GamesCrafters

Snake

Alice Chang (Dev)

Judy Chen (Dev)

Peter Foo (Dev)

Documentation for Developers

Fall 2007: Moonway Lin (Documentation)

Date	Version	Notes
2007.11.27	1.0.0	First version created
2007.12.05	1.0.1	First revision

Table of Contents

Gamescrafters Documentation: Snake

0. Game Overview

Snake is a classic 1970s game in which a snake eats pieces and grows longer and longer until it eats itself or runs out of space. In this variant, two players take turn alternating between the snake's head and tail until the opponent is unable to move.

Snake is a **partizan** game: since each player has control over only one type of piece, both players do not have access to the exact same moves.

Snake is **non-loopy** and **ties are not possible**.

1. Design Overview

The script works by constructing a tree for all possible paths from the head to the tail, then does a depth-first search on it.

The tree-related functions `isAncestorOf()`, `maxDepthOf()`, and `constructTree()` are helper function designed to aid in this matter.

`SnakeHash()` and `SnakeUnhash()` then work with the remaining data to solve the game.

2. Data Structures

The board is arranged as follows:

```
1   2   3   4

5   6   7   8

9   10  11  12

13  14  15  16
```

The default starting position is

```
-   -   -   -

-   H---*   -
        |
-   -   T   -

-   -   -   -
```

The board's current position is stored as an integer:
SUM: index i from 1 to 16: IsOccupied(i) * 2^tilenum(i)

For example, this position

```
-   -   -   -

-   H---*   -
        |
-   -   T   -

-   -   -   -
```

has a position value of 1*2^13 + 1*2^7 + 1*2^5 + 1*2^2 + 1*2^0 = 8357..

There is a struct **`node`** used for constructing the snake's body:

```
struct node {
    int position;
    BlankBHT piece;
    int depth;
    struct node *left;
    struct node *down;
    struct node *right;
    struct node *up;
    struct node *parent;
};
```

There is a basic data structure called a **`BlankBHT`** (body, head, tail) which represents a piece. This data structure is used in the hashing and unhashing functions.

There is a global variable named `SLOT` to determine which player's turn it is.

3. C Text Game Checklist

All help strings have been written, but none of them change with variant.

Debugging options are turned off.

There are potential memory leaks: there is no call to `FreeMoveList()` after `GenerateMoves()`.

`PrintPosition()` does NOT call `GetPrediction()`. However, it tracks the current player's turn with the local variable `usersTurn`.

The code is reasonably well commented, with inline comments explaining what each major section of code does.

There are calls to common game function libraries in `generateMoves()`, `GetAndPrintPlayersMove()`, `DoMove()`, and `SnakeUnhash()`

This particular implementation does NOT support symmetries or GPS.

4. Complexity Analysis

There are 4,194,304 total positions for a 4x4 game board (the default game board size). This is equivalent to **2^16 possible positions**, since each one of the 4x4=16 tiles can be occupied by a part of the snake's body (body, head, or tail) or be left unoccupied (in which case it is stored as a 0). There are two possibilities for each of the 16 tiles, leaving a total of 2^16 possible positions.

As long as the game board is finite, depth first search can be performed on the game tree to find the optimal sequence of moves that leads to a win, i.e. the other player runs out of room to expand his snake.

5. Optimal Strategy

The optimal strategy can be uncovered via game tree search, and allows Player 1 to force a win via an optimal sequence of moves that ensures the other player (Player 2) to run out of possible tiles at the end of the game.