The TUC-ANTS Game

Artificial Intelligence - COMP417

TECHNICAL UNIVERSITY OF CRETE

25 Μαΐου 2013

Tassos Souris (2007030009)

The TUC-ANTS Game

Artificial Intelligence - COMP417

INTRODUCTION

The purpose of this report is to describe the project for the tuc-ants game as I implemented it. The rules and the setup of the game are described in detail in the project description (which can be found at the course site), so I omit them.

OVERVIEW

For the project i used the implementation that is given as part of the project. I used the C++ language to fill in the client (with no compatibility problems since C++ understands C).

I added the files needed to implement the algorithms for the game and then changed the file client.cpp so that i use my algorithms when the server requests a move. So, i begin with the new files in the project.

MINIMAX IMPLEMENTATION

In this section I describe how I implemented the minimax algorithm. I implemented an iterative-deepening expectiminimax with alpha-beta pruning mechanism (except for chance nodes), with a timeout test to stop the iterative-deepening. When the time expires the algorithm returns the move that was chosen from the deepest search that was completed in time. The timeout parameter is given in milliseconds as argument to the algorithm.

I also added support for the following features: Action ordering that orders the moves according to their utility values (as returned by the evaluation function) in ascending order. This mechanism helps during alpha-beta pruning. A cutoff test can be specified by the user of the algorithm as well as a evaluation function.

The algorithm depends heavily in the use of templates which makes it generic enough to be used for many different scenarios. The user has only to specify as template parameters the main features of the algorithm that it needs from the user (the type of the state and moves, the type of the successors function, the type of the evaluation function, etc).

The algorithm is implemented in file minimax.hpp. After i have defined the structure of the game (structs that represent the state, the successor function, the cutoff test, etc), then i tie them together and use them in the client.cpp file when the server requests a move from us. Then i run the algorithm simply as:

```
// here is where we run expectiminimax on the current position
// and it is our move the algorithm returns which action to do
tucants_game_cutoff cutoff;
search::iterative_deepening_alpha_beta_expectiminimax<tucants> minimax(cutoff)

myMove = minimax.decision(gamePosition, timeout);
```

STRUCTURES FOR THE GAME

In the file tucants_game.hpp i implemented all the features i needed for the game. I used a header file since i implemented all of them as inline functions (or structs/classes). The important points to notice are (other more important get a section of their own later!):

1. I have implemented functions to find moves for a particular ant. The function which_moves() returns a list of *all* possible moves that a given ant can make. This function respects the captivity precedence rule.
2. The successor function calls the above function which_moves() for all ants of the player who has turn. Then i find if there are any moves with captivity precedence and i ignore the rest (with pretty usage of C++ algorithm templates and lambda expressions!).
3. The cutoff test simply tests if all the ants have been removed from the game. Remember that the timeout test is implemented by the minimax algorithm.

EVALUATION FUNCTION

The evaluation function i used is:

$$my(utility_{value} + score) - opponents(utility_{value} + score)$$

It remains to explain how is the utility value computed.

What i wanted to favor in different game configurations are situations where:

- i capture as many ants of the opponents as possible
- don't allow the opponent to capture any of my ants

I placed more emphasis on how to save my ants from the opponent. I realized that i have more opportunities to save my ants if they move at the sides of the board or if they form triangles (since the opponent cannot cross-over the ant).

The first portion of the utility value is determined by the placement of the ants on the board. If an ant is placed at the sides on the board it gets a better utility value. At the center it gets the worse since i consider it as a unsafe region. At the opponent side it gets a good utility value since it is close to the end. Also i give a fair good utility value on the starting positions since they are like guard positions. If a ant is there is doesn't let opponent ants promote to queens. The following table shows the utility values from the side of the black player.

0	3	0	3	0	3	0	3
3	0	3	0	3	0	3	0
0	3	0	3	0	3	0	3
5	0	1	0	1	0	1	0
0	1	0	1	0	1	0	5
5	0	1	0	1	0	1	0
0	1	0	1	0	1	0	5
5	0	1	0	1	0	1	0
0	1	0	1	0	1	0	5
2	0	2	0	2	0	2	0
0	2	0	2	0	2	0	2
2	0	2	0	2	0	2	0

Then i further advance the utility value in these situations:

- According to the number of moves an ant can make
- According to the number of capture moves that can be made
- According to the number of other ants an ant protects (the triangle shape)

I also give extra points to a board configuration if food is obtained.
The last thing to point out is that the value is decremented at the end according to the number of ants the opponent can capture in that board configuration.

