Contents

Chapter 1

Package ThreadingUtils

1.1 Class TimeLimitedCodeBlock

Allows one to call a thread with a given timeout. Implementation taken from: https://stackoverflow.com/questions/5715235/java-set-timeout-on-a-certain-block-of-code

1.1.1 Declaration

```
public class TimeLimitedCodeBlock
 extends java.lang.Object
```

1.1.2 Constructor summary

TimeLimitedCodeBlock()

1.1.3 Method summary

runWithTimeout(Callable, long, TimeUnit) Implements the actual runnable code block with timeout.

runWithTimeout(Runnable, long, TimeUnit) Calls a runnable object, with a given timeout.

1.1.4 Constructors

- **TimeLimitedCodeBlock**

```
public TimeLimitedCodeBlock()
```

1.1.5 Methods

- **runWithTimeout**

```
public static java.lang.Object runWithTimeout(java.util.
    concurrent.Callable callable ,long timeout ,java.util.
    concurrent.TimeUnit timeUnit) throws java.lang.Exception
```

  - **Description**

    Implements the actual runnable code block with timeout.

- **runWithTimeout**

```
public static void runWithTimeout(java.lang.Runnable runnable ,
    long timeout ,java.util.concurrent.TimeUnit timeUnit) throws
    java.lang.Exception
```

  - **Description**

    Calls a runnable object, with a given timeout.

  - **Parameters**
    * `runnable` – is the runnable object to run.
    * `timeout` – is the timeout.
    * `timeUnit` – is the unit of the timeout.

  - **Throws**
    * `java.lang.Exception` –

Chapter 2

Package engines

Package Contents *Page*

Interfaces

Classes

2.1 Interface GameEngine

Interface to register requirements for game engines.

2.1.1 Declaration

public interface GameEngine

2.1.2 All known subinterfaces

Team11Engine (in 2.3, page 6), RandomEngine (in 2.2, page 5)

2.1.3 All classes known to implement interface

Team11Engine (in 2.3, page 6), RandomEngine (in 2.2, page 5)

2.1.4 Method summary

computeMove(LaskerMorrisGameState) Computes next state from a given non-final game state.

2.1.5 Methods

- **computeMove**

```
model.LaskerMorrisGameState computeMove(model.
    LaskerMorrisGameState state)
```

  - **Description**
    Computes next state from a given non-final game state. Method computeMove should be "stoppable", i.e., when issued a TimeOutException or ExecutionException, the routine must return immediately. See random engine implementation for an example of how to deal with this issue.
  - **Parameters**
    - `state` – is the game state to move from
  - **Returns** – the computed next state.

2.2 Class RandomEngine

Engine implementing random moves.

2.2.1 Declaration

```
public class RandomEngine
 extends java.lang.Object implements GameEngine
```

2.2.2 Constructor summary

RandomEngine() Default constructor

2.2.3 Method summary

computeMove(LaskerMorrisGameState) Computes move to take, according to random decision value.

2.2.4 Constructors

- **RandomEngine**

```
public RandomEngine()
```

  - **Description**
    Default constructor

2.2.5 Methods

- **computeMove**

  ```
  public model.LaskerMorrisGameState computeMove(model.
      LaskerMorrisGameState state)
  ```

  - **Description**
    Computes move to take, according to random decision value.
  - **Parameters**
    - `state` – is the state from which to move.
  - **Returns** – the resulting state to move to, according to a random decision.

2.3 Class Team11Engine

Engine implementing random moves.

2.3.1 Declaration

```
public class Team11Engine
 extends java.lang.Object implements GameEngine
```

2.3.2 Constructor summary

Team11Engine()

2.3.3 Method summary

computeMove(LaskerMorrisGameState) For a given turn, compute the best possible play.

miniMax(LaskerMorrisGameState, int, int, int) For a given game state, a maximum depth of search, and alpha and beta values for alpha-beta pruning, calculate an optimal move.

successorStates(LaskerMorrisGameState) For a given game state, returns all the possible successor game states.

2.3.4 Constructors

- **Team11Engine**

  ```
  public Team11Engine()
  ```

2.3.5 Methods

- **computeMove**

  ```
  public model.LaskerMorrisGameState computeMove(model.
      LaskerMorrisGameState e)
  ```

  - **Description**
    For a given turn, compute the best possible play.
  - **Parameters**
    * `e` – a game state.
  - **Returns** – the game state to be played in the given turn.

