Finding and Using Chokepoints in Stratagus

Benjamin Brewster
Oregon State University
Fall 2007

Abstract

This paper describes a method for finding areas of interest on a two-dimensional grid map used in the real-time strategy engine Stratagus. The method involves discovering chokepoints where through all simulation agents must pass. Using a set of tunable parameters, a full set of chokepoints are located. The redundant and useless chokepoints are then filtered out of the set. The resulting chokepoints can then be used to create a graph of the high-level map structure. The method used to cull less-useful chokepoints is presented. Secondarily, two algorithms were developed that help decide at which chokepoints a limited number of defensive structures may be placed for the greatest benefit. The results of a series of tests are given that show that these algorithms are valuable: tower placements based on both the optimal and greedy implementations, built on the maximum flow of the resultant graph, perform markedly better than random placement. Further, the framework (also by the author) used in this project is dissected.
Figure 1 – Stratagus, based on Warcraft II. Blue forces storm a red base.
1. Introduction

In December 1995, Blizzard Entertainment released a game entitled Warcraft II: Tides of Darkness. This real-time strategy game features a top-down perspective utilizing two-dimensional units depicting knights, dragons, and other medieval warriors. In 1998, several fans of the game released a free version of the game engine written in a modular form. This program eventually became Stratagus, a game engine that runs the Wargus game data forming a near-perfect clone of Warcraft II. This engine was chosen as a research tool by the Intelligent Systems group at Oregon State University (OSU) to further a DARPA-funded project performing research in Transfer Learning.

2. Stratagus

A game of Stratagus involves one or more units attempting to defeat opposing forces. During the game, a team must collect resources from the map in the form of Gold, Wood, and Oil, and convert these through special purpose-built structures into units of war: footmen, knights, siege engines, and defensive buildings like guard towers.

Each unit has a different set of abilities, behaviors, and possible actions. A footman, for example, can only move and fight, while a peasant can move, fight, build new buildings, and repair damaged buildings.

Control of the units, in the Stratagus program itself, is done by selecting units with the mouse, and clicking on the field to assign actions. The units can also act autonomously – primarily in a defensive capability – when left to themselves.

The game happens in real-time – that is, the units move and fight continuously, without waiting for turns. Thus, each player orders his units and manages his buildings at the same time. The winning player is one who can build an economy rapidly while creating armies and directing decisive attacks on the enemies resource production lines and structures.

3. Interface

Interacting with the Stratagus engine can be done in many ways. The program can be manually started and run using the mouse and keyboard. In this case, the user controls his team of units manually and must play the game at a (slow) speed suitable for humans.

Though the game occurs in real-time, each second is broken into 30 discrete steps (by default). Every temporal action thus occurs for a specified number of ticks. These ticks proceed too quickly for a human player to notice their existence, but a computer interacting with the engine can direct and act in between each tick.

Interaction with the engine can also be done using network sockets. This allows a separate program to control the game at a much finer level of control. After each game time tick, Stratagus will pause until it is told to advance the next tick. Therefore, a separate control program can take as much time in between each tick to plan and issue orders as is desired.

Socket Interface Language

The original sockets interface into Stratagus was written by the Stratagus Transfer Learning team at Berkeley University. At OSU, our socket library was written by Neville Mehta. Mehta’s code allows us to interact directly with the Stratagus engine. The commands sent back and forth between a client application and the Stratagus engine (as a server) are concise and

Listing 1 – Opaque
Stratagus Socket Interface Language

```
Z 3214
LISPGET M
LISPGET S
XYCLE
  C 1 2 17 7
  C 2 2 17 7
  C 3 2 17 7
  C 4 2 17 7
  C 5 2 17 7
  C 11 3 74 28 2
  C 17 8 0 0 7 16
  TRANSITION 1
```
powerful, but difficult to read. A client using these commands would quickly become difficult to maintain (see Listing 1).

