THE GAME OF WAR

ALISON BU, ROBERT DOUGHERTY-BLISS, CHARLES KENNEY

> *Primitive man tossed his arrows into the magic ring which he had drawn on the ground with the proper rites and incantations. And who can say that some power higher than the stars did not direct their flight, and bring him in safety to his journey's end?*
>
> — Catherine Hargrave, *A History of Playing Cards*

1. Introduction

We, the war team, were assigned to study the game of War, a classic, internationally-known card game. To this end, we have written a comprehensive collection of Maple code to simulate and analyze War and various generalizations we have considered.

The main items of our report are as follows:

- A large body of Maple code.
- Pictures of the "War graph" for small hands.
- A definition of the sequence, "longest possible game of War on the deck $[n] = \{1, 2, \ldots, n\}$."

Our report is structured into three parts. Section 2 contains our (failed) attempt to reconstruct the history of War[1]. Section 3 contains all of our Maple code. Section 4 contains some notes on the "War graph" as well as some pictures we made.

2. A History of Cards

The game of war seems nearly universal. In America, we play *War*; in England, they call it *Battle*; the Germans have a varient called *Tod und Leben*. Despite its popularity, the origins of War are completely mysterious. There are references to *Tod und Leben* found in nineteenth century game books, but surely the game of war existed before this in different forms. Lacking any authoritative history on the subject, it

Date: May 11, 2020.

[1]The game, of course.

seems just as good to talk about the history of playing cards themselves, and perhaps surmise when War *could* have been invented based on this.

Playing cards have surely existed in some form for much longer than we have records of them. Stewart Culin, the great American game expert and early ethnographer, believed that playing cards descended from divination rituals, which existed in nearly every culture around the world: "In order to classify objects and events which did not in themselves reveal their proper assignment, resort was had to magic. Our present games are the survivals of these magical processes" [1, p. 1].

Nonetheless, the earliest *known* playing cards are Chinese and from the twelfth century. The cards were copied from Chinese notes invented in the Tang Dynasty (est. 618 CE)— gamblers likely played with the notes themselves. Other popular cards, used interchangeably, were *domino cards*, cards with patterns of dots added for significance. These are quite beautiful—see Figure 1.

FIGURE 1. Chinese domino cards depicting characters from the story of the river banks. From [1].

It is unclear how or in what form playing cards first spread to Europe. One popular theory asserts that "wandering gypsies" brought them from India. Culin believed the opposite, that Indian playing cards were inspired by early European developments. However they arrived, the modern playing card was developed in France.

The first French playing cards were tarot cards—still available today—painted for King Charles VI in 1392 CE, and popular among working-people no later than 1397 CE. Tarot cards come in collections of twenty-two, painted with figures and scenes which hold some significance, such as *Le bateleur* ("The magician"), *Le chariot*, and *La justice*. The cards begin with suites of Cups, Swords, Coins, and Batons, but were changed to Hearts, Clubs, Spades, and Diamonds when, according to legend, the famous Hundred Years' War commander La Hire invented the game of piquet. At this time the cards were numbered linearly *by suite*. Around 1440, the first "court" (face) cards were named. By 1516 the standard 52-card deck had been invented. These are essentially the cards that we still use today, some half-a-millennium later.

In summary, playing cards with numeric values have been around for hundreds of years. Some of the earliest games we have records of are already more complicated than War. It seems difficult to pinpoint exactly when the game started, but it must have been well-over two-hundred years ago, if not *significantly* further in our history.

3. Procedures

3.1. **Setting Up the Game.**

3.1.1. *Distributing the Deck.*

The procedure **RanDeck(n,k,r)** inputs positive integers n, k, and r. It simulates dealing out a randomly shuffled deck of cards containing k copies of n unique, ordered cards to r players.[2] The maximal number of cards such that each player has the same number of cards are dealt out and any remaining cards are removed from the game.

RanDeck(n,k,r) partitions the first $\lfloor \frac{n*k}{r} \rfloor * r$ entries of a random permutation of $[\underbrace{1,...,1}_{r \text{ copies}},...,\underbrace{n,...,n}_{r \text{ copies}}]$ by creating a new part after every $\lfloor \frac{n*k}{r} \rfloor$-th entry. It outputs this partition as a list of length r, where the i-th entry is a list that represents the i-th players deck.

```
1 RanDeck:=proc(n,k,r) local HandSize, i, pi, Deck,
     decks, discard:
2  Deck:=[seq(seq(i,j=1..k),i=1..n)]:
```

[2]In other words, kn cards with k suits.

```
Deck:=randperm(Deck):
HandSize:=floor(n*k/r):
decks:=[seq([seq(Deck[i],i=(j-1)*HandSize+1..j*
  HandSize)],j=1..r)]:
end:
```

3.1.2. *Defining War.*

A war occurs when the highest card played in a round is not unique. In a war, players place m cards face down and then play their next card. The following three procedures define different ways to complete a war when a player does not have enough cards. Each of the following procedures input a non-negative integer m, a list Q, and a positive integer r, where $Q[i]$ represents the i-th player's deck and $|Q| = r$.

