A Prototype System for Automated Tactical Situation Assessment

Drew McDermott

YALEU/CSD/RR #674
January, 1989

This work was supported by the U.S. Army Center for Signals Warfare, contract number DAAB10-86-K-0604.

A Prototype System for Automated Tactical Situation Assessment

Drew McDermott
Yale University

Table of Contents

1 Introduction 2
2 Region Analysis 4
2.1 Grouping Pixels into Regions 4
2.2 Line-of-Sight Analysis 7
2.3 The Terrain-Analysis Graph 12
3 Force-Movement Analysis 13
3.1 Finding Flows 14
3.2 Complexity of the Algorithm 20
3.3 Attrition Models 24
3.4 Counterplanning 26
4 The Implementation 29
5 Conclusions 32
Appendix: A Connection-Machine Simulator 34
Bibliography 35

Abstract

Tactical situation assessment is the analysis of a battle situation to determine enemy capabilities and intentions. One way to perform this analysis automatically is to divide the battlefield into meaningful regions, based on trafficability and capacity, and find which regions can be used to attack or travel to which other regions. After this off-line analysis, a particular tactical situation can be analyzed by generating paths for flow of enemy forces, finding weak spots in that flow, generating counterattacks for friendly forces, and so forth. The static and dynamic relationships among regions and force entities can be displayed graphically, revealing important tactical features to a human user. The user can interactively alter the forces on the battlefield, and call for new battlefield scenarios. The proposed representation of alternative enemy plans may provide an efficient basis for specialized plan recognition.

1 Introduction

Tactical situation assessment is the process of analyzing battlefield terrain, enemy force dispositions and objectives, and intelligence reports, in order to arrive at an assessment of the enemy's intentions and capabilities. The present report describes software developed at Yale over the past two years (under contract number DAAB10-86-K-0604 from the Army Center for Signals Warfare) for performing simple tactical situation assessment. The software does not do the job completely, but represents an experimental foray into some of the harder aspects of the problem. The domain of tactical situation assessment is a good one for AI application, because it allows us to explore tough issues of plan recognition and abduction in a relatively contained environment.

In real life, a battlefield commander brings a creative intelligence to this task. He[1] knows a lot about the enemy's past history, the overall context of this one engagement area, the ultimate objectives of the enemy, and so forth. It is important in applying AI techniques to a problem like this that one not plunge headlong into anthropomorphism, attempting first to code up some "knowledge" and type it into the computer. AI is no different from any other area of Computer Science when it comes to solving problems. We must think carefully about the problem, and about algorithms and data structures for solving it. The main difference between AI and other approaches is a somewhat greater willingness to think heuristically and program symbolically. Hence, it what follows I will be careful to outline exactly what the algorithms we have developed can and cannot do. I will in the end talk about how they could be extended to handle more complicated problems.

I will view tactical situation assessment as fundamentally a problem in *plan recognition*, the inference from the enemy's actions to the plan of the enemy that best explains those actions. This already represents a narrowing of the problem, because in many situations the enemy is doing nothing, or merely reacting to our plans, and the assessor should say so. I will assume that the enemy is planning to do something concrete in the near future, and on a battlefield that implies that he is ready to go somewhere and shoot something. The current set of algorithms will never report that "The enemy is in full retreat under our attacks, and is weakest in Sector R."

On the other hand, the algorithms cannot print out a detailed summary of the enemy's plans, or even a summary of all of his possible plans. There are too many possibilities. Instead, the algorithm must assign one or more *tactical significances* to each piece of terrain on the map, and assume that the observed or predicted presence of the enemy on that piece of terrain means that he is trying to exploit its tactical significance. The print-out will therefore emphasize where the enemy is hypothesized to be, where he is going, and why

[1] Or she, and similarly for all other masculine pronouns in this paper

someone might be trying to go to each of those places, but will leave it to a human to piece those motivations together into a coherent plan.[2]

The *Combat Intelligence* field manual (U.S. Army 1971) distinguishes five "military aspects" of terrain: observation and fire, concealment and cover, obstacles, key terrain, and avenues of approach. A trained intelligence officer can look at a map and see that a certain bridge is a piece of key terrain, or that a forest provides concealment from aerial observation. I will be talking about several of these aspects, not always in terms a soldier would recognize, but one thing should be made clear now: The present research neglects most of the things an intelligence officer sees on a map, including the bridge and trees just mentioned. Instead, I assume that the battles are being fought on a cloudless day on the moon, so that there is nothing in the vicinity to think about except mountains and boulders. The reason for this simplification is that the shapes of the mountains and boulders are harder to think about than bridges and trees, and I wanted to work on the hardest part of the problem first. This decision may seem backwards to a nonexpert in AI, so let me explain. A bridge can just be marked as a crucial conduit from one side of a river to another. A forest can easily just be marked as providing concealment. Such symbolic features are the stock in trade of AI programs, and will present very few difficulties. But it is less plausible that a human will be willing to go carefully over a topographic map and locate all the avenues of approach. Here we demand that the program extract such useful features for itself, from raw elevation data. Much of our effort has been spent on this task, since we are confident that analysis of bridges, forests, and towns can be put in later.

Hence the inputs to the program are as follows:

1 Elevation data: A map of a battlefield, roughly 10 km on a side, divided into 100 m by 100 m *pixels*, with a single elevation number for each pixel.
2 Accessibility data: One bit per pixel stating whether the pixel is traversable by a military unit of significant size at the division level (i.e., a company or battalion). A pixel could be untraversable because of excessive steepness, or for some other reason.
3 Force concentrations: Hypothesized friendly ("Blue") and hostile ("Red") force concentrations, each represented as a location in Cartesian coordinates and an estimated size.
4 Red objectives: Hypothesized objectives for the enemy, represented as a location in Cartesian coordinates and an estimated force requirement to carry out some mission at that point. The program has no other way of figuring out where the important overall objectives are, because those will usually depend on factors outside the map area.

The program, called YATS, operates as follows: Offline, it performs an analysis of the terrain, dividing it into compact regions of approximately uniform width and finding line-of-sight relationships among them. The result is a *terrain-analysis graph* that summarizes the important aspects of the terrain. This analysis is fairly time-consuming, but need be done just once for a battlefield area.

Next, YATS collects force concentrations and objectives. The program traces out possible attack routes for Red forces toward their objectives. It tags every region with an estimate of the "expected" force that will

[2] Here and elsewhere I will admit that our treatment is somewhat abstract, and not based on detailed models of the thought processes of real G-2s and commanders. I really don't know whether the output of this program would be directly useful in its present form. It will be clear later that this really doesn't matter so far, given other limitations of the system.

traverse it from each source to each sink, where "expected" is used be analogy with the statistical concept of expected value. That is, the program does not literally expect an attack in this strength through this region, but over many such situations in which Red attacks these objectives, it expects to see a force of this strength on the average. (Even so qualified, this definition must be taken with a grain of salt, given the factors the current program neglects.)

After generating a preliminary Red attack flow, the program identifies a certain class of pieces of key terrain, namely those that could be used to counterattack against the advancing Red forces. These terrain areas become objectives in a Blue counterattack, which is generated using the same force-flow algorithm. Finally, the Red attack is rerun, taking into account anticipated Blue responses. The results are displayed graphically.

The current program does not attempt to match intelligence data with the projections arrived at, in order to sort out what actually happens from all the possibilities. This is an obvious area for future research.

In the rest of this report, I will describe each of these phases in more detail.

2 Region Analysis

The first phase is an off-line grouping of traversable pixels into regions, resulting in a graph of the interesting relations between regions. There are two criteria of interestingness: mobility and attack. We want to place links between regions that adjoin, and between regions that have a clear view of each other even though they don't adjoin. These two computations are quite different.

2.1 Grouping Pixels into Regions

Every pixel is classified as traversable or untraversable, up front.[3] This dichotomy is based on the idealization that every pixel (which, you will recall, represents a 100 m square portion of the world) may either be traversed by military forces or not. It is hard to be entirely comfortable with this idealization, but it seems reasonable if we focus our attention on the kinds of forces expected at the division level. Small harassing or reconnaissance teams might well go where battalions cannot go, but we can neglect that. Presumably a commander assumes that his movements are always observed and always subject to harassing fire.

The purpose of dividing the map into regions is, in essence, data compression. By thinking of a region as a unit, we save the effort of thinking about every pixel. For many purposes, it doesn't matter exactly how we divide the map into regions; if there are a few too many, our algorithms will survive. There are two main principles we adopt in dividing into regions: A region should be of approximately uniform width; and long thin regions ought to be straight. The first principle derives from the fact that the chief constraint on

[3] It would be easy to derive this information purely from elevation data, but other considerations may enter into it.

the motion of forces through a region is how wide it is. The width of a region varies throughout, and is hard to define for an arbitrary region. We will describe one heuristic attempt below.

The second principle derives from the fact that long thin regions serve as conduits or, in military parlance, "avenues of approach," to other places. If such a region has a sharp bend, then it should be broken at the bend so that it each half will have a definite direction. That way, the presence of troops moving through a region will strongly suggest a destination.

The second principle does not come into play until a preliminary grouping has been found. After that the algorithm finds polygonal approximations to regions, and makes sure that long thin ones are straight. The preliminary grouping is found by propagating numbers through grids of pixels. These numbers are an attempt to approximate the "local width" of an area of the map. A *spine pixel* is a traversable pixel that whose distance to the nearest untraversable pixel is a local minimum. For spine pixels, this distance is half the local width of the region. To extend the local width to other pixels, we must do two things: eliminate spurious spine pixels, and assign every pixel to some nearby spine.

The first phase is to find spine pixels. The algorithm labels every pixel with its Manhattan distance to the nearest untraversable pixel. That is, it first label untraversable pixels 0, every traversable one ∞. Then it labels every pixel with 1+the label of its smallest neighbor (using 8-neighborhoods) This algorithm repeats until there are no changes, at which point every pixel will be labeled with its distance to the nearest untraversable pixel.

We have been running this algorithm on a TI Explorer 2, which provides acceptable performance. However, these algorithms are well adapted to running on a parallel machine such as the Connection Machine (Hillis 1985) For this reason, I have actually written the low-level pixel-propagation code in *Lisp (Thinking Machines 1986), a dialect of Lisp that exploit the parallelism of the CM. On the Explorer, this code runs using a lazy-evaluation simulator, which imposes an overhead over just using arrays of numbers. Apparently, this overhead is quite acceptable. In what follows, I will omit further references to the CM simulator, but a more detailed description of it will be found in the Appendix.

The next step is to find local maxima using 8-neighborhoods. That is, for every pixel, if its eight neighbors are no larger than it, label it a local maximum. Such local maxima correspond to the centers of regions. If a region is long and narrow, the local maxima will be found at a ridge running down its spine. Unfortunately, in practice things don't work out so prettily. If a region varies in width or has wavy borders, there will be lots of local maxima, corresponding to little fragments with different Manhattan distances to their borders.

The next step compensates for those imperfections. The numbers are propagated from local maxima to their neighbors, so that in the end every pixel is located with a good guess as to its local radius. We actually store three numbers with every pixel. One number, M, is the actual maximum assigned to a pixel; when the algorithm is done, every pixel in a region will have the same M value. The second, D, is the distance to the pixel where the M value came from. The third, R, is the value of the nearest local max, which may be smaller than M if a larger M value propagated through a local max.

There are three ways in which these numbers propagate. The simplest is "downhill propagation." Suppose pixel P_1 is sitting next to pixel P_2, and P_1 is not a local maximum. (So its D is greater than 0.) Then if P_2 is closer to a bigger local maximum than P_1 currently is, we label P_1 with the local maximum from P_2. This process eventually labels every pixel that is not a local maximum with the M value from some nearby local maximum. As it goes, it labels pixels with increasing D numbers, keeping track of the distance to the source of the M value; the R values are the same as the M values.

