Carcassonne Solitaire

Algorithms and Computability
Laboratories

Documentation

Przemysław Rosiński
Piotr Janaszek

Faculty of Mathematics and Information Science
Warsaw University of Technology

26 October 2011

DOCUMENT METRIC

Project:	Carcassonne Solitaire			Company:	WUT
Name:	Documentation				
Topics:	Description of algorithms, file formats, inner data structure				
Author:	Przemysław Rosiński, Piotr Janaszek				
File:	Carcassonne_Rosinski_Janaszek_Documentation.pdf				
Version no:	1.0	Status:	Working	Opening date:	23.10.2011
Summary:	Client requirements for the application				
Authorized by:				Last modification date:	26.10.2011

PREFACE

The aim of this project is to develop a set of algorithms able to solve Carcassonne Solitaire problem of creating the biggest square board.
There is given a set of square tiles. Every of the edges of a tile can be of three types: field, city or road. The problem is to place the tiles in such a way that edges of two consecutive tiles are of the same type and they form the biggest possible square.
The score is the length of the square side. Hence, for example, for the given set of 72 tiles the highest possible score is 8 (that is creating a board containing a square consisting of 64 tiles).
The set of tiles is fixed. The list of them can be found in *Description of file formats* section.

DESCRIPTION OF ACCURATE ALGORITHM

We are said to provide an algorithm which for given input will produce the best solution, i.e. the biggest possible square board.

Assumptions:
We need three sets of tiles:

1. Set of available (not used) tiles - A.
2. Set of tiles matching given place - M.
3. Set of tiles already fixed for the board - F.

Algorithm:

1. Fill A with all tiles from input. F and M are empty.
2. Select one tile from A and place it on the board - move it to F.
3. While A is not empty and best possible score is not reached do:
    3.1. Using heuristic function create the list P of the available positions on the board sorted from “the best” ones down to “the worst”.
    3.2. For every position from P:
        3.2.1. Find in A tiles matching requirements of selected position and move them to M.
        3.2.2. For every tile from M place it on the board on given position - move it from M to F. Move remaining tiles in M back to A.

Assuming that:

- void PlaceOnBoard(Tile t, Position p) assigns tile t to position p on the board,
- List<Position> GetPositions(Set<Tiles> F) returns the list of the available positions on the board sorted from “the best” ones down to “the worst” using heuristic function. This function assigns to every position a value equal to minimal number of another positions needed to be filled in order to increase the score. The lower value of heuristic function for given position the better this position is,
- bool Matches(Tile t, Position p) checks whether tile t matches the requirements for placing it in position p, i.e. edges of t match edges of p,
- bool ScoreEqual(int MaximumScore) checks whether current score (length of biggest square edge) is equal to the highest possible score

the following pseudo-code can be considered:

Note: in comment lines (/*...*/) there are marked points of the code corresponding to the algorithm description.

```
void Main()
{
	/* point 1 */
	List<Tile> A = set of tiles;
	List<Tile> F = empty set
	int MaximumScore = sqrt(A.Length);
	/* point 2 */
	Tile t = A.GetRandomElement();
	Position p = (0,0);
	PlaceOnBoard(t, p);
	F' = F + {t}
	A' = A - {t}
	/* point 3 */
	while(!A.IsEmpty() && !ScoreEqual(MaximumScore))
	{
		Next(A', F');
	}
}

Next(List<Tile> A, List<Tile> F)
{
	/* point 3.1 */
	List<Position> P = GetPositions(F);
	List<Tile> M = empty set;
	/* point 3.2 */
	for(int i=0; i<P.Length; i++)
	{
		/* point 3.2.1 */
		 for(int j=0; j<A.Length; j++)
		 {
			if(Matches(A[j], P[i]))
			{
				M = M + {A[j]};
				A' = A - {A[j]};
			}
	}
		/* point 3.2.2 */
		for(int l=0; l<M.Length; l++)
		{
			PlaceOnBoard(M[l], P[i]);
			F' = F + M[l];
			M = M - M[l];
			A' = A' + M;
			Next(A', F');
		}
	}
}
```