The procedure **WARm(m,Q,r)** simulates a war where m cards must be placed face down, and any player that does not have enough cards automatically loses. If no player has enough cards to complete the war, it outputs $\{0\}$ to indicate that the game will have no winners. Otherwise, it outputs a list $[tempQ, b]$, where $tempQ$ is a list of each players deck after placing m cards face down and b is a list of the cards placed face down or played during the war.

```
WARm:=proc(m,Q,r) local HandSize, tempQ,i,output,a
   ,b:
   HandSize:=[seq(nops(Q[i]), i=1..r)]:
   if max(HandSize)>m then tempQ:=Q: a:=Q:
   for i from 1 to r do
       if HandSize[i]<m+1 then tempQ[i]:=[]:   a[i
   ]:=[op(1..-1,Q[i]),seq(0,j=HandSize[i]..m)]:
       else tempQ[i]:=[op(m+1..-1,Q[i])]: a[i]:=[
   op(1..m,Q[i])]:
       fi: od:
   b:=remove(t->t=0,[seq(seq(a[i][j],i=1..r),j=1..
   m)]):
   output:=[tempQ,b]:
   else output:={0}: fi:
   output:
end:
```

The procedure **WARmin(m,Q,r)** inputs a non-negative integer m, a list Q, and a positive integer r such that $|Q| = r$. It simulates a war where $Q[i]$ represents the i-th player's deck, and m cards must be placed face down unless any player still in the game does not have enough cards to do so. If the smallest hand at the start of the war is $k < m + 1$, then every player places $k - 1$ cards face down and plays

their next card. If no player has any remaining cards, it outputs 0 to indicate that the game will end in a tie. Otherwise, it outputs a list $[tempQ, b]$, where $tempQ$ is a list of each players deck after placing their cards face down and b is a list of the cards placed face down or played during the war.

```
WARmin:=proc(m,Q,r) local HandSize, hs, tempQ,i,
   output:
   HandSize:=[seq(nops(Q[i]),i=1..r)]
   if add(HandSize[i],i=1..r)<1 then output:=0:
   else hs:=min(remove(t->t=0,HandSize)):
        if hs>m then output:=WARm(m,Q,r):
        else output:=WARm(hs-1,Q,r):
        fi:
   fi:
   output:
 end:
```

The procedure **WARmax(m,Q,r)** inputs a non-negative integer m, a list Q, and a positive integer r such that $|Q| = r$. It simulates a war where the i-th entry of Q represents the i-th player's deck, and m cards must be placed face down unless no player still in the game has enough cards to do so. If the largest hand at the start of the war is $k < m + 1$, then every player places $k - 1$ cards face down and plays their next card. Any player that does not have nough cards to do so automatically loses. If no player has any remaining cards, it outputs 0 to indicate that the game will end in a tie. Otherwise, it outputs a list $[tempQ, b]$, where $tempQ$ is a list of each players deck after placing their cards face down and b is a list of the cards placed face down or played during the war.

```
WARmax:=proc(m,Q,r) local HandSize,HS, tempQ,i,
   output:
   HandSize:=[seq(nops(Q[i]),i=1..r)]:
   HS:=max(HandSize):
   if HS>m then output:=WARm(m,Q,r):
     elif HS=0 then output:=0:
     else output:=WARm(HS-1,Q,r):
     fi:
   output:
end:
```

3.1.3. *Defining Replacement.*

The following procedures define how cards are placed back into the deck after winning a round. Each procedure inputs a list Q, a positive

integer r where $|Q| = r$, a list b, and a positive integer N. The i-th entry of Q represents the i-th player's deck, b lists the cards on the table in the order they were played where Player i is the i-th player to play a card (i.e. b lists the cards that the winner of the round will take), and Player N is the winner of the round.