In addition, spurious local maxima must be overwritten by dominating alternative maxima. The basic idea is to label pixels with the nearest local maximum. So if a pixel sees a neighbor with a smaller D value, it copies its maximum over (and takes a D value one greater than its neighbor). We also use a smoothing technique, which applies to a pixel P_1 with a smaller M value than its neighbor P_2. If P_2's value is ¡ than 1.2 times P_1's, or if the distance value D for P_1 is exactly equal to P_2's $+1$, then P_2's gets written to P_1. This algorithm tends to merge adjacent regions with "essentially the same" maximum value.

At this point every pixel is labeled with a number that represents the local maximum that propagated to it, which is roughly proportional to the width of the region the pixel is in. We will call this the *radius* of the region. Now find contiguous blocks of pixels with the same radius, and label every pixel in a block with an arbitrary integer which will be used as a region id for that region. In figure 1, the resulting region boundaries have been drawn using heavy lines. There are three regions, the left with radius 4, the upper right with radius 8, and the lower middle with radius 2.

The next task is to find polygonal approximations to the boundaries of each region. There are several ways to define what we mean by approximation. We are currently assuming this definition: One polyline L_1 approximates another L_0 if each vertex of L_0 is also a vertex of L_0, and for all pairs of adjacent vertices v_1,v_2 in L_1, the vertices between v_1 and v_2 in L_0 are sufficiently close to the line between v_1 and v_2 in L_1. The meaning of "sufficiently close" will depend on the width of the region whose boundary is being approximated.

Now, to find a polygonal approximation to a region boundary, we first find, for each boundary vertex v_0, the furthest vertex v_1 on the same boundary such that some line through v_0 is sufficiently close to every vertex between v_0 and v_1. Call v_1 the "partner" of v_0. Then we take the vertex v_0 with the furthest-away partner v_1, and test whether the line from v_0 to v_1 is a legal approximator of all the vertices in between. If not, we set v_1 to the next vertex, one step closer to v_0. Repeat this until a legal approximator is found. We now have approximated one piece of the region boundary. We snip that piece out, and approximate what is left.

There are actually two kinds of boundary to worry about: a boundary between two traversable regions, and a boundary between a traversable region and an obstacle. The former are guaranteed to be short, so

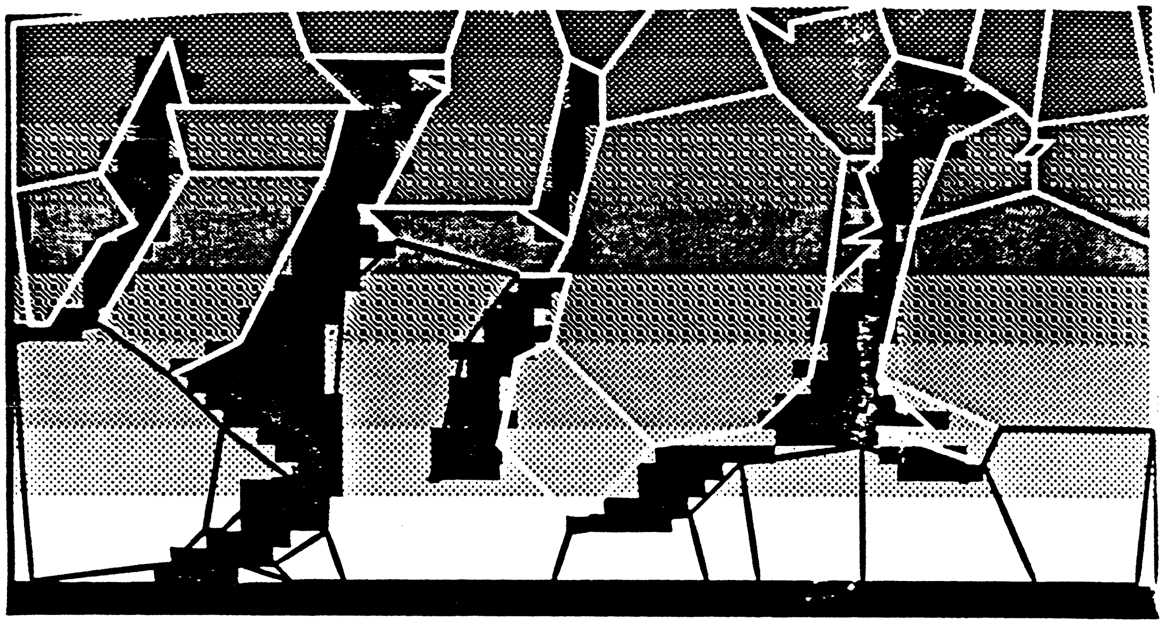

Figure 1 Approximating regions by polygons

we just approximate them with a single straight line, which becomes a portal between the two regions. The latter require the approximation algorithm described.

Applying the polygonal-approximation algorithm to figure 1 might give the result shown using heavy lines.

Now we can classify long, straight regions as conduits, everyone else as a "blob." Unfortunately, it may be the case that a region of constant width corresponds to several conduits. The region on the left in figure 1 is an example. Hence we must take our polygonal approximations, and break them into actual conduit-like pieces if possible. A conduit is defined as a subregion of approximately constant width bounded on at least one side by a long straight line. Since we have already found linear approximations to the boundaries of regions, and the regions are of approximately constant width "everywhere," all we have to do is find long straight lines, and break out the subregions they bound. By "long" here we mean long compared to the region width. This operation may reveal that a region consists entirely of a single conduit; or it may break a thin curving region into a chain of conduits; or it may break a region down into conduits and blobs. Except for the first case, this further breakdown introduces new regions, with portals to each other and to old regions.[4]

[4] The current implementation does not actually include the code to do this breakdown, although it has been written.

2.2 Line-of-Sight Analysis

The observation–fire pair is complementary to the concealment–cover pair. What is crucial to all these concepts is the idea of "line of sight." If a person at point A is to observe an object at point B, there must be a line of sight between them; and if the object at B is to remain concealed, there must not be. It is not always necessary that a defender be visible from the position of an attacking weapon, but he must be visible from whoever is controlling the fire of that weapon.

There are two ways to approach the line-of-sight problem, at the region level or the pixel level. At first glance, it looks as if we are mainly interested in line-of-sight relations between areas, and hence it might be wise to postpone line-of-sight calculations until after segmentation is done, and then search out from each region to find places to attack it from. But searching out from a region is a fairly clumsy operation, and often we are interested in more than just whether anything would be found by such a search. We might, for instance, want to know the shape of the area in region A that can spy some part of region B, and figuring that out requires finding all the pixels in A that can see some pixel in B. Hence we have tried a more ambitious approach, finding all pairs of pixels that can see each other, and later clumping together pixels in region A that can be used to observe pixels in region B.

It may sound as if just storing the intermediate results of this computation would be prohibitive. A map typically consists of about 10^4 pixels,[5] so we might have to examine and output 10^8 pairs. Fortunately, we can use hierarchical techniques for representing and computing the pairs. The line of sight from most pixels terminates fairly quickly; when a pixel has a long view, it can typically cover a wide area as well. These facts suggest analyzing the map at a coarse resolution first, using "fat" pixels called *pixblocks* that cover many actual pixels. There are three possible outcomes to a comparison between two pixblocks $P1$ and $P2$ at the coarse level:

1 You can see any point of $P2$ from any point in $P1$. (*Complete visibility*)
2 You cannot see $P2$ from any point in $P1$. (*Complete invisibility*)
3 You can probably see some point in $P2$ from some point in $P1$.

If the outcome is 1 or 2, then we can stop, storing the result at the coarse level. Future retrievals from any pixel covered by $P1$ or $P2$ must be careful to check the higher-level pixblocks to catch such relationships.

If the outcome is 3, then the algorithm must recurse, checking all pairs of sub-pixblocks, one drawn from $P1$ and one from $P2$. The recursion ceases with pairs that can see each other completely or not at all; or with pairs of elementary pixels, for which outcomes 1 and 3 are counted the same.

We obtain the coarse levels in the obvious way: Map pixels are clumped into two-by-two groups, making coarser pixels four times as big. These are then clumped in the same way, and so on, until a single fat pixel is obtained covering the whole map. We will use the word *pixblock* to cover these things, and count individual pixels as zero-level pixblocks. With each pixblock we store its maximum and minimum altitudes, that is, the

[5] Actually, in our current examples, more like 2×10^3.

max and min of the altitudes of any sub-pixel. In addition, we record whether a pixblock is *flat*, meaning that it is approximated well by a plane, or *bumpy*. I will be more precise about the definition of flatness later.

This algorithm wins over brute force if it succeeds fairly often in analyzing two broad, flat areas without descending to their pieces. It will often happen that the algorithm will not realize that $P1$ is completely visible to $P2$, but after recursion will find that every sub-pixblock of $P1$ can see all of every sub-piece of $P2$. In this case we want to do data compression, and store a single pointer pair between $P1$ and $P2$ rather than sixteen pointer pairs between their sub-pixblocks. (This idea is reminiscent of quadtrees (Samet 1985, Samet and Webber 1988), but differs in that each pixblock summarizes information about its components, instead of serving merely as an indexing structure for them.)

The algorithm starts with a map of elevation data, and imposes a hierarchy of pixblocks on it. It then calls (PAIR-ANALYZE P P), where P is the top pixblock, and PAIR-ANALYZE is defined as follows:

```
; Set up pointers between pixblocks.
 (PAIR-ANALYZE (P1 P2 - pixblock)
    (LET ((VR (Find line-of-sight relation between P1 and P2)))
      ; VR is either TOTAL, PARTIAL, or NONE
       (COND ((It's TOTAL)
               (Set up a VISIBLE pointer pair between P1 and P2))
              ((It's NONE)
               (Set up an INVISIBLE pointer pair between P1 and P2))
              ((It's PARTIAL)
               (COND ((At bottom level)
                       (Set up a VISIBLE pointer pair between
                        P1 and P2))
                      (T
                       (Enumerate traversable sub-pairs
                            S1 and S2 of P1 and P2
                         (PAIR-ANALYZE S1 S2))
                       (Consolidate pointers; that is, if all
                            four sub-pixblocks of P1 or P2
                            are pointed to by some pixblock, replace
                            with a single higher-level pointer.
                            Do this for both VISIBLE and INVISIBLE
                            pointers.))    ))    )))
```

It remains to describe the algorithms for testing pixblock intervisibility. There are three cases: testing whether $P2$ is completely visible from $P1$, testing whether whether $P2$ is completely invisible from $P1$, and testing whether a pixblock is completely visible to itself (i.e., is flat enough that all its sub-pixblocks are completely visible to each other). Of these, we will focus on the first case. The third is basically a matter of whether the terrain inside a pixblock P is flat enough that given a "periscope" of height h we could see over any hill.

When a pixblock is tested for flatness the first time, the program runs a standard, somewhat expensive, flatness test, and the result is stored on the pixblock so the test doesn't have to be repeated. The test is quite straightforward. First, the program finds the *approximating plane* for the pixblock. This is a plane containing the point x_0, y_0, z_0, where x_0, y_0 is the center of the pixblock, and z_0 is its "average" height (i.e., the average of its min and max heights). We need to constrain the remaining three degrees of freedom for the plane, which may be represented as the normal vector through x_0, y_0, z_0. We obtain such a constraint by assuming a slope in the x and y directions equal to the average slopes to neighboring pixblocks. That is, if D is the diameter of the pixblock, the average x-slope is

$$\frac{z_{x+1 \text{ neighbor}} - z_{x-1 \text{ neighbor}}}{2D}$$

and similarly for y. If we let $\mathbf{X}$ and $\mathbf{Y}$ be vectors pointing along the x and y axes with these slopes in the z direction, then the normal vector in the plane is $\mathbf{X} \times \mathbf{Y}$, the cross product of the two.

If a piece of terrain is exactly planar over a wide area, then the pixblocks in that area will coincide with their approximating planes exactly. To the extent that the terrain is bumpy, the quality of the approximation will degrade. Actually, there are two ways it can degrade. It can simply have many bumps, or it can be smooth but far from planar. The program can test for both these conditions the same way. It checks every pixel (not pixblock) inside the area being approximated to see how close it comes to the proposed approximating plane. If every pixel is within a fixed distance, the pixblock passes the test. (The distance is currently 10 m.)

