The Math of Adjudication

by Lucas Kruijswijk

Introduction

Anyone who is introduced to the game of Diplomacy and has an interest in algorithms and discrete mathematics will not only be intrigued by this magnificent game, but also be attracted by the peculiarities of the rules in their own right.

Adjudicating a face to face game manually is not very difficult. Certainly, some experience is required: but it is a job that can be done fairly easily. Yet if we look at the history of automatic adjudication, we see that many adjudicators have been plagued with bugs and mistakes. Even in the forums of the more mature and respected adjudicator programs, one can find the occasional bug report.

Since I always had interest in algorithms and discrete mathematics, I was keenly interested in adjudication. Of course, when my initial attempts didn't work out, I started to look at what had already been written. I searched on the Internet and found the DPTG (Diplomacy Player's Technical Guide). The DPTG is a very complicated document, and I wondered whether it made things easier or more difficult.

Anyway, I was not happy with the DPTG. If the algorithm in the DPTG is the simplest flawless algorithm, then so be it. However, many of the sentences in the DPTG are clearly “derived knowledge”. They are constructed or deduced from more fundamental principles that the authors already know, but that may not be obvious to the casual reader. We should try to get the complete reasoning on paper, instead of only the conclusions.

If we look at mathematics, we see that the ancient Greeks had only 5 axioms for geometry (the five postulates from Euclid). Even the fifth postulate was heavily debated until the early 19th century. So if the DPTG describes the most fundamental axioms of adjudication, then this implies that adjudication in Diplomacy is far more complex than the entire field of geometry! Surely adjudication is a bit simpler than that...

So, I worked on my own ideas about adjudication, and finally I constructed my own algorithm. I wanted to build this in an existing program, but concluded that it needed systematic testing. I thought I could write the necessary test cases in a few evenings, but it turned out to be slightly more work than I expected. The result was the DATC (Diplomacy Adjudicator Test Cases, which was (and still is) used by several new Diplomacy programs. For a new adjudication program, the DATC significantly increases the quality of the first release.

Still, my real interests were in adjudication and in making the simplest flawless algorithm. In this article I will discuss all the issues about adjudication that I have explored over the

last 10 years. It contains the following chapters:

- Equations, the fundamentals of the rules
  In this chapter I will show how some approaches to adjudication won't work, and that the fundamental rules of Diplomacy are best represented with equations rather than a procedure.
- A simple recursive algorithm that works for most cases
  With the rules described as equations, one can construct a simple recursive algorithm that works in most cases other than cyclic dependencies.
- Decisions on partial information
  One way to deal with cyclic dependencies is to make decisions based on partial information. However, this makes programming the equations more complicated.
- Two proofs about the Diplomacy rules
  In this chapter we prove two theorems. First, that an order can only be part of one dependency cycle at a time; and second, that decisions based on partial information are enough to do the job.
- A simple recursive guess algorithm that works in all cases
  Finally, I give an algorithm that is correct and simpler than an algorithm based on partial information. This chapter includes a listing of a part of the algorithm.

NEXT


Lucas Kruijswijk
(L.B.Kruijswijk@inter.nl.net)

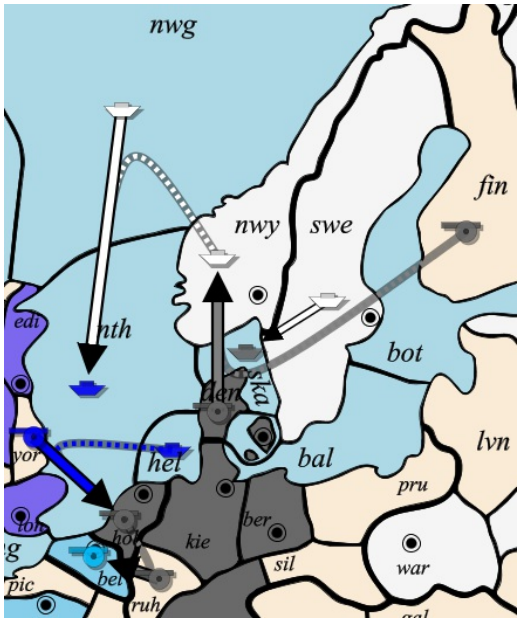
If you wish to e-mail feedback on this article to the author, and clicking on the envelope above does not work for you, feel free to use the "Dear DP..." mail interface.

The Math of Adjudication

by Lucas Kruijswijk

Equations, the fundamentals of the rules

When we try to explain to a person how to adjudicate a situation manually, we would probably teach him the simple rule of thumb “cut the support first”. In most common cases, supports can be adjudicated with this rule, which simplifies the situation. However, convoys can mess up the situation pretty badly, as shown in the following example:



Germany:
A ruh -> bel
A hol Supports ruh -> bel
A den -> nwy
F ska Convoys den -> nwy
A fin Supports den -> nwy

England:
A yor -> hol
F nth Convoys yor -> hol
F hel Supports yor -> hol

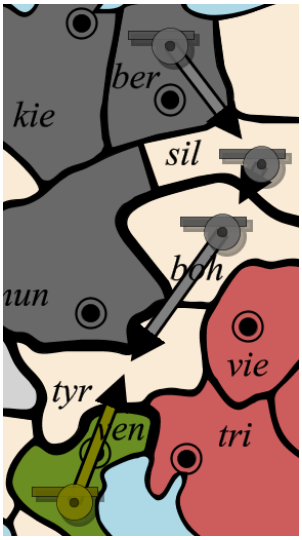
Russia:
F nwg -> nth
F nwy Supports nwg -> nth
F swe -> ska

Figure 1.

When determining whether the support of Army Holland will succeed, we have to know whether the convoy of Army Yorkshire is disrupted. To determine this, we have to adjudicate the move of the fleet in the Norwegian Sea. This move in turn depends on the support from Fleet Norway, which could be cut by the convoy from Denmark — but only if the fleet in Sweden fails to dislodge the fleet in Skagerrak.

We see that the rule of thumb “cut all supports first” does not always work. It is good guidance for manual adjudication, but its use for a more formal description of the rules or an algorithm is questionable.

Another notion, which can be found in many adjudicators, is the search for “hotspots”. Consider the following situation:



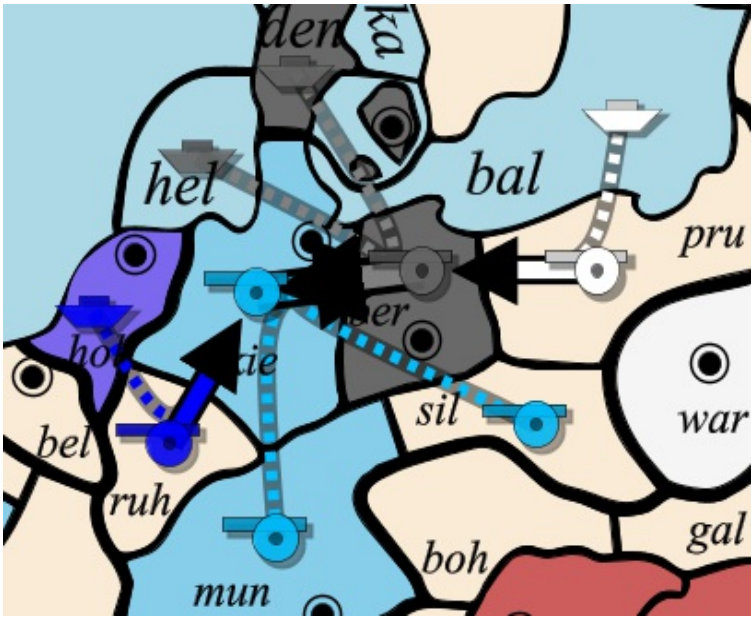
Germany:
A ber -> sil
A sil -> boh
A boh -> tyr

Italy:
A ven -> tyr

Figure 2.

The Berlin, Silesia and Bohemia orders are a chain of moves, where the success of one move depends on the success of another. The head of this chain is considered a hotspot which must be adjudicated first. So if we follow this principle, we start by resolving the conflict in Tyrolia. Since all units trying to move into Tyrolia have equal strength, both Army Venice and Army Bohemia fail to move. After resolving the Tyrolia area, the other orders in the chain can be adjudicated.

Although the idea of hotspots looks sound at first sight, things get less clear when a head-to-head battle is involved:



Germany:
A ber -> kie
F den Supports ber -> kie
F hel Supports ber -> kie

France:
A kie -> ber
A mun Supports kie -> ber
A sil Supports kie -> ber

England:
A ruh -> kie
F hol Supports ruh -> kie

Russia:
A pru -> ber
F bal Supports pru -> ber

Figure 3.

In the situation above, a head-to-head battle takes place between the armies in Kiel and Berlin. Both units attack with a strength of 3, while at the same time, they are stabbed from behind by a foreign unit with one support. Since the situation is symmetric, both Kiel and Berlin could be chosen as a hotspot. If one of the German supports were to be removed,

the hotspot would lean to Berlin, while if a French support were to be removed the hotspot would lean to Kiel.