COMPLEXITY

The analysis of algorithm performance was done in the terms of number of operations done. Below, every step of the algorithm and its complexity analysis can be found:

1. Every element must be put into a set - number of operations is *n*.
2. Only one item is selected - one operation done.
3. Since one element was taken in step 2., there are n-1 elements left in the set of available tiles – the loop is executed *n-1* times, let's iterate the loops with letter *i*.
    - 3.1. Heuristic function must check all 4 (or less) neighbour position of every tile placed on the board - in *i-th* loop there are at most *4*i* comparisons to be done.
    - 3.2. In the set of positions there can be at most *4*i* elements – loop is executed at most *4*i* times.
        - 3.2.1. For every position we need to check all elements left in the set of available tiles. In *i-th* loop execution there are *n-i* of them. Moreover, every tile must be checked 4 times (for initial configuration and three rotations). For every position there are *4*(n-i)* operations.
        - 3.2.2. In the set of matching tiles there are at most *n-i* of them – *n-i* operations

Summing all operations we get: $n+1+\sum_{i=1}^{n-1}[4i]+\sum_{i=1}^{n-1}[4i*4(n-1)]+\sum_{i=1}^{n-1}[4i*(n-i)]$.

The worst-case complexity is $O(n^3)$.

DESCRIPTION OF APPROXIMATED ALGORITHM (Przemysław Rosiński)

Taking into account that there are only three types of edges of tiles one can focus on creating big field of land, city or road. The choice is building big city. This algorithm is a modified version of the accurate algorithm.
Firstly, it creates a board by connecting only tiles with at least one edge of city type. Secondly, it uses remaining tiles to expand the board and create a square out of it.
The algorithm limits number of tiles and number of positions on the board to be considered when building the city.

Assumptions:
We need three sets of tiles:

1. Set of available (not used) tiles - A.
2. Set of tiles matching given place - M.
3. Set of tiles already fixed for the board - F.

Algorithm:

1. Fill A with all tiles from input. F and M are empty.
2. Select tile from A with four edges of city type and place it on the board - move it to F.
3. While A has tiles with edge of city type and there is a position with city edge:
    3.1. Using heuristic function create the list P of the available positions with edges of city type.
    3.2. For every position from P:
        3.2.1. Find in A tiles matching requirements of selected position and move them to M.
        3.2.2. For every tile from M place it on the board on given position - move it from M to F. Move remaining tiles in M back to A.
4. While A is not empty and best possible score is not reached do:
    4.1. Using heuristic function create the list P of the available positions on the board sorted from “the best” ones down to “the worst”.
    4.2. For every position from P:
        4.2.1. Find in A tiles matching requirements of selected position and move them to M.
        4.2.2. For every tile from M place it on the board on given position - move it from M to F. Move remaining tiles in M back to A.

COMPLEXITY

Worst-case complexity is identical with the complexity of the accurate algorithm, beacause set of operations done is exactly the same. The difference is that the main loop (point 3. in the accurate algorithm) is here divided into two part – working on the set of tiles with edge of city type and others. But for the first group we decrease number of positions to work on. It reduces the number of operations done on set of the positions. This part is crucial, so time reduction is expected. Still, the complexity is $O(n^3)$.

DESCRIPTION OF APPROXIMATED ALGORITHM (Piotr Janaszek)

In this algorithm the fact that there are only three types of edges will be also exploited. Program will create many 2x2 squares in which every edge will consist of two tile edges of the same type. In this way there will be created a set of at most 18 squares, which can be considered as big tiles. After creating such a set of big tiles, the computation is done using the accurate algorithm.
The algorithm limits number of executions of complex and time-consuming Next(List<Tile> A, List<Tile> F) function.

Assumptions:
We need one set of tiles B and three sets of big (2x2) tiles:

1. Set of available (not used) big tiles - A.
2. Set of big tiles matching given place - M.
3. Set of big tiles already fixed for the board - F.