The procedure **RepPlayed(Q,r,b,N)** outputs a list of each players hand after adding the won cards to the bottom of Player N's deck in the order that they were played.

```
RepPlayed:=proc(Q,r,b,N) local c:
   c:=remove(t->t=0,b):
   [seq(Q[i],i=1..N-1),[op(Q[N]),op(c)],seq(Q[i],i
   =N+1..r)]:
 end:
```

The procedure **RepAllAsc(Q,r,b,N)** outputs a list of each players hand after sorting all the won cards in ascending order and then placing them at the bottom of Player N's deck.

```
RepAllAsc:=proc(Q,r,b,N) local c:
   c:=remove(t->t=0,b):
   c:=sort(c):
   [seq(Q[i],i=1..N-1),[op(Q[N]),op(c)],seq(Q[i],i
   =N+1..r)]:
end:
```

The procedure **RepAllDesc(Q,r,b,N)** outputs a list of each players hand after sorting all the won cards in descending order and then placing them at the bottom of Player N's deck.

```
RepAllDesc:=proc(Q,r,b,N) local c:
   c:=remove(t->t=0,b):
   c:=sort(b,`>`):
   [seq(Q[i],i=1..N-1),[op(Q[N]),op(c)],seq(Q[i],i
   =N+1..r)]:
end:
```

The procedure **RepAsc(Q,r,b,N)** outputs a list of each players hand after sorting each subround (i.e. each time all the players placed a card on the table) in ascending order and then placing them at the bottom of Player N's deck. In a war, the cards that each player placed face down first are sorted in ascending order, then then the card that each player put down next are sorted in ascending order, and so on.

```
RepAsc:=proc(Q,r,b,N) local n,c:
   n:=nops(b)/r:
```

```
3     c:=[seq(op(sort([op(i*r+1..i*r+r,b)])),i=0..n
      -1)]:
4     c:=remove(t->t=0,c):
5     [seq(Q[i],i=1..N-1),[op(Q[N]),op(c)],seq(Q[i],i
      =N+1..r)]:
6  end:
```

The procedure **RepDesc(Q,r,b,N)** outputs a list of each players hand after sorting each subround (i.e. each time all the players placed a card on the table) in descending order and then placing them at the bottom of Player N's deck.

```
1  RepDesc:=proc(Q,r,b,N) local n,c:
2     n:=nops(b)/r:
3     c:=[seq(op(sort([op(i*r+1..i*r+r,b)],`>`)),i
      =0..n-1)]:
4     c:=remove(t->t=0,c):
5     [seq(Q[i],i=1..N-1),[op(Q[N]),op(c)],seq(Q[i],i
      =N+1..r)]:
6  end:
```

3.2. **Simulating a Game of War.**

3.2.1. *Simulating One Move or Round.*

The procedure **OneMove(P,c,m,WAR,rep)** inputs a list P, a list c, a positive integer m, a war procedure WAR, and a replacement procedure rep. The i-th entry of P is a list representing the i-th player's hand at the beginning of the move, c lists the cards that have already been placed on the table during this round (i.e. cards accumulated during unresolved wars), and m is the default number of cards to be placed face down during a war. If this round ends the game with no winners it outputs $\{0\}$. If the round ends the game with a tie, then it outputs a set of the players that tied. Otherwise, it outputs a list Q where the i-th entry represents each players deck at the end of the round after adding the won cards to the bottom of the winner's deck.

OneMove(P,c,m,WAR,rep) adds 0 to any empty deck to represent no card, then creates a list a of the first card of each player's deck and a new list Q where the i-th entry of Q represents each player's deck after removing their first card. It finds the highest value in a and checks if any other player's had the same card. If the highest card was unique, it places all of the cards at the bottom of the winners deck according to rep. Otherwise, it runs WAR(m,Q,r) to simulate a war. If player's are able to play a card after placing cards face down on the table, it runs $OneMove(Q, b, m, WAR, rep)$ where Q and b are lists such

that the i-th entry of Q represents the i-th player's deck after placing cards on the table, and b lists the cards that have already been placed on the table during this round.

```
OneMove:=proc(P,c,m,WAR,rep) local r, a, b, tempP,
    i, h, w, Q, output, W:
   r:=nops(P):
   a:=[]:
   b:=[op(c)]:
   tempP:=P:
   for i from 1 to r do
       if P[i]=[] then tempP[i]:=[0]: fi: od:
   a:=[seq(tempP[i][1],i=1..r)]:
   Q:=[seq([op(2..-1,tempP[j])],j=1..r)]:
   h:=max(op(a)):
   w:={SearchAll(h,a)}:
   if nops(w)=1 then
       b:=[op(b),op(a)]:
       output:=rep(Q,r,b,op(w)):
   else
       W:=WAR(m,Q,r):
       if type(W,list)=true then
           Q:=W[1]:
           b:=[op(b),op(a),op(W[2])]:
         output:=OneMove(Q,b,m, WAR,rep):
         elif type(W,integer)=true then
             output:=tie:
         else
             output:=W:
         fi: fi:
   output:
   end:
```

