avoid infinite loop when reusing world
    if (MFIsWorldReusable(that) &&
        GSetFirstElem(setVisited, world) != NULL)
        return;
    else
        GSetAppend(setVisited, world);
    // If we are not at the root and the current world is expandable
    if (depth != 0 && MFWorldIsExpandable(world)) {
        // Add the world to the result set ordered by depth or value
        world->_depth = 0;
        if (MFGetExpansionType(that) == MFEExpansionTypeValue) {

```

```

        int sente = MFModelStatusGetSente(MFWorldStatus(world));
        float value = MFWorldGetForecastValue(world, sente);
        GSetAddSort(set, world, value);
    } else if (MFGetExpansionType(that) == MFEExpansionTypeWidth) {
        float value = -1.0 * (float)depth;
        GSetAddSort(set, world, value);
    }
}
// Loop on the transitions of the current world
for (int iTrans = MFWorldGetNbTrans(world);
    iTrans-- && clock() < clockLimit;) {
    // Get the transition
    const MFTransition* const trans = MFWorldTransition(world, iTrans);
    // Get the world reached through this transition
    const MFWorld* const toWorld = MFTransitionToWorld(trans);
    // If this transition is expanded
    if (toWorld != NULL) {
        // If the pruning condition is false for the world reached
        // through this transition
        if (!MFWorldIsPrunedDuringExpansion(toWorld, that, trans)) {
            // Continue searching for worlds to expand from the world
            // reached through this transition
            MFGetWorldsToExpandRec(that, (MFWorld*)toWorld, set,
                clockLimit, depth + 1, setVisited);
        }
    }
}
}
}

// Return true if the MFWorld 'that' has at least one transition to be
// expanded
bool MFWorldIsExpandable(const MFWorld* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    // Declare a variable to memorize the result
    bool isExpandable = false;
    // If the world is not at the end of the game/simulation
    if (!MFModelStatusIsEnd(MFWorldStatus(that))) {
        // Loop on transitions
        for (int iTrans = that->_nbTransition; iTrans-- && !isExpandable;) {
            // If this transition has not been computed
            if (MFTransitionIsExpandable(MFWorldTransition(that, iTrans)))
                isExpandable = true;
        }
    }
    // Return the result
    return isExpandable;
}

// Search in computed worlds of the MiniFrame 'that' if there is
// one with same status as the MFModelStatus 'status'
// If there is one return it, if not return null
MFWorld* MFSearchWorld(const MiniFrame* const that,
    const MFModelStatus* const status) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;

```

```

        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErriCatch(MiniFrameErr);
    }
    if (status == NULL) {
        MiniFrameErr->_type = PBErriTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'status' is null");
        PBErriCatch(MiniFrameErr);
    }
#endif
    // Declare a variable to memorize the returned world
    MFWorld* sameWorld = NULL;
    // If the reuse of worlds is activated
    if (MFIsWorldReusable(that)) {
        // Loop on computed worlds
        GSetIterForward iter = GSetIterForwardCreateStatic(MFWorlds(that));
        do {
            MFWorld* world = GSetIterGet(&iter);
            // If this world is the same as the searched one
            if (MFModelStatusIsSame(status, MFWorldStatus(world))) {
                sameWorld = world;
            }
        } while (sameWorld == NULL && GSetIterStep(&iter));
    }
    // Return the found world
    return sameWorld;
}

// Set the MFWorld 'toWorld' has the result of the 'iTrans' transition
// of the MFWorld 'that'
// Update the value of the transition
void MFWorldSetTransitionToWorld(
    MFWorld* const that, const int iTrans, MFWorld* const toWorld) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErriTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErriCatch(MiniFrameErr);
        }
        if (iTrans < 0 || iTrans >= that->_nbTransition) {
            MiniFrameErr->_type = PBErriTypeInvalidArg;
            sprintf(MiniFrameErr->_msg, "'iTrans' is invalid (0<=%d<%d)",
                iTrans, that->_nbTransition);
            PBErriCatch(MiniFrameErr);
        }
        if (toWorld == NULL) {
            MiniFrameErr->_type = PBErriTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'toWorld' is null");
            PBErriCatch(MiniFrameErr);
        }
    #endif
    // Declare a variable to memorize the transition
    MFTransition* trans = that->_transitions + iTrans;
    // Set the transition result
    trans->_toWorld = toWorld;
    // Add the transition to the sources to the result's world
    GSetAppend(&(toWorld->_sources), trans);
    // Update the forecast value of this transition for each actor
    for (int iActor = MF_NBMAXACTOR; iActor--;)
        MFTransitionSetValue(trans, iActor,
            MFWorldGetValue(toWorld, iActor));
}

```

```

// Return true if the MFTransition 'that' is expandable, i.e. its
// 'toWorld' is null, else return false
bool MFTransitionIsExpandable(const MFTransition* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
#endif
    // If the transition has already been expanded
    if (MFTransitionToWorld(that) != NULL) {
        // Return false
        return false;
    }
    // Else, the transition has not been expanded yet
} else {
    // Get the origin of the transition
    const MFWorld* fromWorld = MFTransitionFromWorld(that);
    // Declare a variable to memorize if the transition has a brother
    // which leads to an end world
    bool hasEndWorldBrother = false;
    // For each brother transition, until we have found an end world
    for (int iTrans = MFWorldGetNbTrans(fromWorld);
        iTrans-- && !hasEndWorldBrother;) {
        // Get the brother transition's toWorld
        const MFWorld* brother =
            MFTransitionToWorld(MFWorldTransition(fromWorld, iTrans));
        // If the brother world is an end world
        if (brother != NULL &&
            MFModelStateIsEnd(MFWorldStatus(brother))) {
            // Set the flag
            hasEndWorldBrother = true;
        }
    }
    // If the transition has a brother leading to an end world
    if (hasEndWorldBrother)
        // This transition is not expandable
        return false;
    // Else, the transition has no brother leading to an end world
    else
        // This transition is expandable
        return true;
}
// Should never reach here, but just in case...
return true;
}

// Return the forecasted value of the MFWorld 'that' for the
// actor 'iActor'.
// This is the best value of the transitions from this world,
// or the value of this world if it has no transition.
float MFWorldGetForecastValue(const MFWorld* const that,
    const int iActor) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
    if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
        MiniFrameErr->_type = PErrTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<%d)",

```

```

        iActor, MF_NBMAXACTOR);
        PBErrCatch(MiniFrameErr);
    }
#endif
// Declare a variable to memorize the highest value among transitions
float valBestTrans = 0.0;
// Declare a variable to memorize the transition with highest value
const MFTransition* bestTrans = NULL;
// Loop on transitions
for (int iTrans = MFWorldGetNbTrans(that); iTrans--;) {
    // Declare a variable to memorize the transition
    const MFTransition* const trans =
        MFWorldTransition(that, iTrans);
    // If this transitions has been expanded
    if (!MFTransitionIsExpandable(trans)) {
        // Get the value of the transition from the point of view of
        // the sente
        float val = MFTransitionGetValue(trans, iActor);
        // If it's not the first considered transition
        if (bestTrans != NULL) {
            // If the value is better
            if (valBestTrans < val) {
                valBestTrans = val;
                bestTrans = trans;
            }
        }
        // Else it's the first considered transition
    } else {
        // Init the best value with the value of this transition
        valBestTrans = val;
        // Init the best transition
        bestTrans = trans;
    }
}
// Return the value for this world
// If there are expanded transitions
if (bestTrans != NULL) {
    // Return the value of the best transition from the point of view
    // of the requested actor
    return MFTransitionGetValue(bestTrans, iActor);
}
// Else this world has no transitions
else {
    // Return the value of this world from the point of view of the
    // requested actor
    return MFWorldGetValue(that, iActor);
}
}

// Get the best MFModelTransition for the 'iActor'-th actor in the
// current MFWorld of the MiniFrame 'that'
// Return NULL if the current world has no transition
const MFModelTransition* MFBestTransition(
    const MiniFrame* const that, const int iActor) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
        MiniFrameErr->_type = PBErrTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<=%d)",

```

```

        iActor, MF_NBMAXACTOR);
    PBErrCatch(MiniFrameErr);
}
#endif
// Return the best transition
return MFWorldBestTransition(MFCurWorld(that), iActor);
}

// Get the best MFModelTransition for the 'iActor'-th actor in the
// MFWorld 'that'
// Return NULL if the world has no transition
const MFModelTransition* MFWorldBestTransition(
    const MFWorld* const that, const int iActor) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
        if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
            MiniFrameErr->_type = PBErrTypeInvalidArg;
            sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<%d)",
                iActor, MF_NBMAXACTOR);
            PBErrCatch(MiniFrameErr);
        }
    #endif
    // Declare a variable to memorize the highest value among transitions
    float valBestTrans = 0.0;
    // Declare a variable to memorize the best transition
    const MFTransition* bestTrans = NULL;
    // Loop on transitions
    for (int iTrans = MFWorldGetNbTrans(that); iTrans--;) {
        // Declare a variable to memorize the transition
        const MFTransition* const trans = MFWorldTransition(that, iTrans);
        // If this transitions has been expanded
        if (MFTransitionIsExpanded(trans)) {
            // Get the value of the transition from the point of view of
            // the requested actor
            float val = MFTransitionGetValue(trans, iActor);
            // Add some random perturbation to avoid always picking
            // the same transitions between those with equal values
            val += rnd() * PB_MATH_EPSILON;
            // If it's not the first considered transition
            if (bestTrans != NULL) {
                // If the value is better
                if (valBestTrans < val) {
                    // Update the best value and best transition
                    valBestTrans = val;
                    bestTrans = trans;
                }
            }
            // Else it's the first considered transition
        } else {
            // Init the best value with the value of this transition
            valBestTrans = val;
            // Init the best transition
            bestTrans = trans;
        }
    }
}
// If the bestTrans is null here it means that none of the transitions
// for the current world were expanded yet
// By default choose a random one

```



```

    if (bestTrans == NULL && MFWorldGetNbTrans(that) > 0) {
        bestTrans = MFWorldTransition(that,
            (int)floor(MIN(rnd(), 0.9999) * (float)MFWorldGetNbTrans(that)));
    }
    // Return the best transition
    return (const MFModelTransition*)bestTrans;
}

// Update backward the forecast values for actor 'iActor' for each
// transitions leading to the MFWorld 'world' in the MiniFrame 'that'
// Use a penalty growing with each recursive call to
// MFUpdateForecastValues to give priority to fastest convergence to
// best solution
// Avoid infinite loop due to reuse of computed worlds
void MFUpdateForecastValues(MiniFrame* const that,
    const MFWorld* const world, int delayPenalty, GSet* const setWorld,
    int iActor) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PErrCatch(MiniFrameErr);
        }
        if (world == NULL) {
            MiniFrameErr->_type = PErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'world' is null");
            PErrCatch(MiniFrameErr);
        }
    #endif
    // Avoid infinite loop
    if (GSetFirstElem(setWorld, world) == NULL)
        GSetAppend(setWorld, (void*)world);
    else
        return;
    // If the world has ancestors
    if (GSetNbElem(MFWorldSources(world)) > 0) {
        // Get the forecast value of the world
        float forecastVal = MFWorldGetForecastValue(world, iActor);
        // Declare a variable to memorize when the transition is updated
        bool updated = false;
        // For each transition to the world
        GSetIterForward iter =
            GSetIterForwardCreateStatic(MFWorldSources(world));
        do {
            // Get the transition
            MFTransition* const trans = GSetIterGet(&iter);
            // If we are at the first level of recursion
            if (delayPenalty == 0) {
                // Initialize the value of the transition
                MFTransitionSetValue(trans, iActor, forecastVal);
                updated = true;
            } else {
                // Update the value of the transition
                updated = MFTransitionUpdateValue(trans, iActor,
                    forecastVal - (float)delayPenalty * PBMATH_EPSILON);
            }
            // If the value has been updated
            if (updated) {
                // Propagate the update from the source world
                MFUpdateForecastValues(that, MFTransitionFromWorld(trans),
                    delayPenalty + 1, setWorld, iActor);
            }
        }
    }
}

```

```

    } while (GSetIterStep(&iter));
}
}

// Update the values of the MFTransition 'that' for actor 'iActor' with
// 'val'
// Return true if the value has been updated, else false
bool MFTransitionUpdateValue(MFTransition* const that, const int iActor,
    const float val) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
        MiniFrameErr->_type = PBErrTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<=%d)",
            iActor, MF_NBMAXACTOR);
        PBErrCatch(MiniFrameErr);
    }
#endif
    // Declare a variable to memorize the returned flag
    bool updated = false;
#ifdef MF_NBMAXACTOR == 1
    if (that->_values[iActor] < val) {
        updated = true;
        that->_values[iActor] = val;
    }
#else
#ifdef MF_SIMULTANEOUS_PLAY

#else
        MFWorld* fromWorld = MFTransitionFromWorld(that);
        int sente = MFModelStateGetSente(MFWorldStatus(fromWorld));
        if (sente == -1 || sente == iActor) {
            if (that->_values[iActor] < val) {
                updated = true;
                that->_values[iActor] = val;
            }
        } else {
            if (that->_values[iActor] > val) {
                updated = true;
                that->_values[iActor] = val;
            }
        }
    }
#endif
#endif
    // Return the flag
    return updated;
}

// Print the MFWorld 'that' on the stream 'stream'
void MFWorldPrint(const MFWorld* const that, FILE* const stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (stream == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;

```

```

        sprintf(MiniFrameErr->_msg, "'stream' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    fprintf(stream, "(");
    MFModelStatePrint(MFWorldStatus(that), stream);
    fprintf(stream, ") values[");
    for (int iActor = 0; iActor < MF_NBMAXACTOR; ++iActor) {
        fprintf(stream, "%f", MFWorldGetValue(that, iActor));
        if (iActor < MF_NBMAXACTOR - 1)
            fprintf(stream, ",");
    }
    fprintf(stream, "]");
    fprintf(stream, " forecast[");
    for (int iActor = 0; iActor < MF_NBMAXACTOR; ++iActor) {
        fprintf(stream, "%f", MFWorldGetForecastValue(that, iActor));
        if (iActor < MF_NBMAXACTOR - 1)
            fprintf(stream, ",");
    }
    fprintf(stream, "]");
}