The test whether a pixblock is entirely visible to itself consists just of the flatness test. The test of visibility between two different pixblocks $P1$ and $P2$ is more complex. We want the test to be as accurate as possible, but we are willing to sacrifice some accuracy for speed. Hence we make a simplifying assumption, that we can find two points, one in each of the two pixblocks, such that if something blocks the view between these two points, then there is no view between any two points in $P1$ and $P2$. These two points are called the *partial-view sighting points* between $P1$ and $P2$. Furthermore, if the two pixblocks are approximately planar, then we assume there are two points, one in each pixblock, such that if nothing blocks the view between them, then nothing blocks the view between any two points in $P1$ and $P2$. These are the *total-view sighting points.* Figure 2 shows how these points are found. We draw a line between the centers of $P1$ and $P2$. We then find points on that line that are highest above the approximating planes to serve as the partial-view sighting points. If the two pixblocks are approximately planar, we choose the lowest points above the approximating planes to serve as the total-view sighting points.

The program that does all this is called `OTHER-VISREL`. It takes two arguments, $P1$ and $P2$, representing two distinct pixblocks, and returns one of `TOTAL`, `PARTIAL`, or `NONE`, depending on the quality of the view between $P1$ and $P2$. It can return `TOTAL` only if $P1$ and $P2$ are flat and bigger than a single pixel. If either is bumpy there is no way to judge whether there is a total view between them without descending to sub-pixblocks; and if they are single pixels there is no point in distinguishing between partial and total visibility.

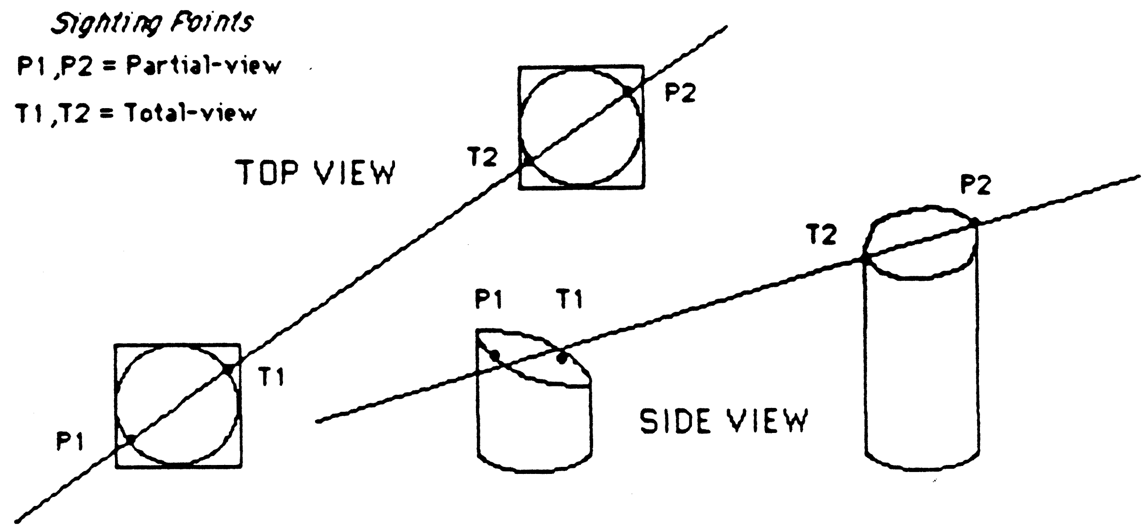

Figure 2 Sighting Points between Pixblocks

Hence the algorithm first computes the approximating planes of $P1$ and $P2$, and finds the partial-view sighting points between them. If the conditions are met, it sets boolean variable COULD-BE-TOTAL to *true*, and finds total-view sighting points. It then executes the following loop:

```
(LOOP FOR (Pixblocks between P1 and P2)
 UNTIL (All pixblocks examined)
 RESULT (COND (COULD-BE-TOTAL 'TOTAL)
              (T 'PARTIAL)   )
 UNTIL (A pixblock blocks the partial-view sighting line)
 RESULT 'NONE
   (COND (COULD-BE-TOTAL
          (!= COULD-BE-TOTAL
              (The pixblock doesn't block the total-view
               sighting line)))   ))
```

The loop repeatedly checks intermediate pixblocks to see if they block the partial-view sighting line (i.e., the line between the partial-view sighting points); and, if the total test is still alive, if they block the total-view sighting line. As soon as the total-view line is blocked, the total testing is turned off. As soon as the partial-view line is blocked, the algorithm stops and returns NONE.

The test for whether intervening pixblocks block a sighting line also sacrifices accuracy for speed. We model each such pixblock as a cylinder with a beveled top. If the pixblock is flat, we use its approximating plane as a model of its surface. (Figure 3) Otherwise, we model its surface as a horizontal plane, at either the maximum height of the pixblock or its minimum height. We use the minimum height for the partial-view test, the maximum for the total-view test. That is, for there to be a total view, the intervening bumpy pixblock must be incapable of blocking the total line no matter where its highest bump occurs. But for a

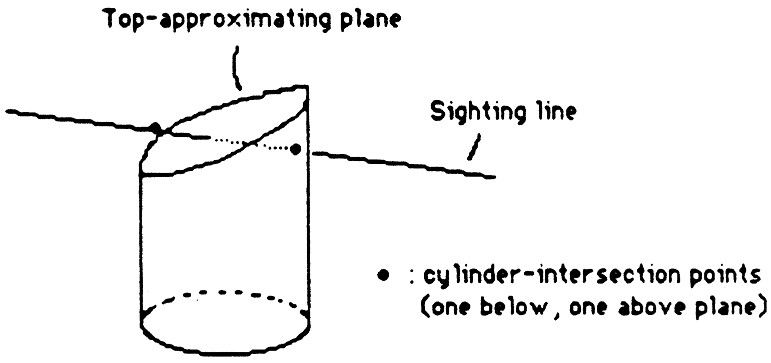
Top-approximating plane
Sighting line
● : cylinder-intersection points
(one below, one above plane)

Figure 3 An Intervening Pixblock and How it Might Block the View

partial view, the algorithm gives the intervening pixblock the benefit of the doubt in letting it get out of the way.

The test whether a pixblock modeled this way might actually block the view is then quite simple. We test to see if the line between the two sighting points intersects the cylinder. This is just a test for whether the projection of the line into the xy plane intersects the projection of the cylinder — i.e., a circle. If it does, there is a further check for whether either intersection point is below the plane of the top.

On simple examples, the algorithm runs fairly quickly, although there is room for improvement. On a map 66 by 35 pixels (i.e., representing a battlefield of 6.5 by 3.5 kilometers), it takes 30 minutes to do a line-of-sight analysis. Part of the speed is obtained by avoiding testing pixblocks that are marked as inaccessible. One way to speed the algorithm up further would be to do the pixel-level analysis in parallel, on a Connection Machine. We have not given this much detailed thought, since the results of the analysis can be saved on disk and restored, which is entirely adequate for experimental purposes.

2.3 The Terrain-Analysis Graph

The output of region analysis is a *terrain-analysis graph*, which gives a description of every region and the relationships among them. Regions are described in terms of their (approximate) *width*, a *boundary*, their *portals* to other regions, and their *attack relations* to other regions. We have dealt with all but the last in Section 2.2. Here I will describe how attack relations are found.

A *view link* between two regions consists of the following four items:

1 The higher of the two regions, R_H
2 The lower of the two, R_L
3 The piece of R_H that has a view of R_L
4 The piece of R_L that can be seen and shot at

The "pieces" in this data structure consist of unstructured collections of pixblocks. It would be easy to provide more structure, such as a polygonal approximation.

Finding view links is straightforward given the data structures the algorithm has already computed. The line-of-sight analyzer keeps track of the regions that each pixblock overlaps. (At the lowest level, of course, each accessible pixel is in exactly one region.) When the analysis is complete, the program enumerates all regions, and, within each region R all its pixblocks, looking for pointers to pixblocks in other regions whose average altitude is lower. For each such partner region R', the pointers from pixblocks in R to pixblocks in R' are collected. When R has been scanned completely, the result will be a list of regions that R can attack, plus pairs of pixblocks that can see each other. These pairs are then divided into a group of pixblocks in R and a group in R', which become subregions A and A', and $< R, R', A, A' >$ then becomes a view link, which is pointed to from both R and R'.

3 Force-Movement Analysis

Once the terrain-analysis graph has been produced, YATS can use it to produce tactical scenarios. These represent courses of action that Red might take, and possible Blue responses. Such predictions can then be matched against incoming intelligence reports to provide a hypothesis about what is happening. However, we do not attempt to provide detailed plans and counterplans for either side. The combinatorics would simply overwhelm us. Instead, we produce an abstract representation that subsumes all of an agent's possible plans, and reason about these representations in doing counterplanning and matching. The abstract representation is called a "flow graph," and it consists of a table specifying the *expected force flow* across every portal in the battlefield.

The algorithm assumes reasonable hypotheses about the current positions of force elements. These do not have to be completely accurate. It also must be given all possible objectives. Each hypothesized concentration is called a *force source*. Currently these are represented as straightforward objects that are associated with locations on the map, and displayed iconically. What the flow graph specifies is a function

$$F : \textit{force-source} \times \textit{portal} \longrightarrow \textit{amount} \times \textit{time}$$

which gives, for every force source and portal, an expected amount of that source that might travel across that portal (in a direction given by the sign of the *amount*), and roughly when the force would get there if it left now.

I use the phrase "expected force flow" guardedly. One can think of it as the amount of force that would make use of this portal on average, if the battle were fought many times, but that is just a convenient image. There is no rigorous event space here that would justify the use of probability. Still, the numbers at all portals are supposed to add up to the total amount of the force source (except when attrition is involved).

If 1/3 of the force from a source goes along route 1, and 2/3 through route 2, that is supposed to indicate that route 2 is more attractive — "twice as attractive" — as route 1, but that either one is definitely a possibility. (Splitting the forces is also a possibility, maybe even a necessity, but a more careful analysis of the capacities of the routes would be required to figure that out.)

The flow graph does not actually tell us about complete routes, but only about fragments of them. In Figure 4, there are are 2^n different routes through the regions shown, but the flow algorithm does not find them. It only reports that a certain amount of force is expected to flow through each of the regions. This insensitivity to combinatorics is the great strength of the algorithm. However, it does make extracting information a bit trickier. If YATS really generated detailed plans and counterplans, then it might generate a tremendous number, but we would know how to evaluate them. As it is, we have to work harder to get meaningful numbers out. An example is the attrition calculation. Suppose we want to decide how the flow will be affected by the presence of hostile troops in a region. We want to do a quick simulation of the possible battle that will occur there. But what size do we assign to each side? The expected force flows are in general too small, because if the battle occurs at all there will probably be bigger forces involved than the fractions generated by the flow algorithm. So we have to guess what the actual size would be in order to predict an outcome.

