APPENDIX P

PAIRING PROCEDURES GUIDELINES

This appendix is a technical aid to help clarify the pairing procedures. It in no way constitutes a set of rules.

P.0 The highest scoregroup constitutes the first bracket.

P.1 Determine bracket variables

1.1 Be **NP** the number of players in the bracket

1.2 Determine **M0** *(number of MDPs)* according to B.1.a
Set **Res** = NP - M0 *(Res: number of residents)*
Set **CMP** = min([NP/2], Res) *(CMP: Candidate Max Pairs)*

1.3 Set **Fls** = NP - 2 * CMP *(Fls: number of floaters)*
Set FFSList = empty *(FFSList: list of forbidden set of floaters, useful when Fls > 0)*

1.4 Set **CM1** = min(M0, CMP) *(CM1: Candidate M1)*

P.2 Find active criteria and set their minima (except CLB)

Note: minima are not set for the CLB. The reason is explained later (see 4.4.b.1.b).

2.1 **C6**

a C6 is active, if the bracket is heterogenous (i.e. M0 > 0)

b For details about the computation of minC6, see P.3.2

2.2 **C8**

a Be **CD2W** the number of topscorers (or possible opponents) with CD > +1
Be **CD2B** the number of topscorers (or possible opponents) with CD < -1
(CD is the colour difference of a player - see A.6)

b C.8 is active if in the bracket neither CD2W nor CD2B are equal to NP, and:
- there are at least a topscorer and another player all with CD>+1 *(or)*
- there are at least a topscorer and another player all with CD<-1

c minC8 is the number of topscorers (or possible opponents) whose |CD| cannot be less than 2.
It is equal to **max(0, max(CD2W, CD2B) - (NP - CMP))**

2.3 **C9**

a Be **WW** the number of topscorers (or possible opponents) who had White in the last two played rounds
Be **BB** the number of topscorers (or possible opponents) who had Black in the last two played rounds

b C9 is active if in the bracket neither WW nor BB are equal to NP, and:

- if BB>0, there are at least a topscorer and another player with an absolute preference for White *(or)*
- if WW>0, there are at least a topscorer and another player with an absolute preference for Black

c minC9 is the number of topscorers who cannot help but receive the same colour for the third consecutive round.
It is equal to **max(0, max(WW, BB) - (NP - CMP))**

2.4 **C10**

a Be **wS** the number of players who expect White
Be **bS** the number of players who expect Black

b C10 is inactive when any of the following conditions applies:

- all players expect the same colour *(NP=max(wS,bS))*
- at most one player per colour has a colour preference *(max(wS,bS) <=1)*
- there are no floaters *(Fls=0)* and at most one player has a colour preference different from all other players *(min(NP-wS,NP-bS)<=1)*

C10 may be inactive also in other conditions (e.g. if at least CMP+Fls players have the same absolute preference in rounds before the last one)

c minC10 (which is also called X) is the number of players who cannot receive the expected colour.
It is equal to **max(0, CMP - (NP - wS - bS) - min(wS, bS))**

2.5 **C11**

a Be **WS** the number of players who have a strong or absolute preference for White
Be **BS** the number of players who have a strong or absolute preference for Black

b C11 is inactive when any of the following conditions applies:

- all players have the same strong preference *(NP=max(WS,BS))*
- at most one player per colour has a strong or absolute colour preference *(max(WS,BS) <=1)*
- there are no floaters *(Fls=0)* and at most one player has a strong colour preference different from all other players *(min(NP-WS,NP-BS)<=1)*

c minC11 (which is also called Z) is the number of players whose strong preference cannot be fulfilled.
It is equal to **max(0, CMP - (NP - WS - BS) - min(WS, BS))**

2.6 **C12**

a Be **D1** the number of residents who received a downfloat in the previous round

b If 0 < D1 < Res, C12 is active when pairing the CLB or when Fls > 0 *(the bracket produces downfloaters)*

c In standard brackets, minC12 is the number of residents who cannot help but receive the same downfloat as the last round.
It is equal to **max(0, D1 - 2 * CMP + CM1)**
Reason: CM1+2(CMP-CM1) is the number of residents who play with MDPs and among themselves. If D1 is bigger than this number, some of the D1 residents will be forced to get a downfloat.*

2.7 **C13**

a Be **U1** the number of residents who received an upfloat in the previous round

b If 0 < U1 < Res, C13 is active when pairing the CLB or when CM1>0 *(i.e. heterogeneous brackets, where not all MDP(s) are in the Limbo)*

c minC13 is the number of residents who cannot help but receive the same upfloat as the last round.
It is equal to **max(0, U1 - Res + CM1)**
Reason: Res-CM1 is the number of residents who are non forced to meet a MDP. If U1 is bigger than this number, some of the U1 residents will be forced to play with a MDP

2.8 **C14**

a Be **D2** the number of residents who received a downfloat two rounds ago

b If 0 < D2 < Res, C14 is active when pairing the CLB or when Fls > 0

c minC14 is the number of residents who cannot avoid to receive the same downfloat as two rounds ago.
It is equal to **max(0, D2 - 2 * CMP + CM1)**

2.9 **C15**

a Be **U2** the number of residents who received an upfloat two rounds ago

b If 0 < U2 < Res, C13 is active when pairing the CLB or when CM1>0

c minC15 is the number of residents who cannot avoid to receive the same upfloat as two rounds ago.
It is equal to **max(0, U2 - Res + CM1)**

2.10 **C16**

a Be **R1** the number of players who received a downfloat in the previous round and have a higher score than the lowest ranked player in the bracket

b If R1 > 0, C16 is active when pairing the CLB or when M0>CM1

c In a standard bracket, the check-value of C16 is a sorted list *(like in PSD computation, see A.7)* of values given by the SD(s) of the Limbo elements who received a downfloat in the previous round (for the other Limbo elements who did not receive a downfloat in the previous round, the list value is 0).
In the CLB, as C16 considerations are made also in resident pairs (as the score of the two players may be unequal), all pairs are considered *(hence the list has a length of CMP+Fls)*, and non-zero values are set for pairs where the higher-ranked-player *(the one from S1)* has a higher score than his opponent and received a downfloat in the previous round.

d minC16 is a list with a zero value for each element.
Note: having initially all zeroes is not particularly accurate (e.g. if M0=3, CM1=2 and two of the MDPs got a downfloat in the previous round, there will be always a non-zero element in the list), but it would be quite complicate to introduce more precise rules.

2.11 **C17**

a C17 is active when U1 > 0 *(see P.2.7.a)* and the players of the bracket have at least three different scores

b The check-value of C17 is a sorted list of values given by the SD(s) of the games where the lower-ranked-player has a lower score than his opponent and has received an upfloat in the previous round.
Such list contains M1 elements in standard brackets and CMP elements in the CLB.

c minC17 is a list with a zero value for each element.

2.12 **C18**

a Be **R2** the number of players who received a downfloat two rounds ago and have a higher score than the lowest ranked player in the bracket

b If R2 > 0, C18 is active when pairing the CLB or when M0>CM1

c In a standard bracket, the check-value of C18 is a sorted list of values given by the SD(s) of the Limbo elements who received a downfloat two rounds ago (for the other Limbo elements who did not receive a downfloat two rounds ago, the list value is 0).
In the CLB, as C18 considerations are made also in resident pairs (as the score of the two players may be unequal), all pairs are considered *(hence the list has a length of CMP+Fls)*, and non-zero values are set for pairs where the higher-ranked-player *(the one from S1)* has a higher score than his opponent and received a downfloat two rounds ago.

d minC18 is a list with a zero value for each element.