// Print the MFTransition 'that' on the stream 'stream'
void MFTransitionPrint(const MFTransition* const that,
    FILE* const stream) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
        if (stream == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'stream' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
    fprintf(stream, "transition from (");
    MFModelStatePrint(
        MFWorldStatus(MFTransitionFromWorld(that)), stream);
    fprintf(stream, ") to (");
    if (MFTransitionToWorld(that) != NULL)
        MFModelStatePrint(
            MFWorldStatus(MFTransitionToWorld(that)), stream);
    else
        fprintf(stream, "<null>");
    fprintf(stream, ") through (");
    MFModelTransitionPrint((MFModelTransition*)that, stream);
    fprintf(stream, ") values[");
    for (int iActor = 0; iActor < MF_NBMAXACTOR; ++iActor) {
        fprintf(stream, "%f", that->_values[iActor]);
        if (iActor < MF_NBMAXACTOR - 1)
            fprintf(stream, ",");
    }
    fprintf(stream, "]");
}

// Print the MFWorld 'that' and its MFTransition on the stream 'stream'
void MFWorldTransPrintln(const MFWorld* const that,
    FILE* const stream) {
    #if BUILDMODE == 0
        if (that == NULL) {

```

```

    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'that' is null");
    PBErrCatch(MiniFrameErr);
}
if (stream == NULL) {
    MiniFrameErr->_type = PBErrTypeNullPointer;
    sprintf(MiniFrameErr->_msg, "'stream' is null");
    PBErrCatch(MiniFrameErr);
}
#endif
MFWorldPrint(that, stream);
fprintf(stream, "\n");
for (int iTrans = 0; iTrans < MFWorldGetNbTrans(that); ++iTrans) {
    fprintf(stream, " %d ", iTrans);
    MFTransitionPrint(MFWorldTransition(that, iTrans), stream);
    fprintf(stream, "\n");
}
}

// Set the current world of the MiniFrame 'that' to match the
// MFModelState 'status'
// If the world is in computed worlds reuse it, else create a new one
void MFSetCurWorld(MiniFrame* const that,
    const MFModelState* const status) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (status == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'status' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    // Declare a flag to memorize if we have found the world
    bool flagFound = false;
    // Loop on computed worlds
    GSetIterForward iter = GSetIterForwardCreateStatic(MFWorlds(that));
    do {
        MFWorld* world = GSetIterGet(&iter);
        // If this is the current world
        if (MFModelStateIsSame(MFWorldStatus(world), status)) {
            // Ensure that the status is exactly the same by copying the
            // MFModelState struct, in case MFModelStateIsSame refers only
            // to a subset of properties of the MFModelState
            memcpy(world, status, sizeof(MFModelState));
            // Update the curWorld in MiniFrame
            that->_curWorld = world;
            flagFound = true;
        }
    } while (!flagFound && GSetIterStep(&iter));
    // If we haven't found the searched status
    if (!flagFound) {
        // Create a new MFWorld with the current status
        MFWorld* world = MFWorldCreate(status);
        // Get the sente for the previous world
        int sente = MFModelStateGetSente(MFWorldStatus(MFCurWorld(that)));
        // Add it to the computed worlds
        MFAddWorld(that, world, sente);
        // Update the current world
    }
}

```

```

        that->_curWorld = world;
    }
}

// Free the memory used by the disposable worlds in the computed worlds
// of the MinFrame 'that'
void MFFreeDisposableWorld(MiniFrame* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PErrCatch(MiniFrameErr);
        }
    #endif
    // Declare a flag to memorize if we have found a disposable world
    bool flag = false;
    // Declare a flag to manage the deletion of element in the set of
    // computed worlds
    bool moved = false;
    // Declare a variable to memorize the number of removed world
    int nbRemovedWorld = 0;
    // Loop until we haven't found any disposable world
    do {
        // Reset the flag to memorize if we have found disposable world
        flag = false;
        // Loop on computed worlds
        GSetIterForward iter = GSetIterForwardCreateStatic(MFWorlds(that));
        do {
            MFWorld* world = GSetIterGet(&iter);
            moved = false;
            // If it's a disposable world
            if (that->_curWorld != world &&
                (GSetNbElem(MFWorldSources(world)) == 0 ||
                 MFModelStateIsDisposable(MFWorldStatus(world),
                 MFWorldStatus(MFCurWorld(that)), MFGetNbComputedWorld(that)))) {
                // Remove this world from its sources
                while (GSetNbElem(MFWorldSources(world)) > 0) {
                    MFTransition* transSource = MFWorldPopSource(world);
                    MFTransitionSetToWorld(transSource, NULL);
                }
                // Remove this world from the sources of its next worlds
                for (int iTrans = MFWorldGetNbTrans(world); iTrans--;) {
                    const MFTransition* trans = MFWorldTransition(world, iTrans);
                    MFWorld* toWorld = (MFWorld*)MFTransitionToWorld(trans);
                    if (toWorld != NULL)
                        MFWorldRemoveSource(toWorld, trans);
                }
                // Remove this world from the computed worlds
                moved = GSetIterRemoveElem(&iter);
                // Free memory
                MFWorldFree(&world);
                // Increment the number of removed world
                ++nbRemovedWorld;
                // Memorize we have found a disposable world
                flag = true;
            }
        } while (moved || GSetIterStep(&iter));
    } while (flag == true);
    // Update the number of removed world
    that->_nbRemovedWorld = nbRemovedWorld;
}

```

```

// Remove the MFTransition 'source' from the sources of the
// MFWorld 'that'
void MFWorldRemoveSource(MFWorld* const that,
    const MFTransition* const source) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (source == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'source' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    // Loop on transitions
    if (GSetNbElem(MFWorldSources(that)) > 0) {
        bool moved = false;
        GSetIterForward iter =
            GSetIterForwardCreateStatic(MFWorldSources(that));
        do {
            moved = false;
            MFTransition* trans = GSetIterGet(&iter);
            if (trans == source) {
                moved = GSetIterRemoveElem(&iter);
            }
        } while (moved || GSetIterStep(&iter));
    }
}

// Pop a MFTransition from the sources of the MFWorld 'that'
#ifdef BUILDMODE != 0
inline
#endif
MFTransition* MFWorldPopSource(MFWorld* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    return GSetPop(&(that->_sources));
}

// Print the best forecasted story from the MFWorld 'that' for the
// actor 'iActor' on the stream 'stream'
void MFWorldPrintBestStoryln(const MFWorld* const that, const int iActor,
    FILE* const stream) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
        MiniFrameErr->_type = PBErrTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<=%d)",
            iActor, MF_NBMAXACTOR);
        PBErrCatch(MiniFrameErr);
    }
#endif
}

```

```

    }
    if (stream == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'stream' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    // Declare a variable to memorize the current displayed world
    const MFWorld* curWorld = that;
    // Declare a GSet to manage infinite loop
    GSet setWorld = GSetCreateStatic();
    // Loop until the end of the forecast
    while (curWorld != NULL) {
        // Display the current world
        //MFWorldPrint(curWorld, stream);
        //fprintf(stream, "\n");
        MFWorldTransPrintln(curWorld, stream);
        // Add the world to the set of visited worlds
        GSetAppend(&setWorld, (void*)curWorld);
        // If we are not at an end status
        if (!MFModelStatusIsEnd(MFWorldStatus(curWorld))) {
            // Get the sente for the current world
            int sente = MFModelStatusGetSente(MFWorldStatus(curWorld));
            // If it's a simultaneous game
            if (sente == -1)
                sente = iActor;
            // Get the best transition from this world
            const MFModelTransition* bestTrans =
                MFWorldBestTransition(curWorld, sente);
            // If there is no transition
            if (bestTrans == NULL) {
                // Stop the story here
                curWorld = NULL;
            } // Else, there is a best transition
            else {
                // Print the best transition
                fprintf(stream, "--> ");
                MFTransitionPrint((const MFTransition*)bestTrans, stream);
                fprintf(stream, "\n");
                // Move to the world resulting from the best transition
                curWorld = MFTransitionToWorld((const MFTransition*)bestTrans);
            }
        } // else {
        fprintf(stream, "--> reached a end status\n");
        curWorld = NULL;
    }
    // If we reach a world already visited
    if (curWorld != NULL && GSetFirstElem(&setWorld, curWorld) != NULL) {
        MFWorldPrint(curWorld, stream);
        fprintf(stream, "\n");
        fprintf(stream, "--> infinite loop in best story, quit\n");
        curWorld = NULL;
    }
}
// Free memory
GSetFlush(&setWorld);
}

// Set the values of the MFWorld 'that' to 'values'
void MFWorldSetValues(MFWorld* const that, const float* const values) {
#ifdef BUILDMODE == 0
    if (that == NULL) {

```

```

        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (values == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'values' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    for (int iActor = MF_NBMAXACTOR; iActor--;) {
        that->_values[iActor] = 0.0;
        for (int jActor = MF_NBMAXACTOR; jActor--;) {
            if (iActor == jActor)
                that->_values[iActor] += values[jActor];
            else
                that->_values[iActor] -= values[jActor];
        }
    }
}

// Get the number of expandable transition for the MFWorld 'that'
int MFWorldGetNbTransExpandable(const MFWorld* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
    // Declare a variable to memorize the result
    int nb = 0;
    // Loop on transitions
    for (int iTrans = MFWorldGetNbTrans(that); iTrans--;) {
        // Get the transition
        const MFTransition* const trans = MFWorldTransition(that, iTrans);
        // If this transition is expandable
        if (MFTransitionIsExpandable(trans))
            // Increment the result
            ++nb;
    }
    // Return the result
    return nb;
}

```

2.2 miniframe-inline.c

```

// ===== MINIFRAME_INLINE.C =====

// ===== Functions implementation =====

// Get the time limit for expansion of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetMaxTimeExpansion(const MiniFrame* const that) {
    #if BUILDMODE == 0

```



```

    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    return that->_maxTimeExpansion;
}

// Get the time unused during last expansion of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetTimeUnusedExpansion(const MiniFrame* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    return that->_timeUnusedExpansion;
}

// Get the time used to search world to expand during next expansion
// of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetTimeSearchWorld(const MiniFrame* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    return that->_timeSearchWorld;
}

// Get the nb of world expanded during the last expansion
// of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbWorldExpanded(const MiniFrame* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    return that->_nbWorldExpanded;
}

// Get the nb of world unexpanded during the last expansion
// of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbWorldUnexpanded(const MiniFrame* const that) {

```

```

#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    return that->_nbWorldUnexpanded;
}

// Get the time used at end of expansion of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetTimeEndExpansion(const MiniFrame* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    return that->_timeEndExpansion;
}

// Get the clock considered has start during expansion
#if BUILDMODE != 0
inline
#endif
clock_t MFGetStartExpandClock(const MiniFrame* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    return that->_startExpandClock;
}

// Set the clock considered has start during expansion to 'c'
#if BUILDMODE != 0
inline
#endif
void MFSetStartExpandClock(MiniFrame* const that, clock_t c) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    that->_startExpandClock = c;
}

// Set the time limit for expansion of the MiniFrame 'that' to
// 'timeLimit', in millisecond
// The time is measured with the function clock(), see "man clock"
// for details
#if BUILDMODE != 0
inline
#endif

```

```

void MFSetMaxTimeExpansion(MiniFrame* const that, const float timeLimit) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
#endif
    that->_maxTimeExpansion = timeLimit;
}

// Get the current MFWorld of the MiniFrame 'that'
#ifdef BUILDMODE != 0
inline
#endif
const MFWorld* MFCurWorld(const MiniFrame* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
#endif
    return that->_curWorld;
}

// Get the GSet of computed MFWorlds of the MiniFrame 'that'
#ifdef BUILDMODE != 0
inline
#endif
const GSet* MFWorlds(const MiniFrame* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
#endif
    return &(that->_worlds);
}

// Add the MFWorld 'world' to the computed MFWorlds of the
// MiniFrame 'that', ordered by the world's value from the pov of
// actor 'iActor'
#ifdef BUILDMODE != 0
inline
#endif
void MFAddWorld(MiniFrame* const that, \
    const MFWorld* const world, const int iActor) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
    if (world == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'world' is null");
        PErrCatch(MiniFrameErr);
    }
    if (iActor < -1 || iActor >= MF_NBMAXACTOR) {
        MiniFrameErr->_type = PErrTypeInvalidArg;
    }
#endif
}

```

```

        sprintf(MiniFrameErr->_msg, "'iActor' is invalid (-1<=%d<%d)",
                iActor, MF_NBMAXACTOR);
        PBErriCatch(MiniFrameErr);
    }
#endif
    GSetAddSort(&(that->_worlds), world,
        MFWorldGetForecastValue(world, iActor));
}

// Return the MFModelStatus of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
const MFModelStatus* MFWorldStatus(const MFWorld* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErriTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErriCatch(MiniFrameErr);
    }
#endif
    return (const MFModelStatus*)that;
}

// Get the number of transition for the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
int MFWorldGetNbTrans(const MFWorld* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErriTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErriCatch(MiniFrameErr);
    }
#endif
    return that->_nbTransition;
}

// Get the percentage of resued world of the MiniFrame 'that' during
// the last MFEpxand()
#if BUILDMODE != 0
inline
#endif
float MFGetPercWordReused(const MiniFrame* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErriTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErriCatch(MiniFrameErr);
    }
#endif
    return that->_percWorldReused;
}

// Get the 'iTrans' MFTransition of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
const MFTransition* MFWorldTransition(const MFWorld* const that,
    const int iTrans) {
#if BUILDMODE == 0

```

```

    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (iTrans < 0 || iTrans >= that->_nbTransition) {
        MiniFrameErr->_type = PBErrTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "'iTrans' is invalid (0<=%d<%d)",
            iTrans, that->_nbTransition);
        PBErrCatch(MiniFrameErr);
    }
}
#endif
return that->_transitions + iTrans;
}

// Compute the MFModelState resulting from the 'iTrans' MFTransition
// of the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
MFModelState MFWorldComputeTransition(const MFWorld* const that,
    const int iTrans) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
        if (iTrans < 0 || iTrans >= that->_nbTransition) {
            MiniFrameErr->_type = PBErrTypeInvalidArg;
            sprintf(MiniFrameErr->_msg, "'iTrans' is invalid (0<=%d<%d)",
                iTrans, that->_nbTransition);
            PBErrCatch(MiniFrameErr);
        }
    #endif
    // Return the resulting MFModelState
    return MFModelStateStep((const MFModelState* const)that,
        (const MFModelTransition* const)MFWorldTransition(that, iTrans));
}

// Return true if the expansion algorithm looks in previously
// computed worlds for same world to reuse, else false
#if BUILDMODE != 0
inline
#endif
bool MFIsWorldReusable(const MiniFrame* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
    return that->_reuseWorld;
}