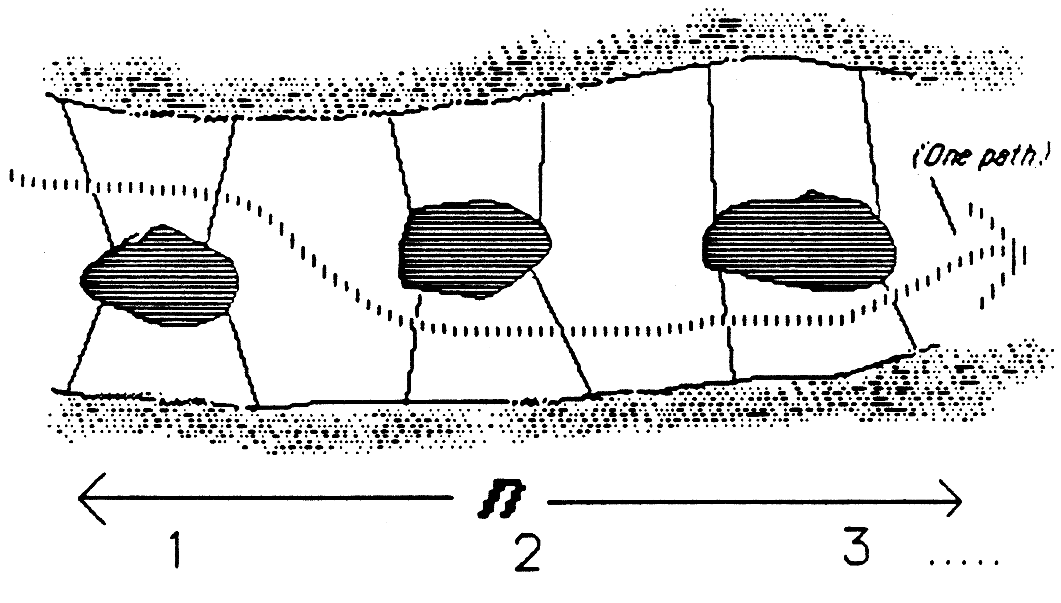

Figure 4 A Potentially Combinatorially Explosive Situation

3.1 Finding Flows

Force flow occurs from *sources* to *sinks*, corresponding to force sources and objectives. Inside the flow algorithm, both of these are modeled as portals, but peculiar portals with just one side. Real portals are straight lines separating regions of different widths, as explained in Section 2.1. The two sides (arbitrarily called `HERE` and `THERE`) are pointed to by the data structure for the portal. Flows are positive in the direction `HERE` to `THERE`. A source is represented by an artificial portal with `HERE=#F`,[6] a sink by one with `THERE=#F`. Flows through these artificial portals are never negative. Portals have Cartesian coordinates, which correspond to the midpoint of a real portal, or the hypothesized location of a source or sink.

Each portal has a *capacity*, which represents the maximum number of troops that can flow through it during the time frame we are reasoning about. The capacity may be thought of as depending on the minimum of the widths of the regions on either side of the portal. As the flow algorithm progresses, its probes through the graph may reveal global capacity limitations that prevent all of the capacity of a portal being used. As forces are backed out of tight squeezes, the capacities in the constrained directions are reduced. At any given time, there is an "out-capacity" from a region through a portal, which is in general a table giving the current capacity for each force source.

A sink portal represents an objective, a point to be captured at the coordinates of that portal. Its capacity stands for the number of troops required to carry out the mission at that point. If there is just one objective, and we want all the force to go there, we can set the capacity very high. The force-flow algorithm is given a set of sources and sinks, plus some information about the time frame of the battle to be predicted. Each sink can have an associated *latency*, specified in a sink-latency table, which is the time within which that objective must be reached by the forces being pushed through the map.

The algorithm operates by pushing force through the map as fairly as possible while delivering the maximum amount of force to each sink. The schedules are represented as tables on each portal showing the forces that have arrived from each source, and when they arrived. Portals start off directionless, meaning that force can flow through them in either direction. But once a portal has been used in one direction, no force can flow through it in the opposite direction. Hence all portals that are used get assigned a `DIRECTION` in which all flows occur.

We divide the forces as "fairly" as possible among alternative routes to avoid overlooking possibilities. *We do* not *assume that the forces will actually be divided into small pieces.*[7] If forces are found divided into small pieces en route to an objective, that is simply an indication that it is hard to predict exactly which

[6] `#F` is Lisp false, which is eq to `NIL` in Common Lisp. Similarly, Lisp true will be written `#T`, which is usually written as `T` in Common Lisp.

[7] It is true that such tactics favored by some military theorists, but we don't want to wire such assumptions into our algorithms.

route they'll take, unless the small pieces actually fit snugly into their respective channels, in which case the troops have been forced to divide into small pieces simply to get through. It will be seen that the output of this algorithm makes all sorts of information explicit for use by later stages of reasoning, but does not by itself print out a useful plan.

It is not enough to find a route to a sink; the route must be traversable within a reasonable time. In war, all forces must arrive simultaneously to have any effect. As we construct a schedule, we keep track of when force bundles arrive at portals. If a force element gets so far off track that it can't get to a sink in the time allowed, then its meanderings are canceled and it is rerouted.

Delay due to terrain conditions is modeled by specifying a *travel speed* parameter with every region, which determines how long a force will take to go one kilometer through it. In addition to the obstacles presented by the terrain, there are also obstacles presented by enemy troops.[8] If enemy troops are located in a region, then any force trying to move through it will be attrited and delayed. These factors are determined by running a "simulation" of the battle that would ensue if Red and Blue forces both arrived at this region as promised. More on this in Section 3.3.

Were it not for the "fairness" and maximum-latency requirements, and the need to model attrition, we could simply use a standard max-flow algorithm to arrive at our result. (Even 1979) (Such algorithms assume steady flows, but a maximal one-shot schedule, repeated indefinitely, is equivalent to a steady flow.) But, with the complications, what we are really doing is simulating various possible battle scenarios in parallel. The algorithm works to the extent that all scenarios get included. Formalizing this requirement is not easy. For now, we focus on ways of extracting important data from flow graphs, and looking for insights into how and why it works, and for ways to improve it. A quasi-formal analysis of the algorithm is given in the next section.

The algorithm operates by maintaining three queues of regions, called `FORWARD`, `BACKWARD`, and `SIDEWAYS`. The first queue contains regions with force on their in-portals that has not yet flowed out. The second contains regions with force coming in that has no way to flow out. The third contains regions with force coming in and only unpromising ways to flow out. A portal is unpromising if it doesn't point toward any sink. (More precisely: if, for every in-portal, the line from its midpoint to the midpoint of the candidate points away from every sink.)

The top level of the algorithm loops through these three queues until they are empty. It starts with `FORWARD` consisting of just the regions with sources, in decreasing order of force level. It loops through this queue, pulling a region off, propagating forces to neighboring regions, and repeating. When force is propagated to a neighboring region, that region goes onto the forward queue. When force reaches a sink,

[8] In case you are confused, in this context "enemy" means, "opposing the side moving the forces." This side will usually be Red, so "enemy" usually means "Blue."

it is recorded there. Hence, in the best case, this loop succeeds in solving the whole problem, propagating everything to the given sinks.

The forward-propagation algorithm is fairly complex because of the fairness requirement. When it examines a region, it can sort its portals into three categories: ins, outs, and neutrals. Recall that all portals are initially neutral, but a region can't get on the FORWARD queue without having force coming through one of its portals toward it, and all such get classified as in-portals. All others are candidates to push force through. However, not all in-portals can direct flow to all candidate out-portals, for three reasons: (1) During the forward push, we avoid portals that go in strange directions; (2) we never allow force to go so far astray that it would be impossible to get to a sink before its maximum-latency period; (3) we forbid force to flow back to a region that it came from (i.e., a loop in the flow graph).

There is a subroutine to do this analysis, which returns a table showing which in-portals can feed which out-portals. The forward propagator then calls a detailed flow model to figure out exactly how much of the force coming can go out, and where to. If it can't all flow out, then the resulting "clog" is handled by backing some of the force out of this region and sending it another way, i.e., by putting the region on the BACKWARD or SIDEWAYS queue. More on this below.

The model of force flow through a region is entirely controllable by the user, because it is a Lisp program bound to the global variable REGION-FLOW-MODEL*. In section 3.3, I will talk about how the user might make use of this feature. In what follows, I will describe the default model supplied by the system.

The model is complicated by two factors: the desire to divide force fairly and thoroughly, and the need to model the attrition of the forces marching through this region. Let's look at the first of these first, the problem of dividing up forces fairly among several out-portals, given that they have entered a region from several in-portals, but not every in-portal is eligible to feed every out-portal. We solve this dilemma by dividing the force at each in-portal proportionally to the capacities of the out-portals. If the resulting distribution causes the capacity of some out-portal to be exceeded, the algorithm trims all the portals feeding it by just enough to put the portal right at its capacity. Now it looks for out-portals with excess capacity, and if it finds one, it sends the surpluses among eligible in-portals there. Then it goes back to remove clogs. The algorithm repeats until either there are no more clogs, or all the out-portals are full. On each iteration, this algorithm leaves one more out-portal full, so it is sure to terminate.

This little algorithm is the most complicated piece of code in the system. One reason for its complexity is that after backtracking, the capacities of portals are reset, as explained later in this section. Initially every portal has a capacity $t \times c$ in either direction, where t is the maximum time alloted to force movements (the "time frame"), and c is the capacity of the portal per unit time. After force has been backed out of a portal, its capacity in that direction is reset to a table of $< source, number >$ pairs, which specifies the maximum force for each *source* that can pass in that direction. Hence the flow model must allocate forces to a backed-through portal in a different way, because increments from different sources are competing for independent pieces of capacity.

Putting attrition into the model raises some subtle issues. The main one was mentioned above. An army's progress through a region will be dramatically slowed if it is under attack. Indeed, it might not make it through an otherwise easily traversable area. But figuring out exactly how much damage can be expected requires knowing what the sizes of the two contending forces are, and this is information that the flow algorithm does not explicitly give us. The current system takes the average of the expected force in the region and the total force everywhere as the number. Hence if 40% of the force is expected in an area, we will assume 70% is there for purposes of modeling battles. This estimation technique is rather crude, and does not even bother to see if 70% of the total force would fit in the current region. I have not bothered to correct this problem, because it is symptomatic of a deeper problem, that the system does not currently calculate the actual maximum amount of force that could be brought to bear on a region. To do this would require running a standard maximum-flow algorithm (Even 1979), and this is definitely a direction worth pursuing. For now, the crude estimator works okay.

Once we have estimated the attacker's force by this method, we have to estimate the defender's force, where the defender is trying to prevent flow through this region. There are two models, static and dynamic. The static model estimates the force based on the locations of forces now. The dynamic model estimates them based on predicted flows already arrived at. In this case, the number we want is the amount of force that is expected to flow through this region by the time the battle occurs, an amount that is easily calculated. Currently, and somewhat arbitrarily, we do *not* amplify these figures by averaging them with the total defending force. The rationale is that there is already an exaggeration due to the fact that we count all the force that has ever been through this region, which will give the defender the same force to use in many different locations.

With our attacking and defending forces arrayed, we can proceed to the actual attrition calculation. The current model is a throwaway that could no doubt be improved upon easily. It is controlled by three parameters: **DEFENDER-KILL-RATIO***, number of attackers killed per defender; **HARASSMENT-RATIO***, the attrition due to artillery fire from a single region that can view this one (a number between 0 and 1); and **DEFENDER-KILL-TIME***, the time for a superior attacking force to wipe out each defender. If **DEFENDER-KILL-RATIO*** × the number of defenders is greater than the number of attackers, then it is assumed the offense will lose, and the region is treated as if it were a dead end. (I.e., it goes on the backtrack queue.)

Otherwise, let V be the number of enemy-occupied regions that can fire upon this one. Define $W =$ $\text{HARASSMENT-RATIO*}^{V}$. Then the quantity

$$A = W \times (1 - \frac{\text{DEFENDER-KILL-RATIO*} \times \textit{defenders}}{\textit{adjusted attackers}})$$

is between 0 and 1, and represents the fraction of the attacking force that will get through. It is multiplied times each packet of force sent from an i-portal to an out-portal. In addition, there is a substantial delay when moving through a region under fire. The travel time under fire is modeled as

$$\frac{\text{DEFENDER-KILL-TIME*} \times \textit{defenders} \times \textit{tt}}{W}$$

where *tt* is the unopposed travel time. That is, the time required to get through the region is the time required to wipe out the defenders, increased by a factor dependent on the amount of hostile fire. No prizes are awarded for improving on this model.

If the forward-push phase does not succeed in getting all of the force from the sources to the sinks (or to its final reward), the algorithm will detect this fact when there is a clog at a particular region: more force comes in than can leave. An absolute clog occurs when all portals to that region have been classified as in or out, and there is no further capacity to be exploited on the out side. A transient clog occurs when there are potential out-portals that are unused because they point away from all the given sinks. Such transient clogs can be relieved by allowing forces to flow in such apparently counterproductive directions. Regions with this kind of clog are queued on the `SIDEWAYS` queue. Regions with absolute clogs go on the `BACKWARD` queue; such a clog is relieved by backing some of the flow out of the region, and sending it a different way.

