Spatial Hotelling Problem with Sequential Selection

written by Cyrus Maz, November 3rd, 2020

Problem Description

Three players A, B, C play the following game. First, A picks a real number between 0 and 1 (both inclusive), then B picks a number in the same range (different from A's choice) and finally C picks a number, also in the same range, (different from the two chosen numbers). We then pick a number in the range uniformly randomly. Whoever's number is closest to this random number wins the game. Assume that A, B and C all play optimally and their sole goal is to maximise their chances of winning. Also assume that if one of them has several optimal choices, then that player will randomly pick one of the optimal choices.

1. If A chooses 0, then what is the best choice for B?
2. What is the best choice for A?
3. Can you write a program to figure out the best choice for the first player when the game is played among four players?

Notation

A^*: A's choice, B^*: B's choice , C^*: C's choice

1 If A chooses 0, then what is the best choice for B?

The best choice for B is 2/3.

Proof by contradiction:

If B chooses 2/3, then $C^* \sim Uniform\Big(0, 2/3\Big)$.

Then $P(\text{B winning}|B^* = 2/3, C^*)$=$1/3 + (2/3 - C^*)/2$ and therefore,

$\mathbb{E}[P(\text{B winning}|B^* = 2/3, C^*)] = 1/3 + 1/3 - \mathbb{E}[C^*]/2 + = 2/3 - 1/6 = 1/2$

Now, for a contradiction, suppose the optimal choice for B is $2/3 + \alpha$

Case 1: $\alpha \in \Big(0, 1/3\Big]$

Then $C^* \sim Uniform\Big(0, 2/3 + \alpha\Big)$ and $P(\text{B winning}|B^* = 2/3 + \alpha)$=$1/3 - \alpha + (2/3 + \alpha - C^*)/2$.

Therefore,

$$\begin{aligned}
\mathbb{E}[P(\text{B winning}|B^* = 2/3 + \alpha, C^*)] &= 1/3 - \alpha + (2/3 + \alpha - \mathbb{E}[C^*])/2 \\
&= 1/3 + 1/3 - \alpha + \alpha/2 - \mathbb{E}[C^*]/2 \\
&= 2/3 - \alpha/2 - (2/3 + \alpha)/4 \\
&= 2/3 - 1/6 - 3/4\alpha \\
&= 1/2 - 3/4\alpha < 1/2 = \mathbb{E}[P(\text{B winning}|B^* = 2/3, C^*)]
\end{aligned} \tag{1}$$

where the final inequality holds because $\alpha > 0$.

Case 2: $\alpha \in \Big(-2/3, 0\Big)$

Then $C^* = 2/3 + \alpha + \epsilon$ for some very small $\epsilon > 0$ and

$$\begin{aligned}
\mathbb{E}[P(\text{B winning}|B^* = 2/3 + \alpha, C^*)] &= P(\text{B winning}|B^* = 2/3 + \alpha) \\
&= B^*/2 + (C^* - B^*)/2 \\
&= C^*/2 \\
&= (2/3 + \alpha + \epsilon)/2 \\
&= 1/3 + \alpha/2 + \epsilon/2 \\
&< 1/2 = \mathbb{E}[P(\text{B winning}|B^* = 2/3, C^*)]
\end{aligned} \tag{2}$$

Where the finally inequality holds because $\alpha \in \Big(-2/3, 0\Big)$.

Since other values of α would make B^* fall outside of the unit interval $[0, 1]$ we conclude that the optimal choice for B is 2/3.

Note: as a sanity check, I used the algorithm outlined in my solution to question 3 to solve for the optimal choice for player B, given that A plays 0 in a three player setting, and it yielded the same answer as the one shown here.

■

2 What is the best choice for A?

The best choice for A is drawing randomly from a (discrete) Unifrom{1/4, 3/4} distribution.

Proof:

First, by contradiction we show that choosing 1/2 is sub-optimal for A.

Suppose A chooses 1/2. Then B must choose from the following set: $[0, 1/2) \cup (1/2, 1]$.

Then the optimal choices for B and C are $1/2 \pm \epsilon$ and $1/2 \mp \epsilon$ for a very small $\epsilon > 0$, and:

$$\mathbb{E}[P(\text{A winning}|A^* = 1/2)] = P(\text{A winning}|A^* = 1/2) = \epsilon$$

Then, considering the scenario in question 1, we note that if $A^* = 0$, then $B^* = 2/3$, and subsequently $C^* \sim Uniform\Big(0, 2/3\Big)$. Therefore, $P(\text{A winning}|A^* = 0) = C^*/2$ and thus

$$\begin{aligned} \mathbb{E}[P(\text{A winning}|A^* = 0)] &= \mathbb{E}[C^*]/2 \\ &= 1/6 \\ &> \epsilon = \mathbb{E}[P(\text{A winning}|A^* = 1/2)] \end{aligned} \tag{3}$$

Hence, 1/2 is a sub-optimal choice for A.