// Set the flag controlling if the expansion algorithm looks in
// previously computed worlds for same world to reuse to 'reuse'
#if BUILDMODE != 0
inline
#endif
void MFSetWorldReusable(MiniFrame* const that, const bool reuse) {

```

```

#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    that->_reuseWorld = reuse;
}

// Get the MFWorld which the MFTransition 'that' is leading to
#if BUILDMODE != 0
inline
#endif
const MFWorld* MFTransitionToWorld(const MFTransition* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    return that->_toWorld;
}

// Set the MFWorld to which the MFTransition 'that' is leading to
// 'world'
#if BUILDMODE != 0
inline
#endif
void MFTransitionSetToWorld(MFTransition* const that,
    MFWorld* const world) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    that->_toWorld = world;
}

// Get the MFWorld which the MFTransition 'that' is coming from
#if BUILDMODE != 0
inline
#endif
const MFWorld* MFTransitionFromWorld(const MFTransition* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    return that->_fromWorld;
}

// Set the value of the MFTransition 'that' for the actor 'iActor' to
// 'val'
#if BUILDMODE != 0
inline
#endif

```

```

void MFTransitionSetValue(MFTransition* const that, const int iActor,
    const float val) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
        if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
            MiniFrameErr->_type = PBErrTypeInvalidArg;
            sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<=%d)",
                iActor, MF_NBMAXACTOR);
            PBErrCatch(MiniFrameErr);
        }
    #endif
    that->_values[iActor] = val;
}

// Return the number of computed worlds in the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbComputedWorld(const MiniFrame* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
    return GSetNbElem(&(that->_worlds));
}

// Get the nb of removed world during the last call to SetCurWorld
// of the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbWorldRemoved(const MiniFrame* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
    return that->_nbRemovedWorld;
}

// Return the value of the MFWorld 'that' for the
// actor 'iActor'.
#if BUILDMODE != 0
inline
#endif
float MFWorldGetValue(const MFWorld* const that, const int iActor) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
}

```

```

    if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
        MiniFrameErr->_type = PBErrTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<%d)", \
            iActor, MF_NBMAXACTOR);
        PBErrCatch(MiniFrameErr);
    }
#endif
    return that->_values[iActor];
}

// Return the value of the MFTransition 'that' for the
// actor 'iActor'.
#if BUILDMODE != 0
inline
#endif
float MFTransitionGetValue(const MFTransition* const that,
    const int iActor) {
    if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
        if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
            MiniFrameErr->_type = PBErrTypeInvalidArg;
            sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<%d)", \
                iActor, MF_NBMAXACTOR);
            PBErrCatch(MiniFrameErr);
        }
    }
#endif
    return that->_values[iActor];
}

// Get the set of MFTransition reaching the MFWorld 'that'
#if BUILDMODE != 0
inline
#endif
const GSet* MFWorldSources(const MFWorld* const that) {
    if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
    }
#endif
    return &(that->_sources);
}

// Return the array of values of the MFWorld 'that' for each actor
#if BUILDMODE != 0
inline
#endif
const float* MFWorldValues(const MFWorld* const that) {
    if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
    }
#endif
}

```



```

    return that->_values;
}

// Return the max depth during expansion for the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetMaxDepthExp(const MiniFrame* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
#endif
    return that->_maxDepthExp;
}

// Set the max depth during expansion for the MiniFrame 'that' to 'depth'
// If depth is less than -1 it is converted to -1
#if BUILDMODE != 0
inline
#endif
void MFSetMaxDepthExp(MiniFrame* const that, const int depth) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
#endif
    that->_maxDepthExp = MAX(-1, depth);
}

// Return the type of expansion for the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
MFEExpansionType MFGetExpansionType(const MiniFrame* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
#endif
    return that->_expansionType;
}

// Set the type expansion for the MiniFrame 'that' to 'type'
#if BUILDMODE != 0
inline
#endif
void MFSetExpansionType(MiniFrame* const that, const MFEExpansionType type) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
#endif
    that->_expansionType = type;
}

```

```

}

// Set the nb of transitions to activate MonteCarlo during expansion
// for the MiniFrame 'that' to 'nb'
#if BUILDMODE != 0
inline
#endif
void MFSetNbTransMonteCarlo(MiniFrame* const that, const int nb) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
    if (nb <= 0) {
        MiniFrameErr->_type = PErrTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "'nb' is invalid (%d>0)", nb);
        PErrCatch(MiniFrameErr);
    }
}
#endif
that->_nbTransMonteCarlo = nb;
}

// Get the nb of transitions to activate MonteCarlo during expansion
// for the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
int MFGetNbTransMonteCarlo(const MiniFrame* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
}
#endif
return that->_nbTransMonteCarlo;
}

// Return true if the MFTransition is expanded, false else
#if BUILDMODE != 0
inline
#endif
bool MFTransitionIsExpanded(const MFTransition* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
}
#endif
return (that->_toWorld != NULL);
}

// Set the pruning threshold during expansion for the MiniFrame 'that'
// to 'val'
#if BUILDMODE != 0
inline
#endif
void MFSetPruningDeltaVal(MiniFrame* const that, const float val) {
#if BUILDMODE == 0
    if (that == NULL) {

```

```

        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    that->_pruningDeltaVal = val;
}

// Get the pruning threshold during expansion for the MiniFrame 'that'
#if BUILDMODE != 0
inline
#endif
float MFGetPruningDeltaVal(const MiniFrame* const that) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    return that->_pruningDeltaVal;
}

```

3 Makefile

```

# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=1

all: main

# Makefile definitions
MAKEFILE_INC=../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Path to the model implementation
MF_MODEL_PATH=$(ROOT_DIR)/MiniFrame/Examples/BasicExample

# Rules to make the executable
repo=miniframe
$$(repo)_EXENAME: \
createLinkToModelHeader \
miniframe-model.o \
$$(repo)_EXENAME.o \
$$(repo)_EXE_DEP \
$$(repo)_DEP
$(COMPILER) 'echo "$$(repo)_EXE_DEP' '$$(repo)_EXENAME.o' | tr ' ' '\n' | sort -u' miniframe-model.o $(LINK_ARG) $

$$(repo)_EXENAME.o: \
$(MF_MODEL_PATH)/miniframe-model.h \
$$(repo)_DIR/$$(repo)_EXENAME.c \
$$(repo)_INC_H_EXE \
$$(repo)_EXE_DEP
$(COMPILER) $(BUILD_ARG) $$(repo)_BUILD_ARG 'echo "$$(repo)_INC_DIR" | tr ' ' '\n' | sort -u' -c $$(repo)_DIR)/

```

```

createLinkToModelHeader:
ln -s -f $(MF_MODEL_PATH)/miniframe-model.h $(repo)_DIR/miniframe-model.h; ln -s -f $(MF_MODEL_PATH)/miniframe-in.

miniframe-model.o: \
$(MF_MODEL_PATH)/miniframe-model.h \
$(MF_MODEL_PATH)/miniframe-model.c \
Makefile
$(COMPILER) $(BUILD_ARG) -c $(MF_MODEL_PATH)/miniframe-model.c

```

4 Unit tests

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "pbmath.h"
#include "miniframe.h"

#define RANDOMSEED 0

void UnitTestMFTransitionCreateFree() {

    MFWorld world;
    MFModelTransition trans = {._move = 1};
    MFTransition act = MFTransitionCreateStatic(&world, &trans);
    if (act._fromWorld != &world ||
        act._toWorld != NULL ||
        memcmp(&(act._transition), &(trans),
            sizeof(MFModelTransition)) != 0) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFTransitionCreateStatic failed");
        PBErrCatch(MiniFrameErr);
    }
    for (int iActor = MF_NBMAXACTOR; iActor--;)
        if (ISEQUALF(act._values[iActor], 0.0) == false) {
            MiniFrameErr->_type = PBErrTypeUnitTestFailed;
            sprintf(MiniFrameErr->_msg, "MFTransitionCreateStatic failed");
            PBErrCatch(MiniFrameErr);
        }
    MFTransitionFreeStatic(&act);

    printf("UnitTestMFTransitionCreateFree OK\n");
}

void UnitTestMFTransitionIsExpandable() {

    MFModelStatus status;
    MFWorld* world = MFWorldCreate(&status);
    MFModelTransition trans = {._move = 1};
    MFTransition act = MFTransitionCreateStatic(world, &trans);
    if (!MFTransitionIsExpandable(&act)) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFTransitionIsExpandable failed");
        PBErrCatch(MiniFrameErr);
    }
}

```

```

    }
    act._toWorld = world;
    if (MFTransitionIsExpandable(&act)) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFTransitionIsExpandable failed");
        PBErrCatch(MiniFrameErr);
    }
    act._toWorld = NULL;
    world->_status._pos = world->_status._tgt;
    world->_transitions[0]._toWorld = world;
    if (MFTransitionIsExpandable(&act)) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFTransitionIsExpandable failed");
        PBErrCatch(MiniFrameErr);
    }
    world->_transitions[0]._toWorld = NULL;
    MFTransitionFreeStatic(&act);
    MFWorldFree(&world);

    printf("UnitTestMFTransitionIsExpandable OK\n");
}

void UnitTestMFTransitionIsExpanded() {
    MFModelState status;
    MFWorld* world = MFWorldCreate(&status);
    MFModelTransition trans = {._move = 1};
    MFTransition act = MFTransitionCreateStatic(world, &trans);
    if (MFTransitionIsExpanded(&act)) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFTransitionIsExpanded failed");
        PBErrCatch(MiniFrameErr);
    }
    act._toWorld = world;
    if (!MFTransitionIsExpanded(&act)) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFTransitionIsExpanded failed");
        PBErrCatch(MiniFrameErr);
    }
    MFTransitionFreeStatic(&act);
    MFWorldFree(&world);

    printf("UnitTestMFTransitionIsExpanded OK\n");
}

void UnitTestMFTransitionGetSet() {
    MFWorld worldFrom;
    MFWorld worldTo;
    MFModelTransition trans = {._move = 1};
    MFTransition act = MFTransitionCreateStatic(&worldFrom, &trans);
    act._toWorld = &worldTo;
    if (MFTransitionToWorld(&act) != &worldTo) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFTransitionToWorld failed");
        PBErrCatch(MiniFrameErr);
    }
    if (MFTransitionFromWorld(&act) != &worldFrom) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFTransitionFromWorld failed");
        PBErrCatch(MiniFrameErr);
    }
    MFTransitionSetValue(&act, 0, 1.0);
}

```

```

    if (ISEQUALF(act._values[0], 1.0) == false) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFTransitionSetValue failed");
        PBErrCatch(MiniFrameErr);
    }
    if (ISEQUALF(MFTransitionGetValue(&act, 0), 1.0) == false) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFTransitionGetValue failed");
        PBErrCatch(MiniFrameErr);
    }
    MFWorld worldB;
    MFTransitionSetToWorld(&act, &worldB);
    if (MFTransitionToWorld(&act) != &worldB) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFTransitionSetToWorld failed");
        PBErrCatch(MiniFrameErr);
    }
    MFTransitionFreeStatic(&act);

    printf("UnitTestMFTransitionGetSet OK\n");
}

void UnitTestMFTransition() {
    UnitTestMFTransitionCreateFree();
    UnitTestMFTransitionIsExpandable();
    UnitTestMFTransitionIsExpanded();
    UnitTestMFTransitionGetSet();
    printf("UnitTestMFTransition OK\n");
}

void UnitTestMFWorldCreateFree() {
    MFModelState modelWorld = {._step = 0, ._pos = 0, ._tgt = 1};
    MFWorld* world = MFWorldCreate(&modelWorld);
    if (world == NULL ||
        GSetNbElem(&(world->_sources)) != 0 ||
        world->_nbTransition != 3) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFWorldCreate failed");
        PBErrCatch(MiniFrameErr);
    }
    float val[MF_NBMAXACTOR] = {0.0};
    val[0] = -1.0;
    for (int iActor = MF_NBMAXACTOR; iActor--;)
        if (ISEQUALF(world->_values[iActor], val[iActor]) == false) {
            MiniFrameErr->_type = PBErrTypeUnitTestFailed;
            sprintf(MiniFrameErr->_msg, "MFWorldCreate failed");
            PBErrCatch(MiniFrameErr);
        }
    MFWorldFree(&world);
    if (world != NULL) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFWorldFree failed");
        PBErrCatch(MiniFrameErr);
    }

    printf("UnitTestMFWorldCreateFree OK\n");
}

void UnitTestMFWorldGetSet() {
    MFModelState modelWorld = {._step = 0, ._pos = 0, ._tgt = 1};
    MFWorld* world = MFWorldCreate(&modelWorld);

```

```

    if (MFWorldStatus(world) != &(world->_status)) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFWorldStatus failed");
        PBErrCatch(MiniFrameErr);
    }
    if (MFWorldGetNbTrans(world) != 3) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFWorldGetNbTrans failed");
        PBErrCatch(MiniFrameErr);
    }
    if (MFWorldSources(world) != &(world->_sources)) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFWorldSources failed");
        PBErrCatch(MiniFrameErr);
    }
    if (MFWorldValues(world) != world->_values) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFWorldValues failed");
        PBErrCatch(MiniFrameErr);
    }
    if (MFWorldTransition(world, 0) != world->_transitions ||
        MFWorldTransition(world, 1) != world->_transitions + 1 ||
        MFWorldTransition(world, 2) != world->_transitions + 2) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFWorldTransition failed");
        PBErrCatch(MiniFrameErr);
    }
    world->_values[0] = 1.0;
    if (ISEQUALF(MFWorldGetValue(world, 0), 1.0) == false) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFWorldGetValue failed");
        PBErrCatch(MiniFrameErr);
    }
    MFWorldFree(&world);
    printf("UnitTestMFWorldGetSet OK\n");
}

void UnitTestMFWorldComputeTransition() {
    MFModelState modelWorld = {._step = 0, ._pos = 0, ._tgt = 1};
    MFWorld* world = MFWorldCreate(&modelWorld);
    MFModelState statusA = {._step = 1, ._pos = -1, ._tgt = 1};
    MFModelState statusB = {._step = 1, ._pos = 0, ._tgt = 1};
    MFModelState statusC = {._step = 1, ._pos = 1, ._tgt = 1};
    MFModelState status = MFWorldComputeTransition(world, 0);
    if (memcmp(&status, &statusA, sizeof(MFModelState)) != 0) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFWorldComputeTransition failed");
        PBErrCatch(MiniFrameErr);
    }
    status = MFWorldComputeTransition(world, 1);
    if (memcmp(&status, &statusB, sizeof(MFModelState)) != 0) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFWorldComputeTransition failed");
        PBErrCatch(MiniFrameErr);
    }
    status = MFWorldComputeTransition(world, 2);
    if (memcmp(&status, &statusC, sizeof(MFModelState)) != 0) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFWorldComputeTransition failed");
        PBErrCatch(MiniFrameErr);
    }
    MFWorldFree(&world);
}