We have a choice about whether to propagate sideways or backwards first. The current algorithm tries backward propagation first. That is, when clogs develop it tries redistributing the flow along existing channels before striking out in new directions.

Redistribution depends on a field called `STILLHERE` that is associated with every portal, and updated by every forward propagation. This field is the amount of force that was stranded at the portal. (Actually, we have a table of source-force pairs, and the `STILLHERE`s are associated with each table entry.) To undo the clog at a portal, the algorithm must send enough force back to the region F on the other side of the portal to make `STILLHERE` zero. Force never sits in regions, but on portals, so this operation requires moving force back to various in-portals of F. See Figure 5, where the portal in question runs from F to B, the region that was found on the `BACKWARD` queue. (If there is no such F, i.e., if the clogged portal is a source, then we have detected a global clog, and we just leave it.) Because of the complex attrition calculation, it is impossible to partially undo the region flow, so we adopt the following stratagem. The backward propagator simply resets the capacity of the clogged portal from F, and then puts F back on the forward queue. When F is found on the queue, the battle there will be rerun, but this time fewer troops will be able to make it through the clogged portal. They will either go another direction, or cause a clog at F that will in turn be backed up. Because `REGION-FLOW-MODEL*` is an arbitrary user procedure, it is possible that a different amount of force will come through B during the rerun, but the default models will send exactly the same amount through, and YATS will avoid rerunning B.

Once the inversion has been accomplished for all the in-portals of a clogged region, the other sides of those portals will have been placed on the `FORWARD` queue. The algorithm then tries to empty that queue before backing up another region, pushing the stuff forward by alternative routes.

As I have mentioned briefly a few times, when the backtracker resets the capacity of one of its in-portals, it actually stores a table of $<$ *source*, *capacity* $>$ pairs. Later propagation through this portal will be limited on a source-by-source basis. For example, suppose that 5 units from two sources come into a portal. Of the 5 from source 1, all 5 continue on to some destination, but all 5 from source 2 are "still here." YATS resets

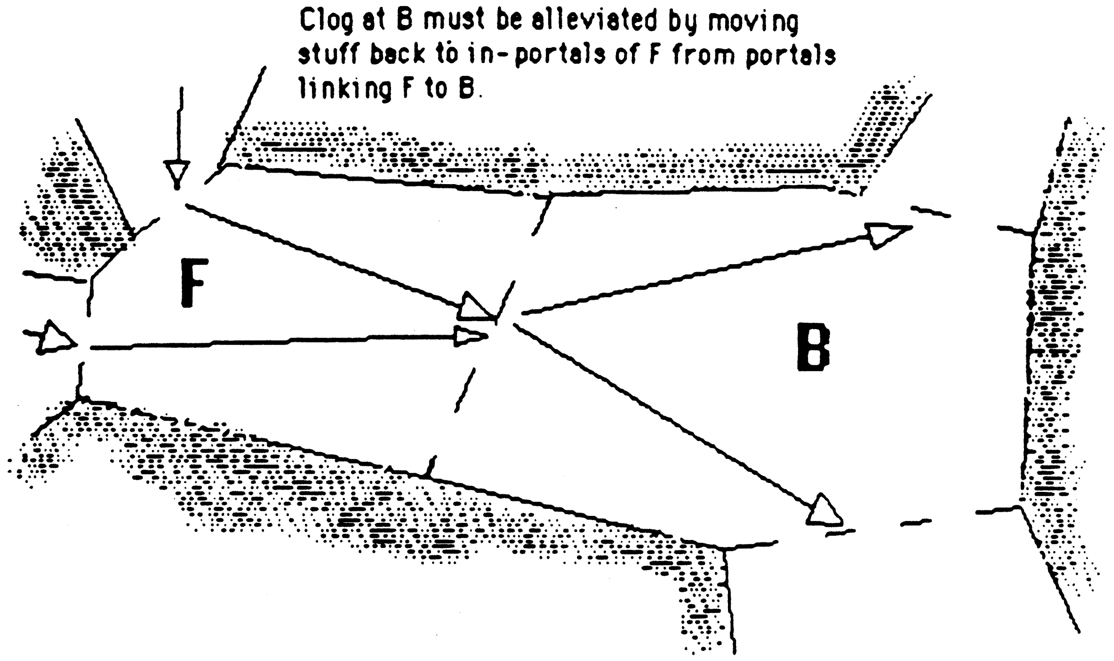
Clog at B must be alleviated by moving stuff back to in-portals of F from portals linking F to B.
F
B

Figure 5 Backing out of a Region

the capacity to $(<\textit{source 1}, 5>, <\textit{source 2}, 0>)$. Later propagations involving source 1 will allow 5 units through, but propagations involving source 2 will have to go elsewhere. If we did not keep track of things this tightly, source-2 forces would repeatedly attempt to go through this portal, and repeatedly bog down somewhere down the line (say, because source 2 is further away than source 1), then get retracted.

The backward-propagation algorithm has one other trick that it tries. It checks to see if one of its in-portals is *reversible*, that is, can be switched to an out-portal. This will be true if all the force that has flowed through the portal is still here, so sending the excess back will zero it out. If there is some reversible portal, the program picks one and reverses it, putting this region back on the forward queue. The reversal may make it possible for force to flow out now; in any case, reversing one portal cannot make things worse than reversing all of them.

In summary, there are two distinct operations the backward propagator can do: *unclogging* a region, which means reversing a single in-portal, converting it to an out-portal, and putting the region back on the forward queue; and *stoppering* a region, which means restricting the capacities of all its in-portals and putting the region on the other side of each in-portal back on the forward queue.

I have as yet not described the SIDEWAYS queue. Whenever the forward-push algorithm finds flow clogged at a region with a candidate out-portal that points in a strange direction, it puts the region on the sideways queue instead of the backwards queue. This queue is examined when the other two queues are empty. Exactly the same processing is done as in the ordinary forwards case, except that portals in strange directions are treated the same as all the other portals. That is, material is pushed through to the other sides of those portals, and if it can't all get through, the region is added to the backwards queue. This way of doing things ensures that flow proceeds as much as possible in the general direction of the sinks, but when all else fails the algorithm takes one step in an odd direction, and pushes forward from there. Once a region has experienced sideways propagation, all propagation out of that region from then on is done the same way.

3.2 Complexity of the Algorithm

This algorithm is quite tricky to analyze rigorously, although it is definitely fast enough in practice. On moderate-sized graphs, it typically finishes in less than 10 seconds. There are several other places in the system that require speeding up, but not this one.

However, it is worth pausing to analyze its behavior more carefully. Many algorithms that traverse graphs can have exponential worst-case behavior, and we need to make sure that this one will run in polynomial time, $O(n^3)$ or better, to be precise. I will analyze a slightly idealized version of the algorithm, and argue that the actual algorithm will not differ in practice. I start with two assumptions:

1 The number of portals at a region is bounded by a small constant, e.g., 8. This is entirely reasonable for the sort of planar graphs we are concerned with.
2 The region flow model is *monotonic* in the sense that more force coming into in-portals does not result in less force going out any out-portal. This certainly seems reasonable, and holds for the default flow models.

The flow algorithm starts with an undirected graph, and ends up imposing a DAG on it, by classifying relevant portals as in or out. (Unclassified portals have nothing flowing across them, and presumably are useless as routes from force sources to objectives.) Part of the complexity of the algorithm is due to this DAG structure not being given to start with. But note that the algorithm almost never changes its mind about the direction of a portal; the only exception is the backward-flow routine's ability to reverse just one portal. Such a reversal occurs at most once per portal, because the capacities are changed to make this impossible again. Hence the algorithm tends to generate a DAG structure quickly, and then adjust the flows within it.

The algorithm has the following structure:

```
(loop
   (loop
      (loop forward-propagate from one region
               ; (Can add to any queue)
       until FORWARD queue empty)
      backward-propagate from one region
         ; (Can add to forward queue)
    until BACKWARD queue empty)
   sideways-propagate from one region
    ; (Can add to forward or backward queue)
until SIDEWAYS queue empty)
```

Obviously, the speed of the innermost loop, the forward-propagation wave, is crucial. If some oracle gave us the DAG structure at the outset, then this loop could be fast. For one thing, the test for cycles in the flow could be eliminated. More fundamentally, we could avoid propagating from each region more than once. Consider Figure 4, in which the flow is from left to right. Suppose, during the forward-flow phase, that a region is never examined before all the regions to its left have been examined. Then each region will be examined exactly once, where by "examination" I mean performing the miniature battle scenarios described above.

The problem is that we won't know the DAG until the algorithm is done. To circumvent this, we must arrange for the system to "notify itself" of the DAG structure as soon as possible. Define the *rank* of a region as follows: the rank is 0 if no regions immediately precede it (i.e., it contains a source and is not fed by any other region); otherwise, the rank is 1+ the greatest rank of all regions that immediately precede it in the DAG. We keep track of an "estimated rank" for each region, initialized to 0, and set to 1+ the estimated rank of any region that precedes it when this precedence is discovered, if the new estimate would be bigger than the old. We then arrange for the forward-propagation algorithm to order regions on the queue by estimated rank. A given region can have its rank estimate changed at most r times, where r is the actual rank; the estimate starts at 0, and eventually attains the actual rank. From change to change, a region R is visited at most once, because if visits to two predecessor regions cause it to be put on the queue, and both have smaller estimated rank than R, then both will be handled before R is. The rank is bounded by n, the number of regions. Hence every region can be visited at most n times before its true rank is known, after which it will be visited at most one more time. So we have this theorem:

Theorem 1 : The rank-estimation version of the forward-sweep algorithm runs in time $O(n^3)$.
Proof: The argument was given above. There are n regions, each is visited at most n times, and each visit takes $O(n)$ time units. This time depends on the loop-detection calculation that is done to prevent the DAG from acquiring a cycle; and on the rank-estimate updates for successor regions. One subtlety is that we must take into account the time to keep the queue sorted. Every time a region is added to the queue, it must be inserted in the proper place. Using standard heap-sort techniques, this can be done in time $O(\log n)$, which is asymptotically insignificant. QED

The actual algorithm does not try to estimate ranks. Instead, it simply sorts regions by average arrival times of the forces waiting to be propagated through them. In practice, this has roughly the same effect. Furthermore, the estimates given above were quite generous. It is extremely rare for a region to be visited more than once during a single forward sweep, so this part of the algorithm is in practice $O(n^2)$.

We have yet to include the operations of the other two queues. Backward propagation would be easy to analyze if it weren't for the possibility of single-portal reversal, which moves an apparently clogged region back to the forward queue. This reversal will lower the estimated ranks of some regions and raise the estimated ranks of others. (A sweep algorithm to revise the estimates will take $O(n)$ time.) Such reversals are the *only* way that rank estimates can change during backward propagation. Hence during the ensuing forward propagation, there is no need to revise rank estimates or do cycle detection. The existing DAG structure can be used as is. So (although we don't bother to do this in the actual implementation) a forward sweep after a stoppering or an unclogging can employ a simpler algorithm, which sees each region at most once, sorting regions by existing rank estimates. We will call this a *settling* forward sweep, versus a *pioneering* forward sweep, which can transform neutral portals into DAG arcs. The time for a settling sweep would be linear except for the priority-queue manipulation, which takes time $O(\log n)$. Hence a settling sweep takes time $O(n \log n)$, while a pioneering one takes time $O(n^3)$, as argued above.

To analyze backward propagation further, I need to make two further idealizations: that all unclogging is done before all stoppering; and that regions are stoppered in decreasing order of estimated rank. (The actual structure will be described later.) In other words, we will break the backward queue into two pieces: an unclogging queue and a stoppering queue. Once all unclogging is done, stoppering can never give rise to any more of it.

Lemma 2 : After a wave of unclogging, further stoppering and settling cannot be the occasion of more unclogging.