3.2.2. *Simulating Games of War.*

The procedure **GenOneGameK(n,k,r,m,WAR,rep,K)** inputs positive integers n,k,r,K, a non-negative integer m, war procedure WAR, and replacement procedure rep. It simulates a game of war that is at most K rounds and has n different cards, k copies of each card, and r players. First it generates a random starting deck using $RandDeck(n, k, r)$. Then it runs $OneMove$ either K times or until the game finishes, whichever occurs first. If the game terminates, it outputs a list $[w, i]$ where w is the winner (or set of winners if there is a tie), and i is the number of rounds the game took. If there are no winners, $w = 0$. If the game does not terminate, the procedure fails.

```
GenOneGameK:=proc(n,k,r,m, WAR,rep,K) local P,
   HandSize, i, Q, w:
   P:=RanDeck(n,k,r):
   HandSize:=[seq(nopsP[i],i=1..r)]:


   for i from 1 to K while nops(remove(t->t=0,
   HandSize))>1 and type(P,list)=true do
       Q:=P:
       HandSize:=[seq(nops(P[i]),i=1..r)]:
       P:=OneMove(P,[],m,WAR,rep): od:


     if i=K+1 then RETURN(FAIL):


     elif nops(remove(t->t=0, HandSize))=1 then w
   :=max[index](HandSize):


     elif type(P,list)=false then w:=P


     else RETURN(FAIL): fi:
  [w,i-1]:
end:
```

The procedure **GenManyGamesK(n,k,r,m,WAR,rep,K,w,t,M)** inputs positive integers n,k,r,m,K,M, war procedure WAR, replacement procedure rep, and variables w and t. It simulates M random games of war that are at most K rounds and has n different cards, k copies of each card, and r players. It runs $GenOneGameK(n, k, r, m, WAR, rep, K)$ M times, keeping track of which games do not terminate in K moves and the lengths and winners of each terminating game. The procedure outputs a list $[W, T, nw, I]$ where W is a list whose i-th entry is the generating function for the duration of terminating games that player i won, T is a list whose i-th entry is the generating function for the duration of terminating games where player i tied with someone else, nw is the generating function for the duration of terminating games with no winners, and I is the number of games that did not end in K moves.

```
GenManyGamesK:=proc(n,k,r,m,WAR,rep,K,w,t,M) local
    f,h,g,i,G,j,nw:
```

```
    for i from 1 to r do
        f[i]:=0:   g[i]:=0: od:
    h:=0: nw:=0:


    for i from 1 to M do
        G:=GenOneGameK(n,k,r,m, WAR,rep,K):


    if G=FAIL then h:=h+1
      elif type(G[1],integer)=true then f[G[1]]:=f[
    G[1]]+w^G[2]
    else
      if nops(G[1])>1 then
          for j in G[1] do
                g[j]:=g[j]+t^G[2]: od:
        else nw:=nw+t^G[2]   fi:
    fi: od:


  [[seq(f[i],i=1..r)],[seq(g[i],i=1..r)],nw,h]:
end:
```

The procedure **GenManyGamesKEDIT(n,k,r,m,WAR,rep,K,w,t,M)** is an edited version of $GenManyGamesK(n,k,r,m,WAR,rep,K,w,t,M)$. It runs similarly to $GenManyGamesK(n,k,r,m,WAR,rep,K,w,t,M)$, but instead outputs a list $[all, win, nw, I]$ where all is the generating function for the duration of all the games that terminated, win is the generating function for the duration of terminating games with one winner, nw is the generating function for the duration of terminating games with no unique winner, and I is the number of games that did not end in K moves.

```
GenManyGamesKEDIT:=proc(n,k,r,m,WAR,rep,K,t,M)
    local a, f,h,g,i,G,j,nw:
    a:=0: f:=0: g:=0: h:=0: nw:=0:


    for i from 1 to M do
        G:=GenOneGame(n,k,r,m, WAR,rep,K):


    if G=FAIL then h:=h+1:
      elif type(G[1],integer)=true then
          a:=a+t^G[2]:
```

```
12            f:=f+t^G[2]:
13      else
14              a:=a+t^G[2]:
15              nw:=nw+t^G[2]:
16      fi: od:
17
18
19    [a,f,nw,h]:
20  end:
```

The procedure **FinishGame(H,r,m, WAR,rep)** inputs a list H, positive integer r such that H has length r, a non-negative integer m, a war procedure WAR, and a replacement procedure rep. It simulates the rest of the game of war where the i-th entry of H represents the i-th player's deck, m is the default number of cards placed face down during a war, WAR indicates which war procedure is being used, and rep indicates which replacement procedure is being used. The procedure terminates when the number of players with cards left is less than or equal to 1 or a periodic orbit has been found. It outputs a list of length 2, where the first entry says the winner(s) of the game and the second entry is how many rounds the game took if it terminated or the cycle found if it does not terminate.