Without loss of generality, assume that $A^* < 1/2$. Then,

i. if $B^* \leq 1/2$ and $A^* > B^*$ then $C^* = A^* + \epsilon$, for a very small $\epsilon > 0$ and
$P(\text{B winning}|A^*, B^*) = B^* + (A^* - B^*)/2 = (B^* + A^*)/2$
$P(\text{A winning}|A^*, B^*) = (B^* - A^*)/2 + \epsilon/2$

ii. if $B^* \leq 1/2$ and $A^* < B^*$ then $C^* = B^* + \epsilon$, for a very small $\epsilon > 0$ and
$P(\text{B winning}|A^*, B^*) = (B^* - A^*)/2 + \epsilon/2$
$P(\text{A winning}|A^*, B^*) = A^* + (B^* - A^*)/2 = (B^* + A^*)/2$

iii. if $B^* > 1/2$ and $max(A^*, (B^* - A^*)/2, 1 - B^*) = 1 - B^*$ then $C^* = B^* + \epsilon$ for a very small $\epsilon > 0$ and
$P(\text{B winning}|A^*, B^*) = (B^* - A^*)/2 + \epsilon/2$
$P(\text{A winning}|A^*, B^*) = A^* + (B^* - A^*)/2 = (B^* + A^*)/2$

iv. if $B^* > 1/2$ and $max(A^*, (B^* - A^*)/2, 1 - B^*) = A^*$ then $C^* = A^* - \epsilon$ for a very small $\epsilon > 0$ and
$P(\text{B winning}|A^*, B^*) = (1 - B^*) + (B^* - A^*)/2 = 1 - (B^* + A^*)/2$
$P(\text{A winning}|A^*, B^*) = \epsilon/2 + (B^* - A^*)/2$

v. if $B^* > 1/2$ and $max(A^*, (B^* - A^*)/2, 1 - B^*) = (B^* - A^*)/2$ then $C^* \sim Uniform(A^*, B^*)$,
$P(\text{B winning}|A^*, B^*) = (1 - B^*) + (B^* - C^*)/2 = 1 - 1/2B^* - C^*/2$; and thus,
$\mathbb{E}[P(\text{B winning}|A^*, B^*)] = 1 - 1/2B^* + \mathbb{E}[C^*]/2 = 1 - 1/2B^* - (A^* + B^*)/4 = 1 - 3/4B^* - A^*/4$
$P(\text{A winning}|A^*, B^*) = A^* + (C^* - A^*)/2 = A^*/2 + C^*/2$; and thus,
$\mathbb{E}[P(\text{A winning}|A^*, B^*)] = A^*/2 + (B^* + A^*)/4$

Let us revise the enumerated list above into explicit equations for expected conditional probabilities of A and B winning.

$$\mathbb{E}[P(\text{B winning}|A^*, B^*)] = \begin{cases} (B^* + A^*)/2 + \epsilon/2 \text{ if } B^* \leq 1/2 \text{ and } A^* > B^* \\ (B^* - A^*)/2 + \epsilon/2 \text{ if } B^* \leq 1/2 \text{ and } A^* < B^* \\ (B^* - A^*)/2 + \epsilon/2 \text{ if } B^* > 1/2 \text{ and } max(A^*, (B^* - A^*)/2, 1 - B^*) = 1 - B^* \\ (B^* + A^*)/2 + \epsilon/2 \text{ if } B^* > 1/2 \text{ and } max(A^*, (B^* - A^*)/2, 1 - B^*) = A^* \\ 1 - 3/4B^* - A^*/4 + \epsilon/2 \text{ if } B^* > 1/2 \text{ and } max(A^*, (B^* - A^*)/2, 1 - B^*) = (B^* - A^*)/2 \end{cases}$$

$$\mathbb{E}[P(\text{A winning}|A^*, B^*)] = \begin{cases} (B^* - A^*)/2 + \epsilon/2 \text{ if } B^* \leq 1/2 \text{ and } A^* > B^* \\ (B^* + A^*)/2 + \epsilon/2 \text{ if } B^* \leq 1/2 \text{ and } A^* < B^* \\ (B^* + A^*)/2 + \epsilon/2 \text{ if } B^* > 1/2 \text{ and } max(A^*, (B^* - A^*)/2, 1 - B^*) = 1 - B^* \\ (B^* - A^*)/2 + \epsilon/2 \text{ if } B^* > 1/2 \text{ and } max(A^*, (B^* - A^*)/2, 1 - B^*) = A^* \\ A^*/2 + (B^* + A^*)/4 + \epsilon/2 \text{ if } B^* > 1/2 \text{ and } max(A^*, (B^* - A^*)/2, 1 - B^*) = (B^* - A^*)/2 \end{cases}$$

We now have everything we need to find the optimal choice for A numerically. Let

$$\tilde{B} = \left\{(a, b) : b = \underset{b}{\text{argmax}}\, \mathbb{E}[P(\text{B winning}|A^* = a, B^* = b)], a \in [0, 1/2)\right\} \text{ and let}$$

$$\tilde{A} = \left\{a : a = \underset{a}{\text{argmax}}\, \mathbb{E}[P(\text{A winning}|A^* = a, B^* = b)], (a, b) \in \tilde{B}\right\}$$

Then $\tilde{A}$ is the set of optimal plays by A in the interval $[0, 1/2)$. We will approximate $\tilde{B}$ by grid search over $(A^* \times B^*) \in [0, 1/2] \times [0, 1]$. Next, we will calculate (approximately) $\tilde{A}$ by searching through the approximation of $\tilde{B}$.