Proof: Settling does not change the current tentative DAG structure. The only way unclogging can occur is when a clog develops at a region with an in-portal such that a certain amount of force has arrived at that portal, and none of it has propagated to an out-portal of the region. But if this is an in-portal, it must have been so labeled during the previous pioneering phase; that is, something must have come through it. That something must have been propagated through to an out-portal, or the resulting clog would have been unclogged already. Hence the in-portal is ineligible for unclogging after this round of settling. QED

The structure of all but the outermost loop of our idealized algorithm is now:

```
(loop ... pioneering sweep through FORWARD queue ...)
(loop through unclogging queue, choosing one portal in
          region to reverse, and putting regions on
          FORWARD queue)
(loop
   (loop ... settling sweep through FORWARD queue ...)

 until Both FORWARD and BACKWARD queues are empty
   Remove a region from the BACKWARD queue and stopper
      it,adding its feeders to FORWARD queue)
```

The unclogging step takes time $O(n)$. The last loop's time depends on how many times it is executed, which depends on how many times a region can get on the stoppering queue.

Theorem 3 : In the algorithm above, a region can be stoppered at most once.

Proof: Let R be the first region to be stoppered twice. The second time, there must be some R' such that there is an out-portal from R to R', and such that R' was just stoppered, causing R to be put on the FORWARD queue, whence it went to the stopper queue. By hypothesis, R' was never stoppered before, and R was. Hence the portal between them is in its pristine state, and must be full to capacity with flow from R. If any of this flow were still here, then R' would have been on the stopper queue, and would have been stoppered earlier, before R, because regions are removed from the stopper queue in decreasing estimated-rank order. Hence, none of the flow is still on the portal between R and R'. But in that case, stoppering R' will not cause R to go on the FORWARD queue. This contradiction refutes the assumption that R is stoppered twice. QED

Putting all this together yields:

Theorem 4 : The forward-backward portion of the algorithm runs in time $O(n^3)$.
Proof: The pioneering sweep takes time $O(n^3)$. The unclogging phase takes time $O(n)$. The stoppering loop takes time $O(n \log n)$, and it is executed $O(n)$ times, once per region stoppered, for a total of $O(n^2 \log n)$. Hence unclogging and stoppering are dominated by pioneering, and the net complexity is $O(n^3)$. QED

Finally, we have to include the time spent on sideways propagation. Each region can be "sideways propagated" at most once, so the maximum number of times the outer loop can be executed is n, giving a net complexity of $O(n^4)$ for the whole algorithm. But this is absurdly pessimistic. The whole point of sideways propagation is to avoid exploring irrelevant parts of the map, but according to this analysis we would do better to turn it off and always explore in any direction — the result would be an $O(n^3)$ algorithm. Intuitively, the reason sideways propagation helps is that it allows us to focus on a subset of the regions of size n_0, and if they don't suffice expand to another subset n_1, and so forth, so that the actual complexity of the algorithm is Σn_i^3, which is never greater than n^3. Unfortunately, this argument is not formally correct, because the subsets are not disjoint. A sideways propagation will take one step in a funny direction, but subsequent steps may well merge back in with previously found flows. To deal with this phenomenon, the algorithm would have to be fiddled further so that there was no hard and fast distinction between pioneering and settling. Every region would be classified as explored or unexplored. At unexplored regions, a cycle-detection test would be necessary before sending flow out of neutral portals. At explored regions, new material would be sent over familiar channels. During a single sideways propagation, material would obviously stay in explored territory once it got there. Hence, if we let n_i be the number of regions explored on the i'th pioneering propagation (letting the zero'th be the one before the sideways queue is examined for the first time), the complexity of each phase would be $O(n \times n_i^2 + n \log n)$. The first term in this sum comes from the pioneering phase (each new region is seen at most n_i times, and each cycle-detection take $O(n)$ time units); the second term comes from the settling phase. The second term is negligible, so, if there are k sideways phases, the net complexity is

$$O(n\Sigma_{i=0}^{k}(n_i^2)) \quad \text{where} \Sigma_{i=0}^{k} n_i \leq n$$

which is the same as $O(n^3)$.

This argument is not exactly rigorous. I will not bother to make it so, because its conclusion is, I believe, overly pessimistic, even for the idealized version of the algorithm. In practice, the actual algorithm has time complexity that is closer to $O(n^2)$, as explained. It is probably possible to prove a tighter bound than the n^3 obtained above, but it seems much more important to get some experience with the algorithm, and understand better what it computes, if anything.

Let me summarize how the idealized algorithm differs from the actual implementation:

1 The real algorithm does not keep queues sorted by estimated rank. The **FORWARD** queue is kept sorted by arrival time of forces; this is probably *better* in practice than the idealization. The **BACKWARD** queue is simply managed LIFO, that is, as a stack. Hence a region can be stoppered more than once. This appears not to matter much.

2 The real algorithm interleaves unclogging and stoppering. As soon as a region is unclogged, it goes back on the forward queue. Unclogging is rare enough that this is unimportant.

3.3 Attrition Models

The current system provides a very crude sort of attrition model. This is the main "hook" the user has for altering the system, so I will discuss in some detail how the user can go about using it. To write a new attrition model requires some knowledge of the way YATS represents regions and other data structures. Indeed, doing anything sophisticated will require careful study of the program listing.

An attrition model is the value of the variable `REGION-FLOW-MODEL*`, which takes a set of inputs to a region, alters the outputs, and returns a list of regions on the other sides of the affected outputs (from which flow should proceed). Each portal in a region contains a table of *portflows*, each a four-tuple $< source, flow, stillhere, latency >$. The *source* is the region the flow started in. The *flow* is the amount that reached this portal; *stillhere* is the amount that could not proceed from here because of capacity or time constraints; *latency* is the average time of arrival of the force. (Different fragments of the total *flow* have arrived at different times.)

The procedure that is the value of `REGION-FLOW-MODEL*` takes two arguments: the region to be flowed through, and list of *transpecs*, each a tuple $< in, pf, outs >$, where *in* is an in-portal to the region, *pf* is a portflow on that portal, and *outs* is a list of candidate out-portals that YATS has decided are legitimate candidates for force to flow to from *in*. (The user's procedure can ignore this.) The procedure must actually clobber the portflows on various out-portals of the region. A low-level procedure for doing this is provided, called

`(CLOBBER-PORTFLOW` *source region in out* d_i d_o Δt`)`

which alters the flow for the *source* within *region* from portal *in* to portal *out*, subtracting d_i from the flow at *in*, and adding d_o to the flow at *out*. (Presumably $d_o \leq d_i$, with equality if there is no attrition.) The Δt argument is the travel time taken to cross the region. The procedure

`(TRAVEL-TIME` *in out region*`)`

will give the raw time, but the user's model may decide the time under the circumstances is longer (or shorter, for that matter).

If the user's procedure uses `CLOBBER-PORTFLOW`, then all low-level bookkeeping is the user's responsibility. In some cases, the user is content to let YATS direct traffic, while his code computes the fatalities and delays due to enemy action. The procedure

`(COMPUTE-TRANSPEC-DISTRIBUTION` *region transpecs*`)`

returns two values: a Boolean telling whether a clog developed; and a list of "io links" spelling out the traffic flow. An iolink is a data structure encoding a tuple $< in, ipf, out, opf, amt >$, specifying a "microflow" of amount *amt* from portflow *ipf* on portal *in* to portflow *opf* on portal *out*. The user needn't inspect these at all, but merely pass a list of iolinks L to the procedure

`(IMPOSE-TRANSPEC-DISTRIBUTION` *region* L F_a F_d`)`

where *region* is the region the flow is going through, and F_a and F_d are attrition and delay functions. F_a takes a force level and returns the amount of it that will get through the region; F_d takes an unopposed travel time, and returns the actual time it will take to traverse the region. `IMPOSE-TRANSPEC-DISTRIBUTION` returns a list of regions, suitable as the return value of `REGION-FLOW-MODEL*`.

The default attrition models barely scratch the surface of what's possible. There is ample scope for further research here in improving on them. For example, the current system computes the firepower brought to bear on a force in a region as a function of the opposing forces that can view that region. There are two versions, static and dynamic. The static version takes the force array that is currently hypothesized by the user. The dynamic version takes a time-varying force array that is derived from a predicted enemy troop flow. (See the next section.) In the second version, the total opposing force is computed by checking to see how much of the enemy's force can flow into a dominating region from time 0 to the time of region transit. But suppose that friendly forces can counterattack against that force and wipe it out. Then it would be advisable to forces transiting the region to wait for that to occur, unless of course they are the forces that are being counted on to suppress enemy fire! See Figure 6

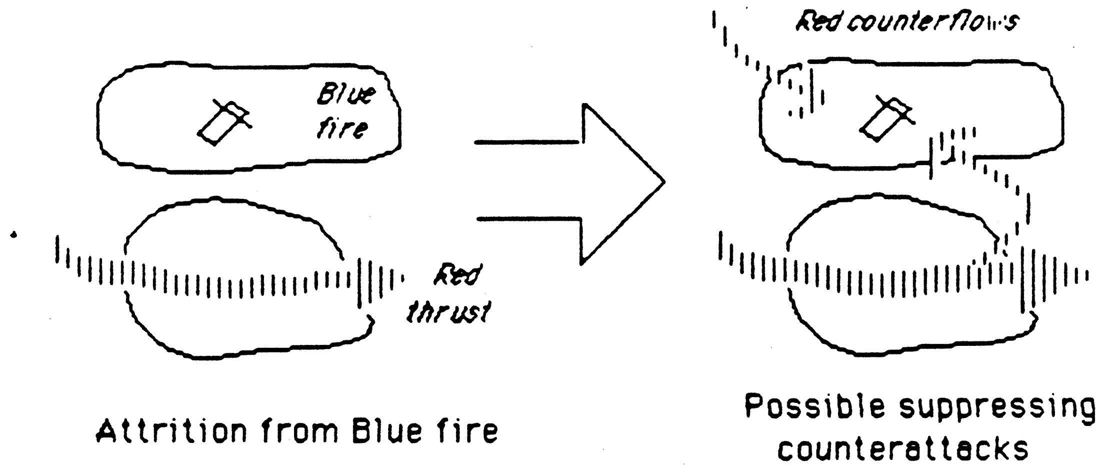

Figure 6 Complex Attrition Considerations

The current system neglects most such subtleties.

3.4 Counterplanning

The force-movement algorithm is not sufficient by itself to generate a complete picture of possible Red plans. There is more to military planning than figuring out possible lines of march. Once these have been found, the program goes back and finds possible Blue counterplans, and revises Red's plans accordingly.

The current algorithm focuses on Blue's opportunities to disrupt Red's line of march by use of firepower. After a preliminary Red flow has been found, YATS looks for sub-paths in the Red flow such that once Red forces have committed to one of them, the Blue forces will have time to maneuver so as to attack it. The areas Blue can use for this purpose become objectives in a Blue flow. Red must cope with anticipated Blue flows along the resulting lines, so YATS runs the flow algorithm again with a dynamic attrition model reflecting the Blue counterattack.

An intuitive example will make this clearer. In figure 7, once Red forces have entered region 1, they have tipped their hand. They must be on their way to region 5 or region 3. After that they have further choice points, but locally they have committed themselves to heading in a predictable direction for a while. Hence the Blue force will have ample time to take up positions in region 4, overlooking 3 and 5, and bring damaging fire to bear on the Red forces. The Red flow must be run again, this time through the gauntlet represented by this attack.