FinishGame(H,r,m, WAR,rep) runs $OneMove$ until the game terminates or a periodic orbit is found, keeping track of how many rounds have occurred and each player's deck at the start of each round. It checks for periodic orbits every 10 rounds. A periodic orbit is found after $b > 0$ moves if $\exists a$ where $0 \leq a < b$ such that each player's deck after b moves is the same as their deck after a moves. If this is true, then the procedure finds the length of the cycle, L by counting the number of unique lists P of each player's deck occurring between moves a and b. It then finds the cycle by creating a list of length L where the i-th entry is the list of each players deck after $a + i - 1$ moves.

```
1  FinishGame:=proc(H,r,m, WAR,rep) local Q, P,
       HandSize, moves, cyc, path, N, cycle, lost,
       output, w:
2      P:=H:
3      HandSize:=[seq(nops(P[i]),i=1..r)]:
4      moves:=0:
5      cyc:=0:
6      path:=[P]:
7
8
```

```
    while nops(remove(t->t=0, HandSize))>1 and type
    (P,list)=true do
        P:=OneMove(P,[],m,WAR,rep):
        HandSize:=[seq(nops(P[i]),i=1..r)]:
        moves:=moves+1:


        if moves mod 10=0 then N:=Search(P,path):
            if N>0 then
                cyc:=nops({op(N..moves,path)}):
                cycle:=[op(N..N+cyc-1,path)]:
                HandSize:=[seq(nops(P[i]),i=1..r)]:
                lost:={SearchAll(0,HandSize)}:
                w:={seq(i,i=1..r)} minus lost:
              break:   fi: fi:
        path:=[op(path),P]:   od:


        if cyc>0 then
            output:=[w,cycle]:
        elif nops(remove(t->t=0, HandSize))=1 then
            output:=[max[index](HandSize),moves]:
        elif type(P,list)=false then
            output:=[P,moves]:
        else RETURN(FAIL): fi:


    output:
end:
```

3.2.3. *Finding information on Simple Games of war.*

Here, we define a simple game of war to be a game of war with two players and $2 * n$ unique cards. A simple game has the replacement procedure *RepAsc*, *RepDesc*, *RepAllAsc*, or *RepAllDesc*. In other words, player 2 will place the cards $[card1, card2]$ at the bottom of his deck in the same order as player 1 would. In the simple game, we can reduce the number of starting decks we need to look at in order to obtain information on all possible games of war.

The procedure **SimpGameInfo(n,rep)** inputs a positive integer n and a replacement procedure *rep*. It outputs a list [[f,C],cycles], where f is the generating function for the duration of terminating games, C is the number of games that lead to cycles, and *cycles* is the list of unique cycles.

First, **SimpGameInfo(n,rep)** reduces the number of starting decks it needs to run. There is no need to distinguish between Player 1 and Player 2 to figure out how all possible games will play out. Because the order that cards are replaced is the same for Player 1 and Player 2, running the simulation where player 1 starts of with the deck H[1] and player 2 starts with the deck H[2] will tell us everything we need to know about the game where player 1 starts with H[2] and player 2 starts with H[1]. Because the player with the (unique) highest card will never lose that card, there will never be a point in the game where player 1 and player 2 swap decks. So, running simulations where we look at the partitions of 2*n into two parts of size n where player 1 always starts with the highest card, we obtain information on every possible game. (Here, we find half of the unique cycles, but can easily find the other half by switching player 1 and player 2's decks.) Furthermore, we do not need to run simulations for the games where player 1's lowest card is higher than player 2's highest card, as these games obviously end in n moves.