Some programs adjudicate this situation incorrectly. They take one of the two areas as a hotspot, depending on how the orders show up in the internal administration. If Kiel is chosen as a hotspot, the result will be that the moves from Ruhr, Kiel and Berlin fail. After this resolution, the algorithm will continue with the other orders. Since the Kiel order has failed, the Russian order will be adjudicated with a positive (but incorrect) result. If instead Berlin is chosen as the hotspot, the army in Ruhr will advance (incorrectly).

Both the idea of cutting support first and the notion of hotspots are based on the idea that putting the orders in a sequence will help. The sequence in which the orders are adjudicated is a kind lifeline for the adjudicators based on this principle. For the most common situations, the sequence will indeed help; but for a truly flawless algorithm, things can still get pretty complicated (although not impossible). In these cases the sequence is not a lifeline that rescues by making things simple. On the contrary, it is more like a shark, something that bites and that we want to avoid.

To get things under control, we must go back to the fundamentals of the rules. If we take a look at the dislodge rule (page 10 of the 2000 rulebook) we see the following sentence:

> *Support is cut if the unit giving support is dislodged.*

This rule implies that the move that dislodges the supporting unit must be resolved first. Since the dislodgement can only succeed when the attacker gets its own support, that support must also be resolved in advance. Yet none of these sequence issues are listed by this rule, or in any other place in the rulebook. The rulebook merely gives conditions with which the resolution must comply, but does not give any hint of the sequence in which the orders must be resolved.

This means that the rules of Diplomacy are more like “equations”, while in most other games the rules are more procedural. The essential difference between the two approaches is that the equations cannot depend on time. An equation defines a relationship between variables that does not change. It is valid at the start of adjudication, during adjudication, and at the end of adjudication. In contrast, a procedure is executed step by step, and each step may change a variable affecting a later step.

This implies that when we construct an equation, obviously we can not make it dependent on time. For instance, the principle “a support is successful until it is cut” means that the resolution of a support changes over time. It may have a different value at the start of the adjudication than at the end. This might work for an algorithm, but is incompatible with equations. On the positive side, an equation may refer to something we don't know yet, while this is considered an error in a procedure.

Furthermore, the equation approach is “all or nothing”. We can not describe a part of the rules with equations and another part with a procedure. The rules are either wholly described as equations (the official rulebook), or wholly described as procedure (the DPTG approach). So the challenge is to transform the rulebook into a more formal description, while keeping the idea of equations (and yet avoiding any sequence). We will start with the simpler parts.

Prior to this transformation, we assume that orders are checked for geographical limitations, that unmatched orders are removed, and that for units moving to each territory, it is determined whether it is a swap or a head-to-head battle.

dislodged

A unit can only be dislodged when it stays in its current space. This is the case when the unit did not receive a move order, or if the unit was ordered to *move* but failed.

If so, the unit is dislodged if another unit has a move order attacking the unit and for which the *move* succeeds.

Note, that in the description, the word *move* is in *italics*. It refers to the *move* equation described further down.

convoy order

A fleet with a successful convoy order can be part of a convoy.

A convoy order is successful when the fleet receiving the order is not *dislodged*.

path

The path of a move order is successful when the origin and destination of the move order are adjacent, or when there is a chain of adjacent fleets from origin to destination each with a matching and successful *convoy order*.

support

A support order is cut when another unit is ordered to move to the area of the supporting unit and the following conditions are satisfied:

- The moving unit is of a different nationality
- The destination of the supported unit is not the area of the unit attacking the support
- The moving unit has a successful *path*

A support is also cut when it is *dislodged*.

Things get more complicated for the move. For a simple case, where a unit attacks another unit, we have to consider the *hold strength* of the defending unit (area) and the *attack strength* of the attacking unit.

hold strength

The hold strength is defined for an area, rather than for an order.

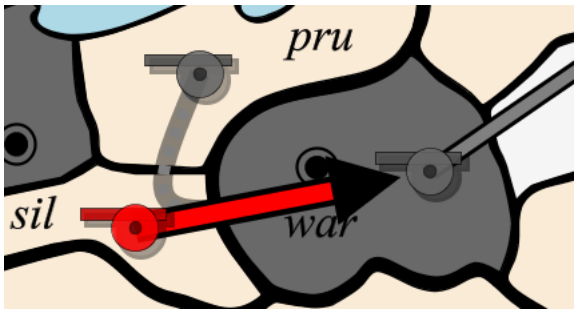
The hold strength is 0 when the area is empty, or when it contains a unit that is ordered to move and for which the *move* succeeds.

It is 1 when the area contains a unit that is ordered to move and for which the *move* fails.

In all other cases, it is 1 plus the number of orders that successfully *support* the unit to hold.

The *attack strength* (sometimes called the “dislodge strength”) is a little bit complicated because it may depend on whether the unit at the destination successfully moves away or

not. Consider the following situation:



Germany:
A war -> mos
A pru Supports sil -> war

Austria:
A sil -> war

Figure 4.

If the German unit in Warsaw successfully moves away, then the Austrian army in Silesia can use the full *attack strength* of 2 to capture Warsaw. However, if not, then the German army in Prussia will not help to dislodge its fellow army in Warsaw.

attack strength

If the *path* of the move order is not successful, then the attack strength is 0.

Otherwise, if the destination is empty, or in a case where there is no head-to-head battle and the unit at the destination has a move order for which the *move* is successful, then the attack strength is 1 plus the number of successful *support* orders.

If not and the unit at the destination is of the same nationality, then the attack strength is 0.

In all other cases, the attack strength is 1 plus the number of successful *support* orders of units that do not have the same nationality as the unit at the destination.

In cases where the unit is engaged in a head-to-head battle, the unit has to overcome the power of the move of the opposing unit instead of the *hold strength* of the area.

defend strength

The defend strength of a unit with a move order is 1 plus the number of successful *support* orders.

The final obstacle that an attacker has to defeat is the strength of the moves that compete for the same territory.

prevent strength

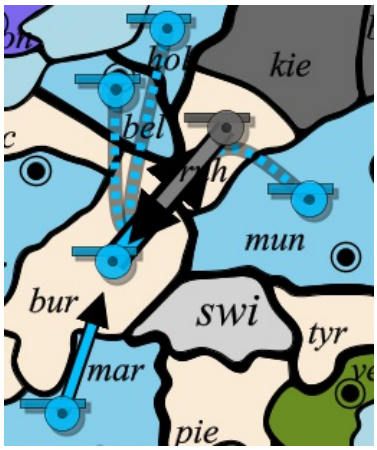
If the *path* of the move order is not successful, then the prevent strength is 0.

In cases where the move is part of a head-to-head battle and the *move* of the opposing unit is successful, then the prevent strength is 0.

In the remaining cases the prevent strength is 1 plus the number of successful *support* orders.

The *defend strength* and the *prevent strength* have a similarity in that they count the *support* regardless of the nationality of the support. Some adjudicators use the same function for calculating these strengths, and interpret it as the *defend strength* or the *prevent strength* depending on the stage of the algorithm. However, when using equations

we can not depend on the stage of an algorithm, and it is vital to distinguish between *defend strength* and *prevent strength*. The following example shows how *attack strength*, *defend strength*, and *prevent strength* can each be different for the same move:



Germany:
A ruh -> bur

France:
A bur -> ruh
A hol Supports bur -> ruh
A bel Supports bur -> ruh
A mun Supports ruh -> bur
A mar -> bur

Figure 5.

The German army in Ruhr may count the unusual support from Munich in its defense against the attack from Burgundy, so the *defend strength* of the Ruhr unit is 2. However, the unit in Munich will not help to dislodge its fellow army in Burgundy, so the *attack strength* of the unit in Ruhr is 1. Since it will be dislodged by the move from Burgundy, it can not prevent the unit in Marseilles from moving to Burgundy, so the *prevent strength* of the unit in Ruhr is just 0.

Finally we can give the conditions that allow a unit to actually move to another area:

move

In case of a head-to-head battle, the move succeeds when the *attack strength* is larger then the *defend strength* of the opposing unit and larger than the *prevent strength* of any unit moving to the same area. If one of the opposing strengths is equal or greater, then the move fails.

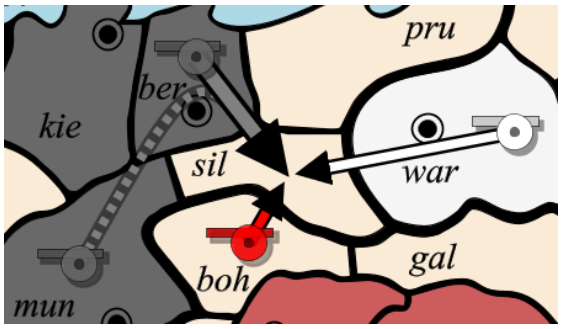
If there is no head-to-head battle, the move succeeds when the *attack strength* is larger then the *hold strength* of the destination and larger than the *prevent strength* of any unit moving to the same area. If one of the opposing strengths is equal or greater, then the move fails.

In case of circular movement or a convoy paradox, the equation may not have exactly one resolution. Sometimes the incompleteness theory of Gödel is suggested, but it is not that complicated. It is just that equations do not always have a single solution. For instance, $x^2 = 4$ has two real solutions, and $x^2 = -4$ has no real solution (for Gödel we should at least introduce an infinite map!).

