CE322 Algorithmic Game Theory

Student ID: 1604678

# Task 1

## Subtask A

The order of iterated elimination of dominated strategies was in the following order: A, B, C, D, E

A will always take the car, due to the cost of driving always being lower then the cost of taking the bus, even in the best-case scenario (as the cost of taking the car, Ca is always less than 2).

B will always drive a car as well, due to the cost of driving (Cb) being lower than the cost of taking the bus due to commuter A always driving, making the minimum cost of taking the bus equal to 4 (whereas Cb is equal to 3).

C will always drive due to the cost of driving (Cc) being lower then the cost of taking the bus due to commuters A and B always driving, making the cost of the bus at best equal to 6.

D will always drive due to the cost of driving (Cd) being lower then the cost of taking the bus due to commuters A, B and C always driving, which makes the cost of the bus at best equal to 8.

E will always drive, as all the other commuters drive and so the cost of the bus is at best equal to 10, while the cost of driving (Ce) is equal to 9 and thus lower than the cost of the bus.

## Subtask B

The optimal solution (the solution with the highest social welfare and/or lowest social cost) is everyone taking the bus, this solution would cost every commuter 2, a lower cost for every commuter except for commuter A, resulting in an overall social cost of 10.

However as commuter A will never take the bus (as doing so would result in an increased cost for commuter A), therefore the only stable equilibrium that is available involves every single commuter driving (as once A decides not to drive, it becomes optimal for B to drive and so on), which results in an outcome with an overall social cost of 25.

This outcome is a pure Nash equilibrium as for each commuter, their current strategy will result in the highest utility for them.

Therefore, this means that the price of stability (the worst-case ratio between the equilibrium of the game and the optimal solution) is 25.

# Task 2

|  |  |  |
| --- | --- | --- |
|  | **A** | **B** |
| **X** | 4 | 7 |
| **Y** | 8 | 6 |

## Subtask A

A pure Nash equilibrium is where the best response to a given strategy S is the same strategy itself (i.e. there is no reason for any of the players to deviate from their current strategy).

In the case of the zero-sum game present above, there is no pure Nash equilibrium, as whatever option is picked by a player alters the best response to that option for the other player.

For instance, if the row player chooses X then the column player’s best response is to pick A. however if the column player chooses A then the row player’s best choice is to pick Y. This results in the optimal choice for the column player to shift to B, at which point the best choice for the row player becomes X again (shifting the column player’s optimal choice, and so on). Due to the complete lack of stability, it is impossible to have a Nash equilibrium.

## Subtask B

### Row Player Strategy

The goal of the row player is to force the column player to be indifferent to whichever option they choose (both expected utilities should be the same score).

The expected utility for the column player, given that they choose A is:

Where is the probability that the row player chooses X, and is the probability that the row player chooses Y.

The expected utility for the column player, given that they choose B is:

These equations can be used to calculate the probability distribution that the row player needs to make the column player indifferent:

As shown above, the row player will have a mixed strategy where they pick X 40% of the time, and they pick Y (as the player must make a choice, if they do not pick X then they will have picked Y) 60% of the time.

### Column Player Strategy

The goal of the column player is to force the row player to be indifferent to whichever option they choose (both expected utilities should be the same score).

The expected utility for the row player, given that they choose X is:

Where is the probability that the column player chooses A and is the probability that the column player chooses B.

Calculating the mixed strategy for the column player is as follows:

The column player will have a mixed strategy where they pick A 20% of the time and they pick B (as the player must make a choice, if they do not pick A, they must pick B) 80% of the time.

The Expected Game Utility for the Row player can be calculated by summing the probability of each game outcome multiplied by the utility generated for that outcome.

As it is a zero-sum game, the column player’s expected utility is the negation of the row player’s expected utility.

# Task 3

|  |  |  |  |
| --- | --- | --- | --- |
|  | **A** | **B** | **C** |
| **X** | 3, 3 | 0, 4 | 0, 0 |
| **Y** | 4, 0 | 1, 1 | 0, 0 |
| **Z** | 0, 0 | 0, 0 | 0.5, 0.5 |

A correlated equilibrium is an assignment of probabilities to each outcome where everyone follows the recommendations of a mediator.