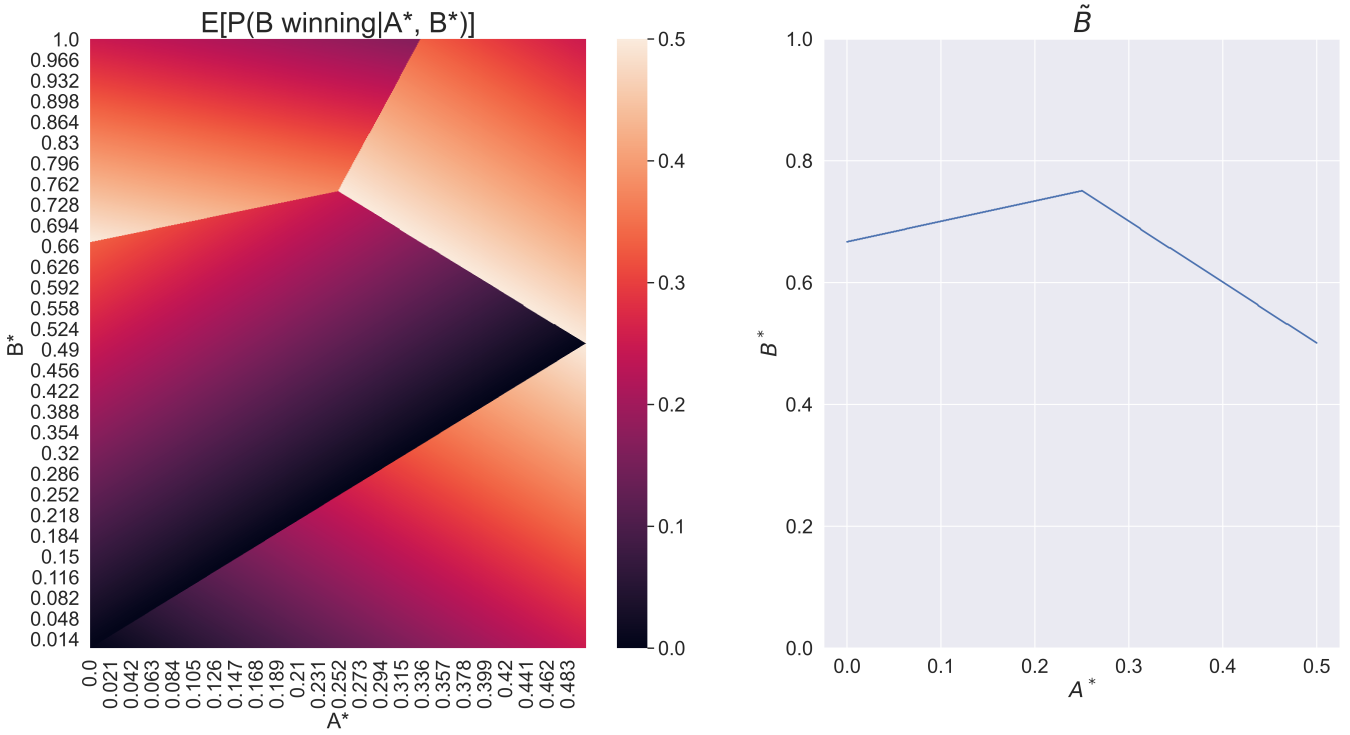


Figure 1: (Question 2) Left: the surface of the expectation of the conditional probability of B winning represented as a heatmap. Right: Approximation of $\tilde{B}$ computed by grid search.

Using the Python code included in Appendix A, with a grid step size of 0.001, we find that $\tilde{A} = \{0.25\}$ Then, by symmetry, 0.75 is also an optimal choice for A. Thus, we have that the optimal choice for A is drawing randomly from a (discrete) Unifrom$\{1/4, 3/4\}$ distribution.

Note: as a sanity check, I used the algorithm outlined in my solution to question 3 to solve for the optimal choice for player A in a three player setting, and it yielded the same answer as the one calculated here. Output from question 3's algorithm used to solve this question is included at the end of Appendix B.