```

```

    printf("UnitTestMFWorldComputeTransition OK\n");
}

void UnitTestMFWorld() {
    UnitTestMFWorldCreateFree();
    UnitTestMFWorldGetSet();
    UnitTestMFWorldComputeTransition();
    printf("UnitTestMFWorld OK\n");
}

void UnitTestMiniFrameCreateFree() {

    MFModelState initStatus = {._step = 0, ._pos = 0, ._tgt = 1};
    MiniFrame* mf = MiniFrameCreate(&initStatus);
    if (mf == NULL ||
        mf->_nbStep != 0 ||
        ISEQUALF(mf->_maxTimeExpansion, MF_DEFAULTTIMEEXPANSION) == false ||
        MFModelStateIsSame(&initStatus, &(MFCurWorld(mf)->_status)) == false ||
        MFCurWorld(mf) != GSetGet(MFWorlds(mf), 0) ||
        GSetNbElem(MFWorlds(mf)) != 1 ||
        ISEQUALF(mf->_timeUnusedExpansion, 0.0) == false ||
        ISEQUALF(mf->_percWorldReused, 0.0) == false ||
        mf->_nbWorldExpanded != 0 ||
        mf->_nbWorldUnexpanded != 0 ||
        mf->_nbRemovedWorld != 0 ||
        mf->_timeEndExpansion <= 0.0 ||
        mf->_maxDepthExp != -1 ||
        mf->_expansionType != MFExpansionTypeValue ||
        mf->_nbTransMonteCarlo != MF_NBTRANSMONTECARLO ||
        mf->_pruningDeltaVal != MF_PRUNINGDELTAVAL ||
        mf->_reuseWorld != false) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MiniFrameCreate failed");
        PBErrCatch(MiniFrameErr);
    }
    MiniFrameFree(&mf);
    if (mf != NULL) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MiniFrameFree failed");
        PBErrCatch(MiniFrameErr);
    }

    printf("UnitTestMiniFrameCreateFree OK\n");
}

void UnitTestMiniFrameGetSet() {
    MFModelState initWorld = {._step = 0, ._pos = 0, ._tgt = 1};
    MiniFrame* mf = MiniFrameCreate(&initWorld);
    if (ISEQUALF(MFGetMaxTimeExpansion(mf),
        mf->_maxTimeExpansion) == false) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFGetMaxTimeExpansion failed");
        PBErrCatch(MiniFrameErr);
    }
    if (MFGetNbComputedWorld(mf) != 1) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFGetNbComputedWorld failed");
        PBErrCatch(MiniFrameErr);
    }
    float t = MF_DEFAULTTIMEEXPANSION + 1.0;
    MFSetMaxTimeExpansion(mf, t);
    if (ISEQUALF(MFGetMaxTimeExpansion(mf), t) == false) {

```



```

    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFSetMaxTimeExpansion failed");
    PBErrCatch(MiniFrameErr);
}
if (ISEQUALF(MFGetTimeEndExpansion(mf),
    mf->_timeEndExpansion) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetTimeEndExpansion failed");
    PBErrCatch(MiniFrameErr);
}
if (MFCurWorld(mf) != mf->_curWorld) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFCurWorld failed");
    PBErrCatch(MiniFrameErr);
}
if (MFWorlds(mf) != &(mf->_worlds)) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorlds failed");
    PBErrCatch(MiniFrameErr);
}
if (MFIsWorldReusable(mf) != mf->_reuseWorld) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFIsWorldReusable failed");
    PBErrCatch(MiniFrameErr);
}
bool reuse = !MFIsWorldReusable(mf);
MFSetWorldReusable(mf, reuse);
if (MFIsWorldReusable(mf) != reuse) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFSetWorldReusable failed");
    PBErrCatch(MiniFrameErr);
}
mf->_percWorldReused = 1.0;
if (ISEQUALF(MFGetPercWordReused(mf), 1.0) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetPercWordReused failed");
    PBErrCatch(MiniFrameErr);
}
MFModelState modelWorld = {._step = 0, ._pos = 0, ._tgt = 1};
MFWorld* world = MFWorldCreate(&modelWorld);
MFAddWorld(mf, world, 0);
if (GSetNbElem(MFWorlds(mf)) != 2 ||
    MFModelStateIsSame(MFWorldStatus(world),
        (MFModelState*)GSetGet(MFWorlds(mf), 1)) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFAddWorld failed");
    PBErrCatch(MiniFrameErr);
}
mf->_nbWorldExpanded = 1;
if (MFGetNbWorldExpanded(mf) != mf->_nbWorldExpanded) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetNbWorldExpanded failed");
    PBErrCatch(MiniFrameErr);
}
mf->_nbWorldUnexpanded = 1;
if (MFGetNbWorldUnexpanded(mf) != mf->_nbWorldUnexpanded) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFGetNbWorldUnexpanded failed");
    PBErrCatch(MiniFrameErr);
}
mf->_timeSearchWorld = 2.0;
if (ISEQUALF(MFGetTimeSearchWorld(mf),

```

```

    mf->_timeSearchWorld) == false) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFGetTimeSearchWorld failed");
        PBErrCatch(MiniFrameErr);
    }
    mf->_timeUnusedExpansion = 3.0;
    if (ISEQUALF(MFGetTimeUnusedExpansion(mf),
        mf->_timeUnusedExpansion) == false) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFGetTimeUnusedExpansion failed");
        PBErrCatch(MiniFrameErr);
    }
    mf->_percWorldReused = 4.0;
    if (ISEQUALF(MFGetPercWordReused(mf),
        mf->_percWorldReused) == false) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFGetPercWordReused failed");
        PBErrCatch(MiniFrameErr);
    }
    clock_t now = clock();
    MFSetStartExpandClock(mf, now);
    if (mf->_startExpandClock != now) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFGetStartExpandClock failed");
        PBErrCatch(MiniFrameErr);
    }
    if (MFGetStartExpandClock(mf) != now) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFGetStartExpandClock failed");
        PBErrCatch(MiniFrameErr);
    }
    if (MFGetMaxDepthExp(mf) != mf->_maxDepthExp) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFGetMaxDepthExp failed");
        PBErrCatch(MiniFrameErr);
    }
    MFSetMaxDepthExp(mf, 3);
    if (MFGetMaxDepthExp(mf) != 3) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFSetMaxDepthExp failed");
        PBErrCatch(MiniFrameErr);
    }
    MFSetMaxDepthExp(mf, -2);
    if (MFGetMaxDepthExp(mf) != -1) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFSetMaxDepthExp failed");
        PBErrCatch(MiniFrameErr);
    }
    if (MFGetExpansionType(mf) != mf->_expansionType) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFGetExpansionType failed");
        PBErrCatch(MiniFrameErr);
    }
    MFSetExpansionType(mf, MFEExpansionTypeWidth);
    if (MFGetExpansionType(mf) != MFEExpansionTypeWidth) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFSetExpansionType failed");
        PBErrCatch(MiniFrameErr);
    }
    if (MFGetNbTransMonteCarlo(mf) != mf->_nbTransMonteCarlo) {
        MiniFrameErr->_type = PBErrTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFGetNbTransMonteCarlo failed");
    }

```

```

        PBErCatch(MiniFrameErr);
    }
    MFSetNbTransMonteCarlo(mf, 10);
    if (MFGetNbTransMonteCarlo(mf) != 10) {
        MiniFrameErr->_type = PBErTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFSetNbTransMonteCarlo failed");
        PBErCatch(MiniFrameErr);
    }
    if (MFGetPruningDeltaVal(mf) != mf->_pruningDeltaVal) {
        MiniFrameErr->_type = PBErTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFGetPruningDeltaVal failed");
        PBErCatch(MiniFrameErr);
    }
    MFSetPruningDeltaVal(mf, 10.0);
    if (!ISEQUALF(MFGetPruningDeltaVal(mf), 10.0)) {
        MiniFrameErr->_type = PBErTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFSetPruningDeltaVal failed");
        PBErCatch(MiniFrameErr);
    }
    MiniFrameFree(&mf);
    printf("UnitTestMiniFrameGetSet OK\n");
}

void UnitTestMiniFrameExpandSetCurWorld() {
    MFModelStatus initWorld = {._step = 0, ._pos = 0, ._tgt = 2};
    MiniFrame* mf = MiniFrameCreate(&initWorld);
    MFSetStartExpandClock(mf, clock());
    MFSetWorldReusable(mf, true);
    MFExpand(mf);
    printf("Time unused by MFExpand: %f\n", MFGetTimeUnusedExpansion(mf));
    printf("Time search world to expand: %f\n", MFGetTimeSearchWorld(mf));
    printf("Nb world expanded: %d\n", MFGetNbWorldExpanded(mf));
    printf("Nb world unexpanded: %d\n", MFGetNbWorldUnexpanded(mf));
    printf("Nb world removed: %d\n", MFGetNbWorldRemoved(mf));
    printf("Perc world reused: %f\n", MFGetPercWordReused(mf));
    printf("Computed worlds:\n");
    GSetIterForward iter = GSetIterForwardCreateStatic(MFWorlds(mf));
    do {
        MFWorld* world = GSetIterGet(&iter);
        MFWorldTransPrintln(world, stdout);
    } while (GSetIterStep(&iter));
    if (mf->_timeUnusedExpansion < 0.0 ||
        MFGetNbWorldExpanded(mf) != 15 ||
        MFGetNbWorldUnexpanded(mf) != 0 ||
        MFGetNbWorldRemoved(mf) != 0 ||
        ISEQUALF(MFGetPercWordReused(mf), 0.666667) == false ||
        ISEQUALF(MFGetTimeSearchWorld(mf), 100.0) == false) {
        MiniFrameErr->_type = PBErTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFExpand failed");
        PBErCatch(MiniFrameErr);
    }
    const MFModelTransition* bestTrans = MFBestTransition(mf, 0);
    printf("Best action: %d\n", bestTrans->_move);
    if (bestTrans->_move != 1) {
        MiniFrameErr->_type = PBErTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFGetBestTransition failed");
        PBErCatch(MiniFrameErr);
    }
    if (ISEQUALF(MFWorldGetForecastValue(MFCurWorld(mf), 0), 0.0) == false) {
        MiniFrameErr->_type = PBErTypeUnitTestFailed;
        sprintf(MiniFrameErr->_msg, "MFWorldGetPOVValue failed");
        PBErCatch(MiniFrameErr);
    }
}

```

```

}
if (ISEQUALF(
    MFWorldGetForecastValue(MFCurWorld(mf), 0), 0.0) == false) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFWorldGetForecastValue failed");
    PBErrCatch(MiniFrameErr);
}
MFModelState nextWorld = {._pos = -1, ._tgt = 2};
MFSetCurWorld(mf, &nextWorld);
if (MFCurWorld(mf) != GSetGet(MFWorlds(mf), 2) ||
    MFGetNbComputedWorld(mf) != 6) {
    MiniFrameErr->_type = PBErrTypeUnitTestFailed;
    sprintf(MiniFrameErr->_msg, "MFSetCurWorld failed");
    PBErrCatch(MiniFrameErr);
}
MiniFrameFree(&mf);
printf("UnitTestMiniFrameExpandSetCurWorld OK\n");
}

void UnitTestMiniFrameFullExample() {
    // Initial world
    MFModelState curWorld = {._step = 0, ._pos = 0, ._tgt = 2};
    // Create the MiniFrame
    MiniFrame* mf = MiniFrameCreate(&curWorld);
    // Set reusable worlds
    MFSetWorldReusable(mf, true);
    // Loop until end of game
    int tgt[7] = {2,2,-1,-1,-1,-1,-1};
    while (!MFModelStateIsEnd(&curWorld)) {
        // Set the start clock
        MFSetStartExpandClock(mf, clock());
        // Correct the current world in the MiniFrame
        MFSetCurWorld(mf, &curWorld);
        // Expand
        MFExpand(mf);
        // Get best transition
        const MFModelTransition* bestTrans = MFBestTransition(mf, 0);
        if (bestTrans != NULL) {
            // Step with best transition
            curWorld = MFModelStateStep(&curWorld, bestTrans);
        }
        // Apply external forces to the world
        curWorld._tgt = tgt[curWorld._step];
        // Display the current world
        printf("mf(");
        MFModelStatePrint(MFWorldStatus(MFCurWorld(mf)), stdout);
        printf(") real(");
        MFModelStatePrint(&curWorld, stdout);
        printf(")\n");
        /*MFWorldTransPrintln(MFCurWorld(mf), stdout);
        printf("--- start of best story ---\n");
        MFWorldPrintBestStoryln(MFCurWorld(mf), 0, stdout);
        printf("--- end of best story ---\n");
        printf("\n");*/
    }
    MiniFrameFree(&mf);
    printf("UnitTestMiniFrameFullExample OK\n");
}

void UnitTestMiniFrame() {
    UnitTestMiniFrameCreateFree();
    UnitTestMiniFrameGetSet();
}

```

```

    UnitTestMiniFrameExpandSetCurWorld();
    UnitTestMiniFrameFullExample();
    printf("UnitTestMiniFrame OK\n");
}

void UnitTestAll() {
    UnitTestMFTransition();
    UnitTestMFWorld();
    UnitTestMiniFrame();
    printf("UnitTestAll OK\n");
}

int main() {
    UnitTestAll();
    // Return success code
    return 0;
}