Different hands can enter the same cycle at different points in the cycle, so we need to make sure that we do not count the same cycle multiple times. To do this, first note that if two cycles share any element, then the cycles are the same. Given the way that maple orders the elements in a set, by turning our cycle into a set and then back into a list, we can ensure that the first element is the same regardless of where the cycle starts. Note that this element will always have player 1's first card as 1 because at some point in any cycle player 1 must have 1 as their first card. **SimpGameInfo(n,rep)** creates a list of these elements and, instead of checking through the whole list to see if a cycle has already been found, it creates sub-lists defined by the second card in player 1's deck. So, if player 1's second card is b, the procedure only checks through the elements where player 1's second card is b.

```
1  SimpGameInfo:=proc(n,rep) local Decks, i,o,i1,j1,
     j2, counter, LedToCyc, cycles, C,N, f, H,G,a,b,
     k,M,N1,DrZ:
2   if rep<>RepAsc and rep<>RepDesc and rep<>
     RepAllAsc and rep<>RepAllDesc then
3        print(‘This is not a simple war game. Can’t
      use this procedure.‘):
4    else
5        Decks:=setpartition([seq(i,i=1..2*n)],n):
```

```
    #Right now, Player 1 always starts with 1. It is simpler if he starts with 2*n since he can never lose this card, so we need to turn all of the current 2*n's into 1's and all of the current 1's into 2*n's.
    Decks:=subs(2*n=DrZ,Decks):
    Decks:=subs(1=2*n,Decks):
    Decks:=subs(DrZ=1,Decks):
#It is obvious that if one player's highest card is lower than the other player's lowest card, then they lose the game in n rounds.
    Decks:=remove(t->t=[[2*n, seq(i,i=2*n-n+1..2*n-1)],[seq(i,i=2..2*n-n),1]],Decks):



    N1:=n!:
    N:=nops(Decks):
    counter:=0:
    LedToCyc:=0:
    cycles:=[]:
    C:=[seq([],i=1..2*n)]:
    f:=0:
    o:=permute(n):

    #Right now each players deck is in a specific order, so we need to look at the various ways to shuffle them.
  for i1 from 1 to N do
    for j1 from 1 to N1 do
        for j2 from 1 to N1 do
            H:=[Decks[i1][1][o[j1]],Decks[i1][2][o[j2]]]:
        if counter mod 50000 = 0 and counter>0 then
            print(`Played`, counter, `games`);
            print(nops(cycles), `distinct cycles`);
            print(LedToCyc, `got a cycle`);
                print(f):
          fi:


        counter:=counter+1:

```

```
39
40          G:=FinishGame(H,2,1, WARm,rep):
41          #Note: m and WAR don't matter since
    every card is unique.
42
43
44          if type(G[2],list)=true then
45                #We have a cycle!
46                LedToCyc:=LedToCyc+1:
47                a:=G[2]:
48                b:=[op({op(a)})][1]:
49                k:=b[1][2]:
50                M:=Search(b,C[k]):
51
52
53              if M=0 then
54                  C:=[seq(C[i],i=1..k-1), [op(C[k
    ]),b], seq(C[i],i=k+1..2*n)]:
55                cycles:=[op(cycles),a]:
56              fi:
57
58
59          elif type(G[2],integer)=true then
60                f:=f+t^G[2]:
61              fi:
62          od: od: od:
63       print(2*nops(cycles), 'distinct cycles'):
64       print(2*LedToCyc, 'got a cycle'):
65       print((2*n)!, 'total games'):
66    fi:
67
68
69          [[2*f+2*n!^2*t^n,2*LedToCyc],[op(cycles
    ),seq(cycles[i][2],cycles[i][1],i=1..-1)]]:
70    end:
```