3 Can you write a program to figure out the best choice for the first player when the game is played among four players?

3.1 Caveats

We must first note that since no player may choose a number already chosen by any of the previous players, the space of potential choices for all but the first player are non-compact sets. In a strictly mathematical sense, this makes "optimal" choices to *not exist* in certain cases. To demonstrate this subtle point, consider the following scenario in a two player setting: player A chooses 1/3; then the choice for player B that *maximizes* their probability of winning is $1/3 + \epsilon$, for a very small $\epsilon > 0$ which yields $P(\text{B winning}|A^* = 1/3) = 1 - (1/3 + \epsilon) = 2/3 - \epsilon$. Note that

$$\lim_{\epsilon \to 0^+} P(\text{B winning}|A^* = 1/3) = 2/3$$

in a monotonically increasing manner, and thus $P(\text{B winning}|A^* = 1/3)$ cannot be *maximized* over $(1/3, 1]$.

To circumvent this mathematical nuance, we will impose a minimum acceptable value for ϵ. Let ϵ_0 denote this minimum acceptable value for ϵ.

Furthermore, to solve the puzzle algorithmically, we will approximate the exact solution, by discretizing $[0, 1]$ into M equally-lengthed intervals, thereby restricting the set of possible choices to $\{i/(M+1) : i = 0, 1, ..., M-1, M, M+1\}$ for some large integer M. In this discrete setting, ϵ_0 must be $\leq 1/M$.

3.2 General Procedure

Let N be the number of players. Let M be the number of equally spaced intervals $[0, 1]$ is discretized into. Consequently, the set of possible choices is restricted $\{i/(M + 1) : i = 0, 1, ..., M - 1, M, M + 1\}$. Let $\epsilon_0 = 1/M$. Start with every possible (optimal and non-optimal) sequence of choices by the first $N - 1$ players. Then for each sequence, we find the optimal choice for the Nth player. Algorithm 1, *optimal_path_calculator(sequence)*, outlines the procedure for this task. A sequence of one choices for the first $N - 1$ players followed by the corresponding optimal choice for the N-th player (conditional on the first $N - 1$ choices), shall be referred to as a *path* henceforth. A set of paths such that an initial sequence (of some length) is shared amongst the paths shall be referred to as a *group* henceforth. The set of all groups shall be referred to as the *path-space* henceforth. We will regroup and prune the path-space, in essence optimizing the choice of each player, according to what the subsequent player(s) will play conditional on the choice of the player in question as well the choice(s) of any preceding player(s). We repeat this procedure iteratively starting with the Nth player and working backwards until we find the optimum choice(s) for the first player. The pseudocode in Algorithm 2, *backward_solver(M,N)*, describes this procedure in more detail. Working Python code as well as example output for various values of M are included in Appendix B.

Note that in the code in the appendix, we restrict the path-space to those whose choice of player A is in $[0, 1/2] \cap \{i/(M + 1) : i = 0, 1, ..., M - 1, M, M + 1\}$. This is to save on computational cost and is justified by the fact that if some value $a*$ is optimal for player A on $[0, 1/2]$, then by symmetry, $1 - a*$ is also optimal.

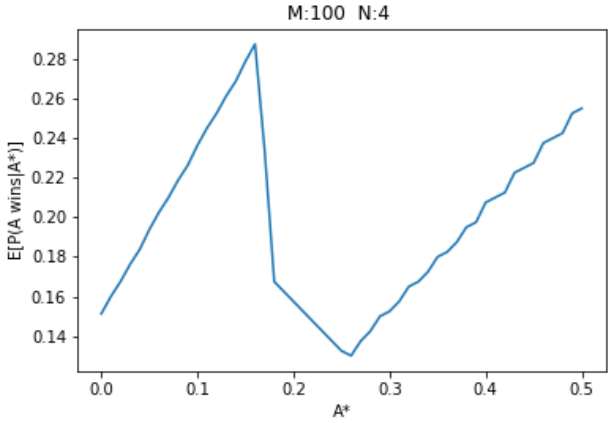
3.3 Results & Discussion

Running the Python script with $N = 4$ and $M = 100, 127, 200$ gives optimal choices for A approximately equal to 0.16 and 0.83. It is plausible that the exact answer is 1/6 and 5/6 as the estimates are close to these fractions.

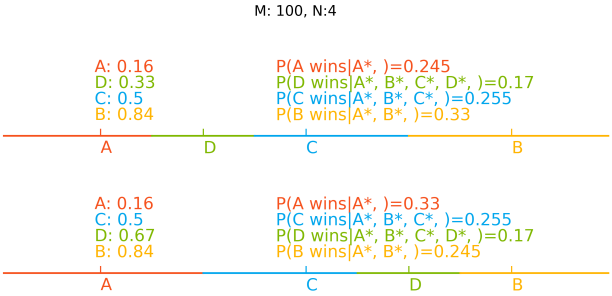
Figure 2 summarizes these results for $N = 4$ and $M = 100, 127, 200$. The plots on the left show the expected probability of A winning for each $A^* \in [0, 1/2] \cap \{i/(M + 1) : i = 0, 1, ..., M - 1, M, M + 1\}$. The plots on the right show the possible sequence of optimal choices by each player. (Note that in our script, whenever D has an interval of optimal choices available to them, we choose the middle point for accuracy and savings on computational cost. More on this in section 3.4.)