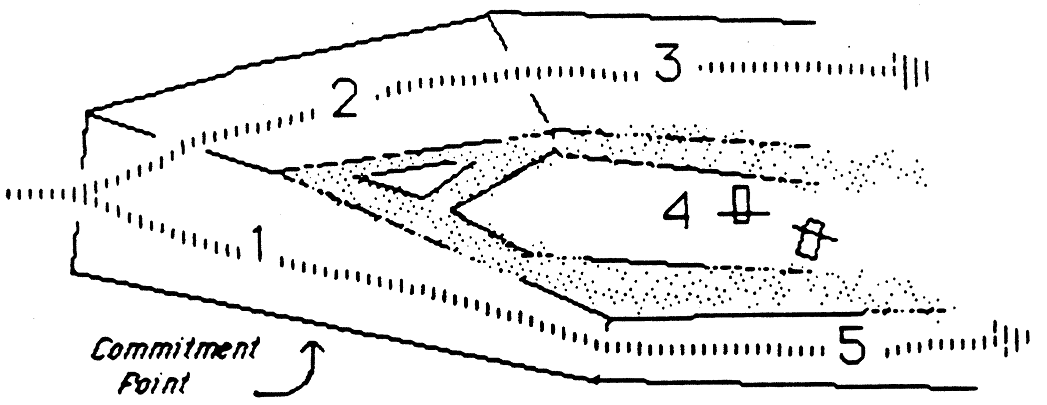

Figure 7 Red Force Commitment Point

The algorithm finds these possible Blue counterattacks by the following algorithm: For every region through which Red will march, the view links from potential attackers are examined. For region A to be a good attack site on region R, A must attack a significant fraction of the total flow (we multiply the fraction of R that A commands times the fraction of the total flow that goes through R); A must be significantly elevated over R; and it must not be the case that Red is already planning to march through A on its way to

R. All such attack sites are collected. The same attack site may serve to attack several regions, called its *target*. (A may be good even for attacking itself, if Red is planning to march through it.)

Next, for each attack site, YATS finds the *commitment point* for that attack site, the region R such that as soon as Red enters R, he has committed to traveling through the target of the attack site. To find that point, we use a marker-propagation algorithm. First, all of the "predecessors" of the target are marked, that is, all the regions Red goes through on his way to some element of the target. Each such predecessor is a candidate for commitment point. Consider a particular predecessor C. If all the force flowing out of C ultimately flows to some element of the target, then C is a commitment point. It is rather hard to verify that this is the case. We opt for an approximate algorithm. We mark all of C's successors that are predecessors of the target, i.e., all the regions Red can get to after going through C, still on the way to the target. So every region is either (a) marked as a successor of C and a predecessor of the target, (b) marked as a predecessor of the target, but not a successor of C, or (c) unmarked. The "leakage" for C is the sum of the local leakages for all the regions in category (a). Intuitively, it is the amount that escapes from the area Blue is trying to confine Red to. C passes the test if the flow into C plus the leakage for C is $\leq 1.2\times$ the total flow into the target.

Total leakage was defined in terms of local leakage. The local leakage for such a region is defined as

$$2 \times (inleak \times (1 - \frac{inleak}{intot}) + outleak \times (1 - \frac{outleak}{outtot}))$$

where *intot* and *outtot* are the total flows into and out of the region, and *inleak* and *outleak* are the total flows into this region from unmarked regions and out of this region to unmarked regions, respectively. (Fractions of the form 0/0 here are taken as 0.) The expression $2l(1 - l/t)$ where $0 \leq l \leq t$ reaches a maximum when $l = t/2$, i.e., when half the flow leaks away. (If all of it leaks away, we are presumably at the target itself, and that doesn't count as leakage.)

Some attack sites have no useful commitment point. If a site does, then we can get a good estimate of the time Blue has to get to it. We just subtract the arrival time at the commitment point from the average exit time from the target. The resulting time frames will vary from attack site to attack site. When YATS runs the Blue counterflow, it will make each useful attack site an objective, and give it the time frame computed in this way. Some places would be great to attack from, but they can't be reached in the time allotted. During this Blue counterflow computation, YATS uses a dynamic attrition model based on the preliminary Red flow. The result will be a prediction of how much force Blue will be able to bring to bear on good attack sites for Red's line of march.

Finally, YATS runs Red's flow again, this time with a dynamic attrition model based on the Blue response. The result will be a more realistic reflection of Red's capabilities, given the likelihood of opposition. In many cases, the net result is that Red will do nothing, because all avenues of attack will be blocked off.

This algorithm, while it draws some interesting pictures on the screen, must be seen as a prototype at best. It suffers from several inadequacies, some easy to correct, some hard.

The most obvious inadequacy, mentioned at the beginning of this paper, is that it is oriented entirely toward Red attack. If Blue is planning an attack, the current algorithm cannot reason about Red's possible defenses.

Currently, no attempt is made for Red to suppress Blue fire by attacking some of Blue's objectives. Conceptually, we should really run the flow algorithm for each side repeatedly, "until it converges," that is, until each side has a flow that, as it were, represents the best response to the other's flow. Unfortunately, it is not clear how to make this work. A straightforward implementation could easily oscillate, shifting back and forth among various plans in response to enemy counterplans, which themselves shift, out of phase. One idea is to average together the flows over several oscillations, and call that the answer.

The commitment-point algorithm has a flaw. Suppose that Blue has two attack sites to choose from, A_1 and A_2, with commitment points C_1 and C_2 so close to A_1 and A_2 respectively that there is no time to get to either of them. Let C_0 be the commitment point for the union of the targets of A_1 and A_2. There might be a staging area A_0 such that (a) once Red enters C_0, there is time to get to A_0 before Red has arrived beyond reach of either A_1 or A_2; and (b) after Blue gets to A_0, he is so close to A_1 and A_2 that he has time to get to the appropriate one once Red's commitment becomes clear. Under these circumstances, Red should assume that he will meet with opposition should he head for C_0.

4 The Implementation

I have implemented a preliminary version of the system, YATS 1.0, written in Nisp (McDermott 1988) for the Texas Instruments Explorer II. When the program is running, the user sees the screen shown in Figure 8. A terrain map appears in the main window. A menu appears to the left. A Lisp "listener" appears at the bottom. Most interaction is via the menu. Under menu control, the user can input a terrain map, have it broken into regions, order a line-of-sight analysis, place forces and objectives on the map, and request a tactical situation assessment. Red forces, Blue forces, and Red objectives are displayed on the map as boxes with serrated edges, boxes with smooth edges, and flags, respectively. The user can have various sorts of information displayed under mouse control, including graphical displays of the tactical significance of various regions and pixels on the map. This display is generated from the current *tactical scenario* which is generated when a situation assessment is requested, and retained until overridden.

The menu entries are in this order:

Clear	Clear the screen and discard the current scenario.
Load map	Prompts for file name to load map from. (Format given below.) The raw map contains information about terrain elevation and traversability. Untraversable areas are displayed in black. Other areas have altitude proportional to their whiteness.
Find regions	The map is divided into regions, as explained in Section 2.1. Region boundaries are displayed as heavy white lines.

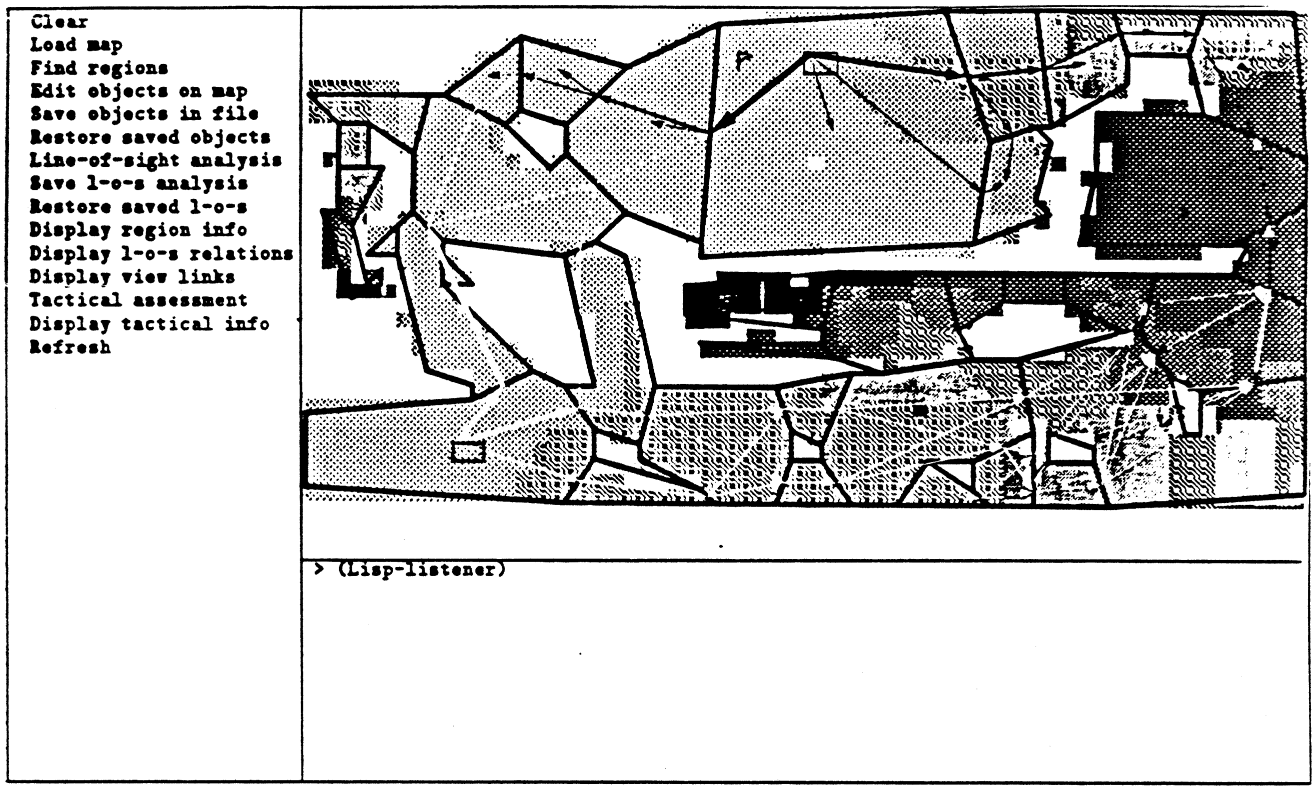

Figure 8 YATS Screen Layout

`Edit objects on map`	The user is prompted to click on various map pixels. If a pixel contains an object already, he is invited to edit its characteristics, changing its type, attributes, or location. Otherwise, he is invited to create a new object.
`Save objects in file`	Prompts for file name to store current things on map.
`Restore saved objects`	Prompts for file name to retrieve a list of things from.
`Line-of-sight analysis`	The analysis described in Section 2.2. This takes about 30 minutes on a small map, so the results should be saved in a file.
`Save l-o-s analysis`	Prompts for file name to store current line-of-sight analysis in.
`Restore saved l-o-s`	Prompts for file name to restore such an analysis from.
`Display region info`	The user is prompted to click on various map pixels with the mouse. Each click causes a small text window of geographical information about the region containing the pixel to be displayed. For debugging, the most useful piece of information is the "id" of the region, which is an integer less than few thousand.

`Display l-o-s relations`	The user is prompted to click on map pixels. Clicking the left button whitens all the pixels that are visible from this one. Clicking the right button blackens all the pixels that are invisible from it. These two sets should cover the whole map. Clicking in the middle refreshes the screen.
`Display view links`	The user is prompted to click on map pixels. Clicking the left button on a region complements every pixel that some pixel in this region looks down on. Clicking the left button complements every pixel that looks down on some pixel on the region. This information is stored in "view links" as described in Section 2.2.
`Tactical assessment`	A tactical assessment is performed using all the of the forces and objectives shown on the map. First, a Red flow is performed, using a static attrition model treating only existing Blue force concentrations. Then a Blue counterflow is done, as described in Section 3.4. Then a Red countercounterflow is done. The first few times this is done, it takes a long time, because various expensive computations are done regarding attack-site quality, and then cached for future reference. The results of these flows are saved as the current *tactical scenario*.
`Display tactical info`	The user is prompted to click on map pixels. Clicking left shows the role of the region as a Blue attack site. If this region is a Blue attack site, arrows are drawn to regions that this region can attack and that Red might be trying to move through. In addition, the first region where these Red forces are committed to going through this zone is marked with a big `X`. Clicking right shows the significance of this region from the viewpoint of Red defenses. Arrows are drawn to this region from all the places Blue might attack it. Clicking middle pops up a text window summarizing in natural language the schedule of Red and Blue troop movements through the region, including attrition, plus a list of the attack relations between this region and others (identified by the integer id).
`Refresh`	Redraw the screen, including the current scenario, shown as black arrows for Red movements and white arrows for Blue movements.