In these situations we set aside the equations, and pass the troublesome part to what I call “the backup rule”. For circular movement all the move orders in the circle must succeed, and for convoy paradoxes a paradox rule should be used. Recently programmed adjudicators, such as jDip, DipTool, Palmpolitik, Paradox Interactive, phpDiplomacy (and its derivate Facebook diplomacy), Stabbeurfou, and Politics have opted for the Szykman rule for dealing with paradoxes. This rule says that the convoy part of the paradox must be treated as disrupted, and the other units are adjudicated normally (although originally it

was formulated differently).

Some conditions expressed in the equations have a one-to-one relation with the official English rulebook. The conditions that cut a support can literally be found in the rulebook. Other parts — for instance, the strength values — are expressed somewhat differently in the equations as compared to the rulebook. Consider the following situation:



Germany:
A ber -> sil
A mun Supports ber -> sil

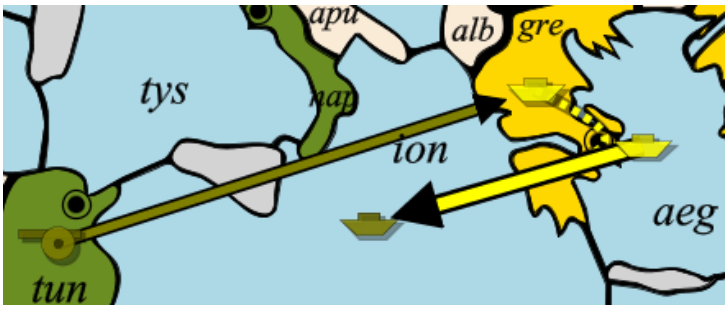
Russia:
A war -> sil

Austria:
A boh -> sil

Figure 6.

To a certain extent, the rulebook sees the Silesia area as a single conflict that must be adjudicated. And as explained earlier, some adjudicators see the Silesia area as a hotspot and look for the unit with the highest support (Berlin in this case). That unit succeeds, and the other units fail to move. The equations are different, in the sense that the conditions for success and failure are given for each of the moving units separately. The Berlin unit succeeds in its move because the *attack strength* is higher than the *prevent strength* of the other units. The other units fail because their *attack strength* does not beat the *prevent strength* of Berlin.

In the equations, the convoy order can succeed or fail. It is more common to talk about a whole convoy being disrupted instead of a single convoy order: however, considering the convoying fleet separately has its advantages when resolving a convoy paradox with the Szykman rule. Consider the following situation:



Turkey:
F aeg -> ion
F gre Supports F aeg -> ion

Italy:
A tun -> gre
F ion Convoys A tun -> gre

Figure 7.

The army in Tunis plays a major role in this convoy paradox. However, the resolution of the order is of no importance, because the army would always fail to move even if the attempt to cut support succeeded. One could argue (this will also be discussed later on in this article) that the resolution of the Tunis order is not part of the paradox. Applying the Szykman rule would let the Tunis move fail. This means that the backup rule has a consequence beyond the paradox. This is not directly problematic, but it is just easier to say that the convoy order of the Ionian Sea fails (and can not take part in a convoy route).

Adjudication of the other orders can then continue as normal.

Although the equations are not an exact one-to-one translation of the official English rulebook, they are close enough for discussing them properly. In this discussion we don't have the burden of the sequence in which the orders must be adjudicated and this keeps the complexity of checking the equations with the rulebook within proportions.

Still, the equations do not tell us**how** to adjudicate. In the next chapter we will take the first steps in transforming the equations into a usable algorithm.

PREV - NEXT

Lucas Kruijswijk
(L.B.Kruijswijk@inter.nl.net)

*If you wish to e-mail feedback on this article to the author, and clicking on the envelope above does not work for you, feel free to use the **"Dear DP..."** mail interface.*

The Math of Adjudication

by Lucas Kruijswijk

A simple recursive algorithm that works for most cases

The great thing about using the equations-based analysis is that it removes the risk of being tripped up by the interpretation of the rulebook while constructing an algorithm. The interpretation has to do with the definition of the equations, but any follow up is a matter of mathematics! So we have gained freedom in reasoning, and we can look at different approaches.

First of all we want to know whether an order is already resolved. This means that during adjudication, an order is in one of the following states: SUCCEEDS, FAILS, or UNRESOLVED.

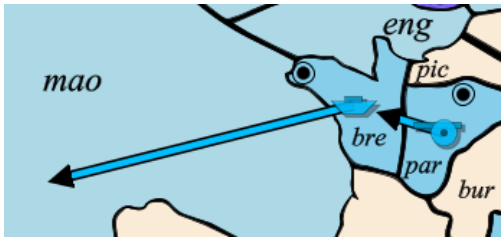
The UNRESOLVED state might not be used in algorithms that are not derived from equations (such as the DPTG), or at least not so explicitly. These algorithms process the orders in a specific sequence; and the sequence guarantees that when an order is processed, any prerequisites are resolved first. They often follow the principle that a move order succeeds until a reason is found for it to fail, and a support order succeeds until it is found to have been cut. This approach requires the construction of a correct sequence. However, in the choice of our algorithm, we don't want to be restricted in this way.

With the UNRESOLVED state, an obvious approach for an algorithm is to make a simple recursive function for resolving an order:

> *If the state of the order is not UNRESOLVED, then return.*
>
> *If the order is UNRESOLVED, look if the resolution of the order is dependent on the resolution of other orders.*
>
> *If so, resolve those orders first, by calling the function in recursion.*
>
> *When all dependencies are resolved, adjudicate the original order given the equations.*
>
> *Set the state of the order according to the adjudication result (so it is no longer UNRESOLVED).*

The complete algorithm initializes the states of all orders to UNRESOLVED, and then calls the function for every order.

For example, suppose we have the following simple situation:



France:
A par -> bre
F bre -> mao

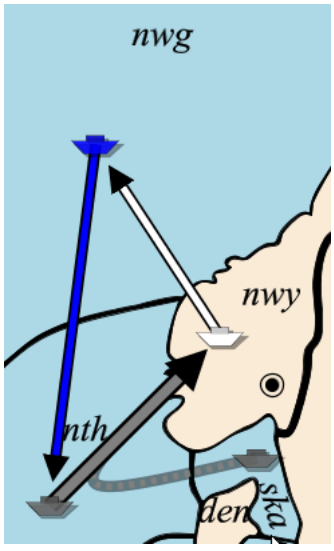
Figure 8.

Initially both orders are assigned the UNRESOLVED state. If we start resolving the Paris order, then we need to know the *attack strength* of the unit in Paris and the*hold strength* of the Brest area. For this, we need to know whether the *move* of the Brest unit is successful. So we try to resolve the Brest order in recursion. Since this order is not dependent on any other orders, it can be adjudicated directly, with a positive result. After this, the adjudication of the Paris order can be completed.

This simple recursive algorithm using the UNRESOLVED state works fine for most situations. However, things get complicated when dealing with a circular movement or convoy paradox. In these cases, the described simple recursive algorithm will enter an endless recursive loop. This means that the algorithm cannot handle all situations.

Actually, the circular movement and convoy paradoxes are not the real problem. If those situations were the only difficulty, then the simple recursive algorithm could easily be adapted, such that it recognizes when it enters an endless loop (that is, when in recursion the recursive function is called for the second time for the same order). In such cases, the orders in the cyclic dependency should be passed to the backup procedure, which determines whether it is a circular dependency or a convoy paradox and adjudicates accordingly.

However, it is not that easy. The really problematic situations are those that have a cyclic dependency, but for which the number of resolutions that respect the equations is nevertheless exactly one. These situations always consist of a circular movement or elements of a convoy paradox, supplemented with additional units and orders. For instance, consider a circular movement with one of the units supported, as shown below:



England:
F nwg -> nth

Germany:
F ska Supports F nth -> nwy
F nth -> nwy

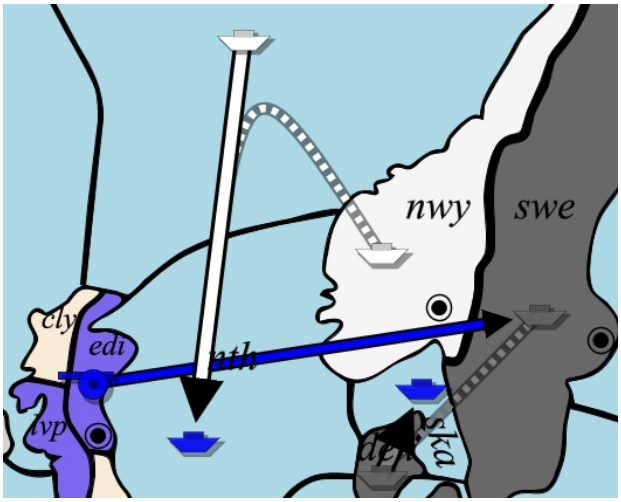
Russia:
F nwy -> nwg

Figure 9.

Normally, a circular movement will have two valid resolutions that respect the equations (if the circular movement rule is not considered). Either all units fail to move, or all units succeed in moving. However, in the situation above, a resolution where all units fail to move is incorrect, because the German fleet in the North Sea should always succeed in its move. (thanks to the support from the fleet in Skagerrak).