However, i have to admit that the weight assigned to each of the above features that i used (which was an integer like 1 or 2), was rather randomly chosen (e.g. captivity moves are more important so i used a greater weight from other features but how much greater was chosen at chance).

CHANCE NODES

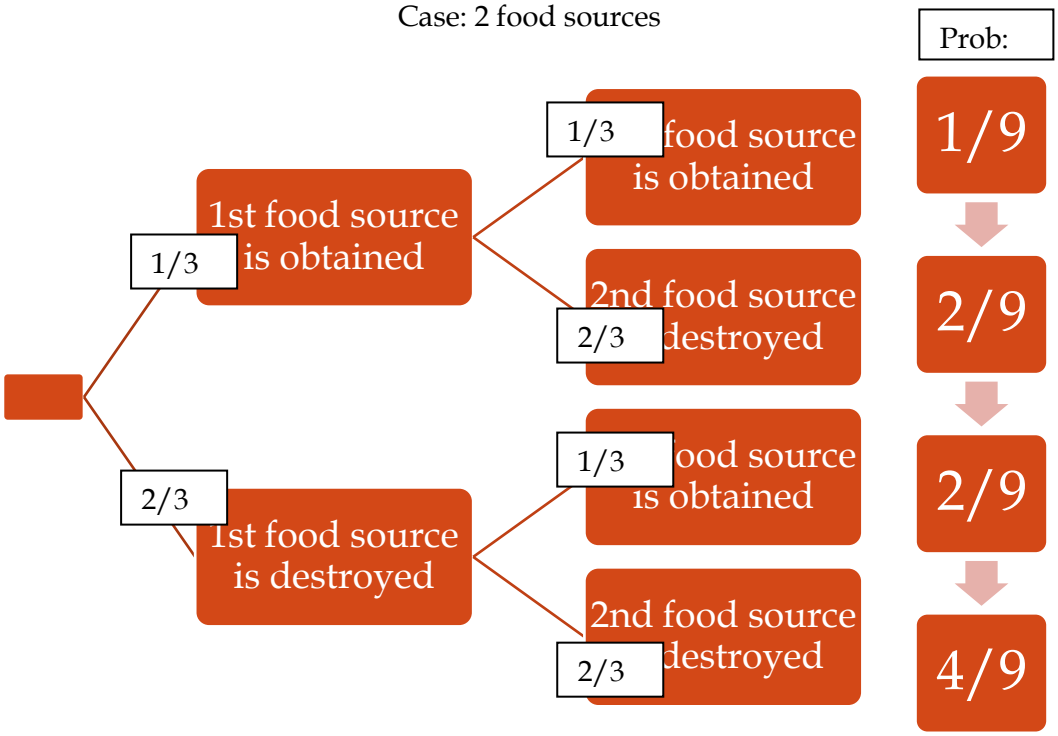
One last think that i need to explain is how i handled chance nodes in this game.

If a particular game configuration has an ant with a food at the position this ant is, this state is marked as a chance node. Then it gets special treatment at the successor function because i must handle both the cases where the food is obtained and when not. However, let us notice that in the case of a long-captivity move there can be more than one source of food and thinks can get complicated. Fortunately, since food only appears in the middle of the board and a captivity move skips two positions it follows trivially that at most 2 food sources can appear in a move. According to the number of food sources we have different scenarios to consider and different probabilities to compute.

Let us first notice, that in the case of only one food source present we have the following cases:

1. The food is obtained with probability 1/3
2. No food is obtained with probability 2/3

Assuming independence among the probabilities of successfully consuming a food source then the following tree diagram can be used to derive all the possible scenarios, as well as their probabilities of occurring in the case of two food sources:



It follows trivially from the chain rule due to independence that we have the following cases:

1. Zero food is consumed with probability 4/9
2. One food source is consumed with probability 4/9
3. Both food sources are consumed with probability 1/9

I used the above cases and probabilities to expand a chance node.

OTHER MINOR THINGS

To my understanding other things are best explained by looking at the source code and thus i didn't devoted a separate section in the report (to honor the 3-4 page limit!).

COMPILATION INSTRUCTIONS AND EXECUTION

In the client folder there is a makefile what can be used by the make utility to compile the client. One think to note is that i have used C++11 that requires a recent conforming compiler (g++ 4.7.2 is the version i used).

The client is run as is explained in the README file that is part of the project. However, i have added two more parameters to the program:

1. -t TIMEOUT can be used to specify the timeout to be used in milliseconds
2. -a AGENT_NAME is used to specify which name to use.

Example invocation from the project folder deliverables:

cd client

make clean

make

./client -i 127.0.0.1 -t 2000 -a tassos