2.13 **C19**

a C19 is active when U2 > 0 *(see P.2.9.a)* and the players have at least three different scores

b The check-value of C19 is a sorted list of values given by the SD(s) of the games where the lower-ranked-player has a lower score than his opponent and has received an upfloat two rounds ago.
Such list contains M1 elements in standard brackets and CMP elements in the CLB.

c minC19 is a list with a zero value for each element.

P.3 First pairing generation

3.0 If the bracket is the CLB then set **target**=<u>best</u>, otherwise set **target**=<u>perfect</u>
<u>target</u> *represents the kind of search that is performed:* ***perfect*** *means look for the perfect pairing;* ***best*** *means look for the best pairing (track the current best, called* ***champ****)*
Set **legal** = false
<u>legal</u> *becomes true as soon as the bracket produces its first legal pairing*

3.1 Generate the first candidate pairing (simply called **candidate**) for the bracket (see Section B).

3.2 If the bracket is not the CLB and C.6 is active, set minC6 equal to the PSD of the first candidate.

P.4 Pairing search

4.0 The first pair that creates "trouble" in a bracket is called pair-of-failure (POF).
If the "trouble" depends on a downfloater, the POF is the last pair *(i.e. the CMP-th pair)*.
The initial set is POF = CMP.

4.1 If there is a champ *(this is only possible if target is <u>best</u>)*

- a if PSD(candidate) > PSD(champ), the candidate is discarded and goto P.4.7 for a new candidate; otherwise set **online**=false
  *<u>**online**</u> represents a state of the candidate: when **false**, it means that the current failure values of the candidate (which may be worsening during the procedure) say that the candidate is (currently) better than the champ. When online is **true**, the candidate has at least the same failure values as the champ - as soon as a failure-value of the candidate becomes worse than the corresponding failure value of the champ, the candidate is discarded.*

4.2 Floater verification (if Fls > 0):

- a if the bracket is the PPB, verify floaters against the SCS ***(call CompletionCheck(floaters, SCS))***.
  If the verification fails, the candidate is discarded and goto P.4.7 for a new candidate; otherwise goto P.4.3
- b if the bracket is not the CLB, check whether the current set of floaters is included in the FFSList. If so, the candidate is discarded and goto P.4.7 for a new candidate.

4.3 If target is <u>perfect</u> and PSD(candidate) = minC6, set **probable**=true; otherwise set **probable**=false
*As long as <u>**probable**</u> is true, the candidate can be a perfect pairing.*

4.4 For each pair P of the candidate (numbered from 1 to CMP) and for each floater (P doesn't change while scrutinizing floaters, i.e. it is equal to CMP):

- a if the pair fails any absolute criterion, the candidate is discarded. If target is <u>perfect</u> and there is a champ, set POF=min(P, POF); otherwise set POF=P. Then goto P.4.7 for a new candidate
  POF=min(P,POF) eliminates all the candidates that have a failure in a pair that precedes the one with an absolute failure; such elimination can be applied when looking for a perfect pairing as long as a champ has already been set.
- b for each criterion (C8-C11,C13,C15,C17,C19 for pairs; C12,C14,C16,C18 for floaters), provided that is active (be C_i such criterion)
  - 1 If the pair or the floater fails C_i:
    - a increment failure counter F_i for criterion C_i
    - b if probable is true and $F_i > minC_i$, set probable = false
      Note: this is the only place where minima are used; hence, minima are used only when probable is true. As in the CLB probable is always false (see P4.0 and P4.3), minima are never used in the CLB.
    - c if target is <u>perfect</u>, set POF = min(P, POF)
    - d if there is a champ, set online to true if, for each criterion C_j that precedes C_i *(including C6)*, the candidate-F_j is equal to the champ-F_j. Otherwise, set online to false.
  - 2 If there is a champ and online is true:
    - a if the candidate-F_i is higher than the champ-F_i, the candidate is discarded, set POF = min(P, POF), and goto P.4.7 for a new candidate

b if the candidate-F_i is less than champ-F_i, set online = false

Note: if candidate-F_i is equal to champ-F_i and there is a candidate-F_k higher than a champ-F_k, with C_k following C_i, the candidate will be discarded when analyzing C_k (i.e. later in the process).

4.5 Set legal = true *(a legal pairing was found)*

4.6 a If probable is true *(assert: target = perfect)*, the candidate is the **Probable Pairing**. Go to 4.8 for the relevant checks

b If online is false, the candidate becomes the new champ

c If online is true *(which means that candidate and champ have exactly the same F_i failure values)*: the candidate is discarded *(as it is being generated later than the champ)*

4.7 Generation of a new pairing (a new candidate) using POF ***(call GetNextPairing(candidate, POF))***:

a if it was possible to generate a new pairing, restart from P.4

b *(assert: a new pairing was not generated, and, obviously no perfect pairing was found)*
if legal is true:

1 if target is perfect, set target=best and restart from P.3.1

2 if a champ exists, such champ is named the **Probable Pairing**: goto P.4.8 for the relevant checks

3 *(no champ exists, but legal being true means that FFSList is not empty)* bestP (see P.4.8.b.3) is the **Definitive Pairing**. Goto P.5 for the completion check

c *(assert: legal is false => no pairing whatsoever was generated)*
if M0 > 0 and M0-CM1 < Fls, set CM1 = CM1 - 1 and restart from P.2.
(assert: CMP > 0; with CMP=0, i.e. all float, legal cannot become false)
Otherwise, set CMP = CMP - 1 and restart from P.1.3.

4.8 If there are floaters in the Probable Pairing, check whether such set of floaters (called **FS**) maximizes the next bracket ***(call FloatersVerification(FS))***; if there are no floaters or the above check succeeded, the Probable Pairing is the **Definitive Pairing** and goto P.5 for the completion check.
Otherwise (*i.e., the FloatersVerification failed)*:

a if target is perfect *(i.e. the Probable Pairing is a candidate)*, the candidate is discarded and goto P.4.7 for a new candidate

b *(assert: target is best, i.e. the Probable Pairing is a champ)*

1 a pairing of the following bracket is returned as the result of the failed check. Be **fbPairs** the number of pairs of such pairing, and **fbPSD** its PSD.

2 add FS to the FFSList

3 if **bestP**, and consequently **nextPairs** and **nextPSD**, do not exist, or, if they exist and either fbPairs > nextPairs or fbPairs = nextPairs and fbPSD < nextPSD, set nextPairs to fbPairs, nextPSD to fbPSD and bestP to the Probable Pairing.