Similarly, if the fleet in Skagerrak were to move to the North Sea, all units would bounce. A resolution with all moving fleets advancing would be inconsistent.

In the following example, there would be a convoy paradox if the Russians did not intervene:



England:
A edi -> swe
F nth Convoys edi -> swe
F ska Convoys edi -> swe

Germany:
F den -> ska
F swe Supports den -> ska

Russia:
F nwg -> nth
F nwy Supports nwg -> nth

Figure 10.

Without the Russian orders, there are two resolutions that respect the equations. The first will advance the fleet in Denmark and disrupt the convoy as consequence, while the second is a successful convoy that cuts the support in Sweden. But with the Russian orders this all becomes irrelevant, because the Russian orders disrupt the convoy in any case.

Other ways for Russia to take away the paradoxical aspect of the scene are to support the fleet in Denmark, or to cut the support in Sweden (both can be done with the fleet in Norway). An alternative convoy path could also be ordered, but the geography in this particular example does not allow it.

The feature that all these situations have in common is that the equations contain a cyclic dependency, while the number of resolutions for the equations is still exactly one. And because there is only one resolution, we want to find this resolution. Passing the situation to the backup rule would give the wrong results. These situations jeopardize the simple recursive algorithm.

One possibility for dealing with these cyclic dependencies is to try to recognize the different situations, and adjudicate those special cases separately. This has been done in many older adjudicators, and since the possibilities are limited this is certainly possible.

For a circular movement situation this does not sound very difficult. The only thing that has to be done is to check whether there is unit outside the circle that disrupts the circular

movement with enough support. If so, the bouncing unit part of the circular movement fails, and all the other orders are adjudicated normally. If there is no unit that disrupts the circular movement, then all units advance, and units that support a movement within the circle are irrelevant. Although this sounds simple, remember that the additional unit may come by convoy, or that one of the crucial supports may be cut by a convoy.

It is far more complex to recognize the convoys correctly. TheDPTG has described this approach in detail, and uses the notion of subverted convoys, with subcategories: futile convoys, indomitable convoys, and confused convoys.

To avoid this complexity, we can look for a generic method of handling cyclic dependencies. A generic method may sound more difficult, but there are only a few requirements for such a generic algorithm:

- It must handle the equations and their dependencies.
- If there is a cyclic dependency with one exclusive resolution, it must choose that single resolution.
- If there is cyclic dependency which does not have one exclusive resolution, it must pass the situation to the backup rule.

This doesn't look impossible. In the next chapter we will look at one such approach, and in the last chapter we will look at an approach that is close to the simple recursive algorithm.

PREV - NEXT

Lucas Kruijswijk
(L.B.Kruijswijk@inter.nl.net)

*If you wish to e-mail feedback on this article to the author, and clicking on the envelope above does not work for you, feel free to use the **"Dear DP..."** mail interface.*

The Math of Adjudication

by Lucas Kruijswijk

Decisions on partial information

One might have noted that in the examples of the cyclic dependencies with one exclusive resolution, there is always at least one order within the cycle that can be adjudicated without the resolution of the other orders.

For instance, in figure 9 Germany orders its fleet in the North Sea to Norway and gives it support from Skagerrak. This move will succeed, regardless of whether the Russian fleet in Norway succeeds in its move or not. Similarly, in figure 10, the support of Sweden will never be cut, because the convoy will always be disrupted. The success of the Denmark move is irrelevant for the failure of the convoy. It appears then, that in case of a cyclic dependency there is always at least one order in the situation that can be resolved independently.

This principle can be handled more formally. Referring again to figure 9, even without resolving the move of the Russian fleet in Norway, the *hold strength* of the Norway area might be 0, and will not exceed 1 in any event. The successful support from the fleet in Skagerrak makes the *attack strength* of the fleet in the North Sea at least 2. So, although we do not know the *hold strength* precisely, we do know that the *attack strength* of 2, will always beat it.

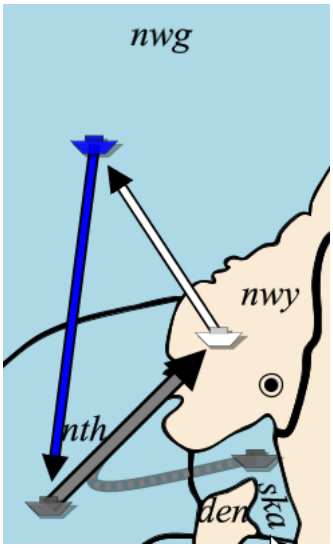
This kind of reasoning can be put in an algorithm. To begin, just as for the simple recursive algorithm, we have a UNRESOLVED state for each order, and all orders start in that state. For all the strength values (*hold strength*, *attack strength*, *defend strength* and, *prevent strength*), we write code that calculates the minimum value and maximum value of the strength given the current state of the orders. In figure 9, the *hold strength* of the Norway area has a minimum of 0 and a maximum of 1 while the order of the Russian fleet is still UNRESOLVED. If the support of the fleet in Skagerrak is still UNRESOLVED, then the *attack strength* of the fleet in the North Sea has a minimum of 1 and a maximum of 2. After resolving the *support*, the minimum also becomes 2.

A move order will get the SUCCEEDS state if the minimum *attack strength* of the unit beats the maximum resisting strengths of any opposing units. For getting the FAILS state, the maximum *attack strength* of the unit should not exceed the minimum strengths of the opposing units. If the success or failure of an order still cannot be determined given the information, then the order remains UNRESOLVED. All the equations can be worked out this way.

With this principle, the algorithm remains quite simple. Just start at the top of the order list, and try to resolve each order. Some orders might be resolved right away, while others may

stay unresolved. When we are at the bottom of the list, and there are still orders unresolved, we just start a second pass and continue to do so until all orders are resolved. When there are still orders remaining to be resolved, but a pass does not resolve any of them, then there is circular movement or a convoy paradox. The situation (which may consist of multiple independent circular movements or convoy paradoxes) must be passed to a backup rule.

Suppose the algorithm runs on the situation of figure 9:



England:
F nwg -> nth

Germany:
F ska Supports F nth -> nwy
F nth -> nwy

Russia:
F nwy -> nwg

Figure 9 repeated.

If we start at the top, then the order for the English fleet will fail to resolve (the *hold strength* of the North Sea is 0 or 1). The second order, the *support* from Skagerrak, will positively resolve. So, does the third order, but the Russian order will remain unresolved.

In the second pass, we know that the *hold strength* of the North Sea is at least 0 and at most 0. From this information, the fleet in the Norwegian Sea can advance. Finally, at the bottom of the list, the Russian order will be resolved too.

The algorithm can be optimized by resolving any dependency on other orders directly in recursion. Some code must be added to the recursive function to prevent an endless loop in case of cyclic dependency. If the function fails to resolve the order, that additional code can also be used to determine the orders which make up the cyclic dependency.

The principles of this algorithm are used in the adjudicators of Palmpolitik, jDip and Stabbeurfou. Together with the test cases of the DATC, they make it possible to create a high quality adjudicator on the first release (although a truly flawless adjudicator on first release is still a challenge). phpDiplomacy too switched to such an algorithm after its initial release.

Although the algorithm looks good and has proven its usefulness in real programs, two questions remain. First of all, is the algorithm indeed flawless? I assumed that decisions based on partial information can always find the single resolution in case of a cyclic dependency with a single resolution. The examples are evidence that this is indeed the case, but they are not a conclusive proof. In the next chapter I will give this proof.

The second question is whether this is the simplest algorithm. Decisions based on partial information require that each equation be programmed twice. For each strength value we must write code that calculates the minimum, and code that calculates the maximum. For the other equations, it should be realized that if a result is not positive, it is not necessarily negative, because the situation may remain unresolved. This means that separate code must be written to resolve a situation into a positive conclusion, and into a negative conclusion. In the final chapter I will show that there is an alternative, without this double coding.

PREV - NEXT

Lucas Kruijswijk
(L.B.Kruijswijk@inter.nl.net)

*If you wish to e-mail feedback on this article to the author, and clicking on the envelope above does not work for you, feel free to use the **"Dear DP..."** mail interface.*

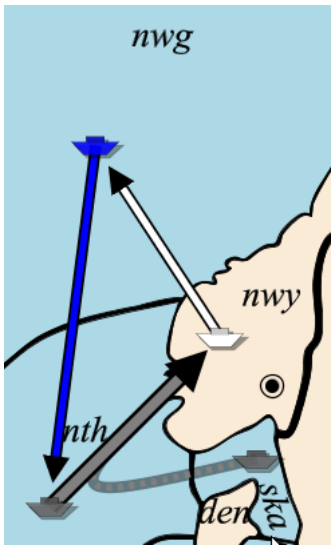
The Math of Adjudication

by Lucas Kruijswijk

Two proofs about the Diplomacy rules

As said earlier, the interpretation of the official rulebook is encoded in the equations, while any follow up is a matter of mathematics. In this chapter we will prove two properties of the rules.