Discretizing the interval $[0, 1]$ results in this solution being an approximation. I believe that for sufficiently large M, the solution produced by this algorithm can be arbitrarily close to the exact solution. However, the computational cost for large choices of M and N are significant. For this reason I was unable to arrive at an answer with $M > 200$, in the four player setting.

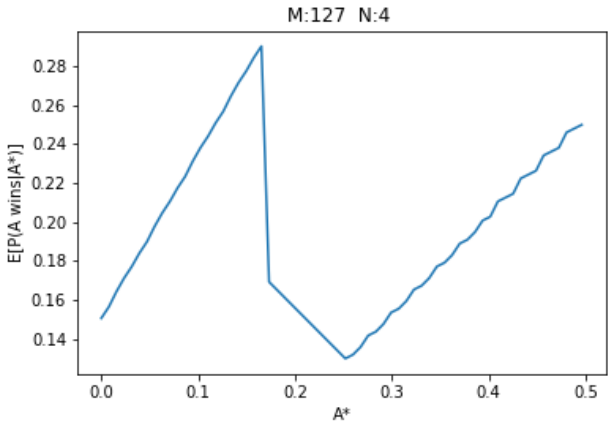
3.3.1 Plots



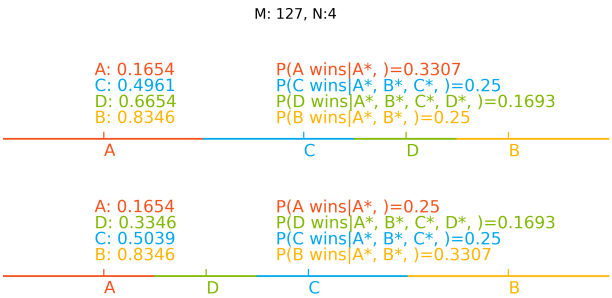
(a)(i)



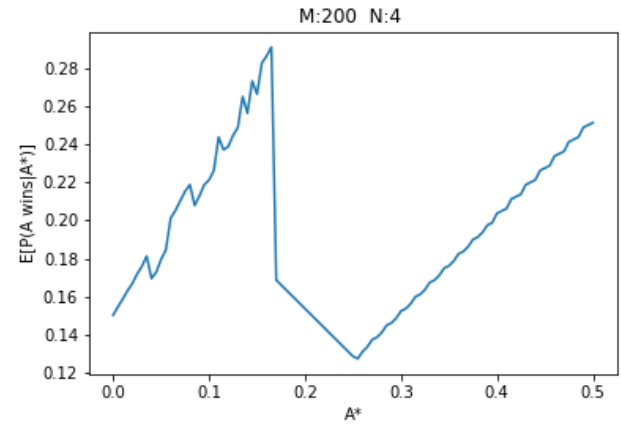
(b)(ii)



(c)(i)

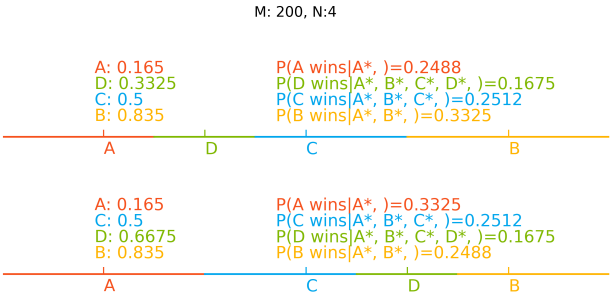


(d)(ii)



h

(e)(i)



(f)(ii)

Figure 2: The general shape of the mapping $[0, 1/2] \mapsto \mathbb{E}[P(\text{A winning}|A^*)]$ is consistent for all values of M tested. Interestingly the mapping appears to be less smooth for M=200 than M=100 or 127.

3.4 Key Algorithms

Algorithm 1: *optimal_path_calculator(sequence)*

Let ϵ_0 denote the minimum acceptable ϵ
Let $\{x_1, x_2, ..., x_{N-1}\}$ denote the sequence of *sorted* choices of the first $N-1$ players
Let $d = \{x_1, max(x_j - x_{j-1})/2, 1 - x_{N-1} : j = 1, ..., N\}$
Let $d^* = max(d)$
Let $p = []$ be an empty list to be filled with optimal choices by the Nth player
for each $j \in \{2, ..., N-1\}$ **do**
  if $(x_j - x_{j-1})/2 = d^*$ **then**
    append $(x_j + x_{j-1})/2$ to p
  end if
end for
if $x_1 = d^*$ **then**
  append $x_1 - \epsilon_0$ to p
end if
if $x_{N-1} = d^*$ **then**
  append $x_{N-1} + \epsilon_0$ to p
end if Return p