4 reset the champ *(i.e. from now on, a champ is no more existent)*, and restart from P.3.1

P.5 Completion Check

5.1 Unless the current bracket is the PPB:

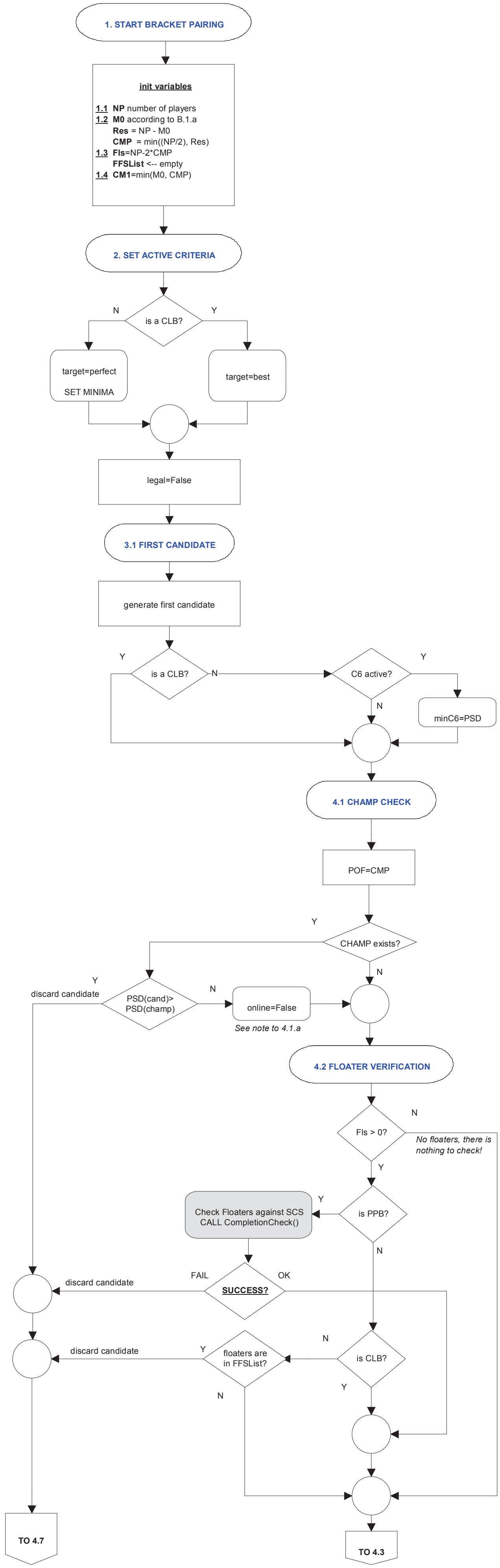
a with the floaters defined by the (perfect) pairing found in P.4.5 or by the (imperfect) pairing found in P.4.7.2.a (which may be none), and all the players coming from the successive scoregroups (**rest**), ***call CompletionCheck(floaters, rest)*** to verify whether it is possible to find a legal pairing in the combined bracket made of the floaters and the rest.

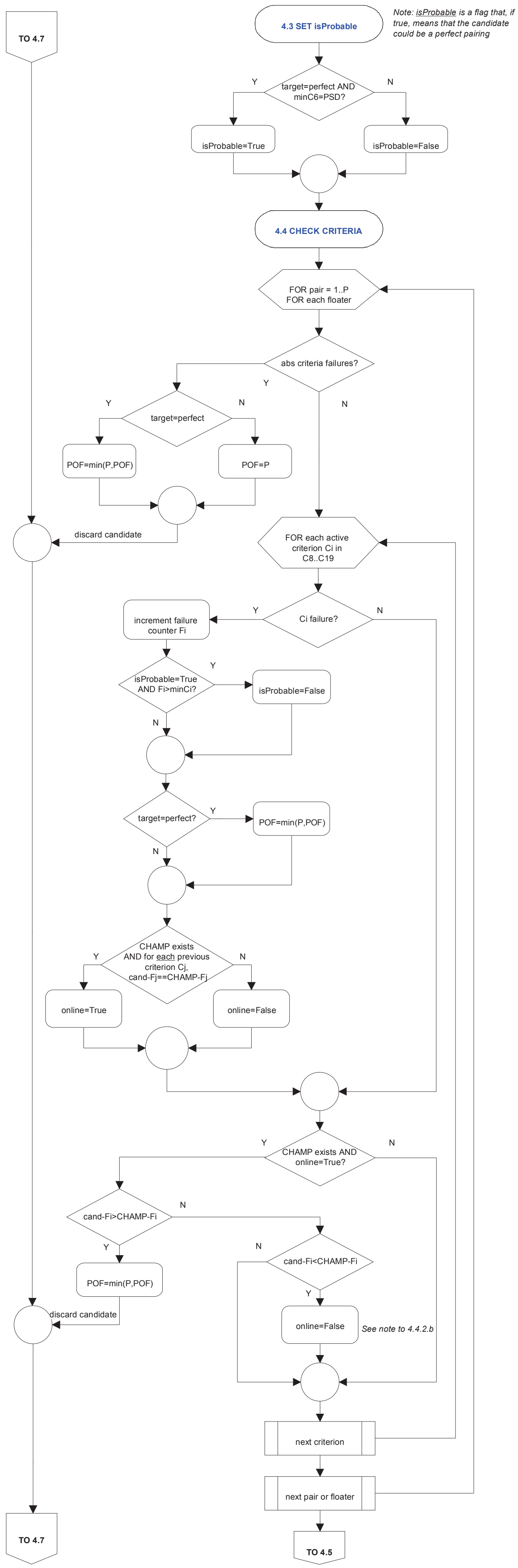
b If the completion-check is successful, the pairing is definitively approved. Continue with P.5.2.

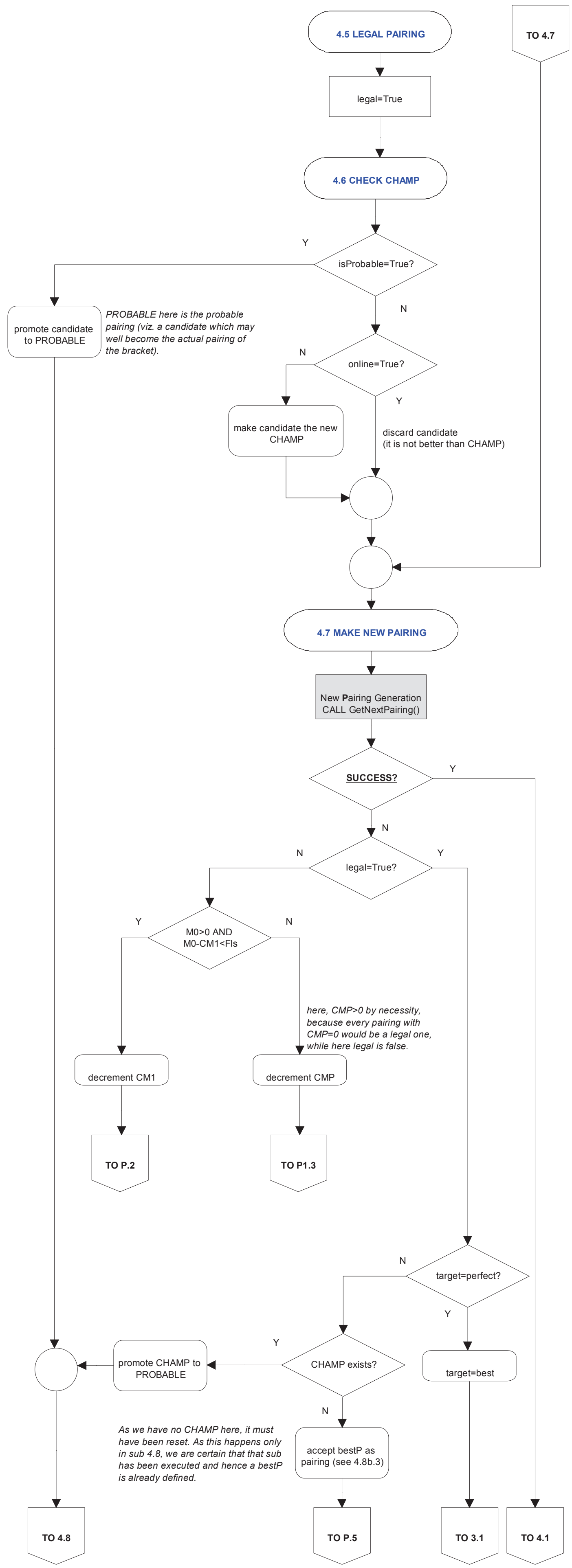
c If the completion-check fails, the current bracket is called PPB and the rest is called SCS.
The pairing process restarts from P.3.