**Wrapper Class**

One of the author’s first tasks in working with Stratagus was to take the command line-style interface that operated over the socket network protocol and wrap a more user-friendly class around it. The outcome of this work is a class called *CGameProxy* that allows a programmer to issue Stratagus commands as pre-built, nicely-named C++ functions and to manage the state of the map and all units. Listing 2 shows a sample listing of code that takes advantage of this class to create a connection from the client program to Stratagus and then command a selection of units to attack an enemy structure. While not shown in Listing 2, *CGameProxy* also provides commands that allow the harvesting of resources.

```cpp
CGameProxy myCGameProxy(false, "localhost", 4870,
   "stratagus.exe -l maps\SampleMap.pud.gz", true);  // Declare and Init
myCGameProxy.SetTrace(true);  // Show all commands for debug purposes
vector<unsigned short>* mui = myCGameProxy.GetMyUnitIndexes();  // Our units
vector<unsigned short>* eui = myCGameProxy.GetEnemyUnitIndexes();  // Enemy units

while (true) { // Main game loop
   myCGameProxy.GetUnitStatesFromStratagus();  // Get the state of all units
   myCGameProxy.GetMapStateFromStratagus();  // Get the state of the map

   for (int currUnit = 0; currUnit < mui->size(); currUnit++)  // Attack
      myCGameProxy.MyUnitCmd_Attack(mui->at(currUnit), eui->front(), true);

   ...
```

*Listing 2 – a CGameProxy example in C++*

4. Strategic Defense

The author’s Master’s Project consisted of analyzing a map for special areas called chokepoints (see Figure 2). These areas are where two impassable walls come close together – units moving through the area must pass through the chokepoint. Below I present a method to find all chokepoints on a Stratagus map, keep only the good ones, construct a graph that shows how the chokepoints divide the map, and analyze the resulting graph for the best locations for defensive structures.
The process of finding a chokepoint between two impassable areas can be thought of in simple terms: what is the shortest distance between two obstructions? A line connecting the closest points on each object represents a chokepoint. For Figure 2, which shows a simple intuitive chokepoint, it is easy to programmatically discover the locations of these points: a search of the grid squares associated with each object yields the closest two.

This is the method used in [Reich 95]. Alfred Reich calls the resulting divided map a cognitive map because it is easy to reason about: having a map separated into compartments allows us to compare a value of each section to our available resources and decide which section to pursue control over.

Although simple, this method breaks down when the objects have significant concavities or are of sufficiently large size – if there are seemingly multiple areas that connect two objects a sufficient distance apart then they may very well be considered different chokepoints. We see this intuitively in Figure 3 in the lower left hand corner: the small island of rock forms two obvious chokepoints with the surrounding concave wall. The width 6 chokepoint on the right side of Figure 3 also indicates what our eyes immediately see: there is a chokepoint that defines the entrance to this enclosure.

**Improved Chokepoint Finding**

The following method is a pseudo-code algorithm for finding all chokepoints on any type of two-dimensional grid map, with a surrounding border or not.

```
Function: FindChokepointOfSize(S)
  Let S be the size of the chokepoints we’re looking for
  Let M be the set of all squares on the map
  Let A be the set of chokepoints found
  For each square Q
    For each square in M
      If (distance between Q and M == S) AND (Q and M are impassable)
        If there is no obstruction between Q and M
          Add to A the chokepoint between Q and M
        End If
      End If
    End Loop
  End Loop
End Function
```

**Listing 3** – A simple algorithm for finding all chokepoints with no culling
A worthwhile optimization that can be done to Listing 3 is to prevent the two “for” loops from rechecking endpoints that have already been checked. This can be solved by starting the search in the upper left hand corner and searching all squares below you to the left and right only out to a distance of S. Move one square to the right and recheck, continuing in this way to the end of the row. Then, move down one square and return to the left most square. After repeating this to the end of the map, the entire board will have been checked with no repetition.

**Chokepoint Culling**

The above algorithm finds all chokepoints on a map. Not all of these chokepoints are valuable, however (see Figure 4). The culling process, however, has many difficulties:

- If only one chokepoint should remain out of many in a densely populated area, are all of these chokepoints in a group?
- How large should a group grow?
- Are all chokepoints that touch the map border in the same group?

One solution utilized in [Reich 95] is to remove all chokepoints that cross over each other. This works in his domain since there are a limited number of chokepoints; however, as the number of overlapping chokepoints grows - a necessary by-product of attempting to find all chokepoints - a point is reached where all chokepoints in a localized area overlap. Further, as a group grows in size (in terms of number of chokepoints and actual space on the map) the group begins to cover an area where two or more chokepoints may make conceptual sense. These eventualities require special cases involving limits of growth. Listing 4 describes the author’s culling method, which involves first placing the chokepoints into groups.
Function: CullChokepoints(C)

Let C be a passed-in vector of all chokepoints
Let G be a vector containing the chokepoints in the current working group
Let longest, shortest be Integers
Let LongestEndpointDistance be an Integer parameter
Let WallWalkingDistanceThreshold be an Integer parameter
Let GroupSeparationThreshold be an Integer parameter

Empty G

// Search through all chokepoints
For each CP in C
    If CP is not in a group yet
        place CP in G
    End If
End For