- **miniMax**

  ```
  public static int miniMax(model.LaskerMorrisGameState e,int
      maxDepth,int alpha,int beta)
  ```

  - **Description**
    For a given game state, a maximum depth of search, and alpha and beta values for alpha-beta pruning, calculate an optimal move.
  - **Parameters**
    * `e` – a game state.
    * `maxDepth` – the depth to which the game tree will be searched.
    * `alpha` – for alpha-beta pruning optimization.
    * `beta` – for alpha-beta pruning optimization.
  - **Returns** – value for the best possible play up to the given depth.

- **successorStates**

  ```
  public static java.util.PriorityQueue successorStates(model.
      LaskerMorrisGameState e)
  ```

  - **Description**
    For a given game state, returns all the possible successor game states.
  - **Parameters**
    * `e` – a game state.
  - **Returns** – a priority queue (whose priority is given by the heuristic evaluator) containing all successor moves given e.

2.4 Class ThreadedRandomEngine

Class that runs a thread with a random engine.

2.4.1 Declaration

```
public class ThreadedRandomEngine
 extends java.lang.Object implements java.lang.Runnable
```

2.4.2 Field summary

result Resulting move is left here

2.4.3 Constructor summary

ThreadedRandomEngine(LaskerMorrisGameState) Default constructor

2.4.4 Method summary

run() Runs a stoppable random engine.

2.4.5 Fields

- `public static model.LaskerMorrisGameState` **result**
  - Resulting move is left here

2.4.6 Constructors

- **ThreadedRandomEngine**

  ```
  public ThreadedRandomEngine(model.LaskerMorrisGameState state)
  ```

  - **Description**
    Default constructor
  - **Parameters**
    - `state` – is the source state to play from

2.4.7 Methods

- **run**

  ```
  public void run()
  ```

  - **Description**
    Runs a stoppable random engine.

2.5 Class ThreadedTeam11Engine

Class that runs a thread with a random engine.

2.5.1 Declaration

```
public class ThreadedTeam11Engine
 extends java.lang.Object implements java.lang.Runnable
```

2.5.2 Field summary

depth
result Resulting move is left here

2.5.3 Constructor summary

ThreadedTeam11Engine(LaskerMorrisGameState) Default constructor

2.5.4 Method summary

run() Runs a stoppable random engine.

2.5.5 Fields

- `public static model.LaskerMorrisGameState` **result**
  - Resulting move is left here
- `public static int` **depth**

2.5.6 Constructors

- **ThreadedTeam11Engine**

  ```
  public ThreadedTeam11Engine(model.LaskerMorrisGameState state)
  ```

  - **Description**
    Default constructor
  - **Parameters**
    - `state` – is the source state to play from

2.5.7 Methods

- **run**

  ```
  public void run()
  ```

  - **Description**
    Runs a stoppable random engine.

Chapter 3

Package model

Package Contents *Page*

Classes

3.1 Class LaskerMorrisBoard

Represents static board information, such as adjacencies and the coordinates of all mills in the board. Since static board info may be accessible by different classes, the class is implemented using the Singleton Pattern.

3.1.1 Declaration

```
public class LaskerMorrisBoard
 extends java.lang.Object
```

3.1.2 Field summary

MAXSTONES Maximum number of stones per player
POSITIONS Number of positions of the board.

3.1.3 Method summary

getAdjacencies(int) Returns the list of coordinates adjacent to a given coordinate
getInstance() Static method for obtaining singleton.
getMills(int) Returns the list of mills for a given position

3.1.4 Fields

- `public static final int` **POSITIONS**

  - Number of positions of the board. See the representation below.

- `public static final int` **MAXSTONES**
  - Maximum number of stones per player

3.1.5 Methods

- **getAdjacencies**

```
public java.util.List getAdjacencies(int i)
```

  - **Description**
    Returns the list of coordinates adjacent to a given coordinate
  - **Parameters**
    - `i` – is the coordinate to query for adjacencies.
  - **Returns** – the list of coordinates adjacent to i.

- **getInstance**

```
public static LaskerMorrisBoard getInstance()
```

  - **Description**
    Static method for obtaining singleton.
  - **Returns** – a reference to a Lasker Morris Board. The object is created on-demand and contains solely static information regarding the board.

- **getMills**

```
public java.util.List getMills(int position)
```

  - **Description**
    Returns the list of mills for a given position
  - **Parameters**
    - `position` –
  - **Returns** –

3.2 Class LaskerMorrisGameState

Represents a board state of the Lasker Morris Game. Contains most of the functionality of the game, including adjacency relations between positions, mills combinations, etc.