5.2 If there is a successive scoregroup (which is the SCS if the current bracket is the PPB), the pairing process continues with the bracket composed of the floaters of the current pairing and the successive scoregroup. With this new bracket, restart from P.1.

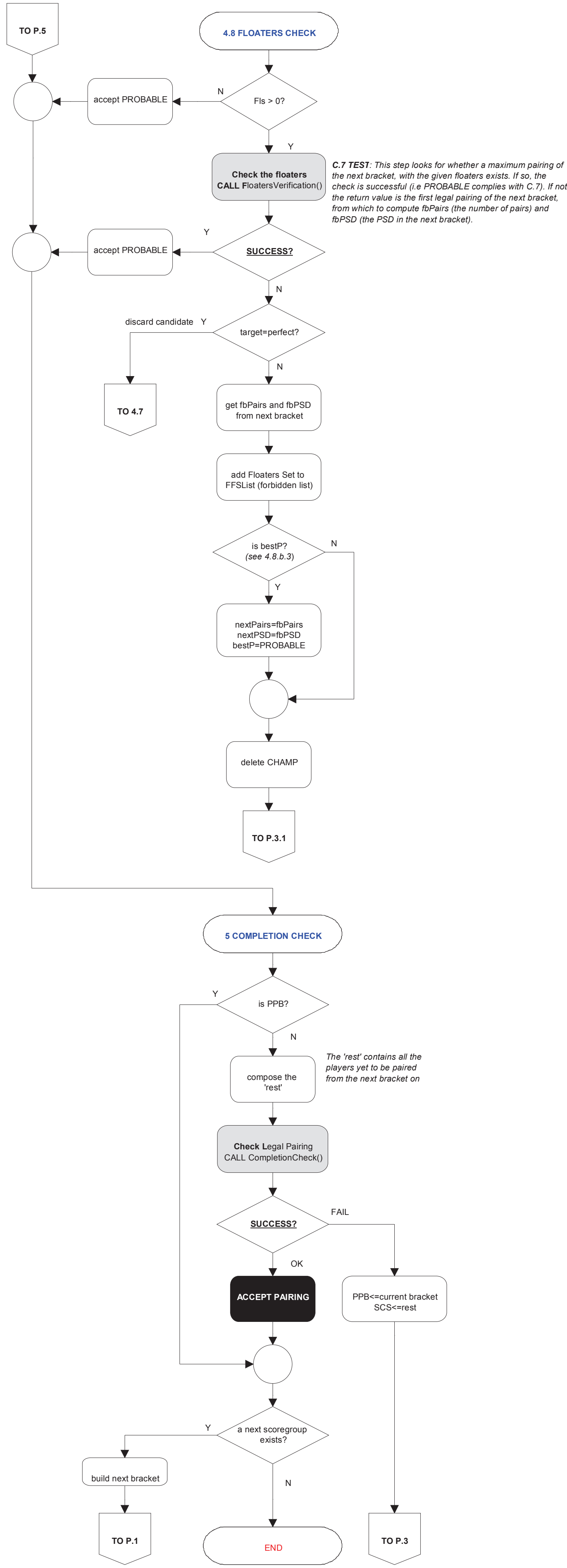
5.3 If there are no more scoregroups, the pairing process ends


1. START BRACKET PAIRING
init variables
1.1 NP number of players
1.2 M0 according to B.1.a
Res = NP - M0
CMP = min((NP/2), Res)
1.3 Fls=NP-2*CMP
FFSList <-- empty
1.4 CM1=min(M0, CMP)
2. SET ACTIVE CRITERIA
is a CLB?
N
Y
target=perfect
SET MINIMA
target=best
legal=False
3.1 FIRST CANDIDATE
generate first candidate
is a CLB?
Y
N
C6 active?
Y
N
minC6=PSD
4.1 CHAMP CHECK
POF=CMP
CHAMP exists?
Y
N
discard candidate
Y
PSD(cand)>
PSD(champ)
N
online=False
See note to 4.1.a
4.2 FLOATER VERIFICATION
Fls > 0?
N
No floaters, there is nothing to check!
Y
is PPB?
Y
Check Floaters against SCS
CALL CompletionCheck()
N
FAIL
OK
discard candidate
SUCCESS?
N
is CLB?
discard candidate
Y
floaters are
in FFSList?
N
Y
TO 4.7
TO 4.3


TO 4.7
4.3 SET isProbable
Note: isProbable is a flag that, if true, means that the candidate could be a perfect pairing
target=perfect AND minC6=PSD?
Y
N
isProbable=True
isProbable=False
4.4 CHECK CRITERIA
FOR pair = 1..P FOR each floater
abs criteria failures?
Y
N
target=perfect
POF=min(P,POF)
POF=P
discard candidate
FOR each active criterion Ci in C8..C19
Ci failure?
increment failure counter Fi
isProbable=True AND Fi>minCi?
isProbable=False
target=perfect?
POF=min(P,POF)
CHAMP exists AND for each previous criterion Cj, cand-Fj==CHAMP-Fj
online=True
online=False
CHAMP exists AND online=True?
cand-Fi>CHAMP-Fi
cand-Fi<CHAMP-Fi
POF=min(P,POF)
discard candidate
online=False
See note to 4.4.2.b
next criterion
next pair or floater
TO 4.7
TO 4.5



4.5 LEGAL PAIRING
TO 4.7
legal=True
4.6 CHECK CHAMP
isProbable=True?
Y
N
promote candidate to PROBABLE
PROBABLE here is the probable pairing (viz. a candidate which may well become the actual pairing of the bracket).
online=True?
N
Y
make candidate the new CHAMP
discard candidate (it is not better than CHAMP)
4.7 MAKE NEW PAIRING
New Pairing Generation CALL GetNextPairing()
SUCCESS?
Y
N
legal=True?
N
Y
M0>0 AND M0-CM1<FIs
Y
N
here, CMP>0 by necessity, because every pairing with CMP=0 would be a legal one, while here legal is false.
decrement CM1
decrement CMP
TO P.2
TO P1.3
target=perfect?
N
Y
CHAMP exists?
Y
N
promote CHAMP to PROBABLE
target=best
As we have no CHAMP here, it must have been reset. As this happens only in sub 4.8, we are certain that that sub has been executed and hence a bestP is already defined.
accept bestP as pairing (see 4.8b.3)
TO 4.8
TO P.5
TO 3.1
TO 4.1



TO P.5
4.8 FLOATERS CHECK
N
Fls > 0?
accept PROBABLE
Y
Check the floaters
CALL FloatersVerification()
C.7 TEST: This step looks for whether a maximum pairing of the next bracket, with the given floaters exists. If so, the check is successful (i.e PROBABLE complies with C.7). If not, the return value is the first legal pairing of the next bracket, from which to compute fbPairs (the number of pairs) and fbPSD (the PSD in the next bracket).
Y
SUCCESS?
accept PROBABLE
N
discard candidate Y
target=perfect?
TO 4.7
N
get fbPairs and fbPSD from next bracket
add Floaters Set to FFSList (forbidden list)
is bestP? (see 4.8.b.3)
N
Y
nextPairs=fbPairs
nextPSD=fbPSD
bestP=PROBABLE
delete CHAMP
TO P.3.1
5 COMPLETION CHECK
Y
is PPB?
N
compose the 'rest'
The 'rest' contains all the players yet to be paired from the next bracket on
Check Legal Pairing
CALL CompletionCheck()
SUCCESS?
FAIL
OK
ACCEPT PAIRING
PPB<=current bracket
SCS<=rest
Y
a next scoregroup exists?
build next bracket
N
TO P.1
END
TO P.3

Subroutine **<u>FloatersVerification(curDF, nextGroup)</u>**

The goal of this routine is to verify whether the set of downfloaters from the current bracket (represented by curDF) maximizes the number of pairs and minimizes the PSD of the following bracket (composed of such downfloaters and by the players in nextGroup).

curDF	*A list of BSN(s) representing the players that should downfloat from the current bracket.*
nextGroup	*A list of BSN(s), representing the players who are to be combined with the players identified by curDF in order to produce the next pairing bracket.*
Return	Either NULL (which means that the verification has been successful) or a legal pairing of the next pairing bracket, which permits to retrieve its number of pairs and its PSD.