For each CP2 in C // Search for other chokepoints that should also be in G
    If (CP2 is already in G) OR (CP2 is already in another group)
        Get next CP2
    End If

    // If chokepoints are close enough to group
    If TotalWallWalkingDistance(CP2, G) <= WallWalkingDistanceThreshold
        If CP2 and any chokepoint in G intersect
            If all endpoints of CP2 and G are on the same wall within LongestEndpointDistance
                Add CP2 to G
            Else If CP2 and each g ∈ G split their endpoints greater than LongestEndpointDistance
                Add CP2 to G
            End If
        End If
    End If
End For

// Now that the group is completely found, decide whether to delete it or not
// Find the longest and shortest distances between all members in G
For each GCP1 in G
    For each GCP2 in G
        i = WallWalkingDistance(GCP1, GCP2)
        If i > longest
            longest = i
        End If
        If i < shortest
            shortest = i
        End If
    End For
End For

// Function continued on next page

Listing 4 – Function pseudo-code for culling extraneous chokepoints
// Remove chokepoints based on parameters and distances
If longest > GroupSeparationThreshold
  If all endpoints G are on the same wall within LongestEndpointDistance
    Delete all chokepoints in G
    If longest > (manhattan distance between longest endpoints) - 1
      Add chokepoint between longest endpoints to G
    End If
  Else
    Remove all chokepoints except the shortest one that is most averagely placed
  End If
Else
  If all endpoints G are on the same wall within LongestEndpointDistance
    Delete all chokepoints in G
  Else
    Remove all chokepoints except the shortest one that is most averagely placed
  End If
End If
End For

// In addition, remove all intersecting chokepoints
For all CP in C
  For all CP2 in C
    If CP and CP2 intersect
      Remove CP and CP2
    End If
  End For
End For
End Function

**Function: TotalWallWalkingDistance(CP, G)** returns an Integer
Let CP be a passed-in chokepoint
Let G be a passed-in vector of chokepoints
A = 0
For all CP2 in G
  A += WallWalkingDistance(endpoint 1 of CP, endpoint 1 of CP2)
  A += WallWalkingDistance(endpoint 1 of CP, endpoint 2 of CP2)
  A += WallWalkingDistance(endpoint 2 of CP, endpoint 1 of CP2)
  A += WallWalkingDistance(endpoint 2 of CP, endpoint 2 of CP2)
End For
Return A
End Function

**Function: WallWalkingDistance(EA, EB)** returns an Integer
Let EA be a passed-in endpoint of a chokepoint
Let EB be a passed-in endpoint of a chokepoint

Return the shortest-length path walking on impassable terrain between EA and EB
End Function

**Listing 4 Continued** – Function pseudo-code for culling extraneous chokepoints
See Figure 5 for a completely culled map where only “valid” chokepoints remain. Note that the parameters used to build the graph were not hard limits: a width 6 chokepoint was created in the upper right corner to completely encompass a group that met a special case, as described in Listing 4.

Further, note that the groups in the lower left and right hand corners were completely eliminated, since the concavities did not involve inner angles smaller than 90 degrees. These are in contrast to the group in the bottom center.

The bottom center group is also interesting to note since the chokepoint kept was not the shortest (the standard method), but the longest.

**Graph Construction**

With Figure 5, we understand intuitively that the map can be thought of as compartments with a cost associated with traveling between them. In undefended cases, the cost is that the path between areas is simply dictated by the location of the chokepoint, and that only a limited number of units may move through at a given moment based on the width of the chokepoint. Where the chokepoint is defended by enemy forces, the cost to move through additionally involves defeating the enemy forces at a cost to your own forces, or in moving through and accepting casualties.

We can see that the structure of a map can be thought of as a series of nodes and edges. In Figure 6, a graph built on the culled map in Figure 5 is presented as a visual example. The image manipulation program ImageMagick was used to draw the nodes and edges on a screenshot of the map. Nodes in the graph are defined as the centroid of a geographical region bounded by impassable terrain and chokepoints. The edges between nodes leave the node and are drawn first to the center of the chokepoint and then to the adjoining area reached through the chokepoint – this is merely a graphical clue as to where the areas connect through. The graph can easily be seen as a weighted graph, where the weights of the edges can represent the cost of moving between areas. These graphs can be rendered to external text files where their structure can be incorporated into other applications. See the Further Research area for several suggestions.
Where to build?

As a sample application of using the graphs generated above, we can attempt to figure out where the best places to put a limited number of defensive structures are. If we restrict our placements to chokepoints, then some chokepoints are more valuable than others at restricting the movement of enemy forces.