The following separate **SimpGameInfo(n,rep)** into two procedures **SimpGameLengths** and **SimpPerDecks** in order to speed up running the procedures if information on only game lengths or only cycles is needed.

```
1  SimpPerDecks:=proc(n,rep) local Decks, i, counter,
       LedToCyc, cycles,o,i1,j1,j2, C,N, H,G,a,b,k,M,
   N1, DrZ:
```

```
2  if rep<>RepAsc and rep<>RepDesc and rep<>
     RepAllAsc and rep<>RepAllDesc then
3        print(‘This is not a simple war game. Can’t
      use this procedure.‘):
4    else
5        Decks:=setpartition([seq(i,i=1..2*n)],n):
6
7
8        #Right now, Player 1 always starts with 1.
    For later in the code, it is easier if he
    starts with 2*n since he can never lose this
    card.
9        #We need to turn all of the current 2*n’s
    into 1’s and all of the current 1’s into 2*n’s.
     Note, that right now the current 1’s are
    always
10       #the first element of P1’s deck
11       Decks:=subs(2*n=DrZ,Decks):
12       Decks:=subs(1=2*n,Decks):
13       Decks:=subs(DrZ=1,Decks):
14
15       #It is obvious that if one player’s highest
     card is lower than the other player’s lowest
    card, then they lose the game in n rounds.
16       Decks:=remove(t->t=[[2*n, seq(i,i=2*n-n
    +1..2*n-1)],[seq(i,i=2..2*n-n),1]],Decks):
17
18
19       N1:=n!:
20       N:=nops(Decks):
21       counter:=0:
22       LedToCyc:=0:
23       cycles:=[]:
24       C:=[seq([],i=1..2*n)]:
25       o:=permute(n):
26
27       #Right now each players deck is in a
    specific order, so we need to look at the
    various ways to shuffle them.
28     for i1 from 1 to N do
29       for j1 from 1 to N1 do
30           for j2 from 1 to N1 do
31               H:=[Decks[i1][1][o[j1]],Decks[i1
    ][2][o[j2]]]:
```

```
32
33          if counter mod 50000 = 0 and counter>0
    t h e n
34              print(‘Played‘, counter, ‘games‘);
35              print(nops(cycles), ‘distinct
    cycles‘);
36              print(LedToCyc, ‘got a cycle‘);
37            fi:
38
39
40          counter:=counter+1:
41
42
43          G:=FinishGame(H,2,1, WARm,rep):
44          #Note: m and WAR don’t matter since
    every card is unique.
45
46
47          if type(G[2],list)=true then
48                #We have a cycle!
49                LedToCyc:=LedToCyc+1:
50                a:=G[2]:
51                b:=[op({op(a)})][1]:
52                k:=b[1][2]:
53                M:=Search(b,C[k]):
54
55
56              if M=0 then
57                  C:=[seq(C[i],i=1..k-1), [op(C[k
    ]),b], seq(C[i],i=k+1..2*n)]:
58                  cycles:=[op(cycles),a]:
59                fi: fi:
60          od: od: od:
61      print(2*nops(cycles), ‘distinct cycles‘):
62      print(2*LedToCyc, ‘got a cycle‘):
63      print((2*n)!, ‘total games‘):
64  fi:
65
66
67            [op(cycles), seq(cycles[i][2],cycles[
    i][1],i=1..-1)]:
68  end:
```

```
1 SimpGameLengths:=proc(n,rep) local Decks, i,o,i1,
    j1,j2, counter, LedToCyc,N, f, H,G,N1,DrZ:
```

```
2  if rep<>RepAsc and rep<>RepDesc and rep<>
     RepAllAsc and rep<>RepAllDesc then
3        print(‘This is not a simple war game. Can’t
      use this procedure.‘):
4    else
5        Decks:=setpartition([seq(i,i=1..2*n)],n):
6
7
8
9
10       #Right now, Player 1 always starts with 1.
    It is easier if he starts with 2*n since he can
     never lose this card.
11       #We need to turn all of the current 2*n’s
    into 1’s and all of the current 1’s into 2*n’s.
     Note, that right now the current 1’s are
    always
12       #the first element of P1’s deck
13       Decks:=subs(2*n=DrZ,Decks):
14       Decks:=subs(1=2*n,Decks):
15       Decks:=subs(DrZ=1,Decks):
16
17       #It is obvious that if one player’s highest
     card is lower than the other player’s lowest
    card, then they lose the game in n rounds.
18       Decks:=remove(t->t=[[2*n, seq(i,i=2*n-n
    +1..2*n-1)],[seq(i,i=2..2*n-n),1]],Decks):
19
20
21       N1:=n!:
22       N:=nops(Decks):
23       counter:=0:
24       LedToCyc:=0:
25       f:=0:
26       o:=permute(n):
27
28       #Right now each players deck is in a
    specific order, so we need to look at the
    various ways to shuffle them.
29     for i1 from 1 to N do
30       for j1 from 1 to N1 do
31           for j2 from 1 to N1 do
32               H:=[Decks[i1][1][o[j1]],Decks[i1
    ][2][o[j2]]]:
```

```
33            if counter mod 50000 = 0 and counter>0
     t h e n
34                 print(‘Played‘, counter, ‘games‘);
35                 print(nops(cycles), ‘distinct
    cycles‘);
36                 print(LedToCyc, ‘got a cycle‘);
37                     print(f):
38               fi:
39
40
41            counter:=counter+1:
42
43
44            G:=FinishGame(H,2,1, WARm,rep):
45            #Note: m and WAR don’t matter since
    every card is unique.
46
47
48            if type(G[2],list)=true then
49                  #We have a cycle!
50                  LedToCyc:=LedToCyc+1:
51
52
53            elif type(G[2],integer)=true then
54                  f:=f+t^G[2]:
55                fi:
56            od: od: od:
57        print(2*LedToCyc, ‘got a cycle‘):
58        print((2*n)!, ‘total games‘):
59    fi:
60
61
62            [2*f+2*n!^2*t^n,2*LedToCyc]:
63    end:
```