For everyone to follow the recommendations of the mediator, the expected utility must be greater or equal to the expected utility of disobeying the mediator.

The obedience conditions for our constraints (the expected utility rule stated above) for the row player are listed below:

The obedience conditions for our constraints for the column player are listed below:

In addition to the obedience conditions for each player, we impose the additional constraints that all the possible probabilities sum to 1 (every player must make a choice from the available options) and that the probability of each outcome is greater than or equal to 0 (there are no negative probabilities).

In order to use the linear programming function built into MATLAB, these equations must be re-arranged to create constraints in the form of , as the linprog function solves for minimums. (Note: min -V is equivalent to max V)

This can be done by balancing the equation to have 0 on the right-hand side, and then by negating each component (and switching from ≥ to ≤)

The rearranged obedience conditions for the row player are as follows:

The rearranged obedience conditions for the column player are as follows:

Each constraint will be entered as a row of A.

The order of each entry must be consistent.

Following the mediator’s recommendation should result in at least an equal utility to the average expected utility for a player, ensuring that there is no reason for a rational player to avoid following the mediator’s recommendations.

The mediator probability table below shows that the mediator is always going to recommend players choose Y and B respectively. This outcome has a utility of 2.

|  |  |  |  |
| --- | --- | --- | --- |
|  | **A** | **B** | **C** |
| **X** | 0 | 0 | 0 |
| **Y** | 0 | 1 | 0 |
| **Z** | 0 | 0 | 0 |

## MATLAB Code

% Order of Constraints

% Pxa Pxb Pxc Pya Pyb Pyc Pza Pzb Pzc

A = [

% Row Constraints

[1, 1, 0, 0, 0, 0, 0, 0, 0]; % 1Pxa + 1Pxb + 0Pxc

[-3, 0, 0.5, 0, 0, 0, 0, 0, 0]; % - 3Pxa + 0Pxb + 0.5Pxc

[0, 0, 0, -1, -1, 0, 0, 0, 0]; % - 1Pya - 1Pyb + 0Pyc

[0, 0, 0, -4, -1, 0.5, 0, 0, 0]; % - 4Pya - 1Pyb + 0.5Pyc

[0, 0, 0, 0, 0, 0, 3, 0, -0.5]; % 3Pza + 0Pzb - 0.5Pzc

[0, 0, 0, 0, 0, 0, 4, 1, -0.5]; % 4Pza - 1Pzb - 0.5Pzc

% Column Constraints

[1, 0, 0, 1, 0, 0, 0, 0, 0]; % 1Pax + 1Pay + 0Paz

[-3, 0, 0, 0, 0, 0, 0.5, 0, 0]; % - 3Pax + 0Pay + 0.5Paz

[0, -1, 0, 0, -1, 0, 0, 0, 0]; % - 1Pbx - 1Pby + 0Pbz

[0, -4, 0, 0, -1, 0, 0, 0.5, 0]; % - 4Pbx - 1Pby + 0.5Pbz

[0, 0, 3, 0, 0, 0, 0, 0, -0.5]; % 3Pcx + 0Pcy - 0.5Pcz

[0, 0, 4, 0, 0, 1, 0, 0, -0.5]; % 4Pcx + 1Pcy - 0.5Pcz

];

% All the above constraints resolve to <= 0

b = [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0];

% All probabilities must be <= 1 (as 1 is 100% chance)

Aeq = [1, 1, 1, 1, 1, 1, 1, 1, 1];

% All probabilities must sum to 1 (every player must make a choice)

beq = 1;

lb = [0, 0, 0, 0, 0, 0, 0, 0, 0];

ub = [1, 1, 1, 1, 1, 1, 1, 1, 1];

% Corresponds to the sum of all player's utilities, negated as we will want

% to maximise this (and linprog minimises)

f = -[6, 4, 0, 4, 2, 0, 0, 0, 1];

[probabilities, utility] = linprog(f, A, b, Aeq, beq, lb, ub);

disp(probabilities);

% Since that result technically minimizes the result, the actual utility is

% negated

disp(-utility);

# Task 4

## Subtask A

The social welfare of an individual bidding vector can be calculated as a sum of the benefits each player gets. The optimum social welfare is an instance where the highest ranked value per click is matched with the highest click through rate (and so on). It is important to note that social welfare does not include the cost of the slot that is paid for by the bidder.