3.2.1 Declaration

```
public class LaskerMorrisGameState
 extends java.lang.Object
```

3.2.2 Field summary

BLACK Constant to represent black stone.
EMPTY Constant to represent empty position in board.
WHITE Constant to represent white stone.

3.2.3 Constructor summary

LaskerMorrisGameState() Default constructor.
LaskerMorrisGameState(boolean, int[], int, int) Constructor that receives turn and board contents.

3.2.4 Method summary

blackWins() Indicates whether black won in current state.
blockedAdversaryStones() Computes the difference between the current state's number of blocked stones and the adversary's.
boardToString() Produces string representation of the board state.
canMoveStone() Checks whether a stone in the board can be moved.
clone() Creates a clone of a game state.
closedMills() Determines if a mill has been formed in the current game state.
doubleMills() Computes the difference between the current state's number of double mills and the adversary's NOTE: A double mill is two mills joint together by a stone present in both mills.
estimatedValue() Computes estimated value for current state.
getNumberOfFreeAdjacent(int) Returns the number of free adjacent positions to a given position in the board.
getValue(int) Returns value in specified position.
isFinal() Checks whether the current state is final, i.e., cannot continue playing
isInMill(int) Checks whether given position belongs to a mill.
isMax() Checks whether current state is max (max is if white plays).
isOccupied(int) Indicates whether a specified position is occupied or not.
isValid() Checks whether the current state is valid
isValidPosition(int) Checks whether a given position is a valid position
isWhitesTurn() Indicates whether white's play.
maxValue() Returns max possible value for estimations.
minValue() Returns min possible value for estimations.
moveStone(int, Integer) Moves a stone in the board.
numberOfBlackStonesOnBoard() Counts number of black stones on the board
numberOfMills() Computes the difference between the number of mills for a given game state.
numberOfStones() Computes the difference between the current state's number of stones and the adversary's.

numberOfWhiteStonesOnBoard() Counts number of white stones on the board
putStone(int) Sets a stone in the specified position, according to the player's turn.
remainingBlackStones() Remaining black stones to put on the board.
remainingWhiteStones() Remaining white stones to put in the board.
removeStone(int) Removes stone from given position
setClosedMill(boolean) True iff a mill was formed in the current game state.
setWhitesTurn(boolean) Sets the turn of the game.
threeStonesConfiguration() Computes the difference between the current state's number of three-stone configurations and the adversary's.
toString() Produces string representation of the state.
twoStonesConfiguration() Computes the difference between the current state's number of two-stone configurations and the adversary's.
whiteWins() Indicates whether white won in current state.
winningConfiguration() Determines if the current game state guarantees a win.

3.2.5 Fields

- `public static final int` **EMPTY**
  - Constant to represent empty position in board.
- `public static final int` **WHITE**
  - Constant to represent white stone.
- `public static final int` **BLACK**
  - Constant to represent black stone.

3.2.6 Constructors

- **LaskerMorrisGameState**

  ```
  public LaskerMorrisGameState()
  ```

  - **Description**
    Default constructor. Board empty, white plays.
- **LaskerMorrisGameState**

  ```
  public LaskerMorrisGameState(boolean whitePlays, int[] board, int
      whiteStonesToPlay, int blackStonesToPlay)
  ```

  - **Description**
    Constructor that receives turn and board contents.
  - **Parameters**
    - `whitePlays` – indicates whether white plays.
    - `board` – is the contents to set in the board.
    - `whiteStonesToPlay` – is the number of white stones still remaining to be put.
    - `blackStonesToPlay` – is the number of black stones still remaining to be put.

3.2.7 Methods

- **blackWins**

  ```
  public boolean blackWins()
  ```

  - **Description**
    Indicates whether black won in current state.
  - **Returns** – true iff state is final and black wins.

- **blockedAdversaryStones**

  ```
  public int blockedAdversaryStones()
  ```

  - **Description**
    Computes the difference between the current state's number of blocked stones and the adversary's.
  - **Returns** – the difference between the current state's number of blocked stones and the adversary's.

- **boardToString**

  ```
  public java.lang.String boardToString()
  ```

  - **Description**
    Produces string representation of the board state. It ignores the turn.
  - **Returns** – a string representing the state of the board.

- **canMoveStone**

  ```
  public boolean canMoveStone()
  ```

  - **Description**
    Checks whether a stone in the board can be moved.
  - **Returns** – true iff current player can move a stone in the board.