But first, I have to introduce the notion of a “dependency graph”. If we look again at figure 9:



England:
F nwg -> nth

Germany:
F ska Supports F nth -> nwy
F nth -> nwy

Russia:
F nwy -> nwg

Figure 9 repeated.

Then we can make a directed graph between the dependencies of the orders:

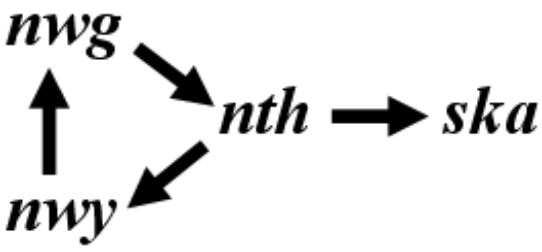


Figure 11.

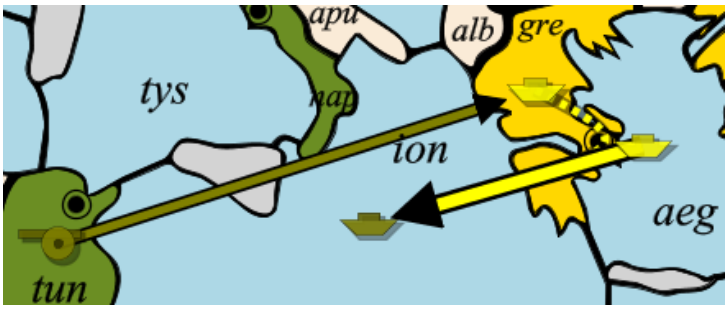
The arrows no longer represent *moves*, but *dependencies*. Resolving the North Sea order depends on both on the resolution of Norway and on the resolution of the support order of Skagerrak. This is shown by the two arrows originating from *nth*.

Some people might like to draw dependency arrows in the opposite direction: however, this way the dependency arrows have the same direction as the movement, and in the case of

the simple recursive algorithm, recursion is entered in the direction of the arrow.

In the dependency graphs we only look at the real*move*, *support* and *convoy orders*. The equations do also have other values — *attack strength*, *hold strength*, and so on — but those are shortcut in the graph.

In case of the convoy paradox of figure 7:



Turkey:
F aeg -> ion
F gre Supports F aeg -> ion

Italy:
A tun -> gre
F ion Convoys A tun -> gre

Figure 7 repeated.

The dependencies are as follows:

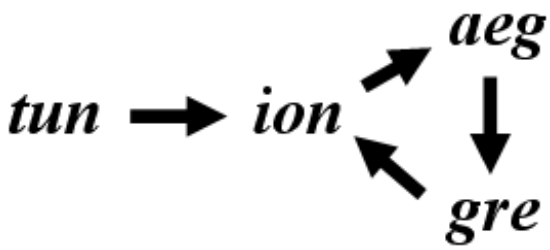
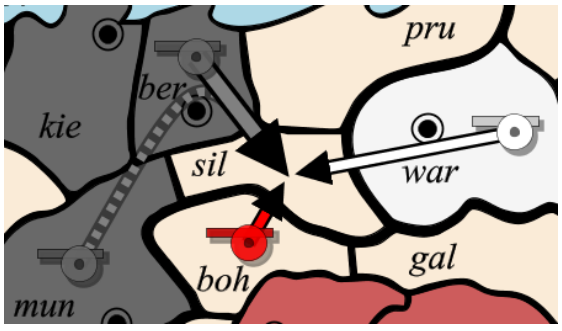


Figure 12.

Although the army in Tunis plays a major role in this convoy paradox, the resolution of the Tunis order is not part of the cyclic dependency. In resolving the support order of Fleet Greece it is important to know whether the fleet in the Ionian Sea will start the convoying operation or not, but the success or failure of the Tunis order is irrelevant.

This example shows that there is a subtle difference between a dependency on the *existence* of an order and a dependency on the*resolution* of an order. The existence of the Tunis order may cut the support in Greece as a result, but the resolution of the Tunis order is of no importance for that matter.

This subtle difference occurs also in much simpler situations. Consider the bounce situation of figure 6:



Germany:
A ber -> sil
A mun Supports ber -> sil

Russia:
A war -> sil

Austria:
A boh -> sil

Figure 6 repeated.

If we remove the move orders for Berlin and Warsaw from the order set, then the army in Bohemia could advance. So the resolution of the Bohemia order depends on the existence of the Berlin and Warsaw orders. However, if we look at the equations, then we see that the resolutions of the Berlin and Warsaw orders are of no importance to the resolution of the Bohemia order. To adjudicate the Bohemia order we need to calculate the *attack strength* of the Bohemia unit and the *prevent strength* of both the army in Berlin and the army in Warsaw. The *prevent strength* of the Berlin unit is dependent on the success of the *support* of Munich. For those calculations we do not need to know the success or failure of any *move*. As a result, the dependency graph is (surprisingly) as follows:

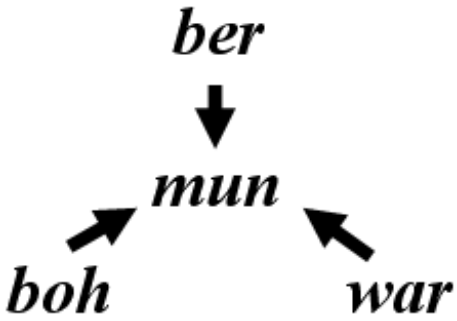


Figure 13.

The dependency graph indicates (correctly) that after resolving Munich, any of the orders Berlin, Bohemia or Warsaw can be adjudicated.

Having established the concept of a dependency graph, we now want to prove the following:

THEOREM I

No order can be part of two different cycles of dependencies.

This means that an order set with the following dependency graph cannot exist:

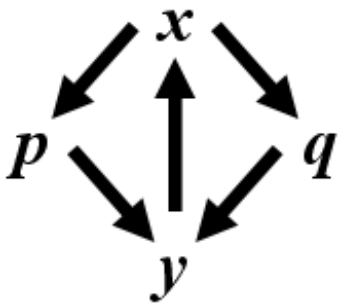


Figure 14.

Both *x* and *y* are part of a cycle with *p* and a cycle with *q*, which violates the theorem.

To prove this theorem, we have to distinguish between 6 types of orders, which are listed in the following table:

Type	Description	Direct or indirect dependent

		on
A	A move order, not part of a head to head battle, with at the destination another unit with a move order.	A, B, C, D, E, F
B	A move order, attacking a fleet with a convoy order.	B, E, F
C	Any other move order.	B, E, F
D	A support order, supporting a move with at the destination an unit attacking the support.	B, C, E, F
E	Any other support order.	B, E, F
F	A convoy order.	B, E, F

The dependencies in the last column list the type of orders on which the resolution of the order can depend directly or indirectly.

The set of support orders is split between orders of type D and orders of type E. The purpose for this split is to distinguish between orders that are subject to the dislodgement rule (a support is cut when it is dislodged), and those that are not. Formally, the dislodge rule would still apply for orders of type E: however, the rule is not relevant for that type, since the support is already cut by the simple fact that the unit is being attacked by the unit that may dislodge.

Again, remember the difference between a dependency on the existence of an order and the resolution of an order. Support orders of type E can be cut due to the *existence* of an order of type C, but cannot depend on the *resolution* of an order of type C.

The detailed proof of each of the dependencies can be derived from the equations and is left as an exercise for the reader.

When we have a situation on the board, we mark every dependency and every order that is in a cycle. We cannot remove the unmarked orders from the scene, because those orders can play a role in the adjudication of the marked orders in various ways.

If we observe the dependencies, then the marked orders (connected by marked dependencies) form pockets (islands) that consist entirely of orders of type A, or a mixture of B, E and F.

Still, a single pocket may contain an order that is part of multiple cycles. For a pocket that consists only of orders of type A, it is rather clear that these orders form one circular movement, with all orders in one cycle. The same counts for a pocket with orders of type B, E and F, because a movement of type B can only disrupt one convoy, a support of type E can only support one unit, and a convoy of type F can only convoy one unit.

Although the above reasoning may look obvious, it gave me a headache to get the right arguments. The tricky part is that the orders that can influence the order of the row are written down in the third column of the table. Thus, an order of type B, E, or F can only be influenced by the resolution of an order of type B, E or F. However, to prove that a pocket of B, E and F orders cannot have orders that are part of multiple cycles, we look at the dependencies in the opposite direction (the influence of the order).

Thus, the overall proof consists first of limiting the scope to pockets of certain types of orders, and second of proving that those pockets constitute a single cycle. For the convoy paradoxes, in the second step we look at the dependencies in the opposite direction.

This ends the proof of Theorem I.

Theorem I implies that if we have a cycle, then any order on which the cycle depends, can be resolved first. Consider the dependency graph based on figure 9 drawn in figure 11:

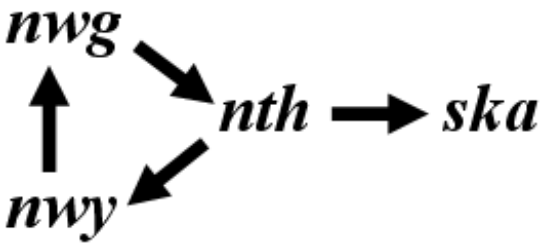


Figure 11 repeated.