<u>Overview</u>

The procedure described below is the most easy-to-explain way to see whether this routine reaches its goal (it may be optimized, though).
It basically is a simplified version of the general pairing algorithm. When the statement P.4.5 is reached (which means that a legal pairing has been found - something that, sooner or later, is going to happen, at worst, with a pairing made of all downfloaters), the key is whether the flow of the process went through P.4.7.c (where either the number of pairable MDP(s) or the number of possible pairs are reduced - both events causing a failure of the goal) or P.4.7.a (in case the newly generated pairing has a PSD higher than minC6).
If any of the above events happens, the legal pairing found in P.4.5 is not maximum, and the current candidate is returned, as such candidate represents the best (in terms of pairs and PSD) that the curDF downfloaters can provide.

V.1	Prepare a bracket, combining the curDF downfloaters with the nextGroup players		
V.2	Set ***<u>maximum</u>*** = True		
V.3	Be **NP** the number of players in the bracket Determine **M0** = #CurDF *(number of downfloaters)* Set **Res** = NP - M0 *(Res: number of residents)* Set **CMP** = min([NP/2], Res) *(CMP: Candidate Max Pairs)*		
V.4	Set **Fls** = NP - 2 * CMP *(Fls: number of floaters)* Set **CM1** = min(M0, CMP) *(CM1: Candidate M1)*		
V.5	Generate the first candidate pairing (simply called **<u>candidate</u>**) for the bracket (see section B).		
V.6	If *<u>maximum</u>* is True, set **minC6** equal to the PSD of <u>candidate</u>.		
V.7	For each pair P of the <u>candidate</u> *(numbered from 1 to CMP)*:		
	7.1	If the pair fails any absolute criterion, <u>candidate</u> is discarded. Set POF=P. Then goto V.9 for a new candidate	
V.8	If *<u>maximum</u>* is True, **return NULL**; else **return <u>candidate</u>**		
V.9	<u>Set candidate</u> = **GetNextPairing**(<u>candidate</u>, POF)		
	9.1	If <u>candidate</u> is not NULL:	
		a	If *<u>maximum</u>* is True and PSD(<u>candidate</u>) > minC6: set *<u>maximum</u>* = False
		b	Restart from V.7
	9.2	*(assert: <u>candidate</u> is NULL - no more available pairings)* Set *<u>maximum</u>* = False	
	9.3	If M0 > 0 and M0 - CM1 < Fls *(i.e. a MDP may float)*	
		a	Set CM1 = CM1 - 1
		b	Restart from V.7
	9.4	Set CMP = CMP - 1	
	9.5	Restart from V.4	

Subroutine **CompletionCheck(DFs, Rest)**

DFs — *A list (possibly empty) of players that should downfloat from the current bracket.*

Rest — *A list of all the players coming from all the scoregroups that follow the current bracket (in other words: all the players that are yet to be paired)*

Return — True, if the completion check is passed - False otherwise

Overview

This routine verifies whether the combined set of players coming from both DFs (if any) and Rest can produce a legal pairing. If the total number of players is odd, add a (fictitious) player called Virtual Bye.

The basic process consists in pairing two compatible players and see whether the remaining players can be paired among themselves. If they do, the verification is successful. If they don't, try with two other compatible players, until all possible combinations of compatible players have been exhausted (take into account that each player, who neither got a PAB nor won a scheduled game by forfeit, is a compatible opponent of the Virtual Bye).

A description of the main variables used by the routine:

- ***Verification*** — *is a container of lists of players to be paired; it is a dynamic container, in the sense that the number of elements it contains may vary and, in different moments, different lists may occupy the same slot*
- ***IndexVerification*** — *it is the istant counter of the number of lists contained in Verification*
- ***WorkingList*** — *represents a list of players to be paired*

O.0		Build **Main**, the combined list of players coming from DFs and Rest
O.1		*Add Main to the Verification container* Set Verification[1] <= Main indexVerification = 1
O.2		*Extract from Verification the latest list it contains and put it in WorkingList* Set WorkingList <= Verification[indexVerification] indexVerification = indexVerification - 1
O.3		*Take any player from WorkingList, for instance the first one* Set ***player*** = WorkingList[1]
O.4		Build **OL**, the list of players in the WorkingList who may face *player* in the current round
O.5		If OL is empty: *It means that the players in the WorkingList cannot be paired among themselves, as there is at least one player (e.g. player) that doesn't have an opponent. Try with another list, if one exists (i.e. when the Verification container is not empty), otherwise it means that no list can produce a pairing (which is a failure)*
	5.1	If indexVerification = 0 *(i.e. the Verification container is empty)*, **return False**
	5.2	Goto O.2 *(since at least one list still exists)*
O.6		*For each possible opponent of player create a new list without the player and his possible opponent (in other words, it simulates the two players have been paired)* For each element (be ***opponent***) of OL:
	8.1	*Exclude from WorkingList player and opponent* Set **newList** <= WorkingList - *player* - *opponent*
	8.2	If newList is empty *(i.e. two at a time, all players have been paired)*, **return True** *(the verification was successful)*
	8.3	*Add the new list to the Verification container* indexVerification = indexVerification + 1 Verification[indexVerification] = newList
O.7		Goto O.2 *(i.e. continue with the latest inserted list which, by construction, has two less players than WorkingList)*

Pairing Generation

The Pairing Generation is a collection of subroutines working together for the goal of producing a new candidate pairing or to inform the main process that it is not possible to generate new pairings for the bracket.

*Invoking the Pairing Generation (with two parameters: the current candidate and the pair-of-failure - POF, from now on) basically means invoking the driver of the aforementioned subroutines, the function **GetNextPairing**, which returns either a new pairing or NULL, if it is impossible to generate a new pairing.*

To process the GetNextPairing, a few other subroutines may be called to manage specific tasks:

Next	*to get the next meaningful transposition*
Exchange	*to perform an exchange in a homogeneous pairing (called by* Next*, when no meaningful transpositions is avaialable for a given POF)*
GenerateSequence	*to create a sequence of possible exchanges*
BuildLimboList	*to create a list of all possible Limbos*

The variables M0, NP, CMP and CM1 are used in the Pairing Generation subroutines. They are inherited from the main process, although NP, CMP and CM1 may also be computed from the input candidate pairing (in other words, only M0 is in an independent information).

*A pairing is made of an ordered list of pairs and a **set** (i.e. unordered) of floaters.*

From any pairing, it is always possible to retrieve two ordered lists of BSN(s). The first list (L1) contains the higher BSN of each pair (ordered following the order of the pairs). The second list (L2) contains the lower BSN of each pair (in the same order) followed by the floaters sorted by BSN. In any FIDE (Dutch) pairing, L1 contains CMP BSNs and L2 contains NP-CMP BSNs.