We assume that there is an origin area for the enemy, and a target area that they are moving towards. We can place a limited number of guard towers – a tower that shoots arrows at any enemy that comes within its range. These close-packed towers not only restrict the free flow of movement through a chokepoint even more by taking up space, they are also able to focus their fire on enemy units such that the most number of individual units can be destroyed.

Flow Networks

How do we know that a given placement of towers is effective? One way to quantize the effect of placing towers in chokepoints on a map is to use a flow network. A flow network is a directed graph consisting of nodes and edges: each edge connects two nodes and has a specified maximum capacity. One node is assigned as the source, and another becomes the sink. A flow is calculated for the graph that represents an amount of flow, based on the capacities of all of the edges, from the source node to the sink.

In our graphs we use the widths of the chokepoints and the effects of a specified number of numbers to affect the capacity of an edge.

The simple example in Figure 8 shows that even though edges A and B together can maintain a flow of 4, edge C can only maintain a flow of 3. Thus, the maximum flow of the entire graph from source S to sink T is 3.

In my implementation, I used the Boost.org libraries which provide a function for solving the maximum flow of a network using the Push-Relabel method implementation described in [Goldberg 94].

A graph edge in our case is a chokepoint, while the nodes represent the areas that chokepoints separate. The equation I used to define the capacity of an edge is found in Listing 5.

\[
\text{Capacity} = (\text{Width} \times 2) - (\text{NumTowers} \times 3)
\]

Listing 5 – Edge capacity equation

Where \text{Width} is the width of the chokepoint rounded to the nearest integer, and \text{NumTowers} is the number of towers assigned to that edge.
Experimental Results

To test the utility of flow network-based placement, the results of three different placement methods of four friendly towers into the flow network were compared. Eight enemy knights were positioned at one node of the map/graph (the source), and would proceed to run to a friendly base (the sink). On the way, they would pass through chokepoints where towers could be placed.

The first method placed the towers randomly on the map. They could be placed in chokepoints, or out in an area. The second method used a greedy method of placement. A tower is tried at each edge (chokepoint). The flow network of the graph between the source and sink nodes is then measured. The placement with the lowest flow is kept, and the next tower is tried at all spots and the lowest one kept, and so on. Thus, the placement speed is $O(nm)$ where there are $n$ towers and $m$ chokepoints. The third method uses recursion to test all possible tower placements by iterating over each one. The one with the lowest flow is kept, and is the optimal assignment of that number of towers for that map. An exhaustive search of all possibilities takes $O(n^m)$ time. Figures 9 and 10 show two maps related to the results shown in Table 1.

<table>
<thead>
<tr>
<th></th>
<th>Random</th>
<th>Greedy</th>
<th>Optimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dragon Run</td>
<td>6.35</td>
<td>4.1</td>
<td>4.1</td>
</tr>
<tr>
<td>One Way In, One Way Out</td>
<td>6.9</td>
<td>4.4</td>
<td>4.4</td>
</tr>
</tbody>
</table>

**Table 1 – Results: Surviving knights with four placed towers**

The greedy and optimal methods when used on One Way In, One Way Out results in two towers being placed between nodes 4 and 11 (which chokepoint was a width of 3). Using the equation in Listing 5, the capacity of this edge becomes zero. Therefore, the next two towers are placed in random chokepoints.

Figure 9, which is the same placement as for the Greedy and Optimal runs, shows that the algorithm correctly blocks off both edges from the source node 1 to the sink node 4 using the numbering given in Figure 6. Each edge (each of width 3) receives two towers, which, when using the equation in Listing 5, has a capacity of zero.

For both of these graphs, since the capacity of the edges referred to above is zero, the total flow of each graph is zero. Modifying the equation in Listing 5 can yields different total flows.

Finally, random placements that placed towers within striking distance of the initial knight position were rejected, with another random instance being generated in its place.
5. Object Oriented Programming

One goal of the project was to incorporate many aspects of Object-Oriented Programming. This section describes what was done to further this goal.

Objects

There are many interacting objects in the various software packages I have used. First and foremost, the CGameProxy object itself understands messages related to manipulating the Stratagus environment, including units and meta-parameters. CGameProxy also tracks the state of the map, all units, and several data structures for quick indexing into other objects.

The following list describes in short form the primary objects involved. There are also a handful of other auxiliary objects that I don’t describe here which primarily hold data.

CGameProxy – Primary interaction object for communication with Stratagus engine. Holds other objects related to the simulation, including the map and all units.