```

5 Unit tests output

```

UnitTestMFTransitionCreateFree OK
UnitTestMFTransitionIsExpandable OK
UnitTestMFTransitionIsExpanded OK
UnitTestMFTransitionGetSet OK
UnitTestMFTransition OK
UnitTestMFWorldCreateFree OK
UnitTestMFWorldGetSet OK
UnitTestMFWorldComputeTransition OK
UnitTestMFWorld OK
UnitTestMiniFrameCreateFree OK
UnitTestMiniFrameGetSet OK
Time unused by MFExpand: 99.987000
Time search world to expand: 100.000000
Nb world expanded: 15
Nb world unexpanded: 0
Nb world removed: 0
Perc world reused: 0.666667
Computed worlds:
(step:3 pos:-3 tgt:2) values[-5.000000] forecast[-0.000030]
  0) transition from (step:3 pos:-3 tgt:2) to (step:3 pos:-3 tgt:2) through (move:-1) values[-4.000000]
  1) transition from (step:3 pos:-3 tgt:2) to (step:3 pos:-3 tgt:2) through (move:0) values[-4.000000]
  2) transition from (step:3 pos:-3 tgt:2) to (step:2 pos:-2 tgt:2) through (move:1) values[-0.000030]
(step:2 pos:-2 tgt:2) values[-4.000000] forecast[-0.000020]
  0) transition from (step:2 pos:-2 tgt:2) to (step:3 pos:-3 tgt:2) through (move:-1) values[-4.000000]
  1) transition from (step:2 pos:-2 tgt:2) to (step:2 pos:-2 tgt:2) through (move:0) values[-0.000030]
  2) transition from (step:2 pos:-2 tgt:2) to (step:1 pos:-1 tgt:2) through (move:1) values[-0.000020]
(step:1 pos:-1 tgt:2) values[-3.000000] forecast[-0.000010]
  0) transition from (step:1 pos:-1 tgt:2) to (step:2 pos:-2 tgt:2) through (move:-1) values[-0.000030]
  1) transition from (step:1 pos:-1 tgt:2) to (step:1 pos:-1 tgt:2) through (move:0) values[-0.000020]
  2) transition from (step:1 pos:-1 tgt:2) to (step:0 pos:0 tgt:2) through (move:1) values[-0.000010]
(step:0 pos:0 tgt:2) values[-2.000000] forecast[0.000000]
  0) transition from (step:0 pos:0 tgt:2) to (step:1 pos:-1 tgt:2) through (move:-1) values[-0.000020]
  1) transition from (step:0 pos:0 tgt:2) to (step:0 pos:0 tgt:2) through (move:0) values[-0.000010]
  2) transition from (step:0 pos:0 tgt:2) to (step:1 pos:1 tgt:2) through (move:1) values[0.000000]
(step:1 pos:1 tgt:2) values[-1.000000] forecast[0.000000]
  0) transition from (step:1 pos:1 tgt:2) to (step:0 pos:0 tgt:2) through (move:-1) values[-0.000010]
  1) transition from (step:1 pos:1 tgt:2) to (step:1 pos:1 tgt:2) through (move:0) values[0.000000]
  2) transition from (step:1 pos:1 tgt:2) to (step:2 pos:2 tgt:2) through (move:1) values[0.000000]

```

```

(step:2 pos:2 tgt:2) values[0.000000] forecast[0.000000]
Best action: 1
UnitTestMiniFrameExpandSetCurWorld OK
mf(step:0 pos:0 tgt:2) real(step:1 pos:1 tgt:2)
mf(step:1 pos:1 tgt:2) real(step:2 pos:2 tgt:-1)
mf(step:2 pos:2 tgt:-1) real(step:3 pos:1 tgt:-1)
mf(step:3 pos:1 tgt:-1) real(step:4 pos:0 tgt:-1)
mf(step:4 pos:0 tgt:-1) real(step:5 pos:-1 tgt:-1)
UnitTestMiniFrameFullExample OK
UnitTestMiniFrame OK
UnitTestAll OK

```

6 Examples

6.1 Basic example

6.1.1 miniframe-model.h

```

// ===== MINIFRAME_MODEL.H =====

// As an example the code below implements a world where one actor
// moves along a discrete axis by step of one unit to reach a fixed
// target position
// Status of the world is defined by the current actor position and
// the target position
// Available actions are -1, 0, +1 (next position = current position
// + action) if the actor hasn't reached the target, else no actions
// The position of the actor is bounded to -5, 5
// The value of the world is given by -abs(position-target)

// ===== Include =====

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>
#include "/home/bayashi/GitHub/PBErr/pberr.h"

// ===== Define =====

// True if all actors act simultaneously, else false. As no effect if
// MF_NBMAXACTOR equals 1
#define MF_SIMULTANEOUS_PLAY false
// Max number of actors in the world
// must be at least one
#define MF_NBMAXACTOR 1
// Max number of transitions possible from any given status
// must be at least one
#define MF_NBMAXTRANSITION 3

// ===== Data structure =====

// Structure describing the transition from one instance of
// MFModelState to another
typedef struct MFModelTransition {
    int _move;

```

```

} MFModelTransition;

// Structure describing the status of the world at one instant
typedef struct MFModelStatus {
    int _step;
    int _pos;
    int _tgt;
} MFModelStatus;

// ===== Functions declaration =====

// Get the number of active actors
int MFModelStatusGetNbActor(const MFModelStatus* const that);

// Copy the properties of the MFModelStatus 'that' into the
// MFModelStatus 'tho'
// Dynamically allocated properties must be cloned
void MFModelStatusCopy(const MFModelStatus* const that,
    MFModelStatus* const tho);

// Free memory used by the properties of the MFModelStatus 'that'
// The memory used by the MFModelStatus itself is managed by MiniFrame
void MFModelStatusFreeStatic(MFModelStatus* that);

// Free memory used by the properties of the MFModelTransition 'that'
// The memory used by the MFModelTransition itself is managed by
// MiniFrame
void MFModelTransitionFreeStatic(MFModelTransition* that);

// Return true if 'that' and 'tho' are to be considered as the same
// by MiniFrame when trying to reuse previously computed status,
// else false
bool MFModelStatusIsSame(const MFModelStatus* const that,
    const MFModelStatus* const tho);

// Return the index of the actor who has preemption in the MFModelStatus
// 'that'
// If no actor has preemption (all the actor act simultaneously)
// return -1
int MFModelStatusGetSente(const MFModelStatus* const that);

// Return true if the actor 'iActor' is active given the MFModelStatus
// 'that'
bool MFModelStatusIsActorActive(const MFModelStatus* const that,
    const int iActor);

// Get the possible transitions from the MFModelStatus 'that' and
// memorize them in the array of MFModelTransition 'transitions', and
// memorize the number of transitions in 'nbTrans'
// 'transitions' as MF_NBMAXTRANSITION size, got MFModelTransition are
// expected in transitions[0~(nbTrans-1)]
void MFModelStatusGetTrans(const MFModelStatus* const that,
    MFModelTransition* const transitions, int* const nbTrans);

// Get the values of the MFModelStatus 'that' from the point of view
// of each actor and memorize them in the array of float 'values'
// 'values' as MF_NBMAXACTOR size, all values are set to 0.0 before
// calling this function
void MFModelStatusGetValues(const MFModelStatus* const that,
    float* const values);

// Return the MFModelStatus resulting from applying the

```

```

// MFModelTransition 'trans' to the MFModelStatus 'that'
MFModelStatus MFModelStatusStep(const MFModelStatus* const that,
    const MFModelTransition* const trans);

// Print the MFModelStatus 'that' on the stream 'stream'
void MFModelStatusPrint(const MFModelStatus* const that,
    FILE* const stream);

// Print the MFModelTransition 'that' on the stream 'stream'
void MFModelTransitionPrint(const MFModelTransition* const that,
    FILE* const stream);

// Return true if the MFStatus 'that' is disposable (its memory can be
// freed) given the current status 'curStatus' and the number of
// world instances in memory, else false
// As many as possible should be kept in memory, especially if worlds
// are reusable, but its up to the user to decide which and when should
// be discarded to fit the physical memory available
// Having too many world instances in memory also slow down the
// exploration of worlds during expansion
bool MFModelStatusIsDisposable(const MFModelStatus* const that,
    const MFModelStatus* const curStatus, const int nbStatus);

// Return true if the MFModelStatus 'that' is the end of the
// game/simulation, else false
bool MFModelStatusIsEnd(const MFModelStatus* const that);

// Init the board
void MFModelStatusInit(MFModelStatus* const that);

#ifdef BUILDMODE != 0
inline
#endif
void toto();

// ===== Inliner =====

#ifdef BUILDMODE != 0
#include "miniframe-inline-model.c"
#endif

```

6.1.2 miniframe-model.c

```

// ===== MINIFRAME_MODEL.C =====

// As an example the code below implements a world where one actor
// moves along a discrete axis by step of one unit to reach a fixed
// target position
// Status of the world is defined by the current actor position and
// the target position
// Available actions are -1, 0, +1 (next position = current position
// + action) if the actor hasn't reached the target, else no actions
// The position of the actor is bounded to -5, 5
// The value of the world is given by -abs(position-target)

// ===== Include =====

#include "miniframe-model.h"
#ifdef BUILDMODE == 0
#include "miniframe-inline-model.c"
#endif

```



```

// ===== Functions implementation =====

// Get the number of active actors
int MFModelStatusGetNbActor(const MFModelStatus* const that) {
    (void)that;
    return MF_NBMAXACTOR;
}

// Copy the properties of the MFModelStatus 'that' into the
// MFModelStatus 'tho'
// Dynamically allocated properties must be cloned
void MFModelStatusCopy(const MFModelStatus* const that,
    MFModelStatus* const tho) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (tho == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'tho' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    (void)memcpy(tho, that, sizeof(MFModelStatus));
}

// Free memory used by the properties of the MFModelStatus 'that'
// The memory used by the MFModelStatus itself is managed by MiniFrame
void MFModelStatusFreeStatic(MFModelStatus* that) {
    (void)that;
}

// Free memory used by the properties of the MFModelTransition 'that'
// The memory used by the MFModelTransition itself is managed by
// MiniFrame
void MFModelTransitionFreeStatic(MFModelTransition* that) {
    (void)that;
}

// Return true if 'that' and 'tho' are to be considered as the same
// by MiniFrame when trying to reuse previously computed status,
// else false
bool MFModelStatusIsSame(const MFModelStatus* const that,
    const MFModelStatus* const tho) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (tho == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'tho' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    if (that->_pos == tho->_pos &&

```

```

        that->_tgt == tho->_tgt)
        return true;
    else
        return false;
}

// Return the index of the actor who has preemption in the MFModelStatus
// 'that'
// If no actor has preemption (all the actor act simultaneously)
// return -1
int MFModelStatusGetSente(const MFModelStatus* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    (void)that;
    return 0;
}

// Return true if the actor 'iActor' is active given the MFModelStatus
// 'that'
bool MFModelStatusIsActorActive(const MFModelStatus* const that, const int iActor) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
        MiniFrameErr->_type = PBErrTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<=%d)", \
            iActor, MF_NBMAXACTOR);
        PBErrCatch(MiniFrameErr);
    }
#endif
    (void)that; (void)iActor;

    return true;
}

// Get the possible transitions from the MFModelStatus 'that' and
// memorize them in the array of MFModelTransition 'transitions', and
// memorize the number of transitions in 'nbTrans'
// 'transitions' as MF_NBMAXTRANSITION size, got MFModelTransition are
// expected in transitions[0~(nbTrans-1)]
void MFModelStatusGetTrans(const MFModelStatus* const that,
    MFModelTransition* const transitions, int* const nbTrans) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (transitions == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'transitions' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (nbTrans == NULL) {

```

```

        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'nbTrans' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    if (that->_pos == that->_tgt) {
        *nbTrans = 0;
    } else {
        *nbTrans = 3;
        transitions[0]._move = -1;
        transitions[1]._move = 0;
        transitions[2]._move = 1;
    }
}

// Get the values of the MFModelStatus 'that' from the point of view
// of each actor and memorize them in the array of float 'values'
// 'values' as MF_NBMAXACTOR size, all values are set to 0.0 before
// calling this function
void MFModelStatusGetValues(const MFModelStatus* const that,
    float* const values) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
        if (values == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'values' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
    values[0] = -1.0 * fabs(that->_tgt - that->_pos);
}

// Return the MFModelStatus resulting from applying the
// MFModelTransition 'trans' to the MFModelStatus 'that'
MFModelStatus MFModelStatusStep(const MFModelStatus* const that,
    const MFModelTransition* const trans) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
        if (trans == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'trans' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
    // Declare a variable to memorize the resulting status
    MFModelStatus status;
    // Apply the transition
    status._step = that->_step + 1;
    status._tgt = that->_tgt;
    status._pos = that->_pos + trans->_move;
    int limit = 3;
    if (status._pos < -limit) status._pos = -limit;
    if (status._pos > limit) status._pos = limit;
    // Return the status

```

```

    return status;
}

// Print the MFModelState 'that' on the stream 'stream'
void MFModelStatePrint(const MFModelState* const that,
    FILE* const stream) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PErrCatch(MiniFrameErr);
        }
        if (stream == NULL) {
            MiniFrameErr->_type = PErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'stream' is null");
            PErrCatch(MiniFrameErr);
        }
    #endif
    fprintf(stream, "step:%d pos:%d tgt:%d", that->_step,
        that->_pos, that->_tgt);
}

// Print the MFModelTransition 'that' on the stream 'stream'
void MFModelTransitionPrint(const MFModelTransition* const that,
    FILE* const stream) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PErrCatch(MiniFrameErr);
        }
        if (stream == NULL) {
            MiniFrameErr->_type = PErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'stream' is null");
            PErrCatch(MiniFrameErr);
        }
    #endif
    fprintf(stream, "move:%d", that->_move);
}

// Return true if the MFStatus 'that' is disposable (its memory can be
// freed) given the current status 'curStatus' and the number of
// world instances in memory, else false
// As many as possible should be kept in memory, especially if worlds
// are reusable, but its up to the user to decide which and when should
// be discarded to fit the physical memory available
// Having too many world instances in memory also slow down the
// exploration of worlds during expansion
bool MFModelStateIsDisposable(const MFModelState* const that,
    const MFModelState* const curStatus, const int nbStatus) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PErrCatch(MiniFrameErr);
        }
        if (curStatus == NULL) {
            MiniFrameErr->_type = PErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'curStatus' is null");
            PErrCatch(MiniFrameErr);
        }
    #endif
}

```

```

    if (nbStatus > 0) {
        if (abs(that->_pos - curStatus->_pos) > 2)
            return true;
        else
            return false;
    } else {
        return false;
    }
}

// Return true if the MFModelStatus 'that' is the end of the
// game/simulation, else false
bool MFModelStatusIsEnd(const MFModelStatus* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PErrCatch(MiniFrameErr);
    }
#endif
    if (that->_step >= 6 || that->_pos == that->_tgt) {
        return true;
    } else {
        return false;
    }
}

```

6.1.3 miniframe-inline-model.c

```

// ===== MINIFRAME-INLINE-MODEL.C =====

// ===== Functions implementation =====

#ifdef BUILDMODE != 0
inline
#endif
void toto() {

}

```

6.2 Oware

6.2.1 miniframe-model.h

```

// ===== MINIFRAME_MODEL.H =====

// ===== Include =====

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <stdbool.h>
#include <time.h>
#include "/home/bayashi/GitHub/PBErr/pberr.h"
#include "/home/bayashi/GitHub/NeuraNet/neuranet.h"

// ===== Define =====