- **clone**

  ```
  public LaskerMorrisGameState clone()
  ```

  - **Description**
    Creates a clone of a game state.

  - **Returns** – a clone object of the game state.

- **closedMills**

```
public int closedMills()
```

  - **Description**
    Determines if a mill has been formed in the current game state.
  - **Returns** – 1 if a mill has been closed in the current game state or zero otherwhise.

- **doubleMills**

```
public int doubleMills()
```

  - **Description**
    Computes the difference between the current state's number of double mills and the adversary's NOTE: A double mill is two mills joint together by a stone present in both mills.
  - **Returns** – the difference between the current state's number of double mills and the adversary's.

- **estimatedValue**

```
public int estimatedValue()
```

  - **Description**
    Computes estimated value for current state. Idea taken from the following paper: http://www.dasconference.ro/papers/2008/B7.pdf
  - **Returns** – an estimated value of the current state.

- **getNumberOfFreeAdjacent**

```
public int getNumberOfFreeAdjacent(int position)
```

  - **Description**
    Returns the number of free adjacent positions to a given position in the board.
  - **Parameters**
    - `position` – is the position to query about.
  - **Returns** – the number of adjacent positions to position, which are free.

- **getValue**

```
public int getValue(int position)
```

- **Description**
  Returns value in specified position.
- **Parameters**
  - `position` – is the position to query about.
- **Returns** – zero if position free, 1 if occupied by white stone, 2 if occupied by black stone.

- **isFinal**

```
public boolean isFinal()
```

- **Description**
  Checks whether the current state is final, i.e., cannot continue playing
- **Returns** – true iff the state is final, arrived to a winner.

- **isInMill**

```
public boolean isInMill(int position)
```

- **Description**
  Checks whether given position belongs to a mill.
- **Parameters**
  - `position` – is the position to query.
- **Returns** – true iff position is in a mill.

- **isMax**

```
public boolean isMax()
```

- **Description**
  Checks whether current state is max (max is if white plays). Useful for minimax function.
- **Returns** – true iff white plays.

- **isOccupied**

```
public boolean isOccupied(int position)
```

- **Description**
  Indicates whether a specified position is occupied or not.

  - **Parameters**
    - `position` – is the position to query.
  - **Returns** – true iff the position is occupied.

- **isValid**

  ```
  public boolean isValid()
  ```

  - **Description**
    Checks whether the current state is valid
  - **Returns** – true iff the current state is valid.

- **isValidPosition**

  ```
  public static boolean isValidPosition(int position)
  ```

  - **Description**
    Checks whether a given position is a valid position
  - **Parameters**
    - `position` – is the position to query about
  - **Returns** – true iff the position is a valid position

- **isWhitesTurn**

  ```
  public boolean isWhitesTurn()
  ```

  - **Description**
    Indicates whether white's play.
  - **Returns** – true iff it's white's turn.

- **maxValue**

  ```
  public static int maxValue()
  ```

  - **Description**
    Returns max possible value for estimations. Value returned by estimatedValue() must always be smaller than this value. Useful for minimax and minimax alpha beta.
  - **Returns** – max possible value for estimations

- **minValue**

```
public static int minValue()
```

- **Description**
  Returns min possible value for estimations. Value returned by estimatedValue() must always be greater than this value. Useful for minimax and minimax alpha beta.
- **Returns** – min possible value for estimations

- **moveStone**

```
public void moveStone(int position,java.lang.Integer adjacent)
```

  - **Description**
    Moves a stone in the board.
  - **Parameters**
    - `position` – is the source position of the stone.
    - `adjacent` – is the target position of the stone.

- **numberOfBlackStonesOnBoard**

```
public int numberOfBlackStonesOnBoard()
```

  - **Description**
    Counts number of black stones on the board
  - **Returns** – number of black stones on the board.

- **numberOfMills**

```
public int numberOfMills()
```

  - **Description**
    Computes the difference between the number of mills for a given game state.
  - **Returns** – the difference between the number of mills for a given game state.

- **numberOfStones**

```
public int numberOfStones()
```

  - **Description**
    Computes the difference between the current state's number of stones and the adversary's.
  - **Returns** – the difference between the current state's number of stones and the adversary's.

- **numberOfWhiteStonesOnBoard**

  ```
  public int numberOfWhiteStonesOnBoard()
  ```

  - **Description**
    Counts number of white stones on the board
  - **Returns** – number of white stones on the board.

- **putStone**

  ```
  public void putStone(int position)
  ```

  - **Description**
    Sets a stone in the specified position, according to the player's turn.
  - **Parameters**
    - `position` – is the position where to set the stone.