We maybe could extend the situation and make the success of the Skagerrak order dependent on other orders. However, we could never make the resolution of the Skaggerak order dependent, directly or indirectly, on North Sea, Norway, or Norwegian Sea. If so, the North Sea order would be in the cycle *nth-nwy-nwg* and in another cycle with *ska*. This violates theorem I.

This is true in general, meaning that in case of a cycle, all orders on which the cycle depends can be resolved first. This counts even when the orders on which the cycle depends contain an independent cycle. Consider the following dependency relationship:

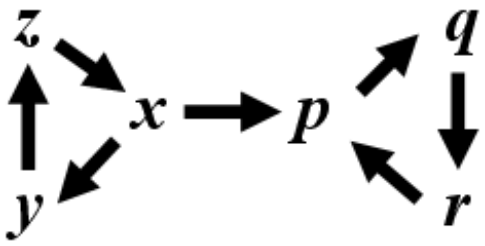


Figure 15.

In such a situation, the *pqr* cycle can be resolved first, resolving all orders on which the *xyz* cycle depends. The dependency relationship of this figure does not violate theorem I, because every order is part of at most one cycle. Nevertheless, this situation cannot occur with the standard rules. The proof is left as exercise for the reader. Of course, a variant rule might introduce this kind of situation, but that does not concern us for now.

Given the fact that any order on which a cycle depends can be resolved first, we want to prove:

THEOREM II

In case of the following conditions:

- There is a cycle;
- All other orders on which the resolution of one or more orders in the cycle depends, and which are not part of the cycle itself, are resolved;
- The cycle has a unique resolution.

Then there is at least one order in the cycle that can be resolved based only on partial information.

We prove this by contradiction. That is, we assume:

- There is a cycle;
- All other orders on which the resolution of one or more orders in the cycle depends, and which are not part of the cycle itself, are resolved.
- The cycle has a unique resolution;
- None of the orders can be resolved based only on partial information.

Back to the dependency graph of figure 11:

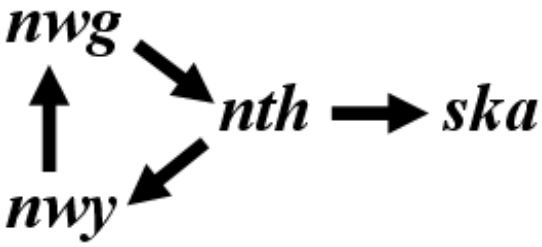


Figure 11 repeated.

If the resolution of the order to Fleet Skagerrak makes the order to Fleet Norway irrelevant for purposes of resolving the order to Fleet North Sea (which is the case when this graph is based on the situation of figure 9), then the order of the North Sea can be adjudicated based on partial information. This means that if we respect the assumption that none of the orders in the cycle can be resolved with only partial information, the cycle may depend on other orders, but that those other orders may not make the dependencies on the orders within the cycle irrelevant.

This restricts the dependency relations within the cycle. Suppose we have a direct $x \rightarrow y$ dependency in the cycle. Since y must be relevant for x, a different value for y must mean a different value for x (note this is only true because y is limited to two values). This means that x has the same resolution as y (x fails when y fails and x succeeds when y succeeds) or x has the opposite resolution of y (x fails when y succeeds and x succeeds when y fails).

From the initial assumption we know that there is one exclusive resolution. From the restricted dependency relations, it follows that the opposite of the resolution must also be a resolution respecting the equations. But this contradicts the assumption that there is a single resolution. And this proves Theorem II.

Theorems I and II together prove that an algorithm that can make decisions based on partial information will find the unique resolution for a cyclic dependency.

PREV - NEXT

Lucas Kruijswijk
(L.B.Kruijswijk@inter.nl.net)

If you wish to e-mail feedback on this article to the author, and clicking on the envelope

*above does not work for you, feel free to use the **"Dear DP..."** mail interface.*

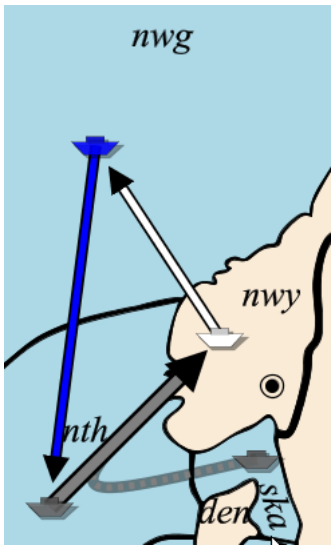
The Math of Adjudication

by Lucas Kruijswijk

A simple recursive guess algorithm that works in all cases

When I was writing theDATC, I had contact with a few people who were developing an adjudicator. One of them was Christian Hagenah, the creator of DipTool. He explained to me that DipTool does not work with decisions based on partial information.

Again we look at the circular movement with one supported move:



England:
F nwg -> nth

Germany:
F ska Supports F nth -> nwy
F nth -> nwy

Russia:
F nwy -> nwg

Figure 9 repeated.

What we can do (as alternative for a decision based on partial information) is to make a guess for one of the units in the cycle. For instance, we could guess that the English fleet in the Norwegian Sea succeeds. After this guess, the Russian in Norway can advance and so does the German fleet in the North Sea. Now, we recalculate the guess. Since the German fleet left the North Sea, the English fleet is free to move there. This means that the resolution of the English fleet equals our initial guess. Thus, we have a consistent resolution.

However, there might be a second resolution. So we also evaluate with the guess that the English fleet fails. As consequence, the fleet in Norway will bounce, but the German fleet in the North Sea will still advance, due to support from Skagerrak. Again, we recalculate the guess. Since, the North Sea is open, the fleet in the Norwegian Sea will advance. This contradicts the guess that the adjudication is based on. Thus, this is an inconsistent resolution.

Of the two guesses, only one led to a consistent result. So we just choose the resolution where the English fleet succeeds in moving. If the unit in Skagerrak is removed, then both guesses will lead to a consistent resolution, and the situation has to be passed to the backup rule.

DipTool builds up an explicit directed graph based on the dependencies of the orders. These are the same type of graphs as drawn in the previous chapter in figures 11, 12, 13, 14 and 15. The algorithm starts to work on the nodes that do not have any outgoing arrows. Those are the orders that are not dependent on any other order. It resolves those orders first, and removes them from the graph. Orders that do have dependencies will get rid of their outgoing arrows at the moment that all dependencies are resolved. When there are no longer any nodes with outgoing arrows, this means either that all orders are resolved, or that there is a cyclic dependency. For the cyclic dependency, it will use the guessing approach.

Both times that the order of the German fleet in the North Sea was resolved, the resolution of the Russian order was known. Although this information was based on a guess, the function that executes the equation does not need to know this. This shows the advantage of the guess algorithm above an algorithm that makes decisions based on partial information. All information for calculating a equation is available, and this avoids having to program the equations twice, which would be required for the partial-information based algorithm.

There are also some disadvantages. First of all, it was not invented by me! More seriously, this method requires additional software for graphs and to set up these graphs. For languages such as Java, standard graph libraries are probably available; but for a language like C, this requires significant coding.

So it would be great to combine the advantages of the different approaches, by using guesses to avoid double-coding for decisions that must be based on partial information, but also avoiding large amounts of generic code.

At first I couldn't figure it out, but a few years later, when the problem popped up in my head again, I finally got it. And as always with these kind of things, when you look back, you wonder why you didn't find the solution earlier.

The algorithm cannot handle orders that are part of multiple cycles: but that doesn't matter, because according to theorem I, such multiple cycles cannot exist. For a badly defined variant rule, such orders might exist, and the graphs of DipTool or partial information decisions might be easier to adapt. In the standard game, however, multiple cycles simply do not exist.

I give here a part of the algorithm in C. It successfully passes tests for circular movement, convoy paradoxes, and cyclic dependent situations with only one resolution. For readers who cannot read C, I will explain what is going on afterwards:

```
1 /* Possible resolutions of an order. */
2 #define FAILS 0
3 #define SUCCEEDS 1
4
```

```
/* For each order we maintain the resolution. */
int resolution[34];

/* The resolution of an order, can be in three states. */
#define UNRESOLVED 0 /* Order is not yet resolved, the
                      * resolution has no meaningful
                      * value.
                      */
#define GUESSING 1   /* The resolution contains a value,
                      * but it is only a guess.
                      */
#define RESOLVED 2   /* The resolution contains a value,
                      * and is final.
                      */
int state[34];

/* A dependency list is maintained, when a cycle is
 * detected. It is initially empty.
 */
int dep_list[34];
int nr_of_dep = 0;

/* Function: resolve(nr)
 * nr - The number of the order to be resolved.
 * Returns the resolution for that order.
 */
int resolve(int nr) {
    int i, old_nr_of_dep, first_result, second_result;

    /* If order is already resolved, just return
     * the resolution.
     */
    if (state[nr] == RESOLVED)
        return resolution[nr];

    if (state[nr] == GUESSING) {
        /* Order is in guess state. Add the order
         * nr to the dependencies list, if it isn't
         * there yet and return the guess.
         */
        i = 0;
        while (i < nr_of_dep)
            if (dep_list[i++] == nr)
                return resolution[nr];
        /* Order not found, add it. */
        dep_list[nr_of_dep++] = nr;
        return resolution[nr];
    }
    /* Remember how big the dependency list is before we
     * enter recursion.
     */
```