```

```

// Current implementation doesn't allow more than 2 players
// due to undefined end condition
#define NBPLAYER 2
#define NBHOLEPLAYER 6
#define NBHOLE (NBHOLEPLAYER * NBPLAYER)
#define NBINITSTONEPERHOLE 4
#define NBSTONE (NBHOLE * NBINITSTONEPERHOLE)
#define NBMAXTURN 200

#define MF_MODEL_NN_NBINPUT NBHOLE
#define MF_MODEL_NN_NBOUTPUT 10
#define MF_MODEL_NN_NBHIDDEN 1
#define MF_MODEL_NN_NBBASES 100
#define MF_MODEL_NN_NBLINKS 100

// True if all actors act simultaneously, else false. As no effect if
// MF_NBMAXACTOR equals 1
#define MF_SIMULTANEOUS_PLAY false
// Max number of actors in the world
// must be at least one
#define MF_NBMAXACTOR NBPLAYER
// Max number of transitions possible from any given status
// must be at least one
#define MF_NBMAXTRANSITION NBHOLEPLAYER

// ===== Data structure =====

// Structure describing the transition from one instance of
// MFModelState to another
typedef struct MFModelTransition {
    // Index of the hole from where stones are moved by the current player
    int _iHole;
} MFModelTransition;

// Structure describing the status of the world at one instant
typedef struct MFModelState {
    int _nbTurn;
    int _nbStone[NBHOLE];
    int _score[NBPLAYER];
    // Flag for special end condition
    char _end;
    // Index of the player who has the sente
    int _curPlayer;
    // NeuraNet for each player
    NeuraNet* _nn[NBPLAYER];
} MFModelState;

// ===== Functions declaration =====

// Get the number of active actors
int MFModelStateGetNbActor(const MFModelState* const that);

// Copy the properties of the MFModelState 'that' into the
// MFModelState 'tho'
// Dynamically allocated properties must be cloned
void MFModelStateCopy(const MFModelState* const that,
    MFModelState* const tho);

// Free memory used by the properties of the MFModelState 'that'
// The memory used by the MFModelState itself is managed by MiniFrame
void MFModelStateFreeStatic(MFModelState* that);

```

```

// Free memory used by the properties of the MFModelTransition 'that'
// The memory used by the MFModelTransition itself is managed by
// MiniFrame
void MFModelTransitionFreeStatic(MFModelTransition* that);

// Return true if 'that' and 'tho' are to be considered as the same
// by MiniFrame when trying to reuse previously computed status,
// else false
bool MFModelStatusIsSame(const MFModelStatus* const that,
    const MFModelStatus* const tho);

// Return the index of the actor who has preemption in the MFModelStatus
// 'that'
// If no actor has preemption (all the actor act simultaneously)
// return -1
int MFModelStatusGetSente(const MFModelStatus* const that);

// Return true if the actor 'iActor' is active given the MFModelStatus
// 'that'
bool MFModelStatusIsActorActive(const MFModelStatus* const that,
    const int iActor);

// Get the possible transitions from the MFModelStatus 'that' and
// memorize them in the array of MFModelTransition 'transitions', and
// memorize the number of transitions in 'nbTrans'
// 'transitions' as MF_NBMAXTRANSITION size, got MFModelTransition are
// expected in transitions[0~(nbTrans-1)]
void MFModelStatusGetTrans(const MFModelStatus* const that,
    MFModelTransition* const transitions, int* const nbTrans);

// Get the values of the MFModelStatus 'that' from the point of view
// of each actor and memorize them in the array of float 'values'
// 'values' as MF_NBMAXACTOR size, all values are set to 0.0 before
// calling this function
void MFModelStatusGetValues(const MFModelStatus* const that,
    float* const values);

// Return the MFModelStatus resulting from applying the
// MFModelTransition 'trans' to the MFModelStatus 'that'
MFModelStatus MFModelStatusStep(const MFModelStatus* const that,
    const MFModelTransition* const trans);

// Print the MFModelStatus 'that' on the stream 'stream'
void MFModelStatusPrint(const MFModelStatus* const that,
    FILE* const stream);

// Print the MFModelTransition 'that' on the stream 'stream'
void MFModelTransitionPrint(const MFModelTransition* const that,
    FILE* const stream);

// Return true if the MFStatus 'that' is disposable (its memory can be
// freed) given the current status 'curStatus' and the number of
// world instances in memory, else false
// As many as possible should be kept in memory, especially if worlds
// are reusable, but its up to the user to decide which and when should
// be discarded to fit the physical memory available
// Having too many world instances in memory also slow down the
// exploration of worlds during expansion
bool MFModelStatusIsDisposable(const MFModelStatus* const that,
    const MFModelStatus* const curStatus, const int nbStatus);

```

```

// Return true if the MFModelStatus 'that' is the end of the
// game/simulation, else false
bool MFModelStatusIsEnd(const MFModelStatus* const that);

// Init the board
void MFModelStatusInit(MFModelStatus* const that);

#if BUILDMODE != 0
inline
#endif
void toto();

// ===== Inliner =====

#if BUILDMODE != 0
#include "miniframe-inline-model.c"
#endif

```

6.2.2 miniframe-model.c

```

// ===== MINIFRAME_MODEL.C =====

// ===== Include =====

#include "miniframe-model.h"
#if BUILDMODE == 0
#include "miniframe-inline-model.c"
#endif

// ===== Functions implementation =====

// Get the number of active actors
int MFModelStatusGetNbActor(const MFModelStatus* const that) {
    (void)that;
    return MF_NBMAXACTOR;
}

// Copy the properties of the MFModelStatus 'that' into the
// MFModelStatus 'tho'
// Dynamically allocated properties must be cloned
void MFModelStatusCopy(const MFModelStatus* const that,
    MFModelStatus* const tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
        if (tho == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'tho' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
    (void)memcpy(tho, that, sizeof(MFModelStatus));
}

// Free memory used by the properties of the MFModelStatus 'that'
// The memory used by the MFModelStatus itself is managed by MiniFrame
void MFModelStatusFreeStatic(MFModelStatus* that) {
    #if BUILDMODE == 0

```



```

    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    (void)that;
}

// Free memory used by the properties of the MFModelTransition 'that'
// The memory used by the MFModelTransition itself is managed by
// MiniFrame
void MFModelTransitionFreeStatic(MFModelTransition* that) {
    (void)that;
}

// Return true if 'that' and 'tho' are to be considered as the same
// by MiniFrame when trying to reuse previously computed status,
// else false
bool MFModelStatusIsSame(const MFModelStatus* const that,
    const MFModelStatus* const tho) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
        if (tho == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'tho' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
    bool ret = true;
    if (that->_curPlayer != tho->_curPlayer ||
        that->_end != tho->_end)
        ret = false;
    for (int iPlayer = NBPLAYER; iPlayer-- && ret;)
        if (that->_score[iPlayer] != tho->_score[iPlayer])
            ret = false;
    for (int iHole = NBHOLE; iHole-- && ret;)
        if (that->_nbStone[iHole] != tho->_nbStone[iHole])
            ret = false;
    return ret;
}

// Return the index of the actor who has preemption in the MFModelStatus
// 'that'
// If no actor has preemption (all the actor act simultaneously)
// return -1
int MFModelStatusGetSente(const MFModelStatus* const that) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
    return that->_curPlayer;
}

```

```

// Return true if the actor 'iActor' is active given the MFModelStatus
// 'that'
bool MFModelStatusIsActive(const MFModelStatus* const that, const int iActor) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (iActor < 0 || iActor >= MF_NBMAXACTOR) {
        MiniFrameErr->_type = PBErrTypeInvalidArg;
        sprintf(MiniFrameErr->_msg, "'iActor' is invalid (0<=%d<=%d)", \
            iActor, MF_NBMAXACTOR);
        PBErrCatch(MiniFrameErr);
    }
#endif
    (void)that; (void)iActor;
    // Incorrect if NBPLAYER > 2
    return true;
}

// Get the possible transitions from the MFModelStatus 'that' and
// memorize them in the array of MFModelTransition 'transitions', and
// memorize the number of transitions in 'nbTrans'
// 'transitions' as MF_NBMAXTRANSITION size, got MFModelTransition are
// expected in transitions[0~(nbTrans-1)]
void MFModelStatusGetTrans(const MFModelStatus* const that,
    MFModelTransition* const transitions, int* const nbTrans) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (transitions == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'transitions' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (nbTrans == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'nbTrans' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    *nbTrans = 0;
    for (int iHole = that->_curPlayer * NBHOLEPLAYER;
        iHole < (that->_curPlayer + 1) * NBHOLEPLAYER;
        ++iHole) {
        if (that->_nbStone[iHole] > 0) {
            transitions[*nbTrans]._iHole = iHole;
            ++(*nbTrans);
        }
    }
}

// Get the values of the MFModelStatus 'that' from the point of view
// of each actor and memorize them in the array of float 'values'
// 'values' as MF_NBMAXACTOR size, all values are set to 0.0 before
// calling this function
void MFModelStatusGetValues(const MFModelStatus* const that,
    float* const values) {

```

```

#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (values == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'values' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
VecFloat* input = VecFloatCreate(MF_MODEL_NN_NBINPUT);
VecFloat* output = VecFloatCreate(MF_MODEL_NN_NBOUTPUT);
for (int iPlayer = NBPLAYER; iPlayer--;) {
    if (that->_nn[iPlayer] == NULL) {
        values[iPlayer] = that->_score[iPlayer];
    } else {
        for (int iHole = NBHOLE; iHole--;) {
            int jHole = iHole + iPlayer * NBHOLEPLAYER;
            if (jHole >= NBHOLE)
                jHole -= NBHOLE;
            VecSet(input, iHole, that->_nbStone[jHole]);
        }
        NNEval(that->_nn[iPlayer], input, output);
        float valMax = VecGetMaxVal(output);
        values[iPlayer] = MAX(valMax, that->_score[iPlayer]);
    }
    if (values[iPlayer] * 2 > NBSTONE)
        values[iPlayer] = NBSTONE;
}
VecFree(&input);
VecFree(&output);
}

// Return the MFModelStatus resulting from applying the
// MFModelTransition 'trans' to the MFModelStatus 'that'
MFModelStatus MFModelStatusStep(const MFModelStatus* const that,
    const MFModelTransition* const trans) {
#if BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (trans == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'trans' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    // Declare a variable to memorize the resulting status
    MFModelStatus status;
    // Apply the transition

    MFModelStatusCopy(that, &status);
    int nbStone = status._nbStone[trans->_iHole];
    // Remove stones from starting hole
    status._nbStone[trans->_iHole] = 0;
    // Distribute stones
    int jHole = trans->_iHole;
    while (nbStone > 0) {

```

```

    ++jHole;
    if (jHole == NBHOLE) jHole = 0;
    // Jump over starting hole
    if (jHole == trans->iHole) ++jHole;
    if (jHole == NBHOLE) jHole = 0;
    ++(status._nbStone[jHole]);
    --nbStone;
}
// Check for captured stones
char flagCaptured = 0;
while ((jHole < status._curPlayer * NBHOLEPLAYER ||
    jHole >= (status._curPlayer + 1) * NBHOLEPLAYER) &&
    (status._nbStone[jHole] == 2 ||
    status._nbStone[jHole] == 3)) {
    status._score[status._curPlayer] += status._nbStone[jHole];
    status._nbStone[jHole] = 0;
    flagCaptured = 1;
    --jHole;
}
// Check for special end conditions
// First, check that the opponent is not starving
int nbStoneOpp = 0;
for (int iHole = 0; iHole < NBHOLE; ++iHole) {
    if (iHole < status._curPlayer * NBHOLEPLAYER ||
        iHole >= (status._curPlayer + 1) * NBHOLEPLAYER)
        nbStoneOpp += status._nbStone[iHole];
}
// If the opponent is starving
if (nbStoneOpp == 0) {
    if (flagCaptured == 1) {
        // If there has been captured stones, it means the current
        // player has starved the opponent. The current player loses.
        status._end = 1;
        status._score[status._curPlayer] = 0.0;
    } else {
        // If there was no captured stones, it means the opponent
        // starved itself. The current player catches all his own stones.
        status._end = 1;
        for (int iHole = 0; iHole < NBHOLE; ++iHole) {
            if (iHole >= status._curPlayer * NBHOLEPLAYER &&
                iHole < (status._curPlayer + 1) * NBHOLEPLAYER)
                status._score[status._curPlayer] +=
                    status._nbStone[iHole];
        }
    }
}
// Step the current player
++(status._curPlayer);
if (status._curPlayer == NBPLAYER)
    status._curPlayer = 0;
// Increment the nb of turn
++(status._nbTurn);

// Return the status
return status;
}

// Print the MFModelStatus 'that' on the stream 'stream'
void MFModelStatusPrint(const MFModelStatus* const that,
    FILE* const stream) {
#ifdef BUILDMODE == 0

```

```

    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
    if (stream == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'stream' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    fprintf(stream, "%d: ", that->_nbTurn);
    for (int iHole = 0; iHole < NBHOLE; ++iHole)
        fprintf(stream, "%d ", that->_nbStone[iHole]);
    fprintf(stream, " score: ");
    for (int iPlayer = 0; iPlayer < NBPLAYER; ++iPlayer) {
        if (iPlayer == MFModelStatusGetSente(that))
            fprintf(stream, "^");
        fprintf(stream, "%d", that->_score[iPlayer]);
        if (iPlayer < NBPLAYER - 1)
            fprintf(stream, ":");
    }
}

// Print the MFModelTransition 'that' on the stream 'stream'
void MFModelTransitionPrint(const MFModelTransition* const that,
    FILE* const stream) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
        if (stream == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'stream' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
    fprintf(stream, "move:%d", that->_iHole);
}

// Return true if the MFStatus 'that' is disposable (its memory can be
// freed) given the current status 'curStatus' and the number of
// world instances in memory, else false
// As many as possible should be kept in memory, especially if worlds
// are reusable, but its up to the user to decide which and when should
// be discarded to fit the physical memory available
// Having too many world instances in memory also slow down the
// exploration of worlds during expansion
bool MFModelStatusIsDisposable(const MFModelStatus* const that,
    const MFModelStatus* const curStatus, const int nbStatus) {
    #if BUILDMODE == 0
        if (that == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'that' is null");
            PBErrCatch(MiniFrameErr);
        }
        if (curStatus == NULL) {
            MiniFrameErr->_type = PBErrTypeNullPointer;
            sprintf(MiniFrameErr->_msg, "'curStatus' is null");
            PBErrCatch(MiniFrameErr);
        }
    #endif
}