The ranking of the players (and the click through rates they will achieve due to their assigned slots) is as follows:

|  |  |  |  |
| --- | --- | --- | --- |
|  | Value Per Click | Click Through Rate (slot) | Welfare |
| Player 1 | 1000000 | 1 (1) | 1000000 |
| Player 2 | 555710 | 0.55071 (2) | 306035.0541 |
| Player 3 | 470400 | 0.4704 (3) | 221276.16 |

The optimal social welfare is when player 1 is allocated the highest slot, followed by player 2 and player 3. This results in an overall social welfare of 1527311.2141.

### MATLAB Code

% A - The optimal/highest social welfare in bidding

% Each index refers to player i

ctr = [1, 0.55071, 0.4704];

values\_per\_click = [1000000, 555710, 470400];

format longG;

% The social welfare of an individual allocation can be calculated as a sum

% of the benefits each player gets.

% The optimum social welfare is an instance where the highest ranked value

% per click is matched with the highest click through rank (and so on).

% As the players are already ranked by value per click and so is ctr, this

% can be calculated as a sum of the element wise multiplication of ctr and

% value per click

optimal\_social\_welfare = ctr .\* values\_per\_click;

disp("Optimal Social Welfare: " + sum(optimal\_social\_welfare));

disp("Optimal Social Welfare Utilities: ");

disp(optimal\_social\_welfare);

## Subtask B

To calculate if a bid is a Nash equilibrium, for each player the bidding vector is altered and a payoff is calculated, if this payoff is higher than the player’s payoff in the original bid then it cannot be a Nash equilibrium.

The calculate cost function takes a bidding vector and a player index and sorts the bids in descending order, it then allocates the player the cost of the next player down in the sorted bidding list, or 0 in the case of the last bidder (who pays nothing).

The calculate ctr\_slot function works in a similar way, returning the ctr\_slot after sorting the bidding vector in descending order.

Calculating the payoff for an individual player is done according to the formula

An attempt at optimizing the code was made by only considering boundary values for potential bids, this resulted in the following equilibrium being calculated.

|  |  |
| --- | --- |
|  | Bid |
| Player 1 | 1 |
| Player 2 | 470399 |
| Player 3 | 470400 |

This combination results in a social welfare of 940800.

To calculate a price of anarchy, the optimal social welfare is divided by the social welfare of the current bid.

The price of anarchy of the given bid is 1.2249504929122

The optimal social welfare can be proven as greater than or equal to the optimal social welfare / 4.

### MATLAB Code

ctr = [1, 0.55071, 0.4704];

values\_per\_click = [1000000, 555710, 470400];

% Each of the possible bids are the boundary values for which a player

% could theoretically bid, it would be preferrable to run through every

% single combination but in practice it would take so long as to be

% meaningless.

p1\_possible\_bids = [1, 470399, 470400, 470401, 55709, 55710, 55711, 999999, 1000000];

p2\_possible\_bids = [1, 470399, 470400, 470401, 555709, 555710];

p3\_possible\_bids = [1, 470399, 470400];

% As the players are already ranked in value order, we can just sum and dot

% multiply here.

optimal\_social\_welfare = sum(values\_per\_click .\* ctr);

welfare\_with\_highest\_anarchy = 0;

equil\_with\_highest\_anarchy = [];

price\_of\_anarchy = -Inf;

alternate\_bids = [p1\_possible\_bids, p2\_possible\_bids, p3\_possible\_bids];

for p1\_bid = p1\_possible\_bids

for p2\_bid = p2\_possible\_bids

for p3\_bid = p3\_possible\_bids

% Assume player's cannot bid the same value

if p1\_bid == p2\_bid || p2\_bid == p3\_bid || p1\_bid == p3\_bid

continue;

end

bid = [p1\_bid, p2\_bid, p3\_bid];

% If the current combination is an equilibrium, append it to the

% vector, we will consider it's price of anarchy and social

% welfare

if (is\_optimized\_equilibrium(bid, values\_per\_click, ctr, alternate\_bids)) == 1

bid\_social\_welfare = sum([

calculate\_welfare(bid, values\_per\_click, ctr, 1);

calculate\_welfare(bid, values\_per\_click, ctr, 2);

calculate\_welfare(bid, values\_per\_click, ctr, 3);