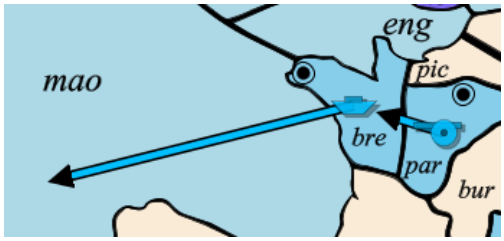
```
56    old_nr_of_dep = nr_of_dep;
57
58    /* Set order in guess state. */
59    resolution[nr] = FAILS;
60    state[nr] = GUESSING;
61
62    /* Adjudicate order. */
63    first_result = adjudicate(nr);
64
65    if (nr_of_dep == old_nr_of_dep) {
66        /* No orders were added to the dependency list.
67         * This means that the result is not dependent
68         * on a guess.
69         */
70
71        /* Set the resolution (ignoring the initial
72         * guess). The order may already have the state
73         * RESOLVED, due to the backup rule, acting
74         * outside the cycle.
75         */
76        if (state[nr] != RESOLVED) {
77            resolution[nr] = first_result;
78            state[nr] = RESOLVED;
79        }
80        return first_result;
81    }
82
83    if (dep_list[old_nr_of_dep] != nr) {
84        /* The order is dependent on a guess, but not our
85         * own guess, because it would be the first
86         * dependency added. Add to dependency list,
87         * update result, but state remains guessing
88         */
89        dep_list[nr_of_dep++] = nr;
90        resolution[nr] = first_result;
91        return first_result;
92    }
93    /* Result is dependent on our own guess. Set all
94     * orders in dependency list to UNRESOLVED and reset
95     * dependency list.
96     */
97    while (nr_of_dep > old_nr_of_dep)
98        state[dep_list[--nr_of_dep]] = UNRESOLVED;
99
100   /* Do the other guess. */
101   resolution[nr] = SUCCEEDS;
102   state[nr] = GUESSING;
103
104   /* Adjudicate with the other guess. */
105   second_result = adjudicate(nr);
106
```

```
107     if (first_result == second_result) {
108         /* Although there is a cycle, there is only
109          * one resolution. Cleanup dependency list first.
110          */
111         while (nr_of_dep > old_nr_of_dep)
112             state[dep_list[--nr_of_dep]] = UNRESOLVED;
113         /* Now set the final result and return. */
114         resolution[nr] = first_result;
115         state[nr] = RESOLVED;
116         return first_result;
117     }
118     /* There are two or no resolutions for the cycle.
119      * Pass dependencies to the backup rule.
120      * These are dependencies with index in range
121      * [old_nr_of_dep, nr_of_dep - 1]
122      * The backup_rule, should clean up the dependency
123      * list (setting nr_of_dep to old_nr_of_dep). Any
124      * order in the dependency list that is not set to
125      * RESOLVED should be set to UNRESOLVED.
126      */
127     backup_rule(old_nr_of_dep);
128
129     /* The backup_rule may not have resolved all
130      * orders in the cycle. For instance, the
131      * Szykman rule, will not resolve the orders
132      * of the moves attacking the convoys. To deal
133      * with this, we start all over again.
134      */
135     return resolve(nr);
136 }
```

The programmer should write an *adjudicate* function that implements all the equations. This function should not update any global data, and if it needs to know whether a *move*, *support* or *convoy order* succeeds or fails, it should call the *resolve* function (instead of looking in the global data). The *resolve* function may return a result based on a guess, but that is of no concern of the *adjudicate* function. So, the *adjudicate* function does not know anything about the state of an order, while the *resolve* function does not know anything about the equations.

To finalize the program, the programmer should implement the *backup_rule* function, and implement a main function that sets all orders to UNRESOLVED at the beginning of adjudication and then calls the *resolve* function for every order.

In cases where there is no cyclic dependency, the algorithm behaves similarly to the simple recursive algorithm (without guessing) as described earlier in this article. Consider the simple situation of figure 8:



A par -> bre
F bre -> mao

Figure 8 repeated.

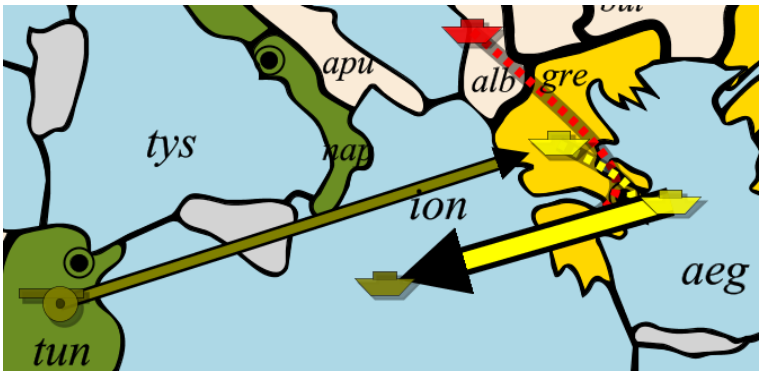
If the *resolve* function is called for the Paris order, then the recursion is as follows:

```
resolve(par)
  adjudicate(par)
    resolve(bre)
      adjudicate(bre)
      return SUCCEEDS (for adjudicate(bre))
    return SUCCEEDS (for resolve(bre))
  return SUCCEEDS (for adjudicate(par))
return SUCCEEDS (for resolve(par))
```

The only difference from the simple recursive algorithm (without guessing) is that the *resolve(par)* call will put the Paris order into a GUESSING state with the resolution FAILS (line 58 to 60), before calling *adjudicate(par)*.

However, when the *adjudicate(par)* function returns with a positive result, the *resolve* function will notice that the adjudication result is not dependent on any guess (line 65). The initial GUESSING state is just ignored and the Paris order is set in a RESOLVED state with resolution SUCCEEDS.

The GUESSING state becomes relevant in case of a cyclic dependency:



Turkey:
F aeg -> ion
F gre Supports F aeg -> ion

Austria:
F alb Supports F aeg -> ion

Italy:
A tun -> gre
F ion Convoys A tun -> gre

Figure 16.

If we could make decisions based on partial information, then we could conclude that the fleet in the Aegean Sea has an *attack strength* of at least 2 and maybe as much as 3, depending on the *support* of Greece. The strength of 2 is enough to advance to the Ionian Sea. With this guessing algorithm such a conclusion is not possible anymore, because the *adjudicate* function needs full information (although the function is not aware of whether the information is final or a guess).

Suppose we call the *resolve* function for one of the orders in the paradox, for instance *resolve(aeg)*. How the recursion will develop, will become more clear if we look at the dependency graph:

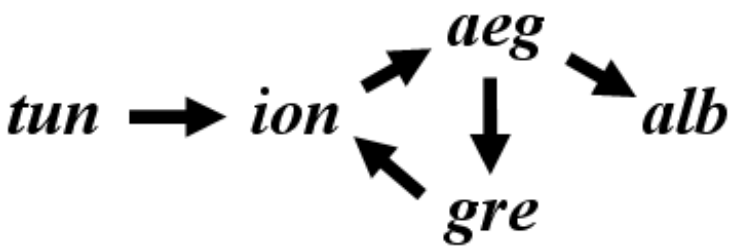


Figure 17.

Recursion will enter in the direction of the arrows, alternating between the *resolve* function and the *adjudicate* function:

resolve(*aeg*)
adjudicate(*aeg*)
resolve(*alb*)
adjudicate(*alb*)
return SUCCEEDS (for adjudicate(*alb*))
return SUCCEEDS (for resolve(*alb*))
resolve(*gre*)
adjudicate(*gre*)
resolve(*ion*)
adjudicate(*ion*)
resolve(*aeg*)

In this case, it is assumed that *adjudicate(aeg)* will first call *resolve(alb)* and then *resolve(gre)*. This will lead to an early RESOLVED state for Albania, because this order is not dependent on any other order. If *resolve(gre)* is called as first, then the Albanian order will be resolved when recursion as result of *resolve(gre)* has unwinded.

Along the way, the *resolve* function will set the state of the orders to GUESSING (line 60) and with an initial guess FAILS (line 59).

Order	*State*	*Resolution*
F aeg -> ion	**GUESSING**	**FAILS**
F gre Supports F aeg -> ion	**GUESSING**	**FAILS**
F alb Supports F aeg -> ion	**RESOLVED**	**SUCCEEDS**
A tun -> gre	**UNRESOLVED**	**-**
F ion Convoys A tun -> gre	**GUESSING**	**FAILS**

Now, things get different, because the *resolve* function is called for the Aegean Sea order, while that order is already in GUESSING state. The function will put this order in the dependency list (line 41 to 50) and return the stored guess resolution (line 51). The unwinding of the recursion will start at this point. While unwinding, the program will

adjudicate according to the guess of the Aegean Sea and ignore the guesses of Ionian Sea and Greece.