```

```

    }
#endif
    (void)nbStatus;
    int nbRemainStoneCurStatus = 0;
    for (int iHole = NBHOLE; iHole--;)
        nbRemainStoneCurStatus += curStatus->_nbStone[iHole];
    int nbRemainStone = 0;
    for (int iHole = NBHOLE; iHole--;)
        nbRemainStone += that->_nbStone[iHole];
    if (nbRemainStone > nbRemainStoneCurStatus)
        return true;
    else
        return false;
}

// Return true if the MFModelStatus 'that' is the end of the
// game/simulation, else false
bool MFModelStatusIsEnd(const MFModelStatus* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif

    /*if (that->_score[0] > 0 || that->_score[1] > 0)
        return true;
    else
        return false;*/

    if (that->_end == 1 ||
        that->_nbTurn == NBMAXTURN)
        return true;
    bool ret = false;
    for (int iPlayer = NBPLAYER; iPlayer--;) {
        // Incorrect if NBPLAYER > 2
        if (that->_score[iPlayer] * 2 > NBSTONE)
            ret = true;
    }
    // For the case NBPLAYER > 2
    /*if (ret == false) {
        int nbRemainStone = 0;
        for (int iHole = NBHOLE; iHole-- && ret == false;)
            nbRemainStone += that->_nbStone[iHole];
        if (nbRemainStone == 0)
            ret = true;
    }*/
    return ret;
}

// Init the board
void MFModelStatusInit(MFModelStatus* const that) {
#ifdef BUILDMODE == 0
    if (that == NULL) {
        MiniFrameErr->_type = PBErrTypeNullPointer;
        sprintf(MiniFrameErr->_msg, "'that' is null");
        PBErrCatch(MiniFrameErr);
    }
#endif
    that->_end = 0;
    for (int iPlayer = NBPLAYER; iPlayer--;) {

```

```

        that->_score[iPlayer] = 0;
        that->_nn[iPlayer] = NULL;
    }
    for (int iHole = NBHOLE; iHole--;)
        that->_nbStone[iHole] = NBINITSTONEPERHOLE;
    that->_curPlayer = 0;
    that->_nbTurn = 0;
}

```

6.2.3 miniframe-inline-model.c

```

// ===== MINIFRAME-INLINE-MODEL.C =====

// ===== Functions implementation =====

#if BUILDMODE != 0
inline
#endif
void toto() {

}

```

6.2.4 main.c

```

#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <string.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>
#include "pberr.h"
#include "pbmath.h"
#include "genalg.h"
#include "elorank.h"
#include "neuranet.h"
#include "miniframe.h"

#define RANDOMSEED 0

void RunDemo(float expansionTime, bool useNN) {
    // Initial world
    MFModelStatus curWorld;
    MFModelStatusInit(&curWorld);
    // Display the current world
    MFModelStatusPrint(&curWorld, stdout);
    printf("\n");
    // Create the MiniFrame
    MiniFrame* mf = MiniFrameCreate(&curWorld);
    // If we use a NeuraNet as evaluation for player #0
    if (useNN) {
        // Try to load the NeuraNet from ./bestnn.txt
        FILE* stream = fopen("./bestnn.txt", "r");
        if (stream != NULL) {
            if (!NNLoad(curWorld._nn, stream)) {
                printf("Couldn't reload the NeuraNet from ./bestnn.txt\n");
                printf("Will use the default evaluation function\n");
            }
            fclose(stream);
        }
    }
}

```

```

    } else {
        printf("Couldn't reload the NeuraNet from ./bestnn.txt\n");
        printf("Will use the default evaluation function\n");
    }
}

// Set the expansion time
MFSetMaxTimeExpansion(mf, expansionTime);
// Set reusable worlds
MFSetWorldReusable(mf, true);
// Flag to end the game
bool flagEnd = false;
// Loop until end of game
while (!MFModelStatusIsEnd(&curWorld) && !flagEnd) {
    printf("-----\n");
    // Set the start clock
    MFSetStartExpandClock(mf, clock());
    // Correct the current world in the MiniFrame
    MFSetCurWorld(mf, &curWorld);
    // Expand
    MFExpand(mf);
    //MFWorldTransPrintln(MFCurWorld(mf), stdout);
    /*printf("--- start of best story ---\n");
    MFWorldPrintBestStoryln(MFCurWorld(mf),
        curWorld._curPlayer, stdout);
    printf("--- end of best story ---\n");*/
    // Display info about expansion
    printf("exp: %d ", MFGetNbWorldExpanded(mf));
    printf("unexp: %d ", MFGetNbWorldUnexpanded(mf));
    printf("comp: %d ", MFGetNbComputedWorld(mf));
    printf("removed: %d ", MFGetNbWorldRemoved(mf));
    printf("reused: %f ", MFGetPercWordReused(mf));
    printf("unused: %fms\n", MFGetTimeUnusedExpansion(mf));
    if (MFGetTimeUnusedExpansion(mf) < 0.0) {
        flagEnd = true;
        curWorld._score[curWorld._curPlayer] = -1;
    } else {
        // Get best transition
        const MFModelTransition* bestTrans =
            MFBestTransition(mf, MFModelStatusGetSente(&curWorld));
        if (bestTrans != NULL) {
            // Display the transition's information
            printf("sente: %d ", curWorld._curPlayer);
            MFModelTransitionPrint(bestTrans, stdout);
            printf(" forecast: %f",
                MFTransitionGetValue((MFTransition*)bestTrans,
                    curWorld._curPlayer));
            printf("\n");
            // Step with best transition
            curWorld = MFModelStatusStep(&curWorld, bestTrans);
        } else {
            flagEnd = true;
        }
        // Apply external forces to the world
        // curWorld. = ... ;
    }
    // Display the current world
    MFModelStatusPrint(&curWorld, stdout);
    printf("\n");
    fflush(stdout);
}

// Free memory
for (int iPlayer = NBPLAYER; iPlayer--;) {

```



```

        if (curWorld._nn[iPlayer] != NULL)
            NeuraNetFree(curWorld._nn + iPlayer);
    }
    MiniFrameFree(&mf);
}

void TrainOneGame(float expansionTime, GenAlgAdn** adns, GSet* result) {
    // Initial world
    MFModelStatus curWorld;
    MFModelStatusInit(&curWorld);
    // Create the MiniFrame
    MiniFrame* mf = MiniFrameCreate(&curWorld);
    // Set the NeuraNet for each actor
    for (int iActor = 0; iActor < NBPLAYER; ++iActor) {
        if (adns[iActor] != (void*)1) {
            NeuraNet* neuraNet = NeuraNetCreate(MF_MODEL_NN_NBINPUT,
                MF_MODEL_NN_NBOUTPUT, MF_MODEL_NN_NBHIDDEN,
                MF_MODEL_NN_NBBASES, MF_MODEL_NN_NBLINKS);
            NNSetBases(neuraNet, GAAdnAdnF(adns[iActor]));
            NNSetLinks(neuraNet, GAAdnAdnI(adns[iActor]));
            curWorld._nn[iActor] = neuraNet;
        } else {
            curWorld._nn[iActor] = NULL;
        }
    }
    // Set the expansion time
    MFSetMaxTimeExpansion(mf, expansionTime);
    // Set reusable worlds
    MFSetWorldReusable(mf, true);
    // Flag to end the game
    bool flagEnd = false;
    // Loop until end of game
    while (!MFModelStatusIsEnd(&curWorld) && !flagEnd) {
        // Set the start clock
        MFSetStartExpandClock(mf, clock());
        // Correct the current world in the MiniFrame
        MFSetCurWorld(mf, &curWorld);
        // Expand
        MFExpand(mf);
        if (MFGetTimeUnusedExpansion(mf) < 0.0) {
            flagEnd = true;
            curWorld._score[curWorld._curPlayer] = -1;
        } else {
            // Get best transition
            const MFModelTransition* bestTrans =
                MFBestTransition(mf, MFModelStatusGetSente(&curWorld));
            if (bestTrans != NULL) {
                // Step with best transition
                curWorld = MFModelStatusStep(&curWorld, bestTrans);
            } else {
                flagEnd = true;
            }
        }
    }
    // Update result
    GSetFlush(result);
    for (int iActor = 0; iActor < NBPLAYER; ++iActor)
        GSetAddSort(result, adns[iActor], curWorld._score[iActor]);
    // Free memory
    for (int iPlayer = NBPLAYER; iPlayer--;) {
        if (curWorld._nn[iPlayer] != NULL)
            NeuraNetFree(curWorld._nn + iPlayer);
    }
}

```

```

    }
    MiniFrameFree(&mf);
}

void Train(int nbEpoch, int sizePool, int nbElite, int nbGameEpoch,
float expansionTime) {
    // Display parameters
    printf("Will train with following parameters:\n");
    printf("nbEpoch: %d\n", nbEpoch);
    printf("sizePool: %d\n", sizePool);
    printf("nbElite: %d\n", nbElite);
    printf("nbGameEpoch: %d\n", nbGameEpoch);
    printf("expansionTime: %fms\n", expansionTime);
    // Create a NeuraNet
    NeuraNet* neuraNet = NeuraNetCreate(MF_MODEL_NN_NBINPUT,
MF_MODEL_NN_NBOUTPUT, MF_MODEL_NN_NBHIDDEN,
MF_MODEL_NN_NBBASES, MF_MODEL_NN_NBLINKS);
    // Create the GenAlg
    GenAlg* genAlg = GenAlgCreate(sizePool, nbElite,
NNGetGAAdnFloatLength(neuraNet), NNGetGAAdnIntLength(neuraNet));
    NNSetGABoundsBases(neuraNet, genAlg);
    NNSetGABoundsLinks(neuraNet, genAlg);
    GASETTypeNeuraNet(genAlg, MF_MODEL_NN_NBINPUT,
MF_MODEL_NN_NBHIDDEN, MF_MODEL_NN_NBOUTPUT);
    GAInit(genAlg);
    // Reload the GenAlg if possible
    FILE* stream = fopen("./bestga.txt", "r");
    if (stream != NULL) {
        printf("Reload the previous GenAlg from ./bestga.txt\n");
        if (GALoad(&genAlg, stream)) {
            printf("Couldn't reload the GenAlg\n");
            exit(1);
        }
    }
    // Declare a stream to save results
    FILE* streamRes = fopen("./res.txt", "w");
    if (streamRes == NULL) {
        printf("Couldn't open ./res.txt\n");
        exit(1);
    }
    // Declare a GSet to memorize the result
    GSet result = GSetCreateStatic();
    // Declare a variable to memorize the current epoch
    int iEpoch = 0;
    // Loop on epochs
    while (iEpoch < nbEpoch) {
        // Create the ELORank
        ELORank* eloRank = ELORankCreate();
        for (int iAdn = 0; iAdn < sizePool; ++iAdn)
            ELORankAdd(eloRank, GSetGet(GAAdns(genAlg), iAdn));
        ELORankAdd(eloRank, (GenAlgAdn*)GABestAdn(genAlg));
        ELORankAdd(eloRank, (void*)1);
        // Declare a variable to memorize the current game
        int iGame = 0;
        // Loop on games
        while (iGame < nbGameEpoch) {
            fprintf(stderr, "Epoch %05d/%05d Game %03d/%03d \r",
iEpoch + 1, nbEpoch, iGame + 1, nbGameEpoch);
            fflush(stderr);
            // Select randomly two adns
            GenAlgAdn* adns[NBPLAYER] = {NULL};
            int iAdn = (int)round(rnd() * (float)(sizePool) - 1.0);

```

```

    if (rnd() < 0.5) {
        adns[0] = (void*)1;
        if (iAdn == -1)
            adns[1] = (GenAlgAdn*)GABestAdn(genAlg);
        else
            adns[1] = GSetGet(GAAdns(genAlg), iAdn);
    } else {
        adns[1] = (void*)1;
        if (iAdn == -1)
            adns[0] = (GenAlgAdn*)GABestAdn(genAlg);
        else
            adns[0] = GSetGet(GAAdns(genAlg), iAdn);
    }
    // Play the game
    TrainOneGame(expansionTime, adns, &result);
    // Update the ELORank with the result
    ELORankUpdate(eloRank, &result);
    // Increment the current game
    ++iGame;
}
fprintf(stderr, "\n");
fflush(stderr);
// Update the values of each adn in the GenAlg with their ELORank
for (int iAdn = 0; iAdn < sizePool; ++iAdn) {
    GenAlgAdn* adn = GSetGet(GAAdns(genAlg), iAdn);
    float elo = ELORankGetELO(eloRank, adn);
    GSetAdnValue(genAlg, adn, elo);
}
// Update the value of the best adn too
GenAlgAdn* bestAdn = (GenAlgAdn*)GABestAdn(genAlg);
bestAdn->_val = ELORankGetELO(eloRank, bestAdn);
// Step the GenAlg
GAStep(genAlg);
// Display the elo of the best and the reference
float eloRef = ELORankGetELO(eloRank, (void*)1);
float eloBest = GAAdnGetVal(bestAdn);
printf("best: %f(age %ld) ref: %f(rank %d)\n", eloBest,
        GAAdnGetAge(bestAdn), eloRef, ELORankGetRank(eloRank, (void*)1));
fflush(stdout);
// Save the result
fprintf(streamRes, "%ld %f %f %d\n", GAGetCurEpoch(genAlg), eloBest,
        eloRef, ELORankGetRank(eloRank, (void*)1));
fflush(streamRes);
// Save the best NeuraNet to ./bestnn.txt
NNSetBases(neuraNet, GAAdnAdnF(bestAdn));
NNSetLinks(neuraNet, GAAdnAdnI(bestAdn));
stream = fopen("./bestnn.txt", "w");
if (stream == NULL) {
    printf("Couldn't open ./bestnn.txt");
    exit(1);
}
if (!NNSave(neuraNet, stream, true)) {
    printf("Couldn't open ./bestnn.txt");
    exit(1);
}
fclose(stream);
// Save the GenAlg to ./bestga.txt
stream = fopen("./bestga.txt", "w");
if (stream == NULL) {
    printf("Couldn't open ./bestga.txt");
    exit(1);
}
}