Algorithm 2: *backward_solver(M,N)*

Find all permutations of possible (optimal and non-optimal) plays by the first N-1 players
For each permutation, find the optimum play(s) by the N-th player
The sequence of each permutation, and the corresponding optimal choices by the N-th player constitutes a group
Set $k = N$
while $k > 0$ **do**
  # (find the optimal paths for the (k-1)th player)
  # Average
  for each group in the path-space **do**
    compute the average payoffs of all paths in the group
  end for
  # Regroup
  if $k - 2 > 0$ **then**
    Regroup: One group for each unique initial permutation sequence of length $k-2$ i.e. the first $k-2$ permutation sequence is shared among members of each group
  else
    Regroup: Take the union of all remaining groups so that there is only one group in the path-space
  end if
  # Optimize
  for each group in the path-space **do**
    Discard every path that is sub-optimal (in terms of the group payoff average) for the (k-1)th player
  end for
end while
Return every unique play by the first player in the remaining paths

It should be noted that *optimal_path_calculator(sequence)* does not explicitly account for the different choices the final player can make when an interval of optimal choices is available to them. This is justified by the fact that when estimating the expected probabilities of winning for the previous players, using the midpoint of an interval that is optimal for the final player as their choice, rather than averaging the probabilities of winning corresponding to multiple discrete points on the interval, as we do for the preceding players, has two advantages: 1. more accurate estimation for the expected conditional probabilities of player $N-1$ winning (this boost of accuracy is preserved and transferred backward at each iteration of *backward_solver(M,N)*), and 2. significant savings on computational cost. Furthermore, when multiple optimal intervals are available to the final player the algorithm records the midpoint of each interval as an optimal choice; and when singular optimal choices are available, *optimal_path_calculator(sequence)* does in fact record each one as an optimal choice.

Appendix A Code for Question 2 (hotelling_q2.py)

```python
import itertools

def P_B_wins(A,B):
    if (B<=1/2) and (A>B):
        return (B+A)/2
    if (B<=1/2) and (A<B):
        return (B-A)/2
    if (B>1/2) and (max(A,1-B,(B-A)/2)==1-B):
        return (B-A)/2
    if (B>1/2) and (max(A,1-B,(B-A)/2)==A):
        return 1-(B+A)/2
    if (B>1/2) and (max(A,1-B,(B-A)/2)==(B-A)/2):
        return 1-3/4*B-A/4
    if (B==A): return 0

def P_A_wins(A,B):
    if (B<=1/2) and (A>B):
        return (B-A)/2
    if (B<=1/2) and (A<B):
        return (B+A)/2
    if (B>1/2) and (max(A,1-B,(B-A)/2)==1-B):
        return (B-A)/2
    if (B>1/2) and (max(A,1-B,(B-A)/2)==A):
        return (B-A)/2
    if (B>1/2) and (max(A,1-B,(B-A)/2)==(B-A)/2):
        return A/2+(B+A)/4
    if (B==A): return 0

mesh_width = 1000
A_vals = [x/mesh_width for x in range(int(mesh_width/2)+1)]
B_vals = [x/mesh_width for x in range(int(mesh_width)+1)]
grid = list(itertools.product(A_vals, B_vals))

Tilde_B = []
for A in A_vals:
    val=[P_B_wins(A,B) for B in B_vals]
    max_val = max(val)
    max_index = [i for i, j in enumerate(val) if j == max_val]
    Tilde_B.append(B_vals[max_index[0]])

Tilde_A = [dict(A=a,P_A_wins=P_A_wins(a,b)) for a,b in zip(A_vals, Tilde_B)]
seq = [p['P_A_wins'] for p in Tilde_A]
max_seq=max(seq)
Tilde_A = list(filter(lambda x: x['P_A_wins']==max_seq, Tilde_A))
print(Tilde_A)

# [{'A': 0.25, 'P_A_wins': 0.37525}]
```