CUnit – A class that represents a Unit. This class understands messages such as GetStatus, GetHitPoints, and GetLocation. All private data (status, health, location, current target, type, etc.) are accessed only via accessor methods.

CStringTokenizer – Since much of the parsing and sending of commands to Stratagus is done via text strings, a robust string tokenizer is required. The one I wrote handles the standard HasNextToken and GetNextToken methods, but also handles methods for resetting the tokenizer to the first token in a string, assigning new separators, and assigning a whole new string.

MapGraphObject – Used in creating the image of the map graph. This object holds MapGraphElements which can be either nodes or edges. It provides for element management and details such as GetNumEdges.

MapGraphElement – A parent class that showcases how nodes and elements can be stored in the same object yet be subclasses of a parent. Supports only one function, the virtual function IAmA, which is overridden to return the type of object that ultimately inherits from it.

MapGraphNode – A node in the map graph which inherits from MapGraphElement. Mostly used for bookkeeping, as most of the functionality is provided by the edge object.

MapGraphEdge – An edge in the map graph which inherits from MapGraphElement. An edge can be loaded with a number of towers and the chokepoint width and can then report what the flow

Figure 10 – Graph of One Way In, One Way Out
through the edge would be. This encapsulation of the flow calculations means that a small change to the flow equation in this class has drastic effects in terms of tower placements.

**Inheritance Hierarchies**

The inheritance hierarchy I’ve chosen to use has `MapGraphElement` as the parent class. `MapGraphNode` and `MapGraphEdge` both subclass from `MapGraphElement` for specialization. The intent here is that all of my map graph objects can be stored in the same vector in any order. The class `MapGraphObject` has a vector called `elements` which is declared as a vector of `MapGraphElements`. This vector then allows storage of any objects that are subclasses of `MapGraphElement`.

**Overriding**

To tell the difference between a `MapGraphNode` and a `MapGraphEdge`, I decided to add a method to the parent class `MapGraphElement` called `IAmA`. This function is overridden by both child classes, and returns a different value in each. Specifically, `MapGraphEdge->IAmA` returns 1, while `MapGraphNode->IAmA` returns 0.

When we iterate through the elements vector, we can query each item in turn to figure out which subclass it is.

**Overloading**

There was less use for overloading in my project, but I did use it for some debug printing operations. Listing 6 shows each function in its entirety.

```c++
void CoutRect(RECT *r)
{
    cout << "RECT top: " << r->top << ", bottom: " << r->bottom
        << ", left: " << r->left << ", right: " << r->right << endl;
}

void CoutRect(RECT *r1, RECT *r2)
{
    cout << "RECT1 top: " << r1->top << ", bottom: " << r1->bottom
        << ", left: " << r1->left << ", right: " << r1->right << endl;
    cout << "RECT2 top: " << r2->top << ", bottom: " << r2->bottom
        << ", left: " << r2->left << ", right: " << r2->right << endl;
}
```

**Listing 6 – Overloaded functions for printing contents of RECT structures**
6. Further Research

First, by examining Figure 10, in concert with the placement of the towers between areas 4 and 11, we can intuitively separate the graph into only two areas – the northern side, and the southern side. This kind of node and edge combination would provide a more high-level grasp of the map structure that could be used by high-level game strategy planners. In combination with a value-based assignment (due to gold mines, tree stands, etc.) to the areas themselves, a clear picture of which areas to pursue control over based on resource requirements can be built.

Secondly, if we know which areas are important, and have an understanding of how effective our defensive structures are, we can create a game playing agent that understands how to pursue geographical control and defend it accordingly. This would be a marked improvement over the existing agents included in Stratagus and in Warcraft II itself, and indeed even in most modern real-time strategy games.

Thirdly, the work above can be extended to real-time strategy domains that use non two-dimensional grid maps. Warcraft 3 uses a two dimensional world with a continuous grid – that is, units can be placed at any fraction position on the board. Homeworld, another popular game, uses a three-dimensional space environment where the positions of ships are specified in three continuous dimensions. Chokepoints in both of these environments can be found and culled by extending the pseudo-code provided above.

7. Conclusion

In this paper I have shown how older methods of chokepoint finding are limited to only simple shapes. I have demonstrated how to find and focus on the chokepoints that are of most worth. These chokepoints naturally separate a map into areas of interest, and I have discussed methods to visualize and analyze these areas.

The CGameProxy framework I developed is currently in use at many universities, and provides an easy to use body of code to further the experimental work of other research. Using this framework with the chokepoint methods provided can be a powerful tool in map analysis, test case generation, and other experimental work in real-time strategy domains.

8. Bibliography