To run the program, the hardest thing to provide is the map data. YATS loads maps from character files with this format:

1. Two integers, the number of columns C and the number of rows R in the map.
2: R lines, each containing C floating-point numbers. These spell out the altitudes of the corresponding map pixels.
3: Any number of blank lines, followed by a line beginning with a hyphen ("-").
4: R lines, each containing blanks for traversable pixels, nonblanks for untraversable ones (e.g., `x`'s). If the character "|" is encountered, then it and everything else on the line is ignored, and the remaining pixels are assumed to be untraversable.
5: An optional line beginning with hyphen. If this line is encountered before R lines of text have been processed, then all remaining rows are assumed to be untraversable.

Each pixel is assumed to represent a square of width and height equal to the value of the global variable `PIXEL-DIAM*`, whose default value is 100 (meters).

To run the program, load the file BUILDYATS.XLD from the top-level YATS directory. Then execute (LOAD-YATS) and (RUN-YATS).

The map is displayed in the map window. What is displayed is actually the upper-left-hand corner of the map. If it will not fit on the screen, changed the global variable SQ-PIX-NUMBER* before loading. This is the number of screen pixels per map pixel. The default value is 6. Setting it to 4 will cram three times as much surface area onto the screen. (Later versions of the program should allow zooming, shifting, and map editing under menu-and-mouse control.)

There are a few other global variables that can be used to control the way the program works. (See also the attrition parameters of Section 3.3. WIDTH-CAPACITY-FACTOR* is the number of battalions that can pass through a portal of width 1 m in 1 hr. The default value is 0.01 (i.e., 1 battalion per hour can travel through a 100m gap.) The variable DEFAULT-SPEED* says how fast a unit can travel, in meters per hour. The default is 2000 (i.e., 2 km per hour). Currently, all regions have this same default speed. To change a region, set its !_SPEED slot. E.g., to set R1's speed to 1 km/hr, execute

```
(!= (!_(region SPEED) R1) 1000.0)
```

When a tactical assessment is done, various comments will be printed saying what the program is doing next. To turn this off, set ASSESS-DBG* to #F (its default value is #T). (Recall that we use #T and #F for true and false.)

If the variable FLOW-DBG* is set to #T (default #F), then a verbose commentary will accompany force-flow generation. You probably don't want to see it.

When you do a line-of-sight analysis, you will see pixels dancing around the screen. If you set VISREL-DBG* to #F (default #T), they will go away. With the variable set to #T, the calculation runs twice as slowly (one hour instead of 30 minutes), but it *seems* faster because there is something to watch.

5 Conclusions

Work is just beginning on this paradigm for tactical situation assessment. It is too early to draw any definite conclusions. So I will draw some tentative ones.

Tactical situation assessment is a variety of *plan recognition* or *motivation analysis*, the inference to an agent's plans from his behavior. The theory of plan recognition is in its infancy, both theoretically and practically. (For some theoretical work, see Birnbaum 1986, Kautz 1987, Kautz and Allen 1986). Most of the practical work has been in the area of natural-language understanding, where plan recognition is required to infer the reasons for story characters' actions. (Charniak 1988, Wilensky 1983) Programs that do plan recognition are plagued by combinatorial explosion.

One reason for the explosion is a natural tendency to see plan recognition as a process of inverting planning. Any given action could be an atomic part of a great many plans, so there is a big search space in the bottom-up direction. Once a plan is proposed, lots of actions could be steps in it (or substeps or subsubsteps, ...), so the top-down search space isn't so great either. One is naturally led to marker-passing schemes (Charniak 1983), which have many problems.

The present work sidesteps all of this. It deals with direct observations of behavior (actually, in the current version, just of current state), so that natural language is not involved. There are more of these observations, and they tend to be at the same level of detail, so we don't have to worry about searching up arbitrarily far from observation to hypothesis. We don't have to worry about indexing an encyclopedia of possible plans, but only the plans that an agent might have on a battlefield. And, most important, we are free to seek a representation that subsumes all likely plans compactly. The force-flow graph is an attempt at such a representation. Rather than generate a large disjunction of all possible Red plans, we can generate a vaguer representation of where he might go and what he would have to attack if he went there. If later observations confirm that he is in that vicinity, we have a good idea of how he got there and in what strength, and where he is going next.

The current system does not attempt to match incoming intelligence reports against these hypotheses.[9] This is among the most important areas for future work. What I envision is something like this: As intelligence reports of Red activity come in, they are matched up with predicted flows. If the match is good, then the system should assume that significant Red forces are located in the areas where they have been observed, and that no forces are in alternative areas. Given these revised force concentrations, the scenario generator should be run again, yielding up-to-date predictions of future Red motion which can be matched against newer reports.

There are lots of other improvements that need to be made to YATS. As I said at the beginning, I have neglected symbolic reasoning about cover, concealment, and such. Forces are treated as vanilla blobs of stuff, with no differentiating properties. All of this would be easy to change. For instance, different force sources could have different travel speeds through different regions. The attack-site finder (section 3.4) could check vegetation cover and decide that infantry in a region could not be attacked effectively from another area.

One particularly interesting modification would be to take roads into account. Currently all movement is assumed to go in straight lines from portal to portal. Clearly, unopposed movement would often do better by staying on roads as much as possible. The region-flow model could check to see if a force moving through a region was under fire, and, if not, do a local search for the most effective use of roads in the region to get from portal to portal. The resulting flow graph would be the same as now, except for reduced travel times.

Some attention should be given to finding maximum flows as well as "expected" flows. The attrition algorithm is just guessing when it picks values for the opposing armies in its simulated battles. What it needs to know is the most force that each side could bring to bear at a point. Military force is usually applied in an "all or none" way. When the flow algorithm says that 1/3 of the force will be at a place and time, that

[9] Some experimentation was done with a numerical-optimization approach to confirming these hypotheses, but it was abandoned.

really means that 1/3 of the time as much force as possible will at that place at that time. It is probably not too hard to run a standard max-flow algorithm (Even 1979) on the DAG output by the expected-flow algorithm.

The current algorithm does not do an adequate job of annotating its outputs. For example, if Red cannot get to any objective in force, the result is an empty flow, which doesn't tell the user much. He can click on different regions, but all he gets is a blurb saying "No Red forces come through here." It would not be hard to print out instead something to the effect that "Red forces could not get here within the time frame allotted, because they would be under attack from Blue forces." Much of this is just a matter of collecting a little more information and digging it out when required.

Acknowledgements: This work profited from discussions with Andrew Gelsey, who convinced me that detailed planning was not the way to go, and with Jeff Maier, who helped develop an earlier version of the force-flow algorithm.

Appendix: Connection-Machine Simulator

Early in this research, I became interested in the idea of running a lot of algorithms using data-parallel hardware like the Connection Machine. In retrospect, this wasn't such a great idea. The slowest part of the system is the line-of-sight calculator, which uses a hierarchical algorithm that is not amenable to data parallelism. (There may be a more suitable algorithm; this is an important avenue to pursue.) I never actually got around to running anything on a real Connection Machine (although Yale owns one), but I did get as far as writing a simulator and writing many of the region-analysis algorithms in *Lisp, the data-parallel Lisp dialect developed for the Connection Machine. Because of simulator overhead, these programs run much slower on a sequential machine than they have to. It would be a relatively straightforward project to rewrite them in ordinary Lisp, and this should be done.

The structure of my simulator may be of interest to language hackers. The basic data structure is the "pvar," which behaves like an array of data that may all be accessed in parallel. For instance, it takes constant time to add two pvars together, because all the element additions occur simultaneously. One way to implement pvars on a sequential machine is as arrays. Adding two pvars would then be a matter of allocating a new array and filling its slots with sums. This is too expensive, because only a small piece of the data structure may ever be used. What we need is a sort of "lazy evaluation" scheme (Abelson and Sussman 1985), in which pieces are not computed unless they are needed.

The basic operation on pvars is PREF-GRID.[10] (PREF-GRID v i j) returns the j'th element of the i'th column of pvar v. Using object-oriented programming techniques, we define various sorts of abstract pvar objects, and cause them to respond to the PREF-GRID message in appropriate ways. For example, one kind of pvar is the *lazy pvar*. It is defined purely in terms of a function that is called to compute the i, j'th element when required. The function is called anew for every invocation of PREF-GRID.

A good example of a lazy pvar is one produced by +!!. Like most *Lisp primitives, the name of this one ends in two explanation marks to indicate that it creates a pvar. In the simulator, (+!! v_1 v_2) produces a lazy pvar that responds to PREF-GRID by finding the i, j elements of v_1 and v_2, adding them, and returning them. The sum is done every time the element is accessed, and no array need be allocated.

Lazy pvars are inappropriate as variable values, for two reasons. First, variables can be destructively altered, a meaningless operation on a lazy pvar. Second, computational results have to be cached somewhere, or the system will recompute the same things over and over. So we need *memo pvars*, which internally consist of an entry generator plus a data structure in which to store entries as they are generated. The entry generator is just another pvar of some sort, which contains the initial values of this one.

The data structure is a vector of vectors, one per column. Initially, the vectors are not allocated, but appear as dummy #F values. When a column is touched for the first time, its vector is allocated, but initially each entry is a special *EMPTY flag. Only when the entry is actually needed for the first time is it computed and stored.

The main subtlety (and biggest headache) with memo pvars is side effects on them. The operation *SET stores some part of one pvar into another. (For details, see the *Lisp manual, Thinking Machines 1986.) To avoid having to compute that part, the system stores with each memo pvar a history of all *SET operations. When the contents of a position are required, this history must be examined to determine what was the last item targeted there, and compute and store it.

Finally, a special case of memo pvars is Boolean pvars, whose entries are either #T or #F. These play a big role in the system, because of their use as masks for gating other operations (e.g., *SET). For efficiency, the simulator works hard to keep from explicitly representing #F elements. It keeps track of columns that are entirely false, and, within each column, the bounds within which all the non-#F entries lie. Rather than allocate a vector of constant size for a column, all that is allocated is a vector large enough to hold the non-#F entries. In this way, the common case of a small rectangle of #Ts can be represented with reasonable efficiency.

[10] Actually, in the real Connection Machine, there is a more fundamental operation PREF, but for the terrain-analysis application we need only a two-dimensional view of pvars.

In spite of all the ingenuity that went into this system, and its possible theoretical interest as a lazy-evaluating array package, getting rid of it would be an improvement for the situation-assessment system.

Bibliography

1 Harold Abelson and Gerald Jay Sussman 1985 *Structure and Interpretation of Computer Programs*. Cambridge: MIT Press

2 Lawrence Birnbaum 1986 *Integrated Processing in Planning and Understanding*. Yale Computer Science Department Report 489.

3 Eugene Charniak 1983 A parser with something for everyone. In Margaret King (ed.) *Parsing Natural Language*, Academic Press

4 Eugene Charniak 1988 Motivation analysis, abductive unification, and nonmonotonic equality. *Artificial Intelligence* **34**, no. 3, pp. 275-296

5 Shimon Even 1979 *Graph Algorithms*. Potomac, MD: Computer science Press

6 W. Daniel Hillis 1985 *The Connection Machine.*. Cambridge: The MIT Press.

7 Henry A. Kautz 1987 *A Formal Theory of Plan Recognition*. U. of Rochester Computer Science Department Report.

8 Henry A. Kautz and Jam Allen 1986 Generalized plan recognition. *Proc. AAAI* **6**, pp. 32–38

9 Drew McDermott 1988 Revised NISP Manual. Yale Computer Science Department Report 642.

10 Hanan Samet 1985 Data structures for quadtree approximation and compression. *Comm. ACM*, pp. 973–993

11 Hanan Samet and Robert E. Webber 1988 Hierarchical data structures and algorithms for computer graphics. *IEEE. Proc. on Comp. Graphics and Applications*, May, pp. 48–68

12 Thinking Machines Inc. 1986 *The Essential *LISP Manual*, release 1, revision 7. Cambridge: Thinking Machines Inc.

13 U.S. Army 1971 *Combat Intelligence*. Department of the Army Field Manual FM 30-5

14 Robert Wilensky 1983 *Planning and Understanding*. Reading, Mass.: Addison-Wesley