]);

current\_price\_of\_anarchy = optimal\_social\_welfare / bid\_social\_welfare;

% Price of anarchy is OPTsw/SW(b)

if (current\_price\_of\_anarchy > price\_of\_anarchy)

welfare\_with\_highest\_anarchy = bid\_social\_welfare;

price\_of\_anarchy = current\_price\_of\_anarchy;

equil\_with\_highest\_anarchy = bid;

end

end

end

end

end

disp("Optimal Social Welfare");

disp(optimal\_social\_welfare);

disp("Social Welfare of highest price of anarchy bid");

disp(welfare\_with\_highest\_anarchy);

disp("Equilibrium with highest price of anarchy");

disp(equil\_with\_highest\_anarchy);

disp("Price of anarchy");

disp(price\_of\_anarchy);

% This function is a modified variant of calculate\_is\_equilibrium that only

% considers possible bids from the provided array, rather then considering

% all values from 1:true\_value\_of\_player

function is\_equil = is\_optimized\_equilibrium(bid, values\_per\_click, ctr, alternate\_bids)

is\_equil = 1;

original\_payoff = zeros(1, length(bid));

for player\_index = 1:length(bid)

original\_payoff(player\_index) = calculate\_payoff(bid, values\_per\_click, ctr, player\_index);

end

for player\_index = 1:length(bid)

new\_bid = bid;

for alternate\_player\_bid = alternate\_bids(player\_index)

new\_bid(player\_index) = alternate\_player\_bid;

new\_payoff = calculate\_payoff(new\_bid, values\_per\_click, ctr, player\_index);

if new\_payoff > original\_payoff(player\_index)

is\_equil = 0;

return;

end

end

end

end

% This is the actual function that should be used to calculate if the

% current bid is an equilibrium. In practice the execution time was too

% high due to the large number of possible values for each player to bid,

% in combination with the terrible scaling performance of this algorithm.

function is\_equil = calculate\_is\_equilibrium(bid, values\_per\_click, ctr)

is\_equil = 1;

original\_payoff = zeros(1, length(bid));

for player\_index = 1:length(bid)

original\_payoff(player\_index) = calculate\_payoff(bid, values\_per\_click, ctr, player\_index);

end

for player\_index = 1:length(bid)

new\_bid = bid;

for alternate\_player\_bid = 1:values\_per\_click(player\_index)

new\_bid(player\_index) = alternate\_player\_bid;

new\_payoff = calculate\_payoff(new\_bid, values\_per\_click, ctr, player\_index);

if new\_payoff > original\_payoff(player\_index)

is\_equil = 0;

return;

end

end

end

end

% Calculates the payoff given to a specific player given a specific set of

% bids and costs

function payoff = calculate\_payoff(bid, value\_per\_click, ctr, player\_index)

cost = calculate\_cost(bid, player\_index);

value = value\_per\_click(player\_index);

ctr\_slot = calculate\_ctr\_slot(bid, player\_index);

payoff = ctr(ctr\_slot) \* (value - cost);

end

function welfare = calculate\_welfare(bid, value\_per\_click, ctr, player\_index)

value = value\_per\_click(player\_index);

ctr\_slot = calculate\_ctr\_slot(bid, player\_index);

welfare = ctr(ctr\_slot) \* value;

end

function cost = calculate\_cost(bid, player\_index)

[sorted\_bid, indicies] = sort(bid, "descend");

% Look for the index that matches the player\_index, then we use the

% next one as the true cost that player pays (GSP second price)

for position = 1:length(indicies)-1

if indicies(position) == player\_index

cost = sorted\_bid(position+1);

return;

end

end

% The last bidder always pays nothing

cost = 0;

end

function ctr\_slot = calculate\_ctr\_slot(bid, player\_index)

[~, indicies] = sort(bid, "descend");

% Ranks the bids in order to figure out which click through rating the

% specified player should be gettting

for position = 1:length(indicies)-1

if indicies(position) == player\_index

ctr\_slot = position;

return;

end

end

ctr\_slot = length(bid);

end