The *adjudicate* function of the convoy order of the Ionian Sea will succeed based on the guess of the Aegean Sea. The *resolve* function of the Ionian Sea sees that adjudication result depends on a guess, because an order was added to the dependency list (condition on line 65 is false). It adds the convoy order to the dependency list, updates the resolution with the adjudication result, keeps the order in GUESSING state, and returns the adjudication result (line 83 to 91).

With further unwinding of the recursion, the*adjudicate* function of the support order of Greece gives a negative result (because it is cut by the successful convoy). This order is also added to the dependency list.

Finally, we are back at the Aegean Sea order. The administration is now as follows:

Order	*State*	*Resolution*
F aeg -> ion	**GUESSING**	**FAILS**
F gre Supports F aeg -> ion	**GUESSING**	**FAILS**
F alb Supports F aeg -> ion	**RESOLVED**	**SUCCEEDS**
A tun -> gre	**UNRESOLVED**	**-**
F ion Convoys A tun -> gre	**GUESSING**	**SUCCEEDS**

Dependency list: aeg, ion, gre.

The recursion was executed as follows:

resolve(*aeg*)
adjudicate(*aeg*)
resolve(*alb*)
adjudicate(*alb*)
return SUCCEEDS (for adjudicate(*alb*))
return SUCCEEDS (for resolve(*alb*))
resolve(*gre*)
adjudicate(*gre*)
resolve(*ion*)
adjudicate(*ion*)
resolve(*aeg*)
return FAILS (for resolve(*aeg*))
return SUCCEEDS (for adjudicate(*ion*))
return SUCCEEDS (resolve(*ion*))
return FAILS (for adjudicate(*gre*))
return FAILS (for resolve(*gre*))
return SUCCEEDS (for adjudicate(*aeg*))

The *adjudicate* function of the Aegean Sea gives a positive result, despite the failure of the *support* from Greece. The *resolve* function will notice that the dependency list has been

extended, but also that its own order is in the dependency list (condition on line 83 is false). So the adjudication result is based on the guess of its own order. Since the guess and the adjudication are different, this resolution is inconsistent.

The function will now restart the adjudication, but with a different guess. It sets all orders in the dependency list to UNRESOLVED. So the Aegean Sea, the Ionian Sea and Greece are all back in UNRESOLVED state, while Albania keeps its final RESOLVED state. The dependency list is also made empty (line 93 to 98).

The Aegean Sea is set to GUESSING, but now with a SUCCEEDS resolution (line 100 to 102). The *adjudicate* function is called for the second time for the Aegean Sea.

If the recursion is worked out in a similar way, the function will return again with a positive result, and the dependency list will contain the same orders.

Based on those two results, it can be concluded that there is only one consistent resolution for the Aegean Sea, and that is the successful move. The orders in the dependency list are again set back to the UNRESOLVED state (line 108 to 112), and the Aegean Sea gets its final state RESOLVED with the resolution SUCCEEDS (line 113 to 116). This result is also returned.

The adjudication of the whole will be completed by calling the*resolve* function for the other orders.

A few things additional things should be noted about the algorithm. When order*x* and *y* are in a cycle, the algorithm would not work properly if the following recursion could appear:

resolve(*y*)
adjudicate(*y*)
resolve(*x*)
adjudicate(*x*)
resolve(*p*)
adjudicate(*p*)
resolve(*y*)
resolve(*q*)
adjudicate(*q*)
resolve(*y*)

In the third call of*resolve(y)* the *y* order is already in the dependency list. When back in the *resolve(q)* function, no increase in the dependency list is observed (while still dependent on a guess). However, this dependency relationship is as in figure 14, and both *x* and *y* are part of a cycle with *p* and part of another cycle with*q*. Again, the algorithm is not designed to handle orders that are part of multiple cycles (which gives numerous complications), so this is not important.

However, when writing the *adjudicate* function, one should take care with this sensitivity for orders that are part of multiple cycles. Earlier, I gave the proof of theorem I — that orders can not be part of multiple cycles with the standard rules. However, in that proof I removed the dislodge rule (a support is cut when dislodged) from the support orders categorized as type E. The dislodge rule is only relevant for the type D supports, and those are the supports for moves where there is a unit attacking the support from the destination

of the move. Since the proper working of the simple recursive guessing algorithm relies on the proof that orders are not part of multiple cycles, we should not re-introduce the dislodge rule for type E supports in the *adjudicate* function.

For instance, when we look at the dependency graph based on the situation of figure 16:

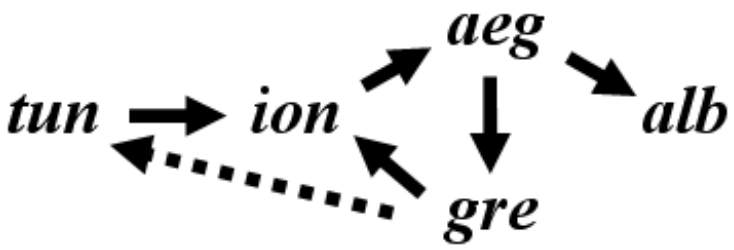


Figure 18.

If the *adjudicate(gre)* calls *resolve(tun)* (to check whether it will dislodge Greece) then we get a dependency as shown by the dashed arrow in the figure. With the addition of this arrow, we have one cycle that includes Tunis and another cycle that excludes Tunis. The *resolve* function is not designed to handle such a dependency relationship. Thus, when programming the equations in the *adjudicate* function, one should take care not to check unnecessarily for supports cut by dislodgement, although the literal equations have specified such cuts without conditions.

With this in mind, we observe that to resolve the order of the Aegean Sea, the state of the Tunis convoy order remains untouched. At the same time we see that the order itself plays a major role in the situation. This shows again that a dependency on the *existence* of an order and the dependency of the *resolution* of an order are two different things.

The *resolve* function is not entirely fragile. If the *adjudicate* function calls the resolve function more than once for the same order, the function will handle it correctly. Suppose we have the situation as in figure 8, where Paris moves to Brest and Brest moves to the Mid-Atlantic Ocean. If we calculate *adjudicate(par)*, we need to know the *attack strength* of Paris and the *hold strength* of Brest. When calculating this, for both strengths the result of *resolve(bre)* is necessary. We get a double dependency:

Figure 19.

The *resolve* function is capable of handling this. One might also choose to prevent that the *adjudicate* function will call the *resolve* function twice for the same order. If written that way, the *resolve* function can be slightly simplified (but beware, there are multiple tricky situations).

Furthermore, the algorithm can handle a situation where a cycle is dependent on the resolution of another cycle. This cannot occur with the standard rules, but might happen with a variant rule. Suppose we have a dependency relationship as shown in figure 15. If the algorithm retreats from recursion and has added orders of the *xyz* cycle to the dependency list, then the *adjudicate* function may enter the branch containing the *pqr* cycle. The unfinished dependencies of the *xyz* cycle stay on the bottom of the dependency list, while the *pqr* cycle is solved at the top of the list. Once the *pqr* cycle is resolved, the

algorithm will continue with the *xyz* cycle.

A final note should be made on the algorithm that determines the convoy path. The simplest way to implement this is to use a depth-first search algorithm with a recursive function. To do this, we have to mark the orders already visited in the search algorithm. We should be careful in taking global data for the marks, since in the search algorithm, the *resolve* function is called for the convoy orders. This in turn, may call the *adjudicate* function and lead to another convoy path calculation. It may even lead to the same convoy path calculation. If the *resolve* function is called for the Tunis order, then the *path* needs to be calculated; and to achieve this *resolve(ion)* is called. After some recursion steps, this leads to *adjudicate(gre)*, which also needs to know whether there is a convoy path for Tunis.

There are several ways to avoid a mess-up of the global mark data for the convoy path search. One could make the mark data as local, by putting it on the stack. However, a real programmer who has also coded for small processors doesn't like to put arrays on the stack (an easy upper limit is 34 x 34 x 1 byte — that's a little bit more than 1 kilobyte, not really a problem). Another way is to assign the state variable UNRESOLVED, GUESSING or RESOLVED to the *path* equation also. This means that this equation is not only calculated in the *adjudicate* function, but it is also administrated by the *resolve* function. This complicates the data structures, because the *nr* parameter passed to the *resolve* and *adjudicate* functions may refer to a convoy path.

A simpler way is to call the *resolve* function for each matching convoy order first, and start the depth-first search algorithm afterwards. Since all the convoy orders are in the RESOLVED or GUESSING state during the execution of the depth-first search algorithm, this will prevent the depth-first search algorithm from being restarted in recursion.

Programming the simple recursive guess algorithm still requires that many details be coded correctly. Nevertheless, I believe this is the simplest flawless adjudication algorithm. The following properties can be observed:

1. The equations are programmed only once,
2. The generic *resolve* function is not very big,
3. No complex data structures are required, and
4. The generic part is nicely separated from the equation code.

And so, 10 years after I was first introduced to Diplomacy, finally, at last...

PREV

Lucas Kruijswijk
(L.B.Kruijswijk@inter.nl.net)

*If you wish to e-mail feedback on this article to the author, and clicking on the envelope above does not work for you, feel free to use the **"Dear DP..."** mail interface.*