Appendix B Code for Question 3 (hotelling_q3.py)

```
from copy import deepcopy
from itertools import permutations, groupby
from functools import reduce
from operator import add
from collections import Counter
from math import isclose
import json

# Given a path, calculate the probability of each player winning
# INPUT:
#    path: a path of the form {'a':0.2, 'b': 0.5, 'c':0.8,... }
# OUTPUT: the payoff for each player, where payoff is defined
#         as the probability of the player winning the game
def payoff_calculator(path):
    sorted_keys=sorted(path, key=path.get)
    output=dict()
    for k in range(len(sorted_keys)):

        if k==0:
            lowbo=0
            upbo=(path[sorted_keys[k]]+path[sorted_keys[k+1]])/2
        elif k==len(sorted_keys)-1:
            lowbo=(path[sorted_keys[k-1]]+path[sorted_keys[k]])/2
            upbo=1
        else:
            lowbo=(path[sorted_keys[k-1]]+path[sorted_keys[k]])/2
            upbo=(path[sorted_keys[k]]+path[sorted_keys[k+1]])/2
        output[sorted_keys[k]]=upbo-lowbo
    return output

# generate every possible sequence of choices by the first N-1 players
# INPUTS:
#   M: determines the choices for players are {i/(M+1) : i=0,1,...,M-1,M,M+1}
#   N: the number of players
# OUTPUT: all possible permutations of choices to play by the
#         first N-1 players
def play_generator(M, N):
    discretized = [x/(M) for x in range(0,M+1)]
    perms = list(permutations(discretized, N-1))
    output = []
    for perm in perms:
        processed_perm = {alphabet[i]:perm[i] for i in range(N-1)}
        output.append(processed_perm)
    return output

### ALGORITHM 1:
# compute the optimal choice for the N-th player, given a sequence of choices by the first
# N-1 players
# INPUTS:
#   path: a path of N-1 plays by the first N-1 players, {'a':0.2, 'b': 0.5, ... }
# OUTPUT: a dictionary consisting of two key/value pairs:
#   'path_group': a list of dicitonaries of paths that are equal to the input
#                 path in the first N-1 plays, and have optimal plays by the Nth player
#   'payoff_group': a list of dictionaries of payoffs corresponding to the respective path
#.                  in the path_group list
```

```
def optimal_path_calculator(path):
    letter = alphabet[len(path)]
    vals = list(path.values())
    vals.sort()
    intervals = []
    max_len = 0
    for i in range(len(vals)+1):
        if i==0: lobo=0
        else: lobo=vals[i-1]

        if i==(len(vals)): upbo=1
        else: upbo=vals[i]

        if not ((i==0) or (i==(len(vals)))):
            length=(upbo-lobo)/2
        elif ((i==0) or (i==(len(vals)))):
            length=upbo-lobo

        if i==0:
            lobo='zero'
        elif i==len(vals):
            upbo='one'

        intervals.append([lobo, upbo, length])
        if length>max_len: max_len=length

    intervals=list(filter(lambda x: isclose(x[2],max_len,abs_tol=r), intervals))

    path_list = []
    for interval in intervals:
        if interval[0]=='zero':
            path_list.append(min(vals)-e)

        elif interval[1]=='one':
            path_list.append(max(vals)+e)
        else:
            path_list.append((interval[0] + interval[1])/2)

    path_group=[]
    payoff_group=[]
    for n in path_list:
        newdict = deepcopy(path)
        newdict[letter]=n
        path_group.append(newdict)
        payoff_group.append(payoff_calculator(newdict))

    return {'path_group':path_group, 'payoff_group':payoff_group}

# compute the average payoff for all paths in the group
def payoff_average_calculator(group):
    payoff_group = group['payoff_group']
    payoff_average = dict(reduce(add, map(Counter, payoff_group)))
    payoff_average = {k:v/len(payoff_group)  for k,v in payoff_average.items()}
    group['payoff_average'] = [payoff_average]*len(payoff_group)
    return group

# regroup path_space such that the first k-2 in each group are identical
```

```python
def regroup(path_space, k):
    path_group_agg = []
    payoff_group_agg = []
    payoff_average_agg = []

    for e in path_space:
        for path, payoff_group, payoff_average in zip(
            e['path_group'],
            e['payoff_group'],
            e['payoff_average']
            ):
            path_group_agg.append(path)
            payoff_group_agg.append(payoff_group)
            payoff_average_agg.append(payoff_average)

    trip = zip(path_group_agg, payoff_group_agg, payoff_average_agg)
    l=alphabet[0:k-2]
    print('regrouping such that choices for ({}) in each group are equal'.format(l))
    regrouped_raw=[
        [*j] for i, j in
            groupby(
                sorted(trip, key=lambda x: [x[0][i] for i in l]),
                key=lambda x: [x[0][i] for i in l])
            ]

    regrouped = []
    for group in regrouped_raw:
        path_group = []
        payoff_group = []
        payoff_average = []
        for path in group:
            path_group.append(path[0])
            payoff_group.append(path[1])
            payoff_average.append(path[2])
        d=dict(   path_group=path_group,
                  payoff_group=payoff_group,