Note: It works also the other way: from two ordered list of BSN(s), the first one containing B1Size BSN(s), the second one containing B2Size BSN(s), with B2Size ≥ B1Size, it is possible to build a pairing made of B1Size pairs (the first element of B1 against the first element of B2; the second element of B1 against the second element of B2; and so on) and B2Size-B1Size floaters.
Such a resulting pairing is represented with the following symbolism: **B1 <=> B2**.

The goal of the Pairing Generation subroutines is to build a new pairing to be analyzed by the main algorithm. In order not to waste time in preparing useless pairings, the following criterion (COGUP) must be fully respected.

<u>Criterion for Optimizing the Generation of Useful Pairings</u> (COGUP)

A pairing is useful if, in each of its pairs, the element coming from L1 has a lower BSN than that of the element coming from L2.

For any pairing that contains a pair in which the L1-element has a BSN higher than that of the L2-element (i.e. a useless pairing), there is a correspondent useful pairing.
Any useful pairing is always generated before any of its correspondent useless pairings, because it has a lower number of exchanges.

Subroutine **GetNextPairing(currentPairing, POF)**

currentPairing *the last candidate that was analyzed by the main algorithm*

POF *(pair-of-failure) represents a pair of the candidate (hence it ranges from 1 to CMP) after the analysis found that its first POF pairs cannot produce a perfect pairing (they may have produced the best pairing, but this cannot be determined until all the meaningful pairings have been evaluated). Consequently, in the new pairing, at least one of the first POF pairs must be different from the first POF pairs of the candidate. Note that when a problem happens in the analysis of the floaters, the value of POF is CMP (changing the last pair of the pairing, will also change the floaters).*

Return Either a new meaningful pairing (*i.e. it complies with COGUP*) or NULL, if it is impossible to generate a new pairing.

Overview

The function behaves in different ways depending on whether it is applied on a homogeneous bracket, a quasi-homogeneous bracket (there are MDP(s), but they are not paired), a remainder of a heterogenous bracket or the MDP-pairing, with or without a Limbo.

G.1	If M0 = 0 *(i.e. the bracket is homogeneous)* or CM1 = 0 *(the bracket is quasi-homogeneous, i.e. all MDP(s) are in the Limbo, hence they are bound to float)*:	
	a.	Define: L1: list of the higher BSN(s) of each pair of currentPairing (in number of CMP) L2: list of the lower BSN(s) of the same pairs as above taken in that order, followed by each **resident** floater (taken in order of BSN) *(i.e. possible Limbo players -unpaired MDP(s)- are excluded from L2)*
	b.	Set *returnPairing* = **Next(L1, L2, POF)**
	c.	If *returnPairing* is NULL, return NULL
	d.	If M0 ≠ 0 *(i.e. there are Limbo players)*, add the unpaired MDP(s) to the list of *returnPairing* floaters and then sort all floaters by BSN
	e.	Return *returnPairing*
G.2	*(assert: M0 ≠ 0 and CM1 ≠ 0 => the bracket is truly heterogeneous; if M0 > CM1, there are Limbo players)* Define: Q1: list of the higher BSN(s) of the first CM1 pairs of currentPairing *(paired MDP(s) - they are in BSN order)* Q2: list of the lower BSN(s) of the first CM1 pairs of currentPairing, taken in that order *(upfloaters)* Q3: list of the higher BSN(s) of the last CMP - CM1 pairs of currentPairing, taken in that order Q4: list of the lower BSN(s) of the same pairs as above taken in that order, followed by each **resident** floater (taken in order of BSN) *The players in Q1 are the paired MDP(s); the players in Q2 are their opponents (upfloaters); hence Q1<=>Q2 is the MDP-Pairing.* *The players in Q3 and Q4 form the remainder; possible Limbo players (existent when M0 > CM1) are not included in any set.*	
G.3	if POF > CM1 *(the failure is in the remainder)*	
	a.	Set tempPairing = **Next(Q3, Q4, POF - CM1)**
	b.	If tempPairing = NULL, put POF = CM1 and goto G.4
	c.	Set *returnPairing* = *Q1<=>Q2 (which will net the same MDP-pairing as before)*
	d.	Attach tempPairing *(including the floaters)* to *returnPairing*
	e.	If M0 > CM1 *(i.e. there are Limbo players)*, add the unpaired MDP(s) to the list of *returnPairing* floaters and then sort all floaters by BSN

	f.		Return *returnPairing*
G.4			*(assert: POF <= CM1 => the failure is in the MDP-Pairing)* Set tempPairing = **Next(Q1, Q2::Q3::Q4, POF)**
G.5			If tempPairing is NULL:
	a.		If M0 = CM1 *(i.e. there is no Limbo)*, return NULL *(i.e. no more pairing exists for Q1)*
	b.		*(assert: there is a Limbo)* If LimboList is NULL:
		1.	Set LimboList = **buildLimboList()** *By construction LimboList[1] is the current Q1, i.e. the one made by the first CM1 elements*
		2.	LimboListSize = *<number of elements of LimboList>*
		3.	LLIndex = 1
	c.		If LLIndex = LimboListSize, return NULL *(all possible Limbos have been exhausted)*
	d.		LLIIndex = LLIndex + 1
	e.		Set NQ1 = LimboList[LLIndex] *(a new set of pairable MDP(s))*
	f.		Sort the Q2::Q3::Q4 players in BSN order (be QS such sorted list).
	g.		Set *returnPairing* = NQ1 <=> [the first CM1 elements of QS]
	h.		Set tempPairing = [elements of QS from the (CM1+1)th to the (CM1+CMP)th] <=> [elements of QS from the (CM1+CMP+1)th to the (CM1+2*CMP)th]
	i.		Attach tempPairing to *returnPairing*
	j.		Add to *returnPairing* as floaters: • the MDP(s) that are not in NQ1 *(and)* • the last (NP - CM1 - 2 *CMP) elements of QS
	k.		Return *returnPairing*
G.6			*(assert: tempPairing is not null; consider also that tempPairing is made of CM1 pairs and NP - 2 * CM1 floaters, where M0 - CM1 of them are unpaired MDP(s).* Create *returnPairing*, initialized with the pairs of tempPairing (i.e. no floaters)
G.7			Set addPairing = [the first CMP-CM1 floaters of tempPairing (taken in BSN order)] <=> [the next CMP-CM1 floaters of tempPairing (taken in BSN order)] The remaining NP - 2 * CMP floaters of tempPairing are the floaters of addPairing.
G.8			If M0 > CM1 *(i.e. there are Limbo players)*, add the unpaired MDP(s) to the list of addPairing floaters and then sort all floaters by BSN
G.9			Attach addPairing to *returnPairing*
G.10			Return *returnPairing*

Subroutine **<u>BuildLimboList()</u>**

This function is called only when a Limbo exists, i.e. when CM1 < M0. It may be called at most M0-1 times for each bracket, i.e. any time that CM1 is reduced and, with the initial Limbo (the one made of the last M0-CM1 MDP(s)), a perfect pairing could not be identified.
Note that unless it is the CLB (where all Limbos are always considered - but M0-CM1 is at most 1, so at most M0 different Limbos are possible), as soon as a legal pairing is found (in P4.5), CM1 is crystallized - and the same holds for the minimum PSD.
The unpaired MDPs (of the first legal pairing) define the structure of the Limbo. Any valid Limbo (from the one of the first legal pairing to the last one, the one made of the first M0-CM1 BSN(s)) must have the same number of elements (obviously) and the corresponding players must have the same scores of the players of the first valid Limbo.

Return A list of all possible Limbos *(actually what is returned is the list of the elements that are **not** in the Limbo).*
Note that the first element of the returned list contains the first CM1 BSN(s) of the bracket, which correspond to the paired MDP(s) belonging to the candidate that has been the last evaluated one in the main process.