- **remainingBlackStones**

  ```
  public int remainingBlackStones()
  ```

  - **Description**
    Remaining black stones to put on the board.
  - **Returns** – the number of black stones that still can be put on the board.

- **remainingWhiteStones**

  ```
  public int remainingWhiteStones()
  ```

  - **Description**
    Remaining white stones to put in the board.
  - **Returns** – the number of white stones that still can be put in the board.

- **removeStone**

  ```
  public void removeStone(int pos)
  ```

  - **Description**
    Removes stone from given position
  - **Parameters**
    - `pos` – is the position from which to remove a stone

- **setClosedMill**

  ```
  public void setClosedMill(boolean closeMill)
  ```

  - **Description**
    True iff a mill was formed in the current game state.

- **setWhitesTurn**

  ```
  public void setWhitesTurn(boolean whitesTurn)
  ```

  - **Description**
    Sets the turn of the game.
  - **Parameters**
    * `indicates` – whether is white's turn or not.

- **threeStonesConfiguration**

  ```
  public int threeStonesConfiguration()
  ```

  - **Description**
    Computes the difference between the current state's number of three-stone configurations and the adversary's. NOTE: A three stone configuration is two two-stone configurations joint together by a stone present in both two-stone configurations.
  - **Returns** – the difference between the current state's number of three-stone configurations and the adversary's.

- **toString**

  ```
  public java.lang.String toString()
  ```

  - **Description**
    Produces string representation of the state. Prints board, turn, remaining stones per player.
  - **Returns** – a string representing the state of the game.

- **twoStonesConfiguration**

  ```
  public int twoStonesConfiguration()
  ```

  - **Description**

    Computes the difference between the current state's number of two-stone configurations and the adversary's. NOTE: A two stone configuration is two aligned stones (both either white or black) with an empty third position available to form a mill.
  - **Returns** – the difference between the current state's number of two-stone configurations and the adversary's.

- **whiteWins**

  ```
  public boolean whiteWins()
  ```

  - **Description**

    Indicates whether white won in current state.
  - **Returns** – true iff state is final and white wins

- **winningConfiguration**

  ```
  public int winningConfiguration()
  ```

  - **Description**

    Determines if the current game state guarantees a win.
  - **Returns** – 0 if it doesn't guarantee a win, 1 or -1 if it guarantees a win for either the white or black stones.

Chapter 4

Package runners

4.1 Class RandomGameWithTimeout

A sample application where white plays black using random engines.

4.1.1 Declaration

```
public class RandomGameWithTimeout
 extends java.lang.Object
```

4.1.2 Field summary

MAXMOVES Max number of total moves before considering a match a draw.
TIMEOUT Timeout in seconds for each player's timeout.

4.1.3 Constructor summary

RandomGameWithTimeout()

4.1.4 Method summary

main(String[]) Creates a game where whites play blacks using random engines for both players.

4.1.5 Fields

- `public static final int` **MAXMOVES**

  - Max number of total moves before considering a match a draw.

- `public static final int` **TIMEOUT**
  - Timeout in seconds for each player's timeout.

4.1.6 Constructors

- **RandomGameWithTimeout**

```
public RandomGameWithTimeout()
```

4.1.7 Methods

- **main**

```
public static void main(java.lang.String[] args)
```

  - **Description**
    Creates a game where whites play blacks using random engines for both players. Game is considered a draw if MAXMOVES total moves are reached without a winner.

4.2 Class Team11GameWithTimeout

A sample application where white plays black using random engines.

4.2.1 Declaration

```
public class Team11GameWithTimeout
 extends java.lang.Object
```

4.2.2 Field summary

MAXMOVES Max number of total moves before considering a match a draw.
TIMEOUT Timeout in seconds for each player's timeout.

4.2.3 Constructor summary

Team11GameWithTimeout()

4.2.4 Method summary

main(String[]) Creates a game where whites play blacks using random engines for both players.

4.2.5 Fields

- `public static final int` **MAXMOVES**
  - Max number of total moves before considering a match a draw.
- `public static final int` **TIMEOUT**
  - Timeout in seconds for each player's timeout.

4.2.6 Constructors

- **Team11GameWithTimeout**

```
public Team11GameWithTimeout()
```

4.2.7 Methods

- **main**

```
public static void main(java.lang.String[] args)
```

  - **Description**
    Creates a game where whites play blacks using random engines for both players. Game is considered a draw if MAXMOVES total moves are reached without a winner.