Algorithm:

1. While B is not empty and a big tile can be created, select one tile from B and create 2x2 square using three tiles matching edges.
2. Fill A with all created big tiles. F and M are empty.
3. Select one big tile from A and place it on the board - move it to F.
4. While A is not empty and best big-tile possible score is not reached then:
    4.1. Using heuristic function create the list P of the available positions on the board sorted from “the best” ones down to “the worst”.
    4.2. For every position from P:
        4.2.1. Find in A big tiles matching requirements of selected position and move them to M.
        4.2.2. For every big tile from M place it on the board on given position - move it from M to F. Move remaining big tiles in M back to A.

COMPLEXITY

As in the case of previous approximated algorithm, worst-case complexity is identical with the complexity of the accurate algorithm, beacause of identical set of operations. But, the difference lies on other field, mainly, while creating bigger tiles (2x2) at most all the small tiles are used. Hence, the number of items to work on is reduced at least 4 times. Obviously, the time of execution of this part is reduced. But, it is still hard to say, how creating big tiles (which is of order $O(n^2)$) affects the time of execution in total. Nevertheless, the complexity is $O(n^3)$.

DESCRIPTION OF FILE FORMATS

TILES

All tiles that can be used in the game are listed below:

tile: left=city, right=field, top=field, bottom=city - 5 times
tile: left=city, right=road, top=road, bottom=road - 3 times
tile: left=city, right=city, top=field, bottom=field - 3 times
tile: left=field, right=field, top=field, bottom=field - 4 times
tile: left=field, right=road, top=field, bottom=field - 2 times
tile: left=road, right=road, top=road, bottom=road - 1 time
tile: left=field, right=road, top=road, bottom=field - 9 times
tile: left=road, right=road, top=field, bottom=field - 8 times
tile: left=field, right=road, top=road, bottom=road - 4 times
tile: left=city, right=city, top=city, bottom=city - 1 time
tile: left=city, right=field, top=city, bottom=city - 4 times
tile: left=city, right=road, top=city, bottom=city - 3 times
tile: left=city, right=field, top=city, bottom=field - 2 times
tile: left=city, right=field, top=field, bottom=field - 5 times
tile: left=city, right=field, top=road, bottom=road - 4 times
tile: left=city, right=road, top=road, bottom=field - 3 times
tile: left=city, right=road, top=field, bottom=road - 3 times
tile: left=field, right=field, top=city, bottom=city - 3 times
tile: left=city, right=road, top=road, bottom=city - 5 times

THE INPUT FILE

We will use XML format to provide input for the application. Each element will have four attributes describing sides of the tile. The attributes will be selected from the set consisting of field, city and road. The number of the same tiles as well as the combination of attributes is fixed. The template of the XML file is given below.

```
<?xml version="1.0"?>
<tiles>
<tile left="" right="" top="" bottom=""></tile>
<tile left="" right="" top="" bottom=""></tile>
</tiles>
```

THE OUTPUT FILE

The output file will have the same structure as the input file. However, there will be two additional attributes responsible for giving the information about the alignment of the tiles. Attribute positionX will provide information about horizontal alignment and positionY attribute about vertical alignment. From that there will be possibility to construct the whole board.

```
<?xml version="1.0"?>
<tiles>
<tile left="" right="" top="" bottom="" positionX="" positionY=""></tile>
<tile left="" right="" top="" bottom="" positionX="" positionY=""></tile>
<tile left="" right="" top="" bottom="" positionX="" positionY=""></tile>
<tile left="" right="" top="" bottom="" positionX="" positionY=""></tile>
</tiles>
```

DESCRIPTION OF INNER DATA STRUCTURE

The inner data structure will consist of the following elements:

- Tile class Tile will have six attributes: left, right, top, bottom, positionX and positionY. For all tiles left, right, top and bottom attributes will be set. positionX and positionY will only be set for already aligned tiles.
- List of type <Tile> consisting of all available tiles.
- List of type <Tile> consisting of matching tiles.
- List of type <Tile> consisting of already fixed tiles with positionX and positionY attributes set.
- Integer value for maximum score. This will store the maximal possible length of square edge.