```

```

    if (!GASave(genAlg, stream, true)) {
        printf("Couldn't save ./bestga.txt");
        exit(1);
    }
    fclose(stream);
    // Increment the current epoch
    ++iEpoch;
    // Free memory
    ELORankFree(&eloRank);
}
// Free memory
fclose(streamRes);
GSetFlush(&result);
GenAlgFree(&genAlg);
NeuraNetFree(&neuraNet);
}

int main(int argc, char** argv) {
    // Init the random generator
    srand(time(NULL));
    // Declare a variable to memorize the mode
    // 0: demo (default)
    // 1: train mode
    // 2: demo with trained NeuraNet as player #0
    int mode = 0;
    // Declare a variable to memorize the expansion time (in millisec)
    float expansionTime = 100.0;
    // Declare a variable to memorize the number of epoch for training
    int nbEpoch = 50;
    // Declare variables to memorize the size of pool, number of elites,
    // number of game per epoch for training
    int nbElite = 5;
    int sizePool = 20;
    int nbGameEpoch = 200;
    // Process argument
    for (int iArg = 0; iArg < argc; ++iArg) {
        if (strcmp(argv[iArg], "-help") == 0) {
            printf("main [-demo] [-demoNN] [-train] [-nbEpoch <nbEpoch>] ");
            printf("[-nbElite <nbElite>] [-sizePool <sizePool>] ");
            printf("[-nbGameEpoch <nbGameEpoch>] [-expTime <expansionTime>]\n");
            exit(0);
        } else if (strcmp(argv[iArg], "-demo") == 0) {
            mode = 0;
        } else if (strcmp(argv[iArg], "-train") == 0) {
            mode = 1;
        } else if (strcmp(argv[iArg], "-demoNN") == 0) {
            mode = 2;
        } else if (strcmp(argv[iArg], "-nbEpoch") == 0 && iArg < argc - 1) {
            ++iArg;
            nbEpoch = atoi(argv[iArg]);
        } else if (strcmp(argv[iArg], "-nbElite") == 0 && iArg < argc - 1) {
            ++iArg;
            nbElite = atoi(argv[iArg]);
        } else if (strcmp(argv[iArg], "-sizePool") == 0 && iArg < argc - 1) {
            ++iArg;
            sizePool = atoi(argv[iArg]);
        } else if (strcmp(argv[iArg], "-nbGameEpoch") == 0 && iArg < argc - 1) {
            ++iArg;
            nbGameEpoch = atoi(argv[iArg]);
        } else if (strcmp(argv[iArg], "-expTime") == 0 && iArg < argc - 1) {
            ++iArg;
            expansionTime = atof(argv[iArg]);
        }
    }
}

```

```

    }
}

if (mode == 0) {
    RunDemo(expansionTime, false);
} else if (mode == 1) {
    Train(nbEpoch, sizePool, nbElite, nbGameEpoch, expansionTime);
} else if (mode == 2) {
    RunDemo(expansionTime, true);
}

// Return success code
return 0;
}

```

6.2.5 Makefile

```

# Build mode
# 0: development (max safety, no optimisation)
# 1: release (min safety, optimisation)
# 2: fast and furious (no safety, optimisation)
BUILD_MODE?=0

all: main

# Makefile definitions
MAKEFILE_INC=../../PBMake/Makefile.inc
include $(MAKEFILE_INC)

# Path to the model implementation
MF_MODEL_PATH=$(ROOT_DIR)/MiniFrame/Examples/Oware

# Include directories
MODEL_INC_DIR=-I$(ROOT_DIR)/PBErr -I$(ROOT_DIR)/GenAlg -I$(ROOT_DIR)/NeuraNet -I$(ROOT_DIR)/PBMath -I$(ROOT_DIR)/PBJS

# Rules to make the executable
main: \
createLinkToModelHeader \
main.o \
miniframe-model.o \
neuranet.o \
genalg.o \
elorank.o \
$(miniframe_EXE_DEP) \
$(miniframe_DEP)
$(COMPILER) 'echo "$(miniframe_EXE_DEP) main.o" | tr ' ' '\n' | sort -u' miniframe-model.o neuranet.o genalg.o elorank.o

main.o: \
main.c \
$(miniframe_INC_H_EXE) \
$(miniframe_EXE_DEP)
$(COMPILER) $(BUILD_ARG) $(MODEL_INC_DIR) $(miniframe_BUILD_ARG) 'echo "$(miniframe_INC_DIR)" | tr ' ' '\n' | sort -u'

createLinkToModelHeader:
ln -s -f $(MF_MODEL_PATH)/miniframe-model.h $(miniframe_DIR)/miniframe-model.h; ln -s -f $(MF_MODEL_PATH)/miniframe-model.c $(miniframe_DIR)/miniframe-model.c

miniframe-model.o: miniframe-model.h miniframe-model.c Makefile
$(COMPILER) $(BUILD_ARG) $(MODEL_INC_DIR) -c miniframe-model.c

```

6.2.6 Example

exampleGame.txt:

```
#0: 4 4 4 4 4 4 4 4 4 4 4 4
score: 0:0 1:0
-----
exp: 580 comp: 2969 unused: 0.536000ms
move:2
#1: 4 4 0 5 5 5 5 4 4 4 4 4
score: 0:0 1:0
-----
exp: 215 comp: 4062 unused: 0.541000ms
move:11
#2: 5 5 1 6 5 5 5 4 4 4 4 0
score: 0:0 1:0
-----
exp: 162 comp: 4885 unused: 0.762000ms
move:3
#3: 5 5 1 0 6 6 6 5 5 5 4 0
score: 0:0 1:0
-----
exp: 131 comp: 5568 unused: 0.838000ms
move:9
#4: 6 6 0 0 6 6 6 5 5 0 5 1
score: 0:0 1:2
-----
exp: 116 comp: 6170 unused: 1.026000ms
move:5
#5: 6 6 0 0 6 0 7 6 6 1 6 0
score: 0:2 1:2
-----
exp: 534 comp: 2810 unused: 0.498000ms
move:9
#6: 6 6 0 0 6 0 7 6 6 0 7 0
score: 0:2 1:2
-----
exp: 216 comp: 3877 unused: 0.557000ms
move:1
#7: 6 0 1 1 7 1 8 7 6 0 7 0
score: 0:2 1:2
-----
exp: 161 comp: 4701 unused: 0.916000ms
move:10
#8: 7 1 2 2 8 0 8 7 6 0 0 1
score: 0:2 1:4
-----
exp: 141 comp: 5336 unused: 0.701000ms
move:1
#9: 7 0 3 2 8 0 8 7 6 0 0 1
score: 0:2 1:4
-----
exp: 448 comp: 3056 unused: 0.325000ms
move:8
#10: 8 1 4 2 8 0 8 7 0 1 1 2
score: 0:2 1:4
-----
exp: 234 comp: 4200 unused: 0.514000ms
move:1
#11: 8 0 5 2 8 0 8 7 0 1 1 2
score: 0:2 1:4
```

```

-----
exp: 180 comp: 5089 unused: 0.750000ms
move:10
#12: 8 0 5 2 8 0 8 7 0 1 0 3
score: 0:2 1:4
-----
exp: 144 comp: 5830 unused: 0.696000ms
move:0
#13: 0 1 6 3 9 1 9 8 1 1 0 3
score: 0:2 1:4
-----
exp: 134 comp: 6481 unused: 0.835000ms
move:6
#14: 1 2 7 4 9 1 0 9 2 2 1 4
score: 0:2 1:4
-----
exp: 121 comp: 7066 unused: 0.919000ms
move:2
#15: 1 2 0 5 10 2 1 10 0 0 1 4
score: 0:8 1:4
-----
exp: 123 comp: 7443 unused: 1.284000ms
move:7
#16: 2 3 1 6 11 0 1 0 1 1 2 5
score: 0:8 1:7
-----
exp: 619 comp: 2985 unused: 0.462000ms
move:3
#17: 2 3 1 0 12 1 2 1 0 0 2 5
score: 0:12 1:7
-----
exp: 254 comp: 4095 unused: 0.653000ms
move:10
#18: 0 3 1 0 12 1 2 1 0 0 0 6
score: 0:12 1:10
-----
exp: 631 comp: 2820 unused: 0.337000ms
move:5
#19: 0 3 1 0 12 0 0 1 0 0 0 6
score: 0:15 1:10
-----
exp: 254 comp: 3860 unused: 0.553000ms
move:7
#20: 0 3 1 0 12 0 0 0 1 0 0 6
score: 0:15 1:10
-----
exp: 549 comp: 2981 unused: 0.492000ms
move:4
#21: 1 4 2 1 0 2 1 1 2 1 1 7
score: 0:15 1:10
-----
exp: 288 comp: 4097 unused: 0.761000ms
move:8
#22: 1 4 2 1 0 2 1 1 0 2 2 7
score: 0:15 1:10
-----
exp: 203 comp: 4957 unused: 0.663000ms
move:5
#23: 1 4 2 1 0 0 0 0 0 2 2 7
score: 0:19 1:10
-----
exp: 177 comp: 5688 unused: 0.793000ms

```

```

move:10
#24: 0 4 2 1 0 0 0 0 2 0 8
score: 0:19 1:12
-----
exp: 535 comp: 3137 unused: 0.411000ms
move:1
#25: 0 0 3 2 1 1 0 0 0 2 0 8
score: 0:19 1:12
-----
exp: 286 comp: 4213 unused: 0.454000ms
move:9
#26: 0 0 3 2 1 1 0 0 0 0 1 9
score: 0:19 1:12
-----
exp: 231 comp: 5065 unused: 0.754000ms
move:5
#27: 0 0 3 2 1 0 1 0 0 0 1 9
score: 0:19 1:12
-----
exp: 201 comp: 5778 unused: 0.716000ms
move:11
#28: 1 1 4 3 2 1 2 1 1 0 1 0
score: 0:19 1:12
-----
exp: 150 comp: 6395 unused: 1.125000ms
move:2
#29: 1 1 0 4 3 2 0 1 1 0 1 0
score: 0:22 1:12
-----
exp: 148 comp: 6962 unused: 1.460000ms
move:7
#30: 1 1 0 4 3 2 0 0 2 0 1 0
score: 0:22 1:12
-----
exp: 782 comp: 2977 unused: 0.257000ms
move:5
#31: 1 1 0 4 3 0 1 1 2 0 1 0
score: 0:22 1:12
-----
exp: 376 comp: 4101 unused: 0.492000ms
move:7
#32: 1 1 0 4 3 0 1 0 3 0 1 0
score: 0:22 1:12
-----
exp: 254 comp: 4947 unused: 0.656000ms
move:1
#33: 1 0 1 4 3 0 1 0 3 0 1 0
score: 0:22 1:12
-----
exp: 227 comp: 5659 unused: 0.571000ms
move:6
#34: 1 0 1 4 3 0 0 1 3 0 1 0
score: 0:22 1:12
-----
exp: 172 comp: 6268 unused: 0.976000ms
move:4
#35: 1 0 1 4 0 1 1 0 3 0 1 0
score: 0:24 1:12
-----
exp: 169 comp: 6823 unused: 1.288000ms
move:6
#36: 1 0 1 4 0 1 0 1 3 0 1 0

```



```
score: 0:24 1:12
-----
exp: 314 comp: 4234 unused: 0.660000ms
move:3
#37: 1 0 1 0 1 2 1 0 3 0 1 0
score: 0:26 1:12
```

training.txt:

Will train with following parameters:

```
nbEpoch: 50
sizePool: 20
nbElite: 5
nbGameEpoch: 200
expansionTime: 2.000000ms
best: 31.328260(age 1) ref: 82.931335(rank 0)
best: 22.864634(age 1) ref: 79.334846(rank 0)
best: 24.208862(age 1) ref: -10.613078(rank 18)
best: 32.224094(age 1) ref: 12.468557(rank 5)
best: 17.549753(age 2) ref: -14.585226(rank 20)
best: 21.162813(age 1) ref: -27.068129(rank 21)
best: 43.162781(age 1) ref: 49.898052(rank 0)
best: 23.102114(age 1) ref: -32.691559(rank 20)
best: 39.671085(age 1) ref: -63.786736(rank 21)
best: 25.883085(age 1) ref: -45.418316(rank 21)
best: 43.656036(age 4) ref: -70.671555(rank 21)
best: 28.096897(age 2) ref: -88.050621(rank 21)
best: 35.324615(age 1) ref: -76.103546(rank 21)
best: 29.285027(age 7) ref: -80.363495(rank 21)
best: 32.384163(age 1) ref: -39.222408(rank 21)
best: 33.087044(age 1) ref: -159.921310(rank 21)
best: 38.447403(age 1) ref: -77.489342(rank 21)
best: 29.453932(age 1) ref: -97.934502(rank 21)
best: 28.433582(age 4) ref: -121.706337(rank 21)
best: 27.938528(age 1) ref: -44.681839(rank 21)
best: 26.743734(age 1) ref: -117.727440(rank 21)
best: 32.887238(age 1) ref: -118.119720(rank 21)
best: 31.815880(age 1) ref: -126.141380(rank 21)
best: 35.271561(age 1) ref: -160.575256(rank 21)
best: 39.056988(age 1) ref: -114.015266(rank 21)
best: 35.051537(age 1) ref: -172.867569(rank 21)
best: 33.369774(age 2) ref: -139.915665(rank 21)
best: 23.945967(age 1) ref: -160.053726(rank 21)
best: 28.880730(age 1) ref: -118.576210(rank 21)
best: 36.596848(age 1) ref: -126.771790(rank 21)
best: 40.642696(age 1) ref: -205.380783(rank 21)
best: 38.710934(age 1) ref: -119.274033(rank 21)
best: 32.663273(age 2) ref: -71.547211(rank 21)
best: 27.899593(age 1) ref: -128.230774(rank 21)
best: 35.756004(age 1) ref: -144.496521(rank 21)
best: 31.073360(age 2) ref: -46.526688(rank 20)
best: 28.740427(age 1) ref: -117.581848(rank 21)
best: 42.407894(age 2) ref: -91.168510(rank 21)
best: 22.859707(age 5) ref: -138.085449(rank 21)
best: 39.806992(age 1) ref: -75.063660(rank 21)
best: 30.126772(age 1) ref: -161.244186(rank 21)
best: 26.507477(age 1) ref: -138.023300(rank 21)
best: 23.815845(age 1) ref: -190.276886(rank 21)
best: 40.388985(age 1) ref: -124.716026(rank 21)
best: 36.726894(age 1) ref: -113.932503(rank 21)
best: 28.655504(age 1) ref: -161.943542(rank 21)
```

best: 32.068508(age 1) ref: -133.815598(rank 21)
best: 29.135687(age 1) ref: -147.955139(rank 21)
best: 40.737518(age 2) ref: -121.294418(rank 21)
best: 37.172047(age 1) ref: -197.387863(rank 21)

ELO rank of the best NeuraNet (blue) as evaluation function and ELO rank of the standard (red) evaluation function, and rank of the standard evaluation function against a pool of 21 NeuraNet:

