|  |
| --- |
|  |
| Silent Auctions: |
| an Exercise in Contract Negotiation |
|  |
| **CS491 Tyler Rusch** |
| **5/20/2015** |

|  |
| --- |
| Silent auctions are a great example of contract bidding. This paper details how to reliably reproduce a simple multi-agent blackboard like net that makes use of simulated annealing, or steepest ascent methodologies. |

Table of Contents

1. Introduction . . . . . . . . . . . . . . . pg 3

2. Background . . . . . . . . . . . . . . . . pg 3

3. Approach . . . . . . . . . . . . . . . . . . pg 3

4. Results . . . . . . . . . . . . . . . . . . . . pg 5

5. Conclusion . . . . . . . . . . . . . . . . . pg 6

6. Appendix. . . . . . . . . . . . . . . . . . . pg 7

- Silent Auction.java . . . . . pg 7

- Agent.java. . . . . . . . . . . . . pg 12

1. Introduction

Silent Auctions are a good way to create a dynamic blackboard system, with multiple agents to serve as the bidders in the auction. In a standard auction, the bidders all can see the current highest bid, and then make a decision based around that information, including the wants of the bidder, and that they think the value of the item is worth. For this research assignment, we constructed a program that holds a successful auction between 1 and 4 agents, using a choice between two common and useful algorithms called “Simulated Annealing” and “Steepest Ascent”.

1. Background

Silent auctions are common ways to auction off many items at once, and are commonplace in fundraisers across the United States. The idea is to reduce the amount of time spent in the bidding phase, normally under a few minutes, into a multiple hour affair, where the bidders have ample time to consider bids, and value of items. Actual bidding occurs when the bidder places the bid in a location that indicates that they would like to bid on the indicated item, where that information is stored until the end of bidding, with the highest bidder winning the item. This creates a situation where if an item is desired, a bidder will check back with the location, and see if they are still winning the bid.

1. Approach

For a software implementation, the same idea is kept with a multi agent system. Each individual agent looks at the highest bid, and then according to the algorithm specified, the agent will made a bid to the indicated place storage location for bidding information which henceforth is called the “Black-board”. On this “Blackboard” the highest bidder’s information is stored to be used by the next bidder in the queue for the item. To keep things straightforward, our program assumes intense interest. The bidders are constantly checking the “Blackboard” to see if they have been outbid, and if they have been, to consider whether to increase the bid or not.

3.1 Agent.java

This class focuses on each individual agent, how they bid, and the different aspects of the algorithm used to determine bids. It is made up of several methods that help the agent to make a decision.

3.1.1 calcWeight()

This method’s sole purpose is to calculate the agent’s value of the item. This is done with the value the user gives the program at the very beginning. These agents use a simple formula to calculate the value

3.1.2 makeABid(integer currBid)

This method’s purpose is to return the actual bids, and determine which sort of algorithm the agent should use to create a bid. It uses a simple decision tree then calls either steepest or anneal accordingly. It also includes a backup situation in case of inexplicable error where the agent will terminate the bidding.

3.1.3 steepest(integer currBid)

This method uses steepest ascent to assess wether the agent should bid, and what it should bid. It uses strict forms of comparison with no wiggle room. If the current bid (currBid) is less than the current item weight, determined by calcWeight(), then the algorithm makes a bid. Otherwise, it will bid (-1), which is determined as a stop for future bidding.

3.1.4 anneal(integer currBid)

This method uses simulated annealing to assess wether the agent should bid, and what amount to bid. In comparison to steepest ascent above, they similarly use the current bid(currBid) to determine what to bid. The difference lies in the situation where steepest ascent stops. If the current bid is less than the item weight here; instead of terminating, it will calculate a probablility of acceptance. If the probability of acceptance is high enough, the agent will continue bidding, in an attempt to win the bid. The probability will change over time to reflect a lower and lower tolerance for bidding over time, eventually stopping the bidding process.

3.1.4.1 checkProb(double currEn, double newEn, double temp)

This method is a helper for simulated annealing. It literally calculates the acceptance probability, then reduces the temperature. Temperature in simulated annealing is the main determinate in the probability equation which is shown as Figure 1 below.

Using the formula above, correctly calculating the acceptance probability becomes very straight forward

3.1.5 printAgent()

This method is used to print out information for each agent. It serves no purpose other than informing the user what the agent has bid, and what they valued the item at.

3.2 SilentAuction.java

This class focuses on the ”Blackboard” aspect, how many agents are around, ensuring they all get the correct information, and finally interacting with the human user. This class essentially functions as the go between for the software agents and the user. The majority of this class is used in initializing the data for the agents correctly to begin the actual process. Once the process of bidding has begun, a more managerial approach is taken, with the current max bid being taken down; determining which of the agents has stopped bidding, along with the all-important task of determining when to stop the auction.

3.2.1 getInfo()

This method is designed as a helper to the main, by taking in and setting up all the data. Literally this method is the main interface for the user, where questions such as “*Enter the Item’s numeric value*” are asked of the user followed by awaiting a response.

3.2.2 startBid()

This method is designed as the main drive for the main method. While getInfo() takes in user input, this method uses that input with the agents to quickly run through the agents, ensure they bid, and double check to make sure that they have not decided to stop bidding. It also determines the winner of the auction by narrowing down the number of participants until there is only one left, at which point that participant is the winner.

4. Results

The results of the above methods were a program that is efficient in time, resources, and returns an accurate answer consistently. However, it occasionally malfunctions by ending the bid early. This is a problem related to the current bid of the agents, and the stopping mechanism. If multiple agents bid the same amount multiple times in a row, then the agents are highly likely to end the bidding, thinking that they are the winner of the bid. This problem only occurs under highly specific data sets though, wherein the amount of agents is low, and the bid that the agents make is identical to the previous agents. This in essence makes the agent think that they are the sole participant, thus leading to the agent stopping bidding, and the manager terminating the bidding. Other than the aforementioned problem the “Blackboard ” functions efficiently.

4.1 possible applications

This program could be expanded upon to include more forethought by the agents, such as predicting what the opponent is going to bid next, or attempting to subvert the other opponents by bidding just high enough to win the auction, but still get a good “personal” outcome. Since simulated annealing and steepest ascent were both used, there is the possibility of including a teaching algorithm to learn from these two similar types of algorithm, and possibly build a knowledge base from them.

5. Conclusion

Silent Auctions have been a commonplace event for a long time, but as of today, there is a way to accurately model a portion of how bidding occurs. In addition to being simple to use, and fast to execute, it is mildly robust, and will work under lots of different conditions. Overall, a dynamically updated, self-contained blackboard agent system is an effective and efficient tool in the tool belt for the future. This project is considered a full success, and is hereby concluded.

Appendix

Silent auction.java:

import java.util.\*;

/\*\*

\* File: SilentAuction.java

\* Author: Tyler Rusch

\* Date: Spring 2015

\* Description:

\* Acts as a user interface for the manager.

\* 1) User -> Item, Description of the item.

\* 2) User -> Number of Agents

\* 3) User -> Bidding Target.

\*/

public class SilentAuction{

private static String itemName;

private static String description;

private static int userValue;

private static int agentNum;

private static Scanner in = new Scanner(System.in);

private static int runningBid;

private static boolean[] stopped;

private static Agent[] agentPool;

private static String winner;

public static void main(String[] args){

getInfo();

startBid();

}

private static void startBid(){

System.out.println("We will start the bidding at 10.");

System.out.println("Let the Bids for " + itemName + " begin!");

stopped = new boolean[agentNum];

for(int b = 0; b < agentNum; b++){

stopped[b] = false;

}

int turns = 0;

int tempBid;

runningBid = 10;

//we're going to start bidding.

while(true){

turns++;

System.out.println("\nBidding Round " + turns + "\n");

for(int a = 0; a < agentNum; a++){

//when stopped is full, we will stop all the bidding, and the winner will have been the last bidder.

if(stopped[a] != true){

tempBid = agentPool[a].makeABid(runningBid);

agentPool[a].printAgent();

if(tempBid == -1){

//when stopped is full, then we want to make sure that we stop.

stopped[a] = true;

}else{

winner = agentPool[a].getName();

runningBid = tempBid;

}

}

}

System.out.println("\nBidding Round " + turns + " has concluded.\n");

//count up all the stopped agents

int count = 0;

for(int x = 0 ; x < stopped.length; x++){

if(stopped[x] == true){

count++;

}

}

if(count == agentNum){

System.out.println("Bidding has Ceased!");

System.out.println("The Winner of the bid for " +itemName + " is " + winner + "!");

System.out.println("the Final Bid was: " + runningBid + " dollars!");

System.out.println("Thank you for testing with our agents. Have a nice day!");

System.exit(0);

}

}

}

private static void getInfo(){

System.out.println("Welcome to the text interfaced Silent Auction!");

//System.out.println("We will be using two agents today to compete for your item!");

System.out.println("First, please name the item: ");

itemName = in.nextLine();

boolean input = false;

while(!input){

if(itemName != null ||itemName != ""){

input = true;

}else{

System.out.println("The string you entered was not valid.");

System.out.println("Please name the item: ");

itemName = in.nextLine();

}

}

//System.out.println("CHECK Item name : "+ itemName);

System.out.println("Second, Please enter the item's value numerically. ");

System.out.println("Please keep the value above 10, and below 8,000.");

input = false;

while(!input){

userValue = in.nextInt();

if(userValue > 10 && userValue < 8000){

input = true;

}else{

System.out.println("Pardon me, but the value you've provided is out of the limits specified.");

System.out.println("Please keep the value above 10, and below 8,000.");

userValue = in.nextInt();

}

}

//System.out.println("CHECK Item value : "+ userVal);

input = false;

System.out.println("Please establish the number of agents you would like to bid in the auction: ");

System.out.println("Please keep this number positive, and at most 5.");

agentNum = in.nextInt();

while(!input){

if(agentNum > 0 && agentNum <= 5){

input = true;

}else{

System.out.println("Re-enter the number of agents you would like to participate: ");

agentNum = in.nextInt();

}

}

agentPool = new Agent[agentNum];

//System.out.println("CHECK number of participants: "+ agentNum);

for(int i = 0; i < agentNum; i++){

//System.out.println("Please name your agent: ");

String temp = null;

if(temp == null || temp.equals("")){

temp = "Agent "+ (i+1) ;

}

//System.out.println("CHECK name: " + temp);

System.out.println("\nPlease enter the type of bidding you would like "+temp+" to do.");

System.out.println("We ask that you please enter 1 for simulated annealing, and 2 for Steepest ascent.");

int choice = in.nextInt();

if(choice < 0 || choice > 2){

System.out.println("We could not discern your input, so simulated annealing has been chosen to expedite the process.");

choice = 1;

}

agentPool[i] = new Agent(userValue,choice);

agentPool[i].setName(temp);

}

}

}

Agent.java

/\*\*

\* File: Agent.java

\* Author: Tyler Rusch

\* Date: Spring 2015

\* Description:

\* An individual agent, controls itself and makes a decision.

\* 1) Implements simulated Annealing / Steepest Ascent.

\* 2) Makes a bidding decision using the value of the item given by the user to determine the bid it will make.

\* This is done by adding a weight attribute to the item, giving a decreased probability of bidding over time.

\* Hypothesis: this may give simulated annealing a challenge to win a bid.

\*/

public class Agent{

public enum Type{SIM, STEP, CONT}

//the agent's initial starting resources.

private int itemWeight;

private int itemValue;

private String name;

Type agentType;

private double temp = 200.0;

private double coolingRate = 5;

private int ourBid;

private int previousBid;

public Agent(){

itemWeight = 1;

itemValue = 1;

agentType = Type.SIM;

}

public Agent(int itemValue, int det){

//Enum determines the algorithm used.

this.itemValue = itemValue;

calcWeight();

if(det == 1){

agentType = Type.SIM;

//this is simulated annealing

}else if(det == 2){

agentType = Type.STEP;

//this is steepest ascent

}else{

agentType = Type.CONT;

//contingency plan. its a backup which will simple bet the value that is given.

}

}

//this is how far ach individual agent will go Above the actual value.

//it gives each agent a different likely hood of stopping, but at ANY point, the agent may stop.

public void calcWeight(){

this.itemWeight = (int)(((itemValue \* Math.random()) + itemValue));

}

//this method is the main method that carries out the bidding process via self examination and calculation.

public int makeABid(int currBid){

//don't bid if you control the lead!

if(ourBid == currBid){

return -1;

}

previousBid = currBid;

int bid;

if(this.agentType == Type.SIM){

//System.out.println("Simulated Annealing chosen");

bid = anneal(currBid);

}else if(this.agentType == Type.STEP){

//System.out.println("Steepest Ascent chosen");

bid = steepest(currBid);

}else{

//Since all bidding is done based on current bid, we can end early via acceleration.

System.out.println("ERROR: BACKUP USED, ENDING BIDDING VIA ACCELERATION. ");

bid = itemValue \* 50;

}

//System.out.println("CHECK Last Bid: " + bid);

ourBid = bid;

return bid;

}

//bids should be the probability of failure in the endeavour to bid. this should make steepest ascent work properly.

//returns a bid if the weight to win is greater than the value of the current bid. Bid may 1.5x the current bid.

private int steepest(int currBid){

int bid;

if(itemWeight > currBid){

bid = (int)Math.round(currBid +(currBid \* (Math.random()/2.0)));

if( itemWeight >= bid){

return bid;

}else{

bid = (int)Math.round(currBid + (currBid \*.05));

return bid;

}

}else{

//rejected, stop bidding.

return -1;

}

}

//will do the probability calculations and then return a bidding amount.

private int anneal(int currBid){

int bid;

//checkprob determines whether we are beyond our limit for bidding.if we are, we will determine acceptance probability.

double prob = checkProb(itemWeight, currBid, temp);

//check to see if we made a probability less than a dice roll!

//if we did, new bigger bid! up to 1.5x.

if(prob == 1) {

//System.out.println("We've decided to go forward with the best solution.");

bid = (int)Math.round(currBid +(currBid \* (Math.random()/2.0)));

}else if(Math.random() > prob){

//we failed the check! We'll make a smaller bid, and see where it gets us.

//System.out.println("We've tried Taking a possibly bad move.");

bid = (int)Math.round(currBid + (currBid \*.05));

}else{

//this is the rejection case, where we want to stop bidding!

return -1;

}

temp = temp - coolingRate;

return bid;

}

private double checkProb(double currEn, double newEn, double temp){

//check to see if the energy of the new place is better...

if(currEn > newEn){

//New move is better, accepting that one;

return 1.0;

}

//if its not, then we want to calculate the acceptance probability.

return Math.exp((currEn - newEn)/temp);

}

//simple print method to see data.

public void printAgent(){

System.out.println("Agent: "+name+"\nPrevious Bid: " + previousBid + "\nItem Weight:" + itemWeight +"\nOur last Bid: "+ ourBid);

if(agentType == Type.SIM){

System.out.println("Type: Simulated Annealing Agent\n");

}else if(agentType == Type.STEP){

System.out.println("Type: Steepest Ascent Agent\n");

}else if(agentType == Type.CONT){

System.out.println("Type: Control Agent. If you see this, Something went wrong.\n");

}

}

//returns the agents name as a String

public String getName(){

return name;

}

//sets the agents name

public void setName(String name){

this.name = name;

}

//returns the agents type as a String

public String getType(){

if(agentType == Type.SIM){

return("Simulated Annealing");

}else if(agentType == Type.STEP){

return("Steepest Ascent Agent");

}else{

return("Control");

}

}

}