3.3. **The War Graph: Procedures.** The procedures **WarGraph(n)** and **ArbRelWG(M)** (short for "arbitrary-relation war graph") use the GraphTheory package in Maple to construct the *game graph* of war. In WarGraph, the game is played with a simple n-card deck $\{1, 2, ..., n\}$; when players skirmish, the winner is always the player of the higher card, and the winner of the game (if there is any) is always the player who began the game with the high-card n. For this reason, we have "modded out" by the symmetry of exchanging the two players' hands, and allowed only the hands in which the left player

Leftie holds the card n. A position in War–a pair of players' hands $[[L_1, L_2, ..., L_m], [R_1, R_2, ..., R_\ell]]$–is called *terminal* if one player (for us, always the right player Rita) has an empty hand []. Apart from terminal positions, every position in simple War[3] has two "children" (two positions that could result from playing one Skirmish), and **ChildLW** and **ChildWL** record these two possibilities. The player who wins the skirmish must put the two cards at the bottom of their deck, and there are two possible orders in which to do this, LW (losing-winning) and WL (winning-losing). For example, here is ChildWL:

```
ChildWL:=proc(Hands):
if nops(Hands[1])>0 and nops(Hands[2])>0 then


if Hands[1][1] > Hands[2][1] then
[[op(Hands[1][2..]),Hands[1][1],Hands[2][1]],Hands
   [2][2..]]:
else
[Hands[1][2..], [op(Hands[2][2..]),Hands[2][1],
   Hands[1][1]]]:
fi:


else
Hands:
fi:
end:
```

The WarGraph is the directed graph in which each node is a position, and directed edges are introduced from each position to its possible children. It was first studied in Lakshtanov and Roshchina 2011. Here is the code:

```
WarGraph:=proc(n) local positions,perms,i,G,p,
   splits,H,spot,V,v,S,s,U:
perms:=permute(n):


positions:=[]:


for p in perms do
 spot:=Search(n,p):
```

[3]That is, war with two players on the linearly ordered deck $\{1, ..., n\}$

```
 splits:=seq(convert([p[..i],p[i+1..]],string),i=
   spot..n):
 positions:=[op(positions),splits]:
od:


G:=Digraph(positions):


for H in positions do
 H:=parse(H):


if nops(H[1]) = 0 or nops(H[2]) = 0 then
else
AddArc(G,{[convert(H,string),convert(ChildWL(H),
   string)],
[convert(H,string),convert(ChildLW(H),string)]}):
fi:
od:


G:
end:
```

See also the War Graph section.

ArbRelWG is a generalization of WarGraph, in which the winners of skirmishes is determined by the matrix M, rather than standard comparison. The matrix M consists of 0s and 1s, with the rule

$$M[i][j] = \begin{cases} 1 \text{ if card } i \text{ beats card } j \\ 0 \text{ else.} \end{cases}$$

For example, M could encode the rock-paper-scissors deck:

$$M = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \end{pmatrix}.$$

The procedures ArbRelChildLW and ArbRelChildWL are the analogs of ChildLW and ChildWL for this generalized version of War. The two major differences between ArbRelWG and WarGraph are: (1) there is now no notion of a "high card," so the entire set of $(n+1)!$ positions show up as vertices, and (2) winners are now determined by appeal to the matrix M, as you will see:

```
1  ArbRelWG:=proc(M) local n, positions,perms,i,G,p,
      splits,H,spot,V,v,S,s,U:
2  n:=nops(M):
3
4
5  perms:=permute(n):
6
7
8  positions:=[]:
9
10
11 for p in perms do
12 splits:=seq(convert([p[..i],p[i+1..]],string),i
      =0..n):
13 positions:=[op(positions),splits]:
14 od:
15
16
17 G:=Digraph(positions):
18
19
20 for H in positions do
21 H:=parse(H):
22
23
24 if nops(H[1]) = 0 or nops(H[2]) = 0 then
25 else
26  AddArc(G,{[convert(H,string),convert(
      ArbRelChildWL(H,M),string)],
27 [convert(H,string),convert(ArbRelChildLW(H,M),
      string)]}):
28 fi:
29 od:
30
31
32 G:
33 end:
```

4. The War Graph