                  payoff_average=payoff_average)
        regrouped.append(d)
    return regrouped

# for each group in path_space, discard every path that is suboptimal for player k-1
def optimize_groups(path_space, k):
    letter = alphabet[k-2]
    print('optimizing {}'.format(letter.upper()))
    optimized = []

    for i in range(len(path_space)):
        group = deepcopy(path_space[i])
        seq=[g[letter] for g in group['payoff_average']]
        max_pay = max(seq)
        filtered_group = list(filter(
            lambda x: isclose(x[2][letter],max_pay,abs_tol=r),
            zip(group['path_group'], group['payoff_group'],group['payoff_average'])
            ))
        filtered_group = list(zip(*filtered_group))
        optimized.append(
            {'path_group': list(filtered_group[0]), 'payoff_group': list(filtered_group[1])})
```

```
    return optimized

# recursively carry out the while loop in Algorithm 1
def recursive_solver(path_space, k):
    print("k={}".format(k))
    path_space_withaverage = [payoff_average_calculator(o) for o in path_space]
    path_space_regrouped = regroup(path_space_withaverage, k)
    path_space_optimized_by_group = optimize_groups(path_space_regrouped, k)
    if k==2:
        return [payoff_average_calculator(o) for o in path_space_optimized_by_group]
    else:
        k-=1
        return recursive_solver(path_space_optimized_by_group,k)

### ALGORITHM 2:
# wrapper for recursive_solver
def backwards_solver(M,N, question_1=False, write=False):
    print('M={}'.format(M))
    print('N={}'.format(N))
    path_space = play_generator(M,N)

    if question_1 is True:
        path_space = list(filter(lambda x: x['a']==0, path_space))
    else:
        path_space = list(filter(lambda x: x['a']<=1/2, path_space))

    path_space = [optimal_path_calculator(play) for play in path_space]
    optimal_paths = recursive_solver(path_space,N)

    path_group=optimal_paths[0]['path_group']
    average_payoff = optimal_paths[0]['payoff_average'][0]

    if question_1 is False:
        A = list(set([x['a'] for x in path_group]))
        print('optimal choice(s) for A: {}'.format(A))
        print('average payoff: {}'.format(average_payoff))
        print('optimal paths: {}'.format(path_group))

    elif question_1 is True:
        B = list(set([x['b'] for x in path_group]))
        print('optimal choice(s) for B: {}'.format(B))
        print('average payoff: {}'.format(average_payoff))
        print('optimal paths: {}'.format(path_group))

    if write is True:
        with open('optimal_paths_M{}_N{}_r{}.txt'.format(M,N,r), 'w') as filehandle:
            json.dump(optimal_paths, filehandle)

##############################################################################
# e: the minimum allowable epsilon
# r: absolute tolerance when comparing floating points
e = 1/100000
r = e/10
alphabet = ['a', 'b', 'c', 'd']
##############################################################################
# QUESTION 3: Optimal choie for player A in four player setting.
backwards_solver(M=100, N=4,question_1=False, write=False)
```

```
# optimal choice(s) for A: [0.16]
# average payoff:
#     {'a': 0.2875, 'd': 0.16999999999999998, 'c': 0.2549999999999999, 'b': 0.2875000000000001}
# optimal paths: [{'a': 0.16, 'b': 0.84, 'c': 0.5, 'd': 0.33},
#                 {'a': 0.16, 'b': 0.84, 'c': 0.5, 'd': 0.6699999999999999}]

# backwards_solver(M=127,N=4,question_1=False, write=False)
# optimal choice(s) for A: [0.16535433070866143]
# average payoff:
     # {'a': 0.29035433070866146,
     #  'c': 0.24999999999999992,
     #  'd': 0.1692913385826772,
     #  'b': 0.29035433070866146}
# optimal paths: [
#      {'a': 0.16535433070866143,
#       'b': 0.8346456692913385,
#       'c': 0.49606299212598426,
#       'd': 0.6653543307086613},
#      {'a': 0.16535433070866143,
#       'b': 0.8346456692913385,
#       'c': 0.5039370078740157,
#       'd': 0.3346456692913386}]

# backwards_solver(M=200,N=4,question_1=False, write=False)
# optimal choice(s) for A: [0.165]
# average payoff: {'a': 0.290625, 'd': 0.16749999999999998, 'c': 0.25125, 'b': 0.290625}
# optimal paths: [{'a': 0.165, 'b': 0.835, 'c': 0.5, 'd': 0.3325},
#                 {'a': 0.165, 'b': 0.835, 'c': 0.5, 'd': 0.6675}]

########################################################################
# QUESTION 1: Optimal choice for player B when A plays 0.
# backwards_solver(M=100,N=3,question_1=True, write=False)
# optimal choice(s) for B: [0.67]
# average payoff: {'a': 0.1675, 'c': 0.3350000000000001, 'b': 0.4974999999999994}
# optimal paths: [{'a': 0.0, 'b': 0.67, 'c': 0.335}]

########################################################################
# QUESTION 2: Optimal choie for player A in three player setting.
# backwards_solver(M=100,N=3,question_1=False, write=False)
# optimal choice(s) for A: [0.25]
# average payoff: {'c': 0.24999666666666664, 'a': 0.3750016666666667, 'b': 0.3750016666666667}
# optimal paths: [{'a': 0.25, 'b': 0.75, 'c': 0.24999},
#                 {'a': 0.25, 'b': 0.75, 'c': 0.5},
#                 {'a': 0.25, 'b': 0.75, 'c': 0.75001}]
```