<u>Overview</u>

The number of possible Limbos is given by the number of combinations of M0 elements, taken CM1 at a time, which is:

$$\frac{\mathbf{M0!}}{\mathbf{(M0\text{-}CM1)! * CM1!}}$$

Such Limbos are sorted following first the score difference (SD) of the MDP(s) in the Limbo (SD(s) are sorted from the highest to the lowest, then the lists are taken in lexicographic order), and then the lexicographic order of the BSN of the MDP(s) that are not part of the Limbo (i.e. the MDP(s) that are going to be paired)

L.1	Set <u>tempLimboList</u> <= empty	
L.2	For each possible set **<u>L</u>** of M0-CM1 BSN(s) not higher than M0: *Example: if M0 is 5 and CM1 is 3, the possible sets are (4,5) (3,5) (3,4) (2,5) (2,4) (2,3) (1,5) (1,4) (1,3) (1,2). The number of different sets is 5!/(2!*3!) = 120/(2*6) = 10.*	
	2.1	Compute <u>S0</u>, a list of BSN(s) not higher than M0 and not in <u>L</u>, sorted from the lowest to the highest
	2.2	For each player whose BSN is in <u>L</u>, compute the score; collect such scores in <u>Weight</u>, and sort them from the highest score to the lowest
	2.3	Add the t-uple <<u>L</u>, <u>Weight</u>, <u>S0</u>> to the <u>tempLimboList</u>
L.3	Sort the t-uples in <u>tempLimboList</u> by <u>Weight</u>, from the lowest to the highest *(<u>Weight</u>(s) are compared in lexicographic order)*. For equal <u>Weight</u>(s), sort the t-uples in order of <u>S0</u> from the lowest to the highest *(also <u>S0</u>(s) are compared in lexicographic order*)	
L.4	Once the sorting of the t-uples has been completed, gather the various S0 elements (in that order), and put them in the list <u>limboList</u>.	
L.5	Return <u>limboList</u>	

Subroutine **<u>Next(L1, L2, POF)</u>**

L1 *A list of BSN(s) (in number of **<u>L1Size</u>**); by construction, all the BSN(s) are either higher than M0 (homogeneous entity) or not higher than M0 (heterogeneous entity)*

L2 *A second list of BSN(s), in number of **<u>L2Size</u>**, with L2Size ≥ L1Size*

POF *A number not lower than 1 and not higher than <u>L1Size</u>; in the L1<=>L2 pairing, it represents the pair that should be changed (actually, at least one of the first POF pairs is to be changed)*

Return Either a new meaningful pairing (*i.e. it complies with the COGUP*), made of elements coming from L1 and L2, or NULL, if it is impossible to generate a new pairing.

<u>Overview</u>

When this function is called for the first time, L1<=>L2 is a candidate pairing or a part of a candidate (MDP-pairing or remainder). However, during the processing, the function may be recursively called many times and the initial condition does not hold anymore.

*Its immediate goal is to find a transposition of L2, be **<u>NL2</u>**, which follows L2 (from the POFth pair on). L1 will not change. If no more transpositions are available, apply an exchange (if the entity is homogeneous), by calling **Exchange**, which will define, if possible, a **<u>NL1</u>** different from L1 (and a consequent <u>NL2</u>); or return NULL.*

*When a new pairing is defined (either because a transposition was found or directly after an exchange), it is subject to the COGUP. If the COGUP is positive, the new pairing is returned, otherwise the **Next** function is recursively called with new input parameters.*

N.1	The first POF-1 BSN(s) of <u>NL2</u> (possibly none) are the first POF-1 BSN(s) of L2	
N.2	The POFth BSN of L2 is called the *<u>pivot</u>*	
N.3	Collect the BSN(s) of L2, from the *<u>pivot</u>* to the last one, in a set called <u>R2</u> *Hence, by construction, <u>R2</u> contains <u>L2Size</u>-POF elements - take notice that <u>R2</u> includes the <u>pivot</u>*	
N.4	Take the lowest BSN of <u>R2</u>, higher than the *<u>pivot</u>* (be <u>B2</u>). If there is none *(i.e. no useful transpositions are available from the current pivot)*, goto N.5.	
	4.1	<u>B2</u> is the POFth BSN of <u>NL2</u> *(this complies with the COGUP)*
	4.2	Sort the other BSN(s) of <u>R2</u> from the lowest to the highest. They constitute, in that order, the next <u>L2Size</u>-POF-1 BSN(s) of NL2.
	4.3	Set *<u>returnPairing</u>* = L1 <=> <u>NL2</u>
	4.4	Goto N.7
N.5	If POF > 1	
	5.1	Set *<u>returnPairing</u>* = ***<u>Next(L1, L2, POF - 1)</u>*** *(continue recursively the search of a useful transposition from the element before the current pivot)*
	5.2	Goto N.7
N.6	*(assert: POF = 1)*	
	6.1	If all BSN(s) of L1 are not higher than M0, **return NULL** *(in L1 there are only MDP(s), which, by definition, cannot be exchanged)*
	6.2	set *<u>returnPairing</u>* = ***<u>Exchange(L1, L2)</u>*** *(the POF is the first pair of the candidate, which means that all transpositions have been used up - hence, look for an useful exchange)*
	6.3	if *<u>returnPairing</u>* = NULL *(i.e. no more available exchanges)*, **return NULL**
N.7	Check the COGUP for *<u>returnPairing</u>* *(which has its own **<u>RL1</u>** and **<u>RL2</u>** lists)*	
	7.1	If the COGUP fails at the Fth pair, **return *<u>Next(RL1, RL2, F)</u>***
	7.2	If the COGUP is OK, **return** *<u>returnPairing</u>*

<u>Example of use of Next</u>
Be L1=[1,2,4,5,6] and L2=[8,3,9,12,11,(7 10)].

Two situations, with <u>POF=5</u> *(failure on 6-11 or on the floaters)* and with <u>POF=2</u> *(failure on 2-3)*

POF is 5	POF is 2
L2=[8,3,9,12,11,(7 10)] NL2=[8, 3, 9, 12 pivot = 11 R2 = {7, 10, 11} B2 does not exist ***<u>call Next(L1, L2, 4)</u>*** NL2=[8,3,9 pivot = 12 R2 = {7, 10, 11, 12} B2 does not exist ***<u>call Next(L1, L2, 3)</u>*** NL2=[8,3 pivot=9 R2={7, 9, 10, 11, 12} B2=10 **NL2=[8,3,10,7,9,(11,12)]** L1=[1,2,4,5,6] COGUP [5,6] vs [7,9] OK => **return 1-8 2-3 4-10 5-7 6-9 F={11,12}**	L2=[8,3,9,12,11,(7 10)] NL2=[8 pivot=3 R2={3, 7, 9, 10, 11, 12} B2=7 **NL2=[8,7,3,9,10 (11,12)]** L1=[1,2,4,5,6] COGUP: [4,5,6] vs [3,9,10] failure at 3 (third pair) ***<u>call Next(L1, NL2, 3)</u>*** NL2=[8,7 pivot=3 R2={3, 9, 10, 11, 12} B2=9 **NL2=[8,7, 9,3,10 (11,12)]** COGUP: [5,6] vs [3,10] failure at 3 (fourth pair) ***<u>call Next(L1, NL2, 4)</u>*** NL3=[8,7,9 pivot=3 R2={3, 10, 11, 12} B2=10 NL3=[8,7, 9, 10,3 (11,12)] COGUP: [6] vs [3] failure at 3 (fifth pair) ***<u>call Next(L1, NL3, 4)</u>*** NL4=[8,7,9,10 pivot=3 R2={3, 11, 12} B2=11 NL4=[8,7, 9, 10,11 (3,12)] COGUP: not needed **return 1-8 2-7 4-9 5-10 6-11 F={3,12}**

Subroutine **Exchange (L1, L2)**

L1Size => *number of elements in L1;* ***L2Size*** => *number of elements in L2;*
LN => *total number of elements (for the exchange), equal to L1Size + L2Size*
FSN =>*number of floaters, equal to L2Size - L1Size*
Premise: this routine works with numbers that go from 1 to LN. If there are holes in the union of L1 and L2 (L1::L2) -something that happens in remainders-, map the L1/L2 BSNs in a list of numbers from 1 to LN before proceding.

Example: if L1 contains [2 4 6] and L2 [8 5 9 7], the mapping is 2=>1, 4=>2, 5=>3, 6=>4, 7=>5, 8=>6, 9=>7; the remapped-L1 is [1 2 4] and the remapped-L2 is [6 3 7 5]

At the end of the procedure, before returning the pairing, remap the numbers used in the process to the original BSNs.

L1 *A list of L1Size BSN(s)*
L2 *A list of L2Size BSN(s)*
L1 and L2 have no common elements.

Return NULL, if no more exchanges are possible. Otherwise, return a pairing that, if expressed in the form NL1 <=> NL2, has NL1 different from L1.

Overview

Be **OS1** *the original S1, i.e. the (possibly remapped) BSN(s) from 1 to L1Size.*
Be **OS2** *the original S2, i.e. the (possibly remapped) BSN(s) from L1Size+1 to LN.*

Although it is possible to move from an exchange to the next one, trying to build a routine that does just that is not worth the while. It is a lot simpler to prepare a full list of the needed exchanges at the beginning (see below, though, for the extended meaning of "beginning") and then, each time that the procedure is invoked, take the next element from this list.
Hence, the first time the procedure is invoked, **GenerateSequence(1)** *is called to generate the sequence of exchanges of one BSN. It returns GS1, a sequence of G1 elements, each one of them composed of a t-uple of two BSNs, the first one is a BSN from OS1, the last one is a BSN from OS2 (the one-by-one BSNs to be exchanged). The G-counter, an index varying from 1 to G1, is set to 1, and the first invocation returns GS1[G].*
Each subsequent invocation of ***Exchange****, as long as the G-counter is less than G1, increments the G-counter of one and returns GS1[G]. When G=G1,* **GenerateSequence(2)** *is called to generate the sequence of exchanges of two BSNs. It returns GS2, a sequence of G2 elements, each one of them composed of a t-uple of four elements, the first two being BSNs from OS1, and the last two being BSNs from OS2 (the two-by-two BSNs to be exchanged). The G-counter, an index varying from 1 to* G2, *is set to 1, and GS2[G] is returned after the first invocation.*
An so on.

COGUP considerations say that the maximum number of exchanges, **Emax***, is given by L2Size/2 rounded downwards (whereas, with a higher number of exchanges, the COGUP will unavoidably fail).*
As a consequence, **GenerateSequence(Emax)** *is the last sequence that will be built. After exhausting all elements of the above sequence, the procedure must return a failure (NULL), meaning that no more exchanges are possible.*

Note: the G-counter (i.e. G), is a global variable, automatically initialized to 0.

X.0	Mapping phase *(needed when maximum(L1::L2) > LN)*	
X.1	If G ≠ 0 and G < GE, then G = G+1 and goto X.3	
X.2	*(assert: G = 0 or G = GE)*	
	2.1	Compute E, the number of elements of L1 with a higher BSN than L1Size *(those are elements of OS2, hence E represents the number of exchanges in the input pairing)*
	2.2	if E = Emax return **NULL** *(no more exchanges are possible)*
	2.3	*(assert E < Emax)* Set E=E+1, GSE=***GenerateSequence(E)***, G=1, GE=#GSE *(number of GSE elements)*
X.3	Be **set1** the first E elements of GSE[G] Be **set2** the last E elements of GSE[G] Set **NL1** <= OS1 - set1 + set2 Set **NL2** <= OS2 - set2 + set1	
X.4	Order the elements of both NL1 and NL2 according to C.04.3.A.2	
X.5	If a mapping was applied in X.0, map back the real BSN(s) in NL1 and NL2	
X.6	Return NL1 **<=>** NL2	

Subroutine **GenerateSequence (E)**

*This routine uses variables that were defined or computed inside the function **Exchange**: LN, FSN, Emax, OS1, OS2.*

E *A number between 1 and Emax*

Return A sequence of t-uples, each of them containing 2*E BSNs, representing the BSNs *(the first E from OS1, the last E from OS2)* that have to be exchanged.

Overview

*The routine is first executed to build the sequence of all the possible exchanges of one element. Then it is executed again when this sequence has been used up (by **Exchange**), to build the new sequence of all the possible exchanges of two elements - and so on until the sequence of exchanges of Emax elements.*

Q.1 Initialization phase

1.1 Sort all possible subsets of E BSN(s) of OS1 in decreasing lexicographic order to an array **S1LIST**, which may have **S1NLIST** elements.

COGUP considerations show that some subsets must be excluded because they are useless, particularly the ones which involve: all the first FSN+1 BSNs; FSN+2 of the first FSN+3 BSNs; FSN+3 of the first FSN+5 BSNs; FSN+4 of the first FSN+7 elements; and so on.

For instance, if the bracket has to produce a floater (FSN=1), and E=2, the subset with the first two BSNs of OS1 (i.e. 1 and 2) is to be excluded from S1LIST, because, if 1-2 were both moved to S2, one of them could float, but the other one would not find a S1-opponent who complies with COGUP.

1.2 Sort all possible subsets of E BSN(s) of OS2 in increasing lexicographic order to an array **S2LIST** which may have **S2NLIST** elements.

COGUP considerations show that some subsets must be excluded because they are useless, particularly the ones which involve: the last element; two of the last three elements; three of the last five elements; and so on.

For instance, with E=2, the subsets containing the last BSN of S2 (i.e. LN) or two of the last three BSNs (i.e. the subset of <LN-2, LN-1>, as the subsets <LN-2, LN> and <LN-1,LN> are already subsets containing LN) are to be excluded from S2LIST: if such elements were moved to S1, it would be impossible for at least one of them to find a S2-opponent who complies with COGUP.

1.3 Assign a difference to each possible exchange. It is a number defined as:

(Sum of BSN(s) from OS2, included in that exchange) - (Sum of BSN(s) from OS1, included in that exchange)

In functional terms:

DIFFERENZ(I,J) = sum of BSN(s) from OS2 in subset J - sum of BSN(s) from OS1 in subset I

This difference has:

a minimum: **DIFFMIN** = DIFFERENZ(1,1)

and a maximum: **DIFFMAX** = DIFFERENZ(S1NLIST, S2NLIST)

Q.2	**CNT**=0, **DELTA**=DIFFMIN
Q.3	**I**=1, **J**=1
Q.4	if DELTA = DIFFERENZ(I,J) then CNT=CNT+1, GSE[CNT]={I,J} *Note: {I,J} means the E BSN(s) from the subset I of S1LIST followed by the E BSN(s) from subset J of S2LIST.*
Q.5	if J < S2NLIST then J=J+1, goto Q.4
Q.6	if I < S1NLIST then I=I+1, J=1, goto Q.4
Q.7	DELTA = DELTA+1
Q.8	if DELTA <= DIFFMAX goto Q.3
Q